МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАВЧАЛЬНО-НАУКОВИЙ КОМПЛЕКС

"ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ"

НАЦІОНАЛЬНОГО ТЕХНІЧНОГО УНІВЕРСИТЕТУ УКРАЇНИ

"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМ.І.СІКОРСЬКОГО"

КАФЕДРА МАТЕМАТИЧНИХ МЕТОДІВ СИСТЕМНОГО АНАЛІЗУ

КУРСОВА РОБОТА

з дисципліни

«Методи та технології обчислювального інтелекту»

на тему «Аналіз кредитних ризиків для фізичних осіб в умовах невизначеності з використанням нечітких нейронних мереж»

Виконав:

студент 5-го курсу

групи КА-02мп

Літвинчук А.М.

Прийняв:

професор, д.т.н.

Зайченко Ю. П.

Оцінка \_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_» \_\_\_\_\_\_\_\_ 2021 р.

Київ - 2021

# РЕФЕРАТ

Курсова робота про застосування нечітких нейронних мереж у задачах кредитного скорингу: 40 с., 20 рис., 1 табл., 1 додаток, 7 джерел.іі

Ключові слова: багатошаровий персептрон, нейро-нечітка мережа, функція належності, лінгвістичні правила, кредитний ризик.

Об’єкт досліджень – нечітка нейронна мережа NEFCLASS.

Мета роботи – реалізувати програмне забезпечення, яке матиме змогу навчати мережу NEFCLASS та перебирати її параметри задля знайдення оптимального рішення.

Метод дослідження – імпементація мережі NEFCLASS та емпіричні експерименти її якості на задачі кредитного скорингу.

У роботі наведено опис архітектури нейронної мережі NEFCLASS та розібрано етапи алгоритму навчання. Дана мережа була застосована для вирішення задачі кредитних ризиків для фізичних осіб. Також у роботі подається програмна реалізація даного рішення.

# ABSTRACT

Course work on the use of fuzzy neural networks in credit scoring problems: 40 pages, 20 figures, 1 table, 1 appendix, 7 sources.

Keywords: multilayer perceptron, neural fuzzy network, membership function, linguistic rules, credit risk.

The object of research is a fuzzy neural network NEFCLASS.

The purpose of the work is to implement software that will be able to train the NEFCLASS network and grid search its parameters in order to find the optimal solution.

The research method is the implementation of the NEFCLASS network and empirical experiments of its quality on the problem of credit scoring.

The paper describes the architecture of the NEFCLASS neural network and analyzes the stages of the learning algorithm. This network was used to solve the problem of credit risks. The program implementation of this solution is also presented in the work.

**s**

**ЗМІСТ**

[ВСТУП 5](#_Toc71229071)

[ПОСТАНОВКА ЗАДАЧІ 6](#_Toc71229072)

[ТЕОРЕТИЧНА ЧАСТИНА 7](#_Toc71229073)

[Нечітка нейронна мережа NEFCLASS 7](#_Toc71229074)

[Архітектура 7](#_Toc71229075)

[Алгоритм навчання 11](#_Toc71229076)

[Створення бази правил 12](#_Toc71229077)

[Навчання нечітких множин 13](#_Toc71229078)

[Метрики 15](#_Toc71229079)

[Точність 15](#_Toc71229080)

[F1 міра 16](#_Toc71229081)

[Крос-валідація 16](#_Toc71229082)

[ОПИС ПРОГРАМНОГО ПРОДУКТУ 19](#_Toc71229083)

[ОПИС ВХІДНИХ ДАНИХ 22](#_Toc71229084)

[РЕЗУЛЬТАТИ РОБОТИ 27](#_Toc71229085)

[ВИСНОВКИ 32](#_Toc71229086)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 33](#_Toc71229087)

[ДОДАТОК А. ЛІСТИНГ 34](#_Toc71229088)

# ВСТУП

Автоматизація прийняття рішень у наші часи добралась до різних сфер. Це економить час експерту та мінімізує можливість людської помилки. У даній роботі ми розглянемо банкову сферу, а саме скорингові моделі.

Алгоритми кредитного скорингу, які дозволяють здогадуватися про ймовірність дефолту – це метод, який банки використовують для визначення того, чи слід надавати позику чи ні. У цій роботі задачею буде передбачення, чи зазнає клієнт фінансового банкрутства у найближчих 2 роки.

Існує безліч алгоритмів для прогнозування фінансового банкрутства, проте у даній роботі ми розглянемо саме нейронні мережі з нечіткою логікою. Їхньою перевагою є можливість враховувати невизначеність параметрів та обробка багатовимірних даних. Розглядатиметься саме модель NEFCLASS, яка об’єднує універсальні можливості нейронних мереж з можливістю обробки невизначеності даних. В результаті отримаємо модель, яка буде класифікувати фізичних осіб на 2 групи : на осіб, які будуть здатні оплатити кредит та осіб, які збанкрутують.

# ПОСТАНОВКА ЗАДАЧІ

Основними задачами цією роботи є:

* програмна реалізація нечіткої нейронної мережі NefClass та її навчання;
* програмна реалізації крос-валідаційного способу навчання та оцінки результатів моделі;
* імплементація метрик для оцінки результатів моделі;
* програмна реалізація перебору основних параметрів параметрів NefClass;
* оцінка результатів перебору параметрів, визначення найкращих варіантів, порівняльний аналіз та оцінка фінальної моделі.

# ТЕОРЕТИЧНА ЧАСТИНА

## Нечітка нейронна мережа NEFCLASS

Пропонується розглянути систему нечіткої класифікації NEuro-Fuzzy CLASSifier, скорочено надалі NEFCLASS, що заснована водночас на архітектурі нечіткого багатошарового персептрона та алгоритмах нечіткого логічного висновку. Зауважимо, що нечіткість даного алгоритму пояснюється недосконалістю або неповнотою вимірювань тих властивостей об'єктів, за якими здійснюється процес класифікації.

Модель NEFCLASS є нейро-нечітким класифікатором, принцип роботи якого заснований на отриманні нечітких правил з безлічі даних, які можна розділити на кілька непересічних класів.

Завдання NEFCLASS полягає в тому, щоб визначити приналежність до класу зразка з набором вхідних параметрів. Однак одна з головних проблем при проектуванні нейро-нечіткої системи полягає у правильні побудові функцій належності та визначенні нечітких правил.

Далі пропонується розглянути більш детально архітектуру нейро-нечіткої мережі NEFCLASS та покроково розібрати її алгоритм навчання.

### Архітектура

Нейро-нечітка мережа NEFClASS має тришарову послідовну архітектуру, до якої входить:

1. вхідний шар;
2. прихований або так званий шар нейронів нечітких правил;
3. вихідний шар нейронів класифікації.

Приклад схематичного зображення архітектури ННМ NEFClASS для задачі з двома входами, п’ятьма лінгвістичними правилами та двома класами подано на рис. 2.1., а кожен із зазначених вище шарів позначено як U1, U2 та U3 відповідно.

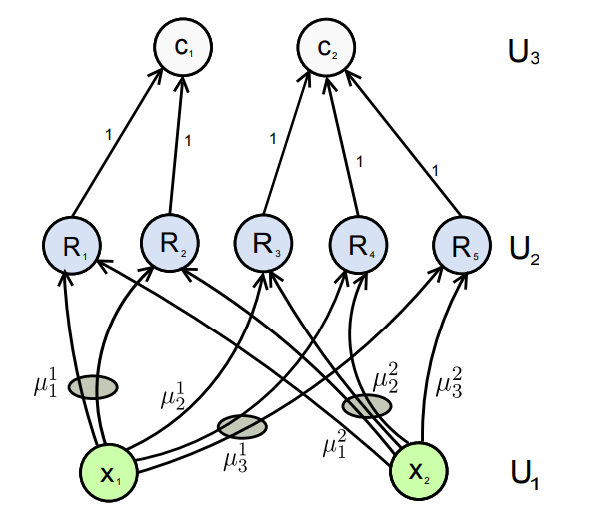
****

Рис. 2.1. – Архітектура нейро-нечіткої мережі NEFCLASS

Далі перейдемо до окремого розгляду кожного з трьох шарів мережі NEFClASS та зазначимо їх характерні особливості.

Тож, перший шар представляється вхідними нейронами, які складаються з вхідних зразків. Вхідні зразки, або як ще їх називають патерни – це вектори ознак певного об’єкта, що записуються як .Величина кожного з таких вхідних зразків представляється нечіткою множиною, при чому для кожної 𝑥𝑖 існує 𝑞𝑖 нечітких множин µ1𝑖 , … , µ𝑖𝑞𝑖.Тобто відбувається процес фазифікації вхідних нейронів, результатом якого є перехід від чітких значень патернів до нечітких значень відповідних лінгвістичних змінних. Таким чином для всіх вхідних змінних визначаються конкретні значення функції приналежності для кожної з лінгвістичних змінних. Їх можна записати у вигляді:

= , (1)

де jj – це той терм, який відповідає максимальному значенню функції належності i-вхідної змінної.

Кожна нечітка множина позначається лінгвістичним термом , що може, наприклад, набувати таких значень: «низький», «достатній», «відмінний» тощо. Тобто лінгвістичний терм ( *є*{1,…, }) є зв’язною ланкою між *xi* *є* *U*1 та *є U*2. Також варто зазначити, що з'єднання, які виходять із одного і того ж вузла вводу *х* та мають однакові мітки й однакову вагу в будь-який час, називаються зв'язними з'єднаннями, а їх вага називається зваженою вагою.

Другим шаром архітектури нейро-нечіткої мережі NEFClASS є прихований шар, який представлений базою правил, що складається з *k* нечітких лінгвістичних правил *R1, …, Rn* та є так званою апроксимацією невідомої функції : → , що описує класифікаційну задачу, де (𝑥) = (, … , ) така, що = 1, = 0 (𝑗 {1, … , 𝑚}) , а *x*  належить класу .

Активація нейронів цього шару з патерном *p* розраховується як функція *t-норми:*

= , (2)

де – це нечітка вага з’єднання вхідного нейрона з нейроном правила – це активація вхідного нейрона.

У загальному лінгвістичним правилом нечіткого логічного висновку *«якщо-то»* називається конструкція типу:

*R = якщо x є А, то у є В,* (3)

де *х* – це вхідна лінгвістична змінна,

*у* – це вихідна лінгвістична змінна,

*A* і *В* – це деякі нечіткі множини (функції приналежності), взяті із терм-множин змінних *x* та *y* відповідно.

Зазначимо, що у записі (3) вираз *«x є А»* – є нечітким висловлюванням, що характеризує умову правила (антецедент) а вираз *«у є В»* – позначає висновок правила (консеквент). Також, нечіткі множини, що направлені на однаковий вузол правил R, називаються антецедентами цього вузла.

Стосовно нейро-нечіткої мережі NEFCLASS, то нечіткі правила формуються наступним чином, реалізуючи висновок Мамдані:

(4)

Тобто це можна трактувати так що, *якщо і і … і , то патерн належить класу i.*

Тож, задача NEFCLASS полягає у тому, щоб визначити належність до класу *C ⊂*  вхідної ознаки .

Останній шар мережі складається з вихідних нейронів кожного класу, активація яких обчислюється на основі активації правил, які в якості висновку вказують на певний клас. Для цього використовується одна з формул, поданих нижче*:*

, або (5)

(6)

де *W(R, с)* – це нечітка вага з’єднання нейрона правил *R* з вихідним нейроном,

– це активація нейрона правил.

Як правило, *W(R, с)* дорівнює 1. Синаптичні ваги між нейроном правила та нейроном вихідного шару отримають значення 1, якщо поточний зразок навчальної вибірки дійсно належить цьому класу, і значення 0 для ваг з іншими класами.

### Алгоритм навчання

Результати роботи нейро-нечіткої мережі NEFCLASS напряму залежать від нечітких множин та лінгвістичних правил, що будуть застосовані до вхідних зразків. Вони обираються із множини вибірок шляхом навчання системи. Перед тим як розглянути сам процес навчання варто зазначити, що експерт має задати два обов’язкових параметри:

1. кількість початкових нечітких множин;
2. максимальна кількість вузлів правил, що можуть бути створені у прихованому шарі.

Зазначимо, що нейро-нечітка мережа NEFClass буде складатися з *n* вхідних нейронів (𝑥1, 𝑥2, ..., 𝑥𝑛), максимальної кількості нечітких правил (𝑘 ≤ 𝑘𝑚𝑎𝑥) та *m* вихідних нейронів (c1, c2, …, c𝑚). Окрім цього буде задано навчальну множину зразків *L* = {(𝑝1, 𝑡1), … , (𝑝𝑠, 𝑡𝑠)}, кожен з яких буде складатися з вхідного зразка та бажаного зразка *t ∈ .* Алгоритм навчання мережі NEFCLASS складається з двох етапів:

1. генерації або задання бази лінгвістичних правил;
2. налаштування ваг багатошарового персептрона за допомогою емпіричного алгоритму чи теоретично обґрунтованих алгоритмів навчання нейронних мереж.

Надалі пропонуємо розглянути кожен із цих етапів більш детально.

#### Створення бази правил

Метою даного етапу є створення *k-кількості* правил для мережі NEFCLASS. Задля цього потрібно:

1. обрати патерн (𝑝, 𝑡) із навчальної множини *L;*
2. для кожного вхідного нейрона *xi* *є U1* знайти таку функцію належності , що:

(7)

1. якщо *k-кількість* вузлів правил є меншою за максимально встановлену і не існує вузла правила *R* з умовою, що *W(x1, R)* = то створюємо такий вузол і з'єднуємо його з вихідним вузлом *c1,* якщо *t1 = 1;*
2. якщо ще лишилися неопрацьовані патерни із навчальної множини *L* та виконується умова, що 𝑘 ≤ 𝑘𝑚𝑎𝑥 , то переходимо до першого кроку. Інакше алгоритм перебору патернів зупиняється.

Наступним кроком є визначення безпосередньо самої бази правил, що може бути сформована за однією з трьох процедур:

1. *просте* навчання правил, тобто коли до бази правил потрапляють перші 𝑘 правил;
2. *найкраще* навчання правил, яке зводиться до наступного: при надходженні всіх зразків навчальної вибірки для кожного антецедента правила підраховується рейтинг:

де – це активація нейрона правил R.

Коефіцієнт може приймати два значення: «1» – у випадку, якщо зразок *p* класифіковано правильно та «-1» – за умови хибної класифікації. Потім з усіх нейронів правил мережі NEFCLASS обираємо і залишаємо в базі *k* нейронів правил з найвищими значеннями .

1. *найкраще для кожного класу* навчання правил подібний до *найкращого* навчання правил за винятком того, що для кожного класу залишаються ті кращі [ ] правил, наслідки яких представляють клас (де [𝑥] − ціла частина від 𝑥).

#### Навчання нечітких множин

Під час цього етапу проводитьсянавчання параметрів функцій належності нечітких множин. Алгоритм навчання з учителем повинен адаптувати нечіткі множини, пробігаючи циклічно через усю навчальну множину *L* до тих пір поки не спрацює один із критеріїв зупинки. Надалі sрозглянемо усі етапи цього алгоритму детальніше:

1. обираєм патерн (𝑝, 𝑡) із навчальної множини *L* і подаємо його на вхід мережі, визначаючи вихідний вектор *с;*
2. для кожного вихідного нейрона *ci* визначаємо значення похибки:

де – це бажаний вихід нейрона,

– це фактичний вихід нейрона *ci*.

1. для кожного нейрона правил R, для якого виконується умова, що вихід

> 0, виконуємо наступні розрахунки:

* обчислюємо значення похибки нейрона правил R:
* знаходимо такий вхідний нейрон 𝑥′ що

𝑊(𝑥′, 𝑅) (𝑎𝑥′ ) =

* для нечітких множин 𝑊(𝑥′, 𝑅) розраховуємо зміщення параметрів 𝛿𝑎, 𝛿𝑏 та 𝛿𝑐 функції приналежності, використовуючи параметр – швидкість навчання 𝜎 > 0:

І за одержаними розрахунками змінюємо 𝑊(𝑥′, 𝑅).

* обчислюємо похибку правила:

Алгоритм навчання параметрів функцій належності зупиняється якщо спрацював один із критеріїв зупинки, наприклад:

1. похибка набуває значення близького до нуля;
2. похибка не зменшується протягом *n* ітерацій.

## Метрики

### Точність

Основна метрика, яка часто використовується у задачах класифікації. Має таке визначення:

Accuracy = (TP + TN) / (TP + FP + TN + FN)

де TP – кількість вгаданих прикладів з класом 1;

TN - кількість вгаданих прикладів з класом 0;

FP – кількість помилок, коли модель дала клас 1, а насправді там 0;

FN – кількість помилок, коли модель дала клас 0, а нсправді там 1.

Проте ця метрика є дуже оманливою. Скажімо, у нас 99 прикладів класу 0 та 1 приклад класу 1. Уявімо, що наша модель будь-якому прикладу присвоєю мітку 0. Тоді у нас точність:

Accuracy = (0 + 99) / (0 + 0 + 99 + 1) = 99 / 100 = 99%

Здається, що точність аж 99%, проте ми не вгадали жодного прикладу з класом 1, що є дуже погано наприклад у випадку важких хвороб. Аналогічно і в нас, нам не потрібно щоб модель завжди казала, що людина збанкрутує або навпаки, що людина завжди віддаватиме борг. Тому введемо ще одну метрику.

### F1 міра

Введемо визначення цієї метрики:

Precision = TP / (TP + FP)

Recall = TP / (TP + FN)

F1 = 2 \* Precision \* Recall / (Precision + Recall)

де TP, FP, FN – те саме, що у точності;

Precision – допоміжна метрика, яка показує відсоток вгаданих прикладів з тих, яким ми дали клас 1;

Recall – допоміжна метрика, яка показує відсоток вгаданих прикладів з усіх прикладів з класом 1 (повнота).

Розглянемо попередній іграшковий приклад:

Precision = 0 / (0 + 0 + 1e-9) = 0 (в даному випадку у знаменнику додаємо маленький епсілон, тому результат 0)

Recall = 0 / (0 + 1) = 0

F1 = 2 \* 0 \* 0 / (0 + 0 + 1e-9) = 0

Тобто цей варіант чесно оцінює нашу модель, крім того він повністю незалежний від незбалансованості класів. У експериментах будемо моніторити обидві метрики, проте орієнтуватись будемо саме на F1-міру.

## Крос-валідація

Кожен експеримент потрібно оцінити за певною метрикою. Є декілька варіантів як отримати значення метрики на тестовій вибірці, один з них – просто розбити дані, що ми маємо на тренувальну та тестову вибірки, навчати на тренувальній, а валідаувати відповідно на тестовій. Проте у цього способу є недостатки, якщо ми хочемо підбирати гіперпараметри мережі, а саме висока ймовірність перенавчання під цю тренувальну та тестові вибірки, тобто результати з кожним експериментом будуть все менш об’єктивиними та збільшується шанс, що підібрані параметри будуть погано переноситись і на інші приклади з даного розподілу даних.

Саме тому використовувати ми будемо механізм під назвою крос-валідація. У неї є параметри – кількість розбиттів. У даній роботі розглядається 5 розбиттів. Алгоритм крос-валідації наступний:

1. Випадково (або за певним принципом) беремо (N – 1) / N відсотків даних у тренувальну вибірку, 1 / N відсотків даних у тестову (при цьому тестові дані завжди мають буть унікальними для кожного розбиття, тобто перетину немає)
2. Навчаємо модель на тренувальній вибірці
3. Валідуємо модель на тестовій вибірці, зберігаємо метрики
4. Робимо пункти 1-3 N разів (тестова вибірка кожного разу має бути унікальна і без перетинів між розбиттями)
5. Усереднюємо отримані метрики

Цей процес можна зобразити так:

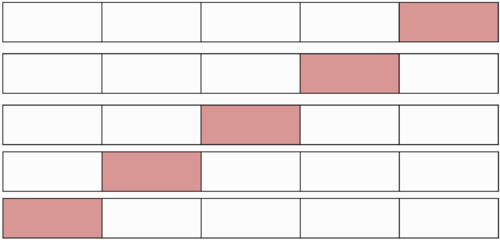


Рис. 1 Крос-валідація, білий блочок – дані, які йдуть у тренувальну вибірку, червоний – у тестову

Завдяки крос-валідації ми можемо отримати більш точну оцінку моделі, оскільки маємо тепер N спостережень (у нашому випадку 5). Всі моделі ми будемо навчати і оцінювати саме за таким принципом.

# ОПИС ПРОГРАМНОГО ПРОДУКТУ

Програмний продукт був розроблений на мові програмування Python. Весь код доступний за посиланням : <https://github.com/litvinich/mtoi_coursework>.

1. Перш за все був розроблений клас, який містив реалізацію мережі NEFCLASS та функцію для його навчання (знаходиться в src/neflcass.py та src/train\_utils.py)
2. Метрики були реалізовані за допомогою бібліотеки з відкритим кодом sklearn.
3. Після цього було розроблено функцію для проведення крос-валідації (знаходиться в src/train\_utils.py).
4. Було реалізовано швидкий перебір параметрів з легким інтерфейсом. Перебір параметрів відбувається паралельно на всіх ядрах процесора для максимального швидкого перебору параметрів (перебір параметрів у мене зайняв приблизно 4.5 години на 20 ядрах, без паралелізації це б зайняло приблизно 50-70 годин).

Розгляньмо інтерфейс основної функції для перебору параметрів (<https://github.com/litvinich/mtoi_coursework/blob/master/grid_search.py>):

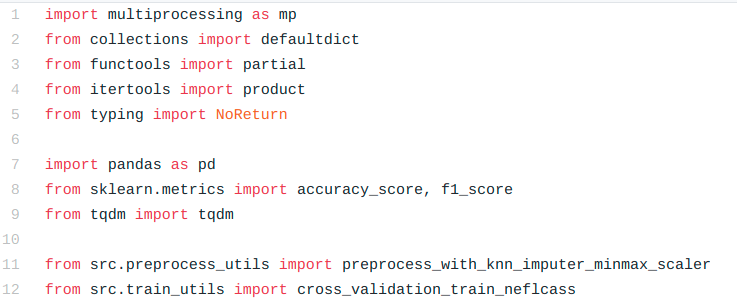


Рис.2 Підгрузка необхідних функцій так класів

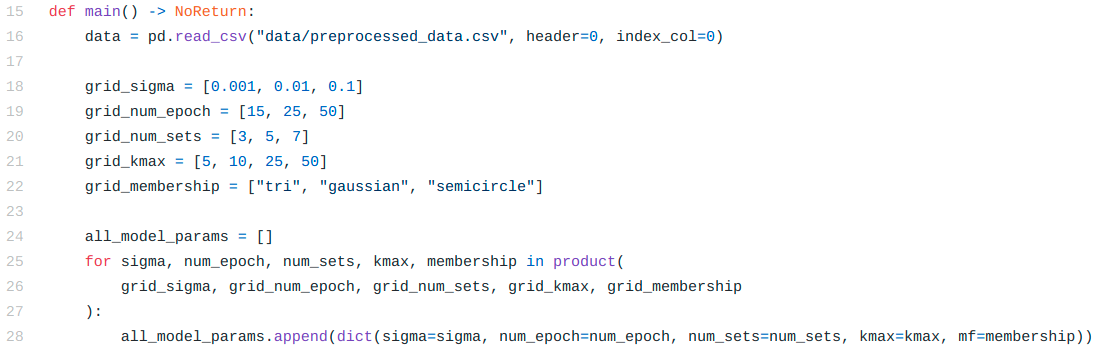


Рис.3 Підгрузка даних та визначення параметрів, які ми хочемо перебирати (можливо перебирати швидкість навчання, кількість епох, тобто довжина навчання, кількість початкових нечітких множин, максимальна кількість вузлів правил, функцію належності)

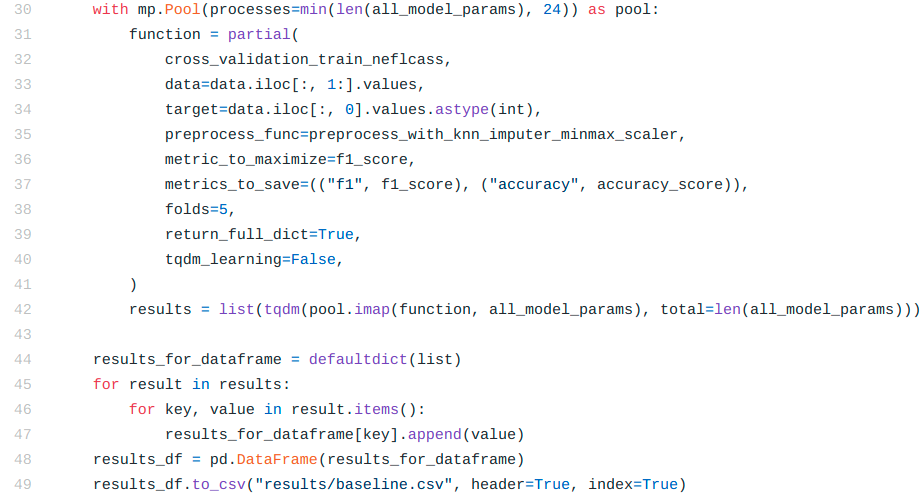


Рис.4 Паралельний запуск крос-валідації для кожного набору параметрів та збереження результатів у “results/baseline.csv”

Для запуску виконуємо наступну команду:

python grid\_search.py

В результаті отримаємо ось таку таблицю з результатами для подальшого аналізу:

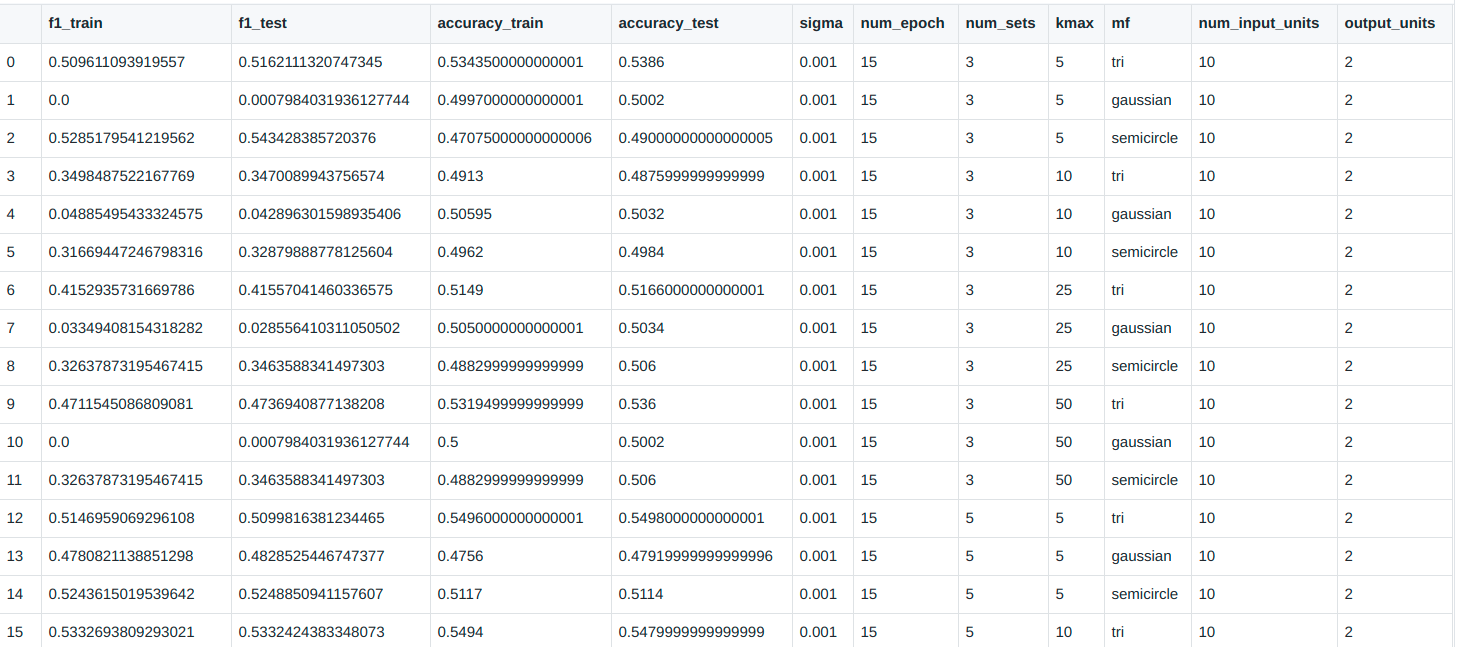


Рис. 5 Результати роботи

# ОПИС ВХІДНИХ ДАНИХ

Дані були взяті з сайту kaggle.com , а саме з змагання Give Me Some Credit (<https://www.kaggle.com/c/GiveMeSomeCredit>). Задача була досить важкою, оскільки для її вирішення залучили велику кількість бажаючих з грошовою винагородою за найкраще рішення. Проте початкова вибірка була занадто великою (більше 150000 кредиторів), тому я взяв підвибірку, опрацювання якої займало б розумний час на комп’ютері, що я маю – 5000 прикладів (2500 з них віддали борг за 2 роки, а 2500 – ні).

Задачею було реалізувати модель для передбачення чи збанкрутує фізична особа у найближчих 2 роки. Для цього пропонувалось використати такі предиктори:

1. RevolvingUtilizationOfUnsecuredLines – загальний залишок за кредитними картками та особистими кредитними лініями, за винятком нерухомості та відсутність боргу в розстрочку, як автокредити, поділений на суму кредитних лімітів (тип даних: відсоток).
2. Age – вік фізичної особли (тип даних: ціле число).
3. NumberOfTime30-59DaysPastDueNotWorse – кількість разів, коли позичальник прострочував 30-59 днів, але не більше за останні 2 роки (тип даних: ціле число).
4. DebtRatio – щомісячні виплати боргу, аліменти, витрати на життя, поділені на місячний валовий дохід (тип даних: відсоток).
5. MonthlyIncome – місячний дохід (тип даних: ціле число).
6. NumberOfOpenCreditLinesAndLoans – кількість відкритих позик (розстрочка, наприклад, автокредитування або іпотека) та кредитних ліній (наприклад, кредитних карток) (тип даних: ціле число).
7. NumberOfTimes90DaysLate – кількість випадків, коли позичальник прострочував 90 днів або більше (тип даних: ціле число).
8. NumberRealEstateLoansOrLines – кількість іпотечних кредитів та позик на нерухомість, включаючи кредитні лінії власного капіталу (тип даних: ціле число).
9. NumberOfTime60-89DaysPastDueNotWorse – кількість випадків, коли позичальник прострочував 60-89 днів, але не більше за останні 2 роки (тип даних: ціле число).
10. NumberOfDependents – кількість утриманців у сім'ї, крім них самих (дружина, діти тощо) (тип даних: ціле число).

Варто підмітити, що всі предиктори – цілі числа або відсотки, тобто ніякої додаткової обробки текстових змінних не потрібно, лише нормалізація. Проведемо більш детальний огляд розподілів цих величин, щоб зрозуміти складність задачі.

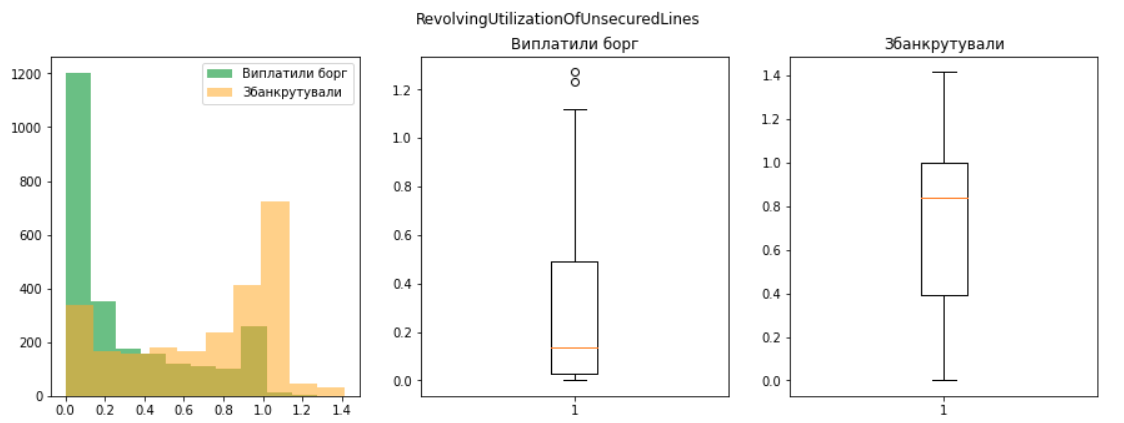


Рис. 6 Розподіли в залежності від класу предиктора 1

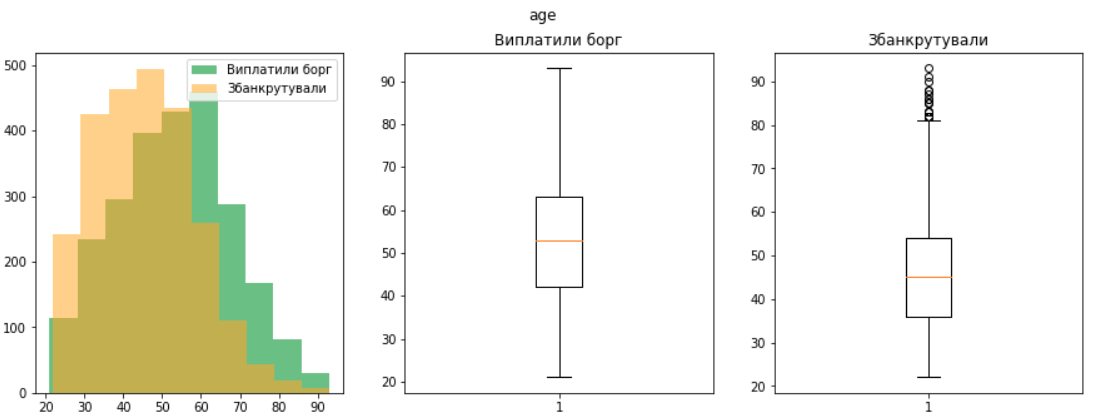


Рис. 7 Розподіли в залежності від класу предиктора 2

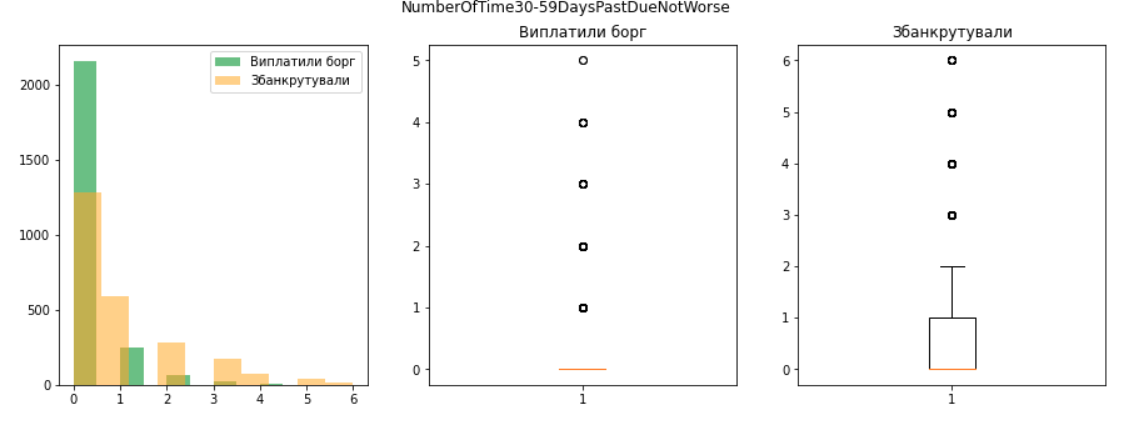


Рис. 8 Розподіли в залежності від класу предиктора 3

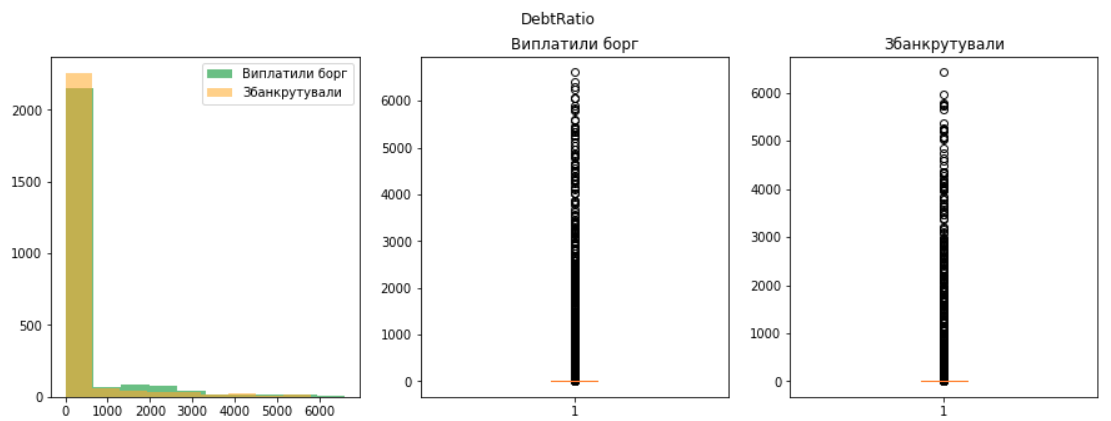


Рис. 9 Розподіли в залежності від класу предиктора 4

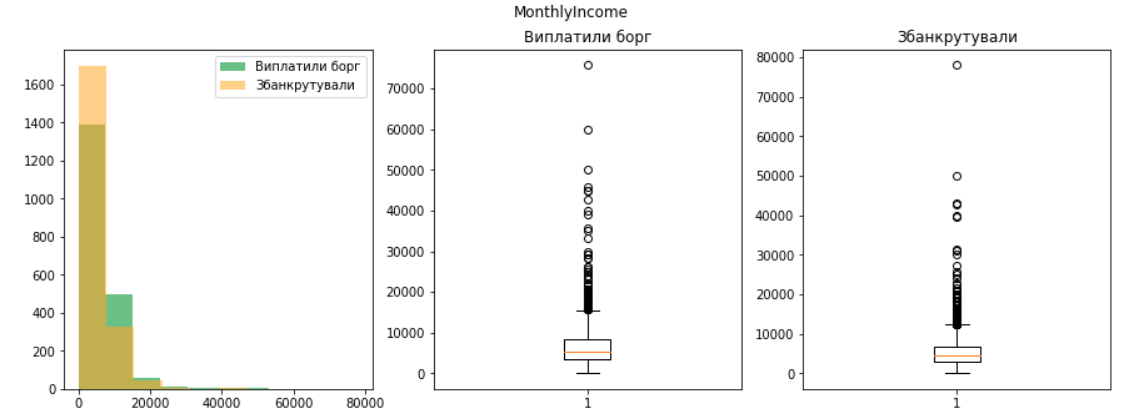


Рис. 10 Розподіли в залежності від класу предиктора 5

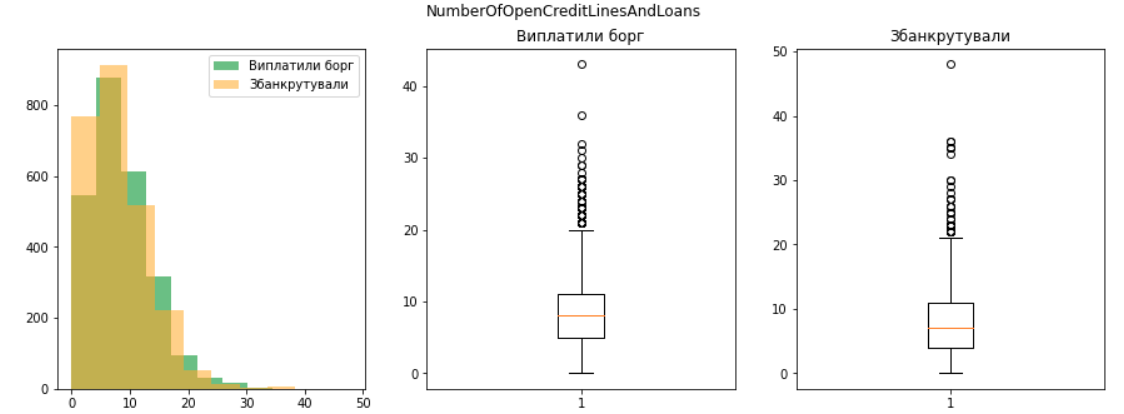


Рис. 11 Розподіли в залежності від класу предиктора 6

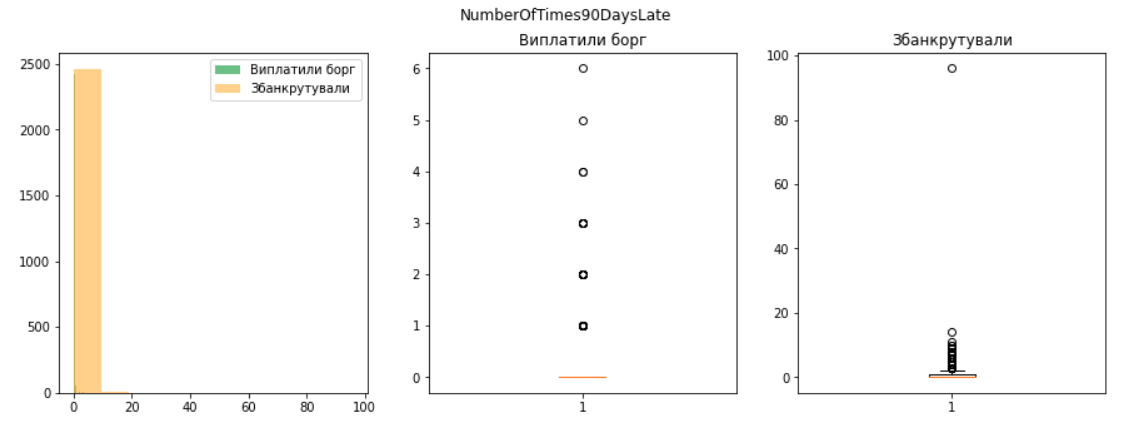


Рис. 12 Розподіли в залежності від класу предиктора 7

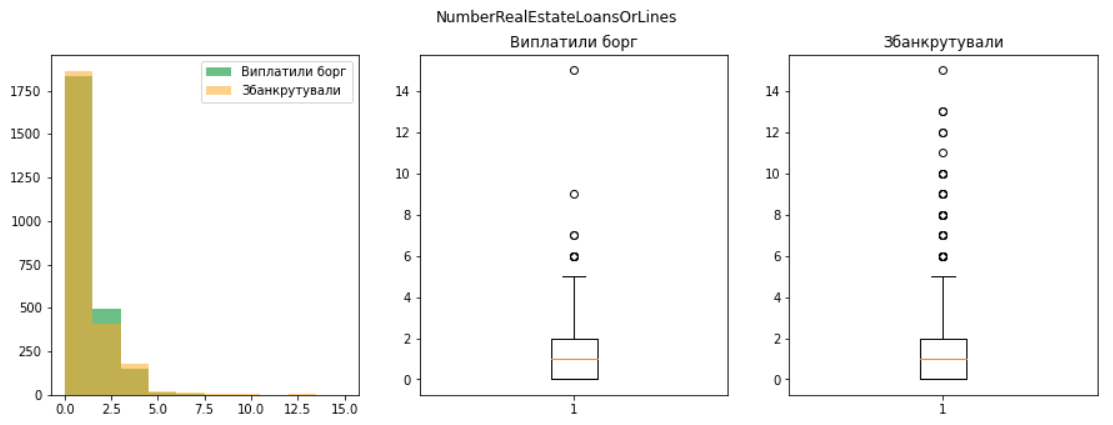


Рис. 13 Розподіли в залежності від класу предиктора 8

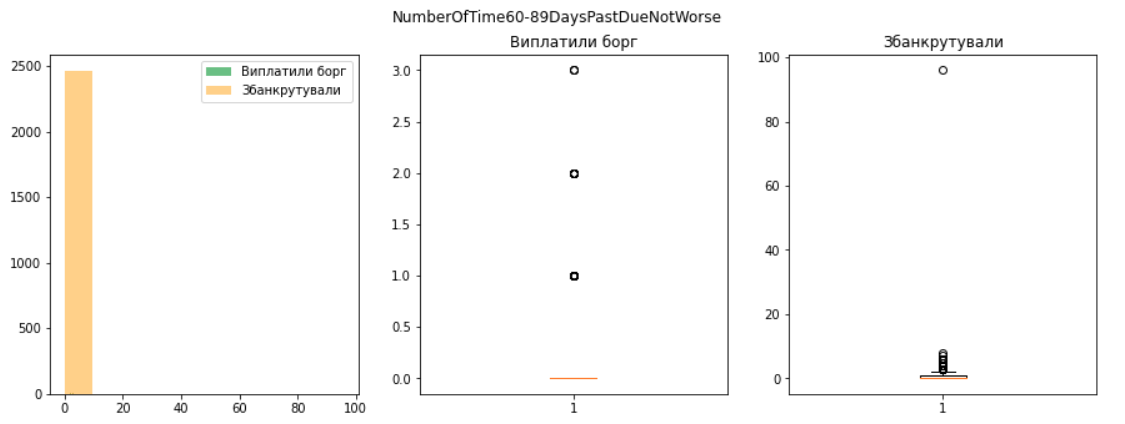


Рис. 14 Розподіли в залежності від класу предиктора 9

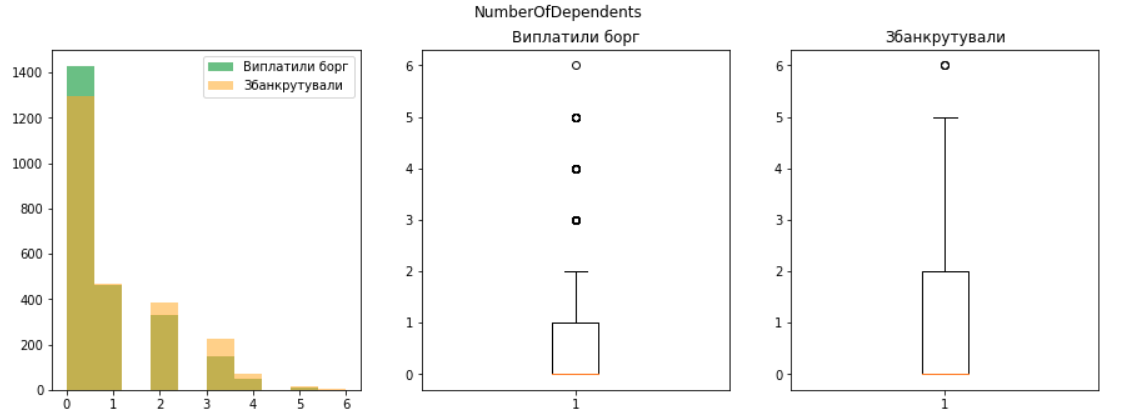


Рис. 15 Розподіли в залежності від класу предиктора 10

Деякі предиктори самі по собі несуть сильні ознаки майбутнього банкрутства (предиктори 1, 2, 3, 7, 9, 10), і це навіть між взаємних залежностей (людина з 3 дітьми і заробітком 3000 доларів набагато ймовірніше збанкрутує за людину з 3 дітьми та доходом 10000 доларів).

Таким чином ми побачили, що задача може вирішуватись з непоганою точністю (немає жодного предиктора, по якому можна було б ідеально відділити класи, тому ідеально задача не вирішиться, варто сподіватись на 60-70% точності).

# РЕЗУЛЬТАТИ РОБОТИ

В результаті було досліджено наступні параметри моделі:

grid\_sigma = [0.001, 0.01, 0.1]

grid\_num\_epoch = [15, 25, 50]

grid\_num\_sets = [3, 5, 7]

grid\_kmax = [5, 10, 25, 50]

grid\_membership = ["tri", "gaussian", "semicircle"]

Це відповідно швидкість навчання, кількість епох (довжина навчання), кількість початкових нечітких множин, максимальна кількість вузлів правил, функція належності.

Крос-валідація відбувалась на 5 розбиттях. Під час навчання зберігалась модель, яка є найкращою по цільовій метриці. Цільову метрику можна обрати – це може бути точність або F1-міра, у цій роботі оптимізувалась F1-міра. Якщо цільова метрика не покращувалась протягом 20 епох – зупиняли навчання (early stopping).

Спочатку оберемо найкращий варіант для кожного параметру. Щоб це зробити, зафіксуємо всі експерименти з цим параметром та візьмемо середнє для кожної метрики. Таким чином отримаємо справедливу оцінку і не буде варіанту, що ми обрали певний параметр через випадковий збіг обставин.

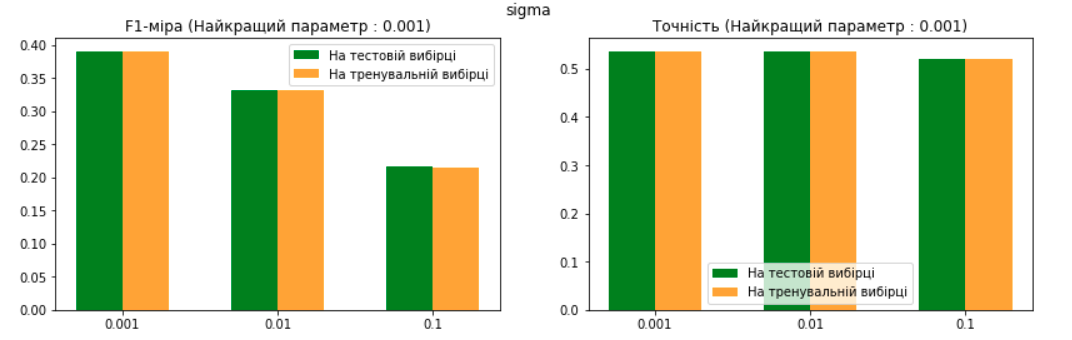


Рис. 16 Результати для швидкості навчання

Однозначно і по точності, і по F1-мірі найкращою швидкістю навчання є 0.001.

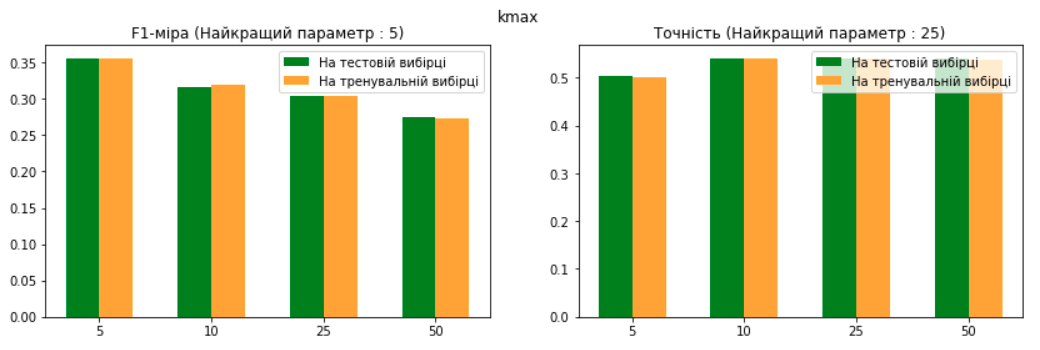


Рис. 17 Результати для максимальної кількості вузлів правил

Бачимо, що для оптимізації F1-міри варто взяти 5, тобто важливо взяти невелику кількість правил, щоб не перенавчитись, а для оптимізації точності – 25, тобто в 5 разів більше. Проте, як вже казалось раніше, точність більш оманлива, тому варто обережно обирати цей параметр. В кінці побачимо найбільш оптимальний набір параметрів для F1-міри та точності.

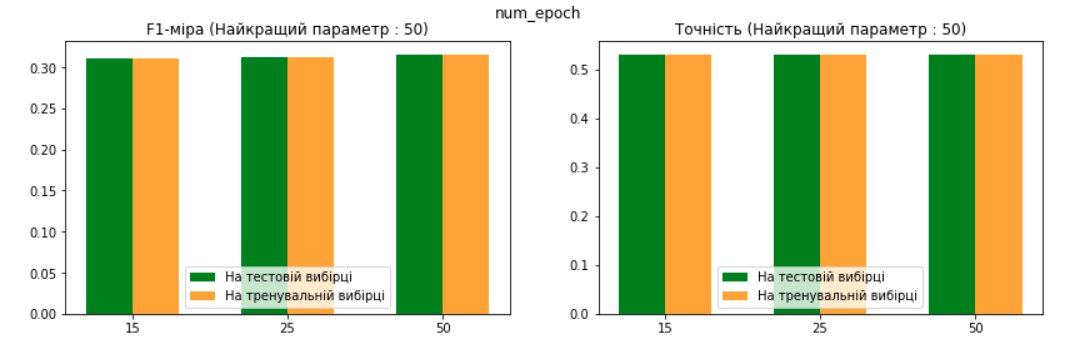


Рис. 18 Результати для довжини навчання (кількості епох)

Однозначно найкращим варінтом є 50 епох. Це означає, що в поєднанні з низькою швидкістю навчання ми отримуємо більш глибокий мінімум, який веде до кращої генералізації моделі.

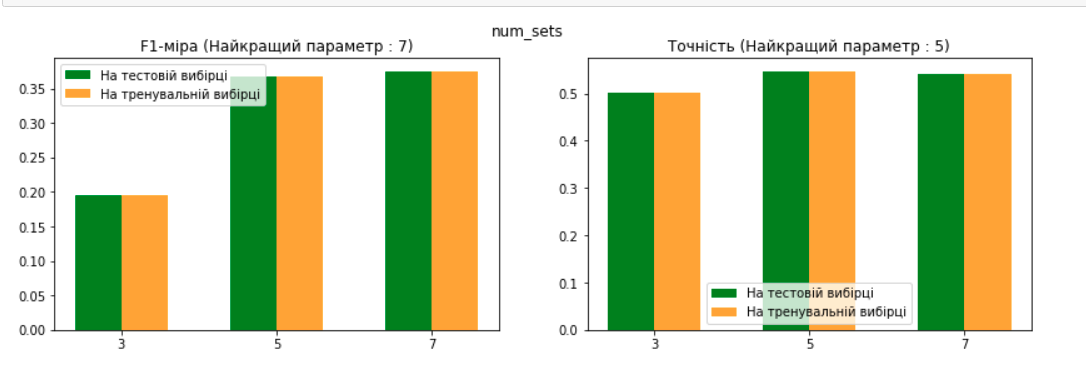


Рис. 19 Результати для кількості початкових нечітких множин

Для оптимізації F1-міри найкраще взяти 7 початкових нечітких множин, для точності – 5 множин. Різниця між 5 та 7 в обох випадках невелика, тому насправді для даного набору даних варто брати більшу кількість цих множин, щоб краще оперувати невизначеністю даних.

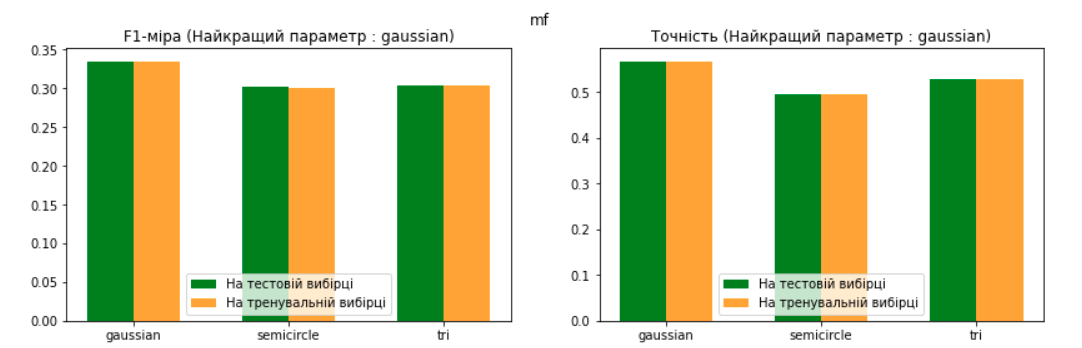


Рис. 20 Результати для функції належності

Однозначно найкращою функцією належності є гаусівська.

Таким чином, на глобальному рівні найкращим набором параметрів для оптимізації F1-міри є :

sigma = 0.001

num\_epoch = 50

num\_sets = 7

kmax = 5

membership = "gaussian"

а для оптимізації точності:

sigma = 0.001

num\_epoch = 50

num\_sets = 5

kmax = 25

membership = "gaussian"

Проте об’єднавши ці параметри не обов’язково можна отримати глобальний мінімум на задачі, оскільки не розглядались комбінації цих параметрів. Це лише глобальні мінімуми, якщо інші параметри вважати випадковими, але цього достатньо щоб зробити висновки. У даній роботі знайдені глобальні найкращі параметри гарантували глобальний мінімум.

Варто помітити, що ніде немає великої різниці між тренувальною та тестовою вибірками. Це свідчить, що крос-валідація дуже добре справилась зі своєю задачею та змогла прибрати зайву дисперсію в обох метриках. У наступній таблиці порівняння простих скоринговийх моделей (логістична регресія, дерево рішень) також навчених на цьому датасеті за тою ж методологією.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Модель | F1 на тренувальній | F1 на тестовій | Точність на тренувальній | Точність на тестовій |
| NEFCLASS з оптимізованими під F1-міру параметрами | 0.54056 | 0.535315 | 0.655 | 0.6536 |
| NEFCLASS з оптимізованими під точність параметрами | 0.527619 | 0.533882 | 0.6652 | 0.669 |
| Логістична Ridge регресія | 0.4543 | 0.4458 | 0.555 | 0.5677 |
| Дерево рішень | 0.951 | 0.499 | 0.96 | 0.621 |

Таблиця 1 Порівняння всіх моделейіі

Бачимо, що моделі NEFCLASS показують себе краще за стандартні моделі, які прийнято використовувати для кредитного скорингу як бейзлайни.

# ВИСНОВКИ

У даній курсовій роботі було розроблено програмне забезпечення для пошуку найкращих параметрів нечіткої нейронної мережі NEFCLASS. Зокрема, це ПП було використано для задачі кредитного скорингу.

Моделі NEFCLASS показали себе краще за базові аналоги для даної задачі та залишили простір для покращення. Зокрема, найкраща модель показала схильність до спрощення (5 максимальних початкових вузлів) та довшого навчання з меншою швидкістю (швидкість = 0.001, а найкраща кількість епох = 50, тобто максимальна з перебору). Це означає, що можна і надалі покращувати отримані результати.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Зайченко Ю. П. Порівняльний аналіз нейро-нечітких систем класифікації електрооптичних зображень в умовах інформаційної невизначеності / Ю. П. Зайченко, І.М. Петросюк // Системні дослідження та інформаційні технології : науково-технічний журнал. – 2006. – № 3. – С. 110–124.
2. Зайченко Ю. П. Исследование нечетких нейронных сетей в задачах распознавания объектов электрооптических изображений / Ю. П. Зайченко, И. М. Петросюк, М. C. Ярошенко // Системні дослідження та інформаційні технології : науково-технічний журнал. – 2009. – № 4 – Бібліогр.: 8 назв.
3. Zaychenko Y. P. The Fundamentals of Computational Intelligence: System Approach / Y. P. Zaychenko, M. Z. Zgurovsky.: Springer International Publishing, 2017. – 375 с.
4. Nauck D. NEFCLASS - a neuro-fuzzy approach for the classification of data [Електронний ресурс] / Nauck D., Kruse R. // ResearchGate. – 1995. – Режим доступу до ресурсу: <https://www.researchgate.net/publication/221000724>.
5. Сівчук Т. Система кредитного скорингу в банках: зарубіжний досвід та українські реалії / Т. Сівчук. // Наука молода. – 2013. – №19
6. Gliwa. B. Hybrid Neuro-Fuzzy Classifier Based On Nefclass Model [Електронний ресурс] / Gliwa B., Byrski A. // ResearchGate. – 2011. – Режим доступу до ресурсу: <https://www.researchgate.net/publication/307782104>.
7. Nauck D. Fuzzy data analysis with NEFCLASS [Електронний ресурс] / Nauck D.// ScienceDirect. – 2003. – Режим доступу до ресурсу: <https://www.sciencedirect.com/science/article/pii/S0888613X02000798>.

# ДОДАТОК А. ЛІСТИНГ

Весь код доступний на <https://github.com/litvinich/mtoi_coursework>

Файл src/nefclass.py

import numpy as np

from .membership import \*

EPSILON = 0.0000001

class NefClassModel:

def \_\_init\_\_(

self, num\_input\_units, num\_fuzzy\_sets, kmax, output\_units, universe\_max, universe\_min, membership\_type

):

self.input = \_input\_layer(num\_input\_units, num\_fuzzy\_sets, universe\_max, universe\_min, membership\_type)

self.rule = \_rule\_layer(kmax, output\_units)

self.output = \_output\_layer(output\_units)

self.universe\_max = universe\_max

self.universe\_min = universe\_min

self.membership\_type = membership\_type

def init\_fuzzy\_sets(self, abcs):

self.input.init\_abcs(abcs)

def \_\_call\_\_(self, x, t):

m, ante = self.input(x)

o = self.rule(m)

c = self.output(o)

return c

def learn\_rule(self, x, t):

m, ante = self.input(x)

o = self.rule.learn(ante, t)

# c = self.output(o)

def update\_fuzzy\_sets(self, sigma, delta):

# print(self.input.abcs)

for n in self.rule.nodes:

interm = n.update\_fuzzy\_set\_node(delta)

if interm is not None:

self.input.update\_fuzzy\_sets(sigma, interm)

# print(self.input.abcs)

def get\_num\_rules(self):

return len(self.rule.nodes)

def get\_antecedents(self, x):

m = []

for i in range(len(x)):

m.append(

[

determine\_membership(x[i], v, self.universe\_max[i], self.universe\_min[i], self.membership\_type)

for k, v in self.input.abcs[i].items()

]

)

ante = [mem.index(max(mem)) for mem in m]

return m, ante

def get\_degree\_of\_fulfilment(self, m, a):

activations = [m[i][a[i]] + EPSILON for i in range(len(a))]

min\_activation = min(activations)

return min\_activation

def add\_rules(self, antecedents, consequents):

for a, c in zip(antecedents, consequents):

self.rule.\_create\_node(a, c)

def predict(self, data: np.ndarray, targets: np.ndarray) -> np.ndarray:

predicts = []

for features, target in zip(data, targets):

output = self.\_\_call\_\_(features, target)

predicts.append(np.argmax(output))

return np.array(predicts)

class \_input\_layer:

def \_\_init\_\_(self, num\_input\_units, num\_fuzzy\_sets, universe\_max, universe\_min, membership\_type):

self.num\_fuzzy\_sets = num\_fuzzy\_sets

self.num\_input\_units = num\_input\_units

self.abcs = None

self.last\_m = None

self.last\_ante = None

self.last\_input = None

self.universe\_max = universe\_max

self.universe\_min = universe\_min

self.membership\_type = membership\_type

def init\_abcs(self, abcs):

self.abcs = abcs

def \_\_call\_\_(self, x):

self.last\_input = x

m = []

for i in range(len(x)):

m.append(

[

determine\_membership(x[i], v, self.universe\_max[i], self.universe\_min[i], self.membership\_type)

for k, v in self.abcs[i].items()

]

)

ante = [mem.index(max(mem)) for mem in m]

self.last\_m = m

self.last\_ante = ante

return m, ante

def update\_fuzzy\_sets(self, sigma, interm):

error\_rule, (j1, j2), mu = interm

key = list(self.abcs[j1].keys())[j2]

abc = self.abcs[j1][key]

# print(abc)

delta\_b = sigma \* error\_rule \* (abc[2] - abc[0]) \* np.sign(self.last\_input[j1] - abc[1])

delta\_a = -sigma \* error\_rule \* (abc[2] - abc[0]) + delta\_b

delta\_c = sigma \* error\_rule \* (abc[2] - abc[0]) + delta\_b

# print(delta\_a, delta\_b, delta\_c)

# update

new\_abc = [abc[0] + delta\_a, abc[1] + delta\_b, abc[2] + delta\_c]

# print(abc)

if self.check\_constraints(j1, key, new\_abc):

self.abcs[j1][key] = new\_abc

# else:

# print('constraints failed')

def check\_constraints(self, input\_node, key, new\_abc):

check1 = self.keep\_relative\_order(input\_node, key, new\_abc)

check2 = self.always\_overlap(input\_node, key, new\_abc)

check3 = self.symmetrical(input\_node, key, new\_abc)

return check1 and check2 and check3

def keep\_relative\_order(self, input\_node, key, new\_abc):

old\_sets = self.abcs[input\_node]

old\_sets[key] = new\_abc

bs = [b for a, b, c in list(old\_sets.values())]

bs\_copy = bs[:]

bs\_copy.sort()

check = bs == bs\_copy

return check

def always\_overlap(self, input\_node, key, new\_abc):

old\_sets = self.abcs[input\_node]

old\_sets[key] = new\_abc

a\_s = [a for a, b, c in list(old\_sets.values())]

bs = [b for a, b, c in list(old\_sets.values())]

cs = [c for a, b, c in list(old\_sets.values())]

for n, c in enumerate(cs):

if c <= self.universe\_min[n]:

return False

if n + 1 != self.num\_fuzzy\_sets:

if c < a\_s[n + 1]:

return False

for n, a in enumerate(a\_s):

if a >= self.universe\_max[n]:

return False

return True

def symmetrical(self, input\_node, key, new\_abc):

old\_sets = self.abcs[input\_node]

old\_sets[key] = new\_abc

for v in old\_sets.values():

a, b, c = v

if not np.allclose(c - b, b - a):

return False

return True

class \_rule\_layer:

def \_\_init\_\_(self, kmax, output\_units):

self.kmax = kmax

self.output\_units = output\_units

self.nodes = []

self.antes = []

def \_\_call\_\_(self, m):

tally = [[] for i in range(self.output\_units)]

for n in self.nodes:

tally = n(m, tally)

return tally

def learn(self, antecedent, consequent):

if len(self.nodes) < self.kmax:

if str(antecedent) not in self.antes:

self.\_create\_node(antecedent, consequent)

self.antes.append(str(antecedent))

# print(len(self.nodes), len(self.antes))

def \_create\_node(self, antecedent, consequent):

self.nodes.append(RuleNode(antecedent, consequent, self.output\_units))

class RuleNode:

def \_\_init\_\_(self, antecedent, consequent, output\_units):

self.antecedent = antecedent

self.consequent = consequent

self.output\_units = output\_units

self.last\_activation = None

self.last\_min\_activation = None

# each rule is connected to exactly 1 output (consequent)

def \_\_call\_\_(self, m, tally):

# min as tnorm

activations = [m[i][self.antecedent[i]] + EPSILON for i in range(len(self.antecedent))]

self.last\_activation = activations

min\_activation = min(activations)

self.last\_min\_activation = min\_activation

tally[self.consequent].append(min\_activation)

return tally

def update\_fuzzy\_set\_node(self, delta):

if self.last\_min\_activation > EPSILON:

error\_rule = self.last\_min\_activation \* (1 - self.last\_min\_activation) \* (delta[self.consequent])

j = np.argmin(self.last\_activation)

mu = self.last\_activation[j]

return error\_rule, (j, self.antecedent[j]), mu

else:

return None

class \_output\_layer:

def \_\_init\_\_(self, output\_units):

self.output\_units = output\_units

def \_\_call\_\_(self, o):

# o is tally

# max as t-conorm

output = [max(node) if len(node) != 0 else 0 for node in o]

# print(output)

total = sum(output)

# print(total)

output = [o / total for o in output]

return output

Файл src/membership.py

import numpy as np

def determine\_membership(x, abc, max, min, type="tri"):

assert len(abc) == 3

a, b, c = abc

if x <= min or x >= max:

return 1

if type == "tri":

if a < x and x <= b:

return (x - a) / (b - a)

elif b < x and x < c:

return (c - x) / (c - b)

else:

return 0

elif type == "gaussian":

return a \* np.exp(-((x - b) \*\* 2) / (2 \* c \*\* 2))

elif type == "semicircle":

r = c - b

y = np.sqrt(r \*\* 2 - (x - b) \*\* 2)

return y

else:

print("mf type not supported")

assert False

def build\_membership\_function(universe, labels):

num\_labels = len(labels)

assert num\_labels % 2 == 1

limits = [np.min(universe), np.max(universe)]

universe\_range = limits[1] - limits[0]

widths = [universe\_range / ((num\_labels - 1) / 2.0)] \* int(num\_labels)

centers = np.linspace(limits[0], limits[1], num\_labels)

abcs = [[c - w / 2, c, c + w / 2] for c, w in zip(centers, widths)]

dic = {}

for label, abc in zip(labels, abcs):

dic[label] = abc

return dic

def \_trimf(x, abc):

assert len(abc) == 3

a, b, c = np.r\_[abc]

assert a <= b and b <= c

y = np.zeros(len(x))

if a != b:

idx = np.nonzero(np.logical\_and(a < x, x < b))[0]

y[idx] = (x[idx] - a) / float(b - a)

if b != c:

idx = np.nonzero(np.logical\_and(b < x, x < c))[0]

y[idx] = (c - x[idx]) / float(c - b)

idx = np.nonzero(x == b)

y[idx] = 1

return y

Файл src/train\_utils.py

import copy

from collections import defaultdict

from typing import Any, Callable, Dict, List, Tuple

import numpy as np

from sklearn.metrics import f1\_score

from sklearn.model\_selection import StratifiedKFold

from tqdm import tqdm

from .membership import build\_membership\_function

from .nefclass import NefClassModel

from .preprocess\_utils import preprocess\_with\_knn\_imputer\_minmax\_scaler

seed = 8

def train\_nefclass(

model\_params: Dict[str, Any],

train\_data: np.ndarray,

train\_targets: np.ndarray,

test\_data: np.ndarray,

test\_targets: np.ndarray,

universe\_max: np.ndarray,

universe\_min: np.ndarray,

metric\_to\_maximize: Callable = f1\_score,

tqdm\_learning: bool = True,

) -> Tuple[NefClassModel, float, float]:

model = NefClassModel(

num\_input\_units=model\_params["num\_input\_units"],

num\_fuzzy\_sets=model\_params["num\_sets"],

kmax=model\_params["kmax"],

output\_units=model\_params["output\_units"],

universe\_max=universe\_max,

universe\_min=universe\_min,

membership\_type=model\_params["mf"],

)

# linguistic variables

labels = [f"label\_{index}" for index in range(model\_params["num\_sets"])]

abcs = [

build\_membership\_function(train\_data[:, feature\_index], labels) for feature\_index in range(train\_data.shape[1])

]

model.init\_fuzzy\_sets(abcs)

# train rules

for features, target in zip(train\_data, train\_targets):

model.learn\_rule(features, target)

best\_metric\_dict: Dict[str, Any] = dict(value=0, epoch=0, model=model)

metrics: Dict[str, List[float]] = dict(train=[], test=[])

# train fuzzy sets

epoch\_bar = tqdm(range(model\_params["num\_epoch"])) if tqdm\_learning else range(model\_params["num\_epoch"])

for epoch in epoch\_bar:

for features, target in zip(train\_data, train\_targets):

output = model(features, target)

delta = [

1 - output[class\_index] if class\_index == target else 0 - output[class\_index]

for class\_index in range(len(output))

]

model.update\_fuzzy\_sets(model\_params["sigma"], delta)

metrics["train"].append(

metric\_to\_maximize(y\_true=train\_targets, y\_pred=model.predict(train\_data, train\_targets))

)

metrics["test"].append(metric\_to\_maximize(y\_true=test\_targets, y\_pred=model.predict(test\_data, test\_targets)))

this\_epoch\_test\_metric = metrics["test"][-1]

if this\_epoch\_test\_metric >= best\_metric\_dict["value"]:

best\_metric\_dict["value"] = this\_epoch\_test\_metric

best\_metric\_dict["epoch"] = epoch

best\_metric\_dict["model"] = copy.deepcopy(model)

# early stopping

if epoch - best\_metric\_dict["epoch"] > min(model\_params["num\_epoch"] / 5, 10):

break

best\_value = best\_metric\_dict["value"]

if tqdm\_learning:

epoch\_bar.set\_description(f"Current: {this\_epoch\_test\_metric:.4f}; Best: {best\_value:.4f}")

return (

best\_metric\_dict["model"],

metrics["train"][best\_metric\_dict["epoch"]],

metrics["test"][best\_metric\_dict["epoch"]],

)

def cross\_validation\_train\_neflcass(

model\_params: Dict[str, Any],

data: np.ndarray,

target: np.ndarray,

preprocess\_func: Callable = preprocess\_with\_knn\_imputer\_minmax\_scaler,

metric\_to\_maximize: Callable = f1\_score,

metrics\_to\_save: Tuple[Tuple[str, Callable], ...] = (("f1", f1\_score),), # type: ignore

folds: int = 5,

return\_full\_dict: bool = False,

tqdm\_learning: bool = True,

) -> Dict[str, float]:

stratified\_kfold = StratifiedKFold(n\_splits=folds, random\_state=seed, shuffle=True)

metrics: Dict[str, Any] = defaultdict(list)

for train\_index, test\_index in stratified\_kfold.split(data, target):

train\_data, train\_targets = data[train\_index], target[train\_index]

test\_data, test\_targets = data[test\_index], target[test\_index]

train\_data, test\_data = preprocess\_func(train\_data=train\_data, test\_data=test\_data)

model\_params["num\_input\_units"] = train\_data.shape[1]

model\_params["output\_units"] = len(np.unique(train\_targets))

model, best\_train, best\_test = train\_nefclass(

model\_params=model\_params,

train\_data=train\_data,

train\_targets=train\_targets,

test\_data=test\_data,

test\_targets=test\_targets,

universe\_max=np.max(train\_data, axis=0),

universe\_min=np.min(train\_data, axis=0),

metric\_to\_maximize=metric\_to\_maximize,

tqdm\_learning=tqdm\_learning,

)

for (metric\_name, metric\_func) in metrics\_to\_save:

# metric\_name, metric\_func = metric\_tuple

metrics[metric\_name + "\_train"].append(

metric\_func(y\_true=train\_targets, y\_pred=model.predict(train\_data, train\_targets))

)

metrics[metric\_name + "\_test"].append(

metric\_func(y\_true=test\_targets, y\_pred=model.predict(test\_data, test\_targets))

)

metrics = {key: np.mean(value) for key, value in metrics.items()}

if return\_full\_dict:

metrics.update(model\_params)

return metrics

Файл src/preprocess\_utils.py

from typing import Tuple

import numpy as np

from sklearn.impute import KNNImputer

from sklearn.preprocessing import MinMaxScaler

def preprocess\_with\_knn\_imputer\_minmax\_scaler(

train\_data: np.ndarray, test\_data: np.ndarray, n\_neighbors: int = 5,

) -> Tuple[np.ndarray, np.ndarray]:

imputer = KNNImputer(n\_neighbors=n\_neighbors)

train\_data\_without\_nans = imputer.fit\_transform(train\_data)

test\_data\_without\_nans = imputer.transform(test\_data)

min\_max\_scaler = MinMaxScaler()

train\_data\_without\_nans\_scaled = min\_max\_scaler.fit\_transform(train\_data\_without\_nans)

test\_data\_without\_nans\_scaled = min\_max\_scaler.transform(test\_data\_without\_nans)

return train\_data\_without\_nans\_scaled, test\_data\_without\_nans\_scaled

Файл grid\_search.py

import multiprocessing as mp

from collections import defaultdict

from functools import partial

from itertools import product

from typing import NoReturn

import pandas as pd

from sklearn.metrics import accuracy\_score, f1\_score

from tqdm import tqdm

from src.preprocess\_utils import preprocess\_with\_knn\_imputer\_minmax\_scaler

from src.train\_utils import cross\_validation\_train\_neflcass

def main() -> NoReturn:

data = pd.read\_csv("data/preprocessed\_data.csv", header=0, index\_col=0)

grid\_sigma = [0.001, 0.01, 0.1]

grid\_num\_epoch = [15, 25, 50]

grid\_num\_sets = [3, 5, 7]

grid\_kmax = [5, 10, 25, 50]

grid\_membership = ["tri", "gaussian", "semicircle"]

all\_model\_params = []

for sigma, num\_epoch, num\_sets, kmax, membership in product(

grid\_sigma, grid\_num\_epoch, grid\_num\_sets, grid\_kmax, grid\_membership

):

all\_model\_params.append(dict(sigma=sigma, num\_epoch=num\_epoch, num\_sets=num\_sets, kmax=kmax, mf=membership))

with mp.Pool(processes=min(len(all\_model\_params), 24)) as pool:

function = partial(

cross\_validation\_train\_neflcass,

data=data.iloc[:, 1:].values,

target=data.iloc[:, 0].values.astype(int),

preprocess\_func=preprocess\_with\_knn\_imputer\_minmax\_scaler,

metric\_to\_maximize=f1\_score,

metrics\_to\_save=(("f1", f1\_score), ("accuracy", accuracy\_score)),

folds=5,

return\_full\_dict=True,

tqdm\_learning=False,

)

results = list(tqdm(pool.imap(function, all\_model\_params), total=len(all\_model\_params)))

results\_for\_dataframe = defaultdict(list)

for result in results:

for key, value in result.items():

results\_for\_dataframe[key].append(value)

results\_df = pd.DataFrame(results\_for\_dataframe)

results\_df.to\_csv("results/baseline.csv", header=True, index=True)

if \_\_name\_\_ == "\_\_main\_\_":

main()