

**Технически Университет – София**

**Факултет Приложна Математика и Информатика**

**Катедра Информатика**

**КУРСОВА РАБОТА ПО ПРИЛОЖЕН ИЗКУСТВЕН ИНТЕЛЕКТ**

**Тема:** Decision trees

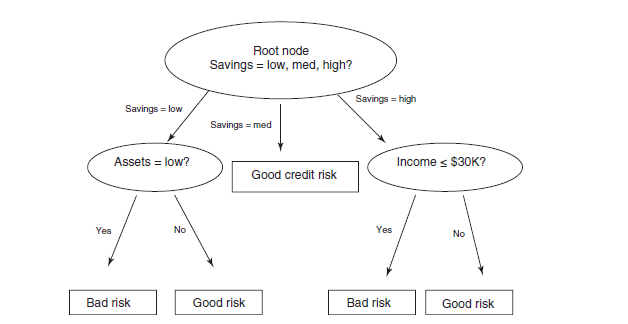
**Имена на студентите:** Тодор Пъков, Георги Донков

**Факултетни номера, групи:** 471218003, 471218026, 78 група

**Съдържание:**

1. Дърво на решенията
   1. Какво представлява дървото на решенията?
   2. Изисквания за използване на дърво на решенията
2. Предимства и недостатъци
3. Метрики за намиране на най-добро разделяне
   1. Gini impurity
   2. Информационна печалба (Information Gain)
   3. Намаляване на дисперсията (Variance reduction)
   4. Мярка за доброта (Measure of “goodness”)
4. Алгоритъм CART (Classification and Regression Tree)
5. Алгоритъм C4.5
6. Разработен софтуер
7. Експерименти
   1. Задача за класификация
   2. Задача за регресия
8. Документация на кода
9. Дърво на решенията
   1. Какво представлява дървото на решенията?

Дървото на решенията е метод за класификация, който представлява конструкция от възли (nodes), които са свързани с клонове (branches). Конструкцията тръгва отгоре надолу, като най – отгоре имаме възел, наречен корен (root node), най – отдолу имаме възли, наречени листа (leaf nodes), а между корена и листата съществуват така наречените decision nodes.



На илюстрацията отгоре е показано дърво на решенията, което оценява кредитния риск, като класифицира даден кандидат с „добър“ или „лош“ кредитен риск в зависимост от няколко прогнозни променливи (predictor variables), а именно спестявания, които биват ниски, средни или високи; активи, които биват ниски или високи и доходи, които се класифицират като по – малко от тридесет хиляди долара или по – големи от тридесет хиляди долара.

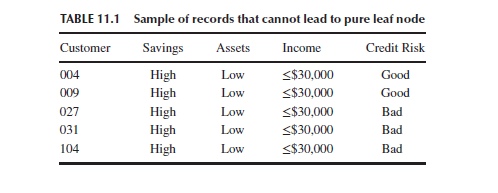
На дадената илюстрация коренът на дървото е всъщност decision node, който проверява дали за даден запис от набора данни (в случая кандидат за кредит) спестяванията са ниски, средни или високи. Според стойността на дадената прогнозна променлива се тръгва по някой от клоните на дървото.

Ако спестяванията са ниски, то ще тръгнем по най – левия клон на дървото, който води до друг decision node, в който се проверява дали активите са ниски. Ако са ниски – то кандидатът за кредит се класифицира с добър кредитен риск, а ако са високи – с лош кредитен риск (тук достигаме до листата на дървото и приключваме обхождането му).

Ако спестяванията са високи, то тръгваме по най – десния клон, който води до друг decision node, където се проверява дали приходите на кандидата са повече или по – малко от тридесет хиляди долара. Ако са повече, то кандидатът се класифицира с добър кредитен риск, а ако са по – малко – с лош кредитен риск.

Последно остана да прегледаме случая, когато спестяванията са средни. Тогава ще тръгнем по средния клон на дървото, който ни свързва директно с едно от листата на дърветата, тоест кандидатите със средни спестявания директно се класифицират с добър кредитен риск, без да се проверяват други прогнозни променливи.

Все пак има случаи, в които чрез даден decision node не можем да достигнем до „чисто“ листо на дървото. Нека вземем за пример следния откъс от данни:



Ако тръгнем от корена на дървото, то ясно се вижда, че за всеки един от кандидатите ще тръгнем по най – десния клон, защото спестяванията на всички за високи. Най – десния клон ни свързва с decision node-а, който проверява дали доходите са по – малки от тридесет хиляди долара.

Имаме два случая, в които доходите са по – малки от тридесет хиляди, но кандидатите в набора от данни са класифицирани с добър кредитен риск. Останалите трима кандидати са класифицирани с лош кредитен риск.

Все пак в случая класифицираните с лош кредитен риск преобладават и следователно можем да достигнем до листо на дървото, което да класифицира кандидатите за кредит с доходи по – малки от тридесет хиляди долара с лош кредитен риск.

Проблемът е, че това развитие ще има 60% успех. В случая можем да се опитаме да разклоним още повече дървото, като първо проверим доходите на кандидатите, а след това добавим още decision nodes, които да проверяват спестяванията или акциите от най – дясната страна на дървото.

Наблюдавайки данните за кандидатите виждаме, че всички от тях имат еднаква стойност за спестяванията и акциите, т.е. няма смисъл от това да правим разклонения. В такъв случай листото на дървото, до което сме достигнали се нарича diverse leaf.

* 1. Изисквания за използване на дърво на решенията

Следните изисквания трябва да бъдат спазени преди използването на алгоритъм за изграждане на дърво на решенията:

1. Алгоритмите, които изграждат дървета на решенията, изискват предварително класифицирани променливи. Трябва да предоставим набор от данни, чрез който ще обучим алгоритъма, предоставяйки стойностите на класифицираните променливи.
2. Наборът от данни трябва да е богат и да предоставя на алгоритъма множество комбинации от различни стойности за класифицираните променливи.
3. Целевите променливи трябва да приемат стойности, които могат да се определят като принадлежащи или непринадлежащи на даден клас.

Важен е въпросът защо например в по – горния пример за корен на дървото е избрана проверката на атрибута спестявания. Отговорът се крие в това, че дървото на решенията се изгражда по начин, по който листата му ще са най „чисти“. Това се случва чрез измерване на чистотата на листата (leaf node purity) с различни алгоритми.

1. Предимства и недостатъци
   1. Предимства

* Лесни са за разбиране тълкуване. Хората могат да разберат моделите на дървета на решенията след кратко обяснение.
* Полезни са дори при малко количество данни. Могат да се направят важни изводи
* Помагат за определяне на най-лошите, най-добрите и очакваните стойности за различни сценарии.
* Могат да се комбинират с други техники за взимане на решения.
  1. Недостатъци
* Те са нестабилни, което означава, че малка промяна в данните може да доведе до голяма промяна в структурата на оптималното дърво на решенията.
* Често са относително неточни. Много други предиктори се представят по-добре при сходни данни. Това може да се коригира чрез замяна на едно дърво на решенията със случайна гора от дървета на решенията (random forest), но случайната гора не е толкова лесна за интерпретиране, колкото едно дърво на решенията.
* За данни, включващи категорийни променливи с различен брой нива, информационната печалба в дърветата за вземане на решения клони към тези атрибути с повече нива.
* Изчисленията могат да станат много сложни, особено ако много стойности са несигурни и/или ако много резултати са свързани.

1. Метрики за намиране на най-добро разделяне

Алгоритмите за конструиране на дърво на решенията обикновено от горе надолу, като на всяка стъпка се избира променливата, която най-добре разделя множеството от данни. Различните алгоритми имат специфични подходи за намирането на това „най-добро“ разделяне. Обикновено се измерва хомогенността на целевата променлива в отделните подмножества. Изчисленията се прилагат към всяко кандидат-множество и резултатите се комбинират, като така осигуряват информация за качеството на съответното разделяне.

* 1. Gini impurity

Използва се от алгоритъма CART и измерва колко често случайно избран елемент от множеството ще бъде грешно определен (класифициран), ако е бил случайно класифициран спрямо разпределението на класовете в подмножеството. Може да се изчисли като сумират вероятностите даден запис от клас да бъде избран, умножени по вероятността за грешка при класифицирането. Функцията достига своя минимум – нула, когато всичките записи във възела принадлежат на един клас.

За изчисление на ф-ята за множество от елементи с J класа и – пропорцията от елементи, принадлежащи към клас , може да се използва формулата:

D:\Documents\ПИИ\{_displaystyle _operatorname {I} _{G}(p)=_sum _{i=1}^{J}_left(p_{i}_sum _{k_neq i}p_{k}_right)=_sum _{i=1}^{J}p_{i}(1-p_{i})=_sum _{i=1}^{J}(p_{i}-{p_{i}}^{2})=_sum _{i=1}^{J}p_{i}-_sum _{i=1}^{J}{p_{i}}^{2}=1-_sum _{i=1}^{J}{p_{.png

* 1. Информационна печалба (Information Gain)

Използва се при алгоритмите ID3, C4.5 и C5.0. Основава се на концепциите за ентропия и информационно съдържание от теорията на информацията. Разгледана е по-подробно в частта за C4.5.

* 1. Намаляване на дисперсията (Variance reduction)

Имплементирана е за първи път в CART. Обикновено се използва когато целевата променлива е непрекъсната (регресионно дърво), тъй като използването на други метрики би изисквало дискретизация на данните. Намаляването на дисперсията в даден връх N се дефинира като общото намаляване на дисперсията на целевата променлива, породено от извършване на разделяне при N.

* 1. Мярка за „доброта“ (Measure of “goodness”)

Използвана е в CART през 1984 г. Това е функция, имаща за цел да оптимизира баланса между способността на възможните разделяния да създават чисти деца (с елементи от един клас) с възможността им да създават деца с еднакъв размер (брой записи). Процесът се повтаря за всеки „нечист“ възел до създаване на цялото дърво. Самата ф-я , където s – едно възможно разделяне при възел t, изглежда така:

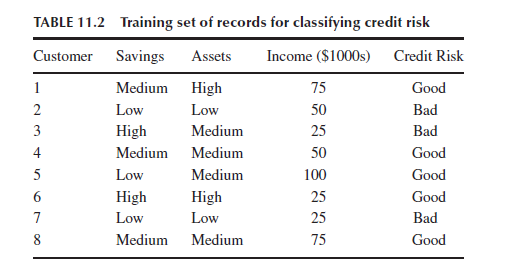
Във формулата и са съответно левия и десния детски възел на t, в резултат от разделяне s, и са пропорциите на елементите от t в и , а и са пропорциите на елементите от клас j в и .

Функцията се увеличава, когато и двата й компонента - и също нарастват. Вторият ще има голяма стойност, когато разстоянията между и са максимални за всеки клас или иначе казано, когато пропорциите на елементите в дъщерните възли за всеки възможен клас са колкото се може по-различни. Теоретичният максимум на тази част от е равен на самия брой класове, налични в данните. Другият компонент - достига максимум, когато пропорциите на елементите в левия и десния дъщерен възел са равни ().

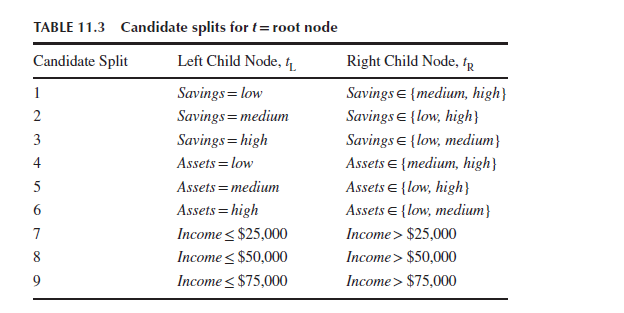
От казаното дотук може да се заключи, че дърво, използващо ф-ята , ще предпочита разделяния, при които дъщерните възли са хомогенни спрямо всички класове и имат приблизително еднакъв брой елементи.

1. Алгоритъм CART (Classification and Regression Tree)

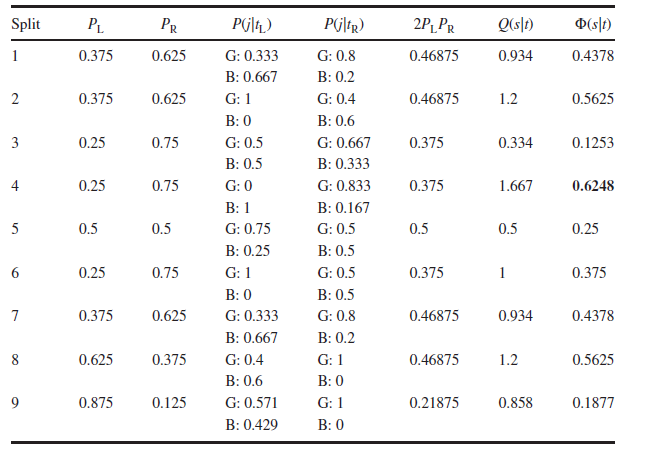
Методът на класификационните и регресионни дървета CART е предложен през 1984 г. от Л. Бреймън, Дж. Фрийдмън, Р. Олшен и С. Стоун. Съществуват различни варианти на CART, напр. CHAID, ExhaustiveCHAID и др. Като регресионна техника CART методът се определя като рекурсивно-разделяща регресия. Целта е разделяне на данните в относително хомогенни (с ниско стандартно отклонение или с минимална обща грешка по метода на най-малките квадрати) крайни възли и получаване на средна наблюдавана стойност при всеки краен възел като прогнозна стойност. За намиране на най – доброто разделяне в дървото на решенията, което се изгражда от CART алгоритъма, можем да използваме мярката за доброта (measure of goodness) или gini impurity, които са обяснени подробно по – нагоре в текущия документ.

Нека разгледаме един пример как чрез CART алгоритъма можем да изградим дърво на решенията при даден набор от данни използвайки метриката за намиране на най – добро разделяне мярка за доброта (measure of goodness). Нека наборът от данни е следният:

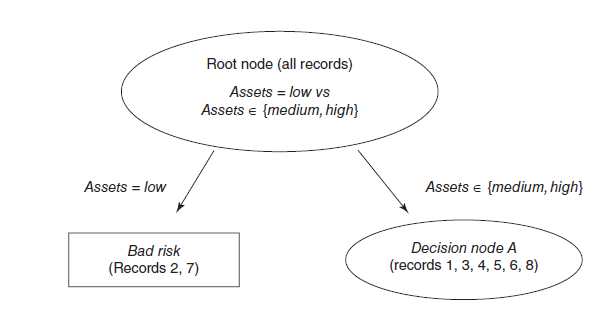
Първата стъпка, която трябва да се направи, е да се реши коя от трите променливи Savings, Assets и Income ще стои в корена на дървото. Нека представим набора от данни със всички възможни разделения:



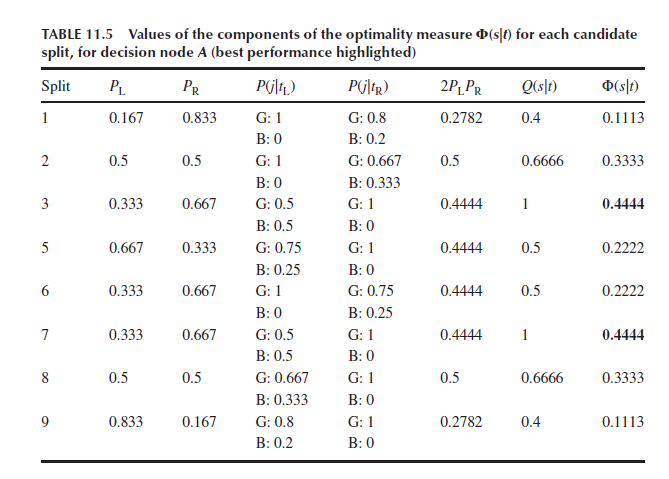
Получихме 9 възможни разделения. Използвайки формулите от метриката мярка за доброта можем да решим с коя от променливите можем да започнем дървото, т.е. коя от променливите ще стои в нашия корен и съответно кои стойности на нашата променлива ще лежат отляво и отдясно на корена. Изчислявайки функцията ще получим следните резултати:



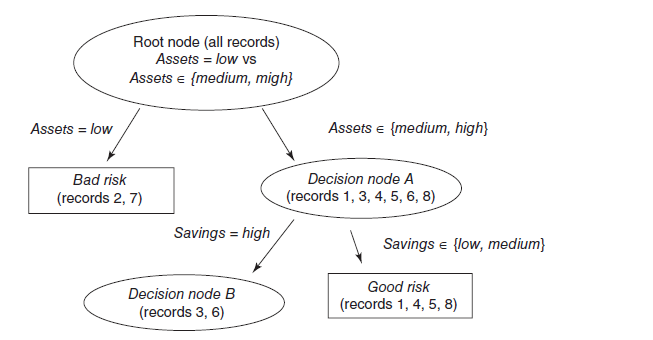
Най – голяма стойност за функцията получихме в четвъртия кандидат, т.е. променливата Assets ще стои в корена на дървото, отдясно на корена ще имаме Assets=low, а отляво на корена ще имаме Assets=medium/high. До тук дървото ще изглежда по следния начин:



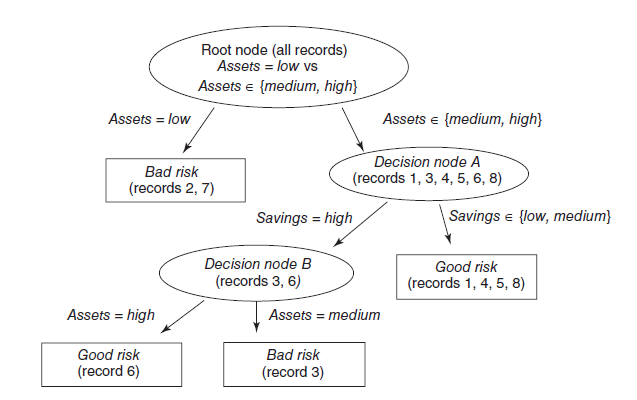
В лявата част на корена получаваме leaf node (листо) с резултат лош риск, защото всички записи в набора от данни с Assets=low са класифицирани с лош кредитен риск (bad risk), тоест кандидати 2 и 7. От дясната страна на дървото остават кандидатите за разделяне 1,2,3,5,6,7,8,9 (всички, без избрания преди малко кандидат). Сега отново, използвайки формулата за изчисления на може да бъде взето решение как да се раздели новополучилия се десен node. Ето и изчисленията:

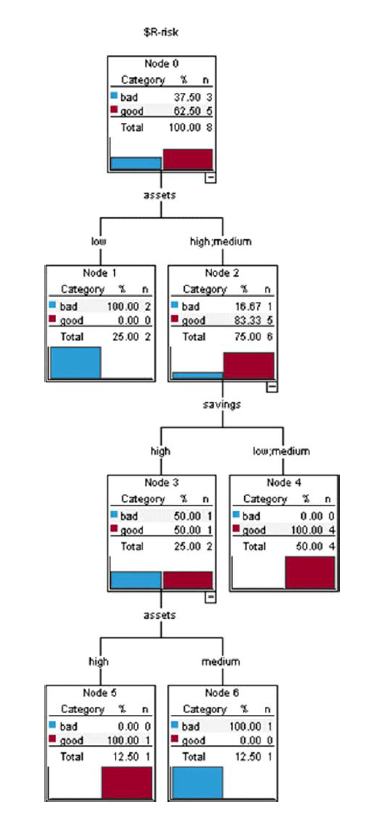


В случая получихме две равни най – големи стойности, а именно за кандидатите 3 и 7. В такъв случай алгоритъмът взима първия кандидат, за който е изчислена функцията, тоест кандидат 3. Ако се върнем малко по – нагоре до таблица 11.3 можем да видим, че избирайки кандидат 3, то в decision node A на дървото ще стои променливата Savings, левият branch ще бъде Savings=high, а десният Savings=low, medium. Дървото ще изглежда по следния начин:



От таблица 11.2 можем да видим, че комбинирайки Assets=medium/high и Savings=low/medium (кандидати 1,4,5,8) получаваме класификация само с добър кредитен риск, в дясно ще имаме leaf node с решение Good risk. Отляво остава да се вземе решение кой кандидат да се постави в Decision node B измежду останалите кандидати 3 и 6. Комбинирайки кандидатите 3 и 6, тоест Savings=High и Assets=Medium/High виждаме, че вземайки кандидат 3 получаваме лош кредитен риск, а за кандидат 6 – добър кредитен риск. Завършеното дърво ще изглежда така:





Ето някои приложения на CART методите:

* Сегментиране: Позволява автоматично разпределяне на наблюденията по групи, например разделяне на клиентите на фирма на групи по най-съществените им предпочитания.
* Класификация в степенувани групи: Разделяне на наблюденията в категории чрез степенуване, напр. висока категория, средна, ниска
* Предсказване: Създаване на правила и използването им за предсказване на бъдещи събития, напр. доколко някой се очаква да изпадне в неплатежоспособност
* Намаляване на размерността на задачата (броя на променливите и/или случаите): Подбор на подмножество от предиктори като най-полезни за анализа и създаване на модел, описващ достатъчно добре целия обем от данни.
* Разкриване на взаимодействия: Отделяне на специфични подгрупи в извадката, които си взаимодействат.
* Сливане на категории и дискретизиране на непрекъснати променливи: Преподреждане на групови предиктори и непрекъснати променливи с минимална загуба на информация

1. Алгоритъм C4.5

C4.5 e един от алгоритмите, използвани за генерирани на т.нар. дърво на решенията. Разработен е от Ross Quinlan и е разширение на неговия по-ранен алгоритъм ID3.

Алгоритъмът рекурсивно обхожда всеки възел и избира оптималното разделяне, докато това е възможно. За разлика от други алгоритми като CART, C4.5 не е ограничен до бинарни разделяния и съответно може да създава дървета с по-голям брой разклонения. Когато разглежданата променлива е категорийна, C4.5 по-подразбиране създава разклонение за всяка нейна стойност. Това не винаги е удачно, тъй като някои от стойностите може да се срещат по-рядко или пък са естествено свързани с други такива.

C4.5 използва концепцията **за информационна печалба** или **намаляване на ентропията**, за да се избере оптималното разделяне. Първо се изчислява най-малкият брой битове средно за символ, необходими за предаването на поток от символи, представящи стойностите на дадена променлива X. За целта се използва формулата:

е вероятността на срещане на j-тата стойност на X. Резултатът се нарича ентропия на X. Формулата се базира на факта, че за събитие с вероятност p средното количество информационни битове за предаване на резултата е . Например резултатът от хвърляне на монета (честна) с вероятност 0.5, може да се предаде, като се използва (0 или 1 в зависимост от резултата). Когато възможните изходи са повече от два, се търси претеглената сума на отделните логаритми, като теглата са равни на вероятностите за появата на съответните изходи.

Концепцията за ентропия се използва по следния начин. Да предположим, че разделяне S, дели тренировъчните данни T на няколко подмножества. Средното информационно изискване може да се изчисли като претеглената сума от ентропиите на отделните подмножества:

Тук e пропорцията на записите, попаднали в подмножество . По този начин информационната печалба може да се дефинира като , тоест увеличението на информацията в резултат от разделянето на тренировъчните данни T спрямо S. Във всеки възел, при който става разклонение, C4.5 избира разделянето S, което води до най-голямата информационна печалба .

Алгоритъмът C4.5 може да бъде представен чрез следната последователност от стъпки:

1. *Проверка дали са удовлетворени условията за край.*
2. *Изчисляване на информационната печалба за всички атрибути.*
3. *Избиране на атрибута, чието разделяне води до най-голямата печалба.*
4. *Създаване на decision node спрямо атрибута, избран в 3.*
5. *Разделяне на множеството от данни спрямо decision node-a от 4.*
6. *Повторно (рекурсивно) извикване на алгоритъма за всички подмножества, получени от разделянето в 5.*
7. *Прибавяне на новите разклонения на дървото, получено от 6. към decision node-а от 4.*
8. *Връщане на дървото.*

Примерен псевдокод:



1. Софтуерна разработка

Разработеният по време на курса софтуер е достъпен в [GitHub](https://github.com/trpakov/DecisionTree). Представлява вариант на C4.5 алгоритъма, имплементиран с езика Python. Някои от основните характеристики на алгоритъма са следните:

* Поддръжка както на категорийни, така и числови атрибути.
* Изграждане на класифициращи и регресионни дървета в зависимост от вида на целевата променлива.
* Използване на Информационна печалба (Information Gain) за определяне на най-доброто разделяне.
* Отриване на оптималния binning за даден атрибут.
* Възможност за ограничаване на разделянията до две подмножества (бинарни дървета).
* Окастряне (pruning) на генерираното дърво до даден брой листа.
* Класифициране на тестови данни на базата на дървото, генерирано от тренировъчните данни.
* Допълнително методи за подготовка на данните (data preparation).
* Множество хиперпараметри, достъпни за промяна от страна на ползвателя на софтуера (минимален брой записи в листо, дали да е възможно повторно разделяне по даден атрибут, дали да се използва binning при числовите атрибути, нов брой на листата след окастряне, запис на дървото във файл, промяна на отстоянието между отделните нива с цел по-добра визуализация, …)

1. Експерименти
   1. Задача за класификация

При задачите за класификация целевата променлива е категорийна и принадлежи към различни класове. Ще разгледаме пример с [набор от данни](https://web.stanford.edu/class/archive/cs/cs109/cs109.1166/problem12.html) за пасажерите на кораба Титаник, като целта е на база на тази информация да определим дали съответния пътник е оцелял.

Използваме следните атрибути:

* PClass (Passenger class) – Класата, в която пътува (discrete)
* Sex - пол (discrete)
* Age – възраст (continuous)
* Siblings/Spouses Aboard – брой роднини на борда (multi-valued discrete)
* Parents/Children Aboard – брой родители/деца на борда (multi-valued discrete)
* Fare – платена цена за билет (в британски лири) (continuous)
* Survived – дали са оцелели (1 – да, 0 – не) (discrete)

Генерираме две модела при следните параметри:

1. минимален брой записи в листо = 1; повтаряне на атрибути = ДА
2. минимален брой записи в листо = 10; повтаряне на атрибути = ДА; окастряне (pruning) = ДА (макс. брой листа = 25)

Нека обърнем внимание на представянето на двата модела. При модел a):

Training data performance: 95.19%

Test data performance: 81.92%

Грешката при тренировъчните данни доста по малка в сравнение с тази при тестовите (4.81% срещу 18.08%). Това е резултат от малкия брой записи в листо (1). Така получаваме голямо дърво с висока успеваемост относно тренировъчните данни, но това оказва негативно влияние върху представянето му при работа с непознати данни. Тази зависимост е характерна за този тип модели. Нарастването на дървото води до намаляване на тренировъчната грешка, но и шумът в данните оказва все по силно влияние и съответно намалява успеваемостта при тестовите данни. Това е т.нар. overfitting, правещ модела недостатъчно добър.

За да го подобрим можем да предприемем няколко стъпки – да увеличим минималния брой записи в листата, както и да извършим окастряне (pruning) до определен брой листа. Така получаваме модела от b):



Неговото представяне е по-различно:

Training data performance: 81.92%

Test data performance: 91.17%

Тук грешката при тренировъчните данни се увеличава до 18.08%, но същевременно тази при тестовите намалява до 8.83%, което е и нашата цел. Така достигнахме до модел, имащ по-добро представяне в реални ситуации – при използване на непознати данни.

* 1. Задача за регресия

В този случай целевата променлива, която се опитваме да прогнозираме, е непрекъсната. За набор от данни се използва [Auto MPG Data Set](http://archive.ics.uci.edu/ml/datasets/Auto+MPG), съдържащ информация за превозни средства с цел определяне на разхода на гориво в градска среда (в мерни единици mpg – miles per gallon).

Информация за използваните атрибути от набора от данни:

* cylinders – брой цилиндри на двигателя (multi-valued discrete)
* displacement – работен обем на двигателя (continuous)
* horsepower – конски сили (continuous)
* weight – тегло (continuous)
* acceleration – ускорение (continuous)
* model year – моделна година (multi-valued discrete)
* origin – произход (multi-valued discrete)
* mpg – miles per gallon (continuous)

За целите на експеримента данните се разделят на две части:

* тренировъчни (training data) – 90% от записите, използват се при генерирането на дървото.
* тестови (test data) – 10% от записите, използват се при оценката на избрания модел.

Когато целевата променлива е непрекъсната, за определяне на най-подходящия атрибут, спрямо който да се извърши дадено разделяне, се изчислява т.нар. намаляване на стандартното отклонение (standard deviation reduction). Търси се атрибутът, за който тази стойност е най-голямо, което води и до създаването на най-хомогенните клони.

Разглеждаме два модела (дървета), генерирани при следните параметри:

1. минимален брой записи в листо = 10; повтаряне на атрибути = НЕ



1. минимален брой записи в листо = 4; повтаряне на атрибути = ДА; окастряне (pruning) = ДА (макс. брой листа = 30)



За оценка на предложените модели използваме следните метрики:

* Explained variance score (обяснена вариация) – измерва степента, в която моделът отчита дисперсията на даден набор от данни. Най-добрата възможна стойност е 1.0.
* Max error (максимална грешка) – показва най-голямата грешка (разлика) между прогнозираните и истинските стойности.
* Mean absolute error (средна абсолютна грешка)
* Mean squared error (средна квадратична грешка)
* Mean squared logarithmic error (MSLE) – вариация на средната абсолютна грешка, измерваща отношението между реалните и прогнозните стойности. Наказва по-силно занижената прогнозна оценка, отколкото завишената прогнозна оценка.
* Mean absolute percentage error (средна абсолютна процентна грешка) – чувствителна е към относителни грешки, не се влияе от скалиране на целевата променлива
* Median absolute error – устойчива е на отклонения (outliers). Загубата се изчислява, като се вземе медианата на всички абсолютни разлики между целевата променлива и прогнозата.
* R² score (коефициент на детерминация) – представлява отношение на обяснената към общата вариация на зависимата променлива. Показва каква част от изменението (вариацията) на резултатната променлива Y се обяснява с факторите в регресионния модел. Добра индикация за това колко успешно могат да бъдат прогнозирани непознати извадки.

Изчисляването на метриките за всеки един от моделите става с помощта на модула sklearn.metrics от scikit-learn. Резултатите са следните:



И двата модела ясно показват основните зависимости между атрибутите и целевата променлива. Прогнозата е превозните средства с най-ниско тегло да имат най-добрия разход на гориво (mpg). Обратното се предполага за най-тежките автомобили. Там разходът е най-голям. Именно такъв резултат и очакваме. Другите атрибути , оказващи голямо влияние върху разхода на гориво са конските сили, както и работния обем на двигателя, което също не е изненада.

1. Документация на кода

В тази част е поместено подробно описание на разработения софтуер (на английски език):

**DecisionTree**

DecisionTree is a variant of C4.5 decision tree algorithm implemented in Python. Developed as an assignment in Applied Artificial Intelligence course.

**Features**

* Supports both categorical and numerical attributes.
* Uses information gain to determine the best split.
* Utilizes [OptBinning](https://github.com/guillermo-navas-palencia/optbinning) to find optimal binning for given attribute.
* Prunes the tree until leaf nodes are fewer than specified threshold.
* Classifies test data based on tree generated from training data.
* Provides additional methods for data preparation.

**Installation**

Clone git repository:

git clone https://github.com/trpakov/DecisionTree.git

Use [pip](https://pip.pypa.io/en/stable/) to install required dependencies:

pip install -r requirements.txt

**Example usage**

import DecisionTree as dt

import pandas as pd

**Tree generaion**

# import training data from csv file

data = pd.read\_csv("data/titanic2.csv")

# set column type as appropriate

data = data.astype({'Pclass':'category', 'Siblings/Spouses Aboard':'category', 'Parents/Children Aboard':'category'}, copy=False)

# create instance

dtg = dt.DecisionTreeGenerator(data)

# generate tree

dtg.generate()

**Tree pruning**

# prune tree so the leaf nodes are no more than 10

dtg3.prune(10)

========== Sex ==========

Sex ['male'] [size = 573]:

========== Age ==========

Age ∈ (-inf, 13.0) [size = 41]:

========== Siblings/Spouses Aboard ==========

Siblings/Spouses Aboard [3, 5, 8, 4] [size = 17]: 0

Siblings/Spouses Aboard [0] [size = 7]: 1

Siblings/Spouses Aboard [1, 2] [size = 17]: 1

Age ∈ (13.0, 20.25) [size = 89]: 0

Age ∈ (20.25, 22.5) [size = 46]: 0

Age ∈ (22.5, inf) [size = 397]: 0

Sex ['female'] [size = 314]:

========== Pclass ==========

Pclass [3] [size = 143]: 0

Pclass [2] [size = 76]: 1

Pclass [1, 4] [size = 95]: 1

**Saving tree to file**

# print generated tree to file

with open('data/titanic.txt', mode='w', encoding='utf-8') as f:

dtg3.treeRoot.print(file=f)

**Classifying new data**

# import test data from csv file

testData = pd.read\_csv("data/TitanicTest.csv")

# classify test data using generated tree

dtg.classify(testData)

# save results to csv file

testData.to\_csv('data/TitanicResult.csv', index=False)

**Tree building algorithm outline**

1. Check if termination criteria are satisfied.
2. Compute highest information gain for each attribute.
3. Choose the attribute whose split produces the best gain.
4. Create a decision node based on attribute chosen in step 3.
5. Split the dataset based on node from step 4.
6. Recursively call the algorithm for all subsets resulting from the split in step 5.
7. Attach new branches of the tree from step 6 to decision node from step 4.
8. Return generated tree.

**Pseudocode**

Tree function GenerateTree(dataset D)

{

Tree = {}

if (D is "pure" OR other stopping criteria met):

terminate

end if

for all attribute a in D do:

Compute information gain if we split on a

end for

a\_best = Best attribute according to above computed criteria

Tree = Create a decision node that splits on a\_best in the root

D\_s = Subsets from D based on a\_best

for all D\_s do:

Tree\_s = GenerateTree(D\_s)

Attach Tree\_s to the corresponding branch of Tree

end for

return Tree

}

**Documentation**

**DecisionTreeGenerator(data)**

* Main class accepting input data and generating a decision tree.

**Parameters**

* data (pandas.DataFrame) - Training data for tree generation.

**Properties**

* data (pandas.DataFrame) - Training data for tree generation.
* inputVars (list) - List of attributes.
* classes (set) - Set of classes (possible values of the target variable).
* treeRoot (Node) - Root node of the tree.
* numOfLeafNodes (int) - Total number of leaf nodes.
* twigs (dict(Node)) - Dictionary with twig nodes (used as keys) whose children are only leaf nodes and their information gain (used as values).
* numericAttrRanges (dict) - Dictionary with previous intervals for repeating numeric attributes. Used when repeatAttributes is True

**Methods**

* checkIfOnlyOneClass(data) - Checks if all entries in given dataframe belong to the same class.

**Parameters**

* data (pandas.DataFrame) - Dataframe to check.

** Returns**

* (Bool, className) - Tuple with first element True and second element - class name if only one class, otherwise False and None.

 getNumberOfRecordsInEachClass(data) - Finds and returns the number of records belonging to each class.

**Parameters**

* data (pandas.DataFrame) - Dataframe to use.

** Returns**

* list((a,b)) - List of tuples (a, b) where a - class name (value of target variable), b - number of entries in data with this class.

 getClassWithMostRecords(data) - Finds and returns the class that corresponds to most of the records.

**Parameters**

* data (pandas.DataFrame) - Dataframe to use.

** Returns**

* className (str) - Name of the class with most records in the dataframe.

 getNumberOfRecordsInClass(data, cls) - Finds and returns the number of entries in the dataframe, belonging to the specified class name.

**Parameters**

* data (pandas.DataFrame) - Dataframe to use.
* cls (str) - Class name to use.

** Returns**

* result (int) - Number of records in data which have class cls.

 calculateEntropy(data) - Calculates the information entropy for given dataframe.

**Parameters**

* data (pandas.DataFrame) - Dataframe to use.

** Returns**

* entropy (float) - Information entropy for data.

 calculateInformationGain(data, dataSubsets) - Calculates the gain information entropy after splitting given dataframe into subsets.

**Parameters**

* data (pandas.DataFrame) - Dataframe to use.
* dataSubsets list((pandas.DataFrame)) - Dataframes resulting from a split.

** Returns**

* informationGain (float) - Increase in information produced by partitioning data into dataSubsets.

 calculateStandardDeviationReduction(data, dataSubsets) - Calculates the standard deviation reduction based on the decrease in standard deviation after a dataset is split on an attribute.

**Parameters**

* data (pandas.DataFrame) - Dataframe to use.
* dataSubsets list((pandas.DataFrame)) - Dataframes resulting from a split.

** Returns**

* stdReduction (float) - Difference between standard deviation for values in data and weighted standard deviation for values in dataSubsets.

 calculateMeasureOfGoodness(data, dataSubsets) - Calculates a value that seeks to optimize the balance of a candidate split's capacity to create pure children with its capacity to create equally-sized children.

**Parameters**

* data (pandas.DataFrame) - Dataframe to use.
* dataSubsets list((pandas.DataFrame)) - Dataframes resulting from a split.

** Returns**

* measureOfGoodness (float) - Calculated value for input data split into dataSubsets.

 checkForCategoricalData(threshold) - Heuristic method that checks if the ratio between the number of unique values and the total number of values for every attribute is lower than a user-defined threshold, if so - changes the type of the dataframe column to "category".

**Parameters**

* threshold (float) (default=0.05) - Threshold to use. Column type is changed if the ratio between the number of unique values and the total number of values for every attribute is lower than this value.

** Returns**

* None

 splitData(data, availableAttributes, numericAttrBinning, repeatAttributes, minNumRecordsLeafNode) - Given a list of available attributes chooses a split that has the largest information gain.

**Parameters**

* data (pandas.DataFrame) - Dataframe to use.
* availableAttributes (list) - List of attribute names considered for splitting on.
* numericAttrBinning (bool) - If True - numeric attributes are split using binning, if False - binary split is performed.
* repeatAttributes (bool) - If True - multiple splits can be performed over the same attribute, if False - once an attribute is part of a split it is no longer considered in further splits.
* minNumRecordsLeafNode (int) - Minimum number of records in a leaf node. A split is ignored if there is a subset with lower number of elements than this value.

** Returns**

* splitResult (splitAttrib, bestSubsets, bestSplitThreshold, bestRanges, bestGain) - Tuple containing information for the split performed:
  + splitAttrib (str) - Name of chosen attribute.
  + bestSubsets (list(pandas.DataFrame)) - List of dataframes after splitting data.
  + bestSplitThreshold (float) - Number indicating the split point of a binary split.
  + bestRanges (list(float, float)) - List with tuples corresponding to the ranges produced after binning.
  + bestGain (float) - Information gain value for the chosen split.

 generate(numericAttrBinning, minNumRecordsLeafNode, repeatAttributes) - Calls generateTree function for the dataframe assigned to the current instance of DecisionTreeGenerator, considering all input variables. Assigns the result to treeRoot attribute of the instance.

**Parameters**

* numericAttrBinning (bool) (default=True) - If True - numeric attributes are split using binning, if False - binary split is performed.
* maxNumRecordsToSkipSplitting (int) (default=30) - Directly return a leaf node if number of records is lower than this value. Prevents overfitting.
* minNumRecordsLeafNode (int) (default=10) - Minimum number of records in a leaf node. A split is ignored if there is a subset with lower number of elements than this value.
* repeatAttributes (bool) (default=False) - If True - multiple splits can be performed over the same attribute, if False - once an attribute is part of a split it is no longer considered in further splits.

** Returns**

* None

 generateTree(data, availableAttributes, numericAttrBinning, dataRange, minNumRecordsLeafNode, repeatAttributes) - Called by generate function. Generates the tree by recursively calling itself, assigning child nodes to decision nodes and returning leaf nodes when stopping criteria are met.

**Parameters**

* data (pandas.DataFrame) - Dataframe to use.
* availableAttributes (list) - List of attribute names considered for splitting on.
* numericAttrBinning (bool) - If True - numeric attributes are split using binning, if False - binary split is performed.
* dataRange ((float, float) (default=None)) - Tuple corresponding to the range of values of the current node produced after binning.
* maxNumRecordsToSkipSplitting (int) - Directly return a leaf node if number of records is lower than this value. Prevents overfitting.
* minNumRecordsLeafNode (int) (default=10) - Minimum number of records in a leaf node. A split is ignored if there is a subset with lower number of elements than this value.
* repeatAttributes (bool) (default=False) - If True - multiple splits can be performed over the same attribute, if False - once an attribute is part of a split it is no longer considered in further splits.

** Returns**

* node (Node) - Leaf node if stopping criterium is met, otherwise - decision node.

 generateTwigsDict(node) - Traverses a tree and fills in twigs (dict(Node)) class property with all current twigs (nodes whose children are all leaf nodes).

**Parameters**

* node (Node) (default=None) - If None - clears the dictionary and starts from treeRoot node.

** Returns**

* None

 prune(maxNumOfLeafNodes) - Prunes out portions of the tree that result in the least information gain. Continues while number of leaf nodes is more than desired.

**Parameters**

* maxNumOfLeafNodes (int) - Pruning stops if number of leaf nodes becomes less than or equal to this value.

** Returns**

* None

 traverseTree(row, node) - Recursively traverses a structure of Nodes using values in dataframe row for navigation. Returns the class corresponding to the leaf node that was reached.

**Parameters**

* row (pandas.Series) - Row from dataframe to use.
* node (Node) (default=None) - - If None - starts from treeRoot node.

** Returns**

* node.name (str) - The class corresponding to the leaf node that was reached.

 classify(data) - Classifies records in given dataset using already constructed tree.

**Parameters**

* data (pandas.DataFrame) - Dataframe to use.
* **Returns**
  + None

**Node(name, threshold, isLeafNode, data, dataRange, gain)**

* Class containing information for a single tree node.

**Parameters**

* name (str) - Name of splitting attibute for decision nodes, class (value of target variable) for leaf nodes.
* threshold (float) - Used only if binary split of numeric attribute is performed.
* isLeafNode (bool) - True - leaf node, False - decision node.
* data (pandas.DataFrame) - Subset of training data.
* dataRange ((float, float)) (default=None) - Used only if binning of numeric attribute is performed.
* gain (float) (default=None) - Assigned only to decision nodes. Information gain of the node. Used for pruning.

**Properties**

* name (str) - Name of the node.
* threshold (float) - Number indicating the split point of a numeric binary split in the node.
* isLeafNode (bool) - Signifies whether the node is a leaf.
* data (pandas.DataFrame) - Records from the initial data that are part of the node.
* dataRange ((float, float)) - Tuple corresponding to the range of values for data in the node.
* gain (float) - Information gain of the node. Used for pruning.
* childNodes (list(Node)) - List of child nodes resulting from node splitting.

**Methods**

* print(indentation, numOfTabsBetweenLevels, file) - Traverses all child nodes of current node and recursively prints info about them in a tree-like fashion. Printing pattern depends on type of attribute (categorical or numerical) and type of split (binary or using binning).

**Parameters**

* indentation (str) (default='') - Initial printing indentation.
* numOfTabsBetweenLevels (int) (default=1) - Increasing this number increases the distance between tree levels when printing. If value is less than or equal to 0 no indentation between levels is used.
* file (file object) (default=None) - File object to print the tree to. If None - standart output is used.
* **Returns**
  + None

**DataPrep(data, attributes)**

* Class which methods help with data preparation

**Parameters**

* data (pandas.DataFrame) - Subset of training data.
* attributes (list) - List of attribute names considered for splitting on.

**Methods**

* fixCapitalization(self) - Lowers the cases of all attributes.
* checkForNumericOutliersWithBoxPlot(self) - Creates a box plot which helps you to determine if there are any numeric outliers.
* checkForNumericOutliersWithDescription(self) - Creates desctiptions of all numerical attributes which helps you to determine if there are any numberic outliers.
* checkForCategoricalOutliers(self) - Creates a plot, which helps you to determine if there are any categorical outliers.
* mergeCategoricalOutliers(self, attrName, thresholdPercent) - Merges categorical outliers, classifies them as 'Other'.
* **Parameters**
  + attrName (str) - The name of the attribute with the outliers that you want to merge.
  + thresholdPercent - The records which are encountered under that percentage will be merged.
* createMissingDataHeapMap(self) - Creates a heap map which helps you to determine if there is any missing data.
* checkMissingDataPercentageList(self) - Creates a list which helps you to determine if there is any missing data.
* checkForNonUniqueValue(self) - Checks for attributes with only non-unique values. If any - the attribute is removed.
* checkForDuplicatesAndRemoveIfAny(self) - Checks for duplicates and removes them if any.

**DecisionMaker(data, treeRoot)**

* Second variation of a classifier, which classifies records in given dataset using already constructed tree.

**Parameters**

* data (pandas.DataFrame) - Subset of training data.
* treeRoot (str) - The name of the root of the tree.

**Properties**

* new\_dataSet (pandas.DataFrame) - A new dataset, where the classified records will be saved.
* last\_column\_name (str) - The name of the last attrbitue(the attribute that we expected to be classified by our algorithm).

**Methods**

* DecisionMaking(node) - The method, which makes the classification.

**Parameters**

* node (Node) - You should always pass the treeRoot node as node in this method, because the crawling of the tree always begins with the root of the tree.
* **Properties**
  + branches\_names (list) - List which contains the names of all the branches of a node.
  + isCategorical (bool) - A property which helps us to determine whether an attribute is categorical or numeric.
  + node\_column\_index (int) - A property which holds the index of the node column

The method first checks whether the root of the tree is NOT signed/unsigned integer; floating point; complex floating point, i.e. if the root is categorical. If it is categorical the method adds the branches names of the root in branches\_names. For categorical nodes the branches\_names are simply the records. For example if we have node 'Assets' the branches names will be 'High', 'Low', 'Medium'. If the root is not categorical the method again adds all the branches names in the list branches\_names, but the names for numeric nodes are not as simple as the names of categorical nodes. The names of the numeric nodes can be intervals, like 'Age ∈ (20.25, 22.5)' or just one or two values, for example 'Age [23]' or 'Age [23, 33]'. All the branches names are added in such way in the list so that we can use them as if statements. For example the branch\_name 'Age ∈ (20.25, 22.5)' in the three will be added in the list as '22 > 20.25 and 22 < 22.5', where '22' is the actual 'Age' of the current node. Then after we have the branches names of the current node in the list out method starts iterating through the rows of the dataset. If the current node is categorical we are checking whether the current row value branch is leaf. If it is leaf, we are adding the value of the leaf to the last column on the row (we are classifying the record) and we are calling out method again with the treeRoot for the next row. If it is not leaf we are recursively calling the method with the non-leaf node. If the current node is numeric our method starts iterating through branches names. For each branch name our method replaces node name in branch name with row value, so that we have 'ROW\_VALUE > 20.25 and ROW\_VALUE < 22.5' instead of 'Age > 20.25 and Age < 22.5' for example. The next step in our method is to use the branch name as an if statement. If the if statement passes, i.e. if our ROW\_VALUE is bigger than 20.25 and smaller than 22.5 for example, we are checking whether the child that the branch led us to is a leaf. If it is leaf, we are adding the value of the leaf to the last column on the row (we are classifying the record) and we are calling out method again with the treeRoot for the next row. If it is not leaf we are recursively calling the method with the non-leaf node.