Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра прикладної математики

Кваліфікаційна робота

на тему: «**РОБОТА З ДЕРЕВАМИ УХВАЛЕННЯ РІШЕНЬ В HASKELL**»

Виконав: студент 4-го року навчання,

Освітньої програми «Прикладна математика», 113

Ященко Павло Олександрович

Керівник Проценко В.С._____

кандидат фіз.-мат. наук, доцент

Рецензент _____

(прізвище та ініціали)

Кваліфікаційна робота захищена з оцінкою_____

Секретар ЕК _____

«____» _____ 20____ р.

Київ – 2024

Календарний план виконання кваліфікаційної роботи

Тема: Робота з деревами ухвалення рішеннь в Haskell

Календарний план виконання роботи:

№ п/п	Назва етапу дипломної	Термін виконання етапу	Примітка
1.	Отримання завдання на кваліфікаційну роботу.	31.10.2023	
2.	Дослідження літератури, пов’язаної з деревами прийняття рішень та їх використанням	20.12.2023	
3.	Пошук наукових та практичних робіт, пов’язаних з застосуванням Haskell у роботі з деревами прийняття рішень	30.12.2023	
3.	Вивчення необхідних інструментів Haskell, реалізація програм для закріплення навичок	31.01.2024	
4.	Визначення структури програми, вибір інструментів та бібліотек Haskell для її реалізації	15.02.2024	
5.	Написання основної частини програми	17.03.2024	
6.	Використання програми на реальних(або згенерованих) даних, аналіз результатів, виправлення помилок, допрацювання програми	16.04.2024	
7.	Завершення оформлення роботи	14.05.2024	
8.	Попередній захист роботи	15.05.2024	
8.	Корегування роботи за результатами попереднього захисту	2.06.2024	
10.	Захист кваліфікаційної роботи.	3.06.2024	

Студент __Ященко П.О._____

Керівник ___Проценко В.С._____

“_____”_____

ЗМІСТ

АНОТАЦІЯ

Дерево ухвалення рішень – ієрархічна модель, яка організовує рішення та їхні можливі наслідки у деревовидну структуру. У цій роботі розглянуті принципи їх побудови та представлена можлива реалізація алгоритму ID3 на мові програмування Haskell.

Ключові слова: дерево ухвалення рішень, Haskell

ВСТУП

Галузь машинного навчання, завдяки нечуваній раніше доступності різноманітних даних та сучасним обчислювальним потужностям, продовжує стрімко розвиватись.

Однією з багатьох її задач є класифікація даних, а побудова дерев ухвалення рішень є одним з розповсюджених інструментів її розв'язання.

Перша мова програмування, яка приходить на думку, коли мова йде про вищезгадану галузь(як і про все, що пов'язане з аналізом даних) – це, звісно, Python. Наявність великої кількості спеціалізованих бібліотек, таких як NumPy, Pandas, scikit-learn, TensorFlow, роблять його зручним «мультитулом» для роботи з даними.

Проте, це лише інструмент. Ще одуим інструментом цілком може бути Haskell. Його доволі помітним недоліком є малий вибір бібліотек, через що робота з даними ускладнюється. Але саме як мова програмування, Haskell має і свої переваги. Статична типізація забезпечує виявлення багатьох помилок ще на етапі компіляції, відсутність неврахованих побічних ефектів у функцій надає коду передбачуваність та надійність, «ліниві» обчислення дуже допомагають з маніпуляцією великими структурами даних(а також з паралелізацією обчислень) і т.д..

Мета роботи: реалізувати алгоритм побудови дерев ухвалення рішень за допомогою Haskell, застосувати його та оцінити результати

Об'єкт дослідження: дерева ухвалення рішень

Предмет дослідження: побудова дерев ухвалення рішень за допомогою Haskell

РОЗДІЛ 1

1.1. Дерева ухвалення рішень

Дерево ухвалення рішень – ієрархічна структура даних, у якій вхідні дані розбиваються на підмножини з метою передбачення значення залежної змінної.

Його можна зобразити у вигляді графу $G = (V, E)$, який складається з непорожньої множини вузлів V та множини ребер(«гілок») E, та має наступні властивості[1, с.3]:

- Ребра є впорядкованими парами вершин (v, w), тобто граф є направленим
- Граф є ациклічним
- Існує єдиний вузол, у який не входить жодного ребра(корінь дерева)
- Усі вузли, окрім кореня, мають єдине вхідне ребро
- Існує шлях $(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$ від кореня до кожного з вузлів
- Якщо існує шлях з вузла v до w, $v \neq w$, v називають *батьківським*(або прабатьківським, якщо шлях довший за 1 ребро) вузлом w. Відповідно, w називають *нащадком*. Якщо у вузла відсутні нащадки – він є «*листком*», або кінцевим вузлом. Решта вузлів(окрім кореня) – *внутрішні*.

Дерево будується на основі вхідних даних – таблиці, у якої стовпці відповідають атрибутам, а рядки – їхнім значенням. Окремо виділяють атрибут-класифікатор(classifier attribute), власне, залежну змінну.

Внутрішні вузли являють собою критерії, на відповідність яким перевіряються вхідні дані за обраним атрибутом. А ребра репрезентують результати цієї перевірки. У випадку задачі класифікації, листки відповідають значенням атрибута-класифікатора.

Таким чином, аби передбачити залежну змінну на основі певного ряду значень, потрібно слідувати по ребрам відповідно до результату для кожного атрибуту, починаючи з кореня і закінчуючи листком, який і буде результатом передбачення.

По суті, кожен шлях від кореня до листка – кон'юнкція перевірок атрибутів, а вибір шляху – диз'юнкція цих кон'юнкцій.

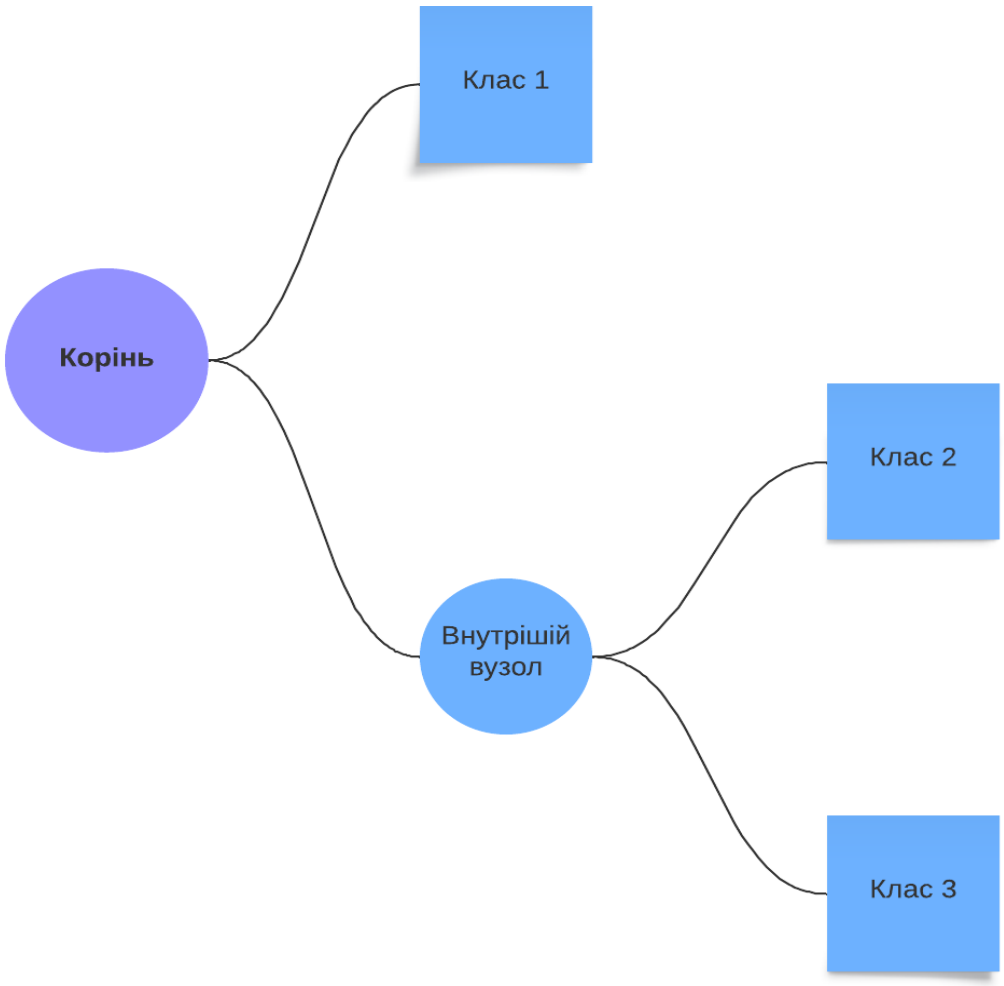
З одних і тих же даних можна отримати безліч різних дерев. У наступному розділі, розглянемо один з можливих підходів для отримання дерев, якомога ближчих до оптимальних.

1.2. Побудова дерева

Найпоширеніші алгоритми побудови дерев рішень належать до сімейства дерев з індукцією «зверху-вниз». До них, зокрема, належать ID3, C4.5, CART, OC1, CHAID та інші. Такі алгоритми рекурсивно розбивають дані на підмножини за атрибутами доки усі вузли на кінцях ребер не виявляться листками, або не буде виконана певна умова зупинки.

Першим з подібних алгоритмів був CLS(Concept Learning System)[2], запропонована Ерлом Гантом. Абстрактно його можна описати таким чином: нехай X_t – множина значень, що відповідають вузлу t та
$y = (y_1, y_2, \dots, y_k)$ – назви класів для задачі з k класами. Тоді:

1. Якщо усі значення в X_t належать до одного класу y_t, то t листок зі значенням y_t
2. Якщо X_t містить значення, що належать до кількох класів, за допомогою функції втрат обирається оптимальне розбиття на підмножини, для кожної з яких створюються відповідні нащадки. Далі елементи X_t розподіляються по нащадках за результатами перевірок на ребрах, що виходять з t і алгоритм рекурсивно застосовується до новостворених вузлів.



Малюнок 1.1. Узагальнений вигляд дерева ухвалення рішень

У подальшому, ця ідея була покращена і розвинута різноманітними способами. Наприклад, ID3 та C4.5 використовують приріст інформації для вибору атрибуту розбиття, CART застосовує коефіцієнт Джині(імовірність того, що навмання обране значення не належить так само випадково обраному класу. Чим ближче дана вірогідність до 0, тим краще розбиття).

Варто зауважити, що такі алгоритми не обов’язково призводять до оптимальних рішень, адже припускають, що з ефективного розбиття на кожному кроці випливає найкращий загальний результат, а це не завжди справджується.

1.3. ID3

Розглянемо детальніше прямого нащадка CLS – Iterative Dichotomiser 3, винайдений Россом Квінланом[3]. Даний алгоритм застосовується для класифікації категорійних даних, серед яких немає відсутніх даних. Розглянемо спосіб, у який даний алгоритм обирає оптимальне розбиття.

Робить він це за допомогою максимізації *приросту інформації,* метрики, заснованої на ентропії.

Ентропія для множини D задається наступним рівнянням:

$$E(D) = -\sum_{i=1}^{m} p_i \cdot log_2(p_i)$$

Де p_i – пропорція елементів D, яка належать класу i, m – кількість класів, а логарифмічна функція за основою 2 використовується через кодування інформації комп'ютерами в бітах. Загалом, ентропія являє собою міру неоднорідності даних. Найменше значення ентропії – 0, у випадку однакових даних, а максимального значення набуває, коли кожна значення є рівноймовірним.

А приріст інформації для атрибуту A відносно множини D:

$$Gain(D, A) = E(D) - \sum_{j=1}^{n} \frac{|D_j|}{|D|} E(D_j)$$

Тут множина розбита на n частин відповідно до значень атрибуту A, D_j – підмножина D, у якої атрибут A має значення j. Приріст інформації працює лише з категорійними даними.

Алгоритм, заснований на вищенаведених принципах можна знайти, наприклад у пакеті aima-haskell[4]

Для подолання вищезгаданого недоліку, Квінсі запропонував покращену версію ID3 – C4.5

1.4. C4.5

Даний алгоритм базується на ID3[5], у якості селектору для вибору найкращого розбиття використовується відношення приросту інформації до $SplitInformation$ – інформації, отриманої від розділення множини D на n підмножин на основі значення атрибуту A

$$GainRatio(D, A) = \frac{Gain(D, A)}{SplitInformation(D, A)}$$

$$SplitInformation(D, A) = -\sum_{j=1}^{n} \frac{|D_j|}{|D|} \cdot log_2\left(\frac{|D_j|}{|D|}\right)$$

Таким чином, алгоритм враховує кількість нових вузлів, які з'являться від виконання розбиття і вибирає кращий з можливих атрибутів.

Також, C4.5 може працювати відсутніми, а також числовими даними.

РОЗДІЛ 2

У цьому розділі наведено реалізацію ID3 на Haskell. Повний код наведено у файлі $Main.hs$ додатку. Тут розглянемо основні моменти.
Для ілюстрації, візьмемо простий датасет fishingData

outlook	*temp*	*humidity*	*wind*	*result*
sunny	hot	high	calm	bad
sunny	hot	high	windy	bad
overcast	hot	high	calm	good
rainy	mild	high	calm	good
rainy	cool	normal	calm	good
rainy	cool	normal	windy	bad
overcast	cool	normal	windy	good
sunny	mild	high	calm	bad
sunny	cool	normal	calm	good
rainy	mild	normal	calm	good
sunny	mild	normal	windy	good
overcast	mild	high	windy	good
overcast	hot	normal	calm	good
rainy	mild	high	windy	bad

Таб. 3.1 fishingData

Маємо відповідні атрибути:

- $outlook = sunny, overcast, rainy$

- $temp = cool, mild, hot$
- $humidity = normal, hight$
- $wind = calm, windy$
- $result = good, bad$

2.2. Представлення дерева:

```
data DecisionTree = MyNull |
                    Leaf AttValue |
                    Node AttName [(AttValue, DecisionTree)]
                  deriving (Eq, Show)
```

Де:

- AttValue – значення атрибуту, AttName – його назва
- Leaf – листок дерева, міститиме AttName залежної змінної
- Node – корінь та внутрішні вузли, міститиме AttName та піддерево, утворене розбиттям за цим атрибутом

А також задамо вигляд розбиття датасету:

```
type Partition = [(AttValue, DataSet)]
```

2.3. Далі реалізуємо сам процес розбиття:

```
partitionData :: DataSet -> Attribute -> Partition
partitionData (title, rows) (attName, attVals)
  = [(attName', (title', rows')) | (attName', rows') <- attNameRows']
  where
    title'    = remove attName title --прибираємо з заголовку
    attVals'  = map (lookUpAtt attName title) rows --знаходимо і зберігаємо значення атрибуту з усіх рядків
```

```
    rows'   = map (removeAtt attName title) rows --прибираємо з рядків
    --оновлюємо прив'язку значеннь атрибутів та рядків
    attNameRows' = foldr addToMapping (zip attVals (repeat [])) (zip attVals' rows')
```

Якщо розбити fishingData за атрибутом outlook, матимемо наступне:

Для значення sunny

temp	***humidity***	***wind***	***result***
hot	high	calm	bad
hot	high	windy	bad
mild	high	calm	bad
cool	normal	calm	good
mild	normal	windy	good

overcast

temp	***humidity***	***wind***	***result***
hot	high	calm	good
cool	normal	windy	good
mild	high	windy	good
hot	normal	calm	good
temp	***humidity***	***wind***	***result***

rainy

temp	***humidity***	***wind***	***result***
mild	high	calm	good
cool	normal	calm	good
cool	normal	windy	bad
mild	normal	calm	good
mild	high	windy	bad

2.4. Вибір атрибуту для розбиття

Задамо наступний шаблон для функції-селектора, аргументами якої буде датасет та атрибут-класифікатор і котра повертатиме найкращий, згідно з результатом її обчислень атрибут:

```
type AttSelector = DataSet -> Attribute -> Attribute
```

Один з варіантів такого селектора – вибір атрибуту на основі найбільшого приросту інформації. Реалізуємо його:

```
bestGainAtt :: AttSelector
bestGainAtt dataset@(title, _) attribute@(attName, _)
  = selectedAtt
  where
    remainingAtts = remove attName title
    gains = map (\att -> gain dataset att attribute) remainingAtts --обчислюємо приріст для решти атрибутів
    maxGain = maximum gains
    selectedAtt = remainingAtts !! fromJust (elemIndex maxGain gains)
```

2.5. Побудова дерева

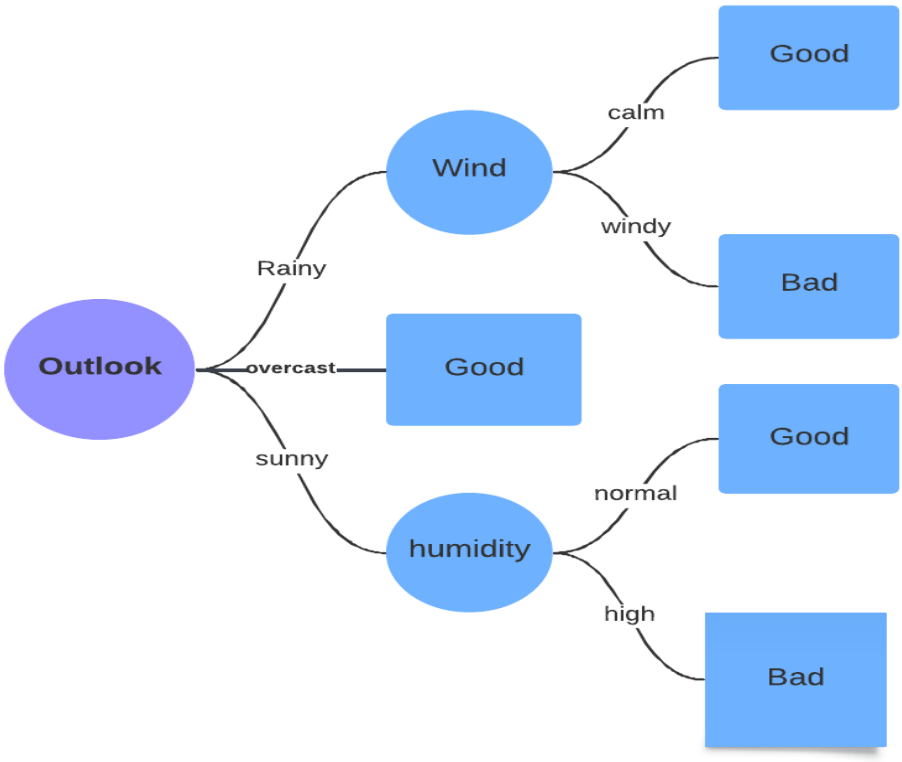
Тепер у нас є все необхідне для цього:

```
buildTree ds@(title, rows) att@(attName, attVals) selector
  -- якщо значення в датасеті для атрибуту однакові додаємо листок
  | allSame attValsInRows = Leaf (head attValsInRows)
  -- якщо ні - додаємо вузол
  | otherwise = Node attName' [(attVal', buildTree ds' att selector) | (attVal', ds') <-
partit]
  where
    -- беремо значення атрибуту з датасету
    attValsInRows = map (lookUpAtt attName title) rows
    -- застосовуємо селектор
    nextAtt@(attName', attVals') = selector ds att
    -- розділяємо датасет за атрибутом
    partit = partitionData ds nextAtt
```

У результаті

```
>>buildTree fishingData result bestGainAtt

"outlook": {
                "sunny":{
                        "humidity": {
                                        "high":"bad",
                                        "normal":"good"}},
                "overcast":"good",
                "rainy":{ "wind": {
                                        "windy":"bad",
                                        "calm":"good"}}}
```



Малюнок 1.2. дерево з fishingData

РОЗДІЛ 3

Тепер, коли основа алгоритму готова, застосуємо його на реальних даних. Для цього знадобиться функція для зчитування даних та способи оцінки отриманих дерев.

3.1 Зчитування даних:

Для отримання даних з CSV файлу використав пакет Data.CSV:

```
parseCsv :: FilePath -> IO [Row]
parseCsv filePath = do
  csvData <- BL.readFile filePath
  case decode NoHeader csvData of
    Left err -> error $ "Помилка читання csv: " ++ err
    Right rows -> return $ map V.toList (V.toList rows)
```

Також, слід перевірити дані на наявність зарезервованих Haskell'ем слів.

3.2. Оцінювання дерев

Відповідно до кількості класів, задачі класифікації поділяються на два типи: бінарну та мультикласову. Наш датасет fishingData з атрибутом-класифікатором $result = good, bad$. Належить до перших. Позначимо good як позитивний результат, P та bad як N. Тоді роботу моделі можна візуалізувати як *матрицю невідповідностей* CM, dim(CM) = (2,2)[Таб. 3.1]

Де

- TP, TN – True Positive та True Negative, кількість правильно передбачених позитивних/негативних
- FP, FN – False Positive, False Nefative – кількість насправді позитивних результатів, неправильно передбачених як негативні і навпаки

Для k класів з атрибутом-класифікатором $y = (y_1, y_2, \dots, y_k)$ вона матиме вигляд [Таб. 3.2]

		Реальні значення	
		P	N
Передбачен значення	P	TP	FP
	N	FN	TN

Таб. 3.1 Матриця невідповідностей, бінарна класифікація

		Реальні значення			
		y_1	y_2	…	y_k
Передбачені значення	y_1	$y_{1,1}$	*FP*	…	$y_{1,k}$
	y_2	*FN*	*TP*	…	*FN*
	…	…	…	…	…
	y_k	$y_{k,1}$	*FP*	…	$y_{k,k}$

Таб. 3.2 Матриця для k класів

Елемент матриці невідповідності для рядку i та стовпця j є кількість хибних передбачень класу і як клас j.

Побудову матриці можна реалізувати наступним чином:

```
type ConfusionMatrix = [[Int]]

computeConfusionMatrix :: DataSet -> DecisionTree -> Attribute -> ConfusionMatrix
computeConfusionMatrix (title, rows) tree classifier =
  let
    --прогнозуємо значення
    predictedLabels = map (evalTree tree title) rows
    --зберігаємо індекс класифікатора в title
    classifierIndex = fromJust $ elemIndex (fst classifier) (map fst title)
    --реальні значення
    actualLabels = map (!! classifierIndex) rows
    uniqueLabels = nub actualLabels

    --створюємо рядок для реального значення
    generateRow actual = map (\pr -> countOccurrences actual pr) uniqueLabels
    --рахуємо співпадіння
    countOccurrences actual pr = length $ filter (\(a, p) -> a == actual && p == pr) $
zip actualLabels predictedLabels
    confusionMatrix = map generateRow uniqueLabels
  in confusionMatrix
```

Дана функція будує дерево та рахує кількості правильно та неправильно передбачених класів для рядків датасету.

На основі матриці невідповідностей можна обчислити різноманітні метрики.

3.3. Точність(accuracy):

$$Acc = \frac{\sum_{i=1}^{k} TP_{y_i}}{\sum_{i=1}^{k}\sum_{i=1}^{k} y_{i,j}}$$

Дана метрика[6, с.3] показує, наскільки часто модель правильно передбачає результат, набуває значень від 0 до 1. Вона чудово підходить для задачі бінарної класифікації для fishingData. Проте, оскільки вона розцінює усі класи як однаково важливі, що робить її менш корисною, якщо класи є незбалансованими, тобто одні значення зустрічаються набагато частіше за інші і, відповідно, модель може просто

передбачати будь-що як найбільш представлений клас і мати високу точність. Наступні метрики, які обчислюються окремо для кожного класу, краще підходять для реальних даних, де випадки незбалансованості вкрай розповсюджені.

3.4. Precision

Або Positive Predictive Value, прецизійність[6, с. 2]. Показує, як часто модель правильно передбачає клас з усіх передбачень для класу y_i.

$$PPV(y_i) = \frac{TP(y_i)}{TP(y_i) + FP(y_i)}$$

Оскільки метрика показує коректність передбачення саме цільового класу(що відповідає атрибуту-класифікатору), вона дозволяє отримувати правдиву оцінку ефективності моделі і для незбалансованих класів. Також, варто відмітити, що вона є корисною, коли перед моделлю стоїть задача допускати мінімальну кількість false positive передбачень.

Проте, є і недолік – ми не враховуємо неправильні передбачення. Для цього нам знадобиться наступна метрика.

3.5. Recall

True Positive Rate, повнота[6, с. 2]. На відміну від прецизійності, вона рахує співвідношення правильних передбачень для класу y_i до усіх прогнозів(у тому числі хибних) для y_i.

$$TPR(y_i) = \frac{TP(y_i)}{TP(y_i) + FN(y_i)}$$

Вона добре підходить для оцінки моделі, якщо саме false negative передбачення є небажаними

У повноти також є недолік, схожий на такий у точності: модель може передбачити усі значення як хибні і отримати 100% повноту.

3.6. F_1-score(F-міра)

Середнє гармонійне прецизійності та повноти[6, с. 4]

$$F_1(y_i) = 2 \cdot \frac{TPR(y_i) \cdot PPV(y_i)}{TPR(y_i) + PPV(y_i)}$$

Вона бере до уваги як false positive так і false negative передбачення, надаючи компроміс між повнотою та прецизійністю. Проте, це не означає, що вона завжди краще підходить для оцінки моделі. Знову ж таки, для конкретних задач може бути краще брати або прецизійність, або повноту окремо(для мінімізації FP та FN відповідно)

3.7. k-fold cross-validation

Аби застосувати вищенаведені метрики, скористаємося k-fold validation[7, с. 69]

```
kFoldCrossValidation :: Title -> [[Row]] -> Attribute -> AttSelector -> IO [(Double, Double, Double, Double)]
kFoldCrossValidation title folds classifier selector = do
  --проходимось по кожному фолду
  results <- forM folds $ \validationRows -> do
    --відділяємо тренувальний від решти
    let trainingRows = concat [fold | fold <- folds, fold /= validationRows]
      trainingSet = (title, trainingRows)
      validationSet = (title, validationRows)
      --будуємо дерево
      tree = buildTree trainingSet classifier selector
      --оцінюємо
```

```
            cm = computeConfusionMatrix validationSet tree classifier
            acc = accuracy cm
            (p, r, f1) = calculateMetrics cm
        printf "Accuracy: %.4f\n" acc
        printf "Precision: %.4f\n" p
        printf "Recall: %.4f\n" r
        printf "F1: %.4f\n" f1
        putStrLn ""
        return (acc, p, r, f1)
    return results


main :: IO ()
main = do
    let k = 10
    rows <- tab
    shuffledRows <- shuffleM rows
    let n = length shuffledRows `div` k
        folds = chunksOf n shuffledRows

    results <- kFoldCrossValidation t folds classifierAttribute bestGainAtt

    --після згортки маємо суми метрик
    let (avgAcc, avgPrecision, avgRecall, avgF1Score) = foldr (\(acc, prec, recl, f1)
(accSum, precSum, recSum, f1Sum) ->
            (accSum + acc, precSum + prec, recSum + recl, f1Sum + f1)) (0, 0, 0, 0) results
        avgAcc' = avgAcc / fromIntegral k
        avgPrecision' = avgPrecision / fromIntegral k
        avgRecall' = avgRecall / fromIntegral k
        avgF1Score' = avgF1Score / fromIntegral k
    printf "Average Accuracy: %.4f\n" avgAcc'
    printf "Average Precision: %.4f\n" avgPrecision'
    printf "Average Recall: %.4f\n" avgRecall'
    printf "Average F1: %.4f\n" avgF1Score'
```

3.8. Дані

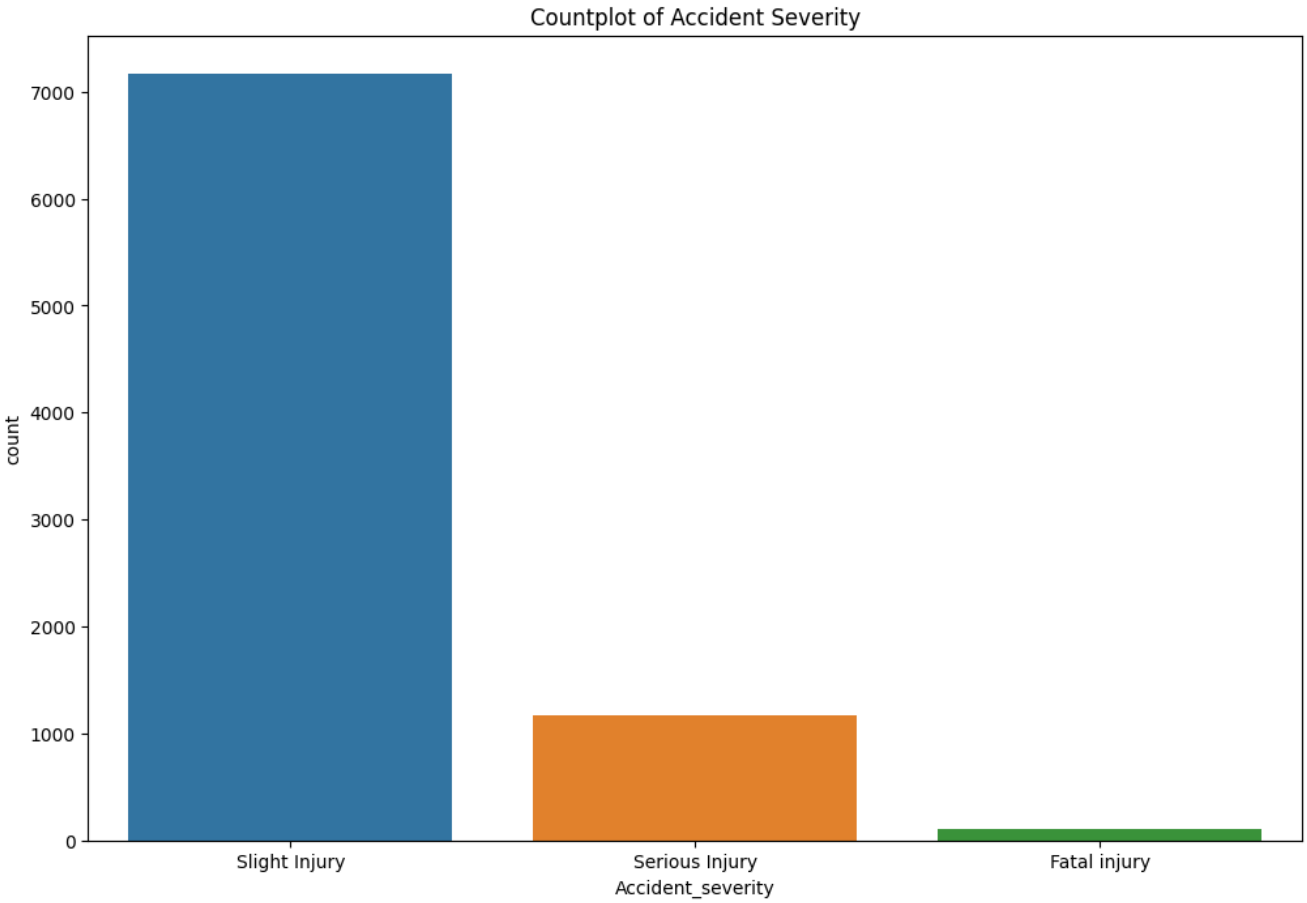
Візьмемо Road Accident Severity In India[8]:
Узгодимо його з припущеннями алгоритму: приберемо дублікати, числові змінні та відсутні значення

Таким чином, отримаємо датасет з 1725 рядками та наступними атрибутами:

k	*AttName*	*AttValue*
1	day	Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
2	driver_age	18-30, 31-50, Over 51, Unknown driver age, Under 18
3	driver_education	Junior high school, Above high school, High school, Unknown driver education, Elementary school, Illiterate, Writing & reading
4	driver_sex	"Male", "Female", "Unknown driver sex"
…	…	…
28	accident_severity	Serious Injury, Slight Injury, Fatal injury

Він містить інформацію про ДТП в Індії з 2017 по 2022.

З самої природи даних, вже можна з впевненістю сказати, що датасет не є збалансованим. Наприклад, поглянемо на атрибут accident_severity [Мал. 3.1]:



Малюнок 3.1

Отже, точність не надасть достовірної інформації про нашу модель

У результаті, при $k = 10$, тобто кожен має приблизно 172 рядки

Average Accuracy: 0.8582

Average Precision: 0.4415

Average Recall: 0.4645

Average F1: 0.2224

При k=23, по 75 рядків:

Average Accuracy: 0.7941

Average Precision: 0.7318

Average Recall: 0.6577

Average F1: 0,346

Хоч алгоритм більш пристосований до менших наборів даних і побудова дерева займає в середньому по 13 секунд на кожен тренувальний датасет, значення precision та recall вказують на непоганий відсоток передбачень, як для таких незбалансованих даних

ВИСНОВКИ

У дані роботі були розглянуті алгоритми побудови дерев ухвалення рішень, наведено можливу реалізацію алгоритму ID3 на Haskell.

Після реалізації модель було застосовано на реальних даних. Результати застосування були оцінені за допомогою методів аналізу даних та машинного навчання.

У майбутній роботі алгоритм можна дуже і дуже покращити, але завдяки набутому досвіду роботи з Haskell це буде вже простіше.

Отримана реалізація все ж може застосовуватися на реальних даних і показувати непогані результати навіть для незбалансованих датасетів.

Список використаних джерел

1. S. Safavian, D. Landgrebe, A survey of decision tree classifier methodology. IEEE Trans. Syst. Man Cybern. 21(3), 660–674 (1991). ISSN: 0018–9472
2. E.B. Hunt, J. Marin, P.J. Stone, Experiments in Induction (Academic Press, New York, 1966)
3. J. R. Quinlan. Induction of decision trees. Mach. Learn., 1(1):81–106, March 1986
4. GitHub - chris-taylor/aima-haskell: Algorithms from AIMA in Haskell. *GitHub*. URL: https://github.com/chris-taylor/aima-haskell (date of access: 27.05.2024).
5. Quilan, J. R. (1993). C4.5: programs for machine learning. San Mateo, CA: Morgan Kaufmann
6. Tharwat A. Classification assessment methods. *Applied Computing and Informatics*. 2020. Ahead-of-print, ahead-of-print. URL: https://doi.org/10.1016/j.aci.2018.08.003 (date of access: 28.05.2024).
7. Kuhn M., Johnson K. Introduction. *Applied Predictive Modeling*. New York, NY, 2013. P. 1–16. URL: https://doi.org/10.1007/978-1-4614-6849-3_1 (date of access: 27.05.2024).
8. Road Accident Severity in India. *Kaggle: Your Machine Learning and Data Science Community*. URL: https://www.kaggle.com/datasets/s3programmer/road-accident-severity-in-india (date of access: 27.05.2024).