УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ Т ХНИЧКИХ НАУКА У НОВОМ САДУ

Богдан Вукобратовић

Хардверска акцелерација неикременталних алгоритама за формирање стабала одлуке и њихових ансамбала

ДОКТОРСКА ДИСЕРТАЦИЈА

Нови Сад, 2016

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:	
Идентификациони број, **ИБР**:	
Тип документације, **ТД**:	Монографска публикација
Тип записа, **ТЗ**:	Текстуални штампани материјал
Врста рада, **ВР**:	Докторска дисертација
Аутор, **АУ**:	Богдан Вукобратовић
Ментор, **МН**:	др Растислав Струхарик, ванредни професор
Наслов рада, **НР**:	Хардверска акцелерација неинкременталних алгоритама за формирање стабала одлуке и њихових ансамбала
Језик публикације, **ЈП**:	Енглески
Језик извода, **ЈИ**:	Српски/Енглески
Земља публиковања, **ЗП**:	Србија
Уже географско подручје, **УГП**:	Војводина
Година, **ГО**:	2016
Издавач, **ИЗ**:	Ауторски репринт
Место и адреса, **МА**:	Факултет техничких наука, Трг Доситеја Обрадовића 6, Нови Сад
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)	7/160/94/45/56/0/0
Научна област, **НО**:	Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:	Електроника
Предметна одредница/Кqучне речи, **ПО**:	Стабла одлуке, хардверска акцелерација, реконфигурабилни хардвер, ансамбли класификатора, еволутивни алгоритми
УДК	
Чува се, **ЧУ**:	Библиотека Факултета техничких наука у Новом Саду, Трг Доситеја Обрадовића 6, 21000 Нови Сад
Важна напомена, **ВН**:	
Извод, **ИЗ**:	У овој дисертацији, представљени су нови алгоритми EFTI и EEFTI за формирање стабала одлуке и њихових ансамбала неинкременталном методом, као и разне могућности за њихову имплементацију. Експерименти показују да је предложени EFTI алгоритам у могућности да произведе драстично мања стабла без губитка тачности у односу на постојеће top-down инкременталне алгоритме, а стабла знатно веће тачности у односу на постојеће неинкременталне алгоритме. Такође су предложене хардверске архитектуре за акцелерацију ових алгоритама (EFTIP и EEFTIP) и показано је да је уз помоћ ових архитектура могуће остварити знатна убрзања.
Датум прихватања теме, **ДП**:	
Датум одбране, **ДО**:	

Чланови комисије, **КО**:	Председник:	др Станиша Даутовић, доцент	
	Члан:	др Вук Врањковић, доцент	
	Члан:	др Иван Мезеи, доцент	Потпис ментора
	Члан:	др Теуфик Токић, редовни професор	
	Члан, ментор:	др Растислав Струхарик, ванредни професор	

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monograph
Type of record, **TR**:	Printed text
Contents code, **CC**:	Ph.D. Thesis
Author, **AU**:	Bogdan Vukobratovic
Mentor, **MN**:	Rastisla Struharik, Ph. D., associate professor
Title, **TI**:	Hardware Acceleration of Nonincremental Algorithms for the Induction of Decision Trees and Decision Tree Ensembles
Language of text, **LT**:	English
Language of abstract, **LA**:	English/Serbian
Country of publication, **CP**:	Serbia
Locality of publication, **LP**:	Autonomous Province of Vojvodina
Publication year, **PY**:	2016
Publisher, **PB**:	Author’s reprint
Publication place, **PP**:	Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000 Novi Sad
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	7/160/94/45/56/0/0
Scientific field, **SF**:	Electrical and Computer Engeneering
Scientific discipline, **SD**:	Electronics
Subject/Key words, **S/KW**:	Decision trees, hardware acceleration, ensemble classifiers, reconfgurable hardware, evolutionary algorithms
UC	
Holding data, **HD**:	
Note, **N**:	
Abstract, **AB**:	The thesis proposes novel full decision tree and decision tree ensemble induction algorithms EFTI and EEFTI, and various possibilities for their implementations are explored. The experiments show that the proposed EFTI algorithm is able to infer much smaller DTs on average, without the significant loss in accuracy, when compared to the top-down incremental DT inducers. On the other hand, when compared to other full tree induction algorithms, it was able to produce more accurate DTs, with similar sizes, in shorter times. Also, the hardware architectures for acceleration of these algorithms (EFTIP and EEFTIP) are proposed and it is shown in experiments that they can offer substantial speedups.
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	

Defended Board, **DB**:	President:	Stanisa Dautovic, Ph.D., ass. professor	
	Member:	Vuk Vranjkovic, Ph.D., ass. professor	
	Member:	Ivan Mezei, Ph.D., ass. professor	Menthor's sign
	Member:	Teufik Tokic, Ph.D., full professor	
	Member, Mentor:	Rastislav Struharik, Ph.D., ass. professor	

Obrazac **Q2.HA.06-05**- Izdanje 1

Абстракт

У овој дисертацији, представљени су нови алгоритми за формирање стабала одлуке неинкременталном методом, као и разне могућности за њихову имплементацију. Прво је дат опис новог EFTI (Evolutionary Full Tree Induction) алгоритма, дизајнираног тако да омогући имплементацију са што мање хардверских ресурса, као и да производи што мања стабла одлуке, а без утицаја на њихову тачност. Ово пружа могућност да се EFTI алгоритам користи у ембедед системима, где је оптимална употреба ресурса од велике важности. Имплементација EFTI алгоритма за PC платформу је онда поређена са PC имплементацијама неколико других постојећих алгоритама за формирање стабала одлуке у погледу тачности и величине произведених стабала. Експерименти показују да је предложени EFTI алгоритам у могућности да произведе драстично мања стабла без губитка тачности, у односу на top-down инкременталне алгоритме. Са друге стране, у поређењу са другим неинкременталним алгоритмима за формирање стабала одлуке, EFTI је успевао да произведе знатно тачнија стабла, сличне величине, за краће време. Након тога, истраживана је могућност хардверске акцелерације овог алгоритма на основу резултата његовог профајлинга и разматрања његове временске комплексности. На основу анализе, предложен је EFTIP (Evolutionary Full Tree Induction co-Processor) његова архитектура је представљена. Даље у дисертацији, дата је хардвер-софтвер имплементација EFTI алгоритма на основу EFTIP ко-процесора који је конструисан да обавља најинтензивнију фазу процеса формирања стабла неинкременталном методом, фазу прорачуна тачности стабла одлуке. Најзад, у експерименталној секцији ће се говорити о предности система који користи EFTIP ко-процесор, у погледу брзине формирања стабла одлуке. Затим, дат је опис алгоритма за формирање ансамбала стабала одлуке EEFTI (Ensembles Evolutionary Full Tree Induction). Након тога, дати су резултати експеримента у којем су поређене тачности које пружају ансамбли формирани уз помоћ EEFTI алгоритма и појединачна стабла одлуке формирана уз помоћ EFTI алгоритма. Резултати показују да је EEFTI алгоритам у могућности да произведе ансамбле који су тачнији од појединачних стабала одлуке. Слично као и за EFTI алгоритам, разматрана је хардвер-софтвер архитектура EEFTI алгоритма, предложен је EEFTIP ко-процесор за његову хардверску акцелерацију и дати су резултати експеримената који приказују предност ове архитектуре у погледу брзине формирања ансамбала стабала одлуке.

Uvod

Машинско учење је грана истраживачке области вештачке интелигенције. Она се бави развојем алгоритама који “уче” извлачећи обрасце из улазних података и као свој излаз дају системе конструисане да праве предикције над новим подацима. Једна од главних снага система машинског учења је моћ генерализације, која им омогућава да остваре добре резултате на новим, до сада невиђеним подацима, након што су претходно били изложени скупу података за тренирање.

Разни системи машинског учења су до сада предложени у литератури, укључујући: стабла одлуке (DT од енг. decision trees), неуронске мреже (ANN од енг. “artificial neural networks”) и “support vector” машине (SVM). Ови системи се посебно широко примењују у области вађења података (енг. “data mining”), са DT, ANN и SVM-овима као најпопуларнијима.

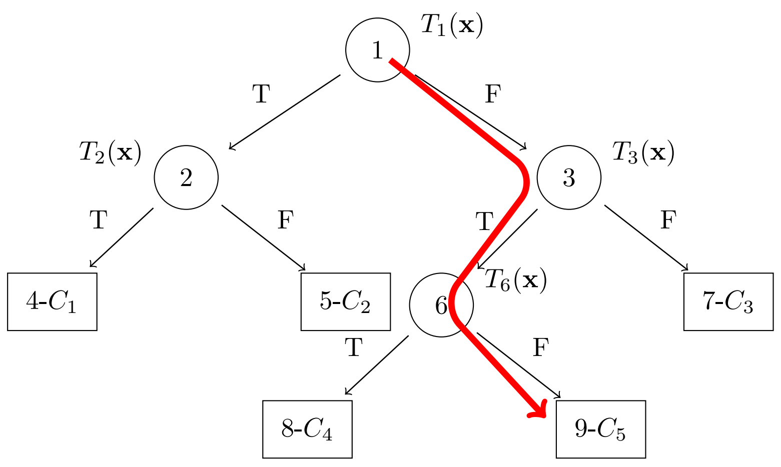
Процес учења, тзв. индукција система машинског учења, може бити како надгледан тако и ненадгледан. Надгледано учење подразумева да је уз сваки улазни податак из тренинг скупа дат и жељени одзив система на тај податак. Са друге стране, у случају када се алгоритму за индукцију пружи само тренинг скуп података без жељеног одзива, реч је о ненадгледаном учењу. У том случају, алгоритам за индукцију мора сам да открије структуру и обрасце у скупу улазних података, што само по себи може бити и циљ решавања неког проблема. Улазни подаци који се користе за учење се обично састоје од скупа инстанци проблема који се решава системом машинског учења и назива се тренинг скуп. Животни век система машинског учења обично има две фазе: тренинг фазу (такође познату као индукцију или обучавање) и фазу коришћења. Конструкција система се врши у тренинг фази уз помоћ тренинг скупа, док се у фази коришћења индуковани систем суочава са новим, до сада невиђеним инстанцама и покушава да да што бољи одзив, користећи знање извучено из тренинг скупа.

Стабла одлуке

Системи машинског учења могу решавати разне проблеме, као што су класификација, регресија, кластерисање, итд. За решавање проблема класификације, за који се често користе стабла одлуке, потребно је распоредити улазне инстанце проблема у неки дискретни скуп класа. Инстанце проблема се најчешће моделују вектором атрибута A, на основу којих се врши класификација. Процес класификације уз помоћ стабла одлуке се може представити дијаграмом који има структуру стабла, као што се види на слици испод. Овај диаграм представља ток близак току људског размишљања, те га је лако разумети, што чини стабла одлуке популарним избором за решавања проблема класификације. Стабла одлуке имају и бројне друге предности у односу на остале системе машинског учења, између осталог: висок степен имуности на шум, могућност класификације инстанци са редудантним или атрибутима који недостају, могућност класификовања инстанци како са категоричким, тако и са нумеричким атрибутима итд.

Теоретски, стабла одлуке могу бити различитог степена, али се најчешће користе бинарна стабла, односно стабла у којима сваки чвор има по два потомка. Слика приказује процес класификације на бинарном стаблу одлуке. Стабло се састоји од 4 чвора означених круговима нумерисаним од 1 до 4. Стабло такође има 5 листова означених квадратима, при чему је сваком листу додељена једна од класа проблема (C_1 до C_5 у овом примеру). Класификација се врши тако што се пусти да се инстанца креће кроз стабло, почевши од корена (нумерисаног бројем 1), све док не стигне до неког од листова. У зависности од листа у коме инстанца заврши свој пут кроз стабло, њој се придружује класа додељена том листу.

Сваком чвору стабла одлуке придружен је по један тест (T_1 до T_4 у овом примеру), који на основу атрибута инстанце одлучује кроз који потомак ће се наставити пут кроз стабло. У случају бинарних стабала, од тестова се очекује бинарни одговор. Коначна путања инстанце кроз стабло ће зависити од резултата тестова у сваком



Слика 1: Процес класификације на бинарном стаблу одлуке.

чвору стабла на који инстанца наиђе у току свог пута. Пуштајући једну по једну инстанцу тренинг скупа, може се добити његова потпуна класификација.

Сваки проблем чија се класификација решава помоћу стабала одлуке, дефинисан је скупом својих инстанци. При дефинисању проблема, потребно је изабрати који атрибути ће чинити вектор атрибута (x) инстанци и једнозначно представљати инстанце проблема. Скуп свих могућих вектора атрибута представља N_A - димензионални простор атрибута, где је N_A број атрибута којима су инстанце описане и уједно и величина вектора x. У контексту простора атрибута, сваки тест бинарног стабла одлуке дели овај простор на два региона, чинећи да је сваком чвору и листу стабла асоциран један под-регион простора. Сваки чвор стабла на основу свог теста дели себи асоцирани под-регион на два и додељује сваки од њих по једном свом потомку. Коначан резултат овог процеса је јасна партиција простора атрибута на дисјунктне регионе асоциране класама проблема.

На основу карактеристика функција којима су имплементирани тестови, стабла одлуке се могу поделити на: ортогонална, неортогонална и нелинеарна. Своје називе, ови типови стабала одлука су добили на основу изгледа површи којом њихови тестови деле простор атрибута. Тако ортогонална стабла одлуке деле простор ортогоналним хиперравнима, неортогонална - неортогоналним хиперравнима, а нелинеарна - нелинеарних хиперповршима.

У овој дисертацији, фокус је на неортогоналним стаблима одлуке јер се жељена тачност на тренинг скупу са њима може постићи са драстично мање чворова у односу на ортогонална стабла. За сличну тачност на тренинг скупу, неортогонална стабла често имају бољу тачност на новим инстанцама проблема. Такође, величина стабала одлуке је значајна у хардверској имплементацији, јер захтева мањи број ресурса. Што се тиче нелинеарних стабала одлуке, она су знатно сложенија од неортогоналних, па и од ортогоналних, а немају већу тачност класификације. Код неортогоналних стабала одлуке, тестови у чворовима генеришу неортогоналне хиперравни којима деле простор атрибута. Неортогонална хиперраван је једнозначно одређена следећом једначином:

$$\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{N_A} w_i \cdot x_i < \theta, \tag{1}$$

где **w** представља вектор коефицијената теста а θ (такође званог праг, или енг. threshold) моделује афини део теста.

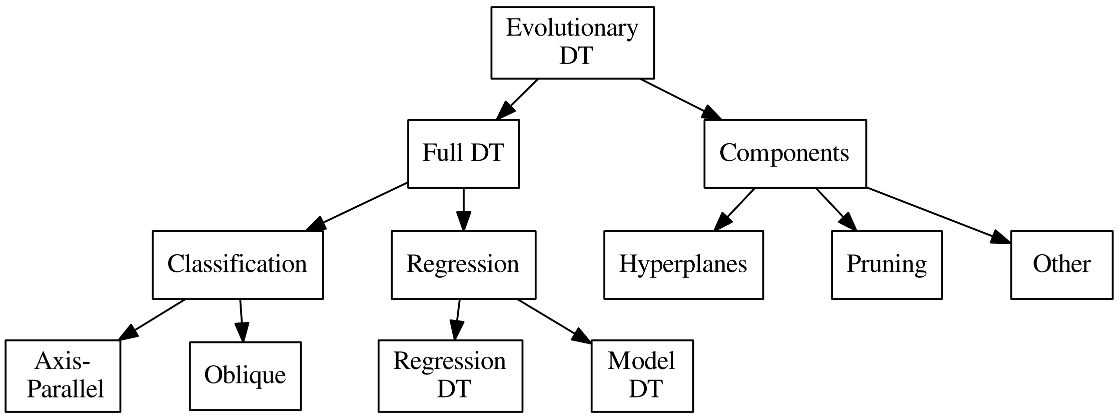
Индукција стабала одлуке

Начелно, стабла одлуке се могу индуковати на два начина: инкрементално (чвор по чвор) или глобално индукујући цело стабло одједном. Већина алгоритама за индукцију неортогоналних стабала одлуке користе неку врсту хеуристике у процесу оптимизације индукованог стабла, која је често неки тип еволутивног алгоритма (EA), јер је проналажење оптималног стабла одлуке NP-тежак алгоритамски проблем.

Инкрементални приступ гради стабло одлуке почевши од корена и додајући му итеративно један по један чвор. Ово је "greedy" приступ, у коме се параметри теста придруженог чвору, тј. вредности вектора коефицијената **w** и вредност прага θ, оптимизују на основу информација о перформансама индукованог стабла, доступних у моменту креирања тренутног чвора, тј. на основу "локалних" информација. Након што је чвор додат у стабло и алгоритам наставља да креира друге чворове, ситуација се променила и доступне су нове информације, али оне неће бити искоришћене за додатну оптимизацију чворова који су већ додати у стабло, те се каже да је оптимизациони процес остао заробљен у локалном оптимуму. Алгоритам обично оптимизује параметре теста у процесу максимизације неке циљне функције која мери квалитет поделе инстанци из тренинг скупа које у процесу класификације успевају да стигну до чвора коме је придружен тест. Овом поделом се добијају два подскупа инстанци, од којих се сваки прослеђује на обраду по једном потомку чвора. За сваки од ова два подскупа се даље проверава да ли се састоје од инстанци које припадају различитим класама или је пак подскуп "чист", у смислу да садржи инстанце само једне класе. У случају да је подскуп чист, као потомак се додаје лист и њему се асоцира класа инстанци из подскупа. У супротном, процес индукције стабла се наставља итеративно и као потомак се додаје нови чвор у циљу даље деобе подскупа инстанци на чисте подскупове. Предност инкременталног приступа је брзина, али индукована стабла су субоптимална по величини и каснијим класификационим резултатима на новим инстанцама.

Други приступ за креирање стабала одлуке је индукција целог стабла одједном, односно, неинкрементални приступ. Овде се у свакој итерацији алгоритма манипулише целим стаблом, тако да су увек на располагању комплетне (глобалне) информације о перформансама индукованог стабла одлуке. У процесу индукције, према неком алгоритму, чворови се додају или бришу и параметри њихових тестова се мењају у циљу оптимизације стабла. Пошто се оптимизација врши на основу глобалних информација о перформансама, овај поступак начелно производи компактнија, а често и тачнија стабла одлуке у односу на инкременталне алгоритме. Са друге стране, ови алгоритми имају већу временску комплексност од инкременталних, што резултује у дужим временима потребним за индукцију.

Као што је речено, проналажење оптималног стабла одлуке је NP тежак проблем, али чак и ако се користи инкрементални приступ индукцији, када је реч о неортогоналним стаблима одлуке, налажење оптималног положаја једне неортогоналне хиперравни је NP-тежак алгоритамски проблем. Из овог разлога, већина алгоритама за индукцију неортогоналних стабала одлуке користе неку врсту хеуристике у процесу оптимизације, која је често нека врста еволутивног алгоритма (EA). Слика приказује таксономију еволутивних алгоритама за индукцију стабала одлуке.



Слика 2: Таксономија еволутивних алгоритама за индукцију стабала одлуке.

У овој дисертацији, предлаже се нови алгоритам за индукцију неортогоналних стабала одлуке неинкременталном методом на бази EA - EFTI алгоритма. Овај алгоритам је осмишљен имајући у виду ембедед системе, где не постоји обиље ресурса, као што су меморија и процесорско време. Другим речима, EFTI алгоритам је осмишљен да може да се имплементира са што мање ресурса и на тај начин омогући његова што лакша интеграција у ембедед системе. Због своје мање временске комплексности, инкрементални алгоритми тренутно доминирају у истраживачком пољу индукције стабала одлуке. Из овог разлога, при дизајну EFTI алгоритма вођено је рачуна о томе да се омогући његова што лакша паралелна имплементација и самим тим омогући развој ефикасног хардверског акцелератора који би драстично скратио време потребно за индукцију, те учинио да овај приступ такође добије на атрактивности. Са друге стране, експериментално је показано да EFTI алгоритам производи компактнија стабла одлуке од инкременталних алгоритама, а без утицаја на њихову тачност. Индукција компактнијих стабала је интересантна са два апекта: компактнија стабла изискују мање хардверских ресурса за чување и манипулацију; компактнија стабла су преферирана према принципу Окамове оштрице, јер представљају једноставнији модел система.

Алгоритми за индукцију стабала одлуке базирани на EA често користе популацију јединки, што није згодно за хардверску акцелерацију јер захтева значајне хардверске ресурсе. Из овог разлога је EFTI алгоритам дизајниран да користи само једну јединку за индукцију. Аутору није познат ни један постојећи алгоритам из научне литературе који испуњава овај услов.

Алгоритми за формирање стабала у хардверу неинкременталном методом

Фаза индукције, у случају да се користи неинкрементални алгоритам за формирање стабла одлуке, може трајати сатима или чак данима за практичне проблеме. Ако би се фаза индукције успела убрзати, могуће би било користити веће тренинг скупове, што би било од посебног значаја у апликацијама “вађења података”. Даље, бржи тренинг стабала одлуке би омогућио краће дизајн циклусе и отворио могућност индукције стабала одлуке у реалном времену за примене које захтевају тако брзо прилагођавање, као што су “web mining”, биоинформатика, машински вид, “text mining” итд.

Проблему акцелерације фазе индукције се може приступити на два начина:

- Развојем нових алгоритамских оквира или нових софтверских алата, при чему је овај поступак доминантан у литератури.
- Развојем нових хардверских архитектура, оптимизованих за убрзано извршавање постојећих алгоритама за индукцију.

У овој дисертацији предложена је хардверска архитектура, названа EFTIP, која се може користити за акцелерацију како EFTI алгоритма, тако и других алгоритама за индукцију стабала одлуке неинкременталном методом. На плану хардверске акцелерације стабала одлука, већина научних радова се фокусира на убрзавање већ индукованих стабала, док је хардверска акцелерација индукције стабала одлуке слабо присутна. Колико је познато аутору, постоје само два рада на тему хардверске акцелерације алгоритама за индукцију стабала одлуке, али оба користе “greedy”, “top-down”, инкрементални приступ. Колико је аутору познато, не постоји ни један рад на тему хардверске акцелерације алгоритама за неинкременталну индукцију стабала одлуке.

Алгоритми за формирање ансамбала

Да би се унапредиле перформансе класификатора, предложено је коришћење ансамбала система за класификацију уместо једног класификатора. Ансамбл класификатора комбинује предикције неколико индивидуалних класификатора у циљу добијања бољих перформанси. Тренирање ансамбала захтева индукцију скупа појединачних класификатора, углавном стабала одлуке или ANN-ова, чије предикције се онда комбинују у фази коришћења ансамбла у процесу класификације нових инстанци. Иако једноставна, ова идеја се показала као веома ефективна, производећи системе који су прецизнији од појединачног класификатора.

Приликом индукције ансамбла класификатора, потребно је решити два проблема:

- Како обезбедити разноврсност чланова ансамбла, тј. разноврсност њихових предикција
- Коју процедуру употребити за комбиновање појединачних предикција сваког класификатора, тако да се појача утицај добрих одлука а потисне утицај лоших.

Међу најпопуларнијим методама које обезбеђују разноврсност чланова ансамбла су Брајманов “bagging” алгоритам, Шапиров “boosting” алгоритам, AdaBoost, Волпертов “stacked generalization” алгоритам, пондерисано већинско гласање и “behavior knowledge spaces”.

Главна предност ансамбала класификатора у односу на појединачне класификаторе је већа тачност предикција и већа робустност на шум. Са друге стране, у односу на појединачне класификаторе, потребне су велике количине меморије да би се сместиле дефиниције чланова ансамбла, а велика рачунарска моћ да би се израчунао одговор ансамбла, што све води ка дужим и у погледу ресурса захтевнијим фазама индукције. Ово је стога што се ансамбли обично састоје од 30 и више појединачних класификатора, те ако би желели исте перформансе класификације што се тиче брзине као у случају појединачних класификатора, било би потребно 30+ пута више меморије и рачунарске моћи.

У овој дисертацији, предложен је EEFTI алгоритам - нови еволутивни алгоритам за индукцију ансамбала неортогоналних стабала одлуке неинкременталном методом који захтева само једну јединку по члану ансамбла, на бази EFTI алгоритма. Исти аргументи у вези погодности за хардверску акцелерацију наведени у вези EFTI алгоритма, важе и за алгоритам за индукцију ансамбала EEFTI. Додатна мотивација за развој EEFTI алгоритма је чињеница да ансамбли имају боље перформансе од појединачних класификатора, као што је већ речено.

Алгоритми за формирање ансамбала у хардверу

Као што је већ речено у претходној секцији, алгоритми за формирање ансамбала имају драстично веће потребе за ресурсима у односу на алгоритме за индукцију појединачних класификатора. Још једном, хардверска акцелерација ансамбала класификатора пружа начин да се омогући да трајање индукције ансамбала буде упоредиво са трајањем индукције појединачног класификатора, те се у овој дисертацији предлаже хардверска архитектура за акцелерацију EEFTI алгоритма, названа EEFTIP.

Што се тиче хардверске акцелерације ансамбала система за класификацију, према знању аутора, већина се предложених решења бави хардверском имплементацијом ансамбала класификатора који су претходно формирани у софтверу. Аутору је познат само један рад у коме је предложена архитектура за хардверску еволуцију хомогених ансамбала класификатора базираних на стаблима одлуке, али у овом раду се чланови ансамбла индукују инкрементално “greedy” алгоритмом.

EFTI

У овом одељку кратко је описан еволутивни алгоритам за индукцију неортогоналних стабала одлуке неинкременталном методом - EFTI. Основна структура EFTI алгоритма, коју деле многи еволутивни алгоритми, дата је псеудо-кодом испод. Као улаз, EFTI алгоритам добија тренинг скуп инстанци (промењлива train_set у псеудо-коду) који у себи садржи информацију којој класи припада која инстанца.

Као резултат, EFTI алгоритам треба да формира што оптималније стабло одлуке по питању тачности класификације и величине стабла.

Алгоритам 1: Структура EFTI алгоритма

```
def efti(train_set, max_iter):
    dt = initialize(train_set)
    fitness_eval(dt, train_set)

    for iter in range(max_iter):
        dt_mut  = mutate(dt)
        fitness_eval(dt_mut, train_set)

        dt = select(dt, dt_mut)

    return dt
```

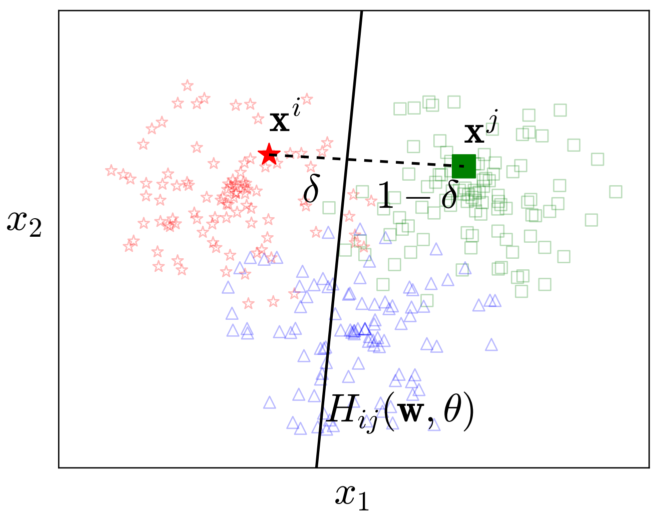
На самом почетку индукције, генерише се стабло од једног чвора и иницијализује његов тест - променљива dt у псеудо-коду. Иницијализација теста се врши на насумичан начин, али је ипак вођена структуром тренинг сета у циљу поспешења конвергенције еволутивног алгоритма. Приступа за насумичну иницијализацију теста коришћен у EFTI алгоритму је базиран на насумично изабраном диполу. Као што је приказано на слици испод, поступак се састоји из постављања хиперравни у простору атрибута $H_{ij}(\mathbf{w},\theta)$, нормално на дуж која спаја две насумично изабране инстанце $\mathbf{x}^i$ и $\mathbf{x}^j$ које припадају различитим класама (приказане црвеним звездама и зеленим квадратима на слици), на раздаљини дефинисаној насумично изабраним параметром δ. Основна претпоставка је да су инстанце у оквиру исте класе на неки начин груписане у простору атрибута, те се овим поступком повећава шанса да ће иако насумична иницијализација теста довести ипак до корисне деобе простора атрибута између ове две класе.

$$\begin{aligned} H_{ij}(\mathbf{w},\theta) &= \mathbf{w}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - \theta, \\ \mathbf{w} &= (\mathbf{x}^i - \mathbf{x}^j), \\ \theta &= \delta\mathbf{w}\cdot\mathbf{x}^i + (1-\delta)\mathbf{w}\cdot\mathbf{x}^j \end{aligned} \tag{2}$$

Након формирања иницијалног теста, функција fitness_eval() рачуна почетни фитнес новонастале јединке. Функција за рачунање фитнеса узима у обзир параметре јединке који су интересантни за оптимизациони процес, комбинује их на основу тежина коју сваки параметар носи (додељених од стране корисника у виду конфигурације алгоритма) и враћа јединствен број који представља фитнес јединке. Параметри стабла од највећег интереса за фазу коришћења су свакако његова тачност и величина, те су ова два параметра и коришћена у имплементацији EFTI алгоритма у овој дисертацији.

Остатак алгоритма покушава да итеративно унапреди фитнес јединке пролазећи у свакој итерацији кроз следеће кораке:

- Мутација - функција mutate() на насумичан начин мења јединку у нади да ће новонастала јединка (променљива dt_mut) напредовати у погледу фитнеса.



Слика 3: Иницијализација теста базирана на насумично изабраном диполу. $H_{ij}(\mathbf{w}, \theta)$ је хиперраван која одговара тесту, $\mathbf{w}$ је вектор коефицијената, а θ праг теста.

Могу се вршити две врсте мутација: мутација вектора коефицијената тестова у чворовима или одузимање/додавање новог чвора у стабло.

- Евалуација фитнеса - функција fitness_eval() на већ поменут начин прорачунава фитнес мутиране јединке
- Селекција - функција select(), у којој се проверава да ли је остварен напредак у погледу фитнеса, у чијем случају се мутирана јединка прихвата за тренутно најбољу. У супротном, да би се омогућило еволутивном алгоритму да напусти локалне оптимуме (и на тај начин има шансу да пронађе глобални оптимум) ипак се даје шанса, обично мала, јединки са нижим фитнесом да буде прихваћена. Другим речима, неке јединке са нижим фитнесом ће насумично бити прихваћене, а остале одбачене.

Након жељеног броја итерација (улазни параметар max_iter), EFTI алгоритам се завршава и враћа тренутно најбољу јединку коју је пронашао.

Копроцесор за еволутивну индукцију целих стабала одлуке - EFTIP

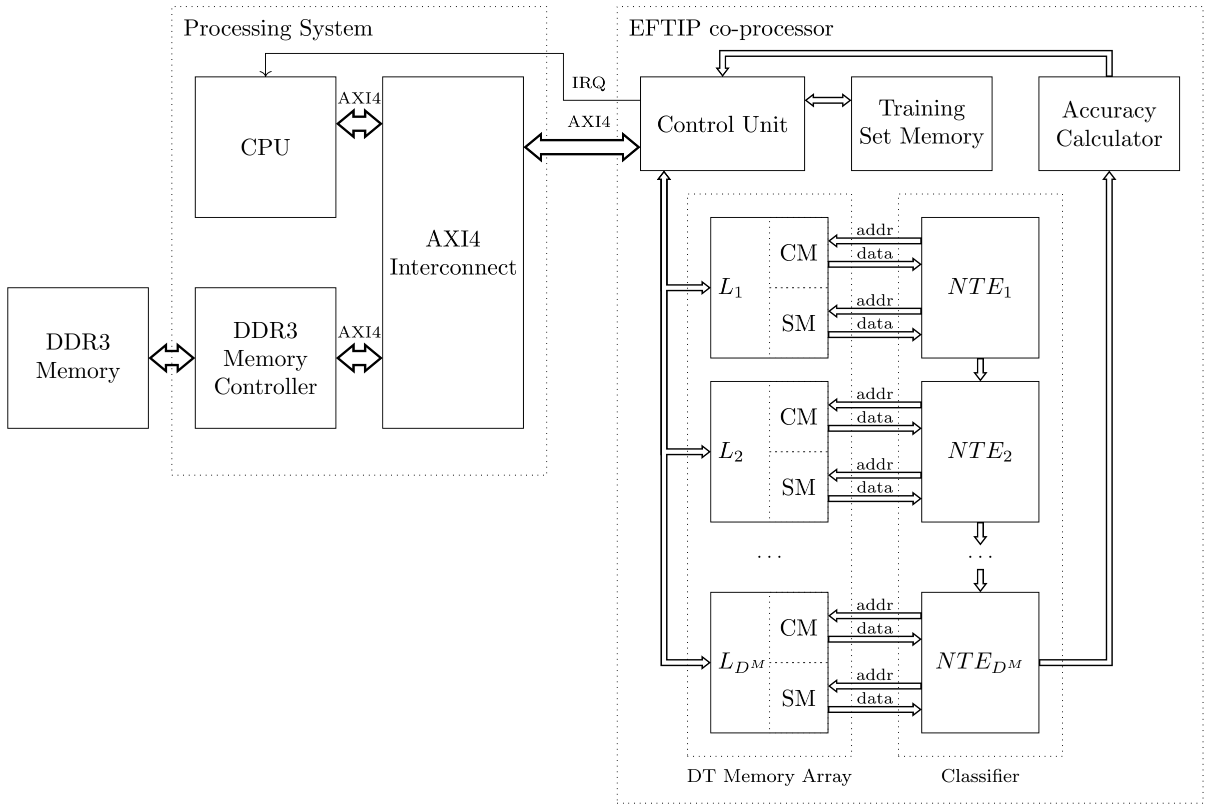
Временски далеко најкомплекснији део EFTI алгоритма је рачунање фитнеса јединке, зато што је у ту сврху потребно извршити класификацију целог тренинг скупа. Да би се извршила класификација, потребно је сваку инстанцу пропустити кроз стабло, при чему је потребно урадити онолико тестова колико је и чворова на путу кроз стабло. У најгорем случају код лоше балансираног стабла, овај број може бити једнак укупном броју чворова у стаблу. Сваки тест се даље састоји од прорачуна суме производа над свим атрибутима, те је комплексност рачунања фитнеса из тог разлога:

$$T(fitness\_eval) = O(N_I \cdot n \cdot N_A) \tag{3}$$

, где је N_I број инстанци у тренинг сету, N_A број атрибута и n број чворова у стаблу.

Анализа временске комплексности алгоритма, као и профајлирање на конкретним примерима, показују да је најзахтевнији део EFTI алгоритма управо прорачун фитнеса. У том светлу, у циљу хардверске акцелерације EFTI алгоритма предложен је HW/SW кодизајн приступ, у којем је најзахтевнији део функције за рачунање фитнеса - рачунање тачности класификације - имплементиран као хардверски копроцесор - EFTIP, а остатак EFTI алгоритма остављен у софтверу да се извршава на централној процесорској јединици (CPU од енг. Central Processing Unit). Додатна предност овакве архитектуре је у томе што се EFTIP копроцесор може користити и за акцелерацију разних других алгоритама за индукцију стабала одлуке базиране на EA, акцелерирајући класификацију тренинг скупа и прорачун тачности стабла, корак који је увек присутан приликом рачунања фитнеса.

Класификација инстанце се врши тако што инстанца почевши од корена пролази кроз стабло одлуке ниво по ниво наниже, где је њен тачан пут одређен исходима тестова у чворовима. За сваку инстанцу врши се само један тест по нивоу стабла. Независно од исхода теста, инстанца увек бива прослеђена на један ниво испод тренутног. Из ових разлога, алгоритам класификације уз помоћ стабла одлуке је згодан за проточну обраду са по једном фазом проточне обраде за сваки ниво стабла. На основу ове анализе предложена је структура EFTIP копроцесора дата на слици испод.



Слика 4: Структура EFTIP копроцесора и његова интеграција са CPU.

EFTIP копроцесор је предвиђен за повезивање са CPU-ом преко AXI4 AMBA магистрале, која је стандардна на ARM архитектурама. Копроцесор пружа следећи интерфејс према софтверу:

- Спуштање тренинг скупа на копроцесор

- Спуштање описа стабла одлуке на копроцесор, како његове структуре, тако и коефицијената свих тестова
- Контрола процеса прорачуна тачности
- Ишчитавање резултата

Главне компоненте EFTIP копроцесора, приказане су на слици изнад:

- Classifier (Класификатор) - Извршава класификацију сваке инстанце тренинг скупа на стаблу одлуке. Овај процес је имплементиран у виду проточне обраде коришћењем одређеног броја NTE модула (од енг. Node Test Evaluator), од којих сваки израчунава тестове за по један ниво стабла одлуке. Параметар D^M представља дубину проточне обраде и самим тим максималну дубину стабла које се може индуковати. На свом излазу, за сваку инстанцу тренинг скупа, Класификатор даје број класе у коју је инстанца класификована.
- Training Set Memory (Меморија за Тренинг Скуп) - Меморија у којој се чувају све инстанце тренинг скупа и шаљу Класификатору на класификацију.
- DT Memory Array (Низ DT Меморија) - Низ меморија у коме се складиште описи стабла одлуке, састоји се од модула L_1 до L_D. Свака фаза проточне обраде у Класификатору захтева сопствену меморију у којој се чувају описи свих чворова на нивоу стабла за који је дата фаза задужена.
- Accuracy Calculator (Калкулатор Тачности) - На основу класификација које Класификатор даје на свом излазу, Калкулатор Тачности рачуна тачност стабла одлуке на тренинг скупу.
- Control Unit (Контролна јединица) - Представља мост између спољашњег AXI4 интерфејса и унутрашњих протокола. Такође координира целокупним процесом прорачуна тачности.

Еволутивни алгоритам за индукцију ансамбала целих неортогоналних стабала одлуке - EEFTI

EEFTI је алгоритам, предложен у овој дисертацији, за индукцију ансамбала стабала одлуке, базиран на EFTI алгоритму. Да би EEFTI алгоритам могао да индукује чланове ансамбла у паралели, згодно је користити "Bagging" алгоритам за индукцију ансамбала. Овај алгоритам предвиђа поделу тренинг скупа на по један подскуп за сваког члана ансамбла који се индукује. Сваки од подскупова се онда користи за индукцију искључиво свог одговарајућег члана ансамбла. Начелно постоје два начина за формирање подскупова:

- насумично одабирање без понављања - формирани подскупови се не преклапају и величине су $N_{IS} = \frac{N_I}{n_e}$, и
- насумично одабирање са понављањем - формирани подскупови су величине $N_{IS} \leq N_I$,

, где је N_{IS} величина подскупова, N_I величина тренинг скупа, а n_e број подскупова, тј. чланова ансамбла.

Сваки појединачни члан ансамбла се даље индукује на основу додељеног подскупа тренинг сета уз помоћ алгоритма сличног EFTI - ју. Псеудо-код EEFTI алгоритма је приказан испод. EEFTI прво дели тренинг скуп на подскупове уз помоћ функције divide_train_set() и чува их у низу task_par, док променљива res чува низ који окупља индуковане чланове ансамбла. Након тога, по један EFTI процес се креира за сваког члана ансамбла, референце на њих се смештају у низ tasks и покреће се њихов рад. Након што су сви EFTI процеси завршили са радом, EEFTI алгоритам је завршен и низ индукованих стабала се враћа као резултат. У оваквој конфигурацији, EFTI процеси су потпуно независни и могу се извршавати у паралели без потребе за међусобном комуникацијом.

Алгоритам 2: Структура EEFTI алгоритма

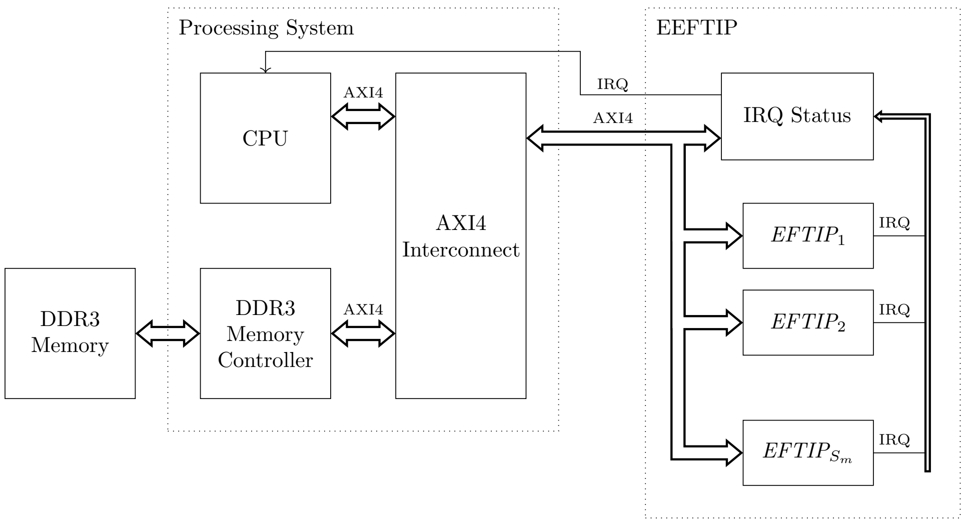
```
def eefti(train_set, ensemble_size):
    train_par = divide_train_set(train_set, ensemble_size)

    res = []
    tasks = []
    for i in range(ensemble_size):
        r = {}
        t = create_task(efti, train_par[i], r)
        res.append(r)
        tasks.append(t)

    while(not all_finished(tasks)):
        pass

    return res
```

Копроцесор за еволутивну индукцију ансамбала целих стабала одлуке - EEFTIP

Пошто је временски најзахтевнији део EFTI алгоритма прорачун тачности стабала одлуке, овај задатак односи највише времена и код EEFTI алгоритма. Из овог разлога, предложен је EEFTIP копроцесор који се састоји од низа EFTIP модула да би омогућио индукцију чланова ансамбла у паралели. Предложена архитектура EEFTIP копроцесора и његова веза са CPU-ом приказана је на слици испод.

Pošto je vremenski najzahtevniji deo EFTI algoritma proračun tačnosti stabala odluke, ovaj zadatak odnosi najviše vremena i kod EEFTI algoritma. Iz ovog razloga, predložen je EEFTIP koprocesor koji se sastoji od niza EFTIP modula da bi omogućio indukciju članova ansambla u paraleli. Predložena arhitektura EEFTIP koprocesora i njegova veza sa CPU-om prikazana je na slici ispod.

EEFTIP копроцесор се повезује са CPU-ом такође преко AXI4 AMBA магистрале, и пружа интерфејс ка сваком од појединачних EFTIP модула, од $EFTIP_1$ до $EFTIP_{SM}$, где је сваки од њих предвиђен да рачуна тачност за по једног члана ансамбла. Параметар S_m представља укупан број EFTIP модула у EEFTIP копроцесору и самим тим максимални број чланова ансамбла које копроцесор може



Слика 5: Структура EEFTIP копроцесора и његова интеграција са CPU-om.

да индукује у паралели. Такође, EEFTIP поседује IRQ Status (од енг. “Interrupt ReQuest Status”) модул, који окупља статусне сигнале свих EFTIP компоненти, омогућава кориснику да их ишчита све заједно и генерише комбиновани сигнал прекида сваки пут када неки од EFTIP модула заврши прорачун тачности.

Што се тиче софтверске стране, EFTI процеси се могу извршавати у паралели, тако што се сваком од њих ексклузивно додели по један EFTIP модул на коришћење. Сваки од EFTI процеса након мутације своје јединке, исту шаље додељеном EFTIP модулу на прорачун тачности. EFTI процес тада враћа програмску контролу оперативном систему, чекајући на прекидни сигнал од стране EEFTIP копроцесора да је прорачун тачности за његову јединку завршен и да резултати могу бити ишчитани. У међувремену, процесорско време се додељује другим EFTI процесима, који га користе на идентичан начин.

UNIVERSITY OF NOVI SAD

FACULTY OF TECHNICAL SCIENCES
NOVI SAD

Bogdan Vukobratović

Hardware Acceleration of Nonincremental Algorithms for the Induction of Decision Trees and Decision Tree Ensembles

PhD Thesis

Novi Sad, 2016

Contents

List of Figures

List of Tables

List of Algorithms

1 Abstract

In this dissertation, new algorithms for the full decision tree (DT) induction are presented, and various possibilities for their implementation are explored. First, the description is given for the novel *EFTI* (Evolutionary Full Tree Induction) algorithm, which was designed in such a way that its implementations can utilize as little hardware resources as possible for the DT induction, as well as to induce as small decision trees as possible, without sacrificing the classification accuracy. This enables the *EFTI* algorithm to be utilized in embedded applications, where the optimal resource utilization is of paramount importance. The implementation of the *EFTI* algorithm for the PC platform is then compared with the PC implementations of the several other existing DT induction algorithms in terms of size and accuracy of the induced DTs and the DT inference times. The experiments show that the proposed *EFTI* algorithm is able to infer much smaller DTs on average, without the significant loss in accuracy, when compared to the top-down incremental DT inducers. On the other hand, when compared to other full tree induction algorithms, it was able to produce more accurate DTs, with similar sizes, in shorter times. Next, the possibility of the hardware acceleration of the *EFTI* algorithm is explored and the results of the algorithm profiling are discussed. Based on the profiling results, the hardware co-processor *EFTIP* (Evolutionary Full Tree Induction co-Processor) is proposed and its architecture is described. Then, the hardware-software (HW/SW) co-design implementation of the *EFTI* algorithm is given, relying on the *EFTIP* co-processor to perform the most computationally intensive part of the evolutionary DT induction, namely the DT accuracy evaluation. Finally, the benefits of using the *EFTIP* co-processor, in terms of the DT induction speed, are discussed in the experimental section, where several *EFTI* algorithm implementations have been compared on the execution times. Next, the algorithm for the induction of the DT ensembles, named *EEFTI* (Ensembles Evolutionary Full Tree Induction) is described. First, the benefits of building an ensemble of DTs is discussed using the results of experiments comparing the accuracy of the DT ensembles with the accuracy of the single DTs. Again, the hardware-software (HW/SW) co-design implementation of the *EEFTI* algorithm is described and the results of the experiments comparing the execution speeds of the different *EEFTI* algorithm implementations are given.

2 Introduction

The research on decision trees is a part of a brother field called machine learning, which in turn is a branch of the artificial intelligence. The machine learning techniques are useful for solving problems when:

- There exists a lot of input data on the problem, but no algorithm (or no efficient one) to produce the output based on the input data is known.
- Either the problem changes with time, or some of its characteristics are not known at the design time, hence an adaptable solution is needed, when the new circumstances arise.

Because of the ever increasing penetration of the machine learning systems into the embedded world, and its even greater potential for in the future, the presented induction algorithms have been tailored for implementation in the embedded systems, in that they use less resources for the operation than the existing solutions. One way of reducing the resource consumption is to induce and thus operate on smaller decision trees. Furthermore, the smaller decision trees also represent a more succinct solution to the problem, which is always preferred in science (Occam's razor *[1]*). Hence, the main motivation for this dissertation was to develop the decision tree induction algorithms that:

1. induce smaller DTs than the existing solutions without the loss of accuracy,
2. can be efficiently used in embedded applications, and
3. are easily parallelizable and hence can be efficiently accelerated in hardware

For big datasets, which are common in practice, the presented decision tree induction algorithms are very time consuming. Hence, the hardware accelerators for *EFTI* and *EEFTI* algorithms are also proposed, namely *EFTIP* and *EEFTIP* co-processors, that significantly reduce their times of execution. Furthermore, the implementations of the proposed induction algorithms that utilize these hardware accelerators are also described.

2.1 Machine learning

Our ever-improving capabilities in collecting the data from the world and constant increase in processing power available to us, have significantly changed our approaches to problem solving in recent decades. Science has also taken advantage of the computers' ability to store massive amounts of data. Ever since it became possible to sequence proteins and the DNA molecule some time after that, immense datasets started emerging from the scientific research in the field of biology, which was followed shortly by other sciences as well. Ever-increasing number and power of telescopes used in astronomy produce larger and larger pools of raw data, with Hubble for an example generating about 140 Gb of raw data each week. Equally, medical science large datasets arise from storing the outcomes of medical tests. The Human Connectome Project aims at mapping the human connectome of a large number of adults and has generated around 2 terabytes of data at the time of writing, and CERN data center processes about 1 petabyte of data each day.

The size and complexity of these datasets mean that humans are unable to extract useful information therefrom, without the help of sophisticated and efficient algorithms. However, there is a scientific field, called the machine learning, that studies the systems that can make use of the abundance of the available data and computational power to solve problems. The machine learning *[2][3]* is a branch of artificial intelligence that studies algorithms and systems that improve their performance with experience, i.e. that can “learn” from the data. In other words, machine learning is about making computers modify or adapt their actions (whether these actions are making predictions, or controlling a robot) so that these actions get more accurate, where accuracy is measured by how well the chosen actions reflect the desired ones. Of particular interest are, of course, the problems that haven’t been satisfactorily solved using other methods.

For an example, one of the challenges to whose solution the machine learning contributed greatly is the problem of self-driving vehicles. There are many aspects of automated driving which are best solved by some type of machine learning system. First of all, the vehicle must be made aware of its surroundings in three dimensions, usually by having multiple cameras that continuously provide the vehicle with images of the space around it. The final goal of this task is to recognize the objects of interest for driving: road lines to follow, pedestrians and other obstacles to avoid, road signs to acknowledge, etc. The object recognition is usually performed in two steps *[4][5]*:

- **clustering** of the image pixels that probably belong to the same object into so called regions of interest (ROI) (also called image segmentation), and
- **classification** of ROIs into classes of known objects

Second of all, based on the surroundings and the driving directions given by the vehicle user, the vehicle needs to devise and maintain a driving strategy, i.e. to determine control signals to vehicle actuators (the steering wheel, gas and break pedals, etc.) in order to, among others, maximize the driving speed within the current speed limit, minimize the risk of collision, etc. These three tasks: the pixel clustering, the ROI classification and driving strategy development are usually solved using machine learning systems that are all induced (built) using different learning strategies, which will be discussed below.

The Figure 2.1 shows an overview of how machine learning is used to address a given task as described in *[2]*. A task has a goal of solving a certain problem of interest regarding the objects of the problem domain, which are in turn defined in terms of its attributes (also called features). The choice of attributes defines a ‘language’ in which all the objects in the problem domain get their relevant aspects described. Once a suitable attribute representation is selected, the machine learning system will not be concerned with the domain objects themselves, but only operate on their attribute representations. Domain objects are usually represented in the form of an attribute vector, also called an instance (since it acts as a problem instance for the machine learning system), which lists the values of all object attributes. Hence, the goal is to obtain an appropriate mapping for a task, called a model, from attributes to the desired outputs, which in turn correspond to the outputs of the problem that is being solved by the machine learning system. Obtaining such a model from training data is what constitutes a learning problem.

Machine learning systems can be constructed using supervised learning, unsupervised learning or any combination of the two techniques *[2][3]*. Supervised learning implies providing the desired responses to the instances of the training set to construct the system, while unsupervised learning implies constructing the system based on the instances only. When the supervised learning is used, the lifetime of a machine learning system usually comprises two distinct phases:

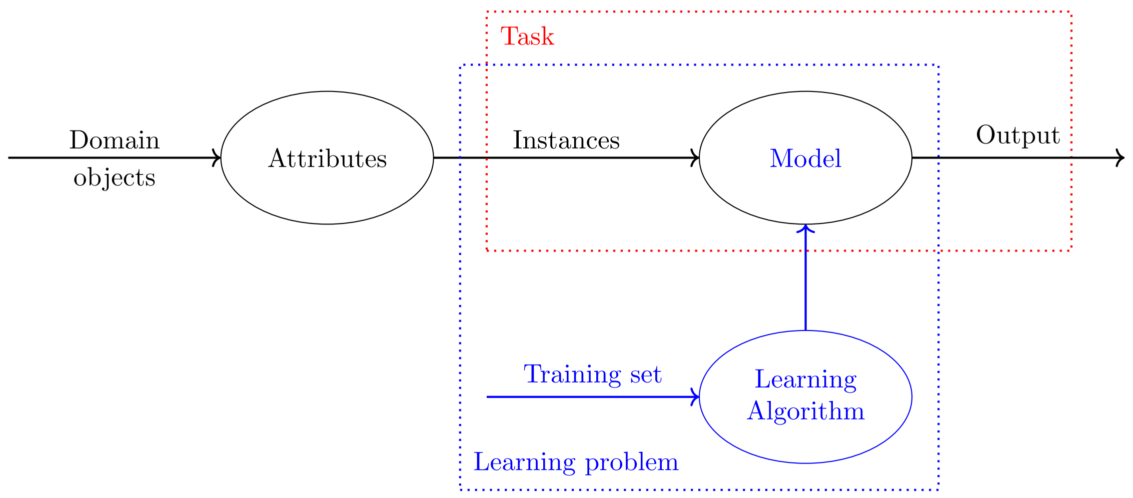


Figure 2.1: *An overview of how machine learning is used to solve problems in a certain domain, by constructing the model via process of learning on the training set.*

- the training phase (induction or learning), during which the learning problem is solved and the model is developed, and
- the deployment phase, during which the model is used to process new data

For an example, the classification of ROIs for self-driving vehicles is usually performed by the machine learning systems, induced by the method of supervised learning. During the training phase, a training set is used to build the system, which comprises input data instances and the desired system responses to them. Once constructed, the system is ready to be used, where new, previously unseen data, will arrive and the system must provide the responses using the knowledge extracted from the training set.

When using unsupervised learning, the correct responses to the input data are not provided, instead the algorithm tries to identify similarities between the inputs, so that instances that have something in common solicit similar outputs. The statistical approach to unsupervised learning is known as density estimation. The clustering of image pixels to obtain ROIs for self-driving vehicles is an example of machine learning system that uses unsupervised learning. The system is never trained with the examples on how to map pixel groups to ROIs (since there are too many possible correct mappings), but has to apprehend it on its own, based on the attributes the pixels in a group share.

Reinforcement learning is somewhere between supervised and unsupervised learning. The learning algorithm gets told when the answer is wrong, but without the advice on how to correct it. It has to explore and try out different possibilities until it discovers how to get the answer right. Reinforcement learning is sometime called learning with a critic, because of the monitor that scores the answer, but does not suggest improvements. Developing the right driving strategies for self-driving vehicles is usually performed by the machine learning system that was trained using the reinforcement learning procedure. To provide for learning purposes the right combination of the positions of the steering wheel, acceleration and breaking pedals, etc. in each time instant, with dynamic circumstances, would be an impossible task to perform. Hence, in order to develop correct driving strategies, the machine learning system can be let to drive the vehicle and be given positive or negative feedback during the process based on some general parameters, for an example: the driving speed or the distance it holds from the objects around.

One of the main features of machine learning systems is the power of generalization, allowing them to perform well on new, unseen data instances, after having experienced a learning procedure. It is of special interest to maintain the power of generalization of the system being trained by the supervised learning method. A machine learning problem may have multiple solutions, i.e. multiple models can perform equally well on the training set. If care is not taken, it is possible for the induced machine learning system to perform excellently on the training set, but fail when used on new data. This phenomenon is called overfitting, in that the induced model learned too many features of the training set that are not shared by other problem instances, i.e. the model was made to overly fit the training set. Good performance on the training data is only a means to an end, not a goal in itself, since it is the performance on the new data that should be maximized. By maximizing the induced model's power of the generalization, it is in the same time made to better deal with noise, which represents small inaccuracies in the data that are inherent in measuring any real world process. The model must not take the instance attribute values too literally, but should expect that each of them has some noise superimposed.

The machine learning systems can perform various tasks, such as classification, regression, clustering, etc. The classification implies categorizing problem instances in some number of discrete classes. Sometimes it is more natural to abandon the notion of discrete classes altogether and instead predict a real number, i.e. perform the task which is called regression. The task of grouping data without prior information on the groups is called clustering, which usually uses models induced by the method of unsupervised learning. A typical clustering algorithm works by assessing the similarity between instances (the things we're trying to cluster, e.g., connected pixels) and putting similar instances in the same cluster and 'dissimilar' instances in different clusters. There are many other patterns that can be learned from the data in an unsupervised way. Association rules are a kind of pattern that are popular in marketing applications, and the result of such learned patterns can often be found on online shopping web sites.

In the open literature, a range of machine learning systems have been introduced, including decision trees (DTs) *[6][7]*, support vector machines (SVMs) *[8]* and artificial neural networks (ANNs) *[9]*.

2.2 Decision Trees

Widely used machine learning model for classification tasks is a DT classifier. The classification process by the DT can be depicted in a flowchart-like tree structure given in the Figure 2.2. Due to their comprehensible nature, which resembles the process of human reasoning, DTs have been widely used to represent classification models. Among other machine learning algorithms DTs have several advantages, such as the robustness to noise, the ability to deal with redundant or missing attributes, the ability to handle both numerical and categorical data and the facility of understanding the computation process.

In theory, DTs can have an arbitrary branching factor (n-ary DTs), but the binary DTs (with the branching factor of 2), i.e. the DTs with only two children per node, are used most often for being easiest to implement and manipulate. Furthermore, a tree with an arbitrary branching factor can always be represented by a functionally equivalent binary DT *[10]*. The Figure 2.2

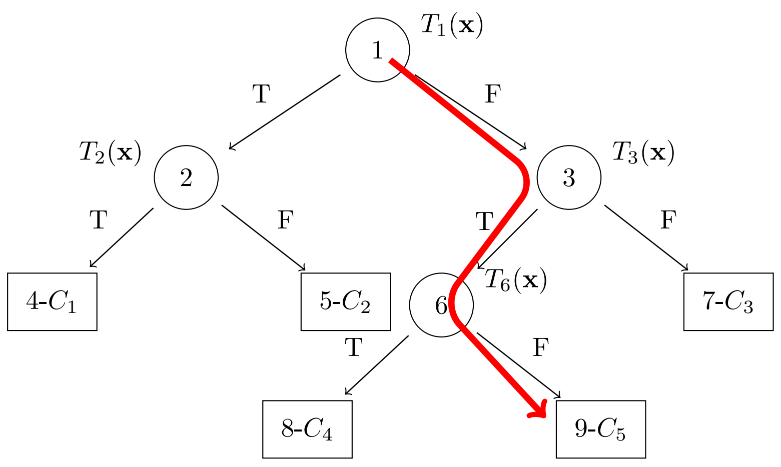


Figure 2.2: *The classification process by the binary DT.*

shows the process of classification by a binary DT. The DT in the figure consists of 4 nodes represented by circles numbered 1, 2, 3 and 6. The DT also has 5 leaves represented by squares numbered 4, 5, 7, 8 and 9, where each of the leaves has a class assigned to it (C_1 through C_5 in this example). The classification is performed by letting instances traverse the tree, starting from the root (enumerated as 1), until they reache one of the leaves. The instance is then classified into the class assigned to the leaf in which it finished the traversal.

Each of the DT nodes is assigned a test: T_1, T_2, T_3 and T_6 in this example. In each node the instance visits during its traversal through the DT, the node test is used to determine through which of the node's children will the traversal continue, based on the instance's attribute values. In case of a binary DT, the node test decision is likewise binary. If the test evaluates to **`True`** (T), the DT traversal is continued via the left child, otherwise if it evaluates to **`False`** (F), it is continued via the right child. The final path of the instance through the DT depends on the test results in all the nodes the instance encounters during the traversal.

Each machine learning problem needs to have a domain defined, which is in turn given as the set of all domain objects. First, the set of attributes is chosen to uniquely represent the domain objects in form of the attribute vector - $\mathbf{x}$. Also, the domain of each attribute needs to be defined, where there are usually two choices:

- the domain can be a finite set of unordered values, in which case the attribute is called categorical, or
- the domain can be a subset of the set of the real numbers, in which case the attribute is called numerical.

The set of all possible attribute vectors forms the N_A - dimensional attribute space, where N_A is the number of attributes that are used to describe the domain object, i.e. the size of the attribute vector $\mathbf{x}$. In the context of the attribute space, each binary DT node test splits the space into two regions, one containing all the instances for which the test produced the result **`True`** and the other containing the rest of the instances, for which the test evaluated to **`False`**. Each DT node can be thus assigned a sub-region of the attribute space, that in turn contains all the instances that pass through that node during their traversal of the DT. Hence, each node splits the region assigned to it by into two sub-regions and assigns each of them to one of its children. This process of attribute space partitioning starts from the DT root, which is assigned whole attribute space (every instance needs to visit the root node), and continues downwards to the DT leaves. The final result of this process is a clear partition of the attribute space into a number of disjoint regions, each associated with one leaf node. Each of these regions in the partition can thus be assigned the associated leaf's class, meaning that all the instances contained in the region will be classified into that class.

Based on the characteristics of the functions implementing the node tests, the DTs can be categorized into: orthogonal (also univariate), oblique (also multivariate) and nonlinear. The names of the categories were derived from the shape of the hypersurface defined by their tests. Hence, the orthogonal DTs divide the attribute space using the hyperplanes orthogonal to some attribute axis, the oblique DTs using oblique hyperplanes, and nonlinear DTs using nonlinear hyperplanes.

This thesis focuses on the oblique binary classification DTs. The tests performed by an oblique DT in each node are afine and have the following form:

$$\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{N_A} w_i \cdot x_i < \theta, \tag{1}$$

where $\mathbf{w}$ represents the coefficient vector and θ (called the threshold) models the afine part of the test.

Next, an example describing the classification process by oblique DTs will be given. The Figure 2.3 shows a dataset named, `yinyang` that will be used for this example, plotted in its attribute space. The dataset instances are conveniently described using only two attributes x_1 and x_2, so that they can be represented in 2-D attribute space. The dataset comprises instances belonging to one of the two classes: C_1 and C_2. Each instance is represented in the figure by either a red star (if it belongs to the class C_1) or a blue square (if it belongs to the class C_2), with its position defined by the values of its attributes.

An example of the oblique binary DT that can be used to accurately classify the instances of the yinyang dataset, is shown in the Figure 2.4. Since this is an oblique DT, each of its node tests follows a form defined by the equation (1). Each DT leaf has one of two classes of the yinyang dataset assigned to it. The classification is performed by letting each instance of the yinyang dataset traverse the DT, starting from the root node, in order to be assigned a class. During the traversal, tests are evaluated at each of the DT nodes along the instance path. Based on the results of the node test conditions (**`True`** or **`False`**), the DT traversal is continued accordingly until a leaf is reached, when the instance is classified into the class assigned to that leaf. One possible traversal path is shown in the Figure 2.4, where the instance got classified into the class C_1 after the traversal.

As it was already discussed, a different way of looking at the classification process by the DT is by examining what happens in the attribute space. The structure of the attribute space regions is defined by the DT node tests, resulting in one region assigned to each node and each leaf of the DT as shown in the Figure 2.5. The dashed lines in the figure represent the 1-D hyperplanes (lines in this case) generated by the node tests that partition the attribute space. The regions of the final partition are the ones assigned to the DT leaves, and each of them is marked with the ID of its corresponding leaf and the class assigned to that leaf. The regions assigned to the non-leaf nodes can be easily obtained from the figure plot and the DT structure from the Figure 2.4, by noticing that the node's region equals the union of its children regions. Working from the bottom up recursively, regions for all DT nodes can be obtained by combining the regions assigned to their descendents.

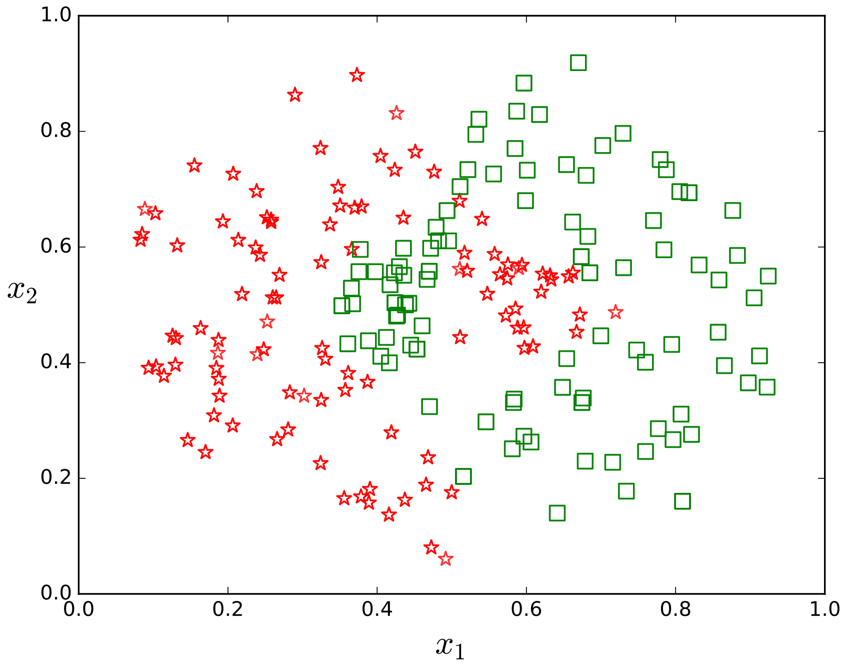


Figure 2.3: *The yinyang dataset used for the demonstration of the classification process by oblique DTs. Instances of the dataset are described using two attributes x_1 and x_2, and can belong to one of the two classes C_1, represented by the red star symbols, and C_2, represented by the blue square symbols.*

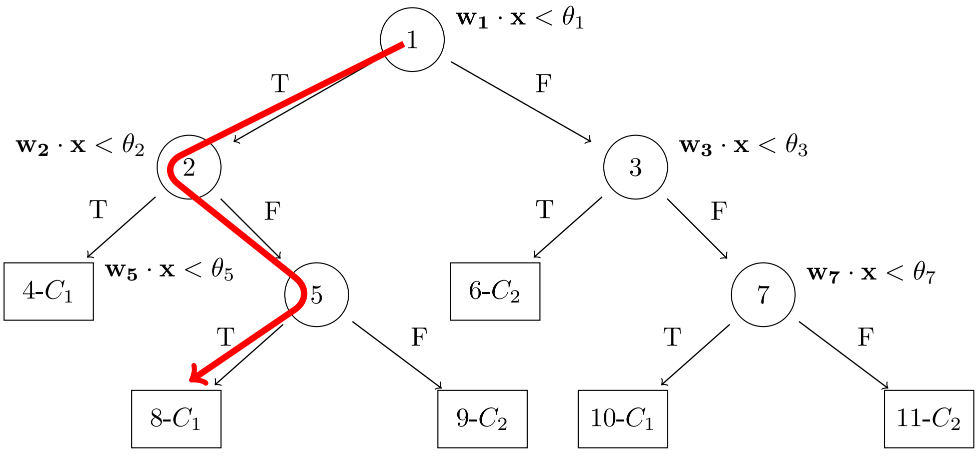

$\mathbf{w_1} \cdot \mathbf{x} < \theta_1$
1
T
F
$\mathbf{w_2} \cdot \mathbf{x} < \theta_2$
2
3
$\mathbf{w_3} \cdot \mathbf{x} < \theta_3$
T
F
T
F
4-C_1
$\mathbf{w_5} \cdot \mathbf{x} < \theta_5$
5
6-C_2
7
$\mathbf{w_7} \cdot \mathbf{x} < \theta_7$
T
F
T
F
8-C_1
9-C_2
10-C_1
11-C_2

Figure 2.4: *Oblique binary DT that could be used to classify the instances of the yinyang dataset ploted in the Figure 2.3. The red curvy line shows the traversal path for one possible instance. This example traversal path can be visually presented via series of dataset attribute space regions, as ploted in the Figure 2.6.*

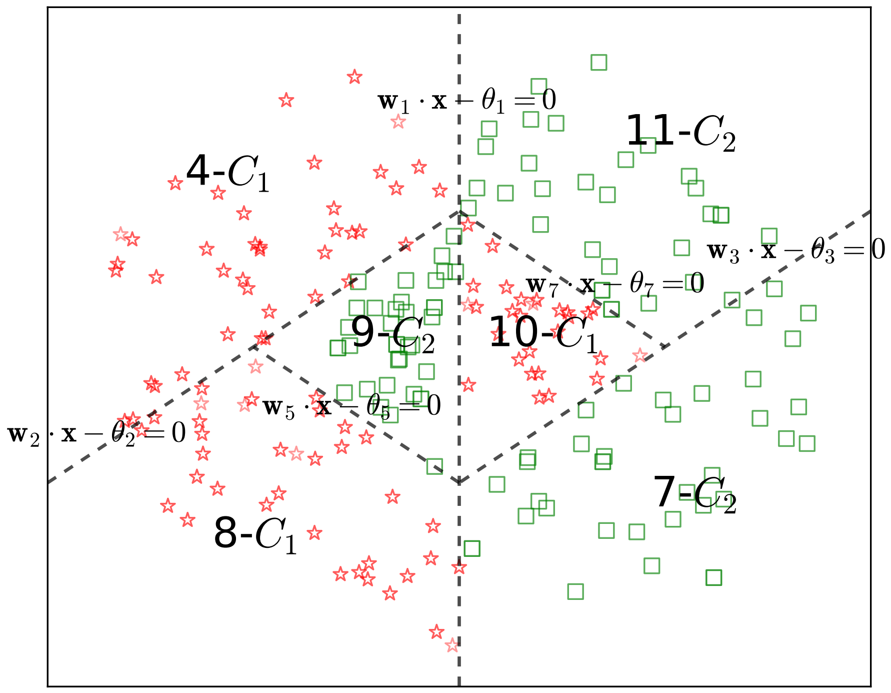
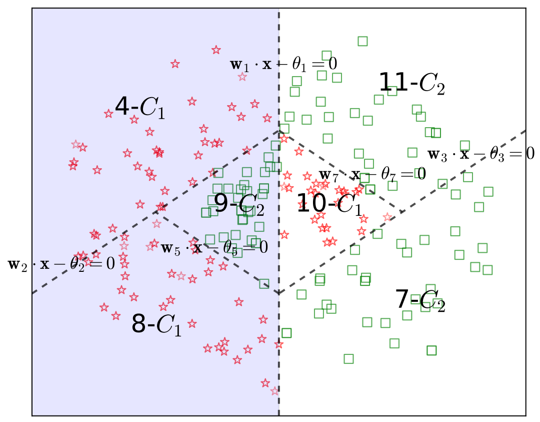
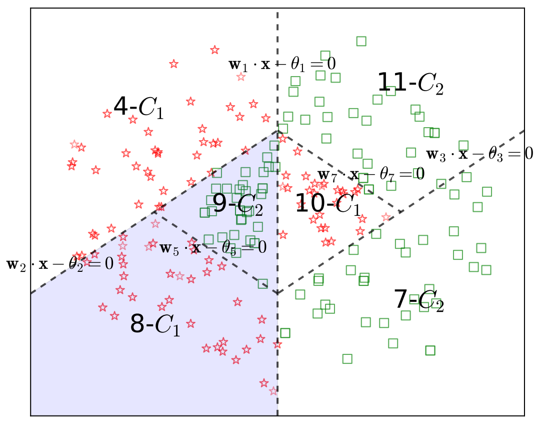


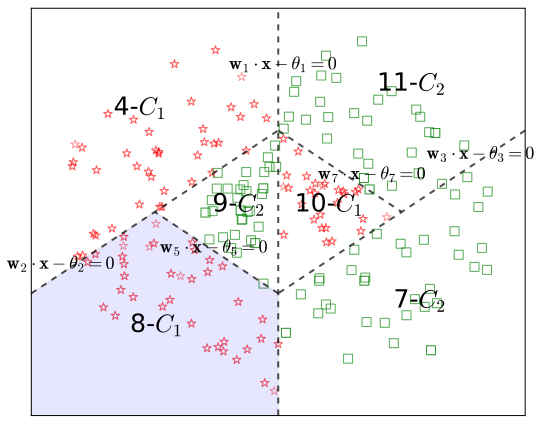
Figure 2.5: *The attribute space partition of the yinyang dataset from the Figure 2.3 generated by the DT from the Figure 2.4. The dashed lines on the figure represent the hyperplanes generated by the node's tests that partition the attribute space into the regions, each corresponding to a leaf of the DT. Each of the attribute space regions is marked with the ID of its corresponding leaf and the class assigned to the leaf.*



(a) *Region of the attribute space assigned to the node 2 of the DT from the Figure 2.4.*



(b) *Region of the attribute space assigned to the node 5 of the DT from the Figure 2.4.*



(c) *Region of the attribute space assigned to the node 8 of the DT from the Figure 2.4.*

Figure 2.6: *The figure shows the attribute space regions assigned to the nodes and leafs an example instance visits during its traversal along the line shown in the Figure 2.4.*

In order to find out in which region the instance resides, and thus to which class it belongs, we need to let the instance traverse the DT. The Figure 2.6 shows this process for the example traversal path shown in the Figure 2.4. At the begining of the classification, when the instance starts at the root, all the regions are valid candidates. After the root node test is evaluated, the location of the instance can be narrowed down to the regions either to the left or to the right of the hyperplane $\mathbf{w_1} \cdot \mathbf{x} - \theta = 0$, generated by the root node test. For this example instance, the root node test evaluated to **`True`**, the instance continues to the node 2, and the location of the instance is narrowed down to the region assigned to the node 2 and shown in the Figure 2.6a. Then, the test of the node 2 is evaluated for the instance, and it turns out to be **`False`**, hence the instance continues to the node 5 and the number of possible regions is reduced again to the ones marked in the Figure 2.6b, i.e. to the part of the attribute space assigned to the node 5. Finally, the node 5 test is evaluated to **`True`**, the instance hits the leaf node 8 and it is finally located in the region marked in the Figure 2.6c and assigned the C_1 class.

2.3 Decision tree induction

In the field of machine learning, as is with most other scientific disciplines, simpler models are preferred over the more complex ones as stated in the principle of Occam's razor *[1]*. The same principle, but in terms of the information theory, was proposed in *[11]* under the name Minimum Description Length (MDL). In essence, it says that the shortest description of something, i.e. the most compressed one, is the best description. The preference for simplicity in the scientific method is based on the falsifiability criterion. For each accepted model of a phenomenon, there is an extremely large number of possible alternatives with an increasing level of complexity, because aspects in which the model fails to correctly describe the phenomenon can always be masked with ad hoc hypotheses to prevent the model from being falsified. Therefore, simpler theories are preferable to more complex ones because they are more testable. Hence, there is an obvious benefit for having the algorithm that induces smaller DTs, since smaller DT corresponds to a simpler description of a phenomenon being modeled by it.

Second, with growth and advancements in the field of electronics, wireless communications, networking, cognitive and affective computing and robotics, embedded devices have penetrated deeper into our daily lives. In order for them to seamlessly integrate with our dynamic daily routine, for execution of any non-trivial task, they need to employ some sort of machine learning procedure. Hence, the *EFTI* algorithm, proposed in this thesis, was designed with its implementation for the embedded systems in mind. In other words, the *EFTI* algorithm was designed to require as little hardware resources for implementation as possible in order for it to be easily integrated into an embedded system. Furthermore, it is shown in this thesis that it induces smaller DTs, without the loss of accuracy, then the other existing induction algorithms, which then require less resources to be operated on and are thus more suitable for the embedded applications.

The DT induction phase can be very computationally demanding and can last for hours or even days for practical problems, especially when run on the less powerful, embedded processors. By accelerating the *EFTI* algorithm in hardware, the machine learning systems could be trained faster, allowing for shorter design cycles, or could process larger amounts of data, which is of particular interest if the DTs are used in the data mining applications *[12]*. This might also allow the DT learning systems to be rebuilt in real-time, for the applications that require such rapid adaptation, such as: machine vision *[13][14][15][16]*, bioinformatics *[17][18]*, web mining *[19][20]*, text mining *[21][22]*, etc. Hence, the *EFTI* algorithm was designed to be parallel in nature and thus be easily accelerated by an application specific co-processor. Furthermore, some of the world leading semiconductor chip makers offer the solutions which consist of a CPU integrated with an FPGA, like Xilinx with its Zynq series and Intel with its new generation Xeon chips. The hardware accelerated implementation of the *EFTI* algorithm can be readily implemented on these devices, with the hardware for the *EFTI* algorithm acceleration built for the integrated FPGA.

2.3.1 General approaches to DT induction

Finding the smallest DT consistent with the training set is an NP-hard problem *[23]*, hence, in general it is solved using some kind of heuristic. The DT is said to be consistent with the training set if and only if it classifies all the training set instances in the same way as defined in the training set. There are two general approaches to DT induction using supervised learning: incremental (node-by-node, also known as Top-Down Induction of Decision Trees, or TDIDT) and nonincremental (or full tree) induction.

The incremental approach uses greedy top-down recursive partitioning strategy of the training set for the tree growth. The algorithm starts with an empty DT and continues by forming the root node test and adding it to the DT. In the attribute space, the root node test splits the training set in two partitions, one that will be used to form the root's left child subtree, and the other the right child subtree. In other words, the root node is assigned the whole training set, which is partitioned in two by the root node test and each partition is assigned to one of the root's two children. The node test coefficients are optimized in the process of maximizing some cost function measuring the quality of the split. Iteratively, the nodes are added to the DT, whose tests further divide the training set partitions assigned to them. If the node is assigned a partition of the training set where all instances belong to the same class (the partition is clean), no further division is needed and the node becomes a leaf with that class assigned to it. Otherwise, the process of partitioning is continued until only clean partitions remain. In this stage, the induced DT is considered overfitted, i.e it performs flawlessly on the training set, but badly on the instances outside the training set. The common approach for increasing the performance of the overfitted DT on new instances is prunning, which strips some subtrees from the DT according to some algorithm.

The incremental approach is considered greedy in the sense that the node test coefficients (coefficient vector $\mathbf{w}$ and threshold value θ) are optimized by examining only the part of the training set assigned to the current node, i.e. based on the "local" information. The information on how the training set partitions are handled in other subtrees of the DT (subtrees not containing the node currently being inserted into the DT) are not used to help optimize the test coefficients. Furthermore, by the time the node has been added to the DT and the algorithm continued creating other nodes, the situation has changed and the new information is available, but it will not be used to further optimize the test of the node already added to the DT. This means that only some local optimum of the induced DT can be achieved.

Incremental algorithms use a simpler heuristic and are computationally less demanding than the full DT inducers. However, the algorithms that optimize the DT as a whole, using complete information during the optimization process, generally lead to more compact and possibly more accurate DTs when compared with incremental approaches. Furthermore, the DTs can be induced both using only axis-parallel node tests or using oblique node tests. The advantage of using only axis-parallel tests is in reduced complexity, as the task of finding the optimal axis-parallel split of the training set is polynomial in terms of N_A and N_I. More precisely, the optimization process needs to explore only $N_A \cdot N_I$ distinct possible axis-parallel splits *[23]*. On the other hand, in order to find the optimal oblique split, total of $2^{N_A} \cdot \binom{N_I}{N_A}$ possible hyperplanes need to be considered *[23]*, making it an NP-hard problem. On the other hand, the DTs induced with oblique tests often have much smaller number of nodes than the ones with axis-parallel tests. Hence, in order to fulfill its goal of inducing smaller DTs than existing solutions, the *EFTI* algorithm needs to implement oblique DT induction.

Various algorithms for incremental DT induction have been proposed in the open literature. The ID3 algorithm proposed in *[24]* was designed to operate mainly on categorical attributes. In the DT created by the ID3 algorithm, each node test operates on a single attribute only. The number of outcomes the test can produce equals the number of different values the attribute can take, and the attribute space will be split into the same number of regions by the test. In order to choose which attribute should be used for the test in a node, the information gain (IG), given by the equation (2), is calculated for all possible attributes. The information gain is a difference between the information entropy of the attribute space region assigned to the node, and the combined entropies of the regions produced by the node test split.

$$IG(A_i, S) = H(S) - \sum_{t \in T} p(t)H(t), \tag{2}$$

where $H(S)$ is information entropy of the region assigned to the node, T is the partition in subregions generated by the node test based on the attribute A_i, $p(t)$ is the proportion of the number of elements in subregion t to the number of elements in the region assigned to the node S and $H(t)$ is the information entropy of the subregion t. The attribute whose test would produce the largest IG is selected to form the node test. As an improvement to ID3, the C4.5 algorithm was published in *[25]*. C4.5 introduced the possibility to handle continuous attributes, to handle instances whose attributes are missing and introduced the prunning step after the DT has been created.

The Classification and Regression Tree (CART) algorithm was introduced in *[26]*, that unlike ID3 induces binary DTs. Similar to ID3, only the value of a single attribute is tested in each node test, hence CART produces axis-parallel binary splits. When searching for the best test for a node, CART evaluates every possible way in which attribute domain could be split in two, hence the attribute domains need to be discrete and finite. Various measures could be used for selecting the best split: Gini index, Twoing, information entropy, etc., which can all be plugged in to the equation (2) instead of the information entropy *H* to get a numerical estimate for the efficiency of the split. An extension to CART that generates oblique tests has also been proposed in *[26]* by the name CART with linear combinations or CART-LC. The OC1 algorithm was proposed in *[23]*, which improves upon the CART-LC algorithm. While considering the best split for a DT node, OC1 first searches for the best axis-parallel test for the node. OC1 then tries to produce an oblique test that will outperform it, and if that fails, the algorithm defaults to the axis-parallel test. Furthermore, unlike CART-LC that is fully deterministic, OC1 incorporates the ideas from simulated annealing algorithm, which address the issue of escaping local optima and enable OC1 to produce different DTs from a single training set. Various extensions to OC1 algorithm based on evolutionary algorithms were introduced in *[27]*, namely: OC1-ES (OC1 extension using evolution strategies), OC1-GA (OC1 extension using genetic algorithms) and OC1-SA (OC1 extension using simulated annealing). These extensions were specifically employed in the process of searching for the best oblique split. The authors of so called C4.45 and C4.55 algorithms claim in [28] to have acheived performance superior to C4.5 algorithm with respect to both accuracy and size, by using various optimizational techniques to improve upon original C4.5 algorithm.

The Univariate Margin Tree (UMT) algorithm given in [29], borrows the ideas from linear SVMs for the way it tries to find the optimal split for a node. Fisher's decision tree algorithm for incremental oblique DT induction, proposed in [30], implements yet a different strategy for obtaining the split using Fisher's linear discriminant, and reported obtaining smaller DTs, with shorter induction time without the loss in accuracy when compared to C4.5. A bottom-up induction approach was explored in [31], resulting in the Bottom-Up Oblique Decision-Tree Induction Framework (BUTIF). This algorithm operates by clustering the instances based on their classes and position in the attribute space, and asssigning those clusters to the leaf nodes prior to creating the trunk of the DT. Starting from the formed leaves, the BUTIF algorithm generates the DT by merging the existing subtrees until finally the root is formed. In [32], authors employed the HereBoy evolutionary algorithm to optimize the positions of the node test hyperplanes.

The alternative to the incremental DT induction is the full DT induction. In this approach a complete DT is manipulated during the inference process. Acording to some algorithm, the tree nodes are added or removed, and their associated tests are modified. Considerable number of full DT inference algorithms has been also proposed. A genetic algorithm operating on full DTs as individuals, called GaTree, was introduced in [33]. Another algorithm based on genetic algorithms, called GALE and proposed in [34], attempted to extract additional parallelism from the induction process by employing ideas from the field of cellular automata and the Pittsburgh approach [35]. In [36], genetic programming was employed to create a nested structure of IF-THEN-ELSE statements that is homologous to a DT. Finally, the ant colony optimization technique was used for the algorithms introduced in [37][38].

2.3.2 Evolutionary oblique full DT induction

Since the process of finding the optimal oblique DT is a hard algorithmic problem, most of the oblique DT induction algorithms use some kind of heuristic for the optimization process, which is often some sort of evolutionary algorithm (EA). The Figure 2.7 shows the taxonomy of EAs for the DT induction as presented in [39].

The evolutionary algorithms for inducing DTs by global optimization (the full DT induction) are usually some kinds of Genetic Algorithms [33][34][40], which in turn operate on a population of candidate solutions. The typical populations used by these algorithms contain tens or even hundreds of individuals. In order to save on needed resources for the implementation, the *EFTI* algorithm operates only on a single candidate solution and single result of its mutation, which classifies it in the class of (1+1)-ES (Evolutionary Strategy). Hence, the proposed algorithm requires one or even two orders of magnitude less hardware resources for the implementation then the existing evolutionary algorithms. Furthermore, stohastic algorithms such as *EFTI*, that do not use populations of candidate solutions and thus do not employ recombination, can also be classified in the class of Stochastic Hill Climbing algorithms [41]. Furthermore, the *EFTI* algorithm utilizes the simple technique of adaptive random search for mutations, which can be implemented efficiently both regarding the time needed for execution and hardware resources needed (having embedded systems as target in

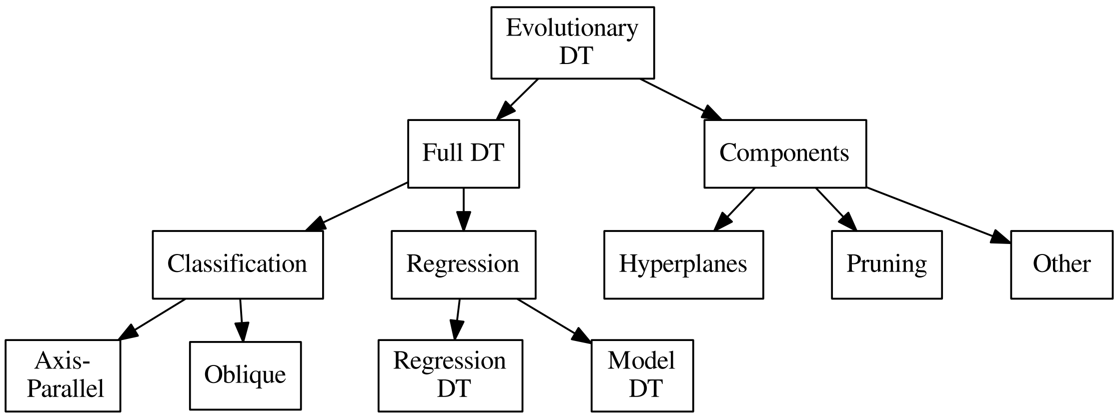


***Figure* 2.7:** *The taxonomy of evolutionary algorithms for DT induction as presented in* [39].

mind). As far as author is aware, *EFTI* is the first full DT building algorithm that operates on a single-individual population. However, it also proved to provide smaller DTs with similar or better classification accuracy than other well-known DT inference algorithms, both incremental and full DT *[42]*.

2.4 Hardware aided decision tree induction

In order to accelerate the DT induction phase, two general approaches can be used. The first approach focuses on developing new algorithmic frameworks or new software tools, and is the dominant way of meeting this requirement *[43][44]*. The second approach focuses on the hardware acceleration of machine learning algorithms, by developing new hardware architectures optimized for accelerating the selected machine learning systems.

The hardware acceleration of the machine learning algorithms receives a significant attention in the scientific community. A wide range of solutions have been suggested in the open literature for various predictive models. The author is aware of the work that has been done on accelerating SVMs and ANNs, where hardware architectures for the acceleration of both learning phase and the execution have been proposed. The architectures for the hardware acceleration of SVM learning algorithms have been proposed in *[45]*, while the architectures for the acceleration of previously created SVMs have been proposed in *[46][47][48][49]*. The research in the hardware acceleration of ANNs has been particularly intensive. Numerous hardware architectures for the acceleration of already learned ANNs have been proposed *[50][51][52]*. Also, a large number of hardware architectures capable of implementing ANN learning algorithms in hardware have been proposed *[53][54][55]*. However, in the field of hardware acceleration of the DTs, the majority of the papers focus on the acceleration of already created DTs *[56][57][58]*. Hardware acceleration of DT induction phase is scarcely covered. The author is currently aware of only two papers on the topic of hardware acceleration of the DT induction algorithms *[59][60]*. However, both of these results focus on accelerating greedy top-down DT induction approaches. In *[59]* the incremental DT induction algorithm, where EA is used to calculate the optimal coefficient vector one node at a time, is completely accelerated in hardware. In *[60]* a HW/SW approach was used to accelerate the computationally most demanding part of the well known CART incremental DT induction algorithm.

In this thesis, a co-processor called *EFTIP* (Evolutionary Full Tree Induction co-Processor) that can be used for the acceleration of the *EFTI* algorithm is proposed. As mentioned earlier, full DT induction algorithms typically build better DTs (smaller and more accurate) when compared to the incremental DT induction algorithms. However, full DT induction algorithms are more computationally demanding, requiring much more time to build a DT. This is one of the reasons why incremental DT induction algorithms are currently dominating the DT field. Developing a hardware accelerator for full DT induction algorithm should significantly decrease the DT inference time, and therefore make it more attractive. As far as the author is aware, this is the first hardware accelerator in open literature concerned with the hardware acceleration of full DT induction algorithm. Being that the EAs are iterative by nature and extensively perform simple computations on the data, the *EFTI* algorithm should benefit from the hardware acceleration, as would any other DT induction algorithm based on the EAs. Proposed *EFTIP* co-processor is designed to accelerate only the most computationally intensive part of the *EFTI* algorithm, leaving the remaining parts of the algorithm in software. It is shown later in the thesis, that the most critical part of the *EFTI* algorithm is the training set classification step from the fitness evaluation phase. *EFTIP* was designed to accelerate this step in hardware. Another advantage of this HW/SW co-design approach is that the proposed *EFTIP* co-processor can be used with a wide variety of other EA-based DT induction algorithms *[39][36][40][34][33]* to accelerate the training set classification step that is always present during the fitness evaluation phase.

2.5 Induction of decision tree ensembles

The ensemble classifier systems can be used to further improve the classification performance *[61]*. The ensemble classifier combines predictions from several individual classifiers in order to obtain a classifier that outperforms every one of them. The ensemble learning requires creation of a set of individually trained classifiers, typically DTs or ANNs, whose predictions are then combined during the process of classification of previously unseen instances. Although simple, this idea has proved to be effective, producing systems that are more accurate than a single classifier.

In the process of creation of ensemble classifiers, two problems have to be solved: ensuring the diversity of ensemble members and devising a procedure for combining individual member predictions in order to amplify correct decisions and suppress the wrong ones. Some of the most popular methods for ensuring ensemble's diversity are Breiman's bagging *[62]*, Shapire's boosting *[62]*, AdaBoost *[62]*, Wolpert's stacked generalization *[63]*, and mixture of experts *[64]*. Most commonly used combination rules include: majority voting, weighted majority voting and behavior knowledge spaces *[65]*.

The main advantages of an ensemble over single classifier systems are the higher accuracy and greater robustness. However, large amounts of memory are needed to store the ensemble classifier and high computing power is required to calculate the ensemble's output, when compared with the single classifier solutions, leading to much longer ensemble inference and instance classification times. This is because ensemble classifiers typically combine 30 or more individual classifiers *[62]* so, if we want to get the same performance as with the single classifier system, 30+ times more memory and computing power would be required. Once more, hardware acceleration of ensemble classifier offers a way of achieving this goal.

In this thesis, a DT ensemble evolutionary induction algorithm *EEFTI* (Ensembles Evolutionary Full Tree Induction), based on the *EFTI* algorithm and the Bootstrap Aggregation (also known as Bagging). The Bagging algorithm was chosen since it makes the induction of the individual ensemble members completely decoupled from each other, making *EEFTI* very well suited for the parallelization and hence hardware acceleration.

2.6 Hardware aided induction of decision tree ensembles

Concerning the hardware acceleration of ensemble classifier systems, according to my best knowledge, most of the proposed solutions are related to the hardware implementation of ensemble classifiers that were previously inferred in the software. Most of the proposed solutions are concerned with the hardware acceleration of homogeneous ensemble classifiers *[66][67][68][69][70]*. As far as the author is aware, there is only one proposed solution to the hardware implementation of heterogeneous ensemble classifiers *[71]*. Please notice, that all these solutions are only capable of implementing ensemble classifiers systems that were previously inferred in software, running on some general purpose processor. Author is aware of only one paper *[59]*, that proposes an architecture for the hardware evolution of homogeneous ensemble classifier systems based on the DTs. This solution uses the DT inference algorithm that incrementally creates DTs that are members of the ensemble classifier system.

Regarding the hardware implementation the main concern is the number of required hardware resources, mainly memory, necessary to implement a DT ensemble classifier. Smaller DTs are preferred because they require less hardware resources for the implementation and lead to ensembles with smaller hardware footprint. Therefore, algorithms for DT ensemble classifier induction that generate small, but still accurate, DTs are of great interest when the hardware implementation of DT ensemble classifiers is considered. This requirement puts the full DT induction algorithms and the proposed *EFTI* algorithm into the focus. As discussed earlier, the *EFTI* algorithm provides smaller DTs with similar or better classification accuracy than the other well-known DT inference algorithms, but is also more computationally demanding than the incremental inducers. Hence the *EEFTI* algorithm could merit greatly from the hardware acceleration to shorten the induction times, making it more attractive. In this thesis, the *EEFTIP* co-processor is proposed to accelerate parts of the *EEFTI* that are most computationally intensive, with the remaining parts of the algorithm running on the CPU. The *EEFTIP* co-processor architecture benefits also from the fact that the *EFTI* algorithm evolves the DT using only one individual, in contrast to many other algorithms based on the EA that require populations *[36][40][34][33]*. The architecture can thus be simplified with hardware resources allocated only for a single individual per ensemble member. Furthermore, by using the HW/SW co-design approach, proposed *EEFTIP* co-processor can be used to accelerate DT ensemble inducers based on the Bagging algorithm which rely on a variety of other EA-based DT induction algorithms *[39][36][40][34][33]*. As far as the author is aware, the *EEFTIP* co-processor is the first solution concerned with the hardware acceleration of full DT ensemble induction algorithm based on bagging proposed in the open literature.

2.7 UCI Database Library

For the various experiments presented in the thesis, datasets from the UCI benchmark datasets database were used *[72]*. The UCI database is commonly used in the machine learning community to estimate and compare the performance of different machine learning algorithms. The Table 2.1 lists the UCI datasets (and their characteristics) that were used throughout the experiments in this thesis.

Table 2.1: *List of datasets (and their characteristics) from the UCI database, that are used in the experiments throughout this thesis*

Short Name	Dataset Name	No. of attributes	No. of instances	No. of classes
adult	Adult	14	32561	2
ausc	Australian Credit Approval	14	690	2
bank	Bank Marketing	16	45211	2
bc	Balance Scale	4	625	3
bch	Bach Choral Harmony	16	5665	60
bcw	Breast Cancer Winsconsin	9	699	2
ca	Credit Approval	15	690	2
car	Car Evaluation	6	1728	4
cmc	Contraceptive Method Choice	9	1473	3
ctg	Cardiotocography	21	2126	10
cvf	Clave Vectors Firm-Teacher Model	15	10800	7
eb	Tamilnadu Electricity Board Hourly Readings	4	45781	31
eye	EEG Eye State	14	14980	2
ger	German Credit Data	24	1000	2
gls	Glass Identification Database	9	214	7
hep	Hepatitis	19	155	2
hrtc	Heart Disease Clevelend	13	303	5
hrts	Heart Statlog	13	270	2
ion	Johns Hopkins University Ionosphere	34	351	2
irs	Iris Plants	4	150	3
jvow	Japanese Vowels	14	4274	9
krkopt	Chess (King-Rook vs. King-Pawn)	6	28056	18
letter	Letter Recognition	16	20000	26
liv	BUPA liver disorders	6	345	2
lym	Lymphography	18	148	4
magic	MAGIC Gamma Telescope	10	19020	2
msh	Mushroom	22	8124	2
nurse	Nursery	8	12960	5
page	Page Block Classification	10	5000	5
pen	Pen-Based Recognition of Handwritten Digits	16	10992	10
pid	Pima Indians Diabetes	8	768	2

Continued on next page

Table 2.1 – continued from previous page

Short Name	Dataset Name	No. of attributes	No. of instances	No. of classes
psd	Parkinson Speech	27	1040	2
sb	Seismic Bumps	18	2584	2
seg	Image Segmentation	18	2310	7
shuttle	Shuttle	9	58000	7
sick	Thyroid Disease 2 Class	29	3772	2
son	Sonar (Mines vs. Rocks)	60	208	2
spect	SPECT Heart	22	267	2
spf	Steel Plates Faults	27	1941	7
thy	Thyroid Disease 4 Class	29	3772	4
ttt	Tic-Tac-Toe Endgame	9	958	2
veh	Vehicle Silhouettes	18	846	4
vote	Congressional Voting Records	16	435	2
vow	Vowel Recognition	10	990	11
w21	Waveform Database Generator - 21 Attributes	21	5000	3
w40	Waveform Database Generator - 40 Attributes	40	4090	3
wfr	Wall Following Robot Navigation	24	5000	4
wilt	Wilt	5	4839	2
wine	Wine	11	4898	7
zoo	Zoo	17	101	7

2.8 The structure of the experiments used in the thesis

Similar experimental setup is used throughout this thesis whenever a quality of a certain feature needs to be assessed for an induction algorithm or its specific implementation. Unless stated otherwise, this procedure comprises the induction of the DTs from all datasets listed in the Table 2.1, and measuring the inference times and the qualities of the produced DTs, such as accuracy and size. All the results reported for the experiments in accompanying tables and figures, are the averages of the five 5-fold cross-validations, usually given with their 95% confidence intervals.

The cross-validation setup for assessing the induction algorithm or its implementation is performed for each dataset selected for the experiment in the following way:

- The dataset D, is partitioned into 5 non-overlapping sets: D_1, D_2, ... D_5, by randomly selecting the instances from D using uniform distribution
- For the i^{th} cross-validation run, where $i \in (1, 5)$, training set is formed by using all the instances from D except the ones from D_i, $train\_set = D \setminus D_i$, and is used to induce the DT by the current algorithm being tested
- Inferred DT is finally tested for accuracy by letting it perform the classification on the

instances form the set D_i.

This whole procedure is repeated 5 times, resulting in 25 inferred DTs for each dataset and for each inference algorithm. For each of the DTs, the information about various features is gathered: classification accuracy, DT size, DT depth, inference time, DT fitness, etc., for which the average values and 95% confidence intervals are calculated.

Often, the aim of an experiment used in this thesis is to discover whether there is a statistical difference between the performance of different algorithms, or the same algorithms with different parameters, or the different implementations of the same algorithm. The well known Student's t-test is used in statistics to determine if two sets of data are significantly different from each other. However, in the experiments throughout this thesis, there are usually more than two sets of data compared, hence the t-test cannot be applied. Instead, for each feature tested and dataset used, first the one-way analysis of variance (ANOVA) *[73]* test is applied on collected data, with the significance level set at 0.05. When ANOVA analysis indicates that at least one of the results is statistically different from the others, the Tukey multiple comparisons test *[74]* is used to group the algorithms into groups of statistically identical results. Hence, for each feature of interest and each dataset, a set of groups is obtained, where the algorithms within the group have similar performance for that feature and dataset. Finally, these groups are ranked with respect to their average performance on that feature and dataset, and each tested algorithm is assigned a number, representing the position of its group within the ranking.

Finally, often the average of all ranking numbers for the algorithm for one feature is taken to represent the overall performance of that algorithm on all datasets with respect to that feature. The average rankings are then compared between the algorithms per feature, to determine the benefits of using one over the other.

3 *EFTI* algorithm

This section describes an evolutionary algorithm for oblique full DT induction using supervised learning - *EFTI*. As we have seen in the introduction (Section 2.3), an algorithm that would take advantage of the full DT induction, but limit its resource consumption to make it attractive for the world of embedded systems is lacking in the open literature. The main motivation for creating *EFTI*, was thus to develop an algorithm that:

- is suitable for the implementation on embedded systems, i.e. has low hardware resource requirements,
- is easy parallelizable and accelerated in hardware, and
- uses nonincremental DT induction to induce smaller DTs than the existing solutions, without the loss in DT accuracy.

Since inferring an optimal DT in terms of both size and accuracy is an NP-hard problem, the *EFTI* algorithm needed to be based on some kind of heuristic. In order to minimize the hardware resource consumption of the algorithm implementation, it was chosen to be operated only on a single candidate solution, effectively excluding all the algorithms that operate on populations, such as particle swarm optimization, memetic algorithms, genetic algorithms, and some types of evolutionary algorithms. For all these reasons, it was chosen to base the *EFTI* algorithm on the (1+1) Evolutionary Strategy, since on one hand it operates on a single individual, while on the other it was supposed to be capable of managing the highly complex problem of searching for the small, yet accurate enough DTs, by using the nature inspired evolutionary process. The following topics will be covered in this section:

- Section 3.1 - Overview of the algorithm
- Section 3.2 - Detailed description of the algorithm
- Section 3.3 - The improvements to the basic algorithm version
- Section 3.4 - Analysis of the algorithm's computational complexity
- Section 3.5 - Experimental section that shows the performance of the *EFTI* algorithm in comparison to the performances of the existing DT induction algorithms

3.1 The algorithm overview

The Algorithm 3.1 shows the algorithmic framework for the *EFTI* algorithm, which is similar for all evolutionary algorithms and comprises mutation, fitness evaluation and selection tasks, but lacks the crossover step, since the algorithm does not employ a population of individuals. The DT is induced from the training set - the argument `train_set` received by the `efti()` function as shown in pseudo-code. Since the *EFTI* algorithm performs supervised learning, the training set should consist of the problem instances, together with their known class memberships. The *EFTI* algorithm maintains a single candidate solution, stored in the variable `dt` in the pseudo-code. The evolution is started from a randomly generated (by the `initialize()` function) one-node DT, consisting only of the root node, and the effort is iteratively made to improve on it. In each iteration, the DT is slightly changed by the `mutate()` function, to obtain the mutated individual which is then stored in the `dt_mut` variable. Two types of mutations are employed on the DT individual:

- Every iteration, a node test coefficient in a certain number of randomly selected nodes is changed, and
- Every few iterations, a node is either added or removed from the DT

Algorithm 3.1: *Overview of the* EFTI *algorithm*

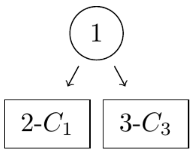
```
def efti(train_set, max_iter):
    dt = initialize(train_set)
    fitness_eval(dt, train_set)

    for iter in range(max_iter):
        dt_mut  = mutate(dt)
        fitness_eval(dt_mut, train_set)

        dt = select(dt, dt_mut)

    return dt
```

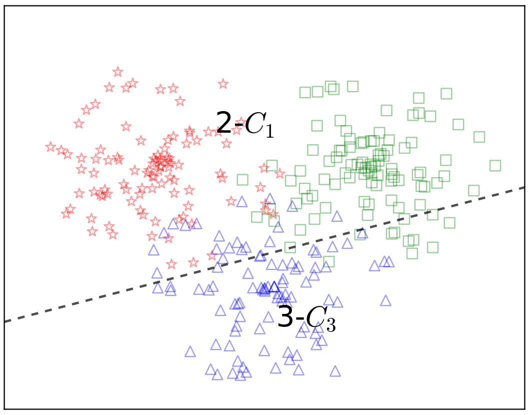
The fitness of the mutated individual, calculated by the `fitness_eval()` function (Algorithm 3.2), is then compared with the fitness of the candidate solution within the `select()` function (Algorithm **??** iterations, the *EFTI* algorithm tries to improve upon the DT candidate solution, after which the algorithm exits and the fittest DT individual found during this process is returned. Once the DT is formed in this way, it can be used to classify problem instances outside of the training set.

In the Figures 3.1 through 3.8, one example evolutionary process performed by the *EFTI* algorithm on the `vene` dataset is shown. The `vene` dataset contains instances of three different classes: C_1, marked by the red stars, C_2, marked by the green squares, and C_3, marked by the blue triangles. Eight specific moments in the DT evolution where significant breakthroughs in the fitness of the DT were made, are presented in these figures by both plotting the tree structure and displaying the partition of the attribute space that the DT individuals at these moments induced. The nodes are drawn in the figures using circles and the leaves using squares, and each node and each leaf is assigned a unique ID. Each leaf node and its corresponding attribute space region are labeled in the format *i-Cj*, where *i* equals the ID of the leaf, and *j* equals the class number assigned to that leaf, hence also to the region. For each of these figures, the following information is given:

- Iteration - the iteration number in which the DT individual was evolved
- Fitness - the fitness of the DT individual
- Size - the size of the DT individual: calculated as the number of leaves in the DT
- Accuracy - the accuracy of the DT individual on the training set: calculated as the percentage of the instances from the training set that the DT individual classifies correctly

At the beginning of the *EFTI* algorithm, the initial individual is generated (Figure 3.1) to contain only one node, since *EFTI* has a goal of creating DTs as small as possible. By the iteration #13 (Figure 3.2), no new nodes were added, but the root node test was modified to

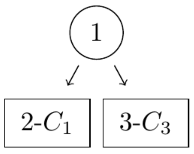


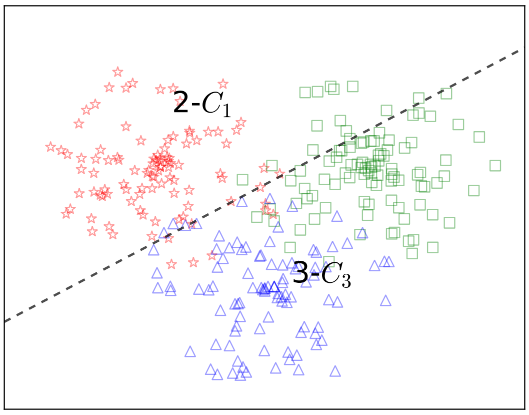


(a) *Initial one-node DT generated by the* `initialize()` *function*

(b) *Initial attribute space partition*

Figure 3.1: *An example evolutionary process by the* EFTI *algorithm. Iteration: 000000, Fitness: 0.6024, Size: 2, Accuracy: 0.6005*

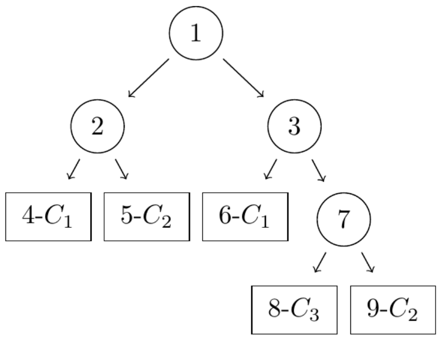




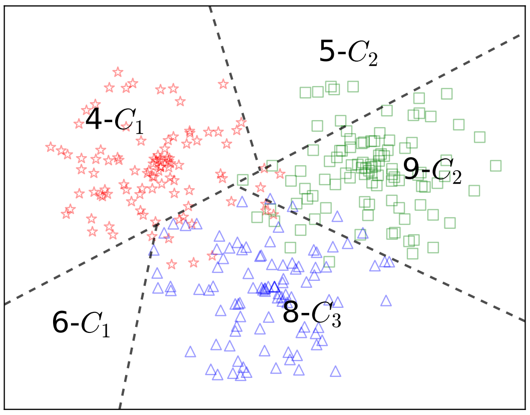
(a) *No added nodes that were tried managed to increase fitness*

(b) *Position of the split shifted to increase the accuracy*

Figure 3.2: *An example evolutionary process by the* EFTI *algorithm. Iteration: 000013, Fitness: 0.6287, Size: 2, Accuracy: 0.6274*

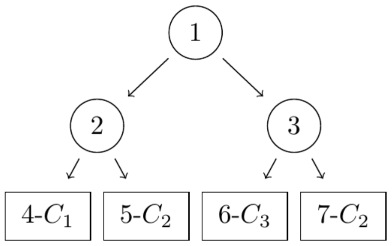


(a) *Three new nodes added to increase the accuracy*

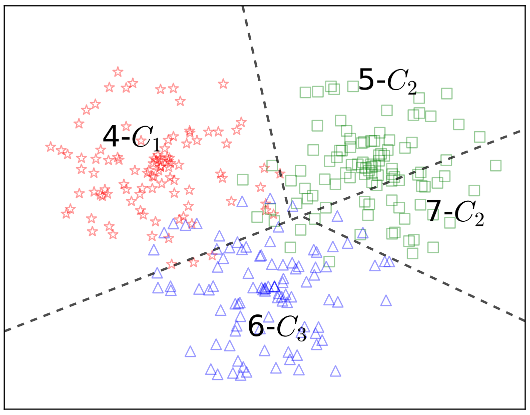


(b) *Three new splits added for finer attribute space partition*

Figure 3.3: *An example evolutionary process by the* EFTI *algorithm. Iteration: 003599, Fitness: 0.9138, Size: 5, Accuracy: 0.9202*

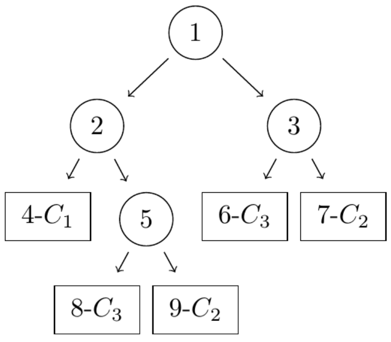


(a) *Since the region of leaf #6 contained almost no individuals in the Figure 3.3b, it was removed and the node #7 was basically moved up to replace node #3 (Figure 3.3a), and thus removing the said empty region.*

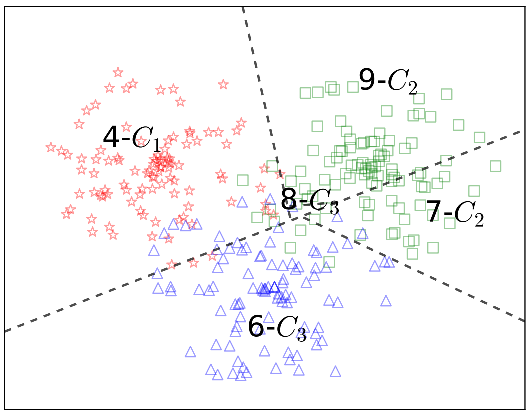


(b) *The region of the leaf #6 (Figure 3.3b) was removed, since it was almost empty and contributed little to accuracy. The resulting DT is smaller, even with a slight increase in accuracy (since the split induced by node 1 has also shifted slightly to a better position).*

Figure 3.4: *An example evolutionary process by the* EFTI *algorithm. Iteration: 007859, Fitness: 0.9265 Size: 4, Accuracy: 0.9297*

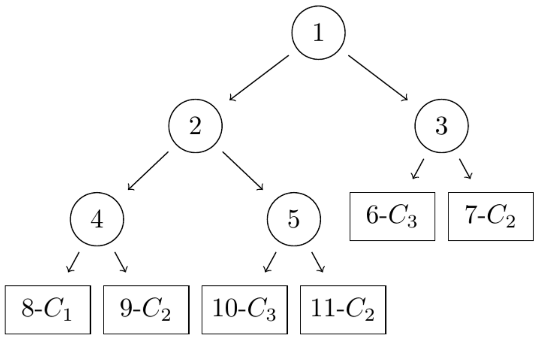


(a) *The leaf #5 was made into a node*

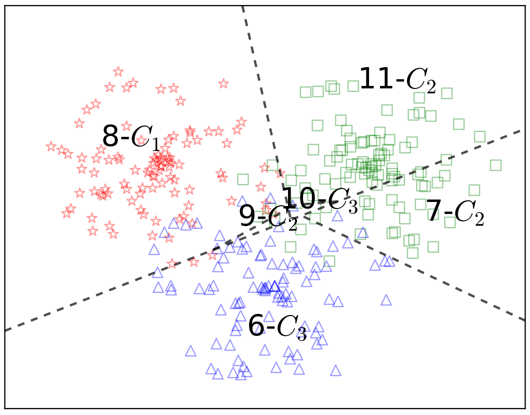


(b) *Small increase in accuracy was obtained by further dividing the central region of the attribute space, where the individuals of all three classes overlap*

Figure 3.5: *An example evolutionary process by the* EFTI *algorithm. Iteration: 030268, Fitness: 0.9272, Size: 5, Accuracy: 0.9331*

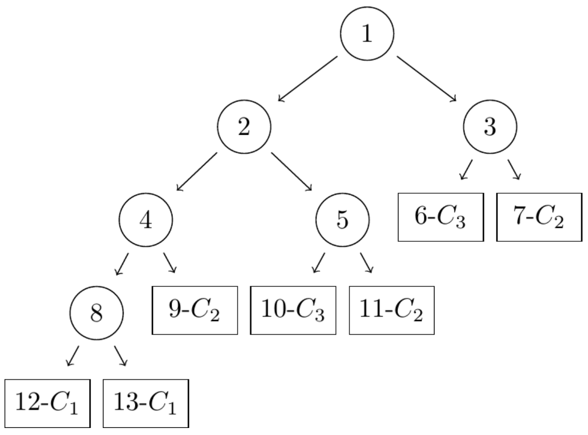


(a) *The leaf #4 was now made into a node*

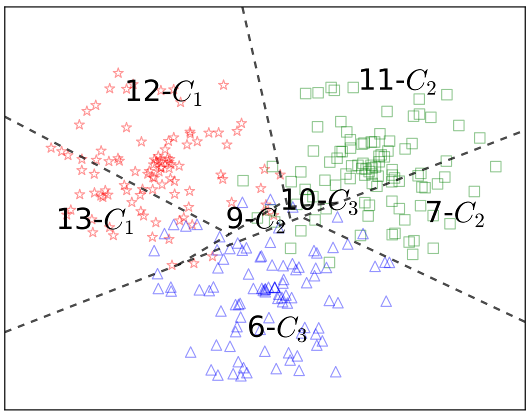


(b) *Again, further division of central attribute space region produced a small increase in accuracy. Fitness has progressed even less, since the addition of a new node diminished the advantage of a small accuracy increase.*

Figure 3.6: *An example evolutionary process by the* EFTI *algorithm. Iteration: 177050, Fitness: 0.9273, Size: 6, Accuracy: 0.9374*

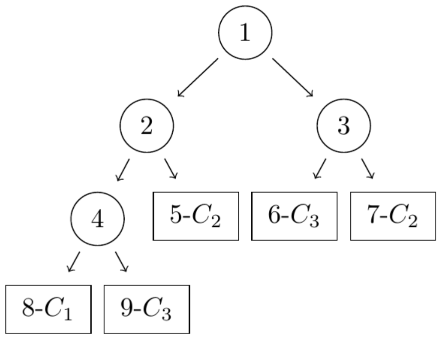


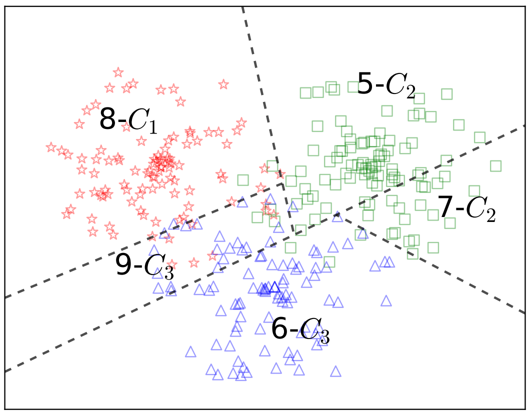
***(a)** The leaf #8 was split into two*



***(b)** The region of leaf #8 was split, bringing no improvement to the class separation, but with some other shifts in the split positions, some small accuracy gain was achieved*

***Figure 3.7:** An example evolutionary process by the* EFTI *algorithm. Iteration: 279512, Fitness: 0.9274, Size: 7, Accuracy: 0.9395*





***(a)** Leaf #9 was removed together with the node #4, which brought the node #8 up in the place of the node #4. Leaves #10 and #11 were removed, and the node #5 was reverted to leaf again.*

(b) EFTI *gave up on finely partitioning the central attribute space region, since very little gain in accuracy could not justify the increase in the DT size, and it managed to produce the smaller DT without sacrificing the accuracy. The split by the node #8 between the regions #12 and #13 in the Figure 3.7, became the split between the regions #8 and #9 after the node #8 moved up to replace the node #4. This, once useless split, has now shifted to turn out very useful in separating instances of the classes C_1 and C_3 and hence contributing to the accuracy.*

***Figure 3.8:** An example evolutionary process by the* EFTI *algorithm. Iteration: 415517, Fitness: 0.9342, Size: 5, Accuracy: 0.9396*

produce the increase in the DT accuracy from 0.6005 to 0.6274. During the further evolution, some nodes were added which raised the accuracy of the DT. Notice how fitness started to deviate from the accuracy when the DT grew bigger. This is because the fitness also depends on the size of the DT to which it applies, in that it is more significantly penalized, the more leaves the DT has. In this example, the biggest drop in the fitness caused by the DT size is in the iteration #279512 of the DT evolution (Figure 3.7), where the DT individual comprised 7 leaves and even though the accuracy climbed to 0.9395 (classification success rate of 94%), the fitness remained at 0.9274. In this way, the evolutionary process was forced to search for the smaller DT solutions, in which it eventually succeeded by the iteration #415517 (Figure 3.8), where the DT size dropped to only 5 leaves with no loss in accuracy.

3.2 Detailed description

In this section, the detailed descriptions of the individual *EFTI* sub-tasks are given. Although *EFTI* is based on the (1+1)-ES, it comprises many additional features which are specific to the DT induction, that need to be discussed here, like tree structure mutation procedure, fitness calculation specifics, etc.

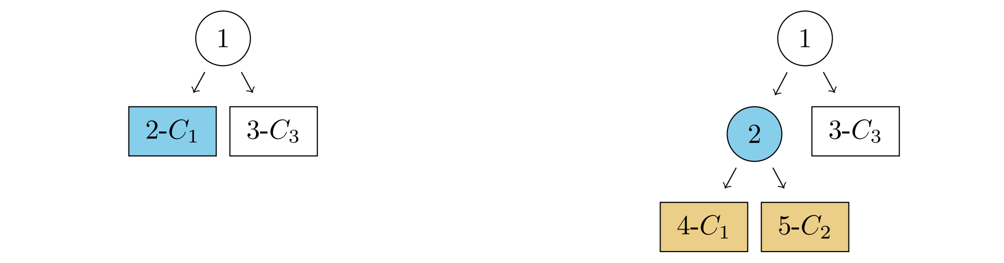
3.2.1 Mutation

For the sake of describing an oblique DT, two different sets of information need to be provided: the coefficient numerical data that describe the oblique tests in the nodes, and the topological data that describes the connections between the nodes. Accordingly, inducing an oblique DT implies inducing the node test coefficients as well as the topological structure. Hence, as it was already discussed in the algorithm overview, the *EFTI* algorithm needs to perform two types of mutations on the DT individual:

- The node test coefficients mutation
- The DT topology mutation

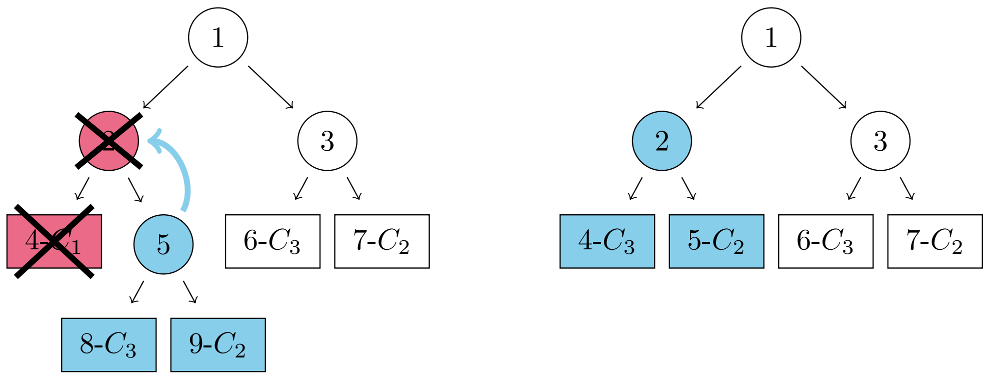
During each iteration of the *EFTI* algorithm, a small number (α) of DT nodes' test coefficients is selected at random and then mutated by adding (or subtracting) to it a small random number. Every change in the node test influences the classification, as the instances take different paths through the DT and get classified in a different way. Finding the optimal oblique split is in itself an NP hard problem (as already discussed in the Section 2.3.1), hence deciding which coefficients should be mutated in order to enhance the DT accuracy is also a hard algorithmic problem. For this reason, the coefficients to be mutated are selected randomly according to the uniform distribution from the set of all coefficients from all DT nodes. Usually, only one to several coefficients (dictated by the parameter α) are mutated in each iteration in order for the classification result to change in small steps. The larger the number of coefficients mutated in each iteration, the more the algorithm starts behaving as a random search.

Once the decision is made which coefficients are to be mutated, the amount by which to change each of the coefficients needs to be specified. Since the algorithm cannot know in advance the optimal order of magnitude of a coefficient value, which would in turn allow it to adjust the size of the coefficient mutation step, the only reference it can take the advantage of is the coefficient's current value. Furthermore, as it will be discussed in the Section 3.2.2, the



(a) *DT before the addition of the node in place of the leaf #2* ***(b)*** *DT after a node has been added in place of the leaf #2*

Figure 3.9: *Example showing how a DT is mutated by adding a node to it*



(a) *DT before the removal of the leaf #4, together with its parent node #2* ***(b)*** *DT after the leaf #4 and its parent node #2 were removed, and the sub-tree induced by former node #5 moved to the position of node #2*

Figure 3.10: *Example showing how a DT is mutated by removing a node from it*

node test coefficients are not initialized completely at random, but are calculated according to an algorithm to provide an improvement to the overall accuracy of the DT, hence their initial values provide a useful starting reference point in searching for their optimal values. Due to all this, the *EFTI* algorithm selects the mutation step for the coefficients according to the normal distribution centered at zero, with the standard deviation proportional to the value of the coefficient to be mutated. However, for the coefficients with small values, the deviation would be likewise low, and it would be hard to escape this situation via process of mutation. Similarly, for the coefficients with large values, the deviation would be likewise high, and these coefficients would be changed in too large increments. Hence, the *EFTI* algorithm saturates the deviation for both small and large coefficient values at σ_{min} and σ_{max} respectively. The saturation points σ_{min} and σ_{max} are fixed throughout the algorithm operation and selected by the user. The random variable representing the mutation step for the coefficient w_i, named X_{mwi} is finally given by the equation:

$$X_{mwi} \sim \mathcal{N}(0, \sigma^2)|\sigma = \begin{cases} \sigma_{min}, & w_i \leq \sigma_{min} \\ w_i, & \sigma_{min} < w_i < \sigma_{max} \\ \sigma_{max}, & \sigma_{max} \leq w_i \end{cases} \tag{3}$$

This means that the mutated value w_i^m for the selected coefficient w_i is obtained as $w_i^m = w_i + X_{mwi}$.

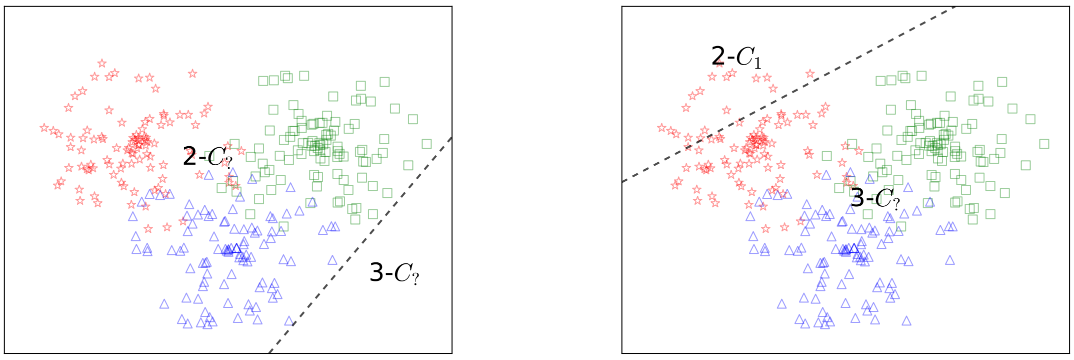
On the other hand, the topology mutations represent very large moves in the search space, so they are performed even less often. In every iteration, there is a chance (β) that a single node will either be added to the DT or removed from it. This change either adds an additional test to the classification process, or removes one from it. The node is always added in place of an existing leaf, i.e. never in place of an internal non-leaf node, as shown in the example in the Figure 3.9. The leaf which is to be turned into a node is selected at random uniformly from all the leaves in the DT. The test coefficients of the newly added non-leaf node are calculated using the same initialization procedure as for the root test coefficients, which is explained in the Section 3.2.2. On the other hand, if a node is to be removed, first a leaf is selected at random uniformly from all the leaves in the DT. Then both the leaf and its parent are removed from the DT, while the leaf's sibling moves up to replace its former parent, as shown in the example in the Figure 3.10. By adding a test, a new point is created where during the classification, instances from different classes might separate and take different paths through the DT and eventually be classified as different, which can in turn increase the accuracy of the DT. On the other hand, by removing unnecessary tests the DT is made smaller, and the size of the DT is also an important factor in the fitness calculation in the *EFTI* algorithm as discussed in the Section 3.2.3.2.

There is a known result regarding (1+1)-ES algorithms called 1/5 success rule *[75]*, stating that the mutation step size should be adapted dynamically in order to keep the mutation success rate close to one-fifth, meaning that approximately every fifth mutation should lead to an individual with higher fitness. To accomplish this, the mutation step is dynamically adapted try to control the success rate. There are at least two problems with adopting the 1/5 strategy here: first there are two different types of mutations (coefficient and topological) with each one having its own mutation rate, and second the success rates were measured to be closer to around 1% when the *EFTI* algorithm was run on practical datasets. Although the effort was made in an attempt to devise a dynamic adaptation strategy akin to the 1/5 success rule that would provide statistically significant benefits to the *EFTI* algorithm, it was futile.

3.2.2 The DT node insertion algorithm

Each time a node is to be added to the DT, whether it is the root node for the DT initialization or any other node in the mutation procedure, the node's test needs to be initialized. Initializing the test coefficients with random numbers proved to be an impediment to the evolutionary process, since there is a rather small probability for a node test generated in this way to provide a useful split in the attribute space, i.e. a split that divides instances of different classes. With this, completely random, procedure, the hyperplane usually lands completely outside the attribute space region where the instances are located, where the Figure 3.11a shows one such hyperplane as an example. Even if the hyperplane intersects the area of the attribute space where the instances reside, the split can still be ineffective in the way that it does not help distinguish between instances of different classes, i.e. it does not contribute to the DT accuracy, where the Figure 3.11b shows one such hyperplane as an example. This influences the algorithm convergence negatively, in that it takes too many generations to relocate the ill-positioned hyperplane to the location where it starts contributing to the accuracy of the DT individual.

However, in order to allow for wider search space exploration, the node tests need to be generated at random, but this process needs to be guided by the structure of the training set,



***(a)** Hyperplane initialized to the position outside the region where the instances reside* ***(b)** Hyperplane initialized to the position where it does not contribute to the DT accuracy*

***Figure 3.11:** Hyperplanes cannot be initialized completely at random, since there is a high chance of them being ineffective*

to speed up the convergence of the evolutionary algorithm towards the optimal solution. One of the approaches for the random initialization basically ensures that two randomly selected training set instances (called a mixed dipole) take different paths during classification at the node being initialized, and is suggested in *[40]*. The mixed dipole comprises two instances from the training set that belong to different classes. As shown in the Figure 3.12, the procedure consists of placing the hyperplane $H_{ij}(\mathbf{w}, \theta)$ in the attribute space, perpendicular to the line connecting the mixed dipole $(\mathbf{x}^i, \mathbf{x}^j)$. The hyperplane corresponds to the node test given by the equation (1), where $\mathbf{w}$ is the test coefficient vector and θ is the test threshold. The attribute space of the `vene` dataset, used in this example has two dimensions, one for each of the attributes x_1 and x_2. The hyperplane's exact position is finally fixed by randomly generated parameter $\delta \in (0, 1)$, which determines whether the hyperplane is placed closer to $\mathbf{x}^i$ (for $\delta < 0.5$), or closer to $\mathbf{x}^j$ (for $\delta > 0.5$). Mathematically, the equation for the hyperplane generated by the method of the mixed dipole described in this paragraph is obtained in the following way:

$$\begin{gathered} H_{ij}(\mathbf{w}, \theta) = \mathbf{w} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - \theta, \\ \mathbf{w} = (\mathbf{x}^i - \mathbf{x}^j), \\ \theta = \delta \mathbf{w} \cdot \mathbf{x}^i + (1 - \delta) \mathbf{w} \cdot \mathbf{x}^j \end{gathered} \tag{4}$$

This procedure aims to introduce a useful test into the DT, based on the assumption that the instances of the same class are somehow grouped in the attribute space, and that the test produced in this way will help separate the instances belonging to the classes of the dipole instances.

3.2.3 Fitness evaluation

The DT can be optimized with respect to various parameters, where the DT accuracy and its size are usually the most important. Hence, in order to solve this multi-objective optimizational problem with the evolutionary approach, a fitness function needs to be defined to effectively

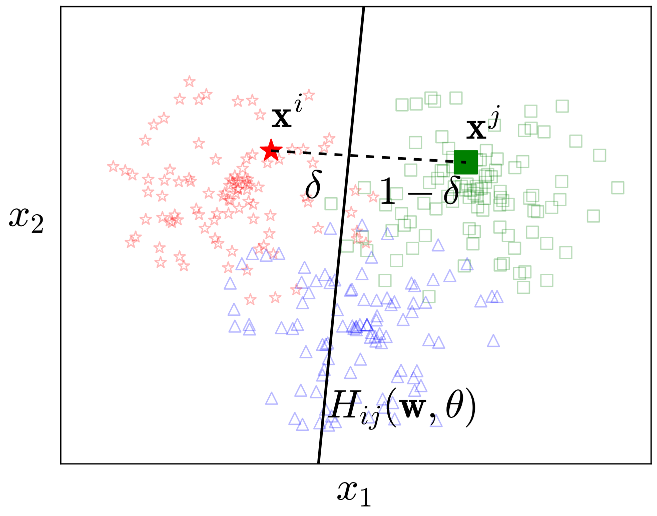


Figure 3.12: *Initialization of the node test based on the randomly chosen dipole. $H_{ij}(\mathbf{w}, \theta)$ is a hyperplane corresponding to the node test, $\mathbf{w}$ is coefficient vector, and θ is the threshold.*

collapse it to a single objective optimizational problem. This can be done in various ways, and here one procedure, employed by the *EFTI* algorithm is given.

Algorithm 3.2: *The pseudo-code of the fitness evaluation task, given by* `fitness_eval()` *function.*

```
def fitness_eval(dt, train_set):

    accuracy = accuracy_calc(dt, train_set)

    Nc = train_set.cls_num()
    oversize = (len(dt.leaves()) - Nc)/Nc

    dt.fit = accuracy*(1 - Ko*oversize*oversize)
```

3.2.3.1 Accuracy calculation

The main task of the optimization process performed by *EFTI* is to maximize the accuracy of the DT individual on the training set. The accuracy is calculated by letting the DT individual classify all problem instances from the training set and then by comparing the classification results to the desired classifications, specified in the training set. The pseudo-code for this task is given in the Algorithm 3.3 by the function `accuracy_calc()`, where the input parameter `dt` receives the DT individual whose accuracy is to be calculated, and `train_set` expects the training set.

Algorithm 3.3: *The pseudo-code of the accuracy calculation task, given by* `accuracy_calc()` *function.*

```
def accuracy_calc(dt, train_set):

    distribution = [[0] * train_set.cls_num()
                    for i in range(len(dt.leaves()))]

    for instance in train_set:
        leaf = find_dt_leaf_for_inst(dt, instance)
        distribution[leaf.id][instance.cls] += 1

    hits = 0
    for leaf in dt.leaves():
        dominant_class_cnt = max(distribution[leaf.id])
        hits += dominant_class_cnt

    return hits / len(train_set)
```

First, the class distribution is determined by letting all instances from the training set traverse the DT, within the `find_dt_leaf_for_inst()` function whose pseudo-code is given in the Algorithm 3.4. This function determines the instance traversal path, and returns the leaf node in which the instance finished the traversal. The traversal starts at the root node (accessed via `dt.root`), and is performed in the manner depicted in the Figure 2.2, where one possible path is given by the red curvy line. Until a leaf is reached, the node tests are evaluated and the decisions to which child to proceed, are made based on the test outcomes. The function `dot_product()`, calculates the scalar product of the node test coefficient vector $\mathbf{w}$ (stored in `cur_node.w` attribute), and the attribute vector of the instance $\mathbf{x}$ (stored in `instance.x` variable). The value returned, is compared with the node test threshold θ (stored in `cur_node.thr` attribute).

Algorithm 3.4: *The pseudo-code of the procedure for determining the end-leaf for an instance, implemented by* `find_dt_leaf_for_inst()` *function.*

```
def find_dt_leaf_for_inst(dt, instance):

    cur_node = dt.root

    while (not cur_node.is_leaf):
        psum = dot_product(instance.x, cur_node.w)

        if psum < cur_node.thr:
            cur_node = cur_node.left
        else:
            cur_node = cur_node.right

    return cur_node
```

Next step in the accuracy calculation process (the first for loop in the Algorithm 3.3) is to calculate the class distribution matrix. The distribution matrix, shown in the Figure 3.13, has one row for each of the leaves in the DT, i.e. for each attribute space partition induced by the DT. Each row in turn contains one element for each of the classes in the training set. Hence, a row of the distribution matrix contains the statistics on how many instances of each of the training set classes finished the traversal in the leaf corresponding to the row.

$$\begin{array}{r|c|c|c|c|l}
 & C_1 & C_2 & & C_{N_c} & \\
\hline
\text{Leaf ID } 1 & d_{1,1} & d_{1,2} & \cdots & d_{1,N_c} & \Rightarrow k_1, d_{(1,k_1)} \\
\hline
\text{Leaf ID } 2 & d_{2,1} & d_{2,2} & \cdots & d_{2,N_c} & \Rightarrow k_2, d_{(2,k_2)} \\
\hline
 & \vdots & \vdots & & \vdots & \\
\hline
\text{Leaf ID } N_l & d_{N_l,1} & d_{N_l,2} & \cdots & d_{N_l,N_c} & \Rightarrow k_{N_l}, d_{(N_l,k_{N_l})} \\
\hline
\end{array}$$

Figure 3.13: *The structure of the distribution matrix. From each matrix row* i*, the dominant class k_i and the number of instances of the dominant class $d_{(i,k_i)}$ that finished the traversal in the leaf with ID* i *are obtained.*

The classes of all the instances from the training set are known and accessed via the instance attribute `instance.cls` (within the `accuracy_calc()` function). For each instance in the training set, based on its class and the ID of the leaf in which it finished the traversal, the distribution matrix is updated. This leaf is obtained via the `find_dt_leaf_for_inst()` function and stored into the `leaf` variable, and its ID is accessed via the attribute `leaf.id`. The $d_{i,j}$ element of the distribution matrix contains the number of instances of the class j (C_j) that finished in the leaf node with the ID i after the DT traversal. After all the instances from the training set traverse the DT, this matrix contains the distribution of classes among the leaf nodes.

The second **`for`** loop of the `accuracy_calc()` function finds the dominant class for each leaf node. The dominant class for a leaf node is the class having the largest percentage of instances, among the ones that finished the traversal in that leaf node. Formally, the dominant class k_i of the leaf node with the ID i is:

$$k_i | (d_{(i,k_i)} = \max_j(d_{i,j})) \tag{5}$$

The structure of the distribution matrix is displayed in the Figure 3.13. Rows correspond to the leaves of the DT, and the columns correspond to the classes of the training set. From each row (i) of the distribution matrix, we obtain the dominant class k_i and the number of instances of the dominant class $d_{(i,k_i)}$ that finished the traversal in the leaf with ID i.

If we were to do a classification run with the current DT individual of the training set, the maximum accuracy would be attained if all leaf nodes were assigned their corresponding dominant classes. Thus, each instance which finishes in a certain leaf node, that belongs to that node's dominant class, is added to the number of classification hits (the `hits` variable of the Algorithm 3.3), otherwise it is qualified as a missclassification. Hence,

$$\texttt{hits} = \sum_{i=1}^{N_l} d_{(i,k_i)}. \tag{6}$$

The accuracy of the DT is, hence, equal to the percentage of the instances whose classifications were declared as hits, as given in the pseudo-code: `accuracy = hits/len(train_set)`.

3.2.3.2 Oversize

The DT oversize is calculated as the relative difference between the number of leaves in the DT and the total number of classes (N_C) in the training set (obtained via the `train_set.cls_cnt()` function). In order to be able to classify correctly all training set instances, the DT needs to have at least one leaf for each class occurring in the training set. Therefore, without knowing anything else about the dataset, our best guess is that the minimal DT that could be consistent with the dataset has one leaf for each of the dataset classes. For that reason, the oversize measure given by the equation (7), was defined in such a way to have the DT start suffering penalties to the fitness when the number of its leaves exceeds the total number of classes in the training set, i.e. the oversize measure is zero when $N_l = N_c$:

$$\begin{gathered} \texttt{oversize} = \frac{N_l - N_c}{N_c}, \\ \texttt{fit} = \texttt{accuracy} \cdot (1 - K_o \cdot \texttt{oversize}^2) \end{gathered} \tag{7}$$

The DT oversize negatively influences the fitness, as it can be seen from the equation (7). The parameter K_o is used to control how much influence the DT oversize will have on the overall fitness. In other words, it determines the shape of the collection of Pareto frontiers for the DT individual. Each DT individual can be represented as a point in a 2-D space induced by the DT oversize and accuracy measures. In a Pareto set all elements have the same fitness value, even though they have different accuracy and oversize measures.

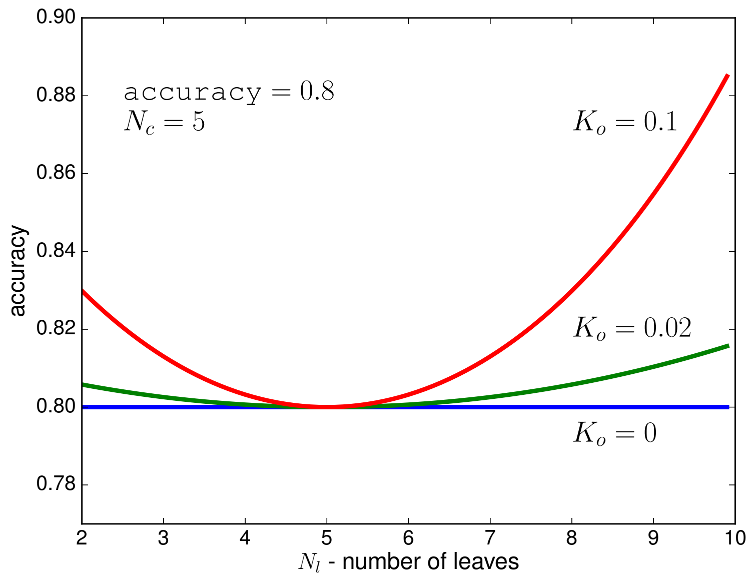


Figure 3.14: *The layout of Pareto frontiers for the accuracy value of 0.8, when N_C equals 5, for K_o parameter values of: 0, 0.02 and 0.1.*

The Figure 3.14 shows the layout of the Pareto frontier for an example of fitness value of 0.8 and few different values of the parameter K_o, with the value of 5 selected for the parameter N_C. It can be seen that if K_o is chosen to be 0, the oversize does not influence the fitness, which is in turn always equal to the value of the accuracy. When $K_o > 0$, the *EFTI* algorithm will be willing to trade accuracy for the DT size. As it can be seen from the figure, when the parameter K_o has a large value, for an example 0.1, the big DTs are highly discouraged in that an individual of size 5 with the accuracy of 0.8 is equally fit in the eyes of the algorithm as the larger one with more than 10% higher accuracy, but of size 10.

As shown in the Algorithm 3.2, the dependence of the fitness on the oversize measure is quadratic. This serves two purposes:

1. Since oversize turns negative when the DT size falls below N_C, such undersized DTs would be getting a boost in fitness if it were not for the squaring. If all classes are to be represented in the DT, the number of leaves should at least match the number of classes, so that it would be at least possible, for each class to have a leaf. By squaring the oversize, the undersized DTs are discouraged in the same way the oversized are.

2. By using the quadratic dependence, the rate at which fitness decreases with the DT size is lower when the size is closer to the N_C, and gets progressively higher as the size increases. This way, the DTs whose size is close to N_C are penalized less then they would be if the dependence of the fitness on oversize were linear.

In order to measure the influence of the oversize on the induced DTs, an experiment has been conducted on all datasets from the Table 2.1. The DTs were induced for a number of values for the parameter K_o, namely $K_o \in \{0, 0.001, 0.01, 0.02, 0.06, 0.1, 0.2\}$. The results are presented in the Table 3.1, Table 3.2, Figure 3.15 and Figure 3.16. The Table 3.1 and the Table 3.2 list the induced DT sizes and accuracies respectively, for all datasets and all values of the oversized weight parameter K_o used in the experiment. In the figures, the plots are organized in pairs, where each pair consists of the accuracy and size plots for the same five algorithms displayed in juxtaposition. Please notice that the x-axis, corresponding to the value of the parameter K_o, is given in logarithmic scale, as well as the y-axis of the DT size plots. Please also notice that the ranges for the y-axis, be it for the accuracy or the size plots, vary from plot to plot and depend on the datasets used for the induction.

The values in the Table 3.1 clearly indicate that the largest DTs are induced when the DT oversize is ignored during the induction, $K_o = 0$. From there, the induced DT sizes drop quickly when the value of K_o is increased, only to start saturating after certain K_o value, which is different for each dataset. This is usually the place where the *EFTI* algorithm needs to start inflicting serious damage to the DT accuracies, only to compress the DTs furher in size by small factors. This trend can be also observed with accuracies in the Table 3.2. The accuracies are, naturaly, largest when there is no size limit imposed, i.e. $K_o = 0$. Then, as the value of K_o increases, the induced DTs of some of the datasets experience a significant drop in the accuracy, where this drop is of course traded-off against a significant drop in their sizes. These datasets, like `bch`, `cmc`, `krkopt`, `letter`, `ttt`, `wfr`, `wine`, etc., are the ones whose internal complexity really demands for bigger DTs in order to describe them more precisely. On the other hand, the induced DTs of some of the datasets, experience little or no change in the accuracy when the K_o value increases up to a certain point. For these datasets, like `ausc`, `bank`, `bcw`, `irs`, `psd`, `shuttle`, `sick`, `zoo`, etc., initial large DTs are indeed excessive in size and the more succinct DT representation was successfully found by the *EFTI* aglorithm. When the *EFTI* algorithm is used in practice, it is a design choice whether the most accurate DTs are needed no matter their size, or we are interested in the smallest DTs at the cost of their accuracy, or we are willing to accept certain trade-off between the DT size and its accuracy. It is obvious from these results that there is a different behavior of the inferred DTs from different datasets, in terms of DT accuracies and sizes, when the oversize fitness weight K_o is varied. Hence, the actual value of the K_o parameter will depend on the domain of the problem being solved.

Table 3.1: *The average sizes of the DTs induced for various values of the parameter K_o*

Dataset	0	0.001	0.01	0.02	0.06	0.1	0.2
adult	273.88	5.72	3.04	2.80	2.12	2.00	2.00
ausc	31.88	9.52	3.76	3.00	2.48	2.04	2.04
bank	172.08	3.44	2.12	2.00	2.00	2.00	2.00
bc	40.76	14.84	6.88	5.08	3.92	3.56	3.16
bch	877.36	814.88	283.84	192.60	133.56	109.88	92.92
bcw	9.72	5.16	2.88	2.04	2.00	2.00	2.00
ca	32.64	9.60	4.00	3.12	2.60	2.12	2.04
car	119.04	23.60	9.84	7.60	5.80	5.04	4.96
cmc	179.76	22.68	8.28	6.48	4.72	4.28	4.00
ctg	262.04	75.12	26.92	21.84	15.80	13.96	12.64
cvf	558.40	33.76	13.04	10.32	8.60	7.92	7.60
eb	1419.56	264.12	113.60	85.24	59.00	51.36	45.00
eye	131.64	8.76	4.20	3.36	2.84	2.08	2.00
ger	45.28	7.92	3.44	3.16	2.76	2.52	2.08
gls	44.00	32.92	22.24	16.12	12.32	10.68	9.60
hep	14.96	10.52	4.72	4.00	3.00	2.76	2.04
hrtc	68.12	53.84	17.12	12.44	8.20	7.52	6.36
hrts	24.16	11.76	4.64	3.84	3.00	2.68	2.00
ion	31.32	10.96	5.36	4.16	3.08	3.04	3.00
irs	8.56	4.76	5.28	4.24	3.88	3.28	3.12
jvow	519.64	83.68	32.72	25.56	17.88	15.72	13.20
krkopt	1973.12	170.96	63.84	48.36	32.96	28.96	24.72
letter	1445.68	254.44	105.88	80.84	55.32	48.40	39.80
liv	46.32	15.08	6.08	4.36	3.16	3.00	2.76
lym	21.68	15.64	11.00	8.04	5.92	5.28	5.00
magic	197.16	6.20	3.44	3.00	2.84	2.24	2.04
msh	49.00	8.88	4.72	3.96	2.92	3.00	2.32
nurse	451.76	23.60	10.04	8.60	6.88	6.32	6.08
page	53.84	12.60	7.44	6.24	5.72	5.20	5.00
pen	355.12	52.12	25.72	21.44	16.76	14.88	13.60
pid	62.48	11.48	4.72	3.48	2.84	2.28	2.08
psd	31.00	6.68	3.56	2.92	2.48	2.20	2.04
sb	17.32	3.60	2.16	2.00	2.00	2.00	2.00
seg	120.56	35.48	17.32	14.48	11.28	10.08	9.20
shuttle	62.84	11.68	8.48	7.76	7.12	7.08	7.04
sick	31.92	3.68	2.48	2.40	2.16	2.04	2.00
son	32.04	14.72	6.48	5.12	3.44	3.00	2.96
spect	19.32	11.84	4.16	3.24	2.68	2.08	2.00
spf	233.84	49.52	20.44	15.96	11.28	10.12	9.00
thy	34.80	8.72	5.00	4.48	4.00	4.00	4.00
ttt	104.00	13.96	5.32	4.12	3.00	2.96	2.20
veh	149.28	34.32	13.76	10.48	7.64	6.88	5.96
vene	15.84	9.84	5.44	4.60	3.96	3.92	3.80
vote	23.32	8.40	4.08	3.08	2.64	2.04	2.04
vow	214.12	100.16	48.00	36.64	25.08	21.64	17.96
w21	178.08	12.20	5.84	5.04	4.00	4.00	4.00
w40	227.12	15.36	6.36	5.44	4.24	4.00	3.96
wfr	350.12	28.48	11.76	9.48	6.92	5.88	5.00
wilt	8.60	3.04	2.00	2.00	2.00	2.00	2.00
wine	340.12	43.76	16.04	12.52	9.40	8.68	8.00
zoo	15.64	9.64	9.88	9.16	9.40	8.76	8.72

Table 3.2: *The average accuracies of the DTs induced for various values of the parameter K_o*

Dataset	0	0.001	0.01	0.02	0.06	0.1	0.2
adult	0.85	0.83	0.82	0.82	0.82	0.83	0.82

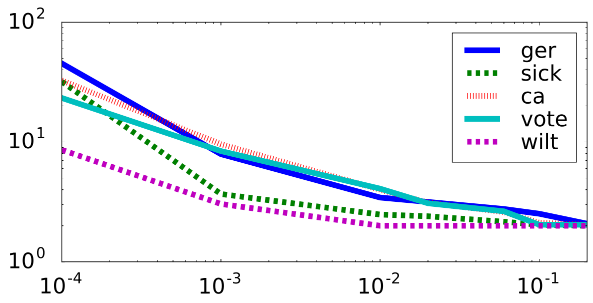
Continued on next page

Table 3.2 – continued from previous page

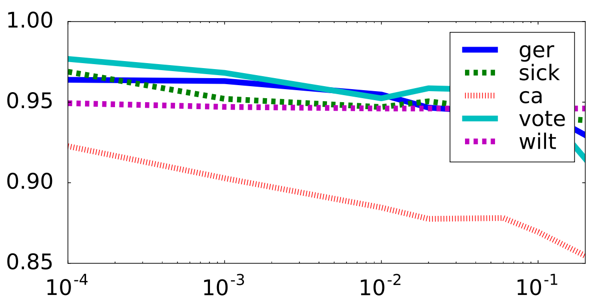
Dataset	0	0.001	0.01	0.02	0.06	0.1	0.2
ausc	0.92	0.90	0.88	0.88	0.88	0.88	0.86
bank	0.89	0.89	0.89	0.89	0.88	0.88	0.88
bc	0.97	0.95	0.93	0.92	0.91	0.92	0.89
bch	0.35	0.36	0.27	0.24	0.22	0.21	0.20
bcw	0.98	0.98	0.98	0.98	0.98	0.98	0.97
ca	0.92	0.90	0.88	0.88	0.88	0.87	0.85
car	0.93	0.91	0.86	0.85	0.84	0.82	0.82
cmc	0.70	0.61	0.57	0.57	0.55	0.55	0.51
ctg	0.87	0.83	0.79	0.77	0.75	0.74	0.74
cvf	0.81	0.78	0.76	0.76	0.76	0.76	0.75
eb	0.62	0.59	0.54	0.54	0.53	0.53	0.52
eye	0.67	0.62	0.60	0.60	0.59	0.58	0.57
ger	0.96	0.96	0.95	0.95	0.94	0.95	0.93
gls	0.89	0.89	0.84	0.82	0.80	0.78	0.77
hep	0.93	0.93	0.90	0.90	0.89	0.89	0.86
hrtc	0.86	0.85	0.75	0.72	0.70	0.69	0.68
hrts	0.92	0.91	0.88	0.88	0.87	0.87	0.84
ion	0.96	0.95	0.92	0.91	0.90	0.90	0.87
irs	0.98	0.98	0.98	0.98	0.98	0.98	0.97
jvow	0.90	0.81	0.73	0.70	0.68	0.67	0.63
krkopt	0.58	0.47	0.40	0.39	0.37	0.36	0.35
letter	0.74	0.66	0.57	0.55	0.52	0.50	0.50
liv	0.83	0.79	0.73	0.73	0.71	0.72	0.68
lym	0.95	0.95	0.93	0.92	0.90	0.89	0.86
magic	0.85	0.83	0.81	0.82	0.82	0.80	0.78
msh	1.00	0.99	0.97	0.97	0.95	0.95	0.92
nurse	0.92	0.90	0.89	0.88	0.86	0.86	0.83
page	0.97	0.96	0.96	0.95	0.95	0.95	0.94
pen	0.98	0.96	0.93	0.92	0.90	0.89	0.87
pid	0.87	0.82	0.79	0.79	0.78	0.78	0.76
psd	0.98	0.98	0.97	0.97	0.97	0.97	0.94
sb	0.94	0.94	0.93	0.93	0.93	0.93	0.93
seg	0.97	0.96	0.93	0.92	0.90	0.90	0.86
shuttle	1.00	1.00	1.00	0.99	0.99	0.99	0.99
sick	0.97	0.95	0.95	0.95	0.94	0.94	0.94
son	0.94	0.91	0.86	0.84	0.80	0.81	0.78
spect	0.91	0.90	0.87	0.87	0.87	0.87	0.85
spf	0.82	0.74	0.69	0.68	0.66	0.65	0.64
thy	0.96	0.95	0.95	0.96	0.95	0.95	0.94
ttt	0.87	0.79	0.74	0.74	0.72	0.72	0.70
veh	0.85	0.75	0.68	0.65	0.63	0.62	0.59
vene	0.95	0.95	0.94	0.93	0.93	0.93	0.91
vote	0.98	0.97	0.95	0.96	0.96	0.95	0.91
vow	0.93	0.88	0.76	0.71	0.64	0.61	0.57
w21	0.90	0.87	0.85	0.85	0.84	0.84	0.80
w40	0.90	0.84	0.82	0.81	0.80	0.80	0.75
wfr	0.88	0.80	0.75	0.73	0.70	0.69	0.68
wilt	0.95	0.95	0.95	0.95	0.95	0.95	0.95
wine	0.67	0.60	0.57	0.56	0.55	0.55	0.55
zoo	0.98	0.98	0.98	0.98	0.98	0.98	0.97

3.2.4 Selection

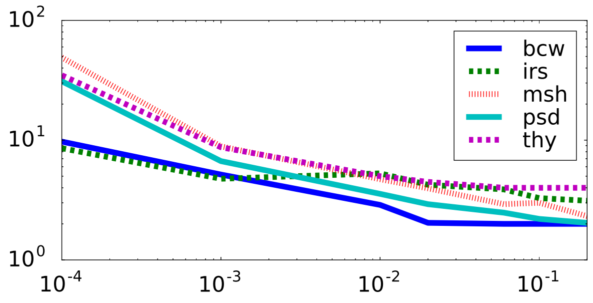
The selection task is responsible for deciding, in each iteration, which DT will be taken for the candidate solution for the next iteration: either the current candidate solution, i.e. the parent (in the evolutionary sense), or the mutated individual. The selection procedure implemented by the Algorithm 3.5 is the most basic one, where whenever the mutated individual outperforms



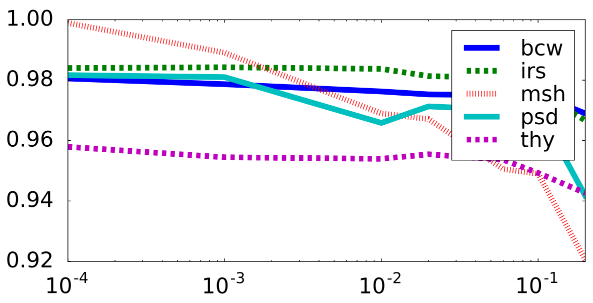
***(a)** DT size: ger, sick, ca, vote, wilt*



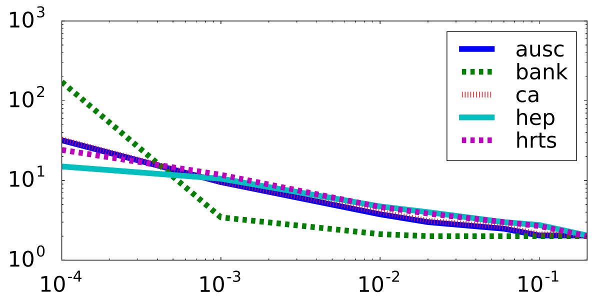
***(b)** DT accuracy: ger, sick, ca, vote, wilt*



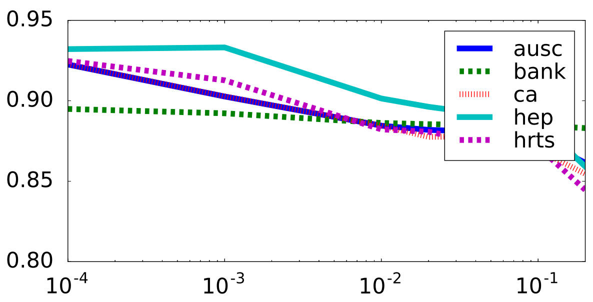
***(c)** DT size: bcw, irs, msh, psd, thy*



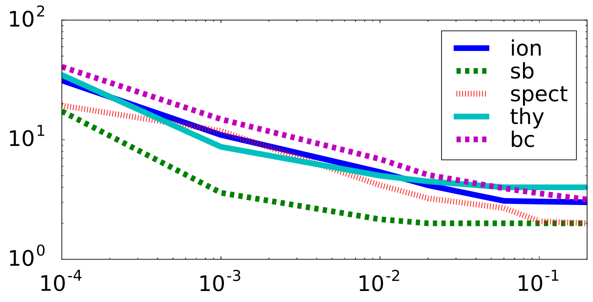
***(d)** DT accuracy: bcw, irs, msh, psd, thy*



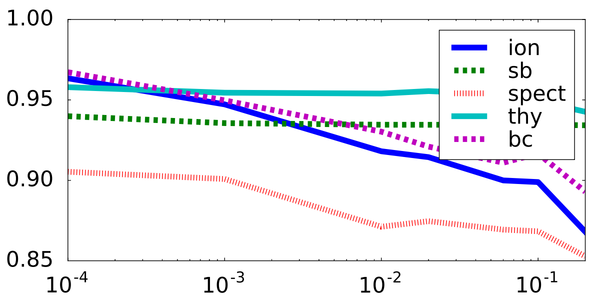
***(e)** DT size: ausc, bank, ca, hep, hrts*



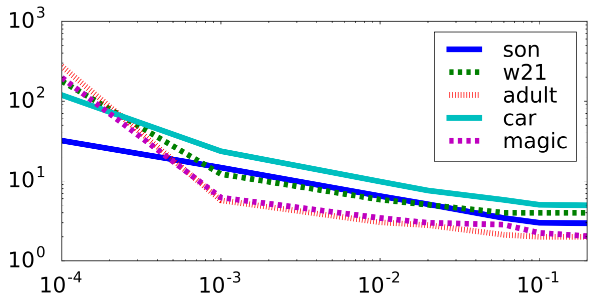
***(f)** DT accuracy: ausc, bank, ca, hep, hrts*



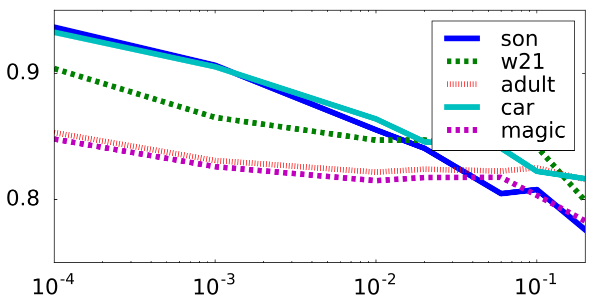
***(g)** DT size: ion, sb, spect, thy, bc*



***(h)** DT accuracy: ion, sb, spect, thy, bc*

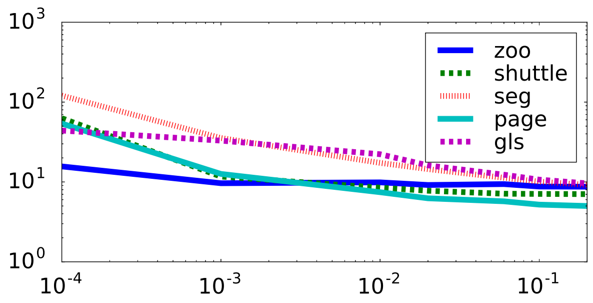


***(i)** DT size: son, w21, adult, car, magic*

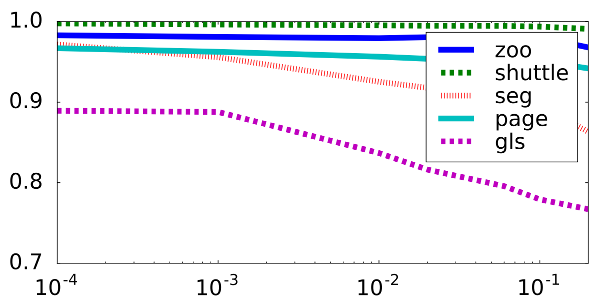


***(j)** DT accuracy: son, w21, adult, car, magic*

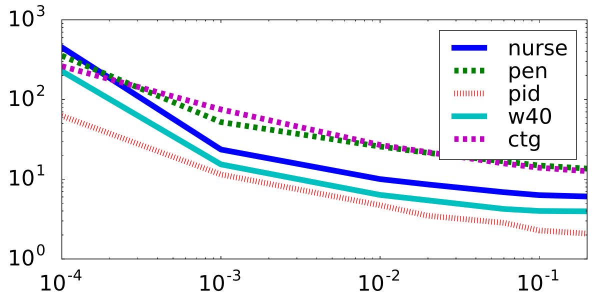
***Figure 3.15:** Dependencies of the induced DT sizes and accuracies on the oversize weight (K_o) parameter values. Datasets 1-25.*



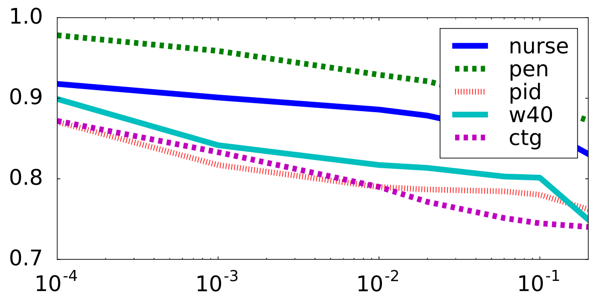
***(a)** DT size: zoo, shuttle, seg, page, gls*



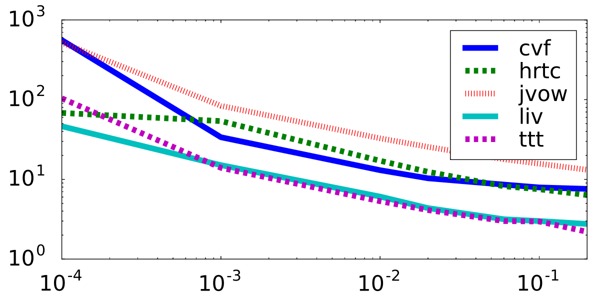
***(b)** DT accuracy: zoo, shuttle, seg, page, gls*



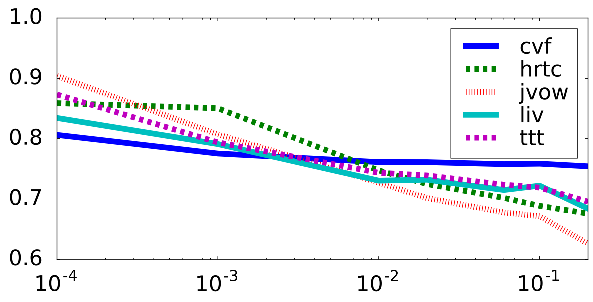
***(c)** DT size: nurse, pen, pid, w40, ctg*



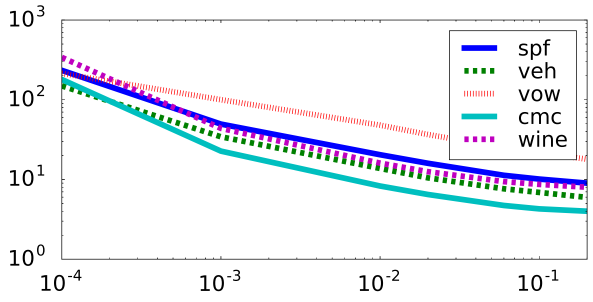
***(d)** DT accuracy: nurse, pen, pid, w40, ctg*



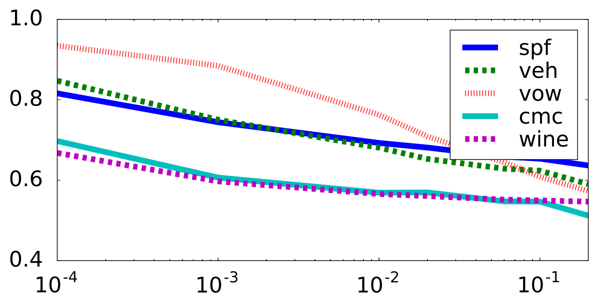
***(e)** DT size: cvf, hrtc, jvow, liv, ttt*



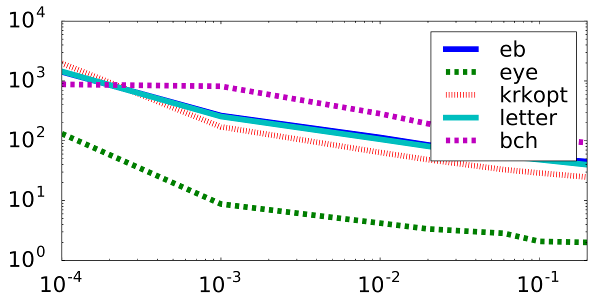
***(f)** DT accuracy: cvf, hrtc, jvow, liv, ttt*



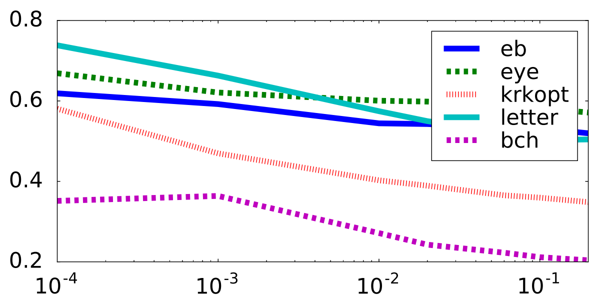
***(g)** DT size: spf, veh, vow, cmc, wine*



***(h)** DT accuracy: spf, veh, vow, cmc, wine*



***(i)** DT size: eb, eye, krkopt, letter, bch*



***(j)** DT accuracy: eb, eye, krkopt, letter, bch*

***Figure 3.16:** Dependencies of the induced DT sizes and accuracies on the oversize weight (K_o) parameter values. Datasets 25-50.*

its parent in fitness, it is always taken as the new candidate solution, and is discarded otherwise. Hence, it can be called greedy. An improvement to this basic version of the selection procedure will be discussed in the Section 3.3.2, in which a less fit individual is sometimes given a chance to be selected.

Algorithm 3.5: *The pseudo-code of the* `select()` *function of the* EFTI *algorithm, that implements the basic individual selection procedure*

```
def select(dt, dt_mut):
    if dt_mut.fit > dt.fit:
        return dt_mut
    else:
        return dt
```

3.3 Improvements to the basic *EFTI* algorithm

In this section several additional features that can improve either the execution time or the quality of solutions produced by the *EFTI* algorithm are discussed:

- Section 3.3.1 - Make fitness dependent on the number of training set classes that are not represented in the DT individual, i.e not assigned to any leaf.
- Section 3.3.2 - Introduce the search probability, i.e. the probability with which a less fit individual can be selected for the candidate solution.
- Section 3.3.3 - Improve the induction times by keeping track of the classification traversal paths, and trying to reuse them between iterations.

3.3.1 Unrepresented classes

When working with highly imbalanced datasets, the induced DT can happen to contain no leaves to which an under-represented class has been assigned. In these cases it might be useful to encourage the *EFTI* algorithm to represent all classes from the dataset within the DT. Here, an extension to the fitness formula is given that aims at discouraging the DTs in which some classes are not represented. The percentage of missing classes is calculated as the percentage of the classes for which the DT does not have a leaf, to the total number of classes in the training set (N_C):

$$missing = \frac{N_c - N_{DTc}}{N_c} \tag{8}$$

where N_{DTc} is the number of classes represented in the DT leaves. The fitness calculation is then updated so that the penalties are taken for the missing classes in the DT individual: `dt.fit = accuracy*(1 - Ko*oversize*oversize)*(1 - Km*missing)`, where the parameter K_m is used to control how much influence the number of missing classes will have on overall fitness.

3.3.2 Search probability

Evolution is inherently an unpredictable process. It is akin to searching for the highest peak in the mountain range but only being able to see one's immediate vicinity, i.e. not being able to peek at distant mountain tops that could guide one's exploration (see Figure 3.17). Simplest strategy for conquering the peak closest to one's current location is to always choose the path that leads upwards. This strategy is thus called the greedy hill-climbing strategy. However, there is no guarantee that the closest peak is in the same time the highest in the mountain range and it often is not. One example of such a peak is the peak marked by the letter A in the Figure 3.17, which is called the local maximum. It is a maximum, since all points in its neighborhood have lower elevation, but it is only local since there is a higher peak in this search space, namely B from the Figure 3.17. The greedy approach described above fails in finding a path from point A to point B, since there exist no monotonically uphill path connecting A to B. In order to get to point B the exploration has to first traverse through the regions with lower elevation, shown by an arrow in the Figure 3.17, in order to get to the base of the hill with the summit at the point B, from which it can start moving up again. However, it is not clear in which direction from the point A the movement should proceed. Nothing is gained if the movement continues towards the point C, since the predominant uphill movement will eventually bring the exploration back to the point A, only wasting the computational time. Even worse, if the exploration step size is large, the position might be moved to the point D, from where it could wander off in the opposite direction from the global maximum B.

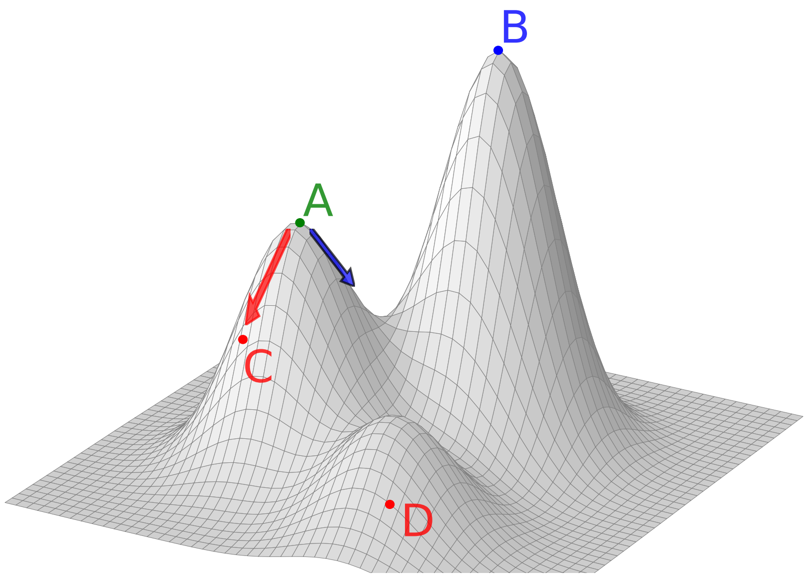


Figure 3.17: *An example of the hill climbing problem and the issue of escaping the local optimum A by a greedy strategy in order to reach point B.*

In terms of the evolutional DT induction, instead of striving for higher elevation, the algorithm is striving for a DT individual with higher fitness, and instead of walking in the mountains, the DT individual is being mutated to move around in the search space. For practical problems, the search spaces for the DT individuals have much higher dimensionality and are thus much more complicated than the hill-climbing problem described above. However, the main idea is the same, in order to visit and discover as many fitness peaks as possible (in order to find the highest one), the algorithm sometimes needs to pursue a less fit individual. Since it is impossible to tell which poorer performing solution will eventually lead to an improved one, the decision of going after a poorer solution is made at random with some probability. Here, this probability will be called the search probability, since it allows for the evolutionary process to search the wider neighborhood of the current solution. Without this search, the systems tend to get stuck at local maximas.

Several approaches to providing the values for the search probability that were tried with the *EFTI* algorithm in an effort to increase its performance, will now be discussed. The results of the experiments used to infer which of the approaches offers statistically significant improvement to the quality of solutions induced by *EFTI*, are discussed in the Section 3.3.2.4.

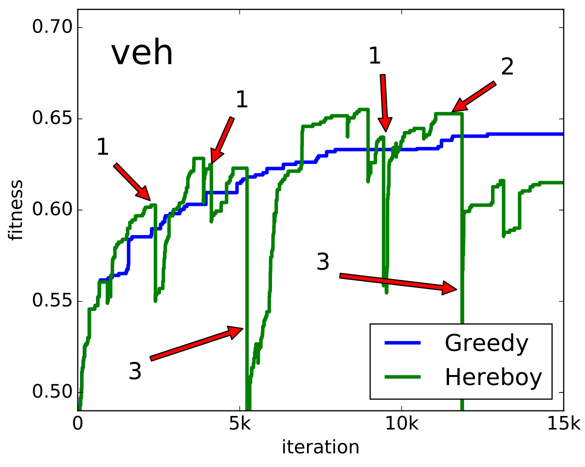
3.3.2.1 HereBoy

One approach for selecting the search probability is implemented by the HereBoy algorithm *[76]*, and is based on the concept used in the Simulated Annealing. The probability is given high value in the beginning and is reduced over time, which is referred as the cooling schedule in the Simulated Annealing literature. The idea behind the cooling schedule is to allow the system a lot of freedom to explore the search space at the beginning when the system is in a high state of disorder, i.e. when only poor solutions are available. Then, slowly, as the desired structures emerge, i.e. better solutions are being found, the freedom to search is restricted so that these structures are not destroyed. The following equation shows how HereBoy calculates the search probability, but in terms of the DT fitness as used by the *EFTI* algorithm, where the constant 1 in the equation corresponds to the maximal possible fitness:

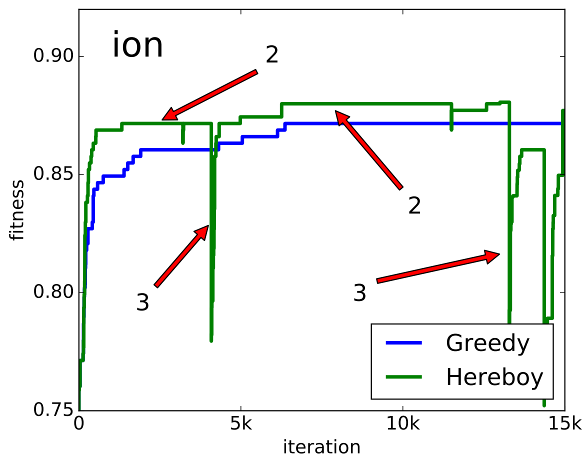
$$\rho = \rho_0(1 - \texttt{fit}) \tag{9}$$

There are several potential issues with using the HereBoy approach to search probability. The Figure 3.18 shows two examples of how the fitness changes during the DT induction when the HereBoy approach is used and when no search probability is used. Several potential issues are pointed out on the plots, by marking the relevant moments in the DT induction when the effects of these issues make an influence on the evolution of the DT fitness.

First, the maximum possible fitness that the DT can attain, which influences the search probability via equation (9), is different for different datasets, and is not known in advance. Second, sometimes during the DT evolution, there are intervals when better solutions are found often, which is akin to standing at the slope of a hill in the hill climbing problem (for an example at the point C in the Figure 3.17). It might be worthwhile to let evolution reach a plateau before trying to search the less fit neighborhood. With the HereBoy approach there is no such mechanism, and it is possible to interrupt the hill climbing any time, which manifests itself in the drops in fitness in the middle of rapid climbing, marked with #1 in the Figure 3.18. On the other hand, by not changing the candidate solution for a large number of iterations, the execution time is wasted by not exploring as large portion of the search space as it was possible. With the HereBoy approach, the search probability remains fixed when there is no change in fitness, making possible for a large iteration intervals when no solutions are accepted, especially if the search probability is low (for an example the current fitness is close to the maximum value of 1). These intervals are marked with #2 in the Figure 3.18. Finally, the search probability is equal for all mutated individuals, no matter their fitness. Sometimes, even small changes to the node test coefficients can produce significant shifts in the way the DT classifies the training



(a) Induction from `veh` dataset.



(b) Induction from `ion` dataset.

Figure 3.18: *Plots of the fitness evolutions during first 15k iterations of the DT induction from `veh` and `ion` datasets when the HereBoy search probability strategy is used (green) and when no search probability is used (blue). Several potential issues with the HereBoy search probability approach are pointed out: 1 - Poorer solution accepted and interrupted a series of fitness advancements, 2 - No new solutions accepted for a long time, wasting execution time, 3 - Solution with significantly less fitness accepted.*

set, especially if the DT is large and the mutated node is near the root. Hence, there is a substantial chance of accepting a significantly less fit individual with this approach, which is akin to jumping to the point D in the search space, as shown in the Figure 3.17. These large jumps can be seen in the Figure 3.18 marked with #3.

3.3.2.2 Metropolis

The Metropolis approach to the search probability calculation was devised for the *EFTI* algorithm based on the idea of the similar method used in the Simulated Annealing called the Metropolis criterion (or Metropolis-Hastings criterion) *[77]*. The adoption of Metropolis criterion is an attempt to remedy the issue where all less fit mutated individuals have the same probability of being accepted, no matter their fitness. Hence in the Metropolis approach, the fitness of the mutated individual (`dt_mut.fit`), more precisely the relative difference between the candidate solution fitness (`dt.fit`) and the mutated individual fitness, will have its influence through the following factor:

$$\rho \sim e^{-\frac{\Delta_F}{S_T}},$$
$$\Delta = \frac{\texttt{dt.fit} - \texttt{dt\_mut.fit}}{\texttt{dt.fit}}, \tag{10}$$

where S_T is the search temperature, which dictates how much less fit an individual can be, and still have a chance to be accepted. This is user supplied parameter, and it is kept constant throughout the induction. Furthermore, for at the same time to allow the algorithm to climb the current hill uninterrupted, and to discourage long iteration intervals where no solution is selected, a concept of stagnation duration D_s is introduced, which is defined as the number of iterations where no improvement to fitness has been made. The search probability is then made proportional to the D_s to finally obtain its final form within the Metropolis approach:

$$\rho(D_s, \Delta_F) = \rho_0 D_s e^{-\frac{\Delta_F}{S_T}} \tag{11}$$

Basically, the search probability restarts to 0 after each advancement in the fitness happens, and increases linearly with each iteration in which no such advancement is made. Since the main idea is not to select a less fit individual either too early or too late after the advancement in the fitness, we are basically interested in determining how high is the probability p_s, of accepting an individual of certain fitness in an iteration interval after the advancement in fitness, as a function of D_s:

$$\begin{aligned} p_s(D_s) &= \sum_{i=1}^{D_s} \rho_i \prod_{j=1}^{i-1} (1-\rho_j) \\ &= \sum_{i=1}^{D_s} \rho_0 i e^{-\frac{\Delta_{Fi}}{S_T}} \prod_{j=1}^{i-1} (1-\rho_0 j e^{-\frac{\Delta_{Fj}}{S_T}}) \end{aligned} \tag{12}$$

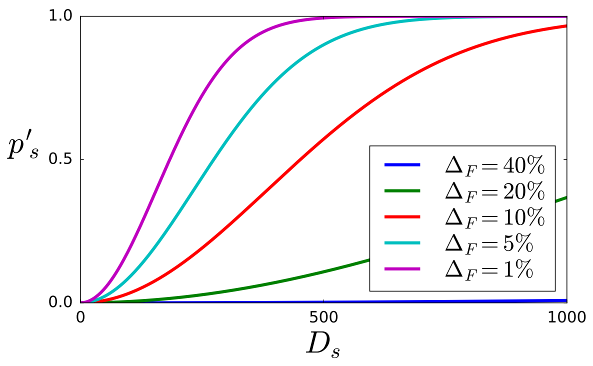
It is obvious from the equation (12), that p_s depends on the fitnesses of all proposed mutated individuals in previous D_s iterations, which are in turn some random variables. Hence to avoid elaborate mathematical procedure of obtaining the distribution for the p_s in general case, a simplified case is considered. The plots in the Figure 3.19 represent $p_s(D_s)$ functions for various values of Δ_F, ρ_0 and S_T, with the simplification that all $\Delta_F i$ values from the (12), are equal to Δ_F. In other words, a case is considered where in all past D_s iterations, all proposed mutated individuals had an equal fitness. This simplified version of $p_s(D_s)$ is called $p'_s(D_s)$.

It can be seen from the plots in the Figure 3.19, that all the functions have a sigmoid shape, which is in fact what was intended. The probability of accepting the less fit solution is low in the interval where the stagnation duration is small, then increases at certain pace (depending on the parameters selected) as the iterations pass, until it approaches a 100% chance of being selected. The plots show that the parameter S_T influences how differently will the individuals with different fitnesses be treated. When $S_T = 0.05$ (Figure 3.19a and Figure 3.19b), the curves are far apart, hence it will be much harder for the individuals with lower fitnesses to get selected, and the algorithm will only explore individuals with fitnesses closer to the fitness of the candidate solution. On the other hand, for higher values of the parameter S_T (Figure 3.19c and Figure 3.19d), the p_s curves are tighter together and the differences between individuals of different fitnesses are blurred. In this case, the algorithm will explore individuals from wide fitness range. As for the parameter ρ_0, the higher its value is, the sooner another less fit individual will get selected.

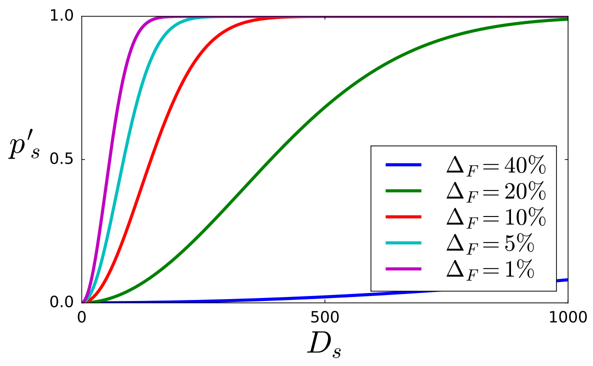
The Figure 3.20 shows the way the fitness of the induced DT individual evolved when Metropolis approach was used. It can be seen that there are significantly less big fitness drops and that the plateaus are shorter. And indeed, this approach succeeded in helping find better solutions in the first 15k iterations than the HereBoy approach did (Figure 3.18a and Figure 3.18b).

3.3.2.3 Multiple restarts

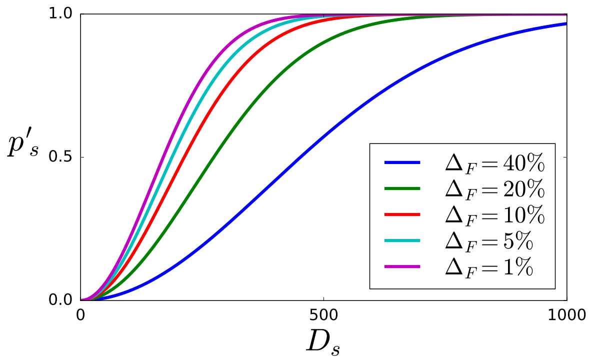
It was observed that sometimes, for some datasets, after the poorer solution has been selected, the evolutionary process never succeeds in bringing the fitness back to the levels where it was



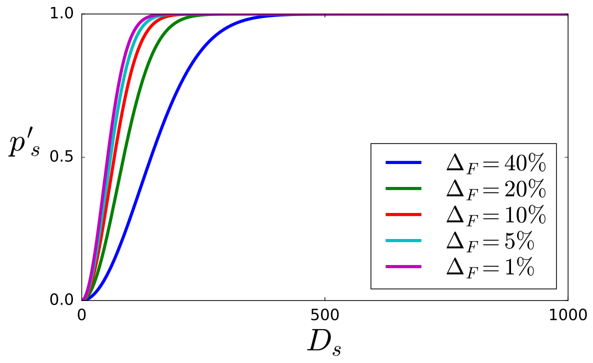
(a) *$S_T = 0.05$, $\rho_0 = 5 \times 10^{-5}$*



(b) *$S_T = 0.05$, $\rho_0 = 5 \times 10^{-4}$*

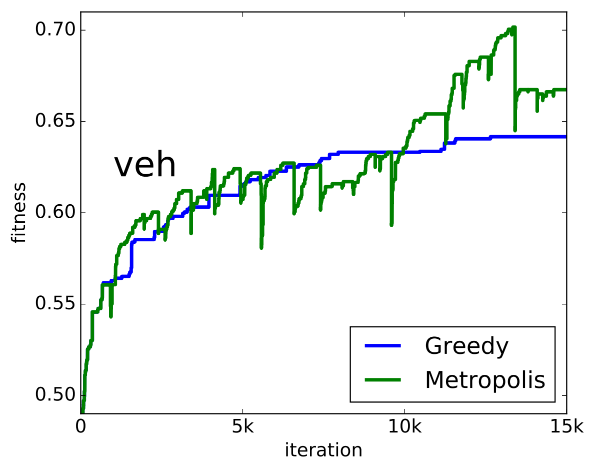


(c) *$S_T = 0.2$, $\rho_0 = 5 \times 10^{-5}$*

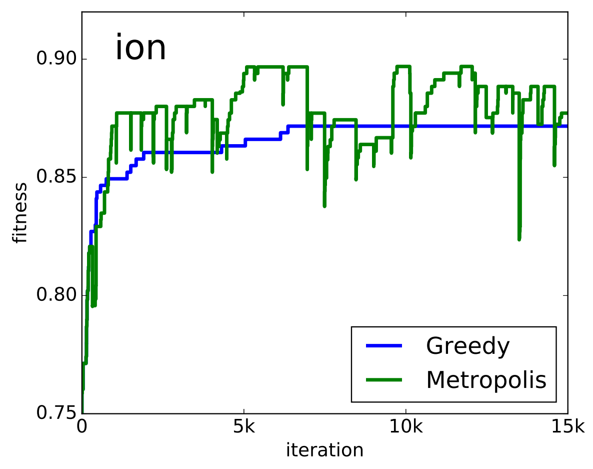


(d) *$S_T = 0.2$, $\rho_0 = 5 \times 10^{-4}$*

Figure 3.19: *The simplified version of the probability of accepting a less fit individual of certain fitness in D_s iterations after the advancement in fitness. In each plot, for different values of S_T and ρ_0, the $p'_s(D_s)$ function is plotted for an individuals whose fitness is smaller than that of the current candidate solution by: 1%, 5%, 10%, 20% and 40%.*



(a) *Induction from `veh` dataset.*



(b) *Induction from `ion` dataset.*

Figure 3.20: *Plots of the fitness evolutions during first 15k iterations of the DT induction from `veh` and `ion` datasets, when the Metropolis search probability strategy is used (green) and when no search probability is used (blue).*

before. Hence an addition to the search method has been proposed that would prevent the evolutionary process to explore for too long with individuals with the fitness less then the current known best fit individual. A parameter called return probability p_R was introduced that determines the probability the evolutionary process has each iteration of returning to the best known candidate solution.

3.3.2.4 Experiments

The experimental procedure explained in the Section 2.8 was used to discover whether any of the proposed approaches statistically influences the induced DTs' fitnesses for the better. The values of the parameters relevant to the search probability that were used in the experiments are given in the Table 3.3 for all tested approaches.

Table 3.3: *The values of the parameters relevant to the search probability set to the* EFTI *algorithm while running the experiments for comparing different search probability approaches*

Approach	ρ_0	S_T	p_R
Greedy	0	–	–
HereBoy	1×10^{-3}	–	0
Metropolis	5×10^{-5}	0.05	0
Metropolis with restarts	5×10^{-5}	0.05	1×10^{-4}

The results are given in the Table 3.4, where for each of the discussed approaches, the mean value of the induced DTs fitness is given together with the 95% confidence intervals, and its ranking based on the Tukey HSD. The fitness values shown in the results table are not the ones used during the induction, but are calculated based on the accuracy of the induced DT on the test set. The results in the first table column, titled Greedy, were obtained without using any search strategy, i.e. by only ever accepting the solution with higher fitness.

Table 3.4: *Average fitness values of the induced DTs using four selection strategies, together with their 95% confidence intervals and Tukey HSD based rankings*

	Greedy		Hereboy		Metropolis		Metropolis with restarts	
Dataset	Fitness	Rank	Fitness	Rank	Fitness	Rank	Fitness	Rank
adult	0.834 ± 0.002	2	0.837 ± 0.001	1	0.833 ± 0.001	2	0.835 ± 0.001	1
ausc	0.885 ± 0.002	3	0.891 ± 0.002	2	0.892 ± 0.002	1	0.896 ± 0.002	1
bank	0.888 ± 0.002	3	0.889 ± 0.001	2	0.889 ± 0.001	2	0.892 ± 0.001	1
bc	0.924 ± 0.006	2	0.935 ± 0.004	1	0.933 ± 0.002	1	0.939 ± 0.003	1
bch	0.241 ± 0.001	2	0.241 ± 0.001	1	0.227 ± 0.001	4	0.230 ± 0.002	3
bcw	0.978 ± 0.001	2	0.978 ± 0.001	1	0.978 ± 0.001	1	0.978 ± 0.001	1
ca	0.883 ± 0.003	3	0.888 ± 0.002	2	0.891 ± 0.002	1	0.892 ± 0.002	1
car	0.865 ± 0.005	2	0.874 ± 0.003	1	0.855 ± 0.002	3	0.868 ± 0.004	1
cmc	0.571 ± 0.007	3	0.596 ± 0.005	2	0.599 ± 0.004	1	0.606 ± 0.005	1
ctg	0.761 ± 0.006	2	0.779 ± 0.005	1	0.776 ± 0.005	1	0.783 ± 0.006	1
cvf	0.767 ± 0.004	3	0.786 ± 0.003	1	0.780 ± 0.002	2	0.790 ± 0.002	1

Continued on next page

Table 3.4 – continued from previous page

	Greedy		Hereboy		Metropolis		Metropolis with restarts	
Dataset	Fitness	Rank	Fitness	Rank	Fitness	Rank	Fitness	Rank
eb	0.621 ± 0.007	3	0.635 ± 0.007	2	0.636 ± 0.004	2	0.649 ± 0.006	1
eye	0.613 ± 0.005	1	0.602 ± 0.003	2	0.594 ± 0.001	3	0.598 ± 0.001	2
ger	0.942 ± 0.009	2	0.966 ± 0.002	1	0.968 ± 0.001	1	0.969 ± 0.002	1
gls	0.797 ± 0.007	2	0.827 ± 0.008	1	0.818 ± 0.008	1	0.823 ± 0.010	1
hep	0.906 ± 0.008	2	0.915 ± 0.005	1	0.917 ± 0.005	1	0.923 ± 0.006	1
hrtc	0.717 ± 0.006	2	0.727 ± 0.006	1	0.723 ± 0.005	1	0.729 ± 0.005	1
hrts	0.873 ± 0.005	2	0.886 ± 0.004	1	0.893 ± 0.003	1	0.891 ± 0.005	1
ion	0.899 ± 0.010	2	0.937 ± 0.004	1	0.937 ± 0.004	1	0.937 ± 0.005	1
irs	0.983 ± 0.004	2	0.987 ± 0.002	1	0.987 ± 0.002	1	0.987 ± 0.002	1
jvow	0.799 ± 0.009	2	0.822 ± 0.007	1	0.835 ± 0.006	1	0.832 ± 0.007	1
krkopt	0.399 ± 0.004	2	0.413 ± 0.006	1	0.420 ± 0.006	1	0.412 ± 0.006	1
letter	0.591 ± 0.006	3	0.608 ± 0.007	2	0.623 ± 0.005	1	0.614 ± 0.009	1
liv	0.760 ± 0.005	2	0.761 ± 0.006	1	0.762 ± 0.004	1	0.766 ± 0.005	1
lym	0.883 ± 0.007	3	0.899 ± 0.008	2	0.909 ± 0.007	1	0.914 ± 0.006	1
magic	0.835 ± 0.003	2	0.838 ± 0.001	1	0.832 ± 0.001	2	0.839 ± 0.001	1
msh	0.960 ± 0.003	3	0.974 ± 0.001	2	0.981 ± 0.001	1	0.980 ± 0.002	1
nurse	0.896 ± 0.009	2	0.909 ± 0.004	1	0.911 ± 0.001	1	0.915 ± 0.002	1
page	0.956 ± 0.002	3	0.961 ± 0.002	2	0.962 ± 0.001	2	0.966 ± 0.001	1
pen	0.928 ± 0.004	3	0.938 ± 0.003	2	0.940 ± 0.002	1	0.943 ± 0.003	1
pid	0.790 ± 0.004	2	0.798 ± 0.002	1	0.799 ± 0.002	1	0.801 ± 0.002	1
psd	0.991 ± 0.005	2	0.999 ± 0.000	1	0.998 ± 0.000	1	0.998 ± 0.001	1
sb	0.935 ± 0.000	2	0.935 ± 0.000	1	0.935 ± 0.000	1	0.935 ± 0.000	1
seg	0.918 ± 0.006	3	0.932 ± 0.003	2	0.939 ± 0.003	1	0.941 ± 0.003	1
shuttle	0.995 ± 0.001	2	0.997 ± 0.000	1	0.996 ± 0.000	1	0.997 ± 0.000	1
sick	0.952 ± 0.006	2	0.956 ± 0.007	1	0.958 ± 0.002	1	0.962 ± 0.003	1
son	0.844 ± 0.011	3	0.875 ± 0.007	2	0.894 ± 0.006	1	0.887 ± 0.007	1
spect	0.923 ± 0.004	2	0.928 ± 0.003	1	0.931 ± 0.003	1	0.934 ± 0.004	1
spf	0.697 ± 0.003	3	0.708 ± 0.004	1	0.707 ± 0.003	2	0.714 ± 0.004	1
thy	0.953 ± 0.004	3	0.959 ± 0.004	2	0.964 ± 0.003	2	0.971 ± 0.004	1
ttt	0.733 ± 0.006	3	0.771 ± 0.010	1	0.763 ± 0.010	2	0.786 ± 0.009	1
veh	0.664 ± 0.010	3	0.704 ± 0.008	2	0.724 ± 0.008	1	0.730 ± 0.010	1
vene	0.935 ± 0.002	2	0.937 ± 0.002	1	0.937 ± 0.002	1	0.937 ± 0.002	1
vote	0.957 ± 0.005	2	0.968 ± 0.003	1	0.968 ± 0.002	1	0.972 ± 0.002	1
vow	0.705 ± 0.010	2	0.731 ± 0.009	1	0.733 ± 0.007	1	0.724 ± 0.012	1
w21	0.859 ± 0.004	2	0.865 ± 0.002	1	0.859 ± 0.001	2	0.863 ± 0.002	1
w40	0.839 ± 0.004	3	0.843 ± 0.003	2	0.838 ± 0.002	3	0.850 ± 0.002	1
wfr	0.740 ± 0.009	2	0.771 ± 0.009	1	0.770 ± 0.011	1	0.770 ± 0.010	1
wilt	0.947 ± 0.001	2	0.946 ± 0.000	1	0.947 ± 0.001	1	0.947 ± 0.002	1
wine	0.567 ± 0.002	2	0.570 ± 0.002	1	0.561 ± 0.001	3	0.567 ± 0.002	2
zoo	0.977 ± 0.006	2	0.981 ± 0.005	1	0.977 ± 0.006	1	0.981 ± 0.005	1
rank	2.33		1.31		1.41		1.08	

The results shown in the Table 3.4 clearly indicate that any approach that allows for exploring the search space via less fit individuals, i.e. any approach that uses some kind of search probability is superior than the greedy hill-climbing, which was ranked lowest with average ranking of 2.33. As for the Hereboy and Metropolis approaches, for some datasets one generated better results and for the others the other one did. However, for vast majority of the datasets used, the best results were obtained by using the Metropolis with multiple restarts that yielded average ranking of 1.08, hence this technique was finally implemented into the *EFTI* algorithm.

3.3.2.5 Search probability implementation in *EFTI*

Metropolis with multiple restarts approach was implemented for the selection procedure, since it was shown in Section 3.3.2.4 to yield the best results among proposed solutions. A new variable `dt_best` needed to be included into the `efti()` function to store the best solution found so far, because when selecting less fit individuals is allowed, the current solution candidate `dt` might not be in the same time the best solution overall. The new pseudo-code for the `efti()` function is given in the Algorithm 3.6.

Algorithm 3.6: *The pseudo-code of the `efti()` function of the* EFTI *algorithm when using Metropolis with multiple restarts*

```
def efti(train_set, max_iter):
    dt_best = dt = initialize(train_set)
    fitness_eval(dt, train_set)

    for iter in range(max_iter):
        dt_mut  = mutate(dt)
        fitness_eval(dt_mut, train_set)

        dt, dt_best = select(dt, dt_mut, dt_best)

    return dt_best
```

Within the `select()` function, the logic for selecting the less fit individual and returning to the best solution need to be implemented, as shown in the Figure 3.7. When the evolution finds a solution better then the current candidate, the selection procedure will also check if it is the overall best, and if so, store it inside `dt_best` variable. On the other hand, if mutation did not advance the fitness, the stagnation duration will be increased and the search probability will be calculated based on it using the Metropolis criterion. A chance will be than given to the selection procedure to terminate the search and return to the best solution overall. Otherwise, the less fit `dt_mut` individual might get selected at random with the current value of search probability.

Algorithm 3.7: *The pseudo-code of the `select()` function of the* EFTI *algorithm when using Metropolis with multiple restarts*

```
def select(dt, dt_mut, dt_best):
    if dt_mut.fit > dt.fit:
        stagnation_duration = 0
        dt = dt_mut

        if dt_mut.fit > dt_best.fit:
            dt_best = dt_mut
    else:
        stagnation_duration += 1
        diff = (dt.fit - dt_mut.fit)/dt.fit
        search_probability = stagnation_duration * rho_0 * \
                             exp(-diff/S_T);
        if random() < restart_probability:
            stagnation_duration = 0
            dt = dt_best
        elif random() < search_probability:
            dt = dt_mut

    return dt, dt_best
```

3.3.3 Partial reclassification

As it was already discussed in the Section 3.2.1, the DT mutations alter only a small portion of the DT individual in each iteration, hence only the classification of the instances on whose traversal paths the mutated nodes happen to reside, will be affected by the mutation. Therefore the majority of instances will travel along identical paths from iteration to iteration, meaning that all related computations will remain the same. Recomputation is thus only necessary for the instances whose paths contain a mutated node. Please also notice that even when the mutated node test coefficients change, only the elements of the vector scalar product sum (given in the equation (1)) that correspond to the mutated coefficients must be recomputed, while the computation of all other elements can be skipped.

Therefore, the traversal paths could be memorized for the candidate DT individual in order to avoid unnecessary recalculations of the node tests during the classification of the mutated DT individual, for the instances whose paths do not cross the mutated nodes. Each instance could start the DT traversal by following its memorized path from the candidate DT individual classification, and checking whether it will encounter any of the mutated nodes while traversing the DT. While no mutated nodes are encountered, no test recalculations need to be executed and the instance moves through the DT as dictated by the path stored in the memory. When the instance encounters a mutated node, its path in the mutated DT might diverge from its memorized path. If the topological mutation produced the changes in the encountered node, where either a new node was added in the place of a leaf (see Figure 3.9 for an example) or the node was removed and a different one took its place (see Figure 3.10 for an example), the sub-tree which the instance has reached has changed, and the rest of the traversal path needs to recomputed. If the instance encounters a node with only some of its coefficients $\mathbf{w}$ mutated, the dot product of the mutated node test ($\mathbf{w^{mut}} \cdot \mathbf{x}$), can be calculated based on the dot product of the original node test ($\mathbf{w} \cdot \mathbf{x}$) in the following way:

$$\mathbf{w^{mut}} \cdot \mathbf{x} = \mathbf{w} \cdot \mathbf{x} + \sum_{i \in M} (w_i^{mut} - w_i) x_i, \tag{13}$$

where M is the set of indices of all the mutated coefficients in that node. Furthermore, the mutation on the encountered node may not be strong enough to deflect the instance from its previous path. Hence, the outcome of the mutated node test is monitored whether it will align with the stored path, in which case the instance has not diverged and the instance can continue following the memorized path. Otherwise, the instance entered a new DT sub-tree and all subsequent node tests need to be recalculated.

In the case the mutated individual is selected for the new candidate solution, the paths which have diverged in the classification run need to be updated to the memory. One possible way to implement this is to keep track of each deviation from the memorized paths during the classification run for the mutated DT individual, and apply all these changes to the stored traversal paths if the individual is selected for the new candidate solution. However, a different method that takes advantage of the fact that usually less than 1% of the mutated individuals get selected, proved to be more efficient with respect to both execution time and the memory resource consumption. In this approach, the *EFTI* algorithm does not keep track of the deviations from the memorized paths in each classification run of a mutated DT, which in turn saves on memory access time and on the memory space for tracking the changes. Only once a mutated DT has been selected for the new candidate solution, is the classification rerun with the instructions to change the stored traversal paths in the memory where needed.

The proposed partial reclassification algorithm has an additional performance issue with the small DT individuals. If the DT individual is only one or two levels deep, there is very large probability that many of the instance paths will be affected by the mutation, and the time consumption overhead of the partial reclassification exceeds its benefits. The *EFTI* algorithm implements a strategy to turn the partial reclassification off when it operates with small individuals.

Algorithm 3.8: *The modified `find_dt_leaf_for_inst()` function that implements the partial reclassification method*

```python
def find_dt_leaf_for_inst(dt, instance, store_paths, recalc_all):

    path_diverged = recalc_all
    cur_node = dt.root

    while not cur_node.is_leaf:
        # if the memorized path is still followed
        if not path_diverged:
            # have we crossed the topologicaly mutated node
            if dt.is_topo_mutated(cur_node):
                psum = dot_product(instance.x, cur_node.w)
                path_diverged = True
            # or only coefficients have mutated
            elif dt.is_coeff_mutated(cur_node):
                # get stored dot product and apply the changes
                psum = get_stored_psum(instance, cur_node)
                for i in dt.mutated_coeff_index(cur_node):
                    psum += (cur_node.w[i] - cur_node.w_orig[i]) \
                            * instance.x[i]

                path_diverged = True
        # else, path has diverged and no testing for crossing
        # mutated nodes is needed
        else:
            psum = dot_product(instance.x, cur_node.w)
        # still have not diverged, look-up the stored next node
        if not path_diverged:
            cur_node = get_stored_next_node(instance, cur_node)
        # path has diverged and the node test needs to be performed
        else:
            if psum < cur_node.thr:
                cur_node = cur_node.left
            else:
                cur_node = cur_node.right

            # has the instance stayed on the path in spite mutation
            if cur_node == get_stored_next_node(instance, cur_node):
                path_diverged = False

        # should the path be memorized
        if store_paths:
            store_node_to_path(instance, cur_node, psum)

    return cur_node
```

The pseudo-code in the Algorithm 3.8 describes the implementation of the partial reclassification method within `find_dt_leaf_for_inst()` function (the original implementation is given by the Algorithm 3.4). If the partial reclassification is turned off by *EFTI* algorithm (by passing the value **`True`** for the argument `recalc_all`), the paths of all the training set instances will be immediately considered to have diverged from the stored paths, and the partial classification algorithm will not be used, making the classification procedure effectively same as the original one. Otherwise, the classification for an instance (variable `instance`) starts by following the stored path (`path_diverged = recalc_all =` **`False`**) from the root node (`cur_node = dt.root`). The path is followed one node at a time (`cur_node = get_stored_next_node(instance, cur_node)`), in order to look out for mutated nodes along its length, by using the functions `dt.is_topo_mutated(cur_node)` and `dt.is_coeff_mutated(cur_node)`, which signal, respectively, if the current node was mutated via topological mutation or only its test coefficients were mutated. If it was changed by a topological mutation, the instance is facing completely different node, hence the dot product is calculated a new. On the other hand if the current node's test coefficients were mutated, the dot product is reconstructed from the stored value (retrieved via `get_stored_psum(instance, cur_node)`), using the equation (13). In both cases, it is considered that the instance has diverged from the memorized path: `path_diverged =` **`True`**. The rest of the node test is carried out by comparing the dot product with the threshold to obtain the next node in the path, and if that node corresponds to the next node in the stored path, instance can safely go back to following it (once again `path_diverged =` **`False`**). Finally, in order not to update the memorized paths in each classification run, the argument `store_paths` is used to signal to `find_dt_leaf_for_inst()` function whether the mutated DT individual has become the new candidate solution and the updates to the memory should take place.

In the Table 3.6 the results of an experiment are shown that tests the performance benefits of utilizing the partial reclassification procedure, obtained by the cross-validation procedure explained in the Section 2.8 and the set of parameters listed in the Table 3.5. The results show that the partial reclassification really shortens the execution time, but that the induction speedups differ between the datasets, and depend on the size of the induced DTs, as was expected and already discussed in this section.

Table 3.5: *The parameter set used for the* EFTI *algorithm in the partial reclassification comparison experiments*

`max_iter`	K_o	α	β	ρ_0	S_T	p_R
500k	0.01	1	0.6	5×10^{-5}	0.05	1×10^{-4}

Table 3.6: *The results of the experiments testing the benefits on the* EFTI *algorithm induction times of using the partial reclassification procedure*

Dataset	Original	Partial reclassification	Dataset	Original	Partial reclassification
adult	387.34 ± 08.90	221.53 ± 05.75	msh	102.89 ± 09.77	68.49 ± 04.10
ausc	4.49 ± 00.08	4.37 ± 00.21	nurse	164.04 ± 08.15	112.53 ± 02.25
bank	619.58 ± 07.99	311.70 ± 03.34	page	47.24 ± 03.04	36.59 ± 01.61
bc	3.95 ± 00.19	4.25 ± 00.11	pen	345.05 ± 11.57	140.94 ± 02.01
bch	378.48 ± 13.19	115.62 ± 01.24	pid	4.10 ± 00.27	4.48 ± 00.31
bcw	3.06 ± 00.02	3.35 ± 00.05	psd	10.08 ± 00.25	8.55 ± 00.18
ca	4.92 ± 00.17	4.45 ± 00.05	sb	15.51 ± 00.03	16.10 ± 00.31
car	16.01 ± 00.56	14.37 ± 00.40	seg	57.50 ± 01.75	29.58 ± 00.62
cmc	16.26 ± 00.69	11.93 ± 00.29	shuttle	1015.40 ± 60.03	841.89 ± 22.17
ctg	69.02 ± 02.47	27.36 ± 00.64	sick	31.03 ± 01.17	31.72 ± 01.23
cvf	177.57 ± 08.23	110.56 ± 02.76	son	4.90 ± 00.19	2.74 ± 00.16
eb	1277.34 ± 92.67	737.24 ± 09.88	spect	2.44 ± 00.17	1.85 ± 00.10
eye	94.41 ± 07.99	96.22 ± 05.96	spf	61.75 ± 01.89	25.95 ± 00.45
ger	10.04 ± 00.44	8.16 ± 00.68	thy	35.77 ± 04.50	28.16 ± 00.91
gls	3.98 ± 00.16	2.24 ± 00.07	ttt	6.31 ± 00.41	6.39 ± 00.32
hep	1.63 ± 00.14	1.29 ± 00.07	veh	17.16 ± 00.43	9.87 ± 00.18
hrtc	5.20 ± 00.21	2.69 ± 00.08	vene	1.52 ± 00.08	2.01 ± 00.05
hrts	2.22 ± 00.10	1.82 ± 00.12	vote	3.18 ± 00.19	2.97 ± 00.10
ion	6.11 ± 00.23	3.19 ± 00.16	vow	24.08 ± 00.63	12.32 ± 00.27
irs	0.90 ± 00.06	0.98 ± 00.05	w21	69.10 ± 01.95	47.53 ± 01.34
jvow	116.28 ± 03.74	119.40 ± 02.90	w40	77.83 ± 02.20	54.74 ± 01.06
krkopt	872.29 ± 22.69	349.76 ± 11.05	wfr	125.74 ± 06.41	62.95 ± 01.00
letter	970.96 ± 35.99	355.80 ± 07.50	wilt	20.72 ± 00.22	20.69 ± 00.44
liv	2.07 ± 00.11	2.26 ± 00.08	wine	75.33 ± 03.29	46.12 ± 01.26
lym	3.01 ± 00.13	1.77 ± 00.05	zoo	2.01 ± 00.08	1.31 ± 00.03
magic	117.75 ± 03.67	117.77 ± 02.44			

3.4 Complexity of the *EFTI* algorithm

The computational complexity of the *EFTI* algorithm can be calculated by following its pseudo-code. The computational complexity is given here in the big O notation, i.e. the worst-case complexity will be calculated. Since the individual selection is performed in constant time it can be omitted, and the total complexity can be computed as:

$$T(EFTI) = \texttt{max\_iter} \cdot (O(\texttt{mutate}) + O(\texttt{fitness\_eval})) \tag{14}$$

The number of leaves, N_l, in binary DT is always by 1 larger then the number of non-leaf nodes. If *n* represents the number of non-leaf nodes in the DT, then:

$$N_l = n + 1 \tag{15}$$

In the worst case, the depth of the DT equals the number of non-leaf nodes, hence:

$$D = N_l - 1 \tag{16}$$

Each iteration α coefficients are mutated, so the complexity of mutating coefficients is constant:

$$T(\texttt{coefficient mutation}) = O(1) \tag{17}$$

The topology can be mutated by either adding or removing the node from the DT. When the node is removed, only a pointer to the removed child is altered so the complexity is:

$$T(\texttt{node removal}) = O(1) \tag{18}$$

When the node is added, the new set of node test coefficients needs to be calculated. hence the complexity is:

$$T(\texttt{node addition}) = O(N_A) \tag{19}$$

Hence, the complexity of the whole DT Mutation task sums to:

$$T(\texttt{mutation}) = O(N_A) \tag{20}$$

Once the number of hits is determined, the fitness can be calculated in constant time $O(1)$, hence the complexity of the whole `fitness_eval()` function is:

$$T(\texttt{fitness\_eval}) = N_I \cdot O(\texttt{find\_dt\_leaf\_for\_inst}) + O(N_l \cdot N_c) + O(1) \tag{21}$$

where N_I is the number of instances in the training set and N_C is the total number of classes in the classification problem, and $O(N_l \cdot N_c)$ is for the dominant class calculation for each leaf. As for the `find_dt_leaf_for_inst()` function, the complexity can be calculated as:

$$T(\texttt{find\_dt\_leaf\_for\_inst}) = D \cdot O(\texttt{dot\_product}), \tag{22}$$

and the complexity of the node test evaluation is:

$$T(\texttt{dot\_product}) = O(N_A) \tag{23}$$

By inserting the equation (23) into the equation (22), and then both of them into the equation (21), we obtain the complexity for the `fitness_eval()` function:

$$T(\texttt{fitness\_eval}) = O(N_I \cdot D \cdot N_A + N_l \cdot N_c) \tag{24}$$

By inserting the equations (24), (20), (15) and (16) into the equation (14), we obtain the total complexity of the *EFTI* algorithm:

$$T(EFTI) = \texttt{max\_iter} \cdot (N_I \cdot N_l \cdot N_A + N_l \cdot N_c + N_A) \tag{25}$$

Since $N_A \ll N_I \cdot N_l \cdot N_A$ the mutation insignificantly influences the complexity and can be disregarded. We finally obtain that the complexity of the *EFTI* algorithm is dominated by the fitness evaluation task complexity, and sums up to:

$$T(EFTI) = O(\texttt{max\_iter} \cdot (N_I \cdot N_l \cdot N_A + N_l \cdot N_c)) \tag{26}$$

3.5 Experiments

In this section, the results of the experiments are presented, that were conducted in order to compare the *EFTI* algorithm to the existing solutions. The algorithms listed in the Table 3.7, available in open literature, were used for the comparison. The experimental procedure explained in the Section 2.8 was used to compare the quality of the induced DTs, in terms of their sizes and accuracies. For the incremental DT inference algorithms, a pruning set was created and the induced DTs were pruned after the induction. For the algorithms: CART-LC, OC1, OC1-AP, OC1-ES and OC1-SA, the default value of 10% randomly selected training set instances were used to form a pruning set, and the Error-Complexity pruning algorithm was used. For the NODT algorithm, the pruning was performed in the manner described in the original publication *[32]*, where a specific pruning algorithm is described and pruning set is created by taking 30% of the training set instances selected at random.

Table 3.7: *The list of the existing algorithms used for the comparison with the proposed* EFTI *algorithm*

Short Name	Name	Description
CART-LC	The Classification and Regression Tree with Linear Combinations	an incremental deterministic algorithm for oblique DT induction. For its implementation, the description provided in *[23]* was used as a reference.
OC1	Oblique Classifier	an incremental randomized algorithm for oblique DT induction,
OC1-AP	Oblique Classifier - Axis-Parallel	the OC1 algorithm limited to inducing only axis-parallel tests,
OC1-ES	Oblique Classifier - Evolutionary Strategy	an extension to OC1 that uses ES to optimize the oblique hyperplanes,
OC1-SA	Oblique Classifier - Simulated Annealing	an extension to OC1 that uses simulated annealing to optimize the oblique hyperplanes,
NODT	HereBoy Decision Tree induction	an incremental randomized algorithm for oblique DT induction, that uses HereBoy *[76]* for the hyperplane optimization process.
GaTree	Genetic Algorithm decision Tree induction	a nonincremental (full tree) DT induction algorithm based on genetic algorithms.
GALE	Genetic and Artificial Life Environment	a nonincremental (full tree) DT induction algorithm based on the cellular automata and the Pittsburgh approach *[35]*.

The software implementation of the *EFTI* algorithm was developed in C, using many optimization techniques in order to maximize its performance regarding the induction speed:

- The node test coefficients are represented in fixed point and all dot product arithmetic operations are performed on 64-bit operands only (optimized for the 64-bit CPU).
- The dot product calculation loop is unfolded for all supported N_A values.
- To save on copying the DT individuals, the current candidate solution (`dt`) and the mutated individual (`dt_mut`) are represented by a single DT in memory. Hence, the mutations are applied directly to the candidate solution. If the mutated solution gets rejected by the `select()` function, the mutations are undone, which since they are sparse is more efficient than copying the whole DT to create a mutated individual. On the other hand if the mutated solution is selected, no actions are needed since the mutated solution is at the same time the candidate solution.
- Special case was introduced for traversing the DT which contains only the root node. The `find_dt_leaf_for_inst()` function contains a lot of programming structures for iterating through the DT and also for deciding whether memorized traversal paths can be reused or not, which is all superfluous for a simple case of one node DT.

- The maximum compiler optimization settings for speed were used.

The software implementation of the *EFTI* algorithm was compiled using the GCC 5.4.1 compiler, and all the experiments were executed on a PC with 64-bit, 4-core, Intel i5-2500K CPU operating at approximately 3.5GHz, with 8GB or RAM, running Ubuntu 16.04 operating system. GALE software, written in java, was run on OpenJDK 1.8, and GaTree, written for Windows OS, was run using Wine 1.6.2 .

3.5.1 Dependence on the number of iterations

First, the results are presented for the set of the experiments that test the dependency of the inferred DT quality to the number of iterations the *EFTI* algorithm was run. The induced DT accuracies and sizes are shown in the Table 3.8 and Table 3.9 respectively, for different number of iterations. The same results are also presented in series of plots in the Figure 3.21 and Figure 3.22. In these figures, the plots are organized in pairs, where each pair consists of the accuracy and size plots for the same five algorithms displayed in juxtaposition. Please notice that the x-axis, corresponding to the number of iterations, is given in logarithmic scale. Please also notice that the ranges for the y-axis, be it for the accuracy or the size plots, vary from plot to plot and depend on which datasets were used for the induction.

Table 3.8: *The average fitness values for the DTs induced using different number of iterations*

Dataset	1k	2k	5k	10k	20k	50k	100k	200k	500k	1000k
adult	80.52	81.50	81.81	82.25	82.50	83.01	83.22	83.48	83.63	83.85
ausc	87.22	87.64	87.77	87.90	88.28	88.99	88.60	89.15	89.77	89.92
bank	88.30	88.34	88.32	88.33	88.42	88.57	88.55	88.74	89.25	89.38
bc	88.16	89.90	90.48	90.94	91.93	93.25	92.51	94.00	94.77	95.16
bch	7.16	7.59	9.70	15.19	21.10	22.73	25.57	26.19	24.89	25.58
bcw	97.49	97.53	97.60	97.71	97.73	97.77	97.68	97.93	97.94	97.96
ca	86.72	87.55	87.66	87.69	88.15	88.85	88.56	89.00	89.46	89.51
car	77.61	78.78	81.11	82.66	84.02	85.30	85.95	87.56	87.71	88.60
cmc	51.54	53.13	53.67	54.80	55.03	57.52	56.74	59.02	61.20	61.29
ctg	59.38	64.44	70.04	72.58	74.23	75.21	76.55	77.91	79.00	79.53
cvf	67.96	69.79	72.52	74.08	74.81	76.89	76.26	78.22	79.06	79.40
eb	14.65	19.26	32.95	41.00	48.13	53.53	59.02	63.29	65.13	65.54
eye	57.74	58.16	58.30	58.55	58.93	59.28	60.00	59.74	60.16	60.34
ger	88.50	90.21	91.88	92.28	92.84	95.70	93.99	95.62	97.08	97.40
gls	72.30	73.59	78.22	79.08	79.36	82.07	81.85	83.85	84.95	85.91
hep	87.43	88.67	89.11	89.57	89.96	91.20	91.12	92.18	93.26	93.70
hrtc	66.56	68.16	69.58	70.17	71.31	72.15	73.39	74.90	74.77	75.62
hrts	84.84	85.75	86.41	87.11	87.13	88.59	87.82	89.11	89.63	90.01
ion	87.05	88.30	88.71	90.18	90.85	93.14	92.23	93.47	94.60	95.13
irs	96.96	97.17	97.44	97.79	98.13	98.29	98.27	98.56	98.83	98.56
jvow	57.46	65.70	70.87	73.35	75.14	78.13	79.10	82.19	84.05	85.73
krkopt	26.05	29.32	33.66	36.18	37.66	39.03	40.09	41.91	42.40	42.93
letter	17.22	27.69	44.63	50.36	53.60	56.73	60.66	62.62	63.38	64.08
liv	66.43	68.77	70.26	71.88	72.65	75.97	75.39	77.01	77.86	78.59
lym	84.16	86.57	87.49	88.57	89.08	90.76	90.86	91.81	93.05	93.62
magic	80.58	80.87	80.87	81.32	82.00	82.63	83.24	83.79	84.09	84.19
msh	91.78	93.40	94.86	95.61	96.10	97.73	97.56	97.98	98.71	98.79
nurse	74.71	78.96	82.50	83.85	86.33	89.35	88.19	90.83	91.60	92.03
page	93.44	94.09	94.56	94.82	95.03	95.74	95.39	96.15	96.66	96.92
pen	78.14	84.14	87.96	89.95	91.35	92.61	93.25	94.89	95.23	95.57
pid	77.10	77.24	77.83	78.57	78.83	79.61	79.53	79.99	80.51	80.85
psd	93.25	94.48	95.68	97.00	97.42	99.32	98.36	99.41	99.80	99.84

Continued on next page

Table 3.8 – continued from previous page

Dataset	1k	2k	5k	10k	20k	50k	100k	200k	500k	1000k
sb	93.42	93.42	93.44	93.43	93.43	93.46	93.46	93.48	93.52	93.54
seg	81.34	84.17	87.74	89.32	90.80	92.40	92.27	93.58	94.69	95.13
shuttle	97.42	97.61	98.49	98.65	98.99	99.35	99.28	99.56	99.69	99.72
sick	94.02	93.95	94.06	94.01	94.30	94.38	94.29	94.71	96.42	96.98
son	78.58	80.52	82.40	81.92	83.29	87.54	85.98	88.67	90.37	90.90
spect	89.27	90.02	90.53	91.22	91.10	92.71	92.46	92.88	93.64	94.32
spf	61.38	63.39	65.87	67.49	68.10	69.45	70.46	72.07	72.41	73.11
thy	93.05	93.70	94.04	94.58	94.78	95.44	95.12	96.01	97.10	97.54
ttt	69.61	70.10	71.33	72.05	73.10	74.79	74.13	75.80	80.15	79.20
veh	58.17	59.55	62.95	62.74	65.03	67.65	65.85	69.97	74.41	75.34
vene	92.45	92.65	92.76	93.23	93.15	93.65	93.61	93.96	94.16	94.36
vote	93.05	93.52	93.75	94.57	94.45	96.35	95.47	96.77	97.49	97.59
vow	47.24	57.76	64.78	69.22	70.46	72.26	74.36	76.28	77.29	78.21
w21	81.83	82.52	83.35	83.75	84.30	85.20	85.36	86.56	86.48	86.88
w40	76.83	78.52	79.29	79.91	80.89	82.69	82.55	84.37	85.25	85.80
wfr	62.45	65.72	68.91	71.17	72.33	74.11	74.03	76.88	78.28	79.95
wilt	94.61	94.61	94.61	94.61	94.61	94.61	94.77	94.65	94.70	94.79
wine	52.25	53.37	54.27	54.80	55.02	55.61	56.04	56.84	56.85	57.14
zoo	94.53	97.03	98.18	97.50	97.94	98.14	97.90	97.54	98.06	98.42

Table 3.9: *The average sizes of the DTs induced using different number of iterations*

Dataset	1k	2k	5k	10k	20k	50k	100k	200k	500k	1000k
adult	2.68	2.68	2.84	2.80	2.68	2.36	2.88	2.72	2.64	2.80
ausc	3.00	3.12	3.16	3.40	3.20	2.84	3.24	3.24	2.96	2.96
bank	2.00	2.04	2.00	2.00	2.00	2.04	2.08	2.08	2.16	2.28
bc	4.28	4.52	4.96	5.40	5.56	5.00	5.80	5.80	5.84	5.84
bch	3.28	4.20	22.04	94.40	213.76	208.56	246.04	251.40	225.00	238.56
bcw	2.12	2.28	2.12	2.32	2.32	2.12	2.28	2.48	2.40	2.36
ca	3.08	3.56	3.40	3.28	3.52	3.00	3.60	3.36	3.04	3.00
car	5.44	6.40	6.40	6.96	7.40	6.76	8.40	8.28	7.92	7.88
cmc	5.68	6.20	6.36	6.72	6.72	6.16	7.56	6.80	5.80	6.36
ctg	16.00	18.28	20.80	23.20	22.92	18.72	24.88	22.80	18.88	19.76
cvf	8.04	9.36	9.80	10.36	10.48	7.72	11.68	8.48	7.92	8.28
eb	6.72	18.20	45.28	58.56	67.32	58.84	73.48	55.92	50.28	53.48
eye	3.16	3.24	3.08	3.24	3.32	3.16	3.52	3.40	3.44	3.40
ger	3.56	3.68	3.44	3.60	3.52	2.84	3.64	3.12	2.68	2.68
gls	12.32	12.64	16.76	16.92	17.56	16.20	17.76	17.40	16.60	16.60
hep	4.04	4.16	4.16	4.32	4.40	3.96	4.40	4.24	4.00	4.00
hrtc	10.92	12.00	12.40	12.56	13.80	12.20	15.12	14.40	12.88	13.24
hrts	3.56	3.76	3.84	3.44	3.88	3.20	4.20	3.80	3.40	3.48
ion	4.76	4.80	4.96	5.00	5.24	4.04	5.08	5.12	3.88	3.84
irs	3.56	3.64	3.84	3.68	4.16	3.44	3.76	3.48	3.72	3.72
jvow	17.88	21.36	22.40	24.36	24.48	19.80	25.72	20.52	17.88	17.24
krkopt	15.48	29.00	40.24	45.68	49.08	45.20	53.84	49.80	47.80	48.96
letter	7.44	28.92	63.16	72.52	82.48	74.96	86.32	76.64	71.36	75.32
liv	3.84	4.24	4.56	4.56	4.64	4.32	5.08	4.56	4.52	4.44
lym	7.48	9.08	10.44	10.64	10.76	9.16	10.88	10.80	9.20	9.52
magic	3.04	3.12	3.00	3.00	3.08	3.00	3.04	3.00	3.00	3.00
msh	3.92	3.96	4.32	4.32	4.56	3.80	4.60	4.44	3.60	3.52
nurse	7.04	7.92	8.00	8.00	8.16	6.20	8.28	6.96	6.28	6.40
page	5.32	5.76	6.28	6.40	6.60	6.08	6.48	6.52	6.16	6.16
pen	20.04	22.64	23.00	24.08	24.08	20.24	24.00	20.60	19.48	19.56
pid	3.28	3.64	3.72	3.56	3.60	3.32	4.00	3.48	3.36	3.40
psd	2.92	3.12	2.92	2.80	2.64	2.16	2.80	2.28	2.00	2.04
sb	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
seg	12.92	14.08	14.88	15.84	15.92	12.88	16.72	14.76	12.56	12.32
shuttle	7.48	7.64	7.56	7.48	7.64	7.48	7.92	7.68	7.24	7.48

Continued on next page

Table 3.9 – continued from previous page

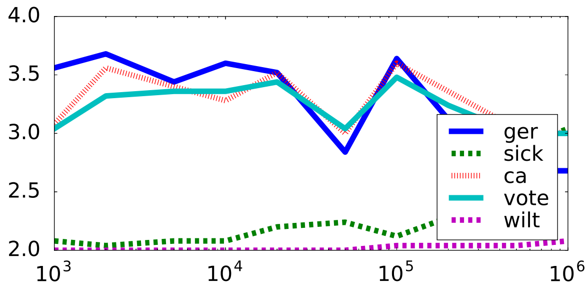
Dataset	1k	2k	5k	10k	20k	50k	100k	200k	500k	1000k
sick	2.08	2.04	2.08	2.08	2.20	2.24	2.12	2.28	2.88	3.04
son	4.92	4.88	5.32	5.84	5.72	4.68	5.84	5.68	4.64	4.60
spect	3.12	3.28	3.40	3.32	3.52	3.00	3.88	3.80	3.04	3.08
spf	12.12	13.40	15.80	16.28	17.00	14.84	18.64	16.88	15.00	16.12
thy	4.04	4.24	4.24	4.32	4.64	4.24	4.92	4.52	4.32	4.40
ttt	3.28	3.56	3.72	4.08	4.44	4.00	4.48	4.28	4.76	4.40
veh	9.48	10.36	10.80	11.28	11.48	9.52	11.80	11.20	9.32	9.36
vene	4.28	4.68	4.68	4.92	4.84	4.76	4.88	4.96	4.96	4.84
vote	3.04	3.32	3.36	3.36	3.44	3.04	3.48	3.24	3.00	3.00
vow	25.32	31.56	37.68	40.20	41.52	38.00	43.00	41.00	38.56	39.44
w21	5.28	5.24	5.44	5.60	5.32	4.52	5.28	4.52	4.32	4.08
w40	5.72	5.60	5.64	5.92	5.84	4.64	5.92	4.68	4.36	4.28
wfr	7.68	8.68	9.60	10.20	10.56	8.76	10.92	10.00	8.96	9.04
wilt	2.00	2.00	2.00	2.00	2.00	2.00	2.04	2.04	2.04	2.08
wine	8.36	8.88	10.08	11.52	11.40	9.76	12.24	11.32	10.80	11.56
zoo	10.44	10.64	10.96	11.68	10.52	8.60	10.68	9.72	7.16	7.08

It can be seen from the results that indeed the more iterations are at disposal, the more accurate the DT solutions become. However, after a certain point, which is different for different datasets, the *EFTI* algorithm is unable to improve on the solution significantly when more iterations are given for the induction. Usually, at around 500k iterations, all advancements in the quality of induced DT individuals have stopped for the vast majority of the datasets. Furthermore, for some datasets like `bank`, `bcw`, `ca`, `irs` and `wilt`, even a 1000 iterations were enough to find a decent solution.

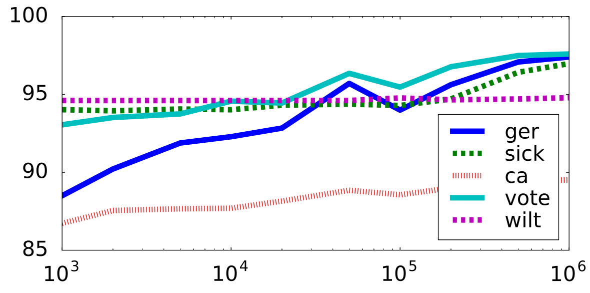
3.5.2 Equitemporal comparison with the existing solutions

This section presents the results of comparison of the quality between DTs induced by the existing algorithms from the Table 3.7, and DTs induced by the *EFTI* algorithm in the same amount of time. Each subsection is devoted to comparison of the *EFTI* algorithm to one of the existing solutions, which was performed by first letting the other algorithm induce the DTs in a five 5-fold cross-validations on all datasets from the Table 2.1, while measuring the induction times. The average induction times were then calculated for each of the dataset, and *EFTI* was then let to perform same five 5-fold cross-validations but constrained per dataset to running only the amount of time that the other algorithm needed on average for the same dataset. For each comparison, two tables were generated: one showing the average induction times per dataset for the algorithm *EFTI* is being compared to, and the other showing the comparison per dataset between the average induced DT accuracies, sizes and fitnesses. The DT fitness used for this comparison is not the same fitness used during the induction by the *EFTI* algorithm, but is recalculated after the induction from the induced DTs' size and the accuracy attained on the test set using the equation (7). The backgrounds of cells in the comparison table are colored in shades of red and blue. The better the performance regarding certain feature (either accuracy, size or fitness), of *EFTI* in comparison to the other algorithm on certain dataset, the darker shade of blue is used. On the other hand if the *EFTI* algorithm performed worse, the worse its performance the darker the shade of red is used as the cell background. The average data in both tables are supplied with their 95% confidence intervals.

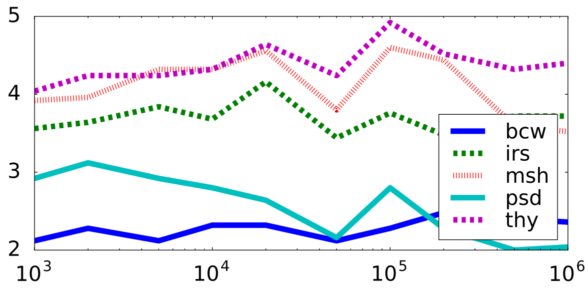
The Table 3.10 shows two sets of the *EFTI* algorithm parameters that were used in the experiments. The “High accuracy” set was used for the comparison with incremental



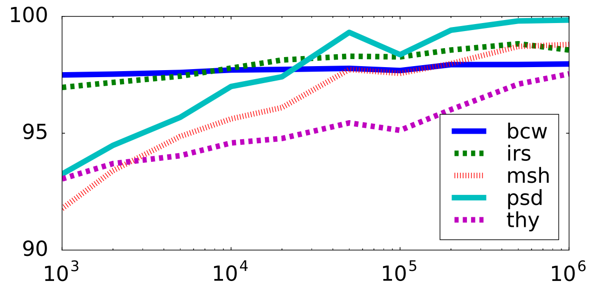
***(a)** DT size: ger, sick, ca, vote, wilt*



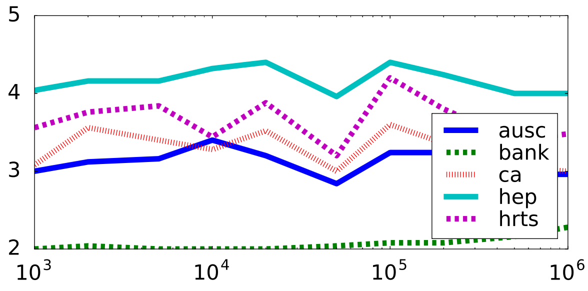
***(b)** DT acc: ger, sick, ca, vote, wilt*



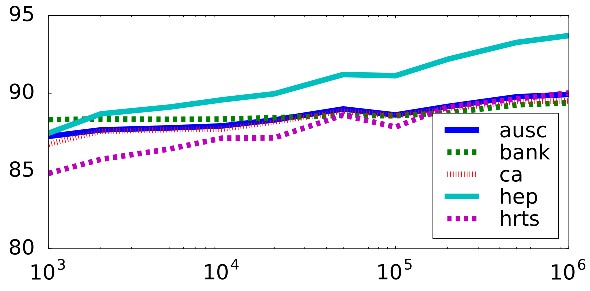
***(c)** DT size: bcw, irs, msh, psd, thy*



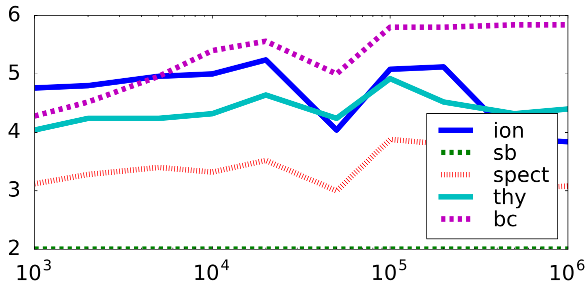
***(d)** DT acc: bcw, irs, msh, psd, thy*



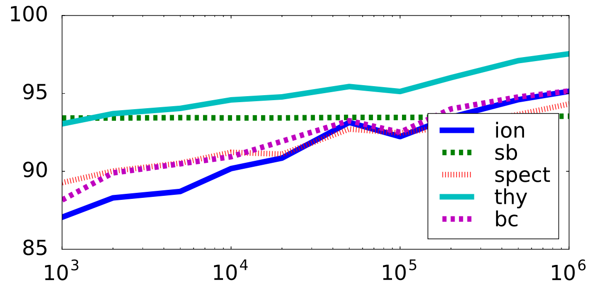
***(e)** DT size: ausc, bank, ca, hep, hrts*



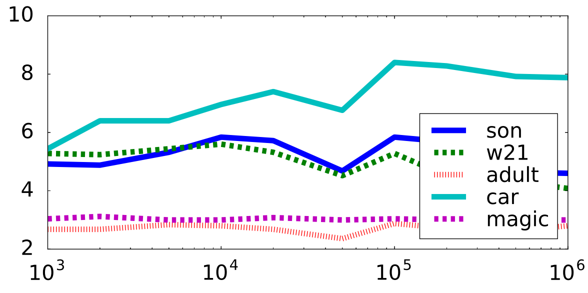
***(f)** DT acc: ausc, bank, ca, hep, hrts*



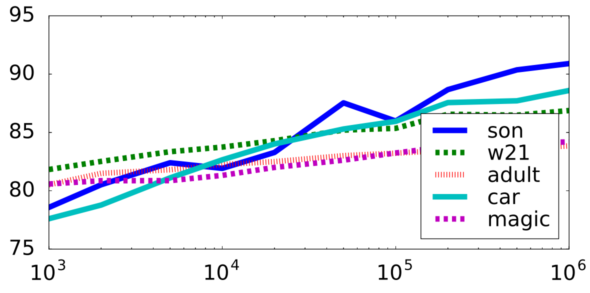
***(g)** DT size: ion, sb, spect, thy, bc*



***(h)** DT acc: ion, sb, spect, thy, bc*

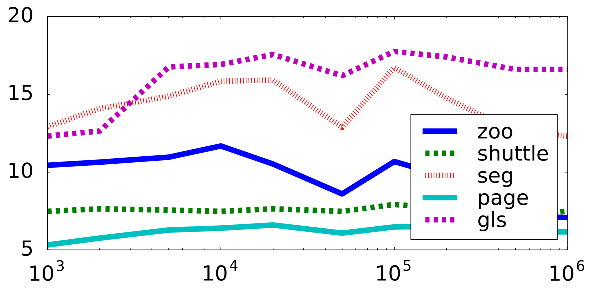


***(i)** DT size: son, w21, adult, car, magic*

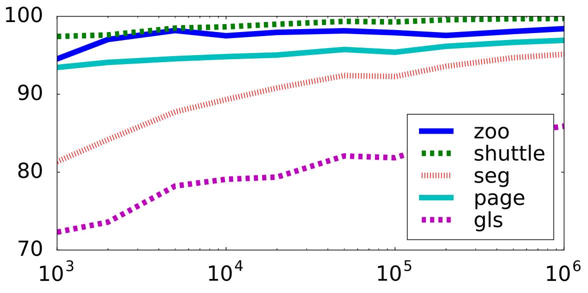


***(j)** DT acc: son, w21, adult, car, magic*

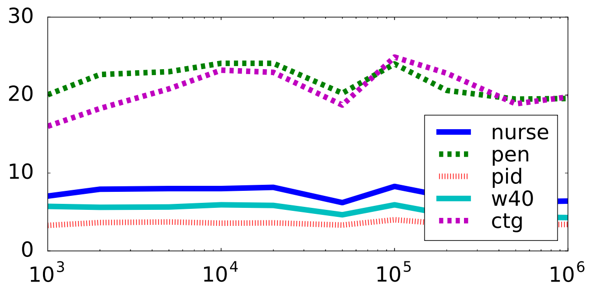
***Figure 3.21:** Dependency of the induced DT sizes and accuracies on the number of iterations the* EFTI *algorithm was run. Datasets 1-25.*



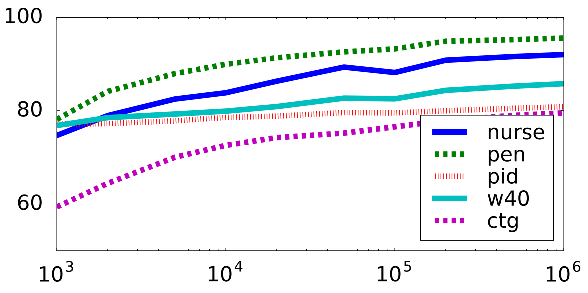
(a) DT size: zoo, shuttle, seg, page, gls



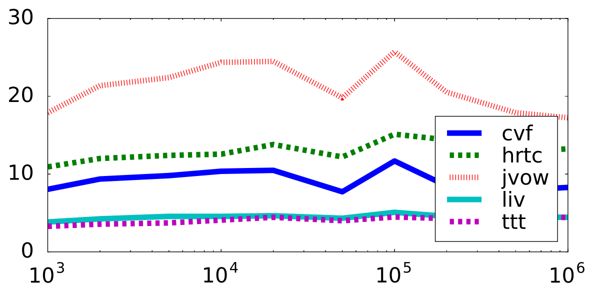
(b) DT acc: zoo, shuttle, seg, page, gls



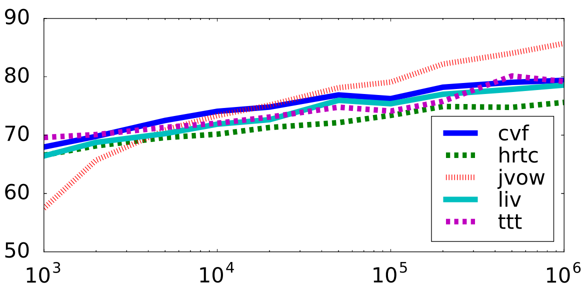
(c) DT size: nurse, pen, pid, w40, ctg



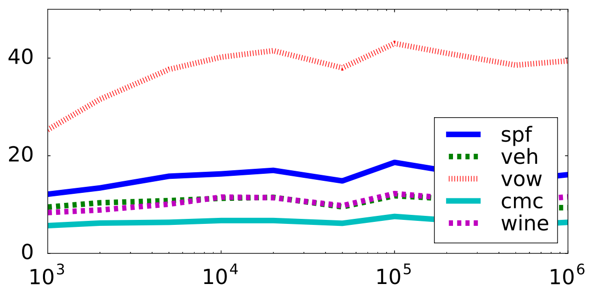
(d) DT acc: nurse, pen, pid, w40, ctg



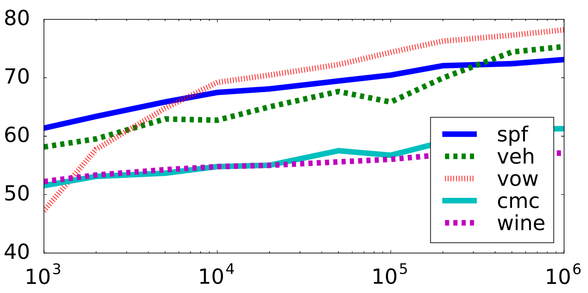
(e) DT size: cvf, hrtc, jvow, liv, ttt



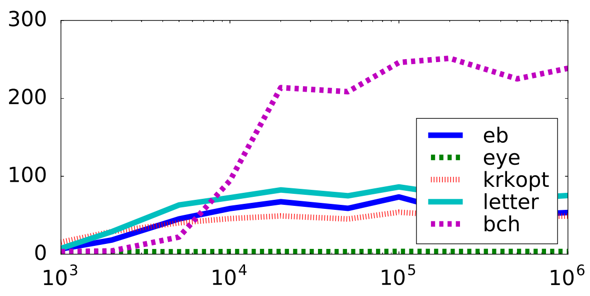
(f) DT acc: cvf, hrtc, jvow, liv, ttt



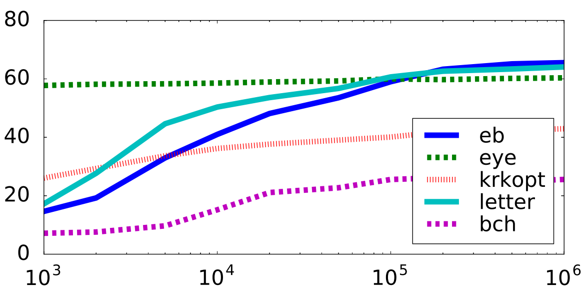
(g) DT size: spf, veh, vow, cmc, wine



(h) DT acc: spf, veh, vow, cmc, wine



(i) DT size: eb, eye, krkopt, letter, bch



(j) DT acc: eb, eye, krkopt, letter, bch

Figure 3.22: *Dependency of the induced DTs on the number of iterations the* EFTI *algorithm was run. Datasets 25-50.*

algorithms, since they tend to create larger, but more accurate DTs, and the "High compression" set was used for the comparison with full DT induction algorithms (namely GaTree and GALE), since they tend to create smaller, but less accurate DTs.

Table 3.10: *Two sets of the parameters set to the* EFTI *algorithm for the comparison experiments*

Approach	K_o	α	β	ρ_0	S_T	p_R
High accuracy	**0.01**	1	0.6	5×10^{-5}	0.05	1×10^{-4}
High compression	**0.2**	1	0.6	5×10^{-5}	0.05	1×10^{-4}

3.5.2.1 CART-LC

The following section presents the results of the comparison between the CART-LC algorithm and the *EFTI* algorithm with the "High accuracy" parameter set. CART-LC is the quickest oblique induction algorithm of the ones used in the experiments, and its induction times are shown in the Table 3.11.

Table 3.11: *The average induction times of the CART-LC algorithm per dataset*

Dataset	Ind. Time [s]	Dataset	Ind. Time [s]	Dataset	Ind. Time [s]
adult	6.14 ± 0.28	hrts	0.01 ± 0.00	shuttle	2.92 ± 0.18
ausc	0.04 ± 0.00	ion	0.03 ± 0.00	sick	0.31 ± 0.02
bank	14.03 ± 0.49	irs	0.00 ± 0.00	son	0.02 ± 0.00
bc	0.01 ± 0.00	jvow	3.03 ± 0.09	spect	0.01 ± 0.00
bch	3.46 ± 0.11	krkopt	4.13 ± 0.14	spf	0.53 ± 0.02
bcw	0.01 ± 0.00	letter	9.10 ± 0.33	thy	0.27 ± 0.02
ca	0.04 ± 0.00	liv	0.01 ± 0.00	ttt	0.04 ± 0.00
car	0.04 ± 0.00	lym	0.01 ± 0.00	veh	0.08 ± 0.01
cmc	0.08 ± 0.01	magic	3.39 ± 0.09	vene	0.00 ± 0.00
ctg	0.45 ± 0.02	msh	0.59 ± 0.06	vote	0.01 ± 0.00
cvf	2.12 ± 0.09	nurse	0.36 ± 0.01	vow	0.10 ± 0.01
eb	6.94 ± 0.11	page	0.59 ± 0.04	w21	1.58 ± 0.07
eye	2.15 ± 0.10	pen	2.59 ± 0.13	w40	2.86 ± 0.08
ger	0.05 ± 0.01	pid	0.03 ± 0.00	wfr	0.99 ± 0.04
gls	0.01 ± 0.00	psd	0.00 ± 0.00	wilt	0.16 ± 0.04
hep	0.01 ± 0.00	sb	0.28 ± 0.04	wine	0.69 ± 0.02
hrtc	0.02 ± 0.00	seg	0.19 ± 0.01	zoo	0.00 ± 0.00

The results of the comparison experiments are displayed side by side in the Table 3.12. The results show that, although the *EFTI* algorithm was not built for time efficiency as its primary objective, it can still readily compete with a fast algorithm such as CART-LC. There are some datasets, such as `car`, `ctg`, `eb`, `eye`, `jvow`, `krkopt`, `letter`, `nurse`, `psd`, `seg`, `vow` and wfr, where *EFTI* significantly underachieved with respect to the DT accuracy. Generally these are the cases which require big DTs, for which the *EFTI* algorithm did not have time in this scenario, or was too constrained by the oversize weight parameter K_o. For all other datasets, the *EFTI* algorithm managed to either produce smaller DTs, or the DTs with increased accuracy by paying a small price in the DT size. For the datasets like: adult, bank, cmc, magic, page, shuttle, sick, spf, ttt, wilt, and wine. *EFTI* managed to compress the DTs up to 20 times (40 in the case of wine dataset), compared to the CART-LC, with the loss in accuracy of only few percent. For the others like ausc, bc, bch, bcw, ca, hrts, liv, pid, w21 and w40, *EFTI* even succeeded in producing more accurate DTs, with their sizes being up to 3 times smaller than the DTs induced by the CART-LC. Finally, for some datasets like: gls, hep, hrtc, irs, lym, son, spect and zoo, *EFTI* created DTs that are 10-20% more accurate, by paying small price in their size, compared to the CART-LC. There are only four datasets, for which the *EFTI* algorithm fitness measure shows poorer combined performance on both fields of accuracy and size: ger, psd, seg and thy.

Table 3.12: *The results of the comparison experiments between the CART-LC algorithm and the* EFTI *algorithm, displayed side by side for different induced DTs' characteristics: accuracy, size and fitness*

	Accuracy		Size		Fitness	
Dataset	CART-LC	EFTI	CART-LC	EFTI	CART-LC	EFTI
adult	85.48	82.45	55.0 ± 17.1	2.6 ± 0.2	-8.66 ± 5.97	0.82 ± 0.00
ausc	84.58	87.92	4.1 ± 1.4	2.9 ± 0.1	0.81 ± 0.04	0.88 ± 0.00
bank	89.75	88.36	42.7 ± 22.2	2.0 ± 0.1	-9.03 ± 9.97	0.88 ± 0.00
bc	88.26	89.09	8.7 ± 2.1	4.2 ± 0.3	0.83 ± 0.04	0.89 ± 0.00
bch	13.25	20.19	259.3 ± 120.4	207.8 ± 7.3	0.09 ± 0.04	0.19 ± 0.00
bcw	93.23	97.55	3.0 ± 0.7	2.2 ± 0.2	0.92 ± 0.02	0.98 ± 0.00
ca	85.54	87.72	6.1 ± 2.3	3.2 ± 0.3	0.76 ± 0.11	0.87 ± 0.00
car	94.72	78.12	36.3 ± 3.7	5.1 ± 0.4	0.28 ± 0.14	0.78 ± 0.01
cmc	53.93	53.46	21.4 ± 14.0	5.9 ± 0.3	-0.29 ± 1.28	0.53 ± 0.01
ctg	81.95	70.04	52.7 ± 11.9	20.5 ± 0.8	0.60 ± 0.11	0.69 ± 0.01
cvf	76.03	73.80	35.6 ± 10.3	8.6 ± 0.5	0.54 ± 0.14	0.74 ± 0.01
eb	65.46	29.19	1818.4 ± 778.8	43.5 ± 3.0	-44.36 ± 28.42	0.29 ± 0.01
eye	83.37	58.62	545.5 ± 73.4	3.1 ± 0.2	-676.92 ± 178.57	0.58 ± 0.00
ger	95.48	91.16	2.8 ± 0.9	3.2 ± 0.3	0.94 ± 0.03	0.91 ± 0.01
gls	66.55	74.15	11.4 ± 2.6	14.1 ± 1.3	0.65 ± 0.04	0.73 ± 0.01
hep	77.29	89.19	2.5 ± 0.4	3.9 ± 0.2	0.77 ± 0.03	0.88 ± 0.01
hrtc	52.66	68.12	6.0 ± 2.5	12.0 ± 0.5	0.52 ± 0.03	0.67 ± 0.01
hrts	76.00	86.50	4.8 ± 2.2	3.3 ± 0.3	0.70 ± 0.08	0.86 ± 0.00
ion	89.71	90.20	4.3 ± 1.2	4.4 ± 0.3	0.87 ± 0.03	0.89 ± 0.01
irs	93.57	97.33	3.2 ± 0.2	3.5 ± 0.3	0.94 ± 0.02	0.97 ± 0.00
jvow	90.64	73.07	233.1 ± 19.7	22.1 ± 1.1	-4.97 ± 0.95	0.71 ± 0.01
krkopt	77.70	33.01	2964.0 ± 98.0	37.0 ± 1.4	-208.72 ± 14.09	0.33 ± 0.00
letter	83.81	49.60	905.7 ± 80.4	72.9 ± 2.4	-9.21 ± 1.66	0.48 ± 0.01

Continued on next page

Table 3.12 – continued from previous page

	Accuracy		Size		Fitness	
Dataset	CART-LC	EFTI	CART-LC	EFTI	CART-LC	EFTI
liv	66.38	68.89	8.8 ± 3.9	4.0 ± 0.3	0.45 ± 0.18	0.68 ± 0.01
lym	74.61	86.22	4.2 ± 1.2	8.6 ± 0.9	0.74 ± 0.04	0.85 ± 0.01
magic	86.12	81.20	65.6 ± 11.2	3.0 ± 0.1	-9.35 ± 3.66	0.81 ± 0.00
msh	99.90	94.41	10.4 ± 0.7	4.3 ± 0.3	0.81 ± 0.03	0.93 ± 0.01
nurse	98.08	70.44	132.4 ± 9.5	8.2 ± 0.7	-5.58 ± 0.96	0.70 ± 0.02
page	96.92	94.74	18.0 ± 6.9	5.9 ± 0.2	0.80 ± 0.16	0.95 ± 0.00
pen	96.64	87.99	98.4 ± 10.8	22.8 ± 1.0	0.15 ± 0.18	0.87 ± 0.00
pid	74.27	77.57	9.1 ± 4.3	3.2 ± 0.2	0.47 ± 0.34	0.77 ± 0.00
psd	100.00	88.28	2.0 ± 0.0	3.3 ± 0.4	1.00 ± 0.00	0.88 ± 0.02
sb	93.17	93.44	3.1 ± 1.1	2.0 ± 0.0	0.91 ± 0.03	0.93 ± 0.00
seg	94.54	85.84	22.1 ± 4.2	13.7 ± 0.7	0.88 ± 0.02	0.85 ± 0.01
shuttle	99.96	96.93	25.0 ± 1.5	7.2 ± 0.3	0.93 ± 0.01	0.97 ± 0.01
sick	97.73	94.03	9.1 ± 2.2	2.1 ± 0.1	0.79 ± 0.12	0.94 ± 0.00
son	71.88	81.29	4.3 ± 1.4	5.0 ± 0.4	0.69 ± 0.05	0.79 ± 0.01
spect	82.64	90.15	3.0 ± 0.9	3.1 ± 0.3	0.82 ± 0.03	0.90 ± 0.00
spf	71.23	66.73	50.3 ± 17.1	14.2 ± 0.8	0.19 ± 0.37	0.66 ± 0.01
thy	98.85	94.32	5.4 ± 1.0	4.1 ± 0.1	0.98 ± 0.01	0.94 ± 0.00
ttt	79.36	71.29	33.8 ± 8.7	3.6 ± 0.3	-2.08 ± 1.24	0.71 ± 0.01
veh	69.20	61.96	23.0 ± 7.7	10.5 ± 0.5	0.39 ± 0.20	0.60 ± 0.01
vene	90.40	92.05	4.8 ± 1.7	4.5 ± 0.3	0.89 ± 0.03	0.92 ± 0.00
vote	93.28	92.94	2.8 ± 0.7	3.2 ± 0.2	0.93 ± 0.01	0.93 ± 0.00
vow	78.18	61.52	63.6 ± 7.4	35.7 ± 0.6	0.58 ± 0.04	0.58 ± 0.01
w21	81.30	84.34	29.4 ± 13.9	4.8 ± 0.3	-0.77 ± 1.44	0.84 ± 0.00
w40	80.66	81.57	20.1 ± 5.1	5.1 ± 0.3	0.41 ± 0.22	0.81 ± 0.00
wfr	98.09	69.32	25.3 ± 3.1	9.0 ± 0.4	0.67 ± 0.09	0.68 ± 0.01
wilt	98.01	94.61	12.7 ± 3.1	2.0 ± 0.0	0.57 ± 0.25	0.95 ± 0.00
wine	57.41	54.39	414.4 ± 93.3	9.5 ± 0.4	-25.04 ± 7.10	0.54 ± 0.00
zoo	85.31	94.46	6.0 ± 0.6	9.9 ± 0.8	0.85 ± 0.03	0.94 ± 0.01

3.5.2.2 OC1-ES

The following section presents the results of the comparison between the OC1-ES algorithm and the *EFTI* algorithm with the "High accuracy" parameter set. The OC1-ES is the second fastest algorithm among the ones used in the experiments, and needs on average (it varies with the dataset) twice as much time for the induction as CART-LC, and its induction times are shown in the Table 3.13. However, for some of the more complex datasets, the average induction times were similar to the CART-LC's (`jvow`, `pen`, `w40`), and some were even shorter (`bch`, `letter`). OC1-ES was run with the default setting of 1000 iterations per node. Several experiments were made to test whether higher iteration counts (2000, 5000, 10000 and 50000) would increase the quality of the solutions, but no benefits were observed over the defaults.

Table 3.13: *The average induction times of the OC1-ES algorithm per dataset*

Dataset	Ind. Time [s]	Dataset	Ind. Time [s]	Dataset	Ind. Time [s]
adult	15.28 ± 0.12	hrts	0.08 ± 0.01	shuttle	3.50 ± 0.18
ausc	0.21 ± 0.01	ion	0.17 ± 0.02	sick	0.55 ± 0.02
bank	20.34 ± 0.30	irs	0.01 ± 0.00	son	0.04 ± 0.00
bc	0.09 ± 0.00	jvow	3.21 ± 0.04	spect	0.06 ± 0.01
bch	2.55 ± 0.01	krkopt	10.39 ± 0.06	spf	0.98 ± 0.03
bcw	0.07 ± 0.00	letter	7.06 ± 0.04	thy	0.35 ± 0.02
ca	0.21 ± 0.01	liv	0.11 ± 0.01	ttt	0.29 ± 0.01
car	0.20 ± 0.01	lym	0.05 ± 0.01	veh	0.35 ± 0.01
cmc	0.58 ± 0.01	magic	7.56 ± 0.07	vene	0.03 ± 0.00
ctg	1.03 ± 0.02	msh	0.88 ± 0.06	vote	0.06 ± 0.00
cvf	3.88 ± 0.05	nurse	1.43 ± 0.03	vow	0.33 ± 0.01
eb	18.24 ± 0.15	page	0.89 ± 0.02	w21	2.42 ± 0.04
eye	6.30 ± 0.07	pen	2.59 ± 0.06	w40	3.71 ± 0.09
ger	0.16 ± 0.01	pid	0.23 ± 0.01	wfr	0.86 ± 0.03
gls	0.07 ± 0.00	psd	0.00 ± 0.00	wilt	0.26 ± 0.01
hep	0.05 ± 0.01	sb	0.93 ± 0.03	wine	2.36 ± 0.03
hrtc	0.10 ± 0.01	seg	0.39 ± 0.01	zoo	0.02 ± 0.00

The results of the comparison experiments are displayed side by side in the Table 3.14. The results show, that OC1-ES has very similar performance with respect to the DT accuracy to the CART-LC for most of the datasets, with a tendency to induce larger DTs. On the other hand, the *EFTI* algorithm managed only slightly to improve on DT accuracy, where it was given more time. Hence, the discussion about the results from the Section 3.5.2.1, can be applied almost verbatim to the results from the Table 3.14. The only differences stem from the fact that OC1-ES produces even larger DTs, hence the compresion ratios of the *EFTI* algorithm are even higher. This resulted in *EFTI* now producing smaller DTs for the datasets: `ger`, `hep`, `son` and `spect` as opposed to the comparison results with CART-LC, while still retaining an advantage in the accuracy, and even increasing it.

Table 3.14: *The results of the comparison experiments between the OC1-ES algorihtm and the* EFTI *algorithm, displayed side by side for different induced DTs' characteristics: accuracy, size and fitness*

	Accuracy		Size		Fitness	
Dataset	OC1-ES	EFTI	OC1-ES	EFTI	OC1-ES	EFTI
adult	85.59	82.77	54.9 ± 16.4	2.4 ± 0.2	-8.35 ± 5.09	0.83 ± 0.00
ausc	85.59	88.56	5.2 ± 2.9	2.8 ± 0.2	0.73 ± 0.15	0.88 ± 0.00
bank	90.10	88.40	75.7 ± 19.8	2.0 ± 0.0	-16.30 ± 10.19	0.88 ± 0.00
bc	85.09	91.43	13.9 ± 4.0	4.8 ± 0.3	0.65 ± 0.12	0.91 ± 0.00
bch	14.16	17.63	379.1 ± 130.1	141.8 ± 12.7	0.07 ± 0.06	0.17 ± 0.00

Continued on next page

Table 3.14 – continued from previous page

	Accuracy		Size		Fitness	
Dataset	OC1-ES	EFTI	OC1-ES	EFTI	OC1-ES	EFTI
bcw	92.23	97.71	5.6 ± 1.9	2.2 ± 0.2	0.84 ± 0.06	0.98 ± 0.00
ca	85.25	88.47	8.7 ± 2.8	3.0 ± 0.1	0.67 ± 0.14	0.88 ± 0.00
car	95.21	82.79	47.3 ± 5.6	6.0 ± 0.4	-0.28 ± 0.29	0.83 ± 0.01
cmc	53.53	56.00	34.0 ± 18.2	6.1 ± 0.5	-1.06 ± 1.47	0.55 ± 0.01
ctg	82.15	72.79	59.6 ± 13.8	19.8 ± 1.0	0.53 ± 0.14	0.72 ± 0.01
cvf	72.61	74.55	219.8 ± 74.6	8.4 ± 0.4	-10.76 ± 7.74	0.75 ± 0.01
eb	65.48	41.23	2278.1 ± 1009.6	57.6 ± 3.3	-73.06 ± 45.38	0.41 ± 0.01
eye	83.40	58.98	616.7 ± 86.3	3.2 ± 0.2	-874.74 ± 229.75	0.59 ± 0.00
ger	96.16	92.05	4.7 ± 1.3	3.0 ± 0.3	0.92 ± 0.04	0.92 ± 0.01
gls	64.91	78.90	12.0 ± 3.5	15.5 ± 0.8	0.63 ± 0.04	0.77 ± 0.01
hep	78.58	90.97	4.5 ± 1.8	4.1 ± 0.2	0.74 ± 0.06	0.90 ± 0.01
hrtc	52.46	70.77	11.7 ± 6.1	12.1 ± 0.2	0.48 ± 0.06	0.69 ± 0.00
hrts	78.00	88.06	7.6 ± 2.7	3.3 ± 0.2	0.64 ± 0.12	0.88 ± 0.00
ion	88.51	91.19	5.7 ± 2.0	4.3 ± 0.3	0.81 ± 0.07	0.90 ± 0.01
irs	94.28	97.55	3.6 ± 0.5	3.4 ± 0.2	0.94 ± 0.02	0.97 ± 0.00
jvow	88.10	71.88	412.6 ± 30.3	22.2 ± 1.0	-17.41 ± 2.45	0.70 ± 0.01
krkopt	76.25	35.42	3600.7 ± 133.2	42.8 ± 1.3	-303.58 ± 22.70	0.35 ± 0.00
letter	84.40	48.05	1255.4 ± 81.7	71.9 ± 1.6	-18.51 ± 2.28	0.47 ± 0.01
liv	64.99	74.46	10.2 ± 4.3	4.2 ± 0.3	0.37 ± 0.26	0.73 ± 0.01
lym	71.79	89.35	7.8 ± 2.7	9.2 ± 0.7	0.69 ± 0.02	0.88 ± 0.01
magic	85.35	81.93	97.9 ± 24.0	3.1 ± 0.1	-25.68 ± 19.45	0.82 ± 0.00
msh	99.83	94.71	19.6 ± 1.6	4.0 ± 0.3	0.19 ± 0.14	0.94 ± 0.01
nurse	97.65	80.42	231.6 ± 13.4	7.6 ± 0.5	-19.47 ± 2.33	0.80 ± 0.01
page	97.10	94.94	23.9 ± 6.0	5.8 ± 0.3	0.75 ± 0.14	0.95 ± 0.00
pen	95.84	88.04	196.8 ± 15.7	23.4 ± 0.9	-2.52 ± 0.55	0.86 ± 0.01
pid	73.30	79.34	7.0 ± 2.4	3.2 ± 0.2	0.63 ± 0.09	0.79 ± 0.00
psd	100.00	89.07	2.0 ± 0.0	3.0 ± 0.4	1.00 ± 0.00	0.89 ± 0.02
sb	93.28	93.46	4.1 ± 3.8	2.0 ± 0.0	0.73 ± 0.40	0.93 ± 0.00
seg	94.36	87.66	31.6 ± 5.5	14.0 ± 0.7	0.79 ± 0.06	0.87 ± 0.01
shuttle	99.95	97.77	24.8 ± 2.6	7.4 ± 0.2	0.93 ± 0.02	0.98 ± 0.00
sick	98.53	93.94	13.9 ± 2.5	2.0 ± 0.1	0.55 ± 0.17	0.94 ± 0.00
son	70.28	83.75	6.5 ± 1.9	5.2 ± 0.3	0.63 ± 0.05	0.81 ± 0.01
spect	86.35	91.75	5.4 ± 2.9	3.1 ± 0.1	0.75 ± 0.13	0.91 ± 0.00
spf	72.83	68.05	80.0 ± 19.7	15.1 ± 0.9	-0.38 ± 0.60	0.67 ± 0.00
thy	99.24	94.23	7.1 ± 1.0	4.2 ± 0.2	0.98 ± 0.01	0.94 ± 0.00
ttt	81.27	73.84	48.7 ± 10.1	3.9 ± 0.3	-4.82 ± 2.06	0.73 ± 0.00
veh	68.79	65.35	37.6 ± 12.3	10.4 ± 0.4	-0.20 ± 0.51	0.64 ± 0.01
vene	88.20	93.13	5.4 ± 1.4	4.5 ± 0.2	0.86 ± 0.02	0.93 ± 0.00

Continued on next page

Table 3.14 – continued from previous page

Dataset	Accuracy		Size		Fitness	
	OC1-ES	EFTI	OC1-ES	EFTI	OC1-ES	EFTI
vote	94.41	94.64	3.8 ± 1.5	3.2 ± 0.2	0.91 ± 0.04	0.94 ± 0.00
vow	76.81	66.58	90.2 ± 7.3	37.6 ± 1.1	0.35 ± 0.06	0.63 ± 0.01
w21	77.54	84.59	75.0 ± 19.5	4.5 ± 0.2	-5.56 ± 3.27	0.84 ± 0.00
w40	76.87	81.86	67.0 ± 19.2	5.0 ± 0.2	-4.51 ± 3.02	0.81 ± 0.00
wfr	99.36	68.85	19.5 ± 1.9	9.5 ± 0.6	0.83 ± 0.04	0.67 ± 0.01
wilt	97.83	94.61	17.5 ± 3.6	2.0 ± 0.0	0.21 ± 0.40	0.95 ± 0.00
wine	56.82	55.15	498.0 ± 105.5	9.8 ± 0.4	-35.25 ± 10.47	0.55 ± 0.00
zoo	77.68	97.58	4.5 ± 0.9	10.2 ± 0.8	0.78 ± 0.07	0.97 ± 0.01

3.5.2.3 OC1-SA

The following section presents the results of the comparison between the OC1-SA algorithm and the *EFTI* algorithm with the "High accuracy" parameter set. OC1-SA takes even more time than OC1-ES to run, and is 10 to 20 times slower than CART-LC. Its induction times are shown in the Table 3.15. OC1-SA was run with the default setting of 20 temperature values with 50 iterations for each of them per node. Several experiments were made to test whether different number of temperature values (10, 20, 40 and 80) and iteration counts (25, 50 and 100) would increase the quality of the solutions, but no benefits were observed over the defaults.

Table 3.15: *The average induction times of the OC1-SA algorithm per dataset*

Dataset	Ind. Time [s]	Dataset	Ind. Time [s]	Dataset	Ind. Time [s]
adult	131.36 ± 1.44	hrts	0.41 ± 0.02	shuttle	30.88 ± 1.88
ausc	1.25 ± 0.05	ion	2.77 ± 0.25	sick	11.23 ± 0.54
bank	234.89 ± 5.59	irs	0.02 ± 0.00	son	1.00 ± 0.01
bc	0.14 ± 0.01	jvow	25.73 ± 0.36	spect	0.56 ± 0.05
bch	19.63 ± 0.13	krkopt	32.41 ± 0.29	spf	12.37 ± 0.34
bcw	0.26 ± 0.02	letter	69.46 ± 0.49	thy	6.80 ± 0.59
ca	1.41 ± 0.07	liv	0.24 ± 0.01	ttt	1.13 ± 0.04
car	0.55 ± 0.01	lym	0.34 ± 0.04	veh	2.66 ± 0.10
cmc	2.17 ± 0.04	magic	47.22 ± 0.65	vene	0.02 ± 0.00
ctg	10.89 ± 0.31	msh	15.61 ± 0.82	vote	0.40 ± 0.02
cvf	33.21 ± 0.39	nurse	7.65 ± 0.18	vow	1.37 ± 0.03
eb	40.22 ± 0.44	page	6.68 ± 0.26	w21	27.99 ± 0.55
eye	50.82 ± 0.48	pen	27.34 ± 0.34	w40	89.97 ± 1.65
ger	2.24 ± 0.08	pid	0.76 ± 0.02	wfr	14.83 ± 0.51
gls	0.24 ± 0.01	psd	0.00 ± 0.00	wilt	0.89 ± 0.03
hep	0.36 ± 0.04	sb	9.36 ± 0.36	wine	12.06 ± 0.10
hrtc	0.51 ± 0.03	seg	4.10 ± 0.07	zoo	0.11 ± 0.00

The results of the comparison experiments are displayed side by side in the Table 3.14. The results show, that OC1-SA produced very similar results in terms of accuracy to OC1-ES, and tended to produce somewhat smaller DTs. Nevertheless, the conclusions for the comparison results are almost identical to the ones discussed for OC1-ES in the Section 3.5.2.2

Table 3.16: *The results of the comparison experiments between the OC1-SA algorihtm and the* EFTI *algorithm, displayed side by side for different induced DTs' characteristics: accuracy, size and fitness*

	Accuracy		Size		Fitness	
Dataset	OC1-SA	EFTI	OC1-SA	EFTI	OC1-SA	EFTI
adult	85.55	83.34	89.5 ± 30.2	2.5 ± 0.2	-26.34 ± 20.87	0.83 ± 0.00
ausc	85.28	89.44	4.6 ± 2.4	2.8 ± 0.2	0.77 ± 0.10	0.89 ± 0.00
bank	89.96	88.94	72.3 ± 23.6	2.2 ± 0.2	-17.25 ± 12.86	0.89 ± 0.00
bc	86.05	91.98	16.0 ± 4.5	4.8 ± 0.3	0.59 ± 0.15	0.92 ± 0.00
bch	13.88	23.09	395.4 ± 150.5	207.7 ± 1.2	0.05 ± 0.08	0.22 ± 0.00
bcw	92.44	97.77	5.8 ± 1.7	2.2 ± 0.2	0.85 ± 0.06	0.98 ± 0.00
ca	84.87	89.18	9.4 ± 3.9	3.0 ± 0.1	0.56 ± 0.32	0.89 ± 0.00
car	94.32	84.00	43.0 ± 4.9	6.4 ± 0.3	-0.04 ± 0.23	0.84 ± 0.00
cmc	52.10	58.83	35.7 ± 13.2	5.7 ± 0.3	-0.67 ± 0.70	0.58 ± 0.01
ctg	82.24	77.60	54.3 ± 15.5	18.8 ± 0.8	0.55 ± 0.17	0.77 ± 0.00
cvf	74.21	78.00	199.9 ± 63.5	7.6 ± 0.2	-8.34 ± 7.06	0.78 ± 0.00
eb	65.47	48.02	1545.2 ± 740.6	62.4 ± 3.5	-36.15 ± 27.60	0.48 ± 0.02
eye	83.52	59.67	560.4 ± 95.3	3.2 ± 0.2	-759.46 ± 227.15	0.59 ± 0.00
ger	96.32	96.51	4.9 ± 1.4	2.8 ± 0.2	0.92 ± 0.04	0.96 ± 0.00
gls	64.67	81.57	15.3 ± 3.5	15.7 ± 0.8	0.62 ± 0.03	0.79 ± 0.01
hep	77.42	92.26	3.6 ± 1.2	4.0 ± 0.1	0.75 ± 0.04	0.91 ± 0.01
hrtc	54.04	73.03	6.4 ± 4.0	12.3 ± 0.4	0.52 ± 0.04	0.71 ± 0.01
hrts	75.19	88.80	5.9 ± 2.8	3.2 ± 0.2	0.64 ± 0.13	0.88 ± 0.00
ion	88.97	94.51	4.9 ± 1.4	3.7 ± 0.3	0.85 ± 0.05	0.94 ± 0.01
irs	93.60	97.81	3.4 ± 0.4	3.7 ± 0.3	0.93 ± 0.02	0.98 ± 0.00
jvow	87.93	80.29	404.4 ± 31.9	18.6 ± 0.8	-16.72 ± 2.61	0.79 ± 0.01
krkopt	75.17	37.86	3504.4 ± 148.1	45.0 ± 1.1	-283.86 ± 23.75	0.37 ± 0.00
letter	85.08	59.04	1260.7 ± 79.4	73.1 ± 1.6	-18.79 ± 2.15	0.57 ± 0.01
liv	65.51	76.13	10.1 ± 4.4	4.3 ± 0.3	0.36 ± 0.26	0.75 ± 0.00
lym	70.43	91.11	5.1 ± 1.9	9.0 ± 0.7	0.69 ± 0.03	0.90 ± 0.01
magic	85.17	83.60	111.7 ± 35.1	3.0 ± 0.0	-39.36 ± 27.47	0.83 ± 0.00
msh	99.91	98.16	21.3 ± 1.4	3.7 ± 0.2	0.04 ± 0.14	0.97 ± 0.00
nurse	96.66	88.34	223.1 ± 24.6	6.6 ± 0.5	-18.77 ± 4.03	0.88 ± 0.01
page	96.85	95.92	22.6 ± 5.9	6.0 ± 0.2	0.77 ± 0.12	0.96 ± 0.00
pen	95.74	93.61	188.3 ± 14.1	19.6 ± 0.7	-2.19 ± 0.47	0.93 ± 0.00
pid	73.90	79.63	10.0 ± 5.7	3.2 ± 0.2	0.30 ± 0.52	0.79 ± 0.00
psd	100.00	79.58	2.0 ± 0.0	3.1 ± 0.3	1.00 ± 0.00	0.79 ± 0.02

Continued on next page

Table 3.16 – continued from previous page

	Accuracy		Size		Fitness	
Dataset	OC1-SA	EFTI	OC1-SA	EFTI	OC1-SA	EFTI
sb	93.34	93.50	2.8 ± 1.1	2.0 ± 0.0	0.91 ± 0.03	0.93 ± 0.00
seg	95.35	92.56	37.9 ± 5.1	13.2 ± 0.7	0.74 ± 0.06	0.92 ± 0.01
shuttle	99.95	98.94	24.2 ± 2.7	7.3 ± 0.2	0.93 ± 0.02	0.99 ± 0.00
sick	98.28	95.60	14.8 ± 3.4	2.7 ± 0.2	0.42 ± 0.23	0.95 ± 0.00
son	70.38	89.42	6.1 ± 1.9	4.3 ± 0.4	0.64 ± 0.06	0.88 ± 0.01
spect	87.47	92.95	3.4 ± 1.7	3.0 ± 0.1	0.84 ± 0.07	0.93 ± 0.00
spf	71.94	72.15	55.6 ± 16.9	14.5 ± 0.9	0.13 ± 0.40	0.71 ± 0.00
thy	99.22	95.81	7.9 ± 1.0	4.3 ± 0.2	0.98 ± 0.01	0.96 ± 0.00
ttt	77.68	75.72	43.4 ± 11.1	4.0 ± 0.2	-4.00 ± 2.18	0.75 ± 0.01
veh	68.20	71.52	32.8 ± 9.3	9.6 ± 0.5	0.10 ± 0.33	0.70 ± 0.01
vene	89.33	92.99	4.5 ± 1.4	4.6 ± 0.2	0.88 ± 0.02	0.93 ± 0.00
vote	94.16	96.41	4.1 ± 1.4	3.0 ± 0.1	0.90 ± 0.03	0.96 ± 0.00
vow	76.63	72.00	90.8 ± 6.9	37.2 ± 1.1	0.34 ± 0.06	0.68 ± 0.01
w21	77.12	86.28	59.3 ± 13.8	4.2 ± 0.2	-2.87 ± 2.06	0.86 ± 0.00
w40	76.25	85.51	55.4 ± 19.6	4.3 ± 0.2	-3.37 ± 3.23	0.85 ± 0.00
wfr	99.35	75.88	20.3 ± 2.2	9.0 ± 0.4	0.81 ± 0.05	0.75 ± 0.01
wilt	97.86	94.60	18.1 ± 3.3	2.0 ± 0.0	0.19 ± 0.33	0.95 ± 0.00
wine	57.84	56.26	504.3 ± 105.5	10.3 ± 0.4	-36.42 ± 10.01	0.56 ± 0.00
zoo	83.10	98.10	6.0 ± 0.9	8.5 ± 0.4	0.83 ± 0.06	0.98 ± 0.01

3.5.2.4 OC1

The following section presents the results of the comparison between the OC1 algorithm and the *EFTI* algorithm with the “High accuracy” parameter set. OC1 takes similar time to run as OC1-SA does, which is 10 to 20 times slower than CART-LC. Its induction times are shown in the Table 3.17.

Table 3.17: *The average induction times of the OC1 algorithm per dataset*

Dataset	Ind. Time [s]	Dataset	Ind. Time [s]	Dataset	Ind. Time [s]
adult	69.24 ± 1.65	hrts	0.08 ± 0.00	shuttle	43.84 ± 2.91
ausc	0.45 ± 0.02	ion	0.24 ± 0.02	sick	3.09 ± 0.22
bank	208.78 ± 123.04	irs	0.02 ± 0.00	son	0.09 ± 0.00
bc	0.16 ± 0.01	jvow	28.23 ± 0.58	spect	0.04 ± 0.00
bch	23.33 ± 0.32	krkopt	72.34 ± 0.55	spf	3.89 ± 0.17
bcw	0.14 ± 0.01	letter	127.33 ± 1.38	thy	2.07 ± 0.25
ca	0.47 ± 0.02	liv	0.12 ± 0.01	ttt	0.58 ± 0.04
car	0.71 ± 0.04	lym	0.03 ± 0.00	veh	0.81 ± 0.02
cmc	0.91 ± 0.02	magic	49.98 ± 1.44	vene	0.04 ± 0.00
ctg	3.88 ± 0.10	msh	7.05 ± 0.60	vote	0.10 ± 0.01
cvf	15.92 ± 0.42	nurse	11.80 ± 0.30	vow	1.06 ± 0.02
eb	127.43 ± 2.64	page	7.66 ± 0.45	w21	18.40 ± 0.54
eye	15.61 ± 0.26	pen	24.50 ± 0.40	w40	28.54 ± 0.68
ger	0.32 ± 0.03	pid	0.37 ± 0.02	wfr	11.70 ± 0.41
gls	0.08 ± 0.00	psd	0.00 ± 0.00	wilt	1.56 ± 0.09
hep	0.04 ± 0.00	sb	2.03 ± 0.15	wine	8.08 ± 0.15
hrtc	0.12 ± 0.01	seg	1.93 ± 0.07	zoo	0.01 ± 0.00

The results of the comparison experiments are displayed side by side in the Table 3.14. The results show, that OC1 has very similar performance with respect to the DT accuracy to the CART-LC, but has a tendency to induce smaller DTs. However, the *EFTI* algorithm had a significant advantage over CART-LC when it comes to the induced DT size, and this remains true when compared to OC1 as well. This means that the discussion about the results from the Section 3.5.2.1, remains valid for the results from the Table 3.18 too. The differences between results of comparisons with CART-LC and OC1 arise mainly because in case of comparison with OC1, *EFTI* had 10 to 20 times more time for the evolution, hence average accuracies have significantly improved for some of the datasets like: `bch`, `car`, `cmc`, `ctg`, `eb`, `jvow`, `nurse`, `veh`, `vow` and `wfr`, while OC1 brought no significant improvement to the accuracies over CART-LC.

Table 3.18: *The results of the comparison experiments between the OC1 algorihtm and the* EFTI *algorithm, displayed side by side for different induced DTs' characteristics: accuracy, size and fitness*

	Accuracy		Size		Fitness	
Dataset	OC1	EFTI	OC1	EFTI	OC1	EFTI
adult	85.22	83.13	33.5 ± 8.8	2.4 ± 0.2	-2.18 ± 1.54	0.83 ± 0.00
ausc	83.48	88.92	4.7 ± 2.0	2.8 ± 0.2	0.78 ± 0.07	0.89 ± 0.00
bank	89.54	88.99	17.0 ± 5.7	2.2 ± 0.2	-0.02 ± 0.87	0.89 ± 0.00
bc	91.94	92.20	8.4 ± 1.8	5.0 ± 0.3	0.87 ± 0.03	0.92 ± 0.00

Continued on next page

Table 3.18 – continued from previous page

Dataset	Accuracy		Size		Fitness	
	OC1	EFTI	OC1	EFTI	OC1	EFTI
bch	12.46	23.17	285.5 ± 142.2	206.9 ± 1.1	0.07 ± 0.06	0.22 ± 0.00
bcw	93.86	97.77	3.1 ± 0.9	2.2 ± 0.2	0.93 ± 0.02	0.98 ± 0.00
ca	84.14	88.87	3.2 ± 0.7	3.0 ± 0.1	0.83 ± 0.01	0.89 ± 0.00
car	93.16	84.35	27.8 ± 3.6	6.3 ± 0.4	0.56 ± 0.10	0.84 ± 0.00
cmc	53.05	57.38	25.1 ± 12.6	6.0 ± 0.3	-0.21 ± 0.75	0.57 ± 0.01
ctg	79.88	75.53	28.8 ± 9.0	18.8 ± 0.6	0.73 ± 0.06	0.75 ± 0.01
cvf	75.80	77.52	37.0 ± 9.5	7.9 ± 0.4	0.54 ± 0.12	0.77 ± 0.00
eb	65.49	56.69	1013.5 ± 587.5	55.3 ± 1.9	-19.21 ± 19.29	0.56 ± 0.01
eye	84.72	59.11	430.8 ± 58.2	3.1 ± 0.1	-429.51 ± 117.77	0.59 ± 0.00
ger	93.96	93.96	4.0 ± 1.4	2.8 ± 0.2	0.91 ± 0.04	0.94 ± 0.01
gls	62.23	78.92	9.8 ± 2.4	16.3 ± 0.6	0.61 ± 0.03	0.77 ± 0.01
hep	77.29	90.74	3.8 ± 1.2	3.9 ± 0.2	0.75 ± 0.04	0.90 ± 0.01
hrtc	52.71	70.93	7.2 ± 3.5	12.2 ± 0.4	0.51 ± 0.03	0.69 ± 0.01
hrts	79.41	88.07	2.9 ± 1.0	3.2 ± 0.2	0.78 ± 0.03	0.88 ± 0.00
ion	86.86	92.08	4.6 ± 1.0	4.2 ± 0.3	0.84 ± 0.02	0.91 ± 0.01
irs	96.16	97.81	3.0 ± 0.1	3.8 ± 0.3	0.96 ± 0.01	0.98 ± 0.00
jvow	90.86	79.19	232.9 ± 21.9	20.0 ± 1.0	-5.03 ± 0.95	0.78 ± 0.01
krkopt	71.41	39.09	2738.3 ± 97.4	45.9 ± 1.4	-163.41 ± 11.48	0.38 ± 0.00
letter	82.10	60.01	882.8 ± 91.1	70.0 ± 1.8	-8.66 ± 1.80	0.58 ± 0.01
liv	65.68	74.45	7.4 ± 3.5	3.9 ± 0.3	0.51 ± 0.17	0.74 ± 0.01
lym	77.08	87.76	3.3 ± 0.8	8.8 ± 0.7	0.77 ± 0.02	0.86 ± 0.01
magic	86.22	83.34	66.2 ± 23.5	3.0 ± 0.0	-14.64 ± 13.71	0.83 ± 0.00
msh	99.73	97.43	10.4 ± 1.0	3.8 ± 0.2	0.81 ± 0.04	0.97 ± 0.00
nurse	95.73	88.52	120.1 ± 17.7	6.3 ± 0.4	-4.79 ± 1.72	0.88 ± 0.01
page	96.85	95.93	14.6 ± 3.0	6.1 ± 0.1	0.91 ± 0.03	0.96 ± 0.00
pen	96.49	93.05	80.4 ± 9.4	20.7 ± 0.8	0.44 ± 0.13	0.92 ± 0.00
pid	73.44	79.46	9.2 ± 3.8	3.1 ± 0.1	0.50 ± 0.22	0.79 ± 0.00
psd	100.00	88.46	2.0 ± 0.0	2.9 ± 0.4	1.00 ± 0.00	0.88 ± 0.02
sb	93.32	93.49	2.2 ± 0.3	2.0 ± 0.0	0.93 ± 0.00	0.93 ± 0.00
seg	93.73	90.82	21.1 ± 3.9	13.8 ± 0.8	0.88 ± 0.03	0.90 ± 0.01
shuttle	99.94	98.97	27.9 ± 2.9	7.3 ± 0.2	0.90 ± 0.02	0.99 ± 0.00
sick	96.57	94.85	11.9 ± 4.1	2.4 ± 0.2	0.50 ± 0.35	0.95 ± 0.00
son	69.59	84.62	3.6 ± 1.1	4.8 ± 0.2	0.68 ± 0.03	0.83 ± 0.01
spect	81.75	91.36	3.5 ± 1.2	3.2 ± 0.2	0.80 ± 0.04	0.91 ± 0.00
spf	69.53	70.06	43.4 ± 16.9	15.0 ± 0.5	0.28 ± 0.32	0.69 ± 0.00
thy	98.48	95.21	4.9 ± 0.5	4.3 ± 0.2	0.98 ± 0.00	0.95 ± 0.00
ttt	75.68	75.01	20.5 ± 7.1	4.0 ± 0.2	-0.44 ± 0.72	0.74 ± 0.01
veh	69.84	67.64	28.8 ± 7.9	9.4 ± 0.4	0.28 ± 0.20	0.66 ± 0.01

Continued on next page

Table 3.18 – continued from previous page

	Accuracy		Size		Fitness	
Dataset	OC1	EFTI	OC1	EFTI	OC1	EFTI
vene	89.40	93.32	4.6 ± 1.1	4.5 ± 0.2	0.88 ± 0.02	0.93 ± 0.00
vote	92.29	95.60	2.7 ± 0.7	3.1 ± 0.1	0.92 ± 0.01	0.95 ± 0.00
vow	78.20	71.33	48.5 ± 5.9	37.3 ± 1.1	0.68 ± 0.03	0.67 ± 0.01
w21	81.99	85.88	17.9 ± 5.4	4.4 ± 0.2	0.46 ± 0.29	0.86 ± 0.00
w40	80.89	84.36	16.0 ± 3.4	4.5 ± 0.2	0.60 ± 0.11	0.84 ± 0.00
wfr	97.40	75.79	23.8 ± 3.5	9.0 ± 0.4	0.70 ± 0.09	0.75 ± 0.01
wilt	97.93	94.60	13.6 ± 4.5	2.0 ± 0.0	0.37 ± 0.58	0.95 ± 0.00
wine	57.17	56.06	352.6 ± 89.3	9.9 ± 0.5	-18.97 ± 6.32	0.56 ± 0.00
zoo	82.44	96.95	5.6 ± 0.5	10.2 ± 0.7	0.82 ± 0.04	0.97 ± 0.01

3.5.2.5 NODT

The following section presents the results of the comparison between the NODT algorithm and the *EFTI* algorithm with the "High accuracy" parameter set. NODT was run with the default settings:

- Number of iterations: 100000
- Search probability: 0
- Percentage of available data used as validation set: 30%,
- Percentage of mutated bits: 10%

Table 3.19: *The average induction times of the NODT algorithm per dataset*

Dataset	Ind. Time [s]	Dataset	Ind. Time [s]	Dataset	Ind. Time [s]
adult	1463.34 ± 46.73	hrts	3.29 ± 0.16	shuttle	1619.40 ± 46.47
ausc	10.47 ± 0.38	ion	4.45 ± 0.33	sick	51.20 ± 2.63
bank	2291.70 ± 68.97	irs	0.98 ± 0.07	son	9.57 ± 0.49
bc	6.18 ± 0.26	jvow	99.46 ± 1.90	spect	1.96 ± 0.17
bch	1713.47 ± 16.41	krkopt	2260.87 ± 13.85	spf	87.37 ± 1.01
bcw	3.51 ± 0.25	letter	976.72 ± 19.54	thy	79.74 ± 2.55
ca	10.86 ± 0.52	liv	6.41 ± 0.27	ttt	18.70 ± 0.79
car	22.22 ± 0.70	lym	5.48 ± 0.22	veh	39.83 ± 0.74
cmc	48.81 ± 1.01	magic	710.17 ± 13.44	vene	2.84 ± 0.12
ctg	77.21 ± 1.56	msh	83.39 ± 4.63	vote	2.86 ± 0.16
cvf	331.90 ± 5.53	nurse	227.43 ± 5.02	vow	28.85 ± 0.46
eb	1836.99 ± 561.92	page	115.45 ± 6.33	w21	162.98 ± 2.05
eye	889.33 ± 44.35	pen	221.76 ± 6.02	w40	130.97 ± 1.69
ger	10.69 ± 0.69	pid	22.16 ± 0.36	wfr	243.89 ± 2.55
gls	5.44 ± 0.17	psd	18.03 ± 0.72	wilt	49.67 ± 2.20
hep	1.57 ± 0.15	sb	55.38 ± 1.38	wine	222.03 ± 2.30
hrtc	8.53 ± 0.26	seg	36.17 ± 0.69	zoo	6.44 ± 0.01

The results of the comparison experiments are displayed side by side in the Table 3.14. It can be seen from the results, that only in few cases has the NODT induced significantly advantageous DTs in terms of accuracy, like from datasets: `eye`, `jvow`, `krkopt`, and `letter`, but usually the *EFTI* algorithm had better results both in terms of the accuracy and size. For other datasets, when NODT produced slightly more accurate DTs then *EFTI*, it was compensated by their size being significantly larger in comparison to the DTs induced by *EFTI*. And vice versa, when NODT produced smaller DTs, their accuracy was usually worse than that of DTs induced by *EFTI*. This can also be seen in the fitness column, where *EFTI* always had advantage over NODT.

Table 3.20: *The results of the comparison experiments between the NODT algorithm and the* EFTI *algorithm, displayed side by side for different induced DTs' characteristics: accuracy, size and fitness*

	Accuracy		Size		Fitness	
Dataset	NODT	EFTI	NODT	EFTI	NODT	EFTI
adult	80.44	83.63	527.4 ± 7.8	2.6 ± 0.2	-555.07 ± 16.67	0.83 ± 0.00
ausc	82.35	89.77	7.8 ± 0.8	3.0 ± 0.1	0.75 ± 0.02	0.90 ± 0.00
bank	87.36	89.25	416.6 ± 5.8	2.2 ± 0.2	-375.53 ± 10.71	0.89 ± 0.00
bc	90.02	94.77	9.4 ± 1.1	5.8 ± 0.3	0.85 ± 0.02	0.94 ± 0.00
bch	12.17	24.89	332.6 ± 10.3	225.0 ± 4.4	0.10 ± 0.00	0.23 ± 0.00
bcw	93.22	97.94	3.4 ± 0.7	2.4 ± 0.2	0.92 ± 0.02	0.98 ± 0.00

Continued on next page

Table 3.20 – continued from previous page

Dataset	Accuracy		Size		Fitness	
	NODT	EFTI	NODT	EFTI	NODT	EFTI
ca	82.61	89.46	8.4 ± 1.5	3.0 ± 0.1	0.72 ± 0.04	0.89 ± 0.00
car	91.44	87.71	26.4 ± 1.5	7.9 ± 0.4	0.62 ± 0.04	0.87 ± 0.00
cmc	46.66	61.20	60.0 ± 2.8	5.8 ± 0.3	-1.24 ± 0.17	0.61 ± 0.00
ctg	74.06	79.00	43.1 ± 2.0	18.9 ± 0.9	0.66 ± 0.01	0.78 ± 0.01
cvf	76.87	79.06	142.8 ± 3.8	7.9 ± 0.3	-2.14 ± 0.17	0.79 ± 0.00
eb	65.62	65.13	2654.7 ± 14.3	50.3 ± 1.6	-46.36 ± 0.51	0.65 ± 0.01
eye	74.94	60.16	469.6 ± 8.9	3.4 ± 0.2	-409.65 ± 15.44	0.60 ± 0.00
ger	89.34	97.08	4.6 ± 0.7	2.7 ± 0.2	0.87 ± 0.02	0.97 ± 0.00
gls	60.19	84.95	8.9 ± 1.2	16.6 ± 0.4	0.60 ± 0.03	0.82 ± 0.01
hep	79.35	93.26	2.4 ± 0.4	4.0 ± 0.1	0.79 ± 0.03	0.92 ± 0.01
hrtc	54.54	74.77	10.1 ± 1.0	12.9 ± 0.4	0.54 ± 0.03	0.73 ± 0.01
hrts	78.89	89.63	3.9 ± 0.6	3.4 ± 0.2	0.78 ± 0.03	0.89 ± 0.00
ion	86.06	94.60	3.6 ± 0.4	3.9 ± 0.2	0.85 ± 0.02	0.94 ± 0.01
irs	93.60	98.83	3.5 ± 0.3	3.7 ± 0.3	0.94 ± 0.02	0.99 ± 0.00
jvow	91.03	84.05	46.9 ± 2.6	17.9 ± 0.8	0.74 ± 0.02	0.83 ± 0.01
krkopt	65.93	42.40	1241.4 ± 20.5	47.8 ± 1.0	-29.81 ± 0.96	0.41 ± 0.01
letter	83.31	63.38	363.4 ± 6.3	71.4 ± 2.2	-0.57 ± 0.05	0.61 ± 0.01
liv	65.86	77.86	11.5 ± 1.1	4.5 ± 0.2	0.50 ± 0.04	0.77 ± 0.00
lym	76.37	93.05	2.7 ± 0.3	9.2 ± 0.6	0.76 ± 0.02	0.91 ± 0.01
magic	82.25	84.09	284.9 ± 6.4	3.0 ± 0.0	-164.18 ± 7.42	0.84 ± 0.00
msh	99.82	98.71	6.2 ± 0.5	3.6 ± 0.2	0.95 ± 0.01	0.98 ± 0.00
nurse	95.65	91.60	71.6 ± 2.0	6.3 ± 0.3	-0.76 ± 0.10	0.92 ± 0.00
page	96.51	96.66	23.6 ± 3.1	6.2 ± 0.2	0.82 ± 0.05	0.97 ± 0.00
pen	96.81	95.23	44.0 ± 2.3	19.5 ± 0.7	0.85 ± 0.02	0.94 ± 0.00
pid	71.69	80.51	15.9 ± 1.0	3.4 ± 0.2	0.35 ± 0.05	0.80 ± 0.00
psd	92.90	99.80	5.9 ± 0.5	2.0 ± 0.0	0.89 ± 0.01	1.00 ± 0.00
sb	90.30	93.52	21.2 ± 2.2	2.0 ± 0.0	0.01 ± 0.18	0.94 ± 0.00
seg	94.23	94.69	18.1 ± 1.0	12.6 ± 0.5	0.92 ± 0.01	0.94 ± 0.00
shuttle	99.80	99.69	40.3 ± 1.6	7.2 ± 0.2	0.77 ± 0.02	1.00 ± 0.00
sick	96.08	96.42	11.7 ± 1.4	2.9 ± 0.1	0.71 ± 0.07	0.96 ± 0.00
son	73.94	90.37	2.7 ± 0.2	4.6 ± 0.2	0.74 ± 0.02	0.89 ± 0.01
spect	84.30	93.64	2.6 ± 0.4	3.0 ± 0.1	0.84 ± 0.03	0.93 ± 0.00
spf	68.34	72.41	46.2 ± 2.5	15.0 ± 0.7	0.47 ± 0.03	0.71 ± 0.00
thy	92.82	97.10	24.2 ± 2.1	4.3 ± 0.2	0.68 ± 0.05	0.97 ± 0.00
ttt	71.75	80.15	19.8 ± 1.5	4.8 ± 0.2	0.12 ± 0.09	0.79 ± 0.01
veh	72.48	74.41	16.7 ± 0.9	9.3 ± 0.5	0.65 ± 0.02	0.73 ± 0.01
vene	88.67	94.16	6.8 ± 1.1	5.0 ± 0.2	0.87 ± 0.01	0.94 ± 0.00
vote	87.20	97.49	3.4 ± 0.4	3.0 ± 0.0	0.87 ± 0.02	0.97 ± 0.00

Continued on next page

Table 3.20 – continued from previous page

Dataset	Accuracy		Size		Fitness	
	NODT	EFTI	NODT	EFTI	NODT	EFTI
vow	73.86	77.29	35.0 ± 1.7	38.6 ± 1.0	0.70 ± 0.02	0.72 ± 0.01
w21	82.03	86.48	45.2 ± 1.9	4.3 ± 0.2	-0.84 ± 0.14	0.86 ± 0.00
w40	80.52	85.25	33.9 ± 2.5	4.4 ± 0.2	-0.08 ± 0.13	0.85 ± 0.00
wfr	80.13	78.28	72.2 ± 1.6	9.0 ± 0.3	-1.54 ± 0.11	0.77 ± 0.01
wilt	97.09	94.70	18.2 ± 3.0	2.0 ± 0.1	0.22 ± 0.26	0.95 ± 0.00
wine	55.94	56.85	198.6 ± 3.9	10.8 ± 0.5	-3.64 ± 0.16	0.57 ± 0.00
zoo	78.72	98.06	5.4 ± 0.2	7.2 ± 0.2	0.79 ± 0.03	0.98 ± 0.00

3.5.2.6 GALE

The following section presents the results of the comparison between the GALE algorithm and the *EFTI* algorithm with the "High compression" parameter set, since GALE operates on full DTs in its induction procedure and thus tends to create smaller DTs. The induction times of the GALE algorithm are shown in the Table 3.21, and are even higher than OC1, since GALE operates on the population of the full DTs, which requires more computational time.

Table 3.21: *The average induction times of the GALE algorithm per dataset*

Dataset	Ind. Time [s]	Dataset	Ind. Time [s]	Dataset	Ind. Time [s]
adult	285.20 ± 7.52	hrts	4.96 ± 0.13	shuttle	503.44 ± 51.77
ausc	6.32 ± 0.09	ion	4.88 ± 0.10	sick	21.08 ± 0.34
bank	429.16 ± 62.63	irs	2.36 ± 0.03	son	3.44 ± 0.06
bc	9.00 ± 0.09	jvow	119.04 ± 1.51	spect	1.96 ± 0.03
bch	14.76 ± 0.61	krkopt	216.56 ± 3.85	spf	16.68 ± 0.23
bcw	5.48 ± 0.07	letter	166.08 ± 14.33	thy	18.68 ± 0.11
ca	6.48 ± 0.16	liv	4.40 ± 0.08	ttt	11.40 ± 0.40
car	10.96 ± 0.31	lym	3.36 ± 0.06	veh	8.84 ± 0.13
cmc	12.60 ± 0.30	magic	216.12 ± 7.25	vene	3.32 ± 0.07
ctg	18.48 ± 0.45	msh	67.80 ± 0.93	vote	3.44 ± 0.10
cvf	114.92 ± 6.76	nurse	120.92 ± 3.41	vow	14.04 ± 0.36
eb	134.52 ± 5.17	page	41.40 ± 1.43	w21	48.96 ± 0.98
eye	156.44 ± 2.20	pen	164.32 ± 2.21	w40	48.92 ± 1.06
ger	7.36 ± 0.12	pid	7.28 ± 0.11	wfr	38.52 ± 1.33
gls	3.96 ± 0.17	psd	7.08 ± 0.10	wilt	21.60 ± 0.18
hep	3.16 ± 0.14	sb	15.36 ± 0.50	wine	34.16 ± 0.73
hrtc	5.84 ± 0.22	seg	22.80 ± 0.30	zoo	2.92 ± 0.04

The results of the comparison experiments are displayed side by side in the Table 3.22. The results show that the *EFTI* algorithm produces more accurate DTs with all datasets used in experiments (except for the `ttt` dataset, where it produced on average 15 times smaller DTs, with 4% loss in accuracy). In addition, for most of the datasets, it was able to produce smaller DTs as well. In case of the datasets where DTs produced by GALE were smaller, like: `bch`, `cvf`, `eb`, `gls`, `krkopt`, `letter`, `seg` and `shuttle`, they were also much less accurate then the ones induced by *EFTI*.

Table 3.22: *The results of the comparison experiments between the GALE algorithm and the* EFTI *algorithm, displayed side by side for different induced DTs' characteristics: accuracy, size and fitness*

	Accuracy		Size		Fitness	
Dataset	GALE	EFTI	GALE	EFTI	GALE	EFTI
adult	81.17	83.02	3.0 ± 0.0	2.0 ± 0.0	0.77 ± 0.00	0.83 ± 0.00
ausc	85.12	89.00	8.0 ± 0.0	2.0 ± 0.0	-0.68 ± 0.01	0.89 ± 0.00
bank	89.02	88.82	2.0 ± 0.0	2.0 ± 0.0	0.89 ± 0.00	0.89 ± 0.00
bc	78.64	92.81	14.0 ± 0.0	3.0 ± 0.0	-1.33 ± 0.03	0.93 ± 0.00
bch	7.21	18.55	8.0 ± 0.0	82.5 ± 1.2	0.06 ± 0.00	0.18 ± 0.00
bcw	92.35	97.80	3.0 ± 0.0	2.0 ± 0.0	0.88 ± 0.01	0.98 ± 0.00
ca	85.45	88.66	8.0 ± 0.0	2.0 ± 0.0	-0.68 ± 0.01	0.89 ± 0.00
car	73.62	84.32	3.0 ± 0.0	4.4 ± 0.2	0.73 ± 0.02	0.84 ± 0.00
cmc	52.18	59.39	4.0 ± 0.0	4.0 ± 0.0	0.51 ± 0.02	0.58 ± 0.00
ctg	53.34	77.09	11.0 ± 0.0	11.0 ± 0.2	0.53 ± 0.03	0.77 ± 0.01
cvf	62.92	78.08	2.0 ± 0.0	7.0 ± 0.0	0.57 ± 0.01	0.78 ± 0.00
eb	15.65	53.37	3.0 ± 0.0	33.5 ± 0.4	0.13 ± 0.01	0.53 ± 0.01
eye	56.68	59.57	3.0 ± 0.0	2.0 ± 0.0	0.54 ± 0.00	0.60 ± 0.00
ger	93.98	96.70	3.0 ± 0.0	2.0 ± 0.0	0.89 ± 0.01	0.97 ± 0.00
gls	60.62	78.19	5.0 ± 0.0	7.5 ± 0.2	0.60 ± 0.04	0.77 ± 0.01
hep	80.91	92.03	10.0 ± 0.0	2.0 ± 0.0	-1.78 ± 0.05	0.92 ± 0.01
hrtc	56.00	70.60	8.0 ± 0.0	6.0 ± 0.0	0.52 ± 0.01	0.70 ± 0.00
hrts	78.00	88.10	18.0 ± 0.0	2.0 ± 0.0	-9.20 ± 0.25	0.88 ± 0.00
ion	90.40	91.50	7.0 ± 0.0	2.0 ± 0.1	-0.23 ± 0.00	0.91 ± 0.01
irs	94.97	98.45	3.0 ± 0.0	3.0 ± 0.0	0.95 ± 0.02	0.98 ± 0.00
jvow	46.19	81.91	15.0 ± 0.0	10.6 ± 0.3	0.42 ± 0.02	0.81 ± 0.01
krkopt	25.03	37.71	5.0 ± 0.0	21.8 ± 0.3	0.22 ± 0.01	0.37 ± 0.01
letter	18.66	56.92	20.0 ± 0.0	33.4 ± 0.6	0.18 ± 0.01	0.56 ± 0.01
liv	60.00	73.99	5.0 ± 0.0	2.0 ± 0.0	0.33 ± 0.01	0.74 ± 0.00
lym	76.46	90.27	5.0 ± 0.0	4.6 ± 0.2	0.76 ± 0.03	0.90 ± 0.01
magic	78.29	80.18	7.0 ± 0.0	2.0 ± 0.0	-0.20 ± 0.00	0.80 ± 0.00
msh	95.08	96.83	9.0 ± 0.0	2.0 ± 0.0	-1.38 ± 0.01	0.97 ± 0.00
nurse	79.18	90.71	10.0 ± 0.0	5.0 ± 0.0	0.63 ± 0.01	0.91 ± 0.00
page	92.36	95.93	6.0 ± 0.0	5.0 ± 0.0	0.92 ± 0.00	0.96 ± 0.00
pen	62.69	92.68	28.0 ± 0.0	12.3 ± 0.2	0.22 ± 0.01	0.92 ± 0.00
pid	73.52	79.34	6.0 ± 0.0	2.0 ± 0.0	0.15 ± 0.00	0.79 ± 0.00

Continued on next page

Table 3.22 – continued from previous page

Dataset	Accuracy GALE	Accuracy EFTI	Size GALE	Size EFTI	Fitness GALE	Fitness EFTI
psd	100.00	98.98	2.0 ± 0.0	2.0 ± 0.0	1.00 ± 0.00	0.99 ± 0.01
sb	93.34	93.51	1.0 ± 0.0	2.0 ± 0.0	0.89 ± 0.00	0.94 ± 0.00
seg	69.15	93.30	6.0 ± 0.0	8.0 ± 0.2	0.69 ± 0.02	0.93 ± 0.00
shuttle	88.08	99.56	4.0 ± 0.0	7.0 ± 0.0	0.85 ± 0.01	1.00 ± 0.00
sick	93.75	94.05	2.0 ± 0.0	2.0 ± 0.0	0.94 ± 0.00	0.94 ± 0.00
son	72.03	88.00	7.0 ± 0.0	2.0 ± 0.1	-0.18 ± 0.01	0.88 ± 0.01
spect	88.04	93.44	2.0 ± 0.0	2.0 ± 0.0	0.88 ± 0.01	0.93 ± 0.00
spf	52.88	69.51	6.0 ± 0.0	8.1 ± 0.1	0.53 ± 0.01	0.69 ± 0.00
thy	93.78	95.15	3.0 ± 0.0	4.0 ± 0.0	0.93 ± 0.00	0.95 ± 0.00
ttt	75.58	72.83	31.0 ± 0.0	2.0 ± 0.0	-31.03 ± 0.99	0.73 ± 0.00
veh	57.88	72.45	6.0 ± 0.0	4.9 ± 0.2	0.55 ± 0.02	0.72 ± 0.01
vene	90.87	92.47	5.0 ± 0.0	3.0 ± 0.0	0.83 ± 0.01	0.92 ± 0.00
vote	93.62	96.51	6.0 ± 0.0	2.0 ± 0.0	0.19 ± 0.00	0.97 ± 0.00
vow	39.09	66.37	18.0 ± 0.0	15.5 ± 0.3	0.36 ± 0.02	0.64 ± 0.01
w21	71.71	85.70	9.0 ± 0.0	3.8 ± 0.2	0.14 ± 0.00	0.84 ± 0.00
w40	68.84	83.54	18.0 ± 0.0	3.4 ± 0.2	-2.75 ± 0.04	0.83 ± 0.00
wfr	71.84	74.82	3.0 ± 0.0	4.9 ± 0.1	0.71 ± 0.02	0.74 ± 0.01
wilt	94.67	94.70	2.0 ± 0.0	2.0 ± 0.0	0.95 ± 0.00	0.95 ± 0.00
wine	49.60	56.15	7.0 ± 0.0	7.1 ± 0.1	0.50 ± 0.01	0.56 ± 0.00
zoo	85.02	97.47	7.0 ± 0.0	7.2 ± 0.2	0.85 ± 0.03	0.97 ± 0.01

3.5.2.7 GaTree

The following section presents the results of the comparison between the GaTree algorithm and the *EFTI* algorithm with the “High compression” parameter set, since GaTree operates on full DTs in its induction procedure and thus tends to create smaller DTs. The induction times of the GaTree algorithm are shown in the Table 3.23.

Table 3.23: *The average induction times of the GaTree algorithm per dataset*

Dataset	Ind. Time [s]	Dataset	Ind. Time [s]	Dataset	Ind. Time [s]
adult	450.57 ± 126.11	hrts	0.60 ± 0.05	shuttle	415.02 ± 15.47
ausc	2.82 ± 0.59	ion	5.91 ± 0.47	sick	9.89 ± 0.94
bank	455.79 ± 49.89	irs	0.71 ± 0.05	son	7.03 ± 0.37
bc	0.91 ± 0.02	jvow	2047.51 ± 222.08	spect	0.26 ± 0.01
bch	14.39 ± 1.78	krkopt	44.00 ± 1.06	spf	55.52 ± 5.74
bcw	1.18 ± 0.06	letter	61.65 ± 2.48	thy	13.50 ± 0.90
ca	2.41 ± 0.44	liv	2.60 ± 0.21	ttt	1.01 ± 0.06
car	1.24 ± 0.09	lym	0.25 ± 0.01	veh	5.92 ± 0.70
cmc	2.19 ± 0.12	magic	3562.43 ± 450.99	vene	12.36 ± 0.79
ctg	33.36 ± 3.72	msh	15.39 ± 0.64	vote	0.39 ± 0.01
cvf	14.35 ± 0.34	nurse	13.36 ± 0.43	vow	86.91 ± 4.00
eb	7704.86 ± 1883.51	page	78.34 ± 3.39	w21	387.71 ± 19.65
eye	389.92 ± 26.98	pen	56.41 ± 4.66	w40	382.56 ± 13.28
ger	3.07 ± 4.98	pid	9.40 ± 0.81	wfr	463.34 ± 51.23
gls	2.71 ± 0.66	psd	32.14 ± 5.47	wilt	251.65 ± 11.21
hep	0.78 ± 0.76	sb	26.99 ± 3.48	wine	68.14 ± 6.00
hrtc	0.73 ± 0.09	seg	63.97 ± 7.80	zoo	0.23 ± 0.01

The results of the comparison experiments are displayed side by side in the Table 3.24. The results show that the *EFTI* algorithm produces more accurate DTs with all datasets used in experiments, with almost all of them being smaller in size as well. In case of the datasets where DTs produced by GaTree were smaller, like: `bch`, `eb`, `letter`, `page`, and `thy`, they were also much less accurate then the ones induced by *EFTI*.

Table 3.24: *The results of the comparison experiments between the GaTree algorithm and the* EFTI *algorithm, displayed side by side for different induced DTs' characteristics: accuracy, size and fitness*

	Accuracy		Size		Fitness	
Dataset	GaTree	EFTI	GaTree	EFTI	GaTree	EFTI
adult	79.15	82.96	9.0 ± 3.3	2.0 ± 0.0	-1.94 ± 2.45	0.83 ± 0.00
ausc	85.25	89.20	7.7 ± 0.8	2.0 ± 0.0	-0.90 ± 0.44	0.89 ± 0.00
bank	88.34	89.04	2.8 ± 0.2	2.0 ± 0.0	0.84 ± 0.02	0.89 ± 0.00
bc	71.68	92.61	14.0 ± 1.6	3.0 ± 0.0	-1.75 ± 0.83	0.93 ± 0.00
bch	6.83	19.67	18.0 ± 1.9	84.4 ± 1.1	0.06 ± 0.00	0.19 ± 0.00
bcw	93.32	97.83	10.6 ± 1.2	2.0 ± 0.0	-3.30 ± 1.06	0.98 ± 0.00
ca	84.49	88.85	7.2 ± 0.8	2.0 ± 0.0	-0.61 ± 0.40	0.89 ± 0.00
car	74.06	84.31	4.7 ± 1.3	4.3 ± 0.2	0.53 ± 0.16	0.84 ± 0.00
cmc	46.38	59.64	11.9 ± 1.1	4.0 ± 0.0	-0.51 ± 0.24	0.58 ± 0.00

Continued on next page

Table 3.24 – continued from previous page

Dataset	Accuracy GaTree	Accuracy EFTI	Size GaTree	Size EFTI	Fitness GaTree	Fitness EFTI
ctg	29.25	77.80	12.3 ± 1.1	10.8 ± 0.2	0.28 ± 0.01	0.78 ± 0.01
cvf	59.47	78.38	13.0 ± 2.0	7.0 ± 0.0	0.38 ± 0.07	0.78 ± 0.00
eb	14.82	63.75	6.2 ± 6.1	33.0 ± 0.3	0.13 ± 0.05	0.64 ± 0.01
eye	55.73	59.49	9.9 ± 1.1	2.0 ± 0.0	-1.37 ± 0.50	0.59 ± 0.00
ger	93.40	96.76	2.8 ± 0.6	2.0 ± 0.0	0.90 ± 0.04	0.97 ± 0.00
gls	34.86	78.19	19.1 ± 3.6	7.3 ± 0.2	-0.13 ± 0.28	0.77 ± 0.01
hep	77.94	91.33	9.1 ± 1.7	2.0 ± 0.0	-1.84 ± 1.24	0.91 ± 0.01
hrtc	53.80	70.84	10.9 ± 1.9	6.0 ± 0.1	0.30 ± 0.13	0.70 ± 0.00
hrts	77.33	88.31	12.0 ± 1.2	2.0 ± 0.0	-3.81 ± 1.14	0.88 ± 0.00
ion	79.89	91.68	12.3 ± 1.2	2.0 ± 0.1	-4.13 ± 1.11	0.91 ± 0.00
irs	67.33	98.35	24.8 ± 2.9	3.0 ± 0.0	-8.61 ± 2.45	0.98 ± 0.00
jvow	17.08	82.32	12.0 ± 1.6	10.3 ± 0.2	0.15 ± 0.01	0.82 ± 0.01
krkopt	24.36	38.85	13.1 ± 1.9	21.8 ± 0.3	0.23 ± 0.01	0.38 ± 0.01
letter	10.98	59.50	17.5 ± 2.5	33.4 ± 0.6	0.11 ± 0.00	0.58 ± 0.01
liv	54.67	74.49	20.2 ± 1.6	2.0 ± 0.0	-9.28 ± 1.60	0.74 ± 0.00
lym	76.97	89.76	11.8 ± 1.4	4.4 ± 0.2	-0.07 ± 0.26	0.89 ± 0.01
magic	64.94	80.18	4.7 ± 0.5	2.0 ± 0.0	0.40 ± 0.09	0.80 ± 0.00
msh	95.74	97.04	9.4 ± 1.3	2.0 ± 0.0	-2.70 ± 1.57	0.97 ± 0.00
nurse	54.61	91.08	6.1 ± 1.4	5.0 ± 0.1	0.44 ± 0.06	0.91 ± 0.00
page	90.22	96.08	4.0 ± 0.4	5.0 ± 0.0	0.88 ± 0.00	0.96 ± 0.00
pen	25.21	92.41	12.8 ± 1.9	12.1 ± 0.3	0.22 ± 0.03	0.92 ± 0.01
pid	66.22	79.59	15.3 ± 1.2	2.0 ± 0.0	-5.83 ± 1.17	0.80 ± 0.00
psd	82.08	99.62	10.2 ± 2.5	2.0 ± 0.0	-4.32 ± 2.39	1.00 ± 0.00
sb	93.27	93.51	3.9 ± 0.4	2.0 ± 0.0	0.68 ± 0.07	0.94 ± 0.00
seg	17.72	93.61	14.5 ± 1.2	7.9 ± 0.2	0.12 ± 0.01	0.93 ± 0.00
shuttle	82.38	99.64	9.0 ± 0.7	7.0 ± 0.0	0.79 ± 0.02	1.00 ± 0.00
sick	94.07	94.21	2.9 ± 0.2	2.0 ± 0.0	0.87 ± 0.03	0.94 ± 0.00
son	54.24	88.13	29.8 ± 2.0	2.2 ± 0.2	-21.83 ± 2.96	0.87 ± 0.01
spect	79.62	93.78	2.6 ± 0.8	2.0 ± 0.0	0.47 ± 0.46	0.94 ± 0.00
spf	42.25	69.92	11.0 ± 1.4	8.1 ± 0.1	0.36 ± 0.04	0.70 ± 0.00
thy	92.36	95.96	2.9 ± 0.3	4.0 ± 0.0	0.90 ± 0.01	0.96 ± 0.00
ttt	72.92	72.78	11.5 ± 2.1	2.0 ± 0.0	-3.51 ± 1.69	0.73 ± 0.00
veh	39.22	71.48	18.7 ± 2.3	5.0 ± 0.1	-1.01 ± 0.40	0.71 ± 0.01
vene	29.87	92.20	31.6 ± 2.1	3.0 ± 0.0	-5.55 ± 0.97	0.92 ± 0.00
vote	95.40	96.74	3.3 ± 0.7	2.0 ± 0.0	0.62 ± 0.24	0.97 ± 0.00
vow	6.16	68.11	22.9 ± 1.6	15.7 ± 0.3	0.04 ± 0.00	0.66 ± 0.01
w21	33.06	85.00	17.0 ± 1.3	3.4 ± 0.2	-1.26 ± 0.26	0.84 ± 0.00
w40	33.34	83.58	17.3 ± 0.9	3.2 ± 0.2	-1.35 ± 0.21	0.83 ± 0.00

Continued on next page

Table 3.24 – continued from previous page

	Accuracy		Size		Fitness	
Dataset	GaTree	EFTI	GaTree	EFTI	GaTree	EFTI
wfr	43.24	74.61	8.2 ± 0.7	4.9 ± 0.1	0.30 ± 0.04	0.74 ± 0.01
wilt	94.55	94.62	3.4 ± 0.3	2.0 ± 0.0	0.82 ± 0.04	0.95 ± 0.00
wine	45.56	56.19	9.2 ± 0.7	7.0 ± 0.1	0.43 ± 0.01	0.56 ± 0.00
zoo	80.60	97.90	13.2 ± 2.0	7.2 ± 0.2	0.51 ± 0.17	0.98 ± 0.01

3.5.3 Group comparison of all algorithms

In this section, the results of the experiments are displayed and discussed, that compare all the algorithms from the Table 3.7 together with the proposed *EFTI* algorithm in terms of induced DT accuracies and sizes. In these experiments, the *EFTI* algorithm was setup using the "High accuracy" configuration for the Table 3.10 and given 1000k iterations for the induction. The cross-validation employed and the rankings devised in the manner described in the Section 2.8. The results are listed in the following tables:

- Table 3.25 shows the average accuracies of the induced DTs,
- Table 3.26 shows the 95% confidence intervals for the accuracies of the induced DTs
- Table 3.27 shows the relative differences in accuracies of the DTs induced by the existing algorithms compared to the DTs induced by the *EFTI* algorithm on the same dataset. Values are given in percents, where the positive numbers show the amount by which an existing algorithm produces more accurate DTs, relative to those induced by *EFTI*, and negative numbers show the opposite.
- Table 3.28 shows the average sizes of the induced DTs,
- Table 3.29 shows the 95% confidence intervals for the sizes of the induced DTs
- Table 3.30 shows the relative differences in sizes of the DTs induced by the existing algorithms compared to the DTs induced by the *EFTI* algorithm on the same dataset. Values are given in percents, where the positive numbers show the amount by which an existing algorithm produces larger DTs, relative to those induced by *EFTI*, and negative numbers show the opposite.
- Table 3.31 shows the ranking of the algorithms based on the accuracies of the induced DTs
- Table 3.32 shows the ranking of the algorithms based on the sizes of the induced DTs

Table 3.25: *The average accuracies of the induced DTs by all algorithms from the Table 3.7 and* EFTI*, on all datasets from the Table 2.1 from five 5-fold cross-validation test.*

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
adult	85.83	85.48	85.22	85.59	85.55	80.44	81.17	79.15	83.85
ausc	85.45	84.58	83.48	85.59	85.28	82.35	85.12	85.25	89.92
bank	90.14	89.75	89.54	90.10	89.96	87.36	89.02	88.34	89.38

Continued on next page

Table 3.25 – continued from previous page

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
bc	79.01	88.26	91.94	85.09	86.05	90.02	78.64	71.68	95.16
bch	14.36	13.25	12.46	14.16	13.88	12.17	7.21	6.83	25.58
bcw	91.11	93.23	93.86	92.23	92.44	93.22	92.35	93.32	97.96
ca	85.10	85.54	84.14	85.25	84.87	82.61	85.45	84.49	89.51
car	96.17	94.72	93.16	95.21	94.32	91.44	73.62	74.06	88.60
cmc	54.75	53.93	53.05	53.53	52.10	46.66	52.18	46.38	61.29
ctg	82.18	81.95	79.88	82.15	82.24	74.06	53.34	29.25	79.53
cvf	75.08	76.03	75.80	72.61	74.21	76.87	62.92	59.47	79.40
eb	65.45	65.46	65.49	65.48	65.47	65.62	15.65	14.82	65.54
eye	83.44	83.37	84.72	83.40	83.52	74.94	56.68	55.73	60.34
ger	96.06	95.48	93.96	96.16	96.32	89.34	93.98	93.40	97.40
gls	64.63	66.55	62.23	64.91	64.67	60.19	60.62	34.86	85.91
hep	78.19	77.29	77.29	78.58	77.42	79.35	80.91	77.94	93.70
hrtc	54.31	52.66	52.71	52.46	54.04	54.54	56.00	53.80	75.62
hrts	77.78	76.00	79.41	78.00	75.19	78.89	78.00	77.33	90.01
ion	88.86	89.71	86.86	88.51	88.97	86.06	90.40	79.89	95.13
irs	92.93	93.57	96.16	94.28	93.60	93.60	94.97	67.33	98.56
jvow	87.11	90.64	90.86	88.10	87.93	91.03	46.19	17.08	85.73
krkopt	80.42	77.70	71.41	76.25	75.17	65.93	25.03	24.36	42.93
letter	86.15	83.81	82.10	84.40	85.08	83.31	18.66	10.98	64.08
liv	65.04	66.38	65.68	64.99	65.51	65.86	60.00	54.67	78.59
lym	69.90	74.61	77.08	71.79	70.43	76.37	76.46	76.97	93.62
magic	85.22	86.12	86.22	85.35	85.17	82.25	78.29	64.94	84.19
msh	99.89	99.90	99.73	99.83	99.91	99.82	95.08	95.74	98.79
nurse	99.12	98.08	95.73	97.65	96.66	95.65	79.18	54.61	92.03
page	96.75	96.92	96.85	97.10	96.85	96.51	92.36	90.22	96.92
pen	95.65	96.64	96.49	95.84	95.74	96.81	62.69	25.21	95.57
pid	74.23	74.27	73.44	73.30	73.90	71.69	73.52	66.22	80.85
psd	100.00	100.00	100.00	100.00	100.00	92.90	100.00	82.08	99.84
sb	93.08	93.17	93.32	93.28	93.34	90.30	93.34	93.27	93.54
seg	95.47	94.54	93.73	94.36	95.35	94.23	69.15	17.72	95.13
shuttle	99.96	99.96	99.94	99.95	99.95	99.80	88.08	82.38	99.72
sick	98.49	97.73	96.57	98.53	98.28	96.08	93.75	94.07	96.98
son	70.55	71.88	69.59	70.28	70.38	73.94	72.03	54.24	90.90
spect	87.37	82.64	81.75	86.35	87.47	84.30	88.04	79.62	94.32
spf	73.58	71.23	69.53	72.83	71.94	68.34	52.88	42.25	73.11
thy	99.29	98.85	98.48	99.24	99.22	92.82	93.78	92.36	97.54
ttt	85.87	79.36	75.68	81.27	77.68	71.75	75.58	72.92	79.20
veh	68.01	69.20	69.84	68.79	68.20	72.48	57.88	39.22	75.34
vene	89.87	90.40	89.40	88.20	89.33	88.67	90.87	29.87	94.36
vote	93.68	93.28	92.29	94.41	94.16	87.20	93.62	95.40	97.59
vow	76.30	78.18	78.20	76.81	76.63	73.86	39.09	6.16	78.21
w21	76.99	81.30	81.99	77.54	77.12	82.03	71.71	33.06	86.88
w40	76.39	80.66	80.89	76.87	76.25	80.52	68.84	33.34	85.80
wfr	99.34	98.09	97.40	99.36	99.35	80.13	71.84	43.24	79.95
wilt	97.99	98.01	97.93	97.83	97.86	97.09	94.67	94.55	94.79
wine	56.37	57.41	57.17	56.82	57.84	55.94	49.60	45.56	57.14
zoo	85.79	85.31	82.44	77.68	83.10	78.72	85.02	80.60	98.42

Table 3.26: *The 95% confidence intervals for the accuracies of the induced DTs by all algorithms from the Table 3.7, on all datasets from the Table 2.1 from five 5-fold cross-validation test.*

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
adult	±0.15	±0.16	±0.17	±0.21	±0.18	±0.20	±0.37	±0.91	±0.14
ausc	±1.16	±1.01	±1.41	±1.03	±1.11	±1.37	±1.43	±0.57	±0.18
bank	±0.14	±0.14	±0.12	±0.11	±0.15	±0.10	±0.11	±0.12	±0.14
bc	±1.24	±1.64	±1.11	±1.55	±1.83	±1.02	±1.70	±1.13	±0.30
bch	±0.42	±0.46	±0.37	±0.43	±0.46	±0.34	±0.24	±0.24	±0.20
bcw	±1.08	±1.15	±1.13	±1.40	±1.35	±1.28	±1.37	±0.61	±0.08
ca	±0.97	±1.13	±1.29	±1.16	±1.14	±1.10	±0.98	±0.83	±0.16
car	±0.49	±0.59	±0.79	±0.64	±0.59	±0.87	±1.63	±1.50	±0.37
cmc	±1.27	±1.13	±1.38	±1.69	±1.28	±0.97	±1.75	±0.57	±0.47
ctg	±0.76	±0.76	±1.01	±0.86	±0.79	±0.84	±3.21	±0.79	±0.43
cvf	±0.51	±0.47	±0.50	±0.59	±0.43	±0.41	±1.26	±0.59	±0.22
eb	±0.23	±0.23	±0.17	±0.21	±0.21	±0.14	±0.68	±4.83	±0.46
eye	±0.47	±0.37	±0.36	±0.31	±0.34	±0.33	±0.45	±0.38	±0.18
ger	±0.49	±0.58	±1.20	±0.51	±0.45	±1.35	±1.35	±5.33	±0.15
gls	±3.91	±3.65	±3.16	±4.20	±3.46	±3.12	±3.58	±2.86	±0.83
hep	±3.34	±3.29	±3.27	±3.84	±3.08	±2.75	±2.29	±2.23	±0.57
hrtc	±2.00	±2.36	±3.03	±2.76	±2.12	±2.50	±1.08	±2.22	±0.57
hrts	±2.15	±2.44	±2.04	±2.63	±2.11	±2.54	±2.13	±0.99	±0.41
ion	±1.49	±1.32	±1.91	±1.79	±1.34	±2.14	±1.77	±1.36	±0.31
irs	±2.04	±1.71	±1.38	±1.57	±1.98	±1.98	±1.69	±3.37	±0.36
jvow	±0.38	±0.34	±0.36	±0.50	±0.32	±0.48	±2.63	±0.23	±0.48
krkopt	±0.31	±0.43	±0.44	±0.35	±0.42	±0.55	±0.73	±0.34	±0.61
letter	±0.31	±0.23	±0.37	±0.31	±0.27	±0.39	±1.35	±0.37	±0.71
liv	±2.09	±2.14	±1.86	±1.64	±2.89	±2.88	±2.08	±1.25	±0.50
lym	±3.86	±3.99	±2.23	±2.61	±3.50	±2.42	±3.51	±2.59	±0.62
magic	±0.19	±0.17	±0.25	±0.24	±0.28	±0.27	±0.50	±0.36	±0.05
msh	±0.06	±0.06	±0.07	±0.08	±0.04	±0.06	±0.80	±0.52	±0.20
nurse	±0.10	±0.15	±0.31	±0.22	±0.28	±0.18	±1.82	±0.53	±0.14
page	±0.18	±0.20	±0.23	±0.20	±0.19	±0.36	±0.25	±0.25	±0.09
pen	±0.13	±0.22	±0.21	±0.20	±0.21	±0.22	±1.74	±0.98	±0.24
pid	±1.63	±1.22	±1.99	±1.13	±1.20	±0.87	±1.55	±0.83	±0.17
psd	±0.00	±0.00	±0.00	±0.00	±0.00	±0.47	±0.00	±3.96	±0.07
sb	±0.35	±0.40	±0.35	±0.40	±0.48	±0.54	±0.10	±0.21	±0.02
seg	±0.46	±0.47	±0.66	±0.65	±0.55	±0.49	±2.41	±0.49	±0.27
shuttle	±0.02	±0.01	±0.01	±0.01	±0.01	±0.01	±1.32	±0.54	±0.01
sick	±0.27	±0.30	±0.25	±0.28	±0.41	±0.29	±0.04	±0.28	±0.27
son	±2.61	±3.36	±2.46	±2.45	±3.17	±2.37	±2.85	±1.67	±0.60
spect	±1.91	±2.45	±2.45	±2.06	±1.78	±2.65	±0.53	±0.99	±0.22
spf	±0.92	±0.90	±0.91	±1.19	±1.18	±1.03	±1.42	±1.26	±0.43
thy	±0.11	±0.18	±0.27	±0.16	±0.14	±0.51	±0.14	±0.18	±0.26
ttt	±1.43	±1.67	±1.67	±1.66	±2.21	±1.92	±2.41	±1.42	±1.06
veh	±1.44	±1.68	±1.66	±1.47	±1.93	±1.32	±2.49	±1.40	±0.83
vene	±2.19	±1.57	±1.61	±1.78	±1.79	±1.51	±1.18	±0.49	±0.16
vote	±1.31	±1.04	±1.26	±1.31	±1.10	±1.67	±1.00	±0.37	±0.24
vow	±1.17	±1.69	±1.57	±1.83	±1.42	±1.31	±1.66	±0.34	±0.77
w21	±0.54	±0.60	±0.57	±0.52	±0.48	±0.25	±0.97	±0.41	±0.15

Continued on next page

Table 3.26 – continued from previous page

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
w40	±0.51	±0.56	±0.57	±0.46	±0.48	±0.46	±0.98	±0.22	±0.26
wfr	±0.11	±0.18	±0.30	±0.10	±0.11	±0.31	±1.67	±0.73	±0.89
wilt	±0.22	±0.23	±0.18	±0.19	±0.27	±0.33	±0.02	±0.19	±0.24
wine	±0.81	±0.86	±0.89	±0.87	±0.86	±0.77	±0.83	±0.31	±0.20
zoo	±3.82	±3.30	±4.29	±7.15	±6.11	±2.69	±2.56	±2.55	±0.37

Table 3.27: *The relative differences in accuracies of the DTs induced by the algorithms from the Table 3.7, compared to the DTs induced by the* EFTI *algorithm on the same dataset.*

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree
adult	2.37	1.95	1.63	2.08	2.04	−4.07	−3.19	−5.60
ausc	−4.97	−5.94	−7.16	−4.81	−5.16	−8.42	−5.33	−5.20
bank	0.85	0.41	0.18	0.81	0.65	−2.26	−0.41	−1.17
bc	−16.97	−7.25	−3.38	−10.58	−9.57	−5.40	−17.35	−24.67
bch	−43.84	−48.22	−51.29	−44.64	−45.73	−52.43	−71.80	−73.29
bcw	−6.99	−4.83	−4.18	−5.85	−5.64	−4.84	−5.73	−4.74
ca	−4.92	−4.44	−5.99	−4.76	−5.18	−7.71	−4.53	−5.60
car	8.55	6.91	5.15	7.46	6.46	3.20	−16.90	−16.40
cmc	−10.67	−12.01	−13.46	−12.66	−14.99	−23.88	−14.87	−24.33
ctg	3.32	3.04	0.43	3.29	3.40	−6.88	−32.94	−63.22
cvf	−5.44	−4.25	−4.53	−8.55	−6.53	−3.18	−20.75	−25.10
eb	−0.14	−0.12	−0.07	−0.10	−0.11	0.12	−76.12	−77.39
eye	38.29	38.17	40.40	38.22	38.42	24.20	−6.06	−7.63
ger	−1.38	−1.97	−3.53	−1.27	−1.11	−8.28	−3.51	−4.11
gls	−24.76	−22.54	−27.56	−24.44	−24.72	−29.94	−29.43	−59.42
hep	−16.55	−17.52	−17.52	−16.14	−17.38	−15.31	−13.65	−16.83
hrtc	−28.17	−30.36	−30.29	−30.63	−28.53	−27.87	−25.94	−28.85
hrts	−13.59	−15.57	−11.78	−13.35	−16.47	−12.36	−13.35	−14.09
ion	−6.60	−5.70	−8.70	−6.96	−6.48	−9.54	−4.98	−16.03
irs	−5.71	−5.06	−2.44	−4.34	−5.03	−5.03	−3.64	−31.68
jvow	1.62	5.73	5.99	2.77	2.57	6.19	−46.11	−80.08
krkopt	87.31	80.98	66.33	77.61	75.08	53.57	−41.69	−43.26
letter	34.43	30.79	28.12	31.71	32.77	30.01	−70.88	−82.87
liv	−17.23	−15.54	−16.42	−17.31	−16.64	−16.20	−23.65	−30.44
lym	−25.34	−20.30	−17.67	−23.32	−24.78	−18.43	−18.33	−17.79
magic	1.22	2.30	2.42	1.38	1.17	−2.31	−7.00	−22.86
msh	1.11	1.12	0.94	1.05	1.13	1.04	−3.76	−3.09
nurse	7.71	6.57	4.02	6.11	5.03	3.94	−13.96	−40.66
page	−0.17	−0.00	−0.08	0.18	−0.07	−0.42	−4.71	−6.92
pen	0.08	1.12	0.96	0.29	0.18	1.30	−34.40	−73.62
pid	−8.20	−8.14	−9.17	−9.34	−8.60	−11.33	−9.07	−18.10
psd	0.16	0.16	0.16	0.16	0.16	−6.95	0.16	−17.79
sb	−0.49	−0.39	−0.23	−0.27	−0.21	−3.46	−0.22	−0.29
seg	0.36	−0.62	−1.47	−0.81	0.23	−0.94	−27.32	−81.37
shuttle	0.24	0.23	0.22	0.23	0.23	0.08	−11.68	−17.39
sick	1.56	0.77	−0.42	1.60	1.34	−0.92	−3.32	−3.00
son	−22.39	−20.93	−23.45	−22.69	−22.57	−18.66	−20.76	−40.33

Continued on next page

Table 3.27 – continued from previous page

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree
spect	−7.37	−12.38	−13.33	−8.45	−7.26	−10.63	−6.66	−15.58
spf	0.64	−2.58	−4.90	−0.39	−1.60	−6.53	−27.68	−42.22
thy	1.79	1.34	0.96	1.74	1.72	−4.84	−3.86	−5.31
ttt	8.43	0.20	−4.44	2.62	−1.92	−9.40	−4.56	−7.93
veh	−9.73	−8.16	−7.31	−8.69	−9.48	−3.80	−23.18	−47.95
vene	−4.76	−4.20	−5.26	−6.53	−5.33	−6.03	−3.70	−68.35
vote	−4.01	−4.41	−5.44	−3.26	−3.51	−10.65	−4.07	−2.24
vow	−2.44	−0.04	−0.01	−1.79	−2.03	−5.56	−50.02	−92.12
w21	−11.39	−6.42	−5.63	−10.75	−11.23	−5.58	−17.46	−61.94
w40	−10.96	−5.99	−5.72	−10.41	−11.12	−6.15	−19.76	−61.14
wfr	24.25	22.70	21.83	24.28	24.28	0.23	−10.14	−45.92
wilt	3.37	3.40	3.32	3.20	3.24	2.43	−0.13	−0.25
wine	−1.34	0.48	0.07	−0.55	1.24	−2.09	−13.19	−20.26
zoo	−12.83	−13.31	−16.23	−21.07	−15.56	−20.01	−13.61	−18.10

Table 3.28: *The average sizes of the induced DTs by all algorithms from the Table 3.7 and* EFTI*, on all datasets from the Table 2.1 from five 5-fold cross-validation test.*

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
adult	86.5	55.0	33.5	54.9	89.5	527.4	3.0	9.0	2.8
ausc	7.7	4.1	4.7	5.2	4.6	7.8	8.0	7.7	3.0
bank	97.8	42.7	17.0	75.7	72.3	416.6	2.0	2.8	2.3
bc	22.4	8.7	8.4	13.9	16.0	9.4	14.0	14.0	5.8
bch	403.8	259.3	285.5	379.1	395.4	332.6	8.0	18.0	238.6
bcw	10.5	3.0	3.1	5.6	5.8	3.4	3.0	10.6	2.4
ca	6.0	6.1	3.2	8.7	9.4	8.4	8.0	7.2	3.0
car	51.8	36.3	27.8	47.3	43.0	26.4	3.0	4.7	7.9
cmc	68.7	21.4	25.1	34.0	35.7	60.0	4.0	11.9	6.4
ctg	73.8	52.7	28.8	59.6	54.3	43.1	11.0	12.3	19.8
cvf	547.4	35.6	37.0	219.8	199.9	142.8	2.0	13.0	8.3
eb	1592.0	1818.4	1013.5	2278.1	1545.2	2654.7	3.0	6.2	53.5
eye	547.0	545.5	430.8	616.7	560.4	469.6	3.0	9.9	3.4
ger	5.2	2.8	4.0	4.7	4.9	4.6	3.0	2.8	2.7
gls	12.4	11.4	9.8	12.0	15.3	8.9	5.0	19.1	16.6
hep	4.8	2.5	3.8	4.5	3.6	2.4	10.0	9.1	4.0
hrtc	7.7	6.0	7.2	11.7	6.4	10.1	8.0	10.9	13.2
hrts	6.6	4.8	2.9	7.6	5.9	3.9	18.0	12.0	3.5
ion	5.5	4.3	4.6	5.7	4.9	3.6	7.0	12.3	3.8
irs	3.1	3.2	3.0	3.6	3.4	3.5	3.0	24.8	3.7
jvow	438.8	233.1	232.9	412.6	404.4	46.9	15.0	12.0	17.2
krkopt	3728.7	2964.0	2738.3	3600.7	3504.4	1241.4	5.0	13.1	49.0
letter	1354.1	905.7	882.8	1255.4	1260.7	363.4	20.0	17.5	75.3
liv	11.1	8.8	7.4	10.2	10.1	11.5	5.0	20.2	4.4
lym	5.5	4.2	3.3	7.8	5.1	2.7	5.0	11.8	9.5
magic	139.2	65.6	66.2	97.9	111.7	284.9	7.0	4.7	3.0
msh	18.6	10.4	10.4	19.6	21.3	6.2	9.0	9.4	3.5
nurse	217.1	132.4	120.1	231.6	223.1	71.6	10.0	6.1	6.4
page	27.8	18.0	14.6	23.9	22.6	23.6	6.0	4.0	6.2

Continued on next page

Table 3.28 – continued from previous page

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
pen	209.4	98.4	80.4	196.8	188.3	44.0	28.0	12.8	19.6
pid	10.6	9.1	9.2	7.0	10.0	15.9	6.0	15.3	3.4
psd	2.0	2.0	2.0	2.0	2.0	5.9	2.0	10.2	2.0
sb	5.7	3.1	2.2	4.1	2.8	21.2	1.0	3.9	2.0
seg	38.2	22.1	21.1	31.6	37.9	18.1	6.0	14.5	12.3
shuttle	27.1	25.0	27.9	24.8	24.2	40.3	4.0	9.0	7.5
sick	13.6	9.1	11.9	13.9	14.8	11.7	2.0	2.9	3.0
son	7.3	4.3	3.6	6.5	6.1	2.7	7.0	29.8	4.6
spect	5.2	3.0	3.5	5.4	3.4	2.6	2.0	2.6	3.1
spf	56.2	50.3	43.4	80.0	55.6	46.2	6.0	11.0	16.1
thy	7.2	5.4	4.9	7.1	7.9	24.2	3.0	2.9	4.4
ttt	56.0	33.8	20.5	48.7	43.4	19.8	31.0	11.5	4.4
veh	30.7	23.0	28.8	37.6	32.8	16.7	6.0	18.7	9.4
vene	6.0	4.8	4.6	5.4	4.5	6.8	5.0	31.6	4.8
vote	4.4	2.8	2.7	3.8	4.1	3.4	6.0	3.3	3.0
vow	94.5	63.6	48.5	90.2	90.8	35.0	18.0	22.9	39.4
w21	71.7	29.4	17.9	75.0	59.3	45.2	9.0	17.0	4.1
w40	73.1	20.1	16.0	67.0	55.4	33.9	18.0	17.3	4.3
wfr	19.9	25.3	23.8	19.5	20.3	72.2	3.0	8.2	9.0
wilt	19.3	12.7	13.6	17.5	18.1	18.2	2.0	3.4	2.1
wine	529.1	414.4	352.6	498.0	504.3	198.6	7.0	9.2	11.6
zoo	5.9	6.0	5.6	4.5	6.0	5.4	7.0	13.2	7.1

Table 3.29: *The 95% confidence intervals for the sizes of the induced DTs by all algorithms from the Table 3.7, on all datasets from the Table 2.1 from five 5-fold cross-validation test.*

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
adult	±23.9	±17.1	±8.8	±16.4	±30.2	±7.8	±0.0	±3.3	±0.2
ausc	±3.8	±1.4	±2.0	±2.9	±2.4	±0.8	±0.0	±0.8	±0.1
bank	±29.9	±22.2	±5.7	±19.8	±23.6	±5.8	±0.0	±0.2	±0.2
bc	±6.4	±2.1	±1.8	±4.0	±4.5	±1.1	±0.0	±1.6	±0.2
bch	±146.2	±120.4	±142.2	±130.1	±150.5	±10.3	±0.0	±1.9	±5.0
bcw	±3.0	±0.7	±0.9	±1.9	±1.7	±0.7	±0.0	±1.2	±0.2
ca	±2.4	±2.3	±0.7	±2.8	±3.9	±1.5	±0.0	±0.8	±0.0
car	±5.1	±3.7	±3.6	±5.6	±4.9	±1.5	±0.0	±1.3	±0.3
cmc	±35.2	±14.0	±12.6	±18.2	±13.2	±2.8	±0.0	±1.1	±0.2
ctg	±17.0	±11.9	±9.0	±13.8	±15.5	±2.0	±0.0	±1.1	±0.8
cvf	±113.7	±10.3	±9.5	±74.6	±63.5	±3.8	±0.0	±2.0	±0.3
eb	±731.7	±778.8	±587.5	±1009.6	±740.6	±14.3	±0.0	±6.1	±1.9
eye	±76.7	±73.4	±58.2	±86.3	±95.3	±8.9	±0.0	±1.1	±0.2
ger	±1.2	±0.9	±1.4	±1.3	±1.4	±0.7	±0.0	±0.6	±0.2
gls	±3.8	±2.6	±2.4	±3.5	±3.5	±1.2	±0.0	±3.6	±0.3
hep	±1.7	±0.4	±1.2	±1.8	±1.2	±0.4	±0.0	±1.7	±0.1
hrtc	±5.4	±2.5	±3.5	±6.1	±4.0	±1.0	±0.0	±1.9	±0.6
hrts	±2.3	±2.2	±1.0	±2.7	±2.8	±0.6	±0.0	±1.2	±0.2
ion	±1.9	±1.2	±1.0	±2.0	±1.4	±0.4	±0.0	±1.2	±0.2
irs	±0.2	±0.2	±0.1	±0.5	±0.4	±0.3	±0.0	±2.9	±0.4

Continued on next page

Table 3.29 – continued from previous page

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
jvow	±32.4	±19.7	±21.9	±30.3	±31.9	±2.6	±0.0	±1.6	±0.9
krkopt	±102.1	±98.0	±97.4	±133.2	±148.1	±20.5	±0.0	±1.9	±1.2
letter	±19.8	±80.4	±91.1	±81.7	±79.4	±6.3	±0.0	±2.5	±2.2
liv	±4.7	±3.9	±3.5	±4.3	±4.4	±1.1	±0.0	±1.6	±0.2
lym	±1.8	±1.2	±0.8	±2.7	±1.9	±0.3	±0.0	±1.4	±0.6
magic	±28.7	±11.2	±23.5	±24.0	±35.1	±6.4	±0.0	±0.5	±0.0
msh	±0.8	±0.7	±1.0	±1.6	±1.4	±0.5	±0.0	±1.3	±0.2
nurse	±5.6	±9.5	±17.7	±13.4	±24.6	±2.0	±0.0	±1.4	±0.3
page	±7.9	±6.9	±3.0	±6.0	±5.9	±3.1	±0.0	±0.4	±0.2
pen	±15.7	±10.8	±9.4	±15.7	±14.1	±2.3	±0.0	±1.9	±0.6
pid	±6.0	±4.3	±3.8	±2.4	±5.7	±1.0	±0.0	±1.2	±0.2
psd	±0.0	±0.0	±0.0	±0.0	±0.0	±0.5	±0.0	±2.5	±0.1
sb	±2.9	±1.1	±0.3	±3.8	±1.1	±2.2	±0.0	±0.4	±0.0
seg	±5.2	±4.2	±3.9	±5.5	±5.1	±1.0	±0.0	±1.2	±0.6
shuttle	±2.2	±1.5	±2.9	±2.6	±2.7	±1.6	±0.0	±0.7	±0.2
sick	±2.6	±2.2	±4.1	±2.5	±3.4	±1.4	±0.0	±0.2	±0.1
son	±2.1	±1.4	±1.1	±1.9	±1.9	±0.2	±0.0	±2.0	±0.3
spect	±2.9	±0.9	±1.2	±2.9	±1.7	±0.4	±0.0	±0.8	±0.1
spf	±14.5	±17.1	±16.9	±19.7	±16.9	±2.5	±0.0	±1.4	±0.7
thy	±0.6	±1.0	±0.5	±1.0	±1.0	±2.1	±0.0	±0.3	±0.2
ttt	±10.1	±8.7	±7.1	±10.1	±11.1	±1.5	±0.0	±2.1	±0.2
veh	±8.6	±7.7	±7.9	±12.3	±9.3	±0.9	±0.0	±2.3	±0.3
vene	±1.9	±1.7	±1.1	±1.4	±1.4	±1.1	±0.0	±2.1	±0.2
vote	±1.4	±0.7	±0.7	±1.5	±1.4	±0.4	±0.0	±0.7	±0.0
vow	±6.3	±7.4	±5.9	±7.3	±6.9	±1.7	±0.0	±1.6	±1.1
w21	±25.8	±13.9	±5.4	±19.5	±13.8	±1.9	±0.0	±1.3	±0.1
w40	±28.9	±5.1	±3.4	±19.2	±19.6	±2.5	±0.0	±0.9	±0.2
wfr	±2.0	±3.1	±3.5	±1.9	±2.2	±1.6	±0.0	±0.7	±0.4
wilt	±4.0	±3.1	±4.5	±3.6	±3.3	±3.0	±0.0	±0.3	±0.1
wine	±109.0	±93.3	±89.3	±105.5	±105.5	±3.9	±0.0	±0.7	±0.4
zoo	±0.6	±0.6	±0.5	±0.9	±0.9	±0.2	±0.0	±2.0	±0.1

Table 3.30: *The relative differences in sizes of the DTs induced by the algorithms from the Table 3.7, compared to the DTs induced by the* EFTI *algorithm on the same dataset.*

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree
adult	2988.6	1864.3	1095.7	1860.0	3097.1	18737.1	7.1	221.4
ausc	160.8	39.2	58.1	74.3	56.8	163.5	170.3	160.8
bank	4187.7	1771.9	643.9	3221.1	3070.2	18173.7	−12.3	22.8
bc	282.9	48.6	43.8	138.4	173.3	61.0	139.7	140.4
bch	69.2	8.7	19.7	58.9	65.7	39.4	−96.6	−92.5
bcw	345.8	25.4	32.2	135.6	147.5	44.1	27.1	349.2
ca	98.7	102.7	6.7	189.3	212.0	178.7	166.7	138.7
car	557.9	360.9	252.3	500.5	446.2	234.5	−61.9	−40.1
cmc	979.9	235.8	294.3	435.2	461.6	842.8	−37.1	86.8
ctg	273.3	166.8	45.5	201.6	174.9	118.2	−44.3	−37.9
cvf	6511.6	329.5	347.3	2554.1	2314.5	1624.2	−75.8	57.5
eb	2876.9	3300.2	1795.1	4159.8	2789.4	4863.9	−94.4	−88.3

Continued on next page

Table 3.30 – continued from previous page

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree
eye	15989.4	15943.5	12571.8	18037.6	16381.2	13710.6	−11.8	191.8
ger	92.5	6.0	50.7	74.6	82.1	71.6	11.9	4.5
gls	−25.1	−31.3	−41.2	−28.0	−7.7	−46.5	−69.9	14.9
hep	19.0	−38.0	−6.0	12.0	−11.0	−40.0	150.0	127.0
hrtc	−41.7	−54.7	−45.9	−11.8	−51.4	−23.9	−39.6	−17.8
hrts	90.8	37.9	−17.2	119.5	69.0	12.6	417.2	244.8
ion	42.7	12.5	18.7	47.9	27.1	−6.2	82.3	220.8
irs	−16.1	−14.0	−18.3	−2.2	−9.7	−5.4	−19.4	566.7
jvow	2445.5	1252.2	1251.0	2293.0	2245.5	172.2	−13.0	−30.2
krkopt	7515.8	5954.0	5492.9	7254.3	7057.7	2435.6	−89.8	−73.3
letter	1697.8	1102.4	1072.1	1566.8	1573.8	382.5	−73.4	−76.8
liv	149.5	98.2	67.6	129.7	127.9	158.6	12.6	354.1
lym	−42.0	−56.3	−65.5	−18.1	−46.2	−71.8	−47.5	24.4
magic	4540.0	2085.3	2106.7	3162.7	3624.0	9396.0	133.3	55.6
msh	428.4	196.6	196.6	458.0	505.7	75.0	155.7	167.0
nurse	3292.5	1968.8	1776.9	3518.7	3385.6	1019.4	56.2	−4.4
page	351.3	192.2	137.0	287.7	266.9	282.4	−2.6	−35.7
pen	970.6	403.3	311.0	906.1	862.6	124.9	43.1	−34.8
pid	210.6	167.1	169.4	107.1	192.9	368.2	76.5	349.4
psd	−2.0	−2.0	−2.0	−2.0	−2.0	190.2	−2.0	400.0
sb	186.0	56.0	10.0	106.0	40.0	958.0	−50.0	96.0
seg	209.7	79.2	71.1	156.2	207.8	47.1	−51.3	17.5
shuttle	262.6	233.7	272.7	232.1	223.0	438.5	−46.5	20.9
sick	347.4	198.7	292.1	356.6	385.5	284.2	−34.2	−3.9
son	58.3	−7.0	−22.6	41.7	32.2	−40.9	52.2	547.8
spect	70.1	−1.3	13.0	74.0	11.7	−16.9	−35.1	−16.9
spf	248.4	212.2	169.5	396.0	244.7	186.4	−62.8	−32.0
thy	64.5	22.7	10.9	61.8	79.1	449.1	−31.8	−33.6
ttt	1171.8	667.3	366.4	1007.3	885.5	350.9	604.5	161.8
veh	227.8	145.3	208.1	302.1	250.0	78.2	−35.9	100.0
vene	23.1	0.0	−5.8	11.6	−6.6	39.7	3.3	553.7
vote	46.7	−6.7	−10.7	25.3	36.0	12.0	100.0	9.3
vow	139.6	61.2	23.0	128.8	130.1	−11.3	−54.4	−42.0
w21	1657.8	621.6	339.2	1738.2	1352.9	1007.8	120.6	315.7
w40	1608.4	369.2	272.9	1464.5	1193.5	692.5	320.6	304.7
wfr	120.4	179.6	162.8	115.9	124.3	698.7	−66.8	−8.8
wilt	828.8	509.6	553.8	742.3	771.2	775.0	−3.8	61.5
wine	4476.8	3485.1	2950.5	4207.6	4262.6	1617.6	−39.4	−20.4
zoo	−16.4	−15.3	−20.3	−36.7	−15.8	−23.7	−1.1	85.9

Table 3.31: *The ranking of the algorithms from the Table 3.7 and* EFTI *based on the induced DT accuracies, calculated using the procedure explained in the Section 2.8.*

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
adult	1	1	2	1	1	5	4	6	3
ausc	3	2	2	2	2	4	2	2	1
bank	1	2	2	1	1	6	4	5	3
bc	5	3	2	4	3	2	5	6	1

Continued on next page

Table 3.31 – continued from previous page

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
bch	2	3	3	2	2	4	5	5	1
bcw	3	2	2	2	2	2	2	2	1
ca	3	2	2	2	2	4	2	2	1
car	1	1	2	1	1	2	4	4	3
cmc	2	2	2	2	3	4	2	4	1
ctg	2	1	1	1	1	3	4	5	1
cvf	3	2	2	4	3	2	5	6	1
eb	2	1	1	1	1	1	3	3	1
eye	3	2	1	2	2	4	6	7	5
ger	2	1	3	1	1	4	3	2	1
gls	3	2	2	2	2	2	2	4	1
hep	3	2	2	2	2	2	2	2	1
hrtc	3	2	2	2	2	2	2	2	1
hrts	3	2	2	2	2	2	2	2	1
ion	3	2	2	2	2	3	2	4	1
irs	2	1	1	1	1	1	1	3	1
jvow	3	1	1	2	2	1	4	5	3
krkopt	1	2	5	3	4	6	8	8	7
letter	1	2	3	2	1	2	5	6	4
liv	3	2	2	2	2	2	4	5	1
lym	3	2	2	2	2	2	2	2	1
magic	3	1	1	2	2	5	6	7	4
msh	2	1	1	1	1	1	4	4	3
nurse	1	1	3	2	2	3	5	6	4
page	2	1	1	1	1	2	3	4	1
pen	2	1	1	1	1	1	3	4	1
pid	3	2	2	2	2	4	2	5	1
psd	1	1	1	1	1	2	1	3	1
sb	2	1	1	1	1	3	1	1	1
seg	1	1	1	1	1	1	2	3	1
shuttle	1	1	1	1	1	1	2	3	1
sick	2	3	4	1	1	5	6	6	4
son	3	2	2	2	2	2	2	4	1
spect	3	4	4	2	2	2	2	4	1
spf	1	1	2	1	1	2	3	4	1
thy	1	1	2	1	1	5	4	5	3
ttt	1	2	3	2	2	4	3	3	2
veh	3	2	2	2	2	1	4	5	1
vene	3	2	2	2	2	2	2	4	1
vote	3	3	3	2	2	4	2	2	1
vow	2	1	1	1	1	2	3	4	1
w21	4	2	2	3	3	2	5	6	1
w40	4	2	2	3	3	2	5	6	1
wfr	2	1	3	1	1	4	5	6	4
wilt	2	1	1	1	1	3	4	4	4
wine	2	1	1	1	1	2	3	4	1
zoo	2	2	2	2	2	2	2	2	1
rank	2.294	1.686	1.961	1.725	1.706	2.725	3.314	4.137	1.804

Table 3.32: *The ranking of the algorithms from the Table 3.7 and* EFTI *based on the induced DT sizes, calculated using the procedure explained in the Section 2.8.*

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
adult	3	2	1	2	2	4	1	1	1
ausc	2	1	1	1	1	2	2	2	1
bank	4	2	1	3	2	5	1	1	1
bc	3	1	1	2	2	1	2	2	1
bch	3	2	2	2	2	2	1	1	2
bcw	3	1	1	1	2	1	1	3	1
ca	2	1	1	2	2	2	2	2	1
car	5	3	2	4	3	2	1	1	1
cmc	3	1	1	1	2	2	1	1	1
ctg	3	2	1	2	2	2	1	1	1
cvf	3	1	1	2	2	2	1	1	1
eb	3	2	1	2	2	3	1	1	1
eye	3	2	2	3	3	2	1	1	1
ger	3	1	1	1	2	1	1	1	1
gls	3	2	1	2	2	1	1	4	2
hep	2	1	1	1	1	1	3	3	1
hrtc	2	1	1	1	1	1	1	1	1
hrts	2	1	1	2	1	1	4	3	1
ion	2	1	1	1	1	1	2	3	1
irs	2	1	1	1	1	1	1	3	1
jvow	4	2	2	3	3	1	1	1	1
krkopt	6	4	3	5	5	2	1	1	1
letter	5	3	3	4	4	2	1	1	1
liv	2	1	1	1	1	2	1	3	1
lym	2	1	1	2	1	1	1	3	3
magic	4	2	2	2	3	5	1	1	1
msh	4	3	3	4	5	2	3	3	1
nurse	4	3	3	4	4	2	1	1	1
page	3	2	2	2	2	2	1	1	1
pen	5	3	3	4	4	2	1	1	1
pid	2	1	1	1	1	2	1	2	1
psd	1	1	1	1	1	2	1	3	1
sb	2	1	1	1	1	3	1	1	1
seg	5	3	3	4	4	2	1	2	1
shuttle	4	3	3	3	3	5	1	2	1
sick	3	2	2	3	3	2	1	1	1
son	3	1	1	2	2	1	2	4	1
spect	2	1	1	1	1	1	1	1	1
spf	3	2	2	3	2	2	1	1	1
thy	4	2	2	3	3	5	1	1	1
ttt	4	2	2	3	3	2	2	1	1
veh	3	2	2	3	3	1	1	2	1
vene	2	1	1	1	1	1	1	3	1
vote	2	1	1	1	1	1	2	1	1
vow	6	4	3	5	5	2	1	1	2
w21	4	2	1	3	3	2	1	1	1
w40	4	1	1	3	2	2	1	1	1

Continued on next page

Table 3.32 – continued from previous page

	OC1-AP	CART-LC	OC1	OC1-ES	OC1-SA	NODT	GALE	GaTree	EFTI
wfr	4	5	3	3	3	6	1	2	2
wilt	3	2	2	2	2	2	1	1	1
wine	4	3	3	3	3	2	1	1	1
zoo	2	1	1	1	1	1	1	3	1
rank	3.176	1.843	1.627	2.294	2.275	2.059	1.275	1.706	1.118

The results of the experiments in this section show that the proposed *EFTI* algorithm is capable of inducing the DTs of the accuracies comparable to other well known incremental algorithms like CART-LC and OC1, but with the significant reduction in their sizes. This can be seen in the average rankings of the algorithms based on their accuracies and sizes. In terms of accuracy, the *EFTI* scored an average of 1.804, compared to 1.686 of CART-LC and 1.961 of OC1. On the other hand, when it comes to size, *EFTI* had a significantly higher average rank of 1.118, compared to 1.843 of CART-LC and 1.627 of OC1. When compared to full DT induction algorithms GALE and GaTree, *EFTI* was better in terms of induced DT size and significantly better when it came to DT accuracies.

4 Co-processor for the DT Induction - the *EFTIP*

Very few theorems exist about evolutionary algorithms that can be used to guarantee some aspect of their behavior, with probably the most famous results being the one that states that (1+1) ES takes $O(nlog(n))$ iterations to find a maximum of any linear function *[78]*. Even worse, the "No Free Lunch" theorem *[79]* implies that no optimization algorithm can have, on average, a superior performance when applied to many different problems, which means that usually optimization algorithms need to be specifically tuned for each problem. However, in order to find the optimal parameter set for an EA or test the efficiency of a new algorithm feature, for the lack of theoretical guidelines usually an experimental approach needs to be used. In order for the experiment to have a level of statistical significance, usually multiple runs of cross-validation technique are used.

For tuning the parameters and testing new features for the *EFTI* algorithm, five 5-fold cross-validations were performed for each dataset with 500k iterations, which for the largest of them, `shuttle` took almost 6 hours on a desktop PC (average induction times when partial reclassification is used can be found in Table 3.6). In order to find an optimal parameter set, some kind of meta-heuristic needs to be employed, where in each of its iterations such a cross-validation test would be needed to evaluate its current candidate solution. This would then amount to days or even weeks of processing time. Embedded CPUs are even less powerful and would take even more time for these operations. Hence, the application of the DT induction using EAs in a dynamically adaptable real-time embedded machine learning system would be impractical.

In this thesis, in an attempt to address the issue of long inference times, a co-processor called *EFTIP* (Evolutionary Full Tree Induction co-Processor) is proposed, that can be used to accelerate the operation of the full DT induction algorithms, hence the *EFTI* algorithm too, by an order of magnitude. The following topics will be covered in this section:

- Section 4.1 - Presentation of the results of the *EFTI* algorithm profiling that reveal its most computationally intensive parts, which in turn make good candidates for the hardware acceleration
- Section 4.2 - Overview of the existing hardware architectures for the DT classification acceleration
- Section 4.3 - Detailed description of the *EFTIP* co-processor.
- Section 4.4 - Analysis of the required hardware resources and the performance of the *EFTIP* co-processor
- Section 4.5 - Discussion on the software routines that need to be added to the *EFTI* algorithm, in order for it to make use of the *EFTIP* co-processor
- Section 4.6 - Experimental section that shows the speedups that can be achieved by using the *EFTIP* co-processor

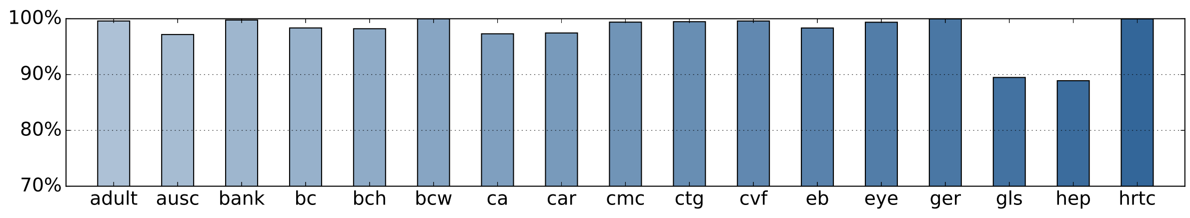
4.1 Profiling Results

It is clear from the equation (26) that the `fitness_eval()` function is a good candidate for the hardware acceleration, since it is the dominant contributor to the algorithm's time complexity. To confirm the results obtained by the computational complexity analysis, the software profiling was performed on the *EFTI* algorithm's C implementation. The *EFTI* algorithm was let to induce DTs from all datasets from the Table 2.1 in order to gather the profiling data. The software implementation of the *EFTI* algorithm was compiled using the GCC 5.4.1 compiler, run on the PC with 64-bit, 4-core, Intel i5-2500K CPU operating at approximately 3.5GHz, with 8GB or RAM, running Ubuntu 16.04 operating system and profiled using the GProf performance analysis tool for each of the datasets individually.

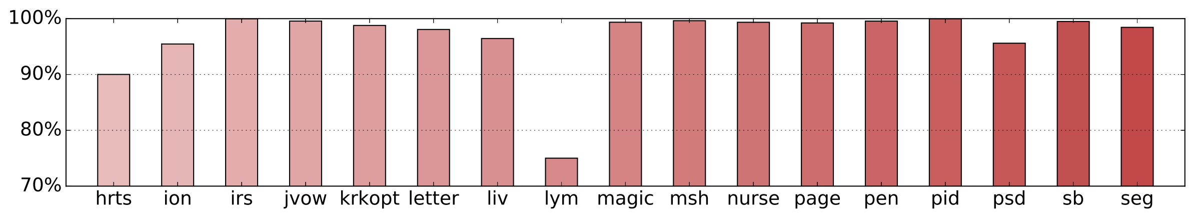
Table 4.1: *Percentages of the induction time that the* EFTI *algorithm spent on average in the sub-functions of the fitness evaluation task, given for each dataset.*

Dataset	FDLFI [%]	AC [%]	ENT [%]	ASPC [%]	Others [%]
adult	63.89	10.86	23.76	1.08	0.41
ausc	51.44	17.15	28.58	0.00	2.83
bank	75.57	6.53	16.79	0.88	0.23
bc	70.01	13.34	8.33	6.67	1.65
bch	67.92	22.59	6.84	0.86	1.79
bcw	71.44	21.43	7.14	0.00	0.01
ca	48.65	10.81	35.14	2.70	2.70
car	66.68	17.95	10.26	2.56	2.55
cmc	62.51	21.25	10.63	5.00	0.61
ctg	72.98	17.30	8.11	1.08	0.53
cvf	65.48	16.85	16.02	1.24	0.41
eb	74.25	16.01	7.46	0.62	1.66
eye	59.23	15.42	22.70	2.02	0.63
ger	69.40	2.04	28.58	0.00	0.02
gls	57.90	15.79	15.79	0.00	10.52
hep	55.56	22.23	11.11	0.00	11.10
hrtc	60.01	13.34	26.67	0.00	0.02
hrts	46.67	33.34	10.00	0.00	9.99
ion	63.64	9.09	22.73	0.00	4.54
irs	57.15	28.58	14.29	0.00	0.02
jvow	71.12	19.19	8.69	0.56	0.44
krkopt	68.80	17.49	11.79	0.70	1.22
letter	70.98	15.51	10.60	0.97	1.94
liv	78.58	0.00	17.86	0.00	3.56
lym	66.68	0.00	8.33	0.00	24.99
magic	64.70	13.07	20.13	1.46	0.64
msh	56.58	16.19	25.72	1.14	0.37
nurse	68.13	19.39	11.02	0.81	0.65
page	68.60	13.09	15.97	1.57	0.77
pen	72.02	19.41	7.41	0.72	0.44
pid	72.23	19.45	5.56	2.78	0.02
psd	63.24	13.24	19.12	0.00	4.40
sb	61.06	21.06	14.21	3.16	0.51
seg	65.11	19.27	13.02	1.04	1.56
shuttle	67.68	8.95	22.10	0.81	0.46
sick	47.73	21.03	28.98	2.27	0.01
son	0.00	9.09	90.92	0.00	0.01
spect	66.68	8.33	25.00	0.00	0.01
spf	65.42	15.04	18.05	0.00	1.49
thy	47.18	14.15	37.74	0.47	0.46
ttt	57.90	21.06	14.48	5.26	1.30
veh	57.38	19.67	21.31	0.00	1.64
vene	63.64	27.28	0.00	0.00	9.08
vote	45.01	25.00	20.00	0.00	9.99
vow	69.01	21.00	6.00	1.00	2.99
w21	53.16	20.00	25.76	0.55	0.53
w40	53.40	13.82	31.85	0.23	0.70
wfr	65.35	13.62	19.93	0.74	0.36

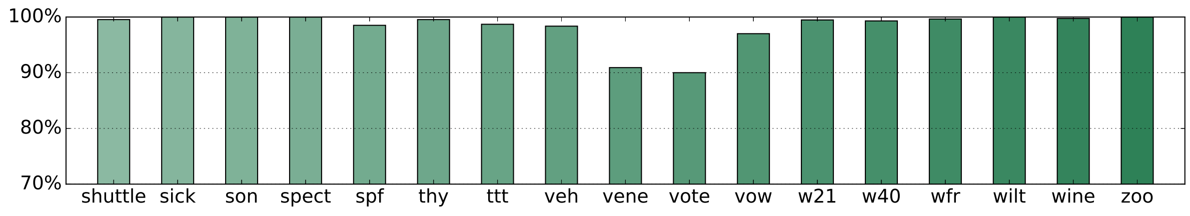
Continued on next page



***(a)** The datasets adult - hrtc*



***(b)** The datasets hrts - seg*



***(c)** The datasets shuttle - zoo*

***Figure 4.1:** The visual representation of the induction time percentages that the* EFTI *algorithm spent on average in the sub-functions of the fitness evaluation task, given for each dataset.*

Table 4.1 – continued from previous page

Dataset	FDLFI [%]	AC [%]	ENT [%]	ASPC [%]	Others [%]
wilt	68.19	14.77	13.07	3.98	0.01
wine	69.79	17.04	12.64	0.27	0.26
zoo	42.86	42.86	14.29	0.00	0.01
average	61.74	16.69	18.09	1.08	2.41

The results obtained by the profiling are listed in the Table 4.1 and displayed graphically in the Figure 4.1. The results displayed in the table represent the percentage of the induction time that the *EFTI* algorithm spent on average in the sub-functions of the fitness evaluation task: FLDFI - `find_dt_leaf_for_inst()`, AC - `accuracy_calc()`, ENT - `evaluate_node_test()` and ASPC - `apply_single_path_change()`, together with the percentage of the time spent inside all other functions given in the column titled "Others". The percentages given for the individual functions represent self-time, i.e. the execution time of the function without the execution times of its sub-functions. On the other hand, the Figure 4.1 shows, for each dataset, the percentage of the time spent in all fitness evaluation related functions combined.

The results presented in this subsection are consistent with the algorithm complexity analysis performed in the Section 3.4. On average, *EFTI* spent 99.0% of time calculating the fitness of the individual, hence the obvious computational bottleneck lays in the fitness evaluation task, which undoubtedly makes it a candidate for the hardware acceleration.

Since all other tasks (mutation, selection, initialization, etc.) take an insignificant amount of time on average to perform, it seems that there is no need to accelerate them in hardware. The *EFTI* algorithm can thus be implemented using HW/SW co-design architecture, where the fitness evaluation task would be implemented in hardware, while the rest of the functionality would remain in software. However, the *EFTI* algorithm could still benefit from moving all the remaining tasks to hardware too, since that would lower the communication overhead between the CPU and the custom hardware.

Nevertheless, for two reasons it was decided for the proposed *EFTIP* co-processor to accelerate only the `accuracy_calc()` function (and all of its sub-functions) from the fitness evaluation task, with the rest of the *EFTI* algorithm functionality left in software. The first reason is that it would be much more difficult to change and experiment with the fitness formula and the tasks of mutation, selection, initialization, etc. if they were implemented in hardware. Second reason is that many other evolutionary algorithms for optimizing the DT structure can then be implemented in software and make use of the hardware accelerated fitness evaluation task, like: Genetic Algorithms (GA), Genetic Programming (GP), Simulated Annealing (SA), etc. This fact significantly expands the potential field of use for the proposed *EFTIP* co-processor core.

4.2 Existing Architectures for Hardware Acceleration of the DT Classification

The accuracy of a DT is calculated by letting the DT classify the instances of a training set. The results of the DT classification are then compared with the known classification of the training set and the accuracy is calculated as a ration of the number of correct classifications to the total number of instances in the training set. The *EFTI* algorithm employs a sequential approach to performing this task, which is described in the Section 3.2.3.1.

First attempt at developing a hardware implementation of this procedure might be to implement every DT node as a separate hardware module, and connect the modules in the form of the DT. The hardware architecture based on this idea is proposed in *[80]*, and shown in the Figure 4.2.

The instance that is to be classified is distributed among all hardware DT nodes, where it is used in the node test evaluations. All the DT classes are made available on the inputs of the Demultiplexer (Figure 4.2). Starting from the root, the node tests are evaluated sequentially along the classification path of the instance, and based on their results the correct class for the output of the Demultiplexer is selected.

The hardware architecture proposed in *[80]* has two major drawbacks, one regarding the amount of hardware resources needed, and the other regarding the time needed to perform the classification. First, the architecture needs one hardware module per DT node, which in turn requires a significant amount of resources in order to be able to perform the dot product calculation of the node test (equation (1)). Second, the time needed to perform the classification is proportional to the depth of the DT and to the time needed to perform the node test calculation. In other words, this architecture does not scale well with the size of the DT.

One possible way of decreasing the classification time using this architecture is to perform all node tests in parallel. This is akin to what has been suggested in *[66]*, where an equivalence between decision trees and threshold networks is used to devise a hardware architecture for

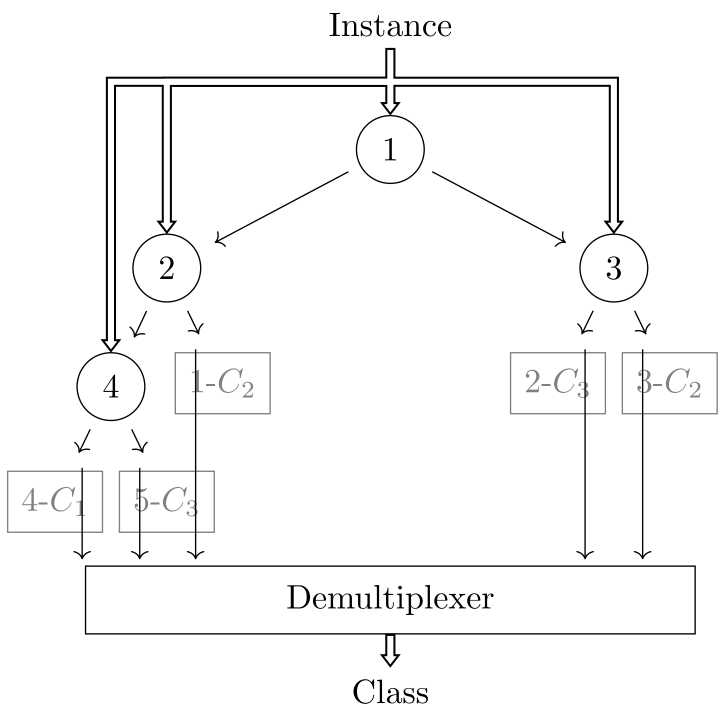


Figure 4.2: *The DT classification hardware implementation using one hardware module per DT node*

decision tree classification, where all node tests are performed in parallel. Once all of the node tests have been evaluated, their results can be combined using a boolean function in order to determine in which node the instance finished the classification, and hence to which class it should be classified into. This way, the time needed to perform the classification equals the time needed to evaluate one node test, plus the time needed to evaluate the output boolean function. Still, the issue with number of node hardware modules remains.

The architecture that remedies both resource and timing problems, and was thus adopted by the *EFTIP* co-processor, is proposed in *[56]* and called *SMPL* (Single Module Per Layer). Instead of implementing each DT node in hardware separately, this architecture requires only one universal node per DT level, which is in turn used to evaluate the tests of all nodes from that DT level. The fact that makes this solution possible is that the instances traverse the DT only in one direction from top to bottom, never returning to already visited nodes.

The Figure 4.3 shows the structure of the *SMPL* architecture implementation for the same example DT used in the Figure 4.2. The architecture implementation consists of three universal nodes L_1 through L_3, one for each of the DT levels that contain non-leaf nodes. The instance starts its traversal of the DT by being input to the L_1 module, which implements the root DT node in every *SMPL* architecture implementation. The universal node L_1 evaluates the root node test and passes the instance along with the test results to the L_2 module, which is akin to the instance continuing its traversal to the level 2 of the DT. The L_2 module has the capability of calculating the node test for all the nodes on the level 2 of the DT, in this case node #2 and #3. Based on the root node test results received from L_1, the L_2 module knows to which root child the instance has been passed, and thus the appropriate level 2 node test is evaluated, whose results, together with the instance, are in turn passed to its successor, and this process is continued until one of the universal nodes detects that the instance has arrived to a leaf node,

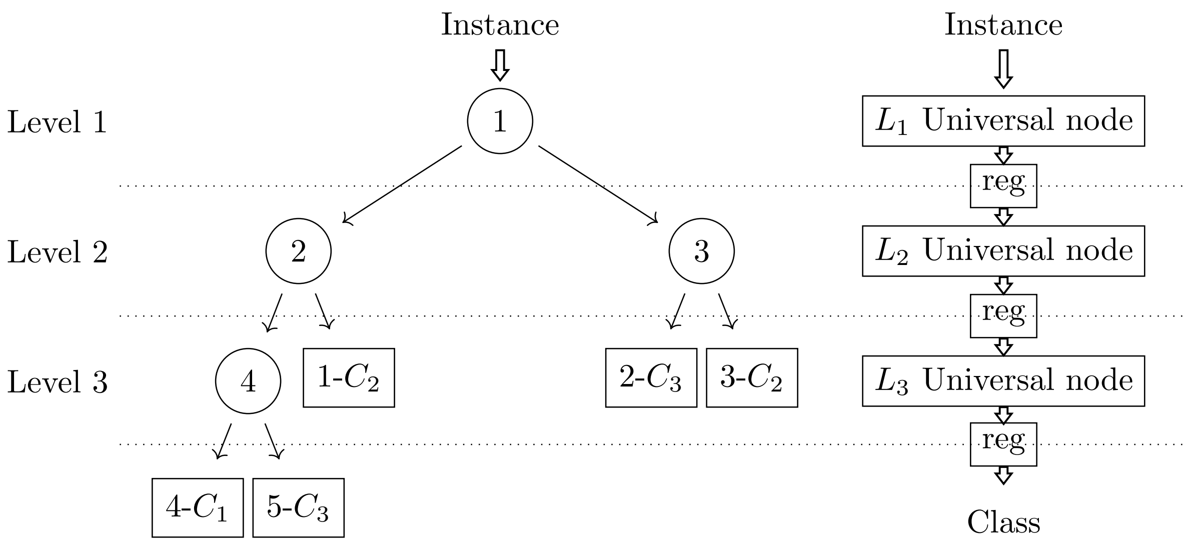


Figure 4.3: *The idea behind the* SMPL *(Single Module Per Layer) architecture. There is one universal hardware module (Universal nodes $L_1 - L_3$) per DT level that implements all DT nodes on the level.*

i.e. it has been classified. Thereafter, the information about the class is passed onward and following universal nodes perform no test evaluations on this instance. Finally, the last module of the *SMPL* architecture outputs the class of the instance.

The *SMPL* is a pipelined architecture, hence the instances can be effectively classified in parallel on all universal nodes, with the small cost of an initial pipeline latency. The node test evaluation results calculated by an universal node, that are to be made available to the next universal node in pipeline, are stored in the register available between every two nodes (blocks named *reg* in the Figure 4.3). That way, once the node test is evaluated for an instance and stored in the output register, the universal node is free to start processing the following instance from the dataset, while the next universal node in pipeline utilizes the stored results from the register.

The *EFTIP* co-processor classification module was decided to be based on the *SMPL* architecture as it requires significantly less hardware resources for the implementation then the architectures *[80]* and *[66]*. In order to evaluate oblique DT node tests, the addition, multiplication and comparison operations are needed. Hence, the *SMPL* architecture requires notably less adders, multipliers and comparators then architectures proposed in *[80]* and *[66]*. However, the memory resources requirements for storing the node test coefficients and leaf classes are identical between all three given architectures.

4.3 *EFTIP* Detailed Description

As it was discussed in the section Section 4.1, the *EFTIP* is designed to accelerate the most time consuming task of the evolutionary DT induction algorithms, which is the task of determining the accuracy of the DT individual, which is in turn needed for the fitness evaluation of the DT (the Algorithm 3.2). More precisely, the *EFTIP* co-processor calculates the number of successful classifications, i.e. the number of classifications hits - the `hits` variable of the Algorithm 3.3.

The *EFTIP* co-processor is designed as an IP core and embedded to the SoC through the interconnect interface AXI4 AMBA bus. The ARM Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for the connection and management of functional blocks in system-on-a-chip (SoC) designs. Today, AMBA is widely used on a range of ASIC and SoC parts including the processors used in modern portable mobile devices like smartphones. Via the AXI4 bus, the software running on the CPU can completely control the *EFTIP* operation:

- Download the training set
- Download the DT description, including the structural organization and the coefficient values for all node tests present in the DT
- Start the accuracy evaluation process
- Read-out the classification performance results

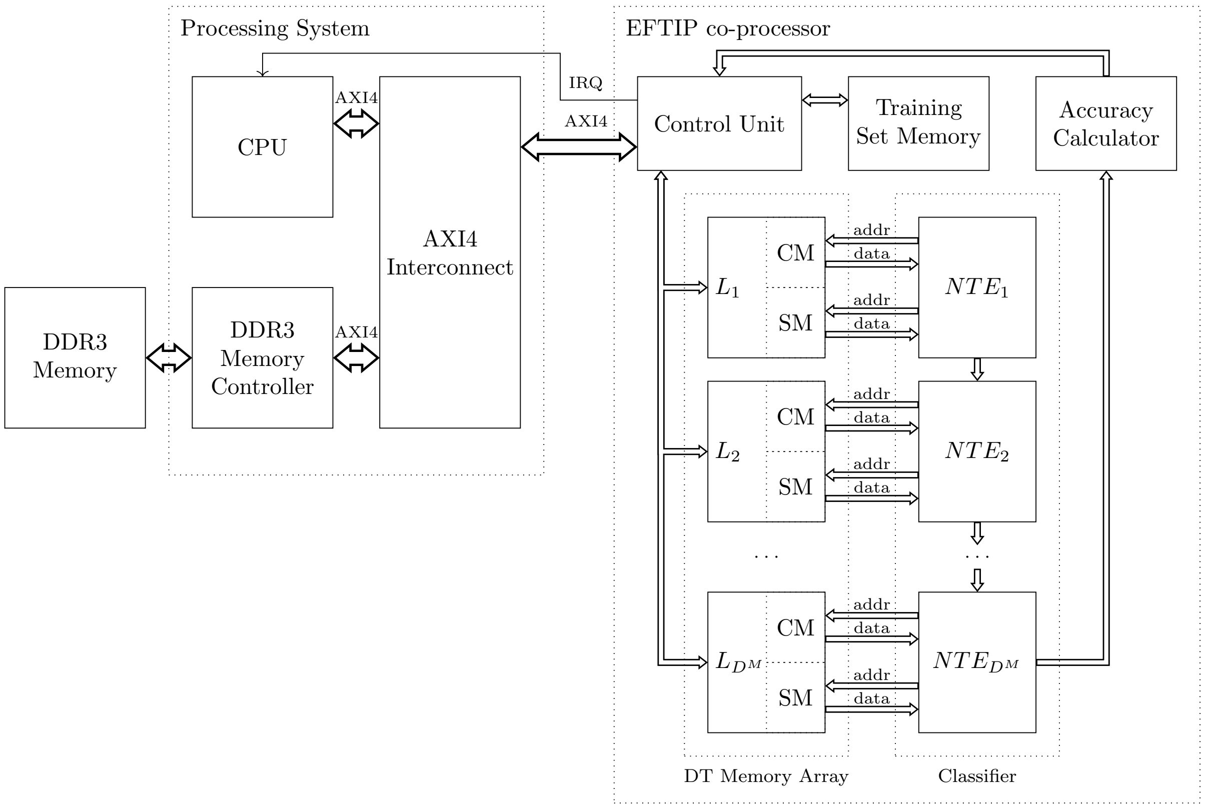


Figure 4.4: *The* EFTIP *co-processor structure and integration with the host CPU*

The major components of the *EFTIP* co-processor and their connections are depicted in the Figure 4.4:

- **Classifier** (Section 4.3.1) - Performs the DT traversal for each training set instance, i.e. implements the `find_dt_leaf_for_inst()` function from the Algorithm 3.4. The classification process is pipelined using a number of Node Test Evaluator modules (*NTEs*) corresponding to the universal nodes of the *SMPL* architecture, with each NTE performing the DT node test calculations for one DT level. The parameter D^M is the number of pipeline stages and thus the maximum supported depth of the induced DT. For each instance in the training set, the Classifier outputs the ID assigned to the leaf in which the instance finished the traversal (please refer to the `accuracy_calc()` function from the Algorithm 3.3).

- **Training Set Memory** (Section 4.3.2) - The memory for storing all training set instances that should be processed by the *EFTIP* co-processor.
- **DT Memory Array** (Section 4.3.3) - The array of memories used for storing the DT description, composed of sub-modules L_1 through L_{DM}. Each Classifier pipeline stage requires its own memory that holds the description of all nodes at the DT level it is associated with. Each DT Memory sub-module is further divided into two parts: the CM (Coefficient Memory - memory for the node test coefficients) and the SM (Structural Memory - memory for the DT structural information).
- **Accuracy Calculator** (Section 4.3.4) - Based on the classification data received from the Classifier, it calculates the accuracy of the DT and keeps track of which training set classes were found to be dominant for each of the DT leaves. For each instance of the training set, the Classifier supplies the ID of the leaf in which the instance finished the DT traversal. Based on this information the Accuracy Calculator updates the distribution matrix. After all the instances have been classified, it calculates the accuracy results and forwards them to the Control Unit, where they can be read by the user.
- **Control Unit** (Section 4.3.5) - Acts as a bridge between the AXI4 interface and the internal protocols. It also controls the accuracy evaluation process and generates an IRQ (Interrupt ReQuest) when the calculation is done.

4.3.1 Classifier

The classifier module performs the classification of an arbitrary set of instances on an arbitrary binary oblique DT. As it was already discussed in the Section 4.2, the Classifier module was implemented by modifying the *SMPL* architecture described in *[56]*. The original architecture from *[56]* was designed to perform the classification using already induced DTs, hence it was adapted so that it could be used with the *EFTI* algorithm for the DT induction as well.

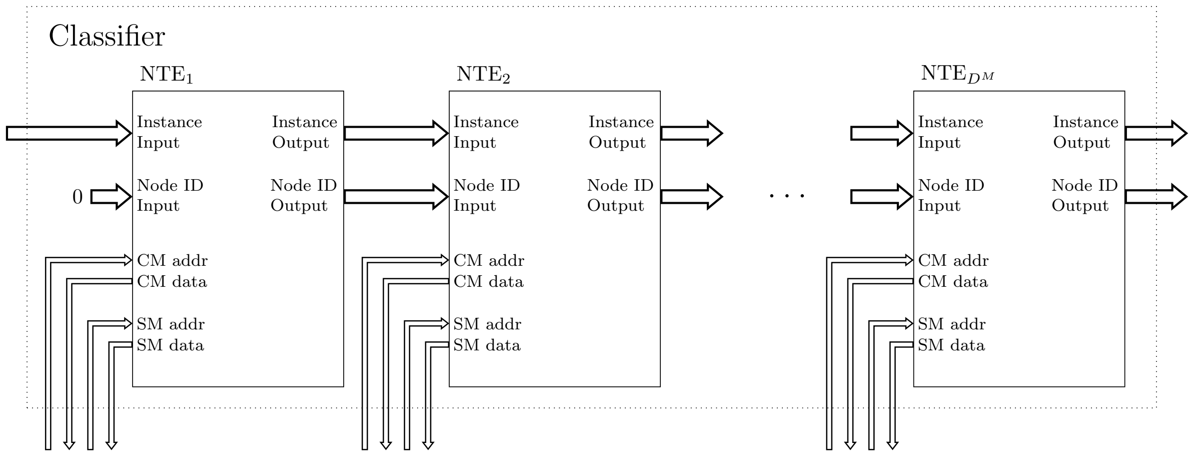


***Figure 4.5:** The architecture of the Classifier module consisting of the* NTE *modules connected in an array.*

In order for the *EFTIP* co-processor to calculate the accuracy of a DT on a dataset, the Classifier needs to perform the DT traversal for each instance of the dataset, i.e. it needs to implement the `find_dt_leaf_for_inst()` function from the Algorithm 3.4 in hardware. As it was discussed in the Section 4.2, during the traversal of an instance, only one node per DT level is visited, i.e. only one node test is performed per DT level for a single instance. Hence, a single module that implements the evaluation of the oblique node test (equation (1)), could be used to incorporate the test evaluations for all nodes on one DT level. Naturally, this module needs to be programmable in that it has to support the node test evaluation with different coefficient vectors in order to be able to evaluate tests for all nodes residing at the same DT level. However, the programmability is needed also since the *EFTIP* co-processor is used for the DT induction, hence the node test coefficients are not known in advance and can change over the course of the induction.

The Figure 4.5 shows the Classifier module as being composed from *NTE* modules, each of which is associated with one DT level, and implements the node test evaluation procedure for all nodes on that DT level. The *NTE* modules correspond to the universal nodes of the *SMPL* architecture. During the traversal of the DT, an instance always descends one DT level at the time, and never returns to the levels it already visited. The *NTE* modules are thus connected into a chain, where an instance is transferred from the first *NTE* to the last one in the chain, in order to calculate its DT traversal path. The number of *NTEs* the Classifier comprises - D^M, determines the maximum depth of the DT whose accuracy can be calculated by that hardware instance of the *EFTIP* co-processor. The D^M value can be specified by the user during the design phase of the *EFTIP* co-processor.

Since an instance always travels down the *NTE* chain, one *NTE* at a time, there is no reason why multiple instances could not traverse the chain simultaneously. The moment the NTE_1 evaluated the node test for the first instance of the dataset and the instance was transferred to the NTE_2, the NTE_1 becomes free to evaluate the node test for the next instance in the dataset. In other words, the *NTE* modules can form the pipeline, with one stage per DT level, capable of accommodating D^M instances in parallel, after the initial latency during which the pipeline is filled.

The NTE_1 always processes the root DT node. However, which nodes are processed by other stages depends on the path of the traversal for each individual instance. Hence each *NTE* module needs to have access to the descriptions of all the nodes on the DT level associated with it. Since each stage of the *NTE* pipeline needs to operate in parallel (in a distributed manner), the node description data needs to be distributed as well, and thus each stage has one sub-module of the DT Memory Array assigned to it that holds the descriptions of all the nodes on the DT level associated with that stage. Furthermore, each DT Memory Array sub-module is divided into two parts in order to save on some *NTE* hardware resources, namely *CM* and *SM*, because the data from these two memory parts is needed at different times in the node test evaluation, which will be discussed in more detail in the following text. Therefore, each *NTE* contains interfaces, comprising the address and data buses, for accessing the *CM* and *SM* parts of the assigned DT Memory Array sub-module: *CM addr*, *CM data*, *SM addr* and *SM data*.

When an instance is transferred from one *NTE* module to the next, the decision via which node the traversal continues (made by evaluating the node test) needs to be communicated along with it too. There are two major cases that need to be handled differently:

1. the instance continues the traversal via one of the children of the node whose test has been evaluated by the current *NTE* module. In this case, the next *NTE* in the chain is sent

the ID of the child (non-leaf) node to which the instance should descend.

2. the instance has already been classified, in which case the traversal is finished. However, in order not to disturb the filled pipeline, the instance is nevertheless transferred down the *NTE* chain. In this case, the next *NTE* in the chain is sent the ID of the leaf in which the instances finished its traversal. Based on that, the next *NTE* will recognize that no further calculations need to be performed for this instance, and that it can simply pass leaf ID onward.

The inter-NTE interface comprises the following buses:

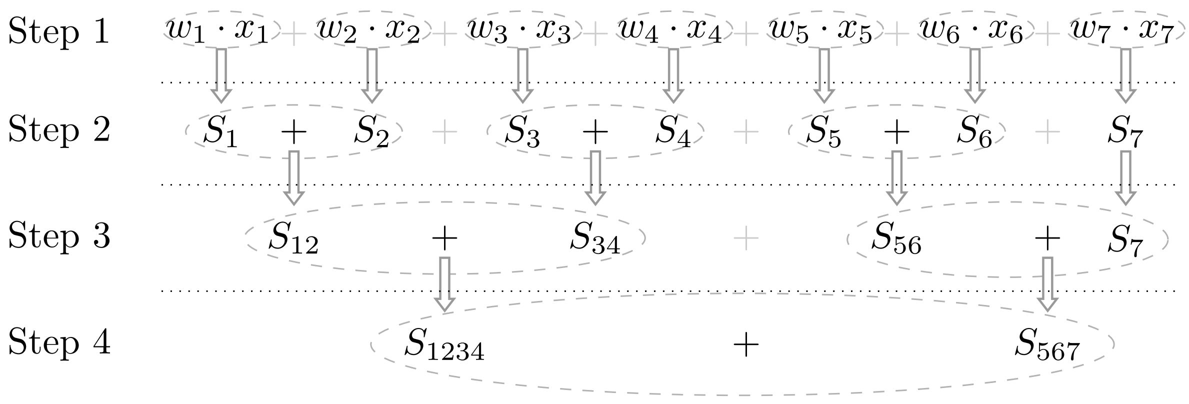
- **Instance bus** - Passes the instance to the next *NTE*, as the instance traverses the DT.
- **Node ID bus** - Passes to the next *NTE* either the ID of a non-leaf node, through which the traversal is to be continued, or the ID of a leaf node into which the instance has already been classified in some of the previous pipeline stages. The leaf and the non-leaf IDs are distinguished by the value of the node ID's MSB. If the value of the MSB is zero, the node ID is a non-leaf ID, otherwise it is a leaf ID.

For each instance, received at the Classifier input, the first NTE block processes the dot product calculation using the attributes of the received instance $\mathbf{x}$ and the root node coefficients $\mathbf{w}$. Based on the result, it then decides on how to proceed with the DT traversal: via the left or via the right child. The ID of the selected child node, which can either be a leaf or a non-leaf, is output via the *Node ID Output* port. If the selected child is a leaf node, the classification is done, and the next stages will perform no further calculations, but only pass forward the ID of the leaf into which the instance has been classified. On the other hand, if the selected child is a non-leaf node, the next stage will continue the traversal through the selected child by calculating the node test associated to it. The calculation of each NTE corresponds to one iteration of the `find_dt_leaf_for_inst()` function loop (Algorithm 3.4), and the NTE output *Node ID* corresponds to the `cur_node` variable, more specifically to its attribute `id`, which is in turn needed for the formation of the distribution matrix in the function `accuracy_calc()` of the Algorithm 3.3 (`leaf.id`). The node ID is output synchronously with the instance via the *Instance Output* port.

All subsequent stages operate in a similar manner, except that in addition, they also receive the calculation results from their predecessor stage. Somewhere along the NTE chain, all instances will finish the traversal into some leaf. The information about this leaf is finally output from the Classifier module via the *Node ID Output* port of the last *NTE* in the chain to the Accuracy Calculator module (together with the corresponding instance description via the *Instance Output* port) in order to update the distribution matrix and calculate the final number of classification hits.

4.3.1.1 Dot Product Parallelism

To evaluate a DT node test, each *NTE* needs to evaluate the dot product between node test coefficient vector $\mathbf{w}$ and instance attribute vector $\mathbf{x}$, which is at the same time by far the most complex operation of the *NTE* module. By extracting the parallelism from the dot product operation, additional speedup could be gained. The Figure 4.6 shows which parts of the dot product calculation can be performed in parallel on an example where $N_A = 7$. If we are only allowed to perform binary addition (which is usually the case when a hardware block performs is used for this task), the calculation could be performed in 4 steps, with all the operations performed in a single step circled with the dashed lines in the figure. In the Step 1 all the multiplications could be performed in parallel since there is no data dependency between them, while in later steps the N_A-ary addition is broken down into the sequence of binary addition operations, where all of them within the same step can be executed in parallel.



Step 1
$w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 + w_4 \cdot x_4 + w_5 \cdot x_5 + w_6 \cdot x_6 + w_7 \cdot x_7$
Step 2
$S_1 + S_2 + S_3 + S_4 + S_5 + S_6 + S_7$
Step 3
$S_{12} + S_{34} + S_{56} + S_7$
Step 4
$S_{1234} + S_{567}$

Figure 4.6: *The dot product calculated for $N_A = 7$, using binary multipliers and adders, broken into 4 steps inside which the operations can be performed in parallel.*

To take advantage of this dot product calculation parallelism, the *NTE* module is again internally pipelined, all in order to achieve the maximal possible throughput. Each of the steps (Figure 4.6) of the dot product calculation is mapped into one internal pipeline stage. The number of stages needed for the dot product pipeline equals 1 for the multiplication step, plus $\lceil ld(N_A) \rceil$ for N_A-ary addition to be performed via binary addition operations. There is never a need to flush the *NTE* pipeline, because of the nature of the DT accuracy calculation, where the instances enter the pipeline one by one in a predefined order, descend through the DT without making any loops and finally get classified, at which point the *NTE* needs to perform no further calculation on them, which in turn makes space for the rest of the instances to be processed.

4.3.1.2 Node Test Evaluator - NTE

The block diagram in the Figure 4.7 shows the architecture of the *NTE* module. When the value received at the *Node ID Input* of an *NTE* contains a non-leaf node ID, it tells the *NTE* which node's test is to be evaluated among all the nodes at the DT level on which that *NTE* module operates. The node test is performed on the dataset instance received at the *Instance Input* port together (at the same time) with the node ID. Each instance carries two types of information: the attribute vector x and the class *C* to which it belongs. The instance and the selected node together make a pair of objects that all procedures in the *NTE* module operate on, and in the text they are called: the current instance and the current node. Please notice that due to the pipelining, different stages of the *NTE* operate in fact on different current nodes and instances. The *NTE* expects the ID of a non-leaf node to equal the node's index in the list of all non-leaf nodes at the DT level on which the *NTE* operates. Hence, the non-leaf node IDs are local to, i.e. only unique within, the DT level they are at, and the node numbering restarts from 0 for each DT level. On the other hand, the leaf IDs need to be global, i.e. unique across the whole DT, since they will be used to identify which leaf was the instance classified into.

When the value received at the *Node ID Input* of an *NTE* contains a leaf node ID, this signals the *NTE* that the corresponding instance has already been classified, hence the dot product calculation is not performed (more precisely, in order to simplify the design it is still performed, but the results are discarded). The received node ID value is simply output verbatim via the "Node ID Output" port along with the corresponding dataset instance.

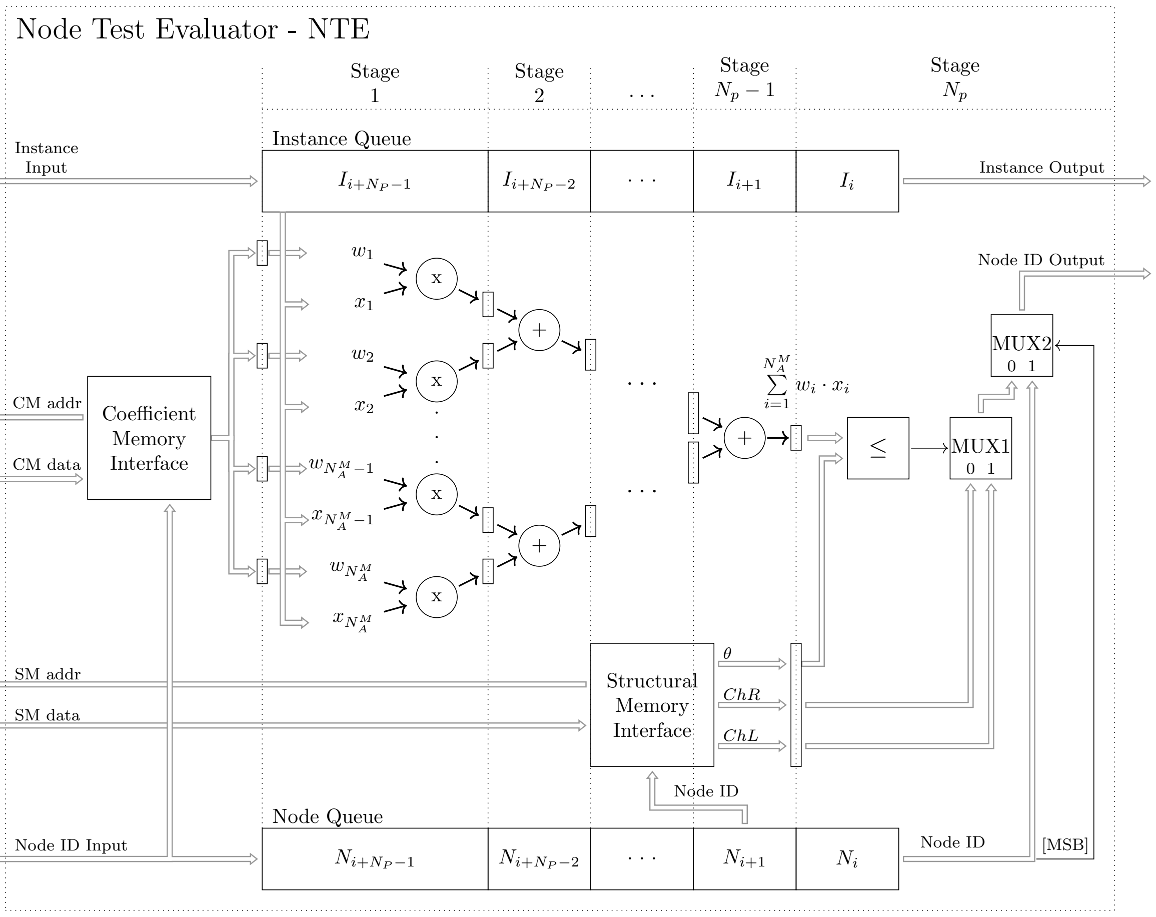


Figure 4.7: *The NTE (Node Test Evaluator) block architecture*

The Classifier hence performs the operations on the current node and instance in the following order:

1. The test coefficient vector $\mathbf{w}$ of the current node is fetched from the CM part of the DT Memory Array sub-module via the *Coefficient Memory Interface*. The current node's ID is used as an index to calculate the address of the node's coefficient vector in the CM memory, which is communicated via the *CM addr* port. If the current node is a leaf, the fetch is not performed and all zeros are loaded for the vector $\mathbf{w}$, but the results of the dot product are discarded anyway in this case.

2. The dot product between the fetched coefficient vector $\mathbf{w}$ and the attribute vector $\mathbf{x}$ of the current instance, is calculated in several steps discussed in the Section 4.3.1.1. First the multiplication step is performed in parallel, and then the obtained element-wise products are summed using the adder tree.

3. The current node's test threshold (θ) and the IDs of the current node's both children (ChL - the ID of the left child and ChR - the ID of the right child) are retrieved from the SM

part of the DT Memory Array sub-module, again using the current node ID as an index to calculate the address where this information is stored in the SM memory. Again, if the current node is a leaf, the fetch is not performed and all zeros are loaded for θ, ChR and ChL.

4. Finally, the decision on where to proceed with the DT traversal is made. If the current node is a non-leaf node, the MSB of its ID has a value 0, which makes the *MUX2* block forward the output of the *MUX1* to the *Node ID Output* port of the *NTE*. The output of the *MUX1* in turn depends on the result of the comparison between the dot product value and the value of θ, and either ChL or ChR is sent to the *Node ID Output*. On the other hand, if the current node is a leaf (meaning that the current instance has already been classified), the MSB of its ID has a value 1, which makes the *MUX2* block forward the current node's ID to the *Node ID Output*. Whichever the case may be, a node ID will be output via *Node ID Output* port along with the current instance via *Instance Output* port, and they will become the current node and the current instance for the next *NTE* in the chain.

The main parameter that needs to be specified by the user during the design phase of the *EFTIP* is the maximum supported number of attributes per instance - N_A^M, i.e. the maximum supported sizes of the vectors $\mathbf{w}$ and $\mathbf{x}$. This parameter affects the size and latency of the *NTE* module as it will be explained in the text that follows.

The *NTE* module's main task is the dot product calculation of the vectors $\mathbf{w}$ and $\mathbf{x}$. By using only two input multipliers and adders, this computation is parallelized and pipelined as much as possible as discussed in the Section 4.3.1.1. The multiplications are performed for all N_A^M coefficient and attribute pairs in parallel, while the tree of two input adders that is $\lceil ld(N_A^M) \rceil$ deep, is necessary to implement the summing operation. In order to achieve higher operating frequency of the implemented *EFTIP* co-processor, the dot product calculation datapath is broken into stages, with one stage per calculation step. Each step comprises multiplication or addition operations that can be performed in parallel. Finally, the outputs of each of the adder and multiplier blocks are registered to form the pipeline.

Second important parameter besides N_A^M, that needs to be specified by the user during the design phase of the *EFTIP* is R_A - the number of bits used for the signed fixed point representation of the elements of the vectors $\mathbf{w}$ and $\mathbf{x}$. Hence, the elements of $\mathbf{w}$ and $\mathbf{x}$ are considered to be in the Q0.($R_A - 1$) format. For an example, if 16 bits are used for the representation of the vector elements, they are considered to be in Q0.15 format. After the multiplication stage, the products will thus be in the Q0.($2R_A - 2$) signed fixed point format. The value of the sum, output by each adder, is larger by 1 bit than the value of its operands, hence the registers increase in size by 1 integer bit per pipeline stage. After the final addition, the sum representation will have reached the size of: $2R_A - 1 + \lceil ld(N_A^M) \rceil$ bits in the Q($\lceil ld(N_A^M) \rceil$).($2R_A - 2$) format. Finally, the value of the final sum, which is compared to the threshold θ, is truncated to the Q($\lceil ld(N_A^M) \rceil$).($R_A - 1 - \lceil ld(N_A^M) \rceil$) format in order to return to the operands of R_A bits in size. Consequently, the *NTE* expects the value of θ to be supplied encoded in the Q($\lceil ld(N_A^M) \rceil$).($R_A - 1 - \lceil ld(N_A^M) \rceil$) format. The *NTE* module also supports datasets with less than N_A^M number of instance attributes, $N_A < N_A^M$. In this case, the surplus coefficients $w_{N_A+1}, w_{N_A+2}, ...w_{N_A^M}$ should be all set to zero, in order not to affect the calculation of the sum.

The necessary number of bits used to encode the non-leaf node and leaf IDs - R_N, can be calculated based on the parameter D^M. Since the non-leaf node IDs are unique only across one DT level, of which the last level can have the largest number of nodes, and the D^M parameter limits the number of levels the induced DT can have, there is a maximum of 2^{D^M-1} different non-leaf node IDs to be encoded for the selected value of the parameter D^M. On the other hand, the leaf IDs need to be globally unique, hence there needs to be one ID available for each leaf in the DT. The possible number of leaves is also related to the parameter D^M, and equals 2^{D^M}. Additionally, the MSB of the ID representation is reserved for differentiating between the leaf and non-leaf node IDs, which finally means that the total number of bits for encoding IDs should be $R_N \geq D^M + 1$ if it is needed for the *EFTIP* co-processor to support complete binary DTs of depth D^M. Usually, if the DT individuals are given enough depth to grow, the *EFTI* algorithm will induce DTs that are much smaller than the complete binary DT of the same depth. Hence, depending on the dataset and other algorithm and co-processor configurations, it might be viable to lower the value of R_N.

The Figure 4.7 shows the *NTE* module partitioned in N_P pipeline stages by the vertical dotted lines, with each part labeled by the stage ID: Stage 1, Stage 2, ... Stage N_P. The total number of pipeline stages needed (N_P), equals the depth of the adder tree, plus the multiplication stage and the decision stage in the end where node test results are interpreted:

$$N_P = \lceil ld(N_A^M) \rceil + 2 \tag{27}$$

Prior to the Stage 1 of the *NTE*, the coefficients of the vector $\mathbf{w}$ are fetched from the *CM* memory, which seems like it requires a separate pipeline stage. However, this step was merged with the Stage N_P of the previous *NTE* to be performed together in a single clock cycle. This implementation choice saved both one clock cycle per *NTE* on the *EFTIP* co-processor latency, and on additional registers that would be needed were these tow steps implemented in two separate pipeline stages.

The Instance Queue and the Node Queue delay lines are necessary due to the pipelining. Each *NTE* performs calculations only for a single DT level, hence once the calculations is finished the instance needs to be transferred to the next *NTE* module in the Classifier chain. This transfer needs to correlate in time with the output of the node test evaluation results via the *Node ID output* port. Hence, the Instance Queue has to have the length equal to N_P, since it needs to delay the output of the instance to the next *NTE* module for N_P clock cycles, which are needed for the calculations.

The Node Queue is necessary for preserving the current node's ID (the signal *Node ID* in the Figure 4.7). If the current node is a non-leaf node, then in the pipeline Stage $N_P - 1$ its ID will be used to calculate the address of the node's structural description in the SM part of the DT Memory Array sub-module. This description comprises three values: the ID of the left child - ChL, the ID of the right child - ChR and the node test threshold value - θ, which are in turn needed in the last pipeline stage, where a decision on how to continue the traversal will be made. On the other hand if the current node is a leaf, then its ID is needed in the last pipeline stage to be output via *Node ID Output* to the next *NTE* in the chain.

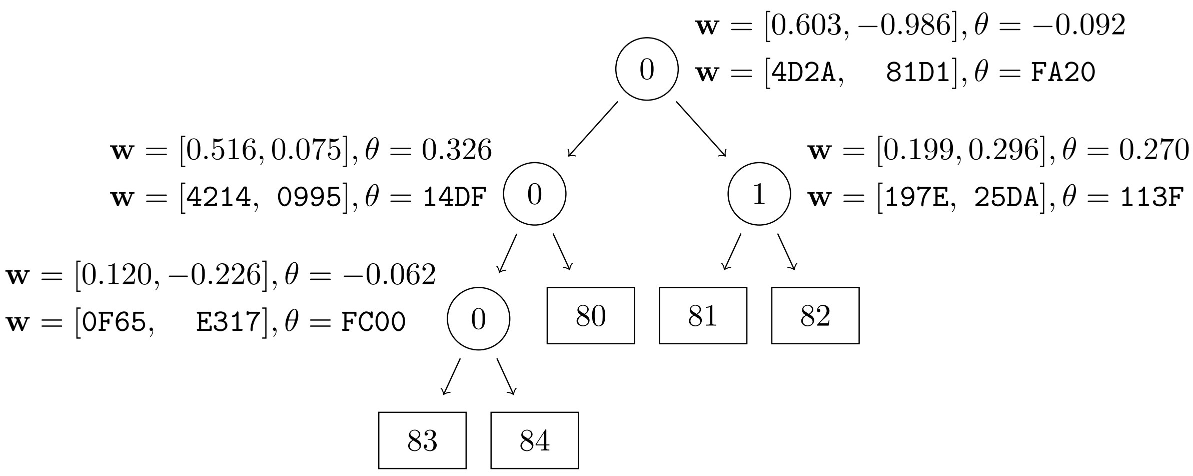
The operations in each pipeline stage depend only on the output of the previous stage, i.e. there are no loops in the design. This allows for each pipeline stage to start processing the next dataset instance immediately after it has finished with the current instance. The indices of the instances and nodes inside the *Instance Queue* and *Node Queue* reflect this feature. While Stage N_P processes the instance I_i, which is currently in the node N_i, the Stage $N_P - 1$ can process in parallel the next instance in the dataset, namely I_{i+1}, which is in the node N_{i+1}. Hence, the total of N_P instances are processed in different pipeline stages by a single *NTE* in parallel.

4.3.1.3 The Classifier Operation Example

Lets use the DT from the Figure 3.8a, whose induction from the `vene` dataset by the *EFTI* algorithm was discussed in the Section 3.1. First, the parameters compatible with the `vene` dataset need to be selected for the Classifier module. The `vene` training set instances are described using two attributes, $N_A = 2$, hence the minimum value that can be chosen for N_A^M is $N_A^M = N_A = 2$. For the sake of simplicity, in this example, N_A^M will be set to this minimum value of 2. The value of R_A can be chosen freely based on the accuracy that needs to be achieved during the dot product calculation, and here it will be set to 16, which should provide the high enough precision to obtain the highest possible classification results. The example DT is 3 levels deep, hence the D^M parameter needs to be set to at least that value. Even though the Classifier would provide correct results even if it contained more levels than that, for the sake of simplicity D^M will be set 3. Also, even though it would suffice to select $R_N = D^M + 1 = 4$, R_N will be set to 8 to gain on the readability of the leaf IDs. Based on these selections the other parameters can be calculated: $\lceil ld(N_A^M) \rceil$ = 1, N_P = 3, $\mathbf{w}$ and $\mathbf{x}$ elements format is Q0.15, and θ format is Q1.14. The list of all relevant parameters for the Classifier module is given in the Table 4.2.

Table 4.2: *The parameter set for configuring the* EFTIP *co-processor compatible with the* `vene` *dataset.*

D^M	N_A^M	R_A	R_N	$\lceil ld(N_A^M) \rceil$	N_P	FP format $\mathbf{x}$, $\mathbf{w}$	FP format θ
3	2	16	8	1	3	Q0.15	Q1.14



$\mathbf{w} = [0.603, -0.986], \theta = -0.092$
$\mathbf{w} = [\texttt{4D2A}, \texttt{81D1}], \theta = \texttt{FA20}$
0
$\mathbf{w} = [0.516, 0.075], \theta = 0.326$
$\mathbf{w} = [\texttt{4214}, \texttt{0995}], \theta = \texttt{14DF}$
0
1
$\mathbf{w} = [0.199, 0.296], \theta = 0.270$
$\mathbf{w} = [\texttt{197E}, \texttt{25DA}], \theta = \texttt{113F}$
$\mathbf{w} = [0.120, -0.226], \theta = -0.062$
$\mathbf{w} = [\texttt{0F65}, \texttt{E317}], \theta = \texttt{FC00}$
0
80
81
82
83
84

Figure 4.8: *The example DT used to discuss the* NTE *operation.* θ *and* $\mathbf{w}$ *are displayed for all nodes, first in decimal format and then in the fixed point representation immediately below.*

The Figure 4.8 shows the induced DT with the values of θ and $\mathbf{w}$ displayed for all nodes, first

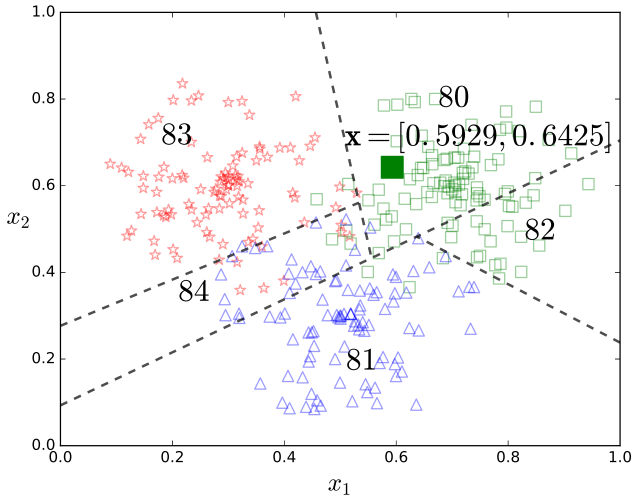


Figure 4.9: *The* `vene` *dataset with the marked instance that will be used for the Classifier module operation demonstration. The attribute space regions are titled by the leaf IDs that they are associated to.*

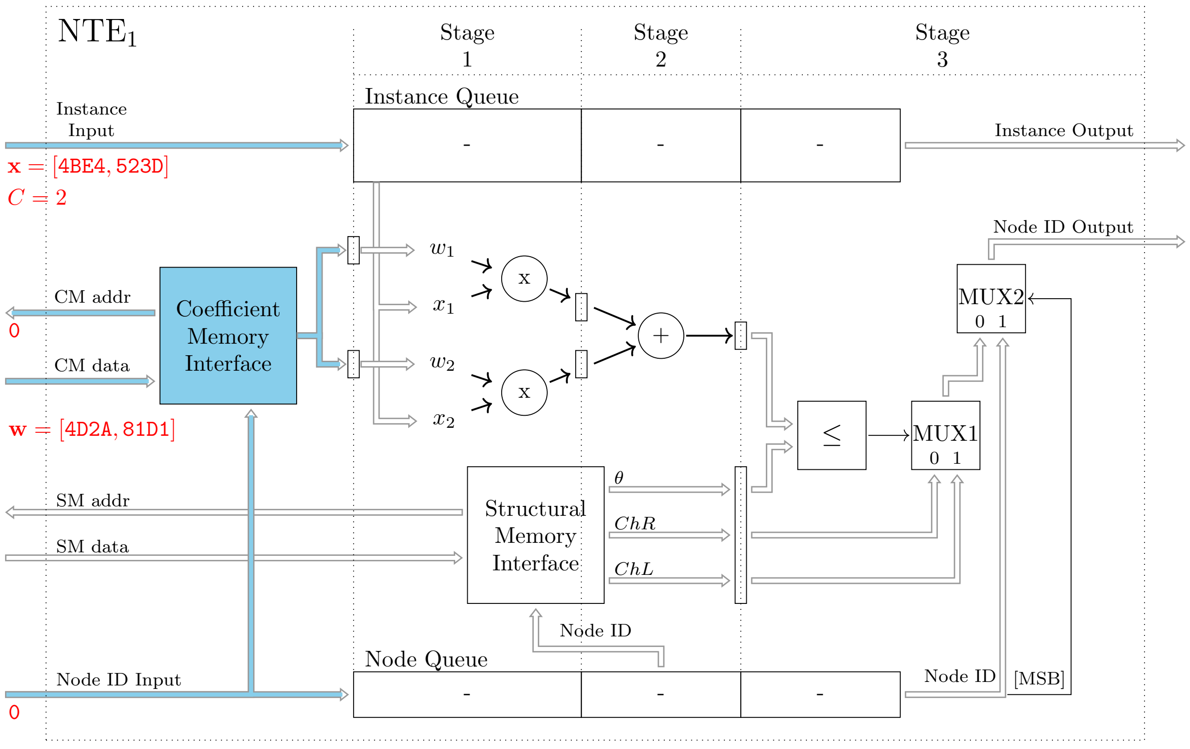


Figure 4.10: *The preparation for the first pipeline stage, where the loading of the coefficient vector for the selected node from the CM memory is performed. All the blocks and the signal paths active in this phase are highlighted in blue.*

in decimal format and then in the fixed point representation immediately below. Next, it will be shown how the Classifier module calculates the classification results of the example DT on a single instance. For this example an instance marked in the Figure 4.9 will be classified. The instance belongs to the class C_2 and has the attribute vector $\mathbf{x} = [0.5929, 0.6425]$, which encoded in Q0.15 becomes $\mathbf{x} = [\texttt{4BE4}, \texttt{523D}]$. As the Figure 4.5 shows, the instance is first input to the NTE$_1$ module's *Instance Input* port. Please notice that the information about the class to which the instance belongs is not used by the Classifier module, and will be used only once the instance is classified and the results are transmitted to the Accuracy Calculator module. The value of the *Node ID Input* on the NTE$_1$ module is fixed to 0, i.e. the node with ID 0 is always selected since the root node is the only possible choice for the first DT level.

Before the first pipeline stage, $\mathbf{w}$ needs to be loaded from memory for the selected node. The read from the CM memory is performed asynchronously and the coefficients are lead to their corresponding registers in order to be used in the first pipeline stage that performs the multiplication operation. The vector $\mathbf{x}$ is led to the Instance Queue, and the current node ID is led to the Node Queue. All blocks and signal paths active in this phase are highlighted in blue in the figure Figure 4.10.

Next, in the first pipeline stage the element-wise multiplication between vectors $\mathbf{w}$ and $\mathbf{x}$ is performed as shown in the Figure 4.11 with all active parts highlighted in blue. The current instance and the current node ID are now stored in the first elements of the Instance and Node queues respectively. The vector $\mathbf{w}$ and $\mathbf{x}$ element values are shown in the figure, as well as the multiplication results which are in Q0.30 fixed point format as it was already described. Please notice, that *NTE* performs signed additions and multiplications, hence the sign extension is needed for all operands, but this is not shown in the figures. Hence, in order to obtain the correct result for the $w_2 \cdot x_2$ multiplication, the coefficient w_2, which is negative in this case, first needs to be sign extended to Q0.30 format: $w_2 = \texttt{7FFF81D1}$, and then only lower 31 bits from the product are kept, while discarding the upper bits which arose from multiplying with the sign extension:

$$w_2 \cdot x_2 = \texttt{7FFF81D1} \cdot \texttt{0000523D} = \texttt{523CD776E0CD} \xrightarrow{Q0.30} \texttt{5776E0CD} \qquad (28)$$

Then in the pipeline Stage 2 (Figure 4.12), the addition of the element-wise products is performed. Since the Classifier module was configured to support only two instance attributes via the N_A^M attribute, the addition can be performed within single pipeline stage. If a higher value were selected for the N_A^M parameter, multiple stages would be needed in order to calculate the dot product sum. The current instance and the current node ID are now stored in the second element of the Instance and Node queues respectively. The vector element-wise products are shown in the figure, as well as the addition result, that is in the same time the final dot product result, and is encoded in Q1.30 fixed point format. Again, the negative element $w_2 \cdot x_2$ needs to be sign extended to the format of the result: $w_2 \cdot x_2 = \texttt{D776E0CD}$, after which the addition can be performed:

$$w_1 \cdot x_1 + w_2 \cdot x_2 = \texttt{16E00768} + \texttt{D776E0CD} = \texttt{EE56E835} \xrightarrow{Q1.14} \texttt{EE56} \qquad (29)$$

The dot product sum is finally converted to the format of θ, which is Q1.14 in this example, by truncating the lower bits. Additionally, the information needed for the final decision on where the traversal will continue is fetched from the SM memory and prepared for the last pipeline stage. The fetched values for θ, ChL and ChR for this example are shown in the figure.

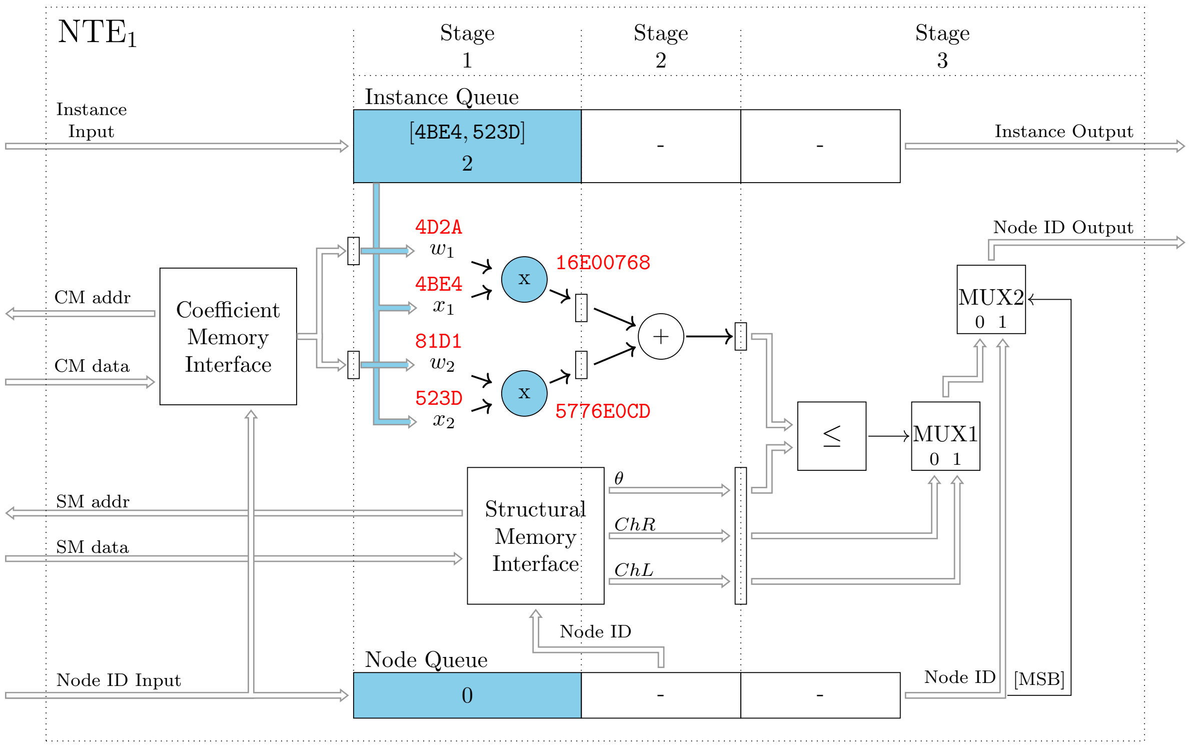


Figure 4.11: *The first pipeline stage, where the element-wise multiplication between vectors* $\mathbf{w}$ *and* $\mathbf{x}$ *is performed. All the active parts are highlighted in blue in the figure.*

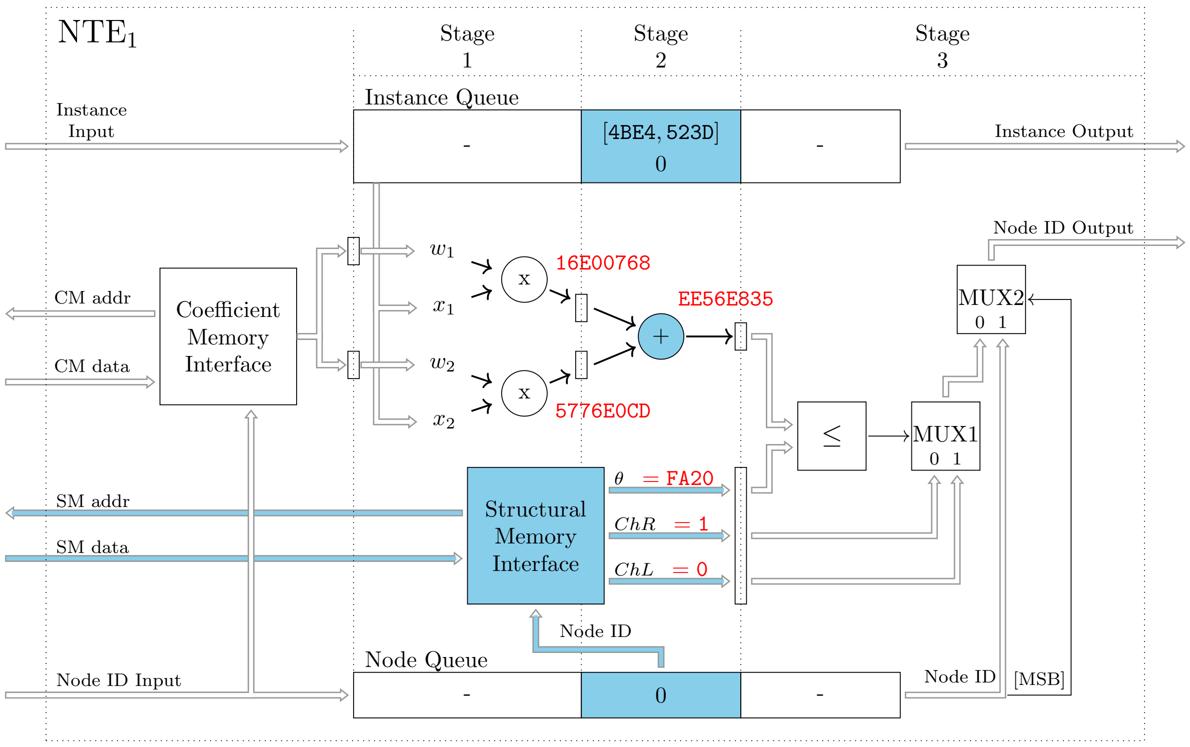


Figure 4.12: *The second pipeline stage, where the final evaluation of the node test is performed and the decision on where the traversal will continue is made. All the blocks and the signal paths active in this stage are marked in the figure.*

Finally in the pipeline Stage 3, the dot product calculation results are compared to the value of θ, to obtain the node test result, which in this case `EE56` $\leq$ `FA20` evaluated to `true` (these are both negative values in Q1.14 and the comparator block performs the signed comparison). Based on the comparison result, the MUX1 block forwards the ID of the left child $ChL = 0$ to its output, which is then passed to the port 0 of the MUX2 block. Since the current node is not a leaf (the current instance is yet to be classified), the *Node ID* MSB has a value 0, which selects the value from the MUX2 port 0 to be forwarded by the MUX2 block to its output, which is in turn lead to the *Node ID Output* port. Hence, the result of the NTE_1 operation in this example is that the $ChL = 0$ value is output via *Node ID Output* port, and the DT traversal for this instance will continue via node with ID 0 on the second DT level, which will in turn be performed by the NTE_2 module.

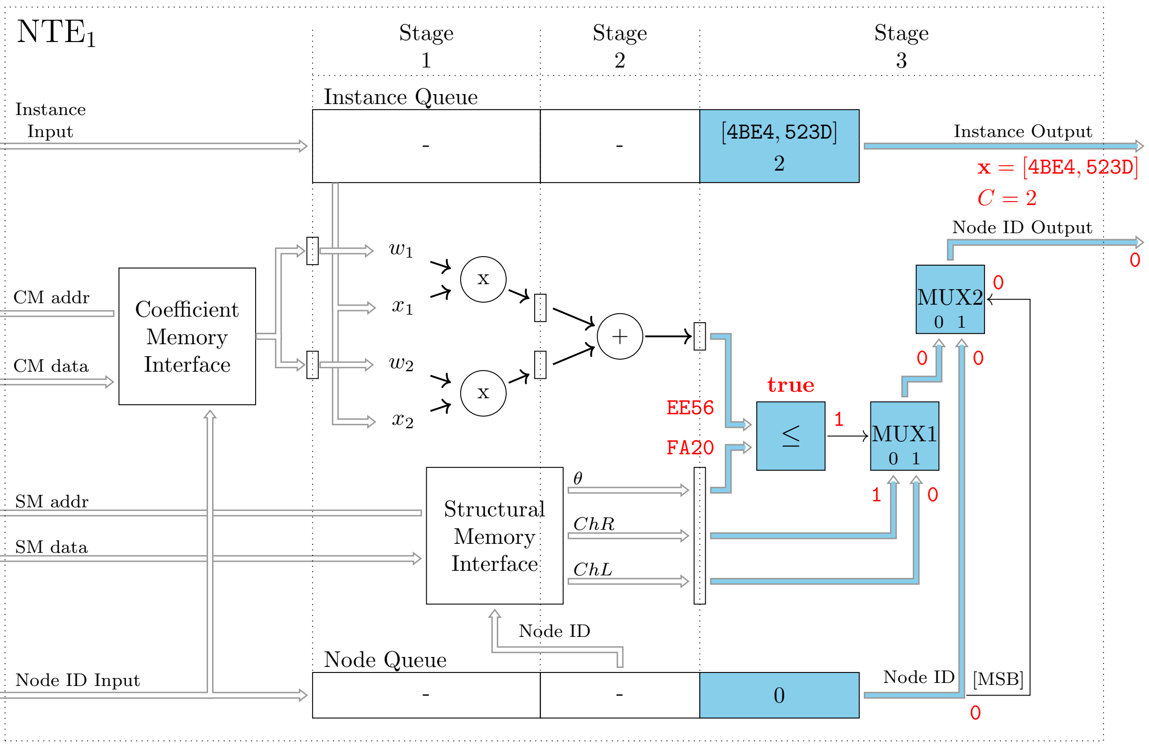


Figure 4.13: *The third pipeline stage, . All the blocks and the signal paths active in this stage are highlighted in blue.*

The outputs *Instance Output* = [`4BE4`, `523D`], $C = 2$ and *Node ID Output* = 0, as shown in the Figure 4.13, are then passed to the NTE_2 module where the traversal of the instance continues. The NTE_2 module performs in the exact same 3 stages as the NTE_1 module did, but on a different DT node. The Figure 4.14 combines the results of the computations from all 3 NTE_2 stages in one image, which in fact occur in successive cycles. This time, the value passed from the previous *NTE* (the value 0 in this example), is used to select the node for the test evaluation, among the two possible nodes on the DT level 2. As it is shown in the figure, the test evaluates to `false`, and hence the traversal is to be continued via the right child. In this case, the right child is a leaf with the ID `80`, and the instance's classification is thus determined.

Notice however, how close the dot product sum `16AA` is to the θ value `14DF`. This is due to the proximity of the instance to the hyperplane separating region `80` and regions `83` and `84` in the attribute space. If the instance were positioned exactly on the hyperplane, these two values would be identical. Anyway, the instance is passed to the next (and also the last)

NTE module, which will recognize that no further computation is needed for the instance, and simply pass the results to the Classifier output. The Figure 4.15 shows likewise the relevant computation results from all 3 stages of NTE_3 module in one image. Basically, the results of the dot product calculations are disregarded (and omitted from the figure for this reason), and the MUX2 component of the *NTE* module recognizes that it has received a leaf ID on its *Node ID Input* port (node ID's MSB value is 1), and simply outputs the same leaf ID value for the instance to the *Node ID Output* port. Since the NTE_3 is the last *NTE* module in the Classifier chain, its *Node ID Output* port is at the same time the output of the whole Classifier module.

The Classifier thus calculated that the instance [4BE4, 523D] finishes its traversal of the DT from the Figure 4.8 in the leaf with the ID 80. From the attribute space partition induced by the DT shown in the Figure 4.9, it can be seen that the classification is indeed correct.

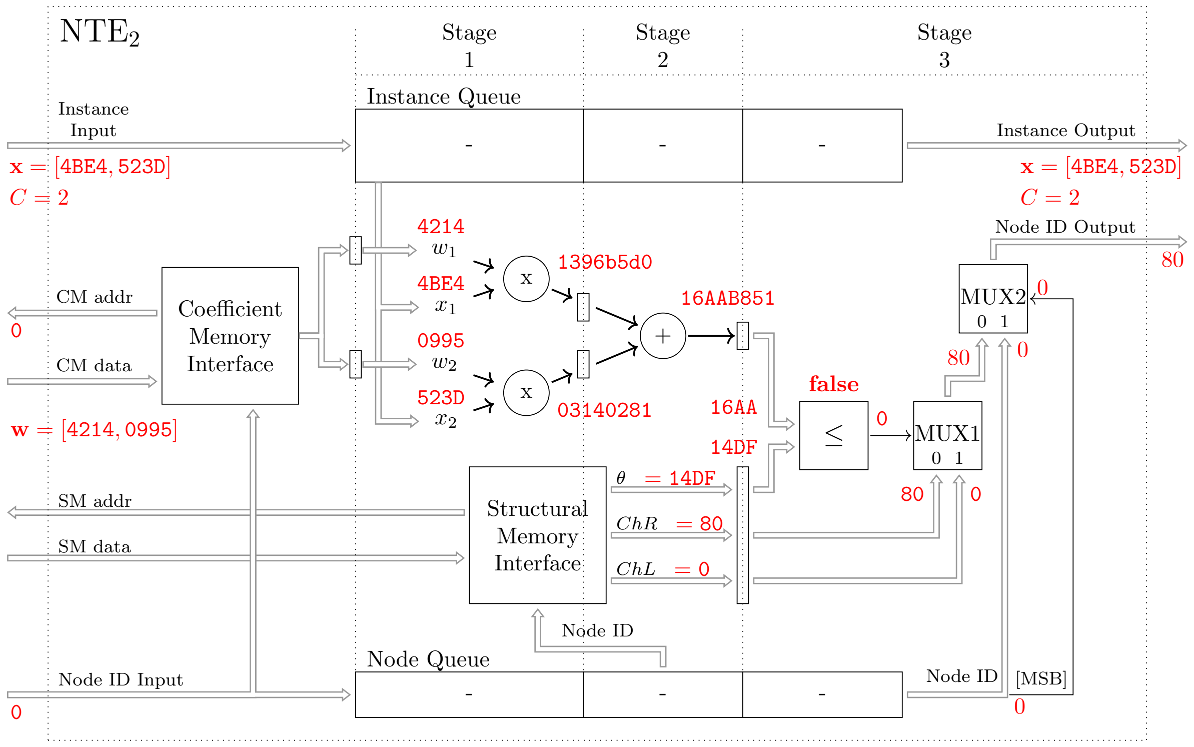


Figure 4.14: *The results of the node test evaluation on the second DT level by the NTE_2 module.*

However, the Classifier module operates on multiple instances of the dataset in parallel using the pipelining technique. The Figure 4.16 shows this process by displaying only the contents of the Instance and Node queues, which is enough to represent which instance is being processed by which stage of which *NTE*. Each pipeline stage is represented by a pair of Instance and Node queue elements which are displayed directly above one another in the figure. The Instance Queue element of the pair shows the attribute vector and the class assigned to the instance it contains, while the Node Queue element shows the current ID of the node this instance is at.

At the beginning, the queues are empty and the first instance I_0 is received from the Training Set Memory as shown in the Figure 4.16a. The node test evaluation computation is carried out in the NTE_1 module stage by stage, and in three clock cycles the I_0 instance is transferred to the NTE_2 module, as shown in the Figure 4.16b. There, its traversal is continued via the node with ID 0 on the DT level 1 (Figure 4.8). By this time, three more instances have been

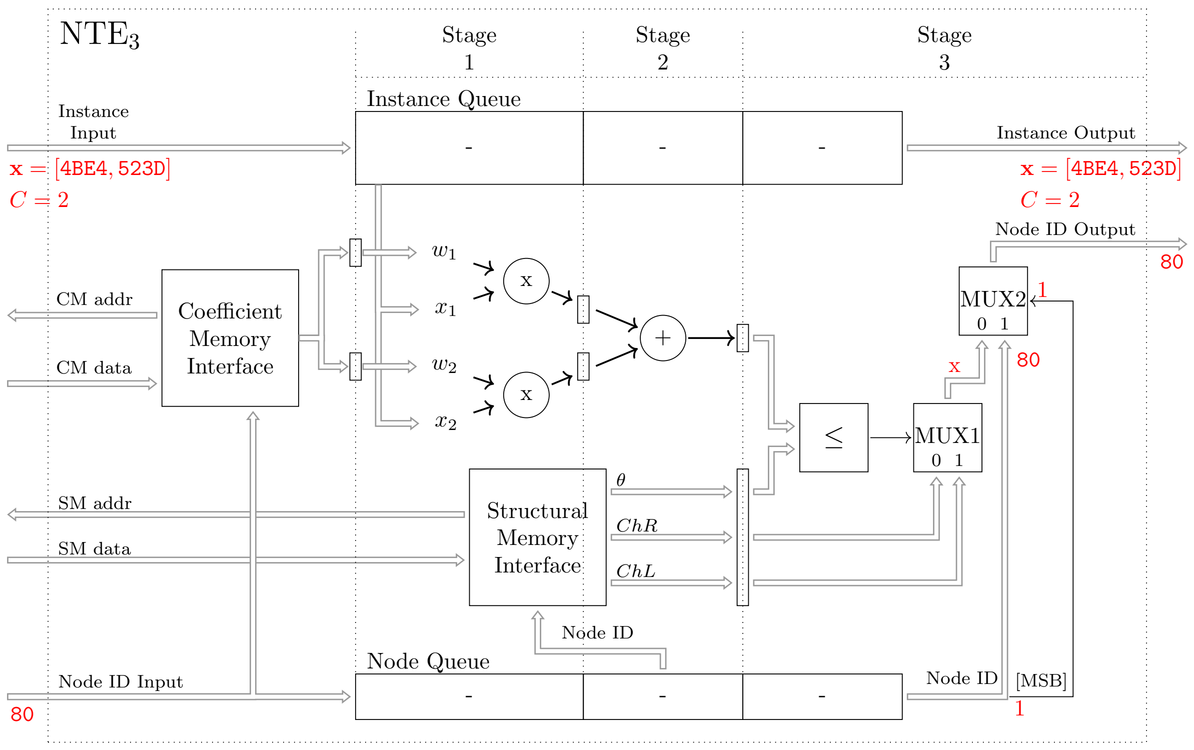


Figure 4.15: The results of the node test evaluation on the third DT level by the NTE_3 module.

loaded from the Training Set Memory, and are in the process of the node test evaluation in three stages of the NTE_1 module. Since they all need to start from the root node, their selected node IDs are all 0. Finally, the Figure 4.16c shows the moment in the classification where the first instance of the dataset I_0 has reached the end of the pipeline and is outputted to the Accuracy Calculator module, along with its classification into the leaf node with the ID 83.

4.3.2 Training Set Memory

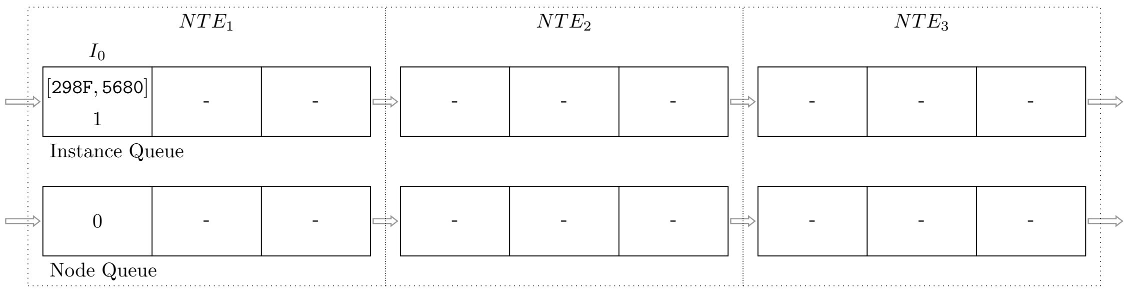
This is the memory that holds all the training set instances that should be processed by the *EFTIP* co-processor. It is a two-port memory with ports of different widths and is shown in the Figure 4.17. It is comprised of the 32-bit wide stripes, in order to be accessed by the host CPU via the 32-bit AXI interface. Each instance description, spanning multiple stripes, comprises the following fields:

- Array of instance attribute values: $x_{i,1}$ to x_{i,N_A^M}, each R_A bits wide (parameter specified by the user at design time),
- Instance class: C_i, which is R_C bits wide (parameter specified by the user at design time)

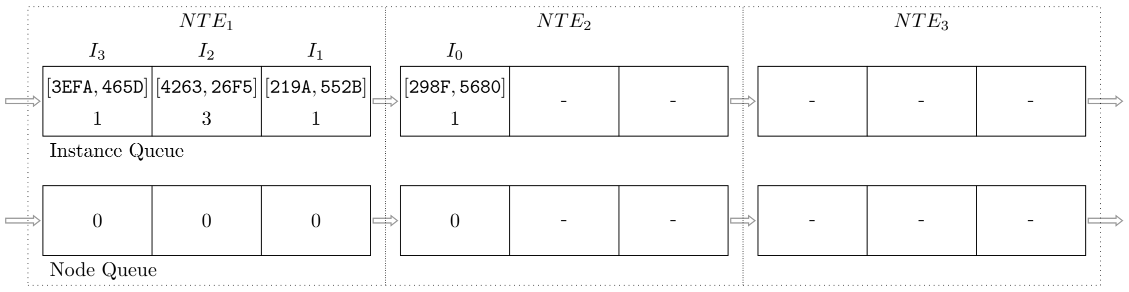
The training set memory can be accessed via two ports:

- **User Port** - Read/Write port accessed by the CPU via the AXI interface, 32-bit wide.
- **NTE Port** - Read port for the parallel read-out of the whole instance, $R_A \cdot N_A^M + R_C$ bits wide.

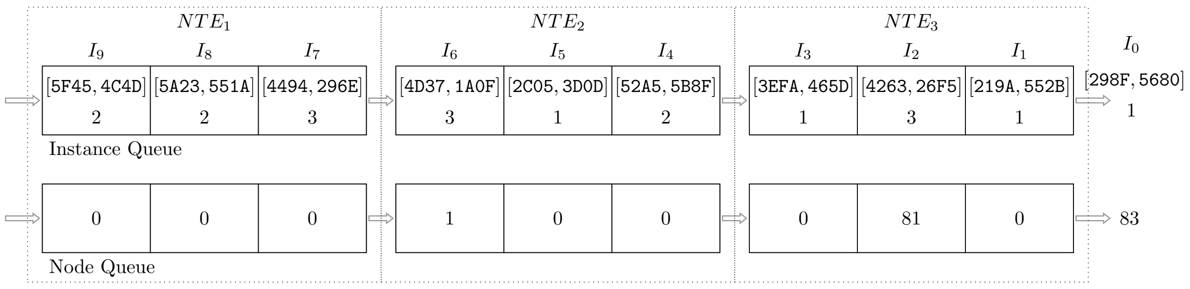
The width of the NTE Port is determined at the design phase of the *EFTIP*, and corresponds



***(a)** State after the 1st clock cycle*



***(b)** State after the 4th clock cycle*



***(c)** State after the 10th clock cycle*

***Figure 4.16:** The process of pipelined operation of the Classifier module with only the contents of the Instance and Node queues displayed, which in turn represent which instance is being processed by which stage of which* NTE*.*

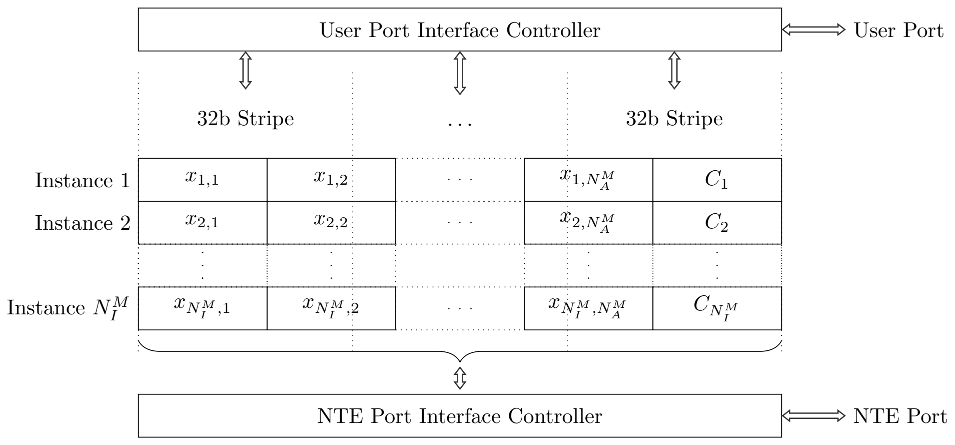


***Figure 4.17:** The Training set memory organization*

to the maximal size of the instance, i.e. the instance with the N_A^M number of attributes, that can be processed. The instance attributes are encoded using an arbitrary fixed point number format, specified by the user. However, the same number format has to be used for all instances' attribute encodings. The total maximum number of instances (N_I^M), i.e. the size of the Training Set Memory, is selected by the user at the design phase of the *EFTIP*, and determines the maximum possible training set size that can be stored inside the *EFTIP* co-processor.

4.3.3 DT Memory Array

DT Memory Array is composed of D^M sub-modules used for storing the DT description, including the structural information and the coefficient values for every node test of the DT. Each sub-module of the DT Memory Array is a three-port memory with ports of different widths (as shown in the Figure 4.18) and is comprised of 32-bit wide stripes in order to be accessed by the host CPU via the 32-bit AXI interface.

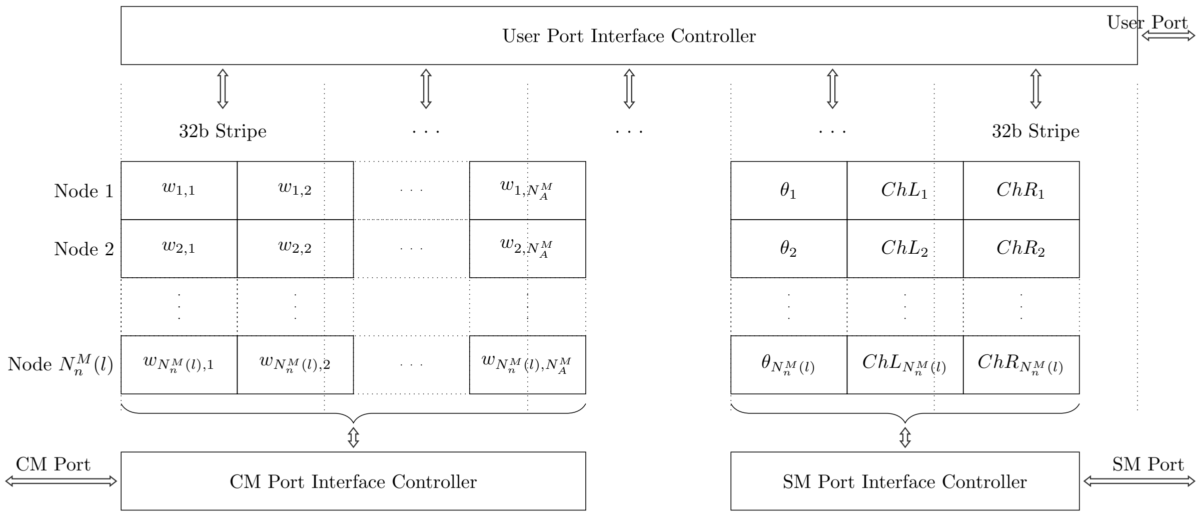


Figure 4.18: *The DT memory organization*

Each DT Memory Array sub-module contains a list of node descriptions as shown in the Figure 4.18, and has two parts. The CM part of the memory comprises the array of the node test coefficients: $w_{i,1}$ to w_{i,N_A^M}, each R_A bits wide. The SM part of the memory contains the following fields:

- The node test threshold: θ_i, which is R_A bits wide
- The ID of the left child: ChL_i, which is R_N bits wide
- The ID of the right child: ChR_i, which is R_N bits wide

An array of parameters, $N_n^M(l), l \in (1, D^M)$, that can be specified by the user at the design stage, is used to control the size of the individual DT Memory Array sub-modules. These parameters impose a constraint on the maximum number of nodes that the induced DT can have on each level. The size of each DT Memory Array sub-module is configured separately, since the first DT level can only have one node (which is the root node). At the worst case, possible number of nodes per DT level increases exponentially with the depth of the DT level. However, in practice, the induced DTs are never complete binary trees, hence the increase of the sub-modules' size with the corresponding DT level depth saturates quickly. To make the addressing of the DT Memory Array sub-modules of different size easier, every sub-module is given the address space of an identical size and it is up to the user to take care of how many DT node descriptions are actually available in each sub-module.

Since the fields ChL_i and ChR_i can either contain a leaf or a non-leaf ID, and the ID's MSB is used to discern the ID type, with their width of R_N, they can encode 2^{R_N-1} IDs. The value of the parameter R_N is calculated at the design time so that the fields ChL_i and ChR_i can encode both the the maximum number of nodes per any DT level and the maximum number of leaves the induced DT can have, i.e.:

$$R_N = 1 + \lceil ld(max(N_n^M(1), ..., N_n^M(D^M), N_l^M)) \rceil \tag{30}$$

DT Memory Array sub-module can be accessed via three ports:

- **User Port** - The read/write port, accessed by the CPU via the AXI interface, 32-bit wide.
- **CM Port** - The read port for the parallel read-out of all node test coefficients for the addressed node, $R_A \cdot N_A^M$ bit wide.
- **SM Port** - The read port for the parallel read-out of the node structural information for the addressed node, $R_A + 2 \cdot R_N$ bit wide.

4.3.4 Accuracy Calculator

This module calculates the accuracy of the DT by forming the distribution matrix as described by the Algorithm 3.3. It monitors the classifications outputted by the Classifier for each instance in the training set, and based on its class (*C*) and the leaf in which it finished the traversal (*Leaf ID*), the appropriate element of the distribution matrix is incremented. The Accuracy Calculator block is shown in the Figure 4.19.

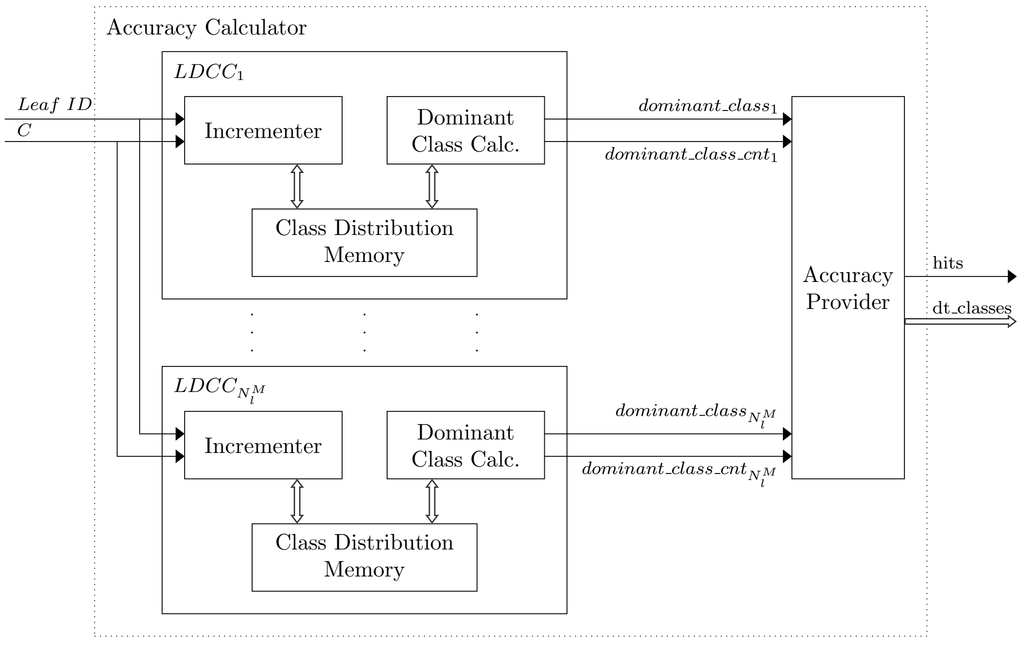


Figure 4.19: *The Accuracy Calculator block diagram*

In order to speed up the dominant class calculation (second loop of the `accuracy_calc()` function in the Algorithm 3.3), the Accuracy Calculator is implemented as an array of calculators, called Leaf Dominant Class Calculator - *LDCC*, whose each element keeps track of the distribution for the single leaf node. Hence, the dominant class calculation for a leaf (the `dominant_class` and the `dominant_class_cnt` variables from the Algorithm 3.3) and counting the total number of instances that finished the traversal in the leaf, can be performed in parallel for each leaf node. In other words, each *LDCC* is responsible for maintaining one row of the distribution matrix from the Figure 3.13. The parameter N_l^M, which can be specified by the user during the design phase of the *EFTIP* co-processor, determines the number of *LDCC* blocks available in the Accuracy Calculator module, and hence imposes a constraint on the maximum number of leaves in the induced DT. Since the width of the node ID representation, parameter R_N, also constraints the maximum number of leaves, these two parameters need to be correlated, with at least $R_N = \lceil ld(N_l^M) \rceil + 1$, with the one additional bit used to discern the node IDs from the leaf IDs. Each *LDCC* comprises:

- **Class Distribution Memory** - For keeping track of the class distribution of the corresponding leaf node.
- **Incrementer** - Updates the memory based on the Classifier output.
- **Dominant Class Calculator** - Finds and outputs the dominant class for the leaf and the number of instances of the dominant class that were classified in the leaf, using the signals $dominant\_class_i$ and $dominant\_class\_cnt_i$ respectively, where $i \in (1, N_l^M)$, as shown in the Figure 4.19.

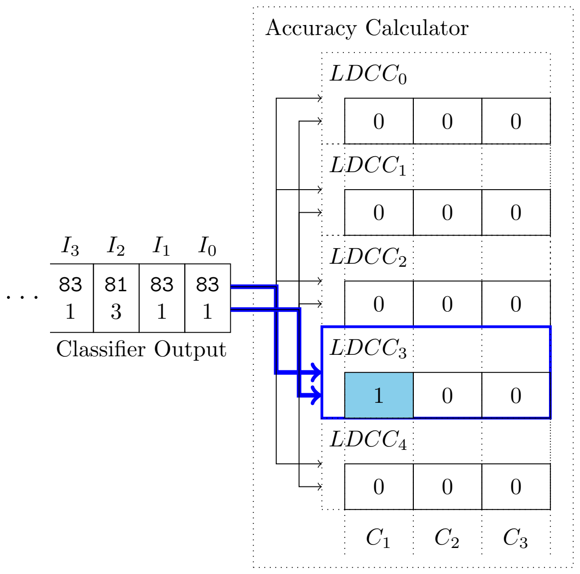
For the leaf it is responsible for, each *LDCC* keeps track of how many instances of each of the training set classes were classified in the leaf. The parameter N_C^M, also specified by the user at the *EFTIP* design time of the, determines the width of the Class Distribution Memory and hence the maximum number of classes of the training set the *EFTIP* co-processor supports. It then finds a class that has the largest number of instances in the leaf (the dominant class corresponding to the `dominant_class` variable in Algorithm 3.3), and outputs its ID via the *dominant_class* port. If the instance's class equals the dominant class of the leaf node it finished the traversal in, it is considered a hit, otherwise it is considered a miss. Hence, the value output to the *dominant_class_cnt* port represents the number of classification hits for the corresponding leaf node and corresponds to the `dominant_class_cnt` variable in Algorithm 3.3. The total number of instances classified in the leaf is output via `hits` port.

When the classification of the training set is finished, the Accuracy Provider block performs the following:

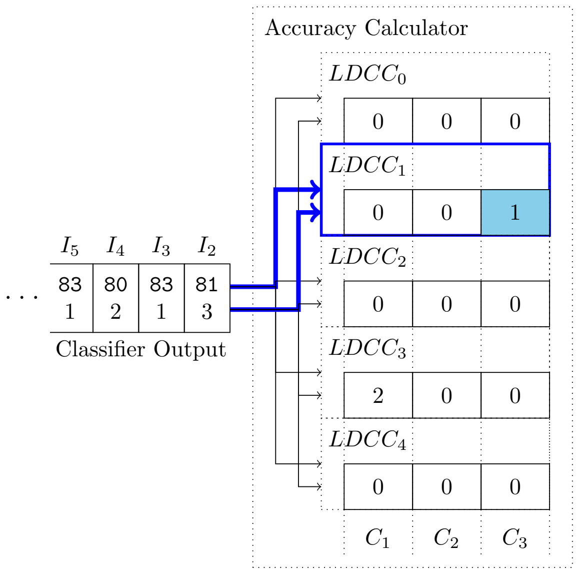
- It sums the classification hits for all leaf nodes and outputs the sum as the number of hits for the whole DT (the *hits* port), which is then stored in the Classification Performance Register of the Control Unit.
- Gathers the information about dominant classes for each of the leaves and outputs this value via *dt_classes* port for storing it in Classes Register in Control Unit.

4.3.4.1 The Accuracy Calculator Operation Example

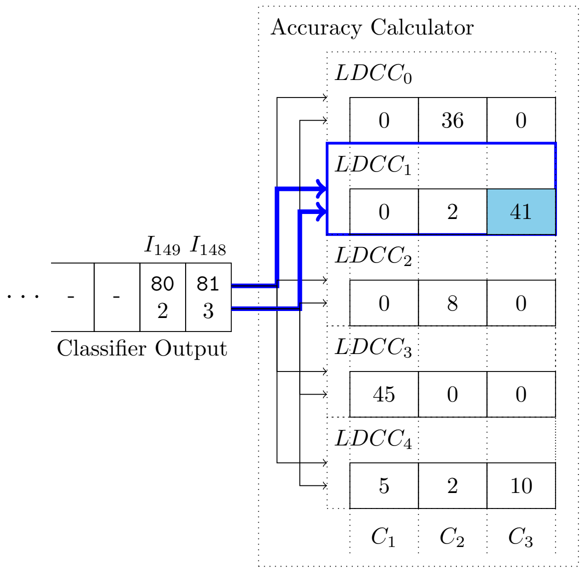
In this subsection a demonstration of the Accuracy Calculator operation is given for the `vene` dataset classified by the DT from the Figure 4.8, and is shown in the Figure 4.20. The



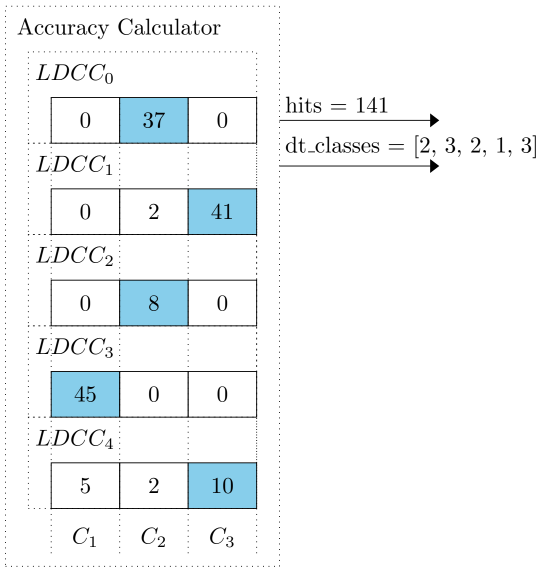
(a) First instance I_0 of the training set arrives



(b) Instances continue to arrive and the matrix gets populated



(c) Distribution matrix almost complete, with only two instances left



(d) Distribution matrix complete, final results output

Figure 4.20: *Demonstration of the Accuracy Calculator operation for the* `vene` *dataset classified by the DT from the Figure 4.8*

classification results arrive from the Classifier module in each clock cycle for a different training set instance and are comprised from the leaf ID and the instance class pairs, as shown in the Classifier Output queue in the figure. The Accuracy Calculator is shown comprising the *LDCC* array of which only five active *LDCC* modules are shown, each responsible for one of the DT leaves `80 - 84`. The remaining $N_l^M - 5$ *LDCC* modules are inactive in this example since there are only five leaves in the DT. Each *LDCC* is shown comprising the Class Distribution Memory consisting of three elements, one for each of the classes (C_1, C_2 and C_3) occurring in the `vene` training set, while the remaining $C^M - 3$ elements are inactive and not shown in the figure. Together all *LDCC* modules with their Class Distribution Memories form the distribution matrix.

Based on the ID of the leaf the instance was classified into, the appropriate *LDCC* is activated. It then uses the instance class information to increment the corresponding item in the distribution matrix row it is responsible for. In the Figure 4.20a, the first instance in the training set I_0 is shown arriving from the classifier module, prior to which the distribution matrix was empty. I_0 was classified into the leaf with the ID `83` for which the $LDCC_3$ module is responsible, and it belongs to the class C_1, represented by the first column in the distribution matrix. Hence, the $LDCC_3$ module increments the first element of its class distribution row as shown in the figure. In the Figure 4.20b, the instance I_2 of the class C_3, which was classified into the leaf `81`, activated $LDCC_3$ module to increment the item corresponding to the class C_3.

The Figure 4.20c displays the moment when the last two instances from the training set arrive, and the distribution matrix is almost complete. Finally, in the Figure 4.20d, the complete distribution matrix is shown and its items corresponding to the dominant classes are highlighted in blue. The Accuracy Provider module then gathers the information from all *LDCC* modules about the dominant classes and combines them to get the total number of `hits` and the array of dominant classes that are sent to the Control Unit.

4.3.5 Control Unit

Control Unit provides the AXI4 interface access to the configuration and the status registers, as well as to the DT Memory Array and the Training Set Memory by providing a unified memory space. Furthermore it generates an IRQ signal when the accuracy calculation is finished. The following registers are provided:

- **Operation Control** - Allows the user to start, stop and reset the *EFTIP* co-processor.
- **Training Set Configuration** - Allows the user to specify the relevant properties of the training set currently used: N_I - the number of instances and N_C the number of classes in the training set.
- **Classification Performance Register** - Informs the user when the accuracy evaluation task is done, and enables the user to read the calculated number of the classification hits.
- **Classes Registers** - Stores the dominant classes associated to each of the DT leaves, received form the Accuracy Calculator's *dt_classes* port.

The accuracy calculation process is performed automatically under the management of the Control Unit and is depicted by the diagram in the Figure 4.21. The *EFTIP* co-processor remains in the Idle state until the start signal is given via the Operation Control register. By that moment, both the Training Set Memory and the DT Memory Array should have been loaded

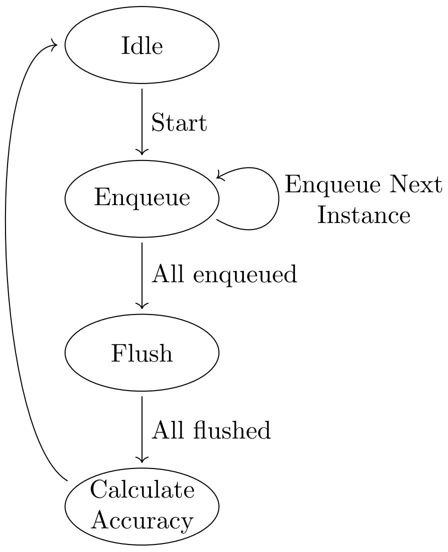


Figure 4.21: *The Control Unit FSM that manages the whole accuracy calculation process of the* EFTIP *co-processor*

with the training set instances and the desired DT description for the *EFTIP* co-processor to use. The Control Unit then moves to the Enqueue state and starts issuing a sequence of read commands to the Training Set Memory, one per clock cycle, in order to retrieve the instances of the training set and forward them to the Classifier module. This process is continued until all the instances have been read out of the Training Set Memory, when the Control Unit moves to the Flush state. In this state, the Control Unit waits for the Classifier to finish the classification of the last training set instance, after which the Accuracy Calculator is instructed to perform the dominant class calculation and the Control Unit enters the Calculate Accuracy state. After the Accuracy Calculator finished populating the Classification Performance Register and the Classes Register, the Control Unit returns to the Idle state once again, ready for the new accuracy calculation cycle.

4.4 Required Hardware Resources and Performance

The *EFTIP* co-processor is implemented as an IP core with many customization parameters discussed in the previous chapters that can be configured at the design phase. These parameters, listed in the Table 4.3, mainly impose constraints on the maximum size of the DT that can be induced, and the maximum size of the training set that can be used. The amount of hardware resources required to implement the *EFTIP* co-processor is a function of the customization parameters and is given in the Table 4.4.

Table 4.3: *The customization parameters that can be configured at the design phase of the* EFTIP *co-processor*

Name	Description	Constraint
D^M	The number of NTEs in the Classifier	The maximum depth of the induced DT
N_A^M	Determines: Training Set Memory width, DT Memory Array sub-module width, NTE adder tree size.	The maximum number of attributes training set can have
R_A	Determines: Training Set Memory width, DT Memory Array sub-module width, NTE adder tree size.	Resolution of induced DT coefficients
N_C^M	Accuracy Calculator memory depth	The maximum number of training set and induced DT classes
R_C	Number of bits class encoding	Parameter must be at least $ld(N_C^M)$
N_l^M	Number of the *LDCC* elements	The maximum number of leaves of the induced DT
R_N	Number of bits for node ID encoding	Parameter must be at least $R_N = \lceil ld(N_l^M) \rceil + 1$
N_I^M	Training Set Memory depth	The number of training set instances that can be stored in the *EFTIP* co-processor
$N_n^M(l)$	DT Memory Array sub-modules' depths	The maximum number of nodes per level of the induced DT

Table 4.4: *Required hardware resources for the* EFTIP *architecture implementation*

Resource	Module	Quantity
RAMs	Training Set Memory	$N_I^M \cdot (R_A * N_A^M + R_C)$
(number of bits)	DT Memory Array	$\sum_{i=l}^{D^M} (N_n^M(l) \cdot ((R_A + 1) * N_A^M + 2 * R_N))$
	Accuracy Calculator	$N_l^M \cdot N_C^M \cdot \lceil log_2(N_I^M) \rceil$
	NTE	$D^M \cdot N_P \cdot (R_A \cdot N_A^M + R_C) +$ $D^M \cdot N_P \cdot R_N$
Multipliers	NTE	$D^M \cdot N_A^M$
Adders	NTE	$D^M \lceil log_2(N_A^M) \rceil$
Incrementers	Accuracy Calculator	N_l^M

Second, the number of clock cycles required to determine the DT accuracy will be discussed. The Classifier has a throughput of one instance per clock cycle, hence all instances are classified in N_I cycles. However, there is an initial latency equal to the total length of the pipeline $D^M \cdot N_P$. Furthermore, the Accuracy Calculator needs extra time after the classification has finished, in order to determine the dominant class which is equal to the total number of classes in the training set N_C, plus the time to sum all dominant class hits, which is equal to the number of active leaves N_l. Finally, the time required to calculate the DT accuracy, expressed in clock cycles, for a given training set can be calculated as follows:

$$\text{accuracy evaluation time} = (N_I + D^M \cdot N_P + N_C + N_l) \text{ clock cycles,} \tag{31}$$

and is thus dependent on the training set size.

4.5 Software for the *EFTIP* Assisted DT Induction

With the *EFTIP* co-processor performing the DT accuracy evaluation task, remaining functionality of the *EFTI* algorithm (Algorithm 3.1) is implemented in software. Furthermore, the software needs to implement procedures for interfacing the *EFTIP* co-processor as well. The needed changes to the main function of the *EFTI* algorithm can be seen in the adapted pseudo-code of the `efti()` function given in the Algorithm 4.1. For the pure software implementation, the reference to the training set is passed as an argument, and can be readily accessed for the accuracy calculation task since it resides in the memory directly accessible to the CPU. However the *EFTIP* co-processor has its own memory, the Training Set Memory, for storing the training set instances that needs to be loaded before the induction process starts. The *EFTI* algorithm performs many fitness evaluations on the same dataset during the DT induction, hence the `hw_load_train_set()` function, given by the pseudo-code in the Algorithm 4.2, is used to load the training set instances to the *EFTIP* co-processor only once at the beginning of the algorithm. Once stored in the Training Set Memory, the information about the training set instances will be reused in every iteration of the algorithm.

Algorithm 4.1: *The pseudo-code of the* EFTI *algorithm using the* EFTIP *co-processor*

```
def efti(train_set, max_iter):
    hw_load_train_set(train_set, fp_format)

    dt_best = dt = initialize(train_set)
    hw_load_dt(dt.root)
    fitness_eval(dt, train_set)

    for iter in range(max_iter):
        dt_mut  = mutate(dt)
        hw_load_dt_diff(dt_mut)

        fitness_eval(dt_mut, train_set)

        dt, dt_best = select(dt, dt_mut, dt_best)

        if dt != dt_mut:
            if dt == dt_best:
                hw_load_dt(dt.root)
            else:
                hw_revert_dt_diff(dt_mut)
    hw_load_dt(dt_best.root)
    hw_populate_classes(dt_best)

    return dt_best
```

Algorithm 4.2: *The pseudo-code of the `hw_load_train_set()` function that performs the transfer of the training set to the* EFTIP *co-processor*

```
def hw_load_train_set(train_set, fp_format):

    for i, instance in enumerate(train_set):
         pack_row = pack_instance(instance, fp_format)

         for e, elem in enumerate(pack_row):
              hw_write(eftip_train_mem_addr(i,e), elem)
```

After the initial DT individual is created, it needs to be transferred to the *EFTIP* co-processor in order for its accuracy to be determined, which is performed by the `hw_load_dt()` function given by the pseudo-code in the Algorithm 4.3. This is the recursive function that loads both coefficient and structural information about the DT node and all of its descendants to the corresponding CM and SM memory parts of the DT Memory Array of the *EFTIP* co-processor. First, the `pack_dt_node()` function, whose implementation was omitted for brevity, packs the node's coefficients and structural information, in a list of 32-bit values in a way that the organizations of the SM and CM DT memory parts dictate (Figure 4.18). As it can be seen from the pseudo-code, the packing depends on the fixed point format (Qx.y) used for the coefficients (the argument `fp_format`) and the width of the node and leaf ID representations R_N (the argument `Rn`). The packed information is then written to the *EFTIP* co-processor memories one 32-bit word at a time, at desired locations whose addresses are calculated by helper functions `eftip_dt_cm_addr()` and `eftip_dt_sm_addr()` whose implementations are again omitted.

Algorithm 4.3: *The pseudo-code of the `hw_load_dt()` function that performs the transfer of the DT individual coefficients and structural data to the* EFTIP *co-processor*

```
def hw_load_dt(node):
    if not node.is_leaf:
        cm_pack, sm_pack = pack_dt_node(node, fp_format, Rn)

        for e, elem in enumerate(cm_pack):
            hw_write(eftip_dt_cm_addr(node.level, node.id, e), elem)

        for e, elem in enumerate(sm_pack):
            hw_write(eftip_dt_sm_addr(node.level, node.id, e), elem)

        hw_load_dt(node.left)
        hw_load_dt(node.right)
```

With both training set and the DT loaded to the co-processor, the accuracy calculation function needs only to send the start signal (the `hw_start()` helper function) and wait for the results (the `hw_get_hits()` helper function), as it can be seen from the Algorithm 4.4.

Algorithm 4.4: *The pseudo-code of the* `accuracy_calc()` *function adapted to use the* EFTIP *co-processor*

```
def accuracy_calc(train_set):
    hw_start()

    hits = 0
    while hits == 0:
        hits = hw_get_hits()

    return hits / len(train_set)
```

In the end of the *EFTI* algorithm the induction procedure settles for the best DT individual (variable `dt_best`). However, the information about the dominant classes of the DT leaves is not retrieved from the *EFTIP* co-processor during each iteration to save time on data transfer since it is not critical for the *EFTI* algorithm operation. Nevertheless, the induced DT returned by the algorithm needs to have classes assigned to all of its leaves, which is performed by the `hw_populate_classes()` function given by the pseudo-code in the Algorithm 4.5. This function invokes one last accuracy calculation on the best DT individual, which will in turn populate the Classes Register of the Control Unit with the dominant classes for all the DT leaves. This information can then be read by the software, at the address calculated by the `eftip_cu_cls_addr()` helper function in the pseudo-code, and assigned to the DT software data structure.

Algorithm 4.5: *The pseudo-code of the* `hw_populate_classes()` *function adapted to use the* EFTIP *co-processor*

```
def hw_populate_classes(dt):
    hw_start()

    hits = 0
    while hits == 0:
        hits = hw_get_hits()

    for leaf in dt.leaves():
        leaf.cls = hw_read(eftip_cu_cls_addr(leaf.id))
```

4.6 Experiments

In this section, the results of the experiments designed to estimate the DT induction speedup of the HW/SW implementation of the *EFTI* algorithm using the *EFTIP* co-processor over its pure software implementation are discussed.

4.6.1 Required Hardware Resources for the *EFTIP* Co-Processor Used in Experiments

The customization parameters of the *EFTIP* co-processor, whose descriptions are given in the Table 4.3, have been set for the experiments to support all training sets from the Table 2.1, and their values are listed in the Table 4.5.

Table 4.5: *The values of customization parameters of the* EFTIP *co-processor instance used in the DT induction speedup experiments*

Parameter	Value
DT Max. depth (D^M)	13
Max. attributes num. (N_A^M)	64
Attribute encoding resolution (R_A)	16
Class encoding resolution (R_C)	8
Class encoding resolution (R_N)	9
Max. training set classes (C^M)	64
Max. number of leaves (N_l^M)	256
Max. number of training set instances (N_I^M)	46000
Max. number of nodes per level ($N_n^M(l)$)	[1, 2, 4, 8, 16, 16, 16, 32, 32, 32, 64, 64, 64]

The *EFTIP* co-processor has been modeled in the VHDL hardware description language and implemented using the Xilinx Vivado Design Suite 2014.4 software for logic synthesis and implementation, with the default synthesis and P&R options. From the implementation report files, the device utilization data has been analyzed and the information about the number of used slices, BRAMs and DSP blocks has been extracted, and is presented in the Table 4.6. The maximum operating frequency of 133 MHz of the system clock frequency for the implemented *EFTIP* co-processor was attained.

Table 4.6: *FPGA resources required to implement the* EFTIP *co-processor for the DT induction with selected UCI datasets*

FPGA Device	Slices	BRAMs	DSPs
XC7Z100	24418 (32%)	755 (100%)	832 (43%)
XC7K410	21156 (32%)	760 (96%)	832 (58%)
XC7VX690	20847 (18%)	760 (43%)	832 (22%)

Given in the brackets, along with each resource utilization number, is the percentage of used resources from the total resources available on the corresponding FPGA devices. Table 4.6 shows that implemented *EFTIP* co-processor fits into the mid-level Kintex7 and Virtex7 Xilinx FPGA devices (XC7K410 and XC7VX690) and high-end XC7Z100 Xilinx FPGA device of the Zynq series. The scalability of the HW/SW solution can be observed from the point of several customization parameters of the *EFTIP* co-processor given in the Table 4.3. The Table 4.4 shows how some of these customization parameters influence the utilization of the hardware resources.

The number of instances the *EFTIP* co-processor can store in its Training Set Memory is limited by the parameter N_I^M, selected at the design phase of the the *EFTIP*. In case that the datasets which cannot fit into the Training Set Memory need to be used, either a double buffering approach could be used or *EFTIP* could be used in the streaming mode. In the streaming mode, the data would be continuously streamed from the host CPU memory using the DMA transfer. In this case, there would be no Training Set Memory, as the instances would be supplied to the Classifier from the outside via the DMA. In the double buffering approach, the Training Set Memory would be used as a ring buffer. While the *EFTIP* is using the NTE port to read the instance descriptions to the Classifier, the User port would be used to load new instances to the Training Set Memory. The DMA transfer from the main memory would be used here as well. *EFTIP* reads instances from the data set in predictable, sequential order, so it is easy to setup the DMA transfer and execute it without the intervention of the software during the transfer. This means that the full bandwidth of the main memory can be used for the data without any overhead.

If the *EFTIP* co-processor were to support the datasets with larger number of attributes, which results in wider training set instance encodings, the training set transfer time could impact the HW/SW implementation performance. In this case, again, the double buffering or the *EFTIP* in streaming mode could be used. The throughput of the *EFTIP* co-processor, i.e. the widest possible training set instance encoding that could be used without degrading the performance, would then be limited only by the bandwidth of the main memory, since there is no overhead to the training set data streaming. If the bandwidth of one main memory module is not enough, the *EFTIP* could use several memory modules simultaneously to read the data out in parallel. The internal memory widths would also increase, but this would pose no significant problem either, because the internal FPGA memory primitives can be easily configured to have arbitrary data widths. Next, the number of attributes affects the size of the adder tree of the NTE module. However, by increasing the size and the depth of the adder tree, only the pipeline depth is increased, resulting only in the increase in the initial latency of the *EFTIP* co-processor, without degrading the *EFTIP* throughput.

If the attribute encodings (R_A) were to be enlarged, other than increasing the encoding width of the training set instance, which was discussed above, the *EFTIP* co-processor multipliers and adders would need to support wider operands. This would not pose a significant constraint for implementing both multipliers and adders, since the arbitrary width multipliers and adders can be built using a number of same blocks of smaller width connected in a pipeline. Hence, the increase in the data widths would not affect the HW/SW implementation performance, because only the pipeline depths would be increased, which would in turn increase the initial latency without affecting the throughput of the system. However, as far as the author is aware, the attribute encodings with more than 32 bits are rarely used in hardware acceleration of the machine learning algorithms, as discussed in *[56][81][82]*.

Finally, if the *EFTIP* co-processor were to support the datasets with the larger number of classes *CM* and the larger number of the DT leaves N_l^M, the equation (31) shows that the *EFTIP* latency would only linearly increase as a function of these two parameters.

4.6.2 Estimation of Induction Speedup

Three implementations of the *EFTI* algorithm have been developed for the experiments, all of them written in the C language:

- **SW-PC** - Pure software implementation for the PC discussed in the Section 3.5.
- **SW-ARM** - Pure software implementation for the ARM Cortex-A9 processor.
- **HW/SW** - HW/SW co-design solution, where the *EFTIP* co-processor implemented in the FPGA was used for the time critical fitness evaluation task. The remaining functionality of the *EFTI* algorithm (shown in the Algorithm **??**) was left in software, and implemented for the ARM Cortex-A9 processor.

For the SW-ARM and the HW/SW implementations, the ARM Cortex-A9 667 MHz (Xilinx XC7Z100 Zynq-7000) platform has been used. The software was built using the Sourcery CodeBench Lite ARM EABI 4.9.1 compiler (from within the Xilinx SDK 2015.2) and the *EFTIP* co-processor was built using the Xilinx Vivado Design Suite 2015.2. The experiments were structured following the description given in the Section 2.8, and all *EFTI* algorithm implementation used in the experiments were setup using the "High accuracy" configuration for the Table 3.10 and given 500k iterations for the induction. The DT inference times were measured by different means for two target platforms:

- For the PC platform, the <time.h> C library was used and timing was output to the console,
- For the ARM and DSP platforms, hardware timer was used and the timing was output via the UART.

Table 4.7: *The DT induction times for various* EFTI *implementations and average speedups of HW/SW implementation over pure software implementations*

Dataset	HW/SW [s]	SW-ARM	Speedup SW-ARM	SW-PC	Speedup SW-PC
adult	17.97 ± 00.31	452.46 ± 02.78	25.18	221.53 ± 05.75	12.33
ausc	0.59 ± 00.01	7.23 ± 00.06	12.19	4.37 ± 00.21	7.36
bank	18.43 ± 00.04	666.21 ± 27.81	36.14	311.70 ± 03.34	16.91
bc	0.55 ± 00.02	6.10 ± 00.48	11.10	4.25 ± 00.11	7.72
bch	13.81 ± 02.19	174.47 ± 16.92	12.64	115.62 ± 01.24	8.37
bcw	0.50 ± 00.00	4.41 ± 00.00	8.88	3.35 ± 00.05	6.74
ca	0.50 ± 00.00	4.94 ± 00.00	9.85	4.45 ± 00.05	8.87
car	0.90 ± 00.00	7.60 ± 00.00	8.40	14.37 ± 00.40	15.89
cmc	1.13 ± 00.01	17.25 ± 00.29	15.29	11.93 ± 00.29	10.57
ctg	3.25 ± 00.05	41.77 ± 00.97	12.86	27.36 ± 00.64	8.43
cvf	9.23 ± 00.14	206.24 ± 05.29	22.33	110.56 ± 02.76	11.97
eb	48.51 ± 00.61	1265.06 ± 16.01	26.08	737.24 ± 09.88	15.20
eye	8.25 ± 00.09	213.12 ± 01.61	25.83	96.22 ± 05.96	11.66

Continued on next page

Table 4.7 – continued from previous page

Dataset	HW/SW [s]	SW-ARM	Speedup SW-ARM	SW-PC	Speedup SW-PC
ger	0.89 ± 00.01	13.90 ± 00.17	15.60	8.16 ± 00.68	9.16
gls	0.60 ± 00.00	3.38 ± 00.07	5.64	2.24 ± 00.07	3.74
hep	0.38 ± 00.00	1.67 ± 00.03	4.43	1.29 ± 00.07	3.43
hrtc	0.63 ± 00.01	4.04 ± 00.13	6.44	2.69 ± 00.08	4.29
hrts	0.39 ± 00.00	2.75 ± 00.03	6.99	1.82 ± 00.12	4.64
ion	0.53 ± 00.05	5.16 ± 00.08	9.73	3.19 ± 00.16	6.02
irs	0.29 ± 00.01	1.38 ± 00.05	4.80	0.98 ± 00.05	3.41
jvow	10.21 ± 00.12	218.21 ± 03.38	21.37	119.40 ± 02.90	11.69
krkopt	26.80 ± 00.41	702.25 ± 16.34	26.20	349.76 ± 11.05	13.05
letter	32.59 ± 01.83	590.89 ± 30.83	18.13	355.80 ± 07.50	10.92
liv	0.42 ± 00.00	3.34 ± 00.06	7.96	2.26 ± 00.08	5.38
lym	0.53 ± 00.01	2.14 ± 00.05	4.03	1.77 ± 00.05	3.33
magic	10.34 ± 00.21	254.57 ± 02.83	24.61	117.77 ± 02.44	11.39
msh	6.26 ± 00.06	142.52 ± 01.33	22.75	68.49 ± 04.10	10.93
nurse	9.66 ± 00.07	242.39 ± 02.59	25.09	112.53 ± 02.25	11.65
page	4.00 ± 00.03	76.63 ± 02.72	19.17	36.59 ± 01.61	9.15
pen	11.96 ± 00.16	240.75 ± 04.04	20.13	140.94 ± 02.01	11.79
pid	0.62 ± 00.01	7.36 ± 00.13	11.94	4.48 ± 00.31	7.27
psd	0.90 ± 00.02	14.37 ± 00.35	16.02	8.55 ± 00.18	9.53
sb	1.39 ± 00.00	32.88 ± 00.02	23.70	16.10 ± 00.31	11.61
seg	2.68 ± 00.03	39.60 ± 00.75	14.75	29.58 ± 00.62	11.02
shuttle	40.73 ± 00.48	979.32 ± 33.66	24.04	841.89 ± 22.17	20.67
sick	1.93 ± 00.02	72.62 ± 00.24	37.60	31.72 ± 01.23	16.42
son	0.72 ± 00.01	3.47 ± 00.06	4.79	2.74 ± 00.16	3.79
spect	0.40 ± 00.01	2.47 ± 00.10	6.09	1.85 ± 00.10	4.57
spf	2.04 ± 00.46	29.80 ± 03.22	14.64	25.95 ± 00.45	12.75
thy	3.30 ± 00.02	55.11 ± 01.15	16.68	28.16 ± 00.91	8.52
ttt	0.73 ± 00.00	9.00 ± 00.09	12.37	6.39 ± 00.32	8.80
veh	1.10 ± 00.01	12.45 ± 00.18	11.34	9.87 ± 00.18	8.99
vene	0.37 ± 00.00	2.99 ± 00.04	8.13	2.01 ± 00.05	5.45
vote	0.48 ± 00.00	4.64 ± 00.06	9.68	2.97 ± 00.10	6.20
vow	1.85 ± 00.02	20.95 ± 00.44	11.30	12.32 ± 00.27	6.65
w21	4.11 ± 00.04	82.29 ± 00.56	20.00	47.53 ± 01.34	11.55
w40	5.39 ± 00.08	100.65 ± 00.59	18.68	54.74 ± 01.06	10.16
wfr	5.48 ± 00.05	99.26 ± 01.41	18.13	62.95 ± 01.00	11.50
wilt	2.24 ± 00.00	34.22 ± 00.03	15.25	20.69 ± 00.44	9.22

Continued on next page

Table 4.7 – continued from previous page

Dataset	HW/SW [s]	SW-ARM	Speedup SW-ARM	SW-PC	Speedup SW-PC
wine	4.11 ± 00.05	80.08 ± 01.84	19.47	46.12 ± 01.26	11.21
zoo	0.53 ± 00.02	1.56 ± 00.04	2.96	1.31 ± 00.03	2.49
Avg.			15.44 ± 2.28		9.30 ± 1.11

All datasets from the Table 2.1 were compiled together with the source code and were readily available in the memory. Therefore, the availability of the training set in the main memory was the common starting point for all three implementations, thus there was no training set loading overhead on the DT induction timings. However, in the HW/SW co-design implementation, the datasets need to be packed in the format expected by the Training Set Memory organization (shown in the Figure 4.17) and loaded to the *EFTIP* co-processor via the AXI bus (performed by the *hw_load_training_set()* function). To make a fair comparison with the pure software implementations, time needed to complete these two operations was also included in the total execution time of the HW/SW implementation.

The results of the experiments are presented in the Table 4.7. For each implementation and dataset, the average induction times of the five 5-fold cross-validation runs are given together with their 95% confidence intervals. The last row of the table provides the average speedups of the HW/SW implementation over the SW-ARM and SW-PC, together with the 95% confidence intervals.

The Table 4.7 indicates that the average speedup of the HW/SW implementation is 15.4 times over the SW-ARM and 9.3 times over the SW-PC implementation. The speedup varies with the datasets used for the induction, which is expected since the *EFTI* algorithm computational complexity is dependent on the dataset characteristics as the equation (26) suggests. The computational complexity increases as N_I, N_A, N_l and N_C increase. The number of leaves in DT, N_l, is dependent on the training set instance attribute values, but can be expected to increase also with N_I, N_A and N_C. By observing the speedups of the HW/SW implementation over the pure software implementations shown in the Figure 4.22, for each dataset and the datasets' characteristics given in the Table **??**, it can be seen that indeed, more speedup is gained for datasets with larger N_I, N_A and N_C values.

Datasets `adult`, `bank`, `eb`, `eye`, `krkopt`, `letter`, `magic` and `shuttle` are the largest of the datasets in terms of N_I, and thus have some of the largest speedup gains. Some other datasets have somewhat less instances, but have a significant number of attributes, like `msh`, `cvf`, `pen`, `thy`, `w21`, `w40` and `wfr`, so that high speedups were achieved there too. A high speedup was achieved also for the following datasets: `jvow`, `nurse`, `page` and `wine`, which have smaller number of instances and smaller number of attributes than the above two groups, but tended to induce deeper DTs. The deeper the DT is the more NTEs participate actively in the accuracy calculation. Since the NTEs operate in parallel, the more of them are active the more speedup is gained. However, the partial reclassification employed by the pure software implementations, for which no analog has been implemented in the *EFTIP* co-processor, influences the speedup significantly, but in an unpredictable way. Hence, the inference times for some datasets with rather small number of instances, attributes or induced DT depths, can still have substantial speedups because the structure of the induced DTs may be such that a significant number of instances need to be reclassified each iteration after the mutation is applied.

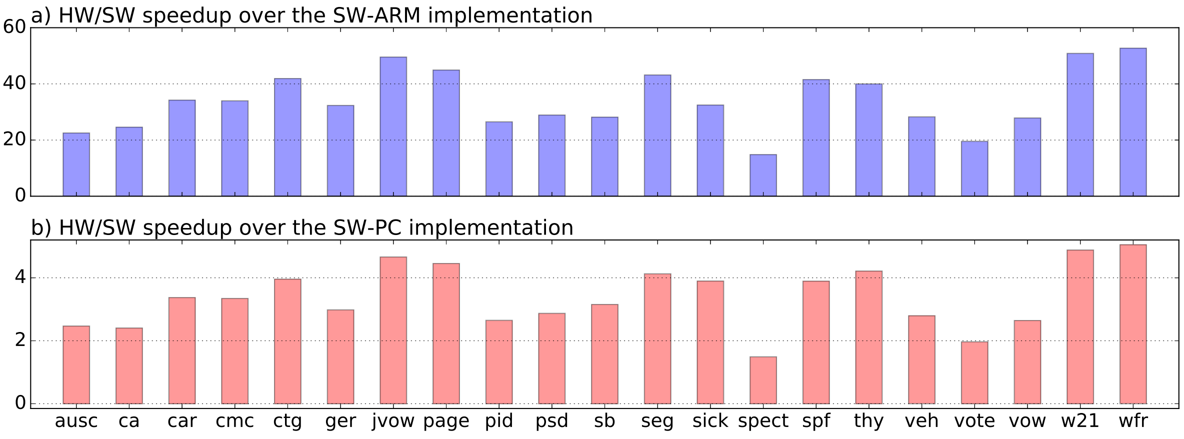


Figure 4.22: *The speedup of the HW/SW implementation over a) the SW-ARM implementation, b) the SW-PC implementation*

The Figure 4.22 and the Table 4.7 suggest that the HW/SW implementation using the *EFTIP* co-processor offers a substantial speedup in comparison to the pure software implementations, for the ARM and PC. This is mainly because both processors that were used in the experiments have a limited number of on-chip functional units that can be used for multiplication and addition operations, as well as the limited number of internal registers to store the node test coefficient values and instance attributes. This means that the loop from the equation (1) can only be partially unrolled, when targeting these processors, which would be the case for any processor type. On the other hand, the *EFTIP* co-processor can be configured to use as many multiplier/adder units as needed, and as many internal memory resources for storing coefficient and attribute values which can be accessed in parallel. Because of this, in case of *EFTIP*, the loop from the equation (1) can be fully unrolled, therefore gaining the maximum available performance. Furthermore, the *EFTIP* implementation used in the experiments, operates at much lower frequency (133MHz) than ARM (667MHz) and PC (3.5GHz) platforms. If the *EFTIP* co-processor were implemented in the ASIC technology, the operating frequency would be increased by an order of magnitude, and the DT induction speedups would increase accordingly.

5 *EEFTI* algorithm

In this section, the *EEFTI* algorithm for the induction of the DT ensembles which uses Bagging on top of the *EFTI* algorithm is proposed. The ability of the *EFTI* algorithm to operate on a single individual and induce small DTs is even more important for the ensembles, since all the operations, be it induction or classification of new instances, are performed on all the DT members of the ensemble at once. The following topics will be covered in this section:

- Section 5.1 - Description of the Bagging algorithm
- Section 4.2 - Description of the *EEFTI* algorithm
- Section 5.3 - Experiments showing the superior performance of the ensembles induced by the *EEFTI* algorithm over single classifiers in terms of the classification accuracy

5.1 Bagging Algorithm

The choice of the Bagging algorithm was made mainly because it generates one subset of the training set for each ensemble member, hence completely decoupling the induction of the individual members from each other, which in turn makes the algorithm suitable for the parallelization and hardware acceleration. Furthermore, the Bagging algorithm was reported to reduce the accuracy variance and help avoid overfitting. Two common ways of forming the subsets are:

- **random sampling without replacement** - forms disjoint subsets of size $N_{IS} = \frac{N_I}{n_e}$, and
- **random sampling with replacement** - forms overlapping subsets of size $N_{IS} \leq N_I$,

where N_{IS} is the size of the subsets, N_I the size of the whole training set and n_e the number of subsets, i.e. the number of the ensemble members. The most important feature of the sampling procedure is the diversity of the ensemble members it helps induce. This is especially important for the deterministic induction algorithms, since given the same training subset they would induce identical DT individual each time. In case of stochastic algorithms on the other hand, this is less of a problem. Hence, the *EEFTI* algorithm can be used even when $N_{IS} = N_I$.

5.2 *EEFTI* Description

The Algorithm 3.1 shows the *EEFTI* algorithm pseudo-code. *EEFTI* first partitions the training set in the subsets using the `divide_train_set()` function that implements one of the techniques discussed in the Subsection 5.1. Next, for each member of the ensemble an *EFTI* tasks is created and assigned its corresponding training subset (`train_par[i]`). In addition, the reference to the result object `r` is passed to the *EFTI* task, to which it can assign the resulting DT and any additional information about the induction, like inference time, etc. All result objects are gathered in the `res` array and are returned to the user when the induction is finished. Handles to the created tasks are gathered in the `tasks` array, which is used by the `all_finished()` helper function, which in turn checks the statuses of the running *EFTI* tasks and returns `true` when all of them have finished the induction and exited. Once all the individual tasks have finished and thus populated their corresponding result objects, the *EEFTI* algorithm exits by returning the `res` array to the user.

Algorithm 5.1: *The main function of the* EEFTI *algorithm*

```
def eefti(train_set, ensemble_size):
    train_par = divide_train_set(train_set, ensemble_size)

    res = []
    tasks = []
    for i in range(ensemble_size):
        r = {}
        t = create_task(efti, train_par[i], r)
        res.append(r)
        tasks.append(t)

    while(not all_finished(tasks)):
        pass

    return res
```

5.3 Advantages of the DT ensembles

As it was already said, the ensemble classifier systems were shown to provide improvement to the classification performance over a single classifier *[61]*. In order to test whether *EEFTI* algorithm is capable of inducing an ensemble that has superior accuracy than the individual classifier induced by the *EFTI* algorithm, an experiment has been conducted whose results are shown in this subsection. The ensembles of sizes 3, 5, 9, 17 and 33 were induced on all datasets from the Table 2.1 using five 5-fold cross-validation techinique together with the Tukey multiple comparisons test as described in the Section 2.8. The induced ensembles' accuracies were measured by performing the classification of the test set using the majority voting technique. In the Table 5.1 the average accuracies of the single classifier and the ensembles of five different sizes used in this experiment are given for each dataset together with their 95% confidence intervals. The accuracy rankings of the induced classifiers are given in the Table 5.2 for each dataset, together with the average rank for each classifier used.

Table 5.1: *The accuracies of the ensembles with various numbers of elements*

Dataset	1	3	5	9	17	33
adult	83.01 ± 0.12	83.26 ± 0.12	83.27 ± 0.05	83.30 ± 0.05	83.35 ± 0.05	83.33 ± 0.03
ausc	88.99 ± 0.25	88.82 ± 0.24	88.88 ± 0.21	89.19 ± 0.18	89.15 ± 0.19	89.27 ± 0.14
bank	88.57 ± 0.12	88.36 ± 0.06	88.32 ± 0.02	88.30 ± 0.00	88.30 ± 0.00	88.30 ± 0.00
bc	93.25 ± 0.41	93.25 ± 0.40	93.45 ± 0.44	93.75 ± 0.43	94.37 ± 0.45	95.20 ± 0.47
bch	22.73 ± 0.11	24.92 ± 0.20	26.63 ± 0.23	27.89 ± 0.23	28.74 ± 0.22	29.15 ± 0.20
bcw	97.77 ± 0.09	97.77 ± 0.08	97.83 ± 0.09	97.87 ± 0.08	97.91 ± 0.07	97.87 ± 0.07
ca	88.85 ± 0.19	88.95 ± 0.26	88.94 ± 0.21	88.92 ± 0.26	89.06 ± 0.21	89.19 ± 0.22
car	85.30 ± 0.36	86.23 ± 0.42	87.15 ± 0.37	87.30 ± 0.32	87.39 ± 0.36	87.97 ± 0.28

Continued on next page

Table 5.1 – continued from previous page

Dataset	1	3	5	9	17	33
cmc	57.52±0.47	58.47±0.70	58.54±0.47	59.61±0.65	59.38±0.57	59.90±0.42
ctg	75.21±0.46	78.84±0.37	80.12±0.21	81.14±0.29	81.65±0.25	81.96±0.20
cvf	76.89±0.39	77.58±0.18	77.96±0.17	78.13±0.15	78.08±0.18	77.98±0.10
eb	53.53±1.07	60.18±0.60	63.73±0.31	65.95±0.21	66.77±0.11	67.16±0.07
eye	59.28±0.19	59.58±0.27	59.53±0.21	59.57±0.15	59.58±0.12	59.66±0.10
ger	95.70±0.52	94.98±0.58	95.15±0.39	95.92±0.36	96.10±0.34	96.04±0.31
gls	82.07±0.71	82.77±0.66	83.83±0.85	83.87±0.89	84.97±0.73	85.08±0.71
hep	91.20±0.62	92.80±0.60	92.59±0.73	92.44±0.62	92.57±0.61	92.93±0.45
hrtc	72.15±0.45	74.92±0.65	75.83±0.58	77.20±0.74	77.57±0.68	77.69±0.63
hrts	88.59±0.35	88.67±0.34	88.84±0.41	89.05±0.41	89.14±0.32	89.42±0.41
ion	93.14±0.63	93.83±0.54	94.05±0.47	94.43±0.41	94.94±0.30	94.97±0.33
irs	98.29±0.23	98.43±0.35	98.43±0.25	98.51±0.30	98.51±0.28	98.48±0.27
jvow	78.13±0.73	83.83±0.34	86.84±0.28	89.24±0.24	90.67±0.21	91.61±0.11
krkopt	39.03±0.47	41.56±0.24	43.59±0.30	44.85±0.21	45.57±0.22	45.89±0.16
letter	56.73±0.68	63.92±0.47	69.68±0.25	73.45±0.22	76.07±0.22	77.93±0.20
liv	75.97±0.64	77.51±0.70	78.37±0.68	79.39±0.63	79.36±0.68	80.13±0.63
lym	90.76±0.64	92.05±0.72	92.24±0.59	92.81±0.53	93.05±0.51	93.03±0.53
magic	82.63±0.27	83.55±0.21	83.63±0.16	83.74±0.12	83.72±0.05	83.79±0.06
msh	97.73±0.25	97.72±0.30	97.85±0.21	98.03±0.17	98.13±0.19	98.22±0.16
nurse	89.35±0.51	89.14±0.42	90.24±0.37	91.08±0.19	91.41±0.15	91.48±0.13
page	95.74±0.16	95.60±0.13	95.36±0.11	95.48±0.10	95.48±0.06	95.49±0.07
pen	92.61±0.36	94.87±0.30	95.61±0.18	96.23±0.17	96.52±0.14	96.78±0.08
pid	79.61±0.22	80.19±0.27	80.57±0.26	80.83±0.27	81.04±0.24	81.06±0.20
psd	99.32±0.32	99.16±0.41	99.68±0.18	99.80±0.08	99.92±0.04	99.93±0.03
sb	93.46±0.02	93.44±0.01	93.43±0.01	93.42±0.00	93.42±0.00	93.42±0.00
seg	92.40±0.46	93.28±0.29	93.68±0.31	94.51±0.27	94.91±0.22	95.04±0.21
shuttle	99.35±0.09	99.36±0.10	99.48±0.08	99.55±0.04	99.56±0.05	99.59±0.03
sick	94.38±0.41	94.26±0.38	94.30±0.39	93.97±0.15	93.89±0.01	93.88±0.01
son	87.54±0.80	89.85±0.78	90.40±0.88	91.67±0.66	92.48±0.78	93.12±0.65
spect	92.71±0.36	92.08±0.42	92.41±0.36	92.58±0.37	92.61±0.39	92.63±0.35
spf	69.45±0.34	71.43±0.33	72.06±0.31	72.43±0.33	72.74±0.27	72.77±0.22
thy	95.44±0.28	95.02±0.23	95.09±0.13	94.95±0.11	94.96±0.09	94.96±0.06
ttt	74.79±0.74	74.50±0.57	74.75±0.65	75.21±0.45	75.67±0.60	75.40±0.54
veh	67.65±0.75	68.30±0.64	70.59±0.61	71.84±0.50	71.90±0.55	72.46±0.59
vene	93.65±0.21	93.69±0.21	93.68±0.17	93.84±0.21	93.96±0.21	93.87±0.17
vote	96.35±0.36	96.07±0.35	96.40±0.31	96.78±0.28	96.68±0.27	96.74±0.24

Continued on next page

Table 5.1 – continued from previous page

Dataset	1	3	5	9	17	33
vow	72.26±0.98	81.35±0.74	87.42±0.60	90.80±0.56	93.14±0.43	94.91±0.44
w21	85.20±0.18	86.68±0.11	87.31±0.12	87.59±0.09	87.93±0.08	88.10±0.07
w40	82.69±0.26	84.86±0.24	86.09±0.15	86.94±0.12	87.55±0.14	87.76±0.08
wfr	74.11±0.68	76.92±0.58	78.58±0.54	80.04±0.44	80.54±0.47	81.25±0.37
wilt	94.61±0.00	94.61±0.00	94.61±0.00	94.61±0.00	94.61±0.00	94.61±0.00
wine	55.61±0.14	56.96±0.24	57.00±0.21	57.32±0.15	57.46±0.16	57.42±0.20
zoo	98.14±0.49	98.69±0.47	98.69±0.44	98.97±0.36	98.97±0.36	99.09±0.37

The results show that an ensemle of classifiers almost always has superior accuracy over the single classifier, with few exceptions with `bank`, `page`, `sb` and `thy` datasets. Also, it can be seen that increasing the number of ensemble members helps the performance until a certain point of saturation, which is different for different datasets. The accuracy on some datasets could not be improved by using ensembles of sizes beyond 3, like `adult`, `bcw`, `ca`, `eye`, `hep`, `irs`, `lym`, `magic` and `zoo`, while for some datasets progressively larger ensembles continued to steadily advance in terms of the accuracy, like `bch`, `jvow`, `letter` and `vow`. Nevertheless, the results in the Table 5.1 show that the accuracy variance decreases the larger the ensembles are used, even when the average value shows no improvement, which is exactly what was expected. Finally, the average ranks in the Table 5.2 show indeed that larger ensembles show statistically significant improvement in the classification accuracy.

Table 5.2: *The accuracies of the ensembles with various numbers of elements*

Dataset	1	3	5	9	17	33	Dataset	1	3	5	9	17	33
adult	2	1	1	1	1	1	msh	2	2	1	1	1	1
ausc	2	2	1	1	1	1	nurse	3	3	2	1	1	1
bank	1	2	2	2	2	2	page	1	1	2	2	2	2
bc	3	2	2	2	1	1	pen	5	4	3	2	1	1
bch	6	5	4	3	2	1	pid	3	2	2	1	1	1
bcw	2	1	1	1	1	1	psd	2	2	1	1	1	1
ca	2	1	1	1	1	1	sb	1	2	2	2	2	2
car	4	3	2	1	1	1	seg	3	2	2	1	1	1
cmc	3	2	2	1	1	1	shuttle	3	2	1	1	1	1
ctg	5	4	3	2	1	1	sick	1	1	1	1	1	1
cvf	3	2	1	1	1	1	son	3	2	2	1	1	1
eb	5	4	3	2	1	1	spect	1	1	1	1	1	1
eye	2	1	1	1	1	1	spf	4	3	2	1	1	1
ger	2	2	2	1	1	1	thy	1	2	2	2	2	2
gls	3	2	1	1	1	1	ttt	2	1	1	1	1	1
hep	2	1	1	1	1	1	veh	4	3	2	1	1	1
hrtc	3	2	2	1	1	1	vene	2	1	1	1	1	1
hrts	3	2	1	1	1	1	vote	2	2	1	1	1	1
ion	3	2	2	1	1	1	vow	6	5	4	3	2	1
irs	2	1	1	1	1	1	w21	5	4	3	2	1	1
jvow	6	5	4	3	2	1	w40	5	4	3	2	1	1
krkopt	5	4	3	2	1	1	wfr	5	4	3	2	1	1
letter	6	5	4	3	2	1	wilt	1	1	1	1	1	1
liv	3	2	2	1	1	1	wine	3	2	2	1	1	1
lym	2	1	1	1	1	1	zoo	2	1	1	1	1	1
magic	2	1	1	1	1	1	**Rank**	2.98	2.29	1.86	1.39	1.16	1.08

6 Co-processor for the DT ensemble induction - *EEFTIP*

For the induction of a single DT, it was already demonstrated that the *EFTIP* co-processor can be used in a HW/SW architecture to achieve substantial speedups over the pure software implementation of the *EFTI* algorithm. Furthermore, it was explained in the Section 4.1 what was behind the decision to accelerate only the accuracy calculation task in hardware. Hence, in an attempt to achieve the same benefits for the DT ensemble induction, the *EEFTIP* co-processor proposed in this section was implemented using *EFTIP* as a module for the accuracy calculation. However, the *EEFTIP* co-processor also takes advantage of the intrinsic parallelism of the Bagging algorithm to achieve even higher speedups when compared to the pure software implementation of the *EEFTI* algorithm.

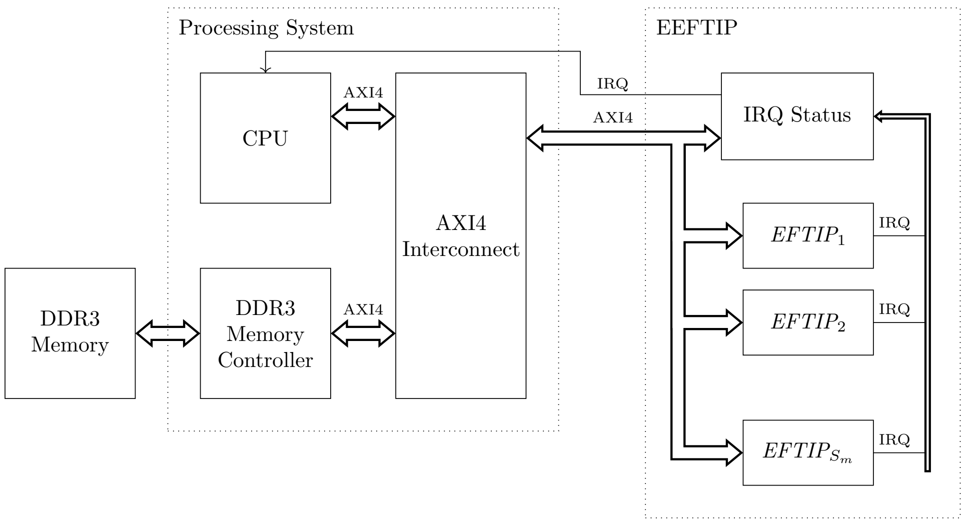


***Figure 6.1:** The* EEFTIP *co-processor structure and integration with the host CPU*

The *EEFTIP* co-processor structure and integration with the host CPU is depicted in the Figure 6.1. The *EEFTIP* consists of an array of *EFTIP* modules (described in the Section 4) *EFTIP*$_1$ to *EFTIP*$_{SM}$, each of which can be used to evaluate the accuracy of the DT individual for the induction of one ensemble member. Each *EFTIP* has its own address space and can be individually accessed for all operation described in the Section 4. In addition, the *EEFTIP* co-processor features the IRQ Status (Interrupt Request Status) block that allows the user to read-out the operation status of all *EFTIP* units. The maximal number of ensemble member accuracy calculations that can be performed in parallel equals the total number of the *EFTIP* units in the *EEFTIP* co-processor, which is a parameter that can be set during the design time of the *EFTIP* co-processor, and is called S_m. The following topics will be covered in this section:

- Section 6.1 - Description of the *EEFTIP* IRQ Status module
- Section 6.2 - Theoretical induction speedup derivation achievable by using the *EEFTIP* co-processor
- Section 6.3 - Discussion on the software routines that need to be added to the *EEFTI* algorithm, in order for it to make use of the *EEFTIP* co-processor
- Section 6.4 - Experimental section that shows the speedups that can be achieved by using the *EEFTIP* co-processor

6.1 IRQ Status Module

IRQ Status module has been implemented in order to provide the user with the means of reading the statuses of all *EFTIP* units with only one AXI4 read operation and thus optimize the AXI bus traffic. Each *EFTIP* unit comprises an IRQ (Interrupt Request) port for signaling the end of the accuracy calculation, which was in turn connected to the IRQ Status block of the *EEFTIP* co-processor. The IRQ Status block comprises an array of IRQ Status Word Registers representing the statuses of all *EFTIP* units, which can all be read in a single burst via the AXI bus. Additionally, the IRQ Status block provides a combined IRQ signal, which is triggered each time any of the *EFTIP* units signal their corresponding IRQ outputs, i.e. each time any of the *EFTIP* units finish the accuracy calculation.

Each IRQ Status Word is a 32-bit register (since *EEFTIP* was optimized for 32-bit AXI) packed from the bits representing the statuses of up to 32 *EFTIP* units each. Each bit is called *EFTIP*$_i$ Status Bit, where *i* denotes the ID of the *EFTIP* unit whose status the bit is tracking, as shown in the Figure 6.2. The figure shows IRQ Status register space for one specific S_m value, but there are no limitations on the number of *EFTIP* units that can be connected to the IRQ Status block. The bits of the IRQ Status Word Register are sticky, i.e. set each time the IRQ is signaled from the corresponding *EFTIP* and cleared when the register is read by the user.

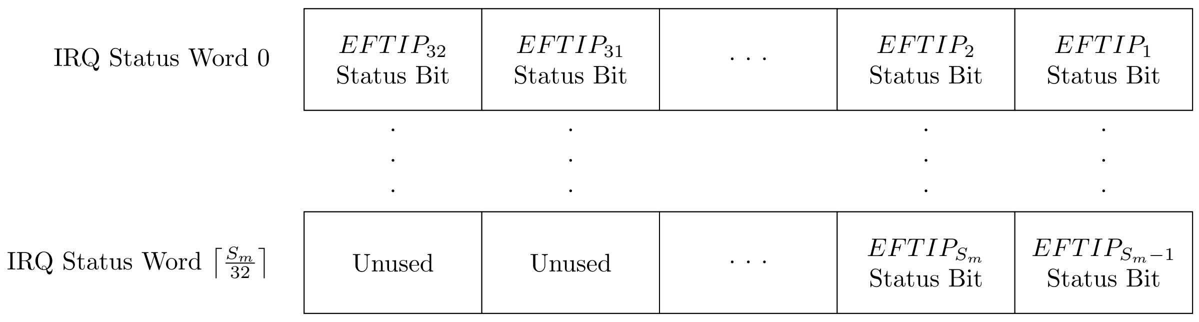


Figure 6.2: *IRQ Status register space*

6.2 Theoretical estimation of the acheivable speedup of the proposed HW/SW system

In this section the speedup of the HW/SW implementation over the pure software implementation of the *EEFTI* algorithm will be calculated as a function of the number of the ensemble members, n_e:

$$\text{speedup}(n_e) = \frac{T_{sw}(n_e)}{T_{hs}(n_e)} \tag{32}$$

where T_{sw} and T_{hs} denote the run times of the pure software and HW/SW implementations respectively. As already discussed, the good candidate for the hardware acceleration of the *EEFTI* algorithm is the accuracy calculation task, while leaving the mutation and selection to be implemented in software has some flexibility benefits. Hence, the contributions of these two parts to the total algorithm runitme will be observed separately:

$$\text{speedup}(n_e) = \frac{T_{sw\_ms}(n_e) + T_{sw\_acc}(n_e)}{T_{hs\_ms}(n_e) + T_{hs\_acc}(n_e)} \tag{33}$$

where T_{sw_ms} and T_{sw_acc} denote the amount of time pure software implementation spends on the mutation/selection and the accuracy calculation tasks respectively, while T_{hs_ms} and T_{hs_acc} represent the same values for the HW/SW implementation. T_{sw_ms} and T_{hs_ms} are linear functions of n_e, since the mutation is performed once per iteration per ensemble member. Hence, if the number of iterations is kept constant, we obtain:

$$\begin{aligned} T_{sw\_ms}(n_e) &= T_{sw\_ms}(1) \cdot n_e, \\ T_{hs\_ms}(n_e) &= T_{hs\_ms}(1) \cdot n_e, \end{aligned} \tag{34}$$

which when combined with the equation (33) yield the following:

$$\text{speedup}(n_e) = \frac{T_{sw\_ms}(1) \cdot n_e + T_{sw\_acc}(n_e)}{T_{hs\_ms}(1) \cdot n_e + T_{hs\_acc}(n_e)} \tag{35}$$

Please observe that the T_{hs_ms} is somewhat greater than the T_{sw_ms} ($T_{hs\_ms} = T_{sw\_ms} + \Delta_t$) since it also comprises the latency of the hardware accelerator interface operations, which is not present in the pure software implementation.

Depending on which approach to forming the training subsets is used, T_{sw_acc} and T_{hs_acc} will behave differently with respect to n_e. When random sampling without replacement is used, T_{sw_acc} and T_{hs_acc} are constant with respect to n_e, since the training set is partitioned amongst ensemble members, making the number of instances being classified and thus the amount of computation, constant. However, when random sampling with replacement is used, these times will tend to grow with n_e with the worst case being when $N_{IS} = N_I$, i.e. when whole training set is used for each individual. Hence, the behavior of the speedup for these two corner cases will be discussed next.

6.2.1 Random sampling without replacement

Because the HW/SW accuracy calculation is performed in parallel for all ensemble members, the calculation time is proportional to the size of the training subset allocated for each ensemble member. Since the training set is divided equally among the ensemble members (using the *EEFTIP* co-processor), T_{hs_acc} is inversely proportional to the n_e:

$$T_{hs\_acc}(n_e) = \frac{T_{hs\_acc}(1)}{n_e} \tag{36}$$

By incorporating the fact that T_{sw_acc} is constant in this case and substituting equation (36) into the (32), we obtain:

$$\text{speedup}(n_e) = \frac{T_{sw\_ms}(1) \cdot n_e + T_{sw\_acc}}{T_{hs\_ms}(1) \cdot n_e + \frac{T_{hs\_acc}(1)}{n_e}} = \frac{T_{sw\_ms}(1) \cdot n_e^2 + T_{sw\_acc} \cdot n_e}{T_{hs\_ms}(1) \cdot n_e^2 + T_{hs\_acc}(1)} \tag{37}$$

T_{sw_acc} term was shown in the Section 3.4 and the Section 4.1 to take almost all of the computational time. The datasets that can be of interest to run DT ensemble induction on using the *EEFTIP* are the ones that require significant time to execute in the software on the CPU. For these datasets $T_{sw\_acc} \gg T_{sw\_ms}$ and thus $T_{sw\_acc} \gg T_{hs\_ms}$. By using the hardware acceleration and massive parallelism, $T_{sw\_acc} \gg T_{hs\_acc}$ is accomplished as well. By taking these parameter relationships into the account, $(speedup)(n_e)$ function given by the equation (37) takes shape depicted in the Figure 6.3.

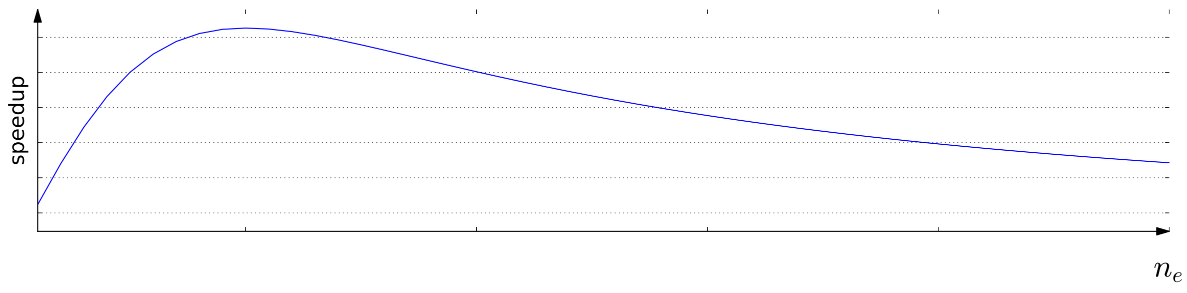


***Figure 6.3:** The shape of the speedup(n_e) function given by the equation (37).*

The plot in the Figure 6.3 suggests that accelerating the *EEFTI* by a co-processor that performs the DT accuracy calculation in parallel for all ensemble members, will provide an increase in the speedup as the number of ensemble members increases in the beginning. Then, after a speedup maximum has been reached, it will slowly degrade, but continue to offer a substantial speedup for all reasonable ensemble sizes. The maximum of the speedup can be found by seeking the maximum of the function given by the equation (37). By taking into the account parameter relationships, the point of the maximum of the speedup(n_e) function can be expressed as follows:

$$max(\text{speedup}(n_e)) \approx \frac{T_{sw\_acc}}{2\sqrt{T_{hs\_acc}(1)T_{hs\_ms}(1)}} \; at \; n_e \approx \sqrt{\frac{T_{hs\_acc}(1)}{T_{hs\_ms}(1)}} \tag{38}$$

Furthermore, the Figure 6.3 shows that even though the speedup starts declining after reaching its maximum value for certain n_e, the downslope is slowly flattening, and the significant speedup is achieved even for large ensemble sizes.

6.2.2 Whole training set for each member

In this case, the total number of instances in the ensemble rises linearly with the number of ensemble members. This means that T_{sw_acc} will rise linearly and T_{hs_acc} will remain constant being that it is performed in parallel. This yields the following form for the speedup function:

$$\text{speedup}(n_e) = \frac{T_{sw\_ms}(1) \cdot n_e + T_{sw\_acc}(1) \cdot n_e}{T_{hs\_ms}(1) \cdot n_e + T_{hs\_acc}(1)} \tag{39}$$

Taking into the account that $T_{sw} = T_{hs\_ms} + T_{sw\_acc}$, and rearanging the equation (39), the following is obtained:

$$\text{speedup}(n_e) = \frac{T_{sw}(1)}{T_{hs\_ms}(1)} \cdot \frac{1}{1 + \frac{T_{hs\_acc}(1)}{T_{hs\_ms}(1) \cdot n_e}} \tag{40}$$

The equation (40) shows that the speedup increases with the number of ensemble members induced and asymptotically converges to the ratio of the total time needed for the single member induction in software ($T_{sw}(1)$) to the time needed for the mutation/selection tasks in the HW/SW co-design implementation ($T_{hs\_ms}(1)$), which basically means that the speedup can be increased by optimizing the execution time of the mutation/selection tasks and the communication with the co-processor.

6.3 Software for the *EEFTIP* assisted DT ensemble induction

As it was described in the previous chapters, the *EEFTIP* co-processor can perform accuracy evaluation task in parallel for as many ensemble members as there are *EFTIP* units within. Hence, in the HW/SW implementation of the *EEFTI* algorithm, each of the *EFTI* tasks is assigned one *EFTIP* unit to use exclusively for the acceleration of the accuracy evaluation for its DT individual. Since there is a single AXI bus connecting the CPU to the *EEFTIP* co-processor, no two *EFTI* tasks can access it in the same time.

The *EFTI* tasks could be left alone to compete for the rights to use *EEFTIP* and check whether their corresponding *EFTIP* unit has finished computing the accuracy, but there is a more economical approach that utilizes the IRQ Status module of the *EEFTIP* co-processor. In this approach, the *EFTI* tasks are disallowed to poll the status registers of their *EFTIP* units. Their responsibility is to load the DT individuals and start the accuracy calculation process. On the other hand a management task, called the Scheduler, is introduced to exclusively monitor the registers of the IRQ Status module and inform the individual *EFTI* tasks about the completion of their accuracy calculation processes. This is done by using semaphores of the underlying operating system, which are used to signal the *EFTI* tasks that the accuracy calculation has been completed and the access to the *EEFTIP* has been now granted to them, so that they can update the DT information and start the new calculation cycle. As soon as the accuracy calculation on their corresponding *EFTIP* unit is started, the control is given back to the Scheduler task and *EFTI* waits for the new completion signal via semphore.

Algorithm 6.1: *The pseudo-code of the Scheduler task used in the HW/SW co-design implementation*

```
def scheduler(tasks, semaphores):
    while(not all_finished(tasks)):
        status = hw_read(eeftip_irq_status_addr())

        for eftip_id, eftip_stat in enumerate(status):
            if eftip_stat == 1:
                semaphore_give(semaphores[eftip_id])

        context_switch()
```

The pseudo-code for the Scheduler task is given in the Algorithm 6.1. The main task of the Scheduler task is to poll the IRQ Status register (whose address is returned by the `eeftip_irq_status_addr()` helper function) of the *EEFTIP* co-processor in a loop. It then iterates through the received status value to check which *EFTIP* units have reported to have finished the accuracy calculation, and activates the correponding *EFTI* tasks. After all the required tasks have been informed, the Scheduler issues a call to the `context_switch()` function of the underlying OS, so that the OS can serve the other tasks that have been activated via emited semaphores. The loop ends when all the tasks have finished the induction and exited, which is monitored by the `all_finished()` helper function.

Algorithm 6.2: *The pseudo-code of the* EEFTI *algorithm using the* EEFTIP *co-processor*

```
def eefti(train_set, ensemble_size):
    train_par = divide_train_set(train_set, ensemble_size)

    res = []
    semaphores = []
    tasks = []
    for eftip_id in range(ensemble_size):
        r = {}
        s = create_semaphore()
        t = create_task(efti, train_par[eftip_id], r, eftip_id, s)
        res.append(r)
        semaphores.append(s)
        tasks.append(t)

    scheduler(tasks, semaphores)

    return res
```

The *EEFTI* top level pseudo-code with the added instantiation of the synchronization mechanism in the form of the Scheduler task and the semaphores is presented in the Algorithm 6.2. In addition to the training set and the reference to the result object `r`, each of the *EFTI* tasks created is assigned a semaphore handle, and the unique ID (variable `eftip_id`) that serves as a handle to the *EFTIP* unit of the *EEFTIP* co-processor assigned to the task. After all the *EFTI* tasks have been created, the control is transfered to the Scheduler task until all ensemble members have been induced.

The HW/SW implementation of almost all of the *EFTI* tasks, which were described in the Section 4.5, is used almost verbatim for the HW/SW implementation of the *EEFTI* algorithm. One difference is that here a co-processor with multiple *EFTIP* units is accessed by the software. Hence, all the helper functions of the HW/SW implementation of the *EFTI* algorithm for calculating the appropriate hardware memory addresses, need now take into the account the ID of the *EFTIP* unit (`eftip_id`) they are interfacing. The second needed change was to adapt the code from the Algorithm 3.3 for the `accuracy_calc()` function to support the described protocol for the access rights delegation using semaphores. The adapted function pseudo-code is shown in the Algorithm 6.3.

Algorithm 6.3: *The pseudo-code of the fitness evaluation function used in the HW/SW co-design implementation*

```
def accuracy_calc(train_set, eftip_id, semaphore):
    hw_write(eftip_operation_control_addr(eftip_id), EFTIP_START)
    semaphore_wait(semaphore)
    hits = hw_read(eftip_result_addr(efipt_id))

    return hits/len(train_set)
```

The Figure 6.4 shows the benefits of careful scheduling scheme over the naive solution where each *EFTI* task is let to finish whole iteration before the other is let to start. The diagram in the figure shows how occupied with different *EFTI* tasks is the CPU and the *EEFTIP* co-processor for these two different scenarios, where the operations related to the different *EFTI* tasks are given in different colors. Time periods marked with letters M and S represent the mutation and selection tasks respectively, and the idle periods of the CPU are showed hatched in figure.

With the naive approach shown in the Figure 6.4a, a lot of CPU time is waisted on waiting for the accuracy calculation to finish, and the potential of the parallel *EFTIP* units is not exploited. By introducing the Scheduler task and making the `accuracy_calc()` function suspend its execution and return the control back as soon as it finishes with the mutation task, sets and starts the accuracy calculation on its corresponding *EFTIP* unit, the HW/SW architecture that uses the *EEFTIP* co-processor can be exploited to its full potential, which leads to the timing diagram shown in the Figure 6.4b.

6.4 Experiments

To estimate the DT ensemble induction speedup of the HW/SW implementation over the pure software implementation of the *EEFTI* algorithm, the experiments have been performed on the induction of the ensembles of up to 25 members and the results are given in this section. In order to support the datasets with higher number of intances, the random sampling without replacement was used to form the training subsets, in order to make them smaller.

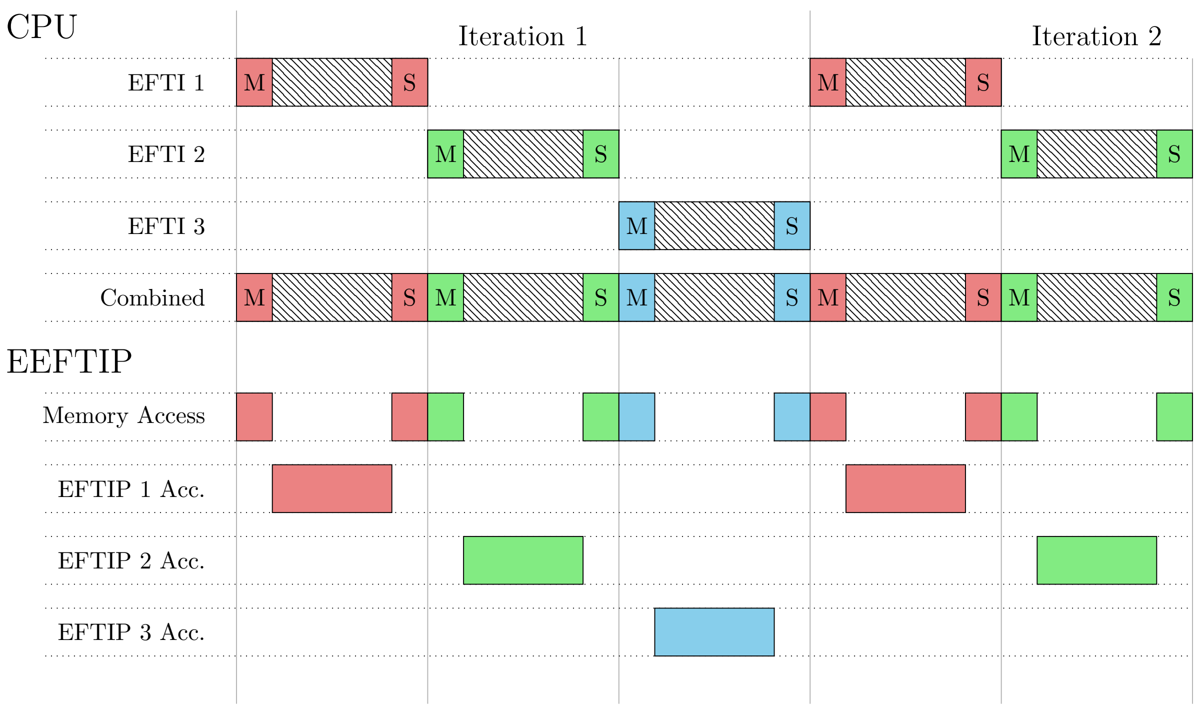
6.4.1 Required Hardware Resources for the *EEFTIP* co-processor

For the experiments, five different instances of the *EEFTIP* co-processor were generated, one for each of the following ensemble sizes: 2, 4, 8, 16 and 25. The values of the customization parameters, given in the Table 6.1, were chosen so that the generated co-processors could fit inside the XC7Z100 Xilinx Zynq device that was used for testing.

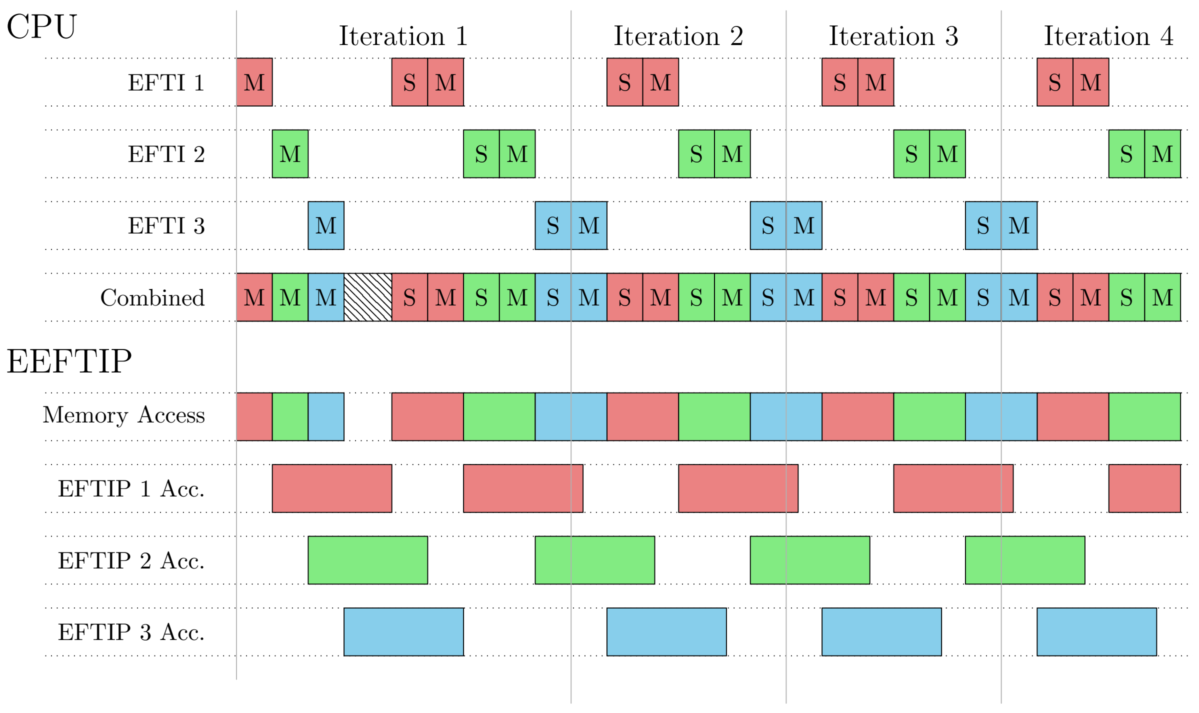
Table 6.1: *Values of the customization parameters of the* EEFTIP *co-processor instances, one for each of the ensemble sizes used in the experiments.*

Parameter	$S_m = 2$	$S_m = 4$	$S_m = 8$	$S_m = 16$	$S_m = 25$
DT max. depth (L_m)	5	5	5	5	5
Max. attributes num. (A_m)	16	16	16	16	16
Attribute encoding resolution (R_A)	16	16	16	16	16
Class encoding resolution (R_C)	8	8	8	8	8
Max. training set classes (C_M)	64	64	64	64	64
Max. number of leaves (ACE_m)	16	16	16	16	16
Max. number of training set instances (I_m)	24000	12000	6000	4096	2048
Max. number of nodes per level (N_{lm})	16	16	16	16	16

The VHDL language has been used to model the *EEFTIP* co-processor and it was implemented using the Xilinx Vivado Design Suite 2015.2 software for the logic synthesis and implementation with the default synthesis and P&R options. From the implementation



(a) Sequential operation



(b) Interlaced operation

Figure 6.4: *Achieving the maximum CPU utilization by interlacing the inducion operations of different ensemble members **(b)**, as opposed to performing these operations sequentially **(a)**.*

report files, device utilization data has been analyzed for the *EEFTIP* co-processor instance with S_m = 25 (Table 4.5), which has the largest footprint. The information about the number of used slices, BRAMs and DSP blocks has been extracted, and is presented in the Table 6.2, for different target FPGA devices. The operating frequency of 100 MHz of the system clock frequency was attained for all the implemented *EEFTIP* co-processor instances from the Table 4.5.

Table 6.2: *FPGA resources required to implement the* EEFTIP *co-processor with 25* EFTIP *units and the configuration given in the Table 4.5.*

FPGA Device	Slices/CLBs	BRAMs	DSPs
XC7Z100	62091 (89%)	412.5 (55%)	2000 (99%)
XCKU115	33231 (40%)	412.5 (19%)	2000 (36%)
XC7VX690	63885 (59%)	412.5 (28%)	2000 (56%)

Given in the brackets along with each resource utilization number is a percentage of used resources from the total resources available in the corresponding FPGA devices. Table 6.2 shows that the implemented *EEFTIP* co-processor fits into xc7z100 Xilinx FPGA device of the Zynq series, and into mid- to high-level Virtex7 and UltraScale Kintex7 Xilinx FPGA devices (XC7VX690 and XCKU115).

6.4.2 Estimation of the Induction Speedup

For the experiments, the *EEFTI* algorithm was implemented for three platforms (all software was written in the C programming language):

- **SW-PC**: Pure software implementation for the PC
- **SW-ARM**: Pure software implementation for the ARM Cortex-A9 processor
- **HW/SW**: The *EEFTIP* co-processor implemented in the FPGA was used for the fitness evaluation, while all other tasks of the *EEFTI* algorithm (shown in the Algorithm **??**) were implemented in software for the ARM Cortex-A9 processor.

For the software implementations of the *EEFTI* algorithm on the ARM platform, at first the FreeRTOS was used as the operating system since it has a port for the ARM Cortex-A9 and it is open source. However, experiments showed that it has rather high task switching latency, which degraded the execution speed of the HW/SW implementation. In lack of other open source RTOSes ported for the ARM Cortex-A9 that we could find, a simple simple cooperative scheduler was developped to be used for the SW-ARM and HW/SW implementations.

For the PC implementation, a 64-bit, 4-core, Intel i5-2500K CPU operating at approximately 3.5GHz, with 8GB or RAM, running Ubuntu 16.04 operating system platform was used and the software was built using the GCC 5.4.1 compiler. For the SW-ARM and HW/SW implementations, ARM Cortex-A9 was used running at 667MHz. The software was built using the Sourcery CodeBench Lite ARM EABI 4.9.1 compiler (from within the Xilinx SDK 2015.2) and the *EEFTIP* co-processor was built using the Xilinx Vivado Design Suite 2015.2.

Not all of the datasets from the Table 2.1 were used in these experiments for two different reasons. Some of the datasets, like `ausc`, `bc`, `bcw`, `ger`, `gls`, `hep`, `hrtc`, `hrts`, `ion`, `irs`, `liv`, `lym`, `pid`, `son`, `ttt`, `veh`, `vote`, `vow` and `zoo`, have too few instances to support induction of up to 25 members using training set partitioning by sampling without replacement. On the other hand, some datasets like `sick`, ``spf`, `thy` and `w40` had too many attributes to fit into the implemented co-processors. The others, like `mushroom`, `w21` and `wfr` were preprocessed using the PCA (Principal Component Analysis) to reduce their number of attributes to 16, what the implemented co-processors support. For each of the datasets, five experiments were performed in which the ensembles were induced with: 2, 4, 8, 16 and 25 members. For each of these experiments five 5-fold cross-validations has been carried out and the DT ensemble classifier induction times have been measured.

The results of the experiments are presented in the Table 6.3. The table contains the speedups of the HW/SW implementation over the SW-ARM and SW-PC implementations for each dataset and the ensemble size. At the bottom of the table, the average speedups are given for each ensemble size.

Table 6.3: *The speedups of the HW/SW implementation over the SW-ARM and SW-PC implementations for each dataset and ensemble size.*

	SW-ARM					PC-ARM				
Dataset	2	4	8	16	25	2	4	8	16	25
adult	24.68	43.87	60.32	74.89	48.04	8.92	16.88	19.42	24.28	15.76
bank	26.86	50.18	76.21	98.37	73.82	9.28	17.38	31.84	31.64	23.66
bch	37.57	54.48	44.13	17.66	10.86	13.52	19.04	15.88	7.10	4.36
cvf	31.14	48.55	63.44	33.70	18.58	11.22	15.70	20.14	11.16	6.32
eb	30.25	52.83	70.47	63.44	43.12	16.86	30.34	30.12	27.52	18.52
eye	22.03	31.39	48.99	34.58	17.77	8.46	9.62	15.06	11.34	6.06
jvow	34.84	50.48	48.28	24.13	14.44	16.40	19.88	19.24	10.22	6.60
krkopt	30.10	45.60	56.04	41.16	26.72	15.34	27.16	24.52	18.72	11.66
letter	47.84	68.32	72.24	45.85	30.34	20.92	32.54	30.38	19.04	11.94
magic	19.43	26.70	43.77	36.47	20.06	9.20	10.12	16.06	14.02	7.96
msh	17.90	30.80	37.53	16.44	8.96	6.20	9.72	12.12	5.52	3.46
nurse	24.33	40.07	52.21	29.02	16.47	12.46	16.60	21.28	11.70	6.90
page	19.43	28.50	24.20	10.92	6.42	6.80	10.22	9.34	4.10	2.52
pen	42.50	54.34	50.63	27.42	16.46	19.30	22.52	20.96	11.64	7.30
shuttle	28.62	55.75	93.88	119.97	93.21	12.92	23.24	43.38	43.58	33.50
w21	25.37	40.33	34.95	14.41	8.83	10.36	16.06	13.26	5.92	3.54
wfr	26.89	42.53	37.70	16.17	9.71	9.80	15.10	12.94	6.18	3.74
wine	23.59	33.09	24.66	10.91	6.85	9.08	12.62	10.06	4.76	3.00
Avg.:	28.52	44.32	52.20	39.75	26.15	12.06	18.04	20.34	14.92	9.82

Table 6.3 indicates that the average speedup of the HW/SW implementation is between 26 and 52 times over the SW-ARM and between 10 and 20 times over the SW-PC implementation,

depending on the number of the ensemble members induced. It can be seen that the speedups follow the theoretical curve from the Figure 6.3 shown in the Section *Theoretical estimation of the acheivable speedup of the proposed HW/SW system*, which is also visible in the Figure 6.5. In the Figure 6.5 each bar represents the speedup for one ensemble size, hence the envelope of the bar graph for each dataset correlates with the theoretical speedup curve. It should be noted that the envelopes appear distorted, since ensemble sizes for which the speedups are drown as bars are not equidistant, but follow the exponential function. By observing the speedup of the HW/SW implementation over the pure software implementations shown in the Table 6.3 for each dataset used in the experiments, it can be seen that more speedup is gained for datasets with larger N_I, N_A and N_C.

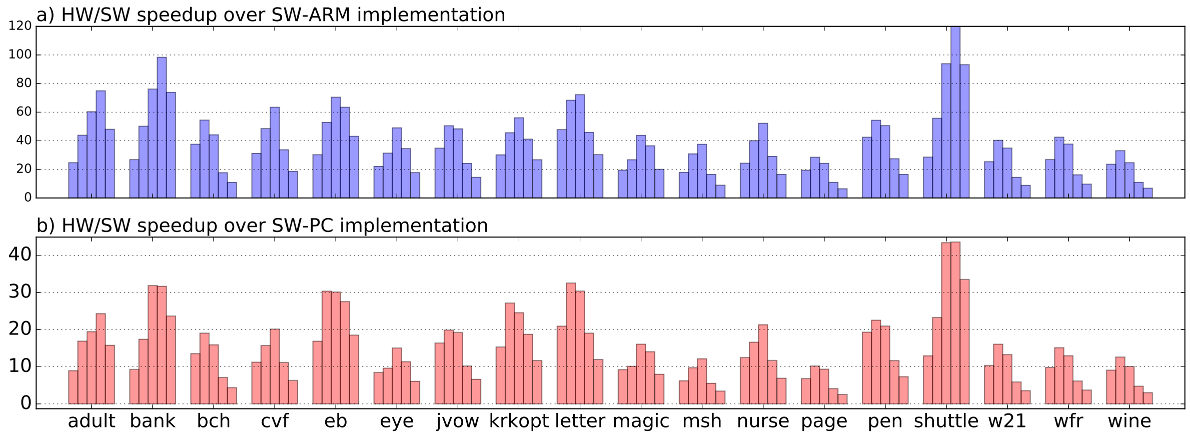


Figure 6.5: *Speedup of the HW/SW implementation over a) SW-ARM implementation and b) SW-PC implementation, given for each dataset used in the experiments. Each bar represents a speedup for one ensemble size.*

Figure 4.22 and Table 6.3 suggest that the HW/SW implementation using *EEFTIP* co-processor offers a substantial speedup in comparison to the pure software implementations for both PC and ARM. Furthermore, the *EEFTIP* implementation used in the experiments operates at much lower frequency (100MHz) than both ARM (667MHz) and PC(3.5GHz) platforms. If *EEFTIP* co-processor were implemented in ASIC, the operating frequency would be increased by an order of magnitude, and the DT induction speedup would increase accordingly.

7 Conclusion

This thesis was concerned with the evolutionary induction of the oblique binary decision trees using nonincremental approach. Two algorithms were presented, one for a single classifier induction called *EFTI*, and the other for a decision tree ensemble induction called *EEFTI*. Furthermore, two architectures were proposed for hardware acceleration of the two induction algorithms.

The proposed *EFTI* algorithm was created in an attempt to devise a strategy for the DT induction that would induce smaller DTs than the existing solutions without the loss in accuracy, but try to use as little resources for the induction as possible. It was shown in the thesis that the *EFTI* algorithm succeeds in fulfilling the requirements that were set in the beginning:

- It operates only on one DT individual, unlike many full DT induction algorithms that use the populations of 20 to 100 or more individuals. This implies a 20 to 100 fold times less resources needed for its implementation, which are critical in embedded systems. Furthermore, the most time consuming task of accuracy calculation, besides the control flow, comprises only the simple operations of multiplication and addition performed for the node test evaluations. Hardware blocks that perform these operations are found in abundance within the chips used in embedded systems, such as DSPs and FPGAs.
- It is easily parallelizable. Within the accuracy calculation, each instance from the training set traverses the DT by itself and the traversal is decoupled from the traversals of other instances, which is suitable for parallelization by either pipelining or completely performing the instance traversal in parallel.
- It produces smaller DTs than the existing solutions, without the loss in DT accuracy, which was proved by the experiments in the Section 3.5.

Second, a parameterizable co-processor for the hardware aided DT induction using an evolutionary approach, called *EFTIP*, was proposed. The *EFTIP* co-processor can be used for the hardware acceleration of the DT accuracy evaluation task, since this task was proven in the Section 3.4 and Section 4.1 to be the execution time bottleneck. The *EFTI* algorithm was adapted to take advantage of the *EFTIP* co-processor in a HW/SW co-design architecture. Comparison of the HW/SW *EFTI* algorithm implementation with the pure software implementations suggests that the proposed HW/SW architecture offers substantial speedups for all the tests performed on the selected UCI datasets.

Next, the *EEFTI* algorithm was presented which uses the *EFTI* algorithm together with Bagging to induced the DT ensembles. The experimental results discussed in the Section 5.3, show that the ensembles induced by the *EEFTI* algorithm are superior in terms of the classification accuracies than the single classifier DTs induced by the *EFTI* algorithm.

Finally, a parameterizable co-processor, called *EEFTIP*, is proposed. The *EEFTIP* co-processor can be used for the hardware aided induction of the DT ensembles using EA. It was shown in the paper that the *EEFTI* algorithm spends most of the execution time in the DT accuracy evaluation process, hence the *EEFTIP* co-processor was developed to accelerate that task. The *EEFTI* algorithm has been implemented in the software and modified to use the *EEFTIP* co-processor implemented in the FPGA as a co-processor. Comparison of the HW/SW implementation of the *EEFTI* algorithm with the pure software implementations suggests that the proposed HW/SW architecture offers substantial speedups for all tests performed on selected UCI datasets.

References

[1] Hugh G Gauch. *Scientific method in practice*. Cambridge University Press, 2003.

[2] Peter Flach. *Machine learning: the art and science of algorithms that make sense of data*. Cambridge University Press, 2012.

[3] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[4] Jesse Levinson, Jake Askeland, Jan Becker, Jennifer Dolson, David Held, Soeren Kammel, J Zico Kolter, Dirk Langer, Oliver Pink, Vaughan Pratt, and others. Towards fully autonomous driving: systems and algorithms. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, 163–168. IEEE, 2011.

[5] Jesmin F Khan, Sharif MA Bhuiyan, and Reza R Adhami. Image segmentation and shape analysis for road-sign detection. *IEEE Transactions on Intelligent Transportation Systems*, 12(1):83–96, 2011.

[6] Lior Rokach. *Data mining with decision trees: theory and applications*. World scientific, 2007.

[7] Lior Rokach and Oded Maimon. Top-down induction of decision trees classifiers-a survey. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 35(4):476–487, 2005.

[8] Shigeo Abe. *Support vector machines for pattern classification*. volume 53. Springer, 2005.

[9] Simon S Haykin, Simon S Haykin, Simon S Haykin, and Simon S Haykin. *Neural networks and learning machines*. volume 3. Pearson Education Upper Saddle River, 2009.

[10] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[11] Jorma Rissanen. *Minimum description length principle*. Wiley Online Library, 1985.

[12] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[13] Simon JD Prince. *Computer vision: models, learning, and inference*. Cambridge University Press, 2012.

[14] Sudha Challa. *Fundamentals of object tracking*. Cambridge University Press, 2011.

[15] Usman Ali and Mohammad Bilal Malik. Hardware/software co-design of a real-time kernel based tracking system. *Journal of Systems Architecture*, 56(8):317–326, 2010.

[16] Matteo Tomasi, Francisco Barranco, Mauricio Vanegas, Javier Díaz, and E Ros. Fine grain pipeline architecture for high performance phase-based optical flow computation. *Journal of Systems Architecture*, 56(11):577–587, 2010.

[17] Arthur Lesk. *Introduction to bioinformatics*. Oxford University Press, 2013.

[18] Pierre Baldi and Søren Brunak. *Bioinformatics: the machine learning approach*. MIT press, 2001.

[19] Bing Liu. *Web data mining: exploring hyperlinks, contents, and usage data*. Springer Science & Business Media, 2007.

[20] Matthew A Russell. *Mining the Social Web: Data Mining Facebook, Twitter, LinkedIn, Google+, GitHub, and More*. O'Reilly Media, Inc., 2013.

[21] Sholom M Weiss, Nitin Indurkhya, and Tong Zhang. *Fundamentals of predictive text mining*. Springer Science & Business Media, 2010.

[22] Charu C Aggarwal and ChengXiang Zhai. *Mining text data*. Springer Science & Business Media, 2012.

[23] Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *Journal of artificial intelligence research*, 1994.

[24] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[25] JR Quinlan. C4. 5: programs for empirical learning morgan kaufmann. *San Francisco, CA*, 1993.

[26] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[27] Erick Cantu-Paz and Chandrika Kamath. Inducing oblique decision trees with evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 7(1):54–68, 2003.

[28] Ali Mirza Mahmood, K Mrutunjaya Rao, Kiran Kumar Reddi, and others. A novel algorithm for scaling up the accuracy of decision trees. *International Journal on Computer Science and Engineering*, 2(2):126–131, 2010.

[29] Olcay Taner Yıldız. Univariate decision tree induction using maximum margin classification. *The Computer Journal*, 55(3):293–298, 2012.

[30] Asdrúbal López-Chau, Jair Cervantes, Lourdes López-García, and Farid García Lamont. Fisher's decision tree. *Expert Systems with Applications*, 40(16):6283–6291, 2013.

[31] Rodrigo C Barros, Pablo A Jaskowiak, Ricardo Cerri, and Andre CPLF de Carvalho. A framework for bottom-up induction of oblique decision trees. *Neurocomputing*, 135:3–12, 2014.

[32] Rastislav Struharik, Vuk Vranjkovic, Stanisa Dautovic, and Ladislav Novak. Inducing oblique decision trees. In *Intelligent Systems and Informatics (SISY), 2014 IEEE 12th International Symposium on*, 257–262. IEEE, 2014.

[33] Athanassios Papagelis and Dimitrios Kalles. Ga tree: genetically evolved decision trees. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, 0203–0203. IEEE Computer Society, 2012.

[34] Xavier Llora and Stewart W Wilson. Mixed decision trees: minimizing knowledge representation bias in lcs. In *Genetic and Evolutionary Computation–GECCO 2004*, 797–809. Springer, 2004.

[35] Stephen F Smith. Flexible learning of problem solving heuristics through adaptive search. In *IJCAI*, volume 83, 422–425. 1983.

[36] Martijn CJ Bot and William B Langdon. Application of genetic programming to induction of linear classification trees. In *Genetic Programming*, pages 247–258. Springer, 2000.

[37] Fernando EB Otero, Alex A Freitas, and Colin G Johnson. Inducing decision trees with an ant colony optimization algorithm. *Applied Soft Computing*, 12(11):3615–3626, 2012.

[38] Urszula Boryczka and Jan Kozak. Enhancing the effectiveness of ant colony decision tree algorithms by co-learning. *Applied Soft Computing*, 30:166–178, 2015.

[39] Rodrigo Coelho Barros, Marcio Porto Basgalupp, ACPLF De Carvalho, and Alex Alves Freitas. A survey of evolutionary algorithms for decision-tree induction. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(3):291–312, 2012.

[40] Marek Krętowski and Marek Grześ. Global induction of oblique decision trees: an evolutionary approach. In *Intelligent Information Processing and Web Mining*, pages 309–318. Springer, 2005.

[41] Jason Brownlee. *Clever algorithms: nature-inspired programming recipes*. Jason Brownlee, 2011.

[42] B Vukobratovic and R Struharik. Evolving full oblique decision trees. In *Computational Intelligence and Informatics (CINTI), 2015 16th IEEE International Symposium on*, 95–100. IEEE, 2015.

[43] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.

[44] Alok N Choudhary, Daniel Honbo, Prabhat Kumar, Berkin Ozisikyilmaz, Sanchit Misra, and Gokhan Memik. Accelerating data mining workloads: current approaches and future challenges in system architecture design. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):41–54, 2011.

[45] Davide Anguita, Andrea Boni, and Sandro Ridella. A digital architecture for support vector machines: theory, algorithm, and fpga implementation. *Neural Networks, IEEE Transactions on*, 14(5):993–1009, 2003.

[46] Markos Papadonikolakis and C Bouganis. Novel cascade fpga accelerator for support vector machines classification. *Neural Networks and Learning Systems, IEEE Transactions on*, 23(7):1040–1052, 2012.

[47] Davide Anguita, Luca Carlino, Alessandro Ghio, and Sandro Ridella. A fpga core generator for embedded classification systems. *Journal of Circuits, Systems, and Computers*, 20(02):263–282, 2011.

[48] Davood Mahmoodi, Ali Soleimani, Hossein Khosravi, Mehdi Taghizadeh, and others. Fpga simulation of linear and nonlinear support vector machine. *Journal of Software Engineering and Applications*, 4(05):320, 2011.

[49] Vuk Vranjkovic and Rastislav Struharik. New architecture for svm classifier and its application to telecommunication problems. In *Telecommunications Forum (TELFOR), 2011 19th*, 1543–1545. IEEE, 2011.

[50] Antony Savich, Medhat Moussa, and Shawki Areibi. A scalable pipelined architecture for real-time computation of mlp-bp neural networks. *Microprocessors and Microsystems*, 36(2):138–150, 2012.

[51] Dmitri Vainbrand and Ran Ginosar. Scalable network-on-chip architecture for configurable neural networks. *Microprocessors and Microsystems*, 35(2):152–166, 2011.

[52] J Echanobe, I del Campo, K Basterretxea, MV Martinez, and Faiyaz Doctor. An fpga-based multiprocessor-architecture for intelligent environments. *Microprocessors and Microsystems*, 38(7):730–740, 2014.

[53] Janardan Misra and Indranil Saha. Artificial neural networks in hardware: a survey of two decades of progress. *Neurocomputing*, 74(1):239–255, 2010.

[54] Amos R Omondi and Jagath Chandana Rajapakse. *FPGA implementations of neural networks*. volume 365. Springer, 2006.

[55] Hirokazu Madokoro and Kazuhito Sato. Hardware implementation of back-propagation neural networks for real-time video image learning and processing. *Journal of Computers*, 8(3):559–566, 2013.

[56] Rastislav Struharik and Ladislav Novak. Intellectual property core implementation of decision trees. *IET computers & digital techniques*, 3(3):259–269, 2009.

[57] Qingzheng Li and Amine Bermak. A low-power hardware-friendly binary decision tree classifier for gas identification. *Journal of Low Power Electronics and Applications*, 1(1):45–58, 2011.

[58] Fareena Saqib, Aindrik Dutta, Jim Plusquellic, Philip Ortiz, and Marios S Pattichis. Pipelined decision tree classification accelerator implementation in fpga (dt-caif). *Computers, IEEE Transactions on*, 64(1):280–285, 2015.

[59] Rastislav Struharik and Ladislav Novak. Evolving decision trees in hardware. *Journal of Circuits, Systems, and Computers*, 18(06):1033–1060, 2009.

[60] Grigorios Chrysos, Panagiotis Dagritzikos, Ioannis Papaefstathiou, and Apostolos Dollas. Hc-cart: a parallel system implementation of data mining classification and regression tree (cart) algorithm on a multi-fpga system. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):47, 2013.

[61] Lior Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1-2):1–39, 2010.

[62] Peter Bühlmann. Bagging, boosting and ensemble methods. In *Handbook of Computational Statistics*, pages 985–1022. Springer, 2012.

[63] M Ozay and F Vural. Performance analysis of stacked generalization classifiers. In *2008 IEEE 16th Signal Processing, Communication and Applications Conference*. 2008.

[64] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[65] YS Huang and CY Suen. The behavior-knowledge space method for combination of multiple classifiers. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 347–347. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 1993.

[66] Amine Bermak and Dominique Martinez. A compact 3d vlsi classifier using bagging threshold network ensembles. *Neural Networks, IEEE Transactions on*, 14(5):1097–1109, 2003.

[67] Hassab Elgawi Osman. Random forest-lns architecture and vision. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, 319–324. IEEE, 2009.

[68] Brian Van Essen, Chris Macaraeg, Maya Gokhale, and Ryan Prenger. Accelerating a random forest classifier: multi-core, gp-gpu, or fpga? In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 232–239. IEEE, 2012.

[69] Hanaa Hussain, Khaled Benkrid, Chuan Hong, and Huseyin Seker. An adaptive fpga implementation of multi-core k-nearest neighbour ensemble classifier using dynamic partial reconfiguration. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, 627–630. IEEE, 2012.

[70] Rastislav JR Struharik and Ladislav A Novak. Hardware implementation of decision tree ensembles. *Journal of Circuits, Systems, and Computers*, 22(05):1350032, 2013.

[71] Minghua Shi, Amine Bermak, Shrutisagar Chandrasekaran, Abbes Amira, and Sofiane Brahim-Belhouari. A committee machine gas identification system based on dynamically reconfigurable fpga. *Sensors Journal, IEEE*, 8(4):403–414, 2008.

[72] David J Newman, Seth Hettich, Cason L Blake, and Christopher J Merz. Uci repository of machine learning databases. 1998.

[73] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. *Applied linear statistical models*. volume 4. Irwin Chicago, 1996.

[74] Yosef Hochberg and Ajit C Tamhane. *Multiple comparison procedures*. Wiley, 2009.

[75] Anne Auger. Benchmarking the (1+ 1) evolution strategy with one-fifth success rule on the bbob-2009 function testbed. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, 2447–2452. ACM, 2009.

[76] Delon Levi. Hereboy: a fast evolutionary algorithm. In *Evolvable Hardware, 2000. Proceedings. The Second NASA/DoD Workshop on*, 17–24. IEEE, 2000.

[77] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[78] Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+ 1) evolutionary algorithm. *Theoretical Computer Science*, 276(1):51–81, 2002.

[79] David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996.

[80] Santos Lopez-Estrada and Rene Cumplido. Decision tree based fpga-architecture for texture sea state classification. In *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, 1–7. IEEE, 2006.

[81] Vuk S Vranjković, Rastislav JR Struharik, and Ladislav A Novak. Reconfigurable hardware for machine learning applications. *Journal of Circuits, Systems and Computers*, 24(05):1550064, 2015.

[82] Davide Anguita, Alessandro Ghio, Stefano Pischiutta, and Sandro Ridella. A support vector machine with integer parameters. *Neurocomputing*, 72(1):480–489, 2008.

[83] Xindong Wu and Vipin Kumar. *The top ten algorithms in data mining*. CRC Press, 2009.

[84] Lipo Wang and Xiuju Fu. *Data mining with computational intelligence*. Springer Science & Business Media, 2006.

[85] David Heath, Simon Kasif, and Steven Salzberg. Induction of oblique decision trees. In *IJCAI*, 1002–1007. Citeseer, 1993.

[86] Md Zahidul Islam. Explore: a novel decision tree classification algorithm. In *Data Security and Security Data*, pages 55–71. Springer, 2010.

[87] X Liu, D Wang, L Jiang, and F Chen. An improved algorithm for oblique decision tree classification based on rough set theory. *J Comput Inf Syst*, 7(11):4042–4049, 2011.

[88] Naresh Manwani and PS Sastry. Geometric decision tree. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 42(1):181–192, 2012.

[89] UCI. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/.

[90] Pong P Chu. *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. John Wiley & Sons, 2006.

[91] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[92] Shlomo Weiss and Casimir Kulikowski. *Computer systems that learn*. San Mateo, CA: Morgan Kaufmann, 1991.

[93] JS Urban Hjorth. *Computer intensive statistical methods: Validation, model selection, and bootstrap*. CRC Press, 1993.

[94] Mark Plutowski, Shinichi Sakata, and Halbert White. Cross-validation estimates imse. *training (as training can be faster on smaller datasets)*, 2:4, 1994.