НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК 004.023

«До захисту допущено»
Завідувач кафедри СПСКС

_____Віталій РОМАНКЕВИЧ
(підпис) (ім'я, прізвище)

"___"_____2020р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія
Комп'ютерні системи та компоненти

на тему: Алгоритм глибинного аналізу даних для задачі класифікації на основі штучного бджолиного рою

Виконав: студент ІІ курсу, групи КВ-93мп
Абдураімов Таір Заірович
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник доцент, к.т.н., Юрій ЗОРІН
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант з нормоконтролю доцент, с.н.с., к.т.н. Юлія БОЯРІНОВА
_____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп‘ютерних систем

Рівень вищої освіти – другий (магістерський)

Спеціальність 123 комп‘ютерна інженерія

Комп‘ютерні системи та компоненти

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
_____ Віталій РОМАНКЕВИЧ
(підпис)

2019р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Абдураімову Таіру Заіровичу

1. Тема дисертації: алгоритм глибинного аналізу даних для задачі класифікації на основі штучного бджолиного рою, науковий керівник дисертації к.т.н., доцент Зорін Юрій Михайлович, затверджені наказом по університету від «12» листопада 2020 р. №3298-с

2. Термін подання студентом дисертації 07 грудня 2020 р.

3. Об‘єкт дослідження: глибинний аналіз даних для задачі класифікації.

4. Предмет дослідження: алгоритм штучного бджолиного рою для задачі класифікації.

5. Перелік завдань, які потрібно розробити
   – аналіз існуючих рішень для задачі класифікації;
   – алгоритм бджолиного рою для задачі класифікації;
   – представлення і аналіз результатів розробленого алгоритму;

6. Перелік ілюстративного матеріалу: презентація і 38 рисунків.

7. Перелік публікацій
   – «Модифікований мурашиний алгоритм для розв'язання задачі глобальної оптимізації», XII конференція молодих вчених ПМК-2019. – 2019;
   – «Модифікований мурашиний алгоритм для задачі комівояжера», XIII конференція молодих вчених ПМК-2020. – 2020;

8. Дата видачі завдання 5 вересня 2019 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вибір тематики роботи	10.09.2019	
2	Вивчення літератури за обраною тематикою	10.01.2020	
3	Постановка задачі для магістерської дисертації	01.02.2020	
4	Аналіз існуючих рішень для поставленої задачі	01.05.2020	
5	Підготовка першого розділу магістерської дисертації	10.09.2020	
6	Підготовка другого розділу магістерської дисертації	01.10.2020	
7	Підготовка третього розділу магістерської дисертації	10.10.2020	
8	Підготовка четвертого розділу магістерської дисертації	01.11.2020	
9	Підготовка графічної частини	10.11.2020	
10	Оформлення документації магістерської дисертації	20.11.2020	
11	Попередній розгляд магістерської дисертації на кафедрі	25.11.2020	

Студент _____ Таір АБДУРАІМОВ
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ Юрій ЗОРІН
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Актуальність теми. Оскільки розмір цифрової інформації зростає в геометричній прогресії, потрібно витягувати великі обсяги необроблених даних. На сьогоднішній день існує кілька методів налаштування та обробки даних відповідно до наших потреб. Найбільш поширеним методом є використання інтелектуального аналізу даних (Data Mining). Data Mining застосовується для вилучення неявних, дійсних та потенційно корисних знань із великих обсягів необроблених даних. Видобуті знання повинні бути точними, читабельними та легкими для розуміння. Крім того, процес видобутку даних також називають процесом виявлення знань, який використовувався в більшості нових міждисциплінарних областей, таких як бази даних, статистика штучного інтелекту, візуалізація, паралельні обчислення та інші галузі.

Одним із нових і надзвичайно потужних алгоритмів, що використовуються в Data Mining, є еволюційні алгоритми та підходи, що базуються на рії, такі як мурашиний алгоритм та оптимізація рою частинок. В даній роботі запропоновано використати для інтелектуального аналізу даних досить нову ідею алгоритма бджолиного рою для широко розповсюдженої задачі класифікації.

Мета роботи: покращення результатів класифікації даних в сенсі в точності і сталості за допомогою алгоритму інтелектуального аналізу даних на основі алгоритму бджолиного рою.

Об’єктом дослідження є процес інтелектуального аналізу даних для задачі класифікації.

Предметом дослідження є використання алгоритму бджолиного рою для інтелектуального аналізу даних.

Методи дослідження. Використовуються методи параметричного дослідження евристичних алгоритмів, а також методи порівняльного аналізу для алгоритмів інтелектуального аналізу даних.

Наукова новизна одержаних результатів роботи полягає в тому, що після проведеного аналізу існуючих рішень, запропоновано використати алгоритм бджолиного рою для задачі класифікації, точність і сталість якого перевищує показники існуючих класифікаторів.

Практичне значення одержаних результатів полягає в тому, що розроблений алгоритм показує кращі результати в сенсі точності і сталості в порівнянні з іншими алгоритмами інтелектуального аналізу даних. Тобто адаптація бджолиного алгоритму може розглядатися як корисне та точне рішення для такої важливої проблеми, як задача класифікації даних.

Апробація роботи. Основні положення й результати роботи були представлені та обговорювались на науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2019 (Київ, 2019 р.), а також на науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2020 (Київ, 2020 р.).

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів, висновків та додатків.

У вступі надано загальну характеристику роботи, виконано оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи, наведено відомості про апробацію результатів і їх впровадження.

У першому розділі розглянуто алгоритми інтелектуального аналізу даних, які використовуються для задачі класифікації. Обґрунтовано можливість використання евристичних алгоритмів, а саме алгоритму бджолиного рою для цієї задачі.

У другому розділі детально розглянуто алгоритм бджолиного рою та принципи його роботи, також описано запропоновану методику його застосування для інтелектуального аналізу даних, а саме для задачі класифікації.

У третьому розділі описано розроблений алгоритм та програмний додаток, в якому він реалізований.

У четвертому розділі приведена оцінка ефективності запропонованого алгоритм, на основі тестування алгоритму, а також порівняльного аналізу між розробленим алгоритмом та вже існуючими.

У висновках представлені результати магістерської дисертації.

Робота виконана на 81 аркуші, містить посилання на список використаних літературних джерел з 18 найменувань. У роботі наведено 38 рисунків та 5 додатків.

Ключові слова: data mining, класифікація, евристичні алгоритми, оптимізація рою частинок, мурашиний алгоритм, алгоритм бджолиного рою.

ABSTRACT

Actuality of theme. As the size of digital information grows exponentially, large amounts of raw data need to be extracted. To date, there are several methods to customize and process data according to our needs. The most common method is to use Data Mining. Data Mining is used to extract implicit, valid and potentially useful knowledge from large amounts of raw data. The knowledge gained must be accurate, readable and easy to understand. In addition, the data mining process is also called the knowledge discovery process, which has been used in most new interdisciplinary fields, such as databases, artificial intelligence statistics, visualization, parallel computing, and other fields.

One of the new and extremely powerful algorithms used in Data Mining is evolutionary algorithms and swarm-based approaches, such as the ant algorithm and particle swarm optimization. In this paper, it is proposed to use a fairly new idea of the swarm of bee swarm algorithm for data mining for a widespread classification problem.

Purpose: to develop an algorithm for data mining for the classification problem based on the swarm of bee swarms, which exceeds other common classifiers in terms of accuracy of results and consistency.

The object of research is the process of data mining for the classification problem.

The subject of the study is the use of a swarm of bee swarms for data mining.

Research methods. Methods of parametric research of heuristic algorithms, and also methods of the comparative analysis for algorithms of data mining are used.

The scientific novelty of the work is as follows:

1. As a result of the analysis of existing solutions for the classification problem, it is decided to use such metaheuristics as the swarm of bee swarm.

2. The implementation of the bee algorithm for data mining is proposed.

The practical value of the results obtained in this work is that the developed algorithm can be used as a classifier for data mining. In addition, the proposed adaptation of the bee algorithm can be considered as a useful and accurate solution to such an important problem as the problem of data classification.

Approbation of work. The main provisions and results of the work were presented and discussed at the scientific conference of undergraduates and graduate students "Applied Mathematics and Computing" PMK-2019 (Kyiv, 2019), as well as at the scientific conference of undergraduates and graduate students "Applied Mathematics and Computing" PMK-2020 (Kyiv, 2020).

Structure and scope of work. The master's dissertation consists of an introduction, four chapters, conclusions and appendices.

The introduction provides a general description of the work, assesses the current state of the problem, substantiates the relevance of research, formulates the purpose and objectives of research, shows the scientific novelty of the results and the practical value of the work, provides information on testing and implementation.

The first section discusses the data mining algorithms used for the classification problem. The possibility of using heuristic algorithms, namely the bee swarm algorithm for this problem, is substantiated.

The second section discusses in detail the algorithm of the bee swarm and the principles of its operation, also describes the proposed method of its application for data mining, namely for the classification problem.

The third section describes the developed algorithm and the software application in which it is implemented.

In the fourth section the estimation of efficiency of the offered algorithm, on the basis of testing of algorithm, and also the comparative analysis between the developed algorithm and already different is resulted.

The conclusions present the results of the master's dissertation.

The work is performed on 89 sheets, contains a link to the list of used literature sources with 18 titles. The paper presents 38 figures and 2 appendices.

Keywords: data mining, classification, heuristic algorithms, particle swarm optimization, ant algorithm, bee swarm algorithm.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

ABC (artificial bee colony) – алгоритм бджолиного рою (колонії).

ACO (ant colony optimization) – мурашиний алгоритм.

ANOVA (analysis of variance) – дисперсійний аналіз.

ANN (Artificial neural networks) – штучні нейронні мережі.

CNN (Convolutional Neural Networks) – згорткові нейронні мережі.

DM (Data Mining) – інелкетуальний аналіз даних.

DT (Decision tree) – дерево рішень.

ID3 (Iterative Dichotomiser 3) – алгоритм дерева рішень для класифікації.

KDD (knowledge discovery in databases) – виявлення знань у базах даних.

KNN (K-Nearest Neighbor) – алгоритм k-найближчих сусідів.

LDA (Linear discriminant analysis) – лінійний дискримінант Фішера (або Дискрімінантний аналіз).

LSTM (Long Short-Term Memory) – Довга короткочасна пам'ять.

MANOVA (multivariate analysis of variance) – багатовимірний дисперсійний аналіз.

MAP (maximum a posteriori probability) – оцінка максимуму апостеріорної ймовірності.

PSO (particle swarm optimization) – метод рою часток.

RCNN (Recurrent Convolutional Neural Networks) – Повторювані згорткові нейронні мережі.

RF (Random forest) – випадковий ліс.

RNN (Recurrent Neural Networks) – Періодичні нейронні мережі.

TDIDT (top down induction of decision trees) – індукція дерев рішень зверху вниз.

SVM (support vector machines) – метод опорних векторів.

SOM (self-organizing map) – самоорганізаційна карта Кохонена.

QDA (Quadratic discriminant analysis) – Квадратичний дискрімінантний аналіз.

ВСТУП

Інтелектуальний аналіз даних (Data Mining, DM) - це процес виявлення закономірностей у великих наборах даних, що включають методи на стику машинного навчання, статистики та систем баз даних. Data Mining є міждисциплінарним підполем інформатики та статистики з загальною метою вилучення інформації (за допомогою інтелектуальних методів) з набору даних та перетворення інформації в зрозумілу структуру для подальшого використання. DM є етапом аналізу процесу "виявлення знань у базах даних" (knowledge discovery in databases, KDD). Окрім етапу необробленого аналізу, він також включає аспекти управління базами даних, попередню обробку даних, міркування щодо моделі та інтерфейсів, цікавості, складності, пост-обробку виявлених структур, візуалізацію та оновлення в Інтернеті.

Метою Data Mining є вилучення шаблонів та знань із великих обсягів даних. Це слово часто застосовується до будь-якої форми широкомасштабної обробки даних або інформації (збору, вилучення, складування, аналізу та статистики), а також до будь-якого застосування комп’ютерної системи підтримки прийняття рішень, включаючи штучний інтелект (наприклад, машинне навчання) та бізнес-інтелект.

Фактичним завданням видобутку даних є напівавтоматичний або автоматичний аналіз великої кількості даних для вилучення раніше невідомих, цікавих закономірностей, таких як групи записів даних (кластерний аналіз), незвичні записи (виявлення аномалії) та залежності (видобуток правил асоціації, послідовний вибірок шаблонів).

Класифікація - це функція аналізу даних, яка призначає елементи в колекції цільовим категоріям або класам. Мета класифікації - точно передбачити цільовий клас для кожного випадку в даних.

У термінології машинного навчання [1] класифікація вважається екземпляром контрольованого навчання, тобто навчання, де доступний навчальний набір правильно визначених спостережень. Відповідна непідконтрольна процедура відома як кластеризація і передбачає групування

даних за категоріями на основі певної міри притаманної подібності чи відстані.

Одним із нових та надзвичайно потужних алгоритмів, що використовуються в DM є метаевристичні алгоритми. У цій роботі пропонується використати основою для DM як класифікатор алгоритм бджолиного рою (Artificial Bee Colony, ABC), заснований на розумній поведінці колоній медоносних бджіл, який запропонував Д. Карабога в 2005 році [2].

В цій роботі алгоритм ABC було реалізовано для задачі класифікації та оцінено його ефективність, порівнюючи з іншими найбільш поширеними класифікаторами.

1. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ ДЛЯ ЗАДАЧІ КЛАСИФІКАЦІЇ ДАНИХ

1.1. Інтелектуальний аналіз даних і задача класифікації

Класифікація - це функція аналізу даних, яка призначає елементи в колекції цільовим категоріям або класам. Мета класифікації - точно передбачити цільовий клас для кожного випадку в даних. Наприклад, модель класифікації може бути використана для ідентифікації заявників позик як низьких, середніх або високих кредитних ризиків.

Завдання з класифікації починається з набору даних, у якому відомі призначення класу. Наприклад, модель класифікації, яка передбачає кредитний ризик, може бути розроблена на основі спостережуваних даних для багатьох заявників позики протягом певного періоду. На додаток до історичного кредитного рейтингу, дані можуть відстежувати історію зайнятості, власність чи оренду житла, роки проживання, кількість та тип інвестицій тощо. Кредитний рейтинг був би цільовим, інші атрибути були б предикторами, а дані для кожного клієнта становили б випадок.

Класифікації дискретні і не передбачають порядку. Безперервні значення з плаваючою крапкою означатимуть числову, а не категоричну ціль. Прогностична модель із числовою ціллю використовує алгоритм регресії, а не алгоритм класифікації.

Найпростіший тип класифікаційної задачі - це двійкова класифікація. У двійковій класифікації цільовий атрибут має лише два можливих значення: наприклад, високий кредитний рейтинг або низький кредитний рейтинг. Багатокласні цілі мають більше двох значень: наприклад, низький, середній, високий або невідомий кредитний рейтинг.

У процесі побудови моделі (навчання) алгоритм класифікації знаходить взаємозв'язок між значеннями предикторів та значеннями цілі. Різні класифікаційні алгоритми використовують різні техніки пошуку зв’язків. Ці зв’язки узагальнені у моделі, яка потім може бути застосована до іншого набору даних, у якому призначення класу невідомі.

Моделі класифікації перевіряються шляхом порівняння передбачуваних значень із відомими цільовими значеннями у наборі даних тесту. Історичні дані для класифікації зазвичай поділяються на два набори даних: один для побудови моделі; інший для тестування моделі.

Оцінка класифікаційної моделі призводить до виконання завдань та ймовірностей для кожного випадку. Наприклад, модель, яка класифікує клієнтів як низьку, середню або високу вартість, також передбачає ймовірність кожної класифікації для кожного клієнта.

Класифікація має багато застосувань у сегментації споживачів, бізнес-моделюванні, маркетингу, кредитному аналізі та біомедичному та моделюванні реакції на наркотики.

У наступних розділах розглянемо найпоширеніші класифікаційні алгоритми, їх переваги і недоліки.

1.2. Метод k-найближчих сусідів

Класифікатори K-Найближчих сусідів базуються на навчанні за аналогією. Навчальні екземпляри описуються n-мірними числовими атрибутами. Кожен зразок являє собою точку в n-мірному просторі. Таким чином, усі навчальні зразки зберігаються у n-вимірному просторі шаблону. Коли дається невідомий зразок, класифікатор k-найближчих сусідів здійснює пошук у просторі шаблонів для k навчальних зразків, які є найближчими до невідомої вибірки [3].

X=(x1,x2,…….,xn) and *Y=(y1,y2,….,yn)* is *d(X, Y)= 2*

$$d(X,Y) = \sqrt{\sum_{i=1}^{n} (\mathrm{x}_i - \mathrm{y}_i)^2}$$

"Близькість" визначається в термінах евклідової відстані, де евклідова відстань знаходиться між двома точками. На відміну від індукції дерева рішень та зворотного розповсюдження, класифікатори найближчих сусідів присвоюють однакову вагу кожному атрибуту. Це може спричинити

плутанину, коли в даних є багато нерелевантних атрибутів. Класифікатори найближчих сусідів також можуть бути використані для прогнозування, тобто повернення реального значення прогнозу для даної невідомої вибірки. У цьому випадку класифікатор повертає середнє значення дійсного значення, пов’язаного з *k* найближчими сусідами невідомої вибірки. Алгоритм k-найближчих сусідів є найпростішим з усіх алгоритмів машинного навчання. Об'єкт класифікується більшістю голосів сусідів, при цьому об'єкт віднесено до класу, найпоширенішого серед k найближчих сусідів. *k* - ціле додатне число, як правило, невелике. Якщо *k = 1*, то об’єкт просто присвоюється класу найближчого сусіда. У двійкових (двокласних) проблемах класифікації корисно вибрати k як непарне число, оскільки це дозволяє уникнути пов’язаних голосів. Той самий метод можна використовувати для регресії, просто присвоївши значення властивості для об'єкта середнім значенням його k найближчих сусідів. Це може бути корисним для зважування внесків сусідів, щоб ближчі сусіди зробили більший внесок у середнє, ніж більш віддалені.

Сусіди беруться з набору об'єктів, для яких відома правильна класифікація (або, у випадку регресії, вартість властивості). Це можна сприймати як навчальний набір для алгоритму, хоча явного кроку навчання не потрібно. Для того, щоб ідентифікувати сусідів, об’єкти представлені векторами позицій у багатовимірному просторі ознак. Зазвичай використовують евклідівську відстань, хоча інші міри відстані, такі як відстань Манхантана, в принципі можуть використовуватись замість цього.

Алгоритм k-найближчого сусіда чутливий до локальної структури даних. Невідомому екземпляру присвоюється найпоширеніший клас серед його найближчих сусідів. Коли *k = 1*, невідомому зразку присвоюється клас навчального зразка, який є найближчим до нього в просторі шаблонів. Найближчі класифікатори сусідів - це ледачі методи, що базуються на екземплярах, оскільки вони зберігають усі навчальні зразки і не створюють класифікатор, доки не потрібно буде класифікувати новий (без маркування) зразок. Це контрастує з нетерплячими методами навчання, такими як індукція

дерева рішень та зворотне поширення, які будують модель узагальнення до отримання нових зразків для класифікації. Ледачі учні можуть понести дорогі обчислювальні витрати, коли кількість потенційних сусідів (тобто збережених навчальних зразків), з якими можна порівняти дану малу вибірку, велика. Тому вони вимагають ефективних методів індексації. Очікувані ледачі методи навчання швидші на тренуванні, ніж нетерплячі методи. Коли використовується алгоритм k-найближчого сусіда, вибір відповідного значення *k* є важливим. Якщо значення *k* занадто мале, воно сприйнятливе до переозброєння та призведе до неправильної класифікації деяких традиційно простих для класифікації ситуацій.

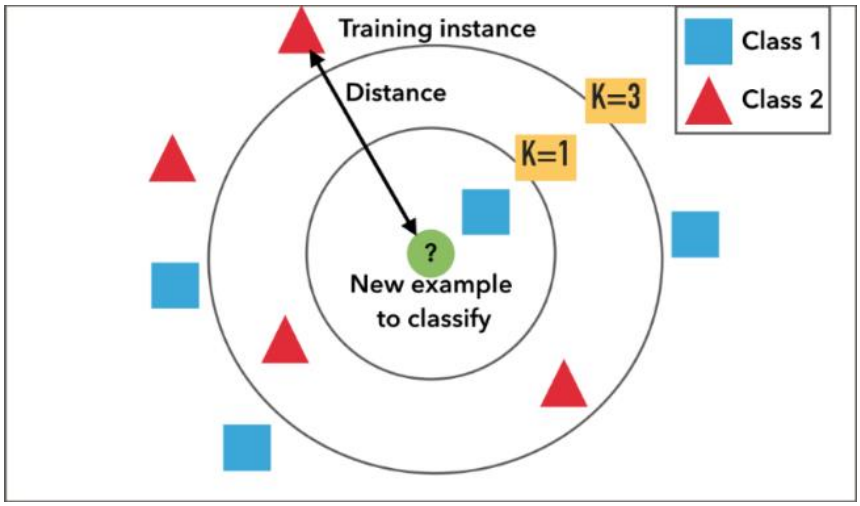


Рисунок 1.1 – Принцип роботи алгоритму k-найближчих сусідів.

Наприклад, уявімо кластер записів, у яких усі є мітка класу під назвою "квадрат", за винятком однієї точки в кластері, позначеної як " трикутник ". Якщо для вхідного сигналу, що знаходиться в кластері, було вибрано k з одиниці, але воно виявляється найближчим до трикутнику, тоді є велика ймовірність того, що ця точка була неправильно класифікована (рисунок 1.1). Це видно з того факту, що якби k було 2 або більше, класифікація в результаті була б іншою. Окрім занадто малого значення k, важливо вибрати значення, яке не є занадто великим, оскільки це також може призвести до неправильної класифікації. Це можна побачити в ситуації з кластером одного класу в оточенні кластера іншого класу. Навіть якщо введення знаходиться прямо в

середині першого кластера, якщо можливо розглянути занадто багато точок, воно також починає підраховувати записи із навколишнього кластера.

Отже, для алгоритму k-найближчих сусідів можна виділити наступні переваги:

- Ефективність для текстових наборів даних.
- Непараметричність.
- Розглядаються більш локальні характеристики даних.
- Природно обробляє багатокласні набори даних.

А також недоліки:

- Обчислення цієї моделі дуже дорогі.
- Складно знайти оптимальне значення k.
- Обмеження для проблеми пошуку для виявлення найближчих сусідів.
- Знайти значущу функцію відстані важко для наборів текстових даних.

1.3. Лінійний дискримінантний аналіз

Оригінальний дихотомічний дискримінантний аналіз був розроблений сером Рональдом Фішером у 1936 р. Він відрізняється від ANOVA (дисперсійний аналіз) або MANOVA (багатовимірний дисперсійний аналіз), який використовується для прогнозування однієї (ANOVA) або множинної (MANOVA) безперервних залежних змінних за допомогою однієї або декількох незалежних категоріальних змінних. Аналіз дискримінантної функції корисний для визначення того, чи ефективний набір змінних для прогнозування приналежності до категорії.

Лінійний дискримінантний аналіз (LDA) найчастіше використовується як техніка зменшення розмірності на етапі попередньої обробки для класифікації шаблонів та програм машинного навчання. Метою є проектування набору даних на простір нижчих розмірів з хорошою роздільністю класів, щоб уникнути надмірності ("прокляття розмірності"), а

також зменшити обчислювальні витрати. На рисунку 1.2 можно побачити основний принцип роботи алгоритму.

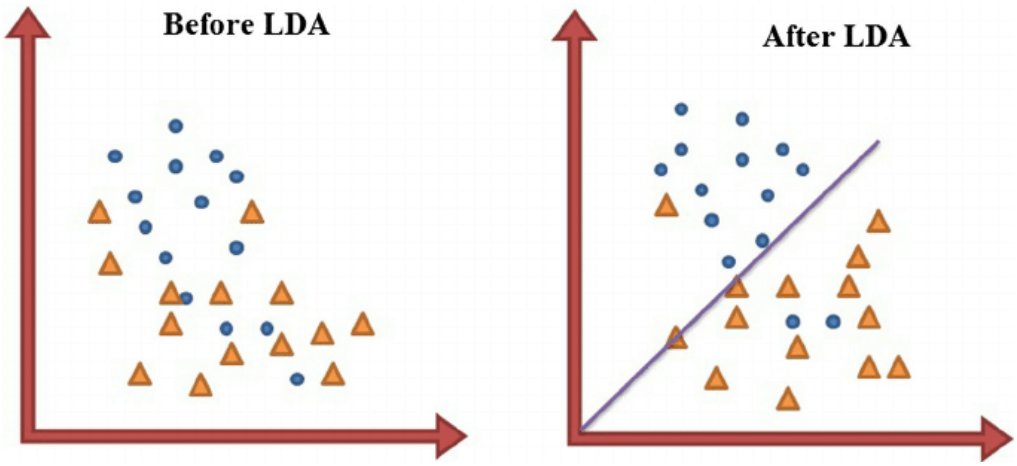


Рисунок 1.2 – Принцип роботи алгоритму LDA.

Розглянемо комплекс спостережень $\vec{x}$ (їх також називають ознаками, атрибутами, змінними або вимірами) для кожного зразка об'єкта чи події з відомим класом y. Цей набір зразків називається навчальним набором . Потім проблема класифікації полягає у пошуку хорошого провісника для класу y з будь-якої вибірки з однаковим розподілом (не обов'язково з навчального комплексу) з урахуванням лише спостереження $\vec{x}$ [4].

LDA підходить до проблеми, припускаючи, що функціонує умовна щільність ймовірності $p(\vec{x}|y=0)$ і $p(\vec{x}|y=1)$ яка нормально розподіляється із середнім значенням і параметрами коваріації $(\overrightarrow{\mu_0},\ S_0)$ і $(\overrightarrow{\mu_1},\ S_1)$ відповідно. Згідно з цим припущенням, оптимальним рішенням Байєса є прогнозування балів, що належать до другого класу, якщо журнал коефіцієнтів правдоподібності більший за деякий поріг T, так що:

$$(\overrightarrow{x_0} - \overrightarrow{\mu_0})^T \sum_0^{-1} (\overrightarrow{x_0} - \overrightarrow{\mu_0}) + \ln|S_0| - (\vec{x} - \overrightarrow{\mu_1})^T \sum_1^{-1} (\vec{x} - \overrightarrow{\mu_1}) - \ln|S_1| > T$$

Без будь-яких подальших припущень, отриманий класифікатор називається QDA (квадратичний дискримінантний аналіз).

Натомість LDA робить додаткове спрощення припущення про гомосцедастичність (тобто коваріантності класу ідентичні, тому $S_0 = S_1 = S$) і що коваріації мають повний ранг. У цьому випадку скасовується кілька термінів:

$$\vec{x}^T \sum\nolimits_0^{-1} \vec{x} = \vec{x}^T \sum\nolimits_0^{-1} \vec{x}$$

$\vec{x}^T \sum_0^{-1} \overrightarrow{\mu_i} = \overrightarrow{\mu_i}^T \sum_0^{-1} \vec{x}$, оскільки S_1 це ермітова матриця

І вищевказаний критерій рішення стає пороговим значенням для скалярного добутку:

$$\vec{\omega} \cdot \vec{x} > c$$

для деякої порогової константи c , де

$$\vec{\omega} = S^{-1}(\overrightarrow{\mu_1} - \overrightarrow{\mu_0})$$

$$c = \vec{\omega} \cdot \frac{1}{2}(\overrightarrow{\mu_1} + \overrightarrow{\mu_0})$$

Це означає, що критерій введення $\vec{x}$ перебування в класі y є суто функцією цієї лінійної комбінації відомих спостережень.

Часто корисно бачити цей висновок у геометричному плані: критерій введення $\vec{x}$ перебування в класі y є суто функцією проекції багатовимірної просторової точки $\vec{x}$ на вектор $\vec{\omega}$ (таким чином, ми розглядаємо лише його напрямок). Іншими словами, спостереження належить y якщо відповідає $\vec{x}$ знаходиться на певній стороні гіперплощини, перпендикулярної до $\vec{\omega}$. Розташування площини визначається порогом c.

Припущення щодо дискримінантного аналізу такі ж, як і для MANOVA. Аналіз досить чутливий до різних випадків, і розмір найменшої групи повинен бути більшим, ніж кількість змінних предикторів.

- Багатовимірна нормальність: Незалежні змінні є нормальними для кожного рівня змінної групування.
- Однорідність дисперсії / коваріації (гомосцедастичність): Варіації між груповими змінними однакові на рівнях предикторів. Однак лінійний дискримінантний аналіз використовується, коли коваріанції рівні, і квадратичний дискримінантний аналіз можна використовувати, коли коваріанти не рівні.
- Мультиколінеарність: Прогностична сила може зменшуватися із збільшенням кореляції між змінними предиктора.

- Незалежність: вважається, що для учасників проводиться вибіркова вибірка, а оцінка учасника за однією змінною вважається незалежною від оцінок за цією зміною для всіх інших учасників.

Дискримінантний аналіз є відносно стійким до незначних порушень цих припущень, а також LDA все ще може бути надійним при використанні дихотомічних змінних (де багатовимірна нормальність часто порушується).

LDA працює, коли вимірювання, проведені на незалежних змінних для кожного спостереження, є безперервними величинами. При роботі з категоріальними незалежними змінними еквівалентним методом є аналіз дискримінантної відповідності.

Дискримінантний аналіз використовується, коли групи відомі апріорі (на відміну від кластерного аналізу). Кожен випадок повинен мати оцінку за одним або кількома показниками кількісного прогнозування та оцінку за груповою оцінкою. Простіше кажучи, дискримінантним аналізом функцій є класифікація - акт розподілу речей по групах, класах або категоріях одного типу.

1.4. Наївний баєсів класифікатор

Naive Bayes - це проста техніка побудови класифікаторів: моделі, які присвоюють мітки класів екземплярам проблем, представленим у вигляді векторів значень ознак, де мітки класів витягуються з деякого кінцевого набору. Існує не єдиний алгоритм навчання таких класифікаторів, а сімейство алгоритмів, заснованих на загальному принципі: всі наївні класифікатори Байєса припускають, що значення певної ознаки не залежить від значення будь-якої іншої ознаки, враховуючи змінну класу. Наприклад, фруктом можна вважати яблуко, якщо воно червоне, кругле і має діаметр близько 10 см. Наївний класифікатор Байєса вважає, що кожна з цих особливостей незалежно сприяє ймовірності того, що цей фрукт є яблуком, незалежно від будь-якої кореляція між кольором, округлістю та особливостями діаметра.

Для деяких типів ймовірнісних моделей наївні класифікатори Байєса можна дуже ефективно навчити в контрольованому навчальному середовищі. У багатьох практичних додатках для оцінки параметрів наївних моделей Байєса використовується метод максимальної ймовірності; іншими словами, можна працювати з наївною моделлю Байєса, не приймаючи байєсівської ймовірності або використовуючи будь-які байєсівські методи.

Незважаючи на наївний дизайн та, мабуть, спрощені припущення, наївні класифікатори Байєса працювали досить добре у багатьох складних реальних ситуаціях. У 2004 р. аналіз проблеми класифікації Байєса показав, що існують вагомі теоретичні причини очевидно неправдоподібної ефективності наївних класифікаторів Байєса. Тим не менше, всебічне порівняння з іншими алгоритмами класифікації в 2006 році показало, що класифікація Байєса перевершує інші підходи, такі як дерева рішень або випадкові ліси.

Перевагою наївного Байєса є те, що для оцінки параметрів, необхідних для класифікації, потрібна лише невелика кількість навчальних даних.

Абстрактно, наївний Байєс є умовною моделлю ймовірності: заданий екземпляр задачі, який потрібно класифікувати, представлений вектором $x = (x_1, \dots, x_n)$ показуючи деякі n ознак (незалежних змінних), і присвоює цьому екземпляру ймовірності [5]:

$$p(C_k|x_1, \dots, x_n)$$

Для кожного з K можливих результатів або класів C_k.

Проблема з наведеним вище формулюванням полягає в тому, що якщо кількість ознак n велика або якщо ознака може приймати велику кількість значень, то базування такої моделі на таблицях ймовірностей неможливо. Тому можна переформулювати модель, щоб зробити її більш доступною. Використовуючи теорему Байєса (рисунок 1.3), умовну ймовірність можна розкласти як:

$$p(C_k|x) = \frac{p(C_k)p(x|C_k)}{p(x)}$$

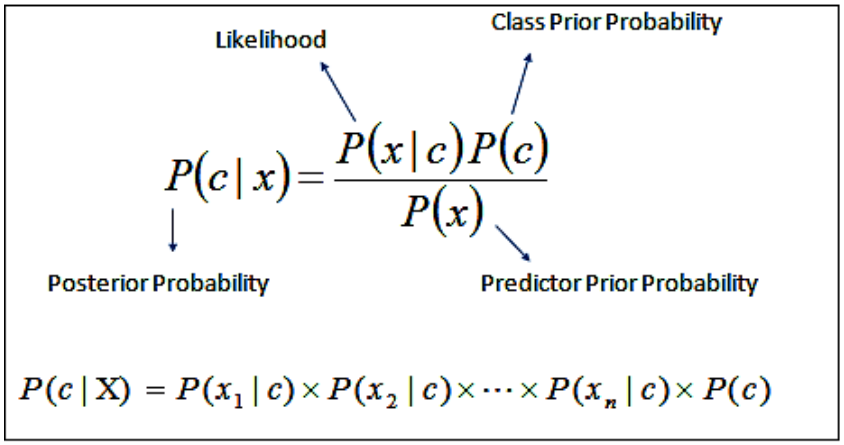


Рисунок 1.3 – Принцип теореми Байеса.

Чисельник можна представити наступним чином:

$$p(C_k|x_1, \ldots, x_n) = \frac{1}{Z} p(C_k) \prod_{i=1}^{n} p(x_i|C_k)$$

Де $Z = p(x) = \sum_k p(C_k)p(x|C_k)$ - коефіцієнт масштабування, що залежить лише від $x_1, \ldots, x_n$, тобто якщо значення змінних ознак відомі.

Наївний класифікатор Байєса поєднує цю модель із правилом прийняття рішення. Одним загальним правилом є вибір найбільш імовірної гіпотези; це відоме як максимальне правило апостеріорного або рішення щодо MAP (оцінка максимуму апостеріорної ймовірності). Відповідний Байєсів класифікатор є функцією, яка присвоює мітку класу $\hat{y} = C_k$ для деяких k наступним чином:

$$\hat{y} = \underset{k \in \{1, \ldots, K\}}{argmax} p(C_k) \prod_{i=1}^{n} p(x_i|C_k)$$

Незважаючи на те, що далекосяжні припущення про незалежність часто є неточними, наївний класифікатор Байєса має кілька властивостей, які роблять його напрочуд корисним на практиці. Зокрема, роз'єднання розподілів умовних ознак класу означає, що кожен розподіл може бути незалежно оцінений як одновимірний. Це допомагає полегшити проблеми, пов'язані з прокляттям розмірності, наприклад, необхідність наборів даних, які експоненційно масштабуються за кількістю функцій. У той час як наївний Байєс часто не може дати хорошої оцінки для правильних ймовірностей класу,

це не може бути вимогою для багатьох додатків. Наприклад, наївний класифікатор Байєса зробить правильну класифікацію правил прийняття MAP до тих пір, поки правильний клас є більш вірогідним, ніж будь-який інший клас. Це справедливо незалежно від того, чи є оцінка ймовірності незначною чи навіть вкрай неточною. Таким чином, загальний класифікатор може бути достатньо надійним, щоб ігнорувати серйозні недоліки в основі своєї наївної моделі ймовірності.

Проте баєсів класифікатор має і наступні суттєві недоліки:

- Складно визначити форми розподілу даних.
- Існує обмеження дефіциту даних, оскільки для якого будь-яке можливого значення в просторі об'єктів значення ймовірності повинно бути оцінене за частотою.

1.5. Логістична регресія

Логістична регресія - це статистична модель, яка у своїй базовій формі використовує логістичну функцію для моделювання двійкової залежної змінної, хоча існує багато більш складних розширень. У регресійному аналізі, логістична регресія - оцінка параметрів логістичної моделі (форма двійковій регресії) [6]. Математично, двійкова логістична модель має залежну змінну з двома можливими значеннями, наприклад, пропуск / невдача, яка представлена змінною індикатора , де ці два значення позначені як "0" та "1". У логістичній моделі лог-коефіцієнти (логарифм з шансів) для значення з написом «1» представляє собою лінійну комбінацію з одного або декількох незалежних змінних («провісників»); незалежні змінні можуть бути двійковою змінною (два класи, кодовані індикаторною змінною) або неперервною змінною (будь-яке дійсне значення). Відповідна ймовірність значення, позначеного як "1", може коливатися від 0 (безумовне значення "0") до 1 (безумовне значення "1"); функція, яка перетворює лог-коефіцієнти на ймовірність, є логістичною функцією, звідси і назва. Одиниця виміру для шкали лог-коефіцієнтів називається логіт, звідси і альтернативні назви. Також

можуть бути використані аналогічні моделі з іншою сигмовидною функцією замість логістичної, наприклад, модель пробіта; визначальною характеристикою логістичної моделі є те, що збільшення однієї із незалежних змінних мультиплікативно масштабує шанси даного результату з постійною швидкістю, причому кожна незалежна змінна має свій параметр; для двійкової залежної змінної це узагальнює співвідношення шансів. На рисунку 1.4 можна побачити принцип роботи логістичної регресії.

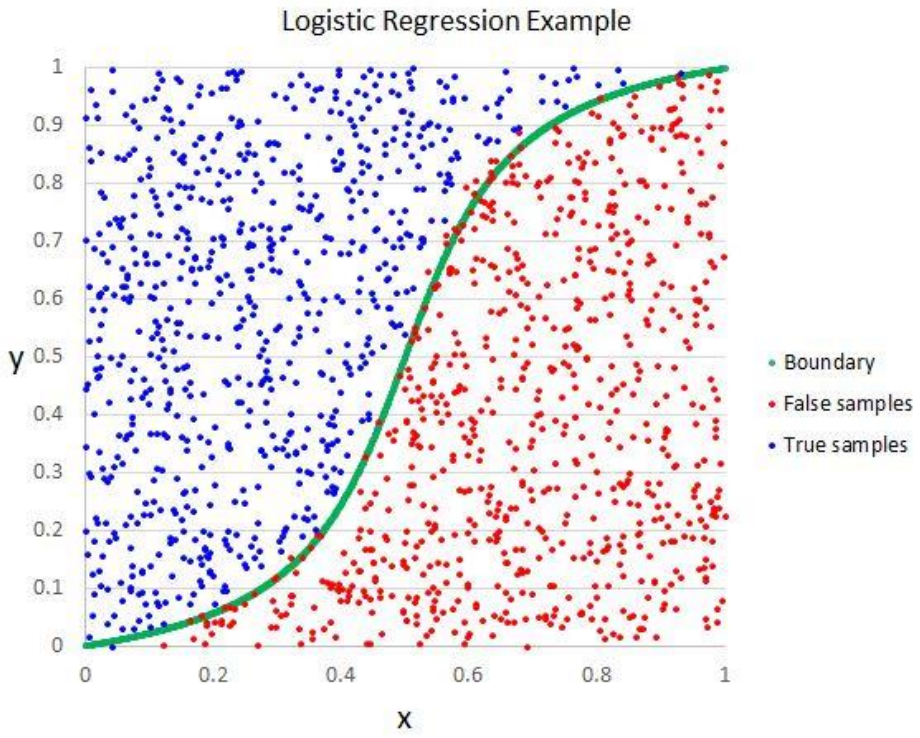


Рисунок 1.4 – Принцип роботи алгоритму Логістичної регресії.

У двійковій логістичній моделі регресії залежна змінна має два рівні (категоріальності). Виходи з більш ніж двома значеннями моделюються багаточленною логістичною регресією та, якщо кілька категорій упорядковані, порядковою логістичною регресією (наприклад, пропорційна шансова порядкова логістична модель). Сама модель логістичної регресії просто моделює ймовірність випуску з точки зору вхідних даних і не проводить статистичної класифікації(це не класифікатор), хоча його можна використовувати для створення класифікатора, наприклад, вибираючи граничне значення та класифікуючи входи з імовірністю більшими за граничне значення, як один клас, нижче граничного значення як інший; це

звичайний спосіб зробити двійковий класифікатор . Коефіцієнти, як правило, не обчислюються виразом із замкнутою формою, на відміну від лінійних коефіцієнтів найменших квадратів.

Розглянемо модель з двома предикторами, x_1 і x_2, і одна бінарна змінна відповіді *Y*, який позначаємо *p=P(Y=1)*. Ми припускаємо лінійну залежність між змінними предиктора та лог-коефіцієнти події, яка *Y=1*. Цей лінійний зв'язок можна записати у такій математичній формі (де l – лог-коефіцієнти, b – основа логарифму, і β_i є параметрами моделі):

$$l = log_b \frac{p}{1-p} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

Можна відновити шанси, підсиливши лог-коефіцієнти:

$$\frac{p}{1-p} = b^{\beta_0+\beta_1 x_1+\beta_2 x_2}$$

За допомогою простих алгебраїчних маніпуляцій ймовірність того, що *Y=1* становить:

$$p = \frac{b^{\beta_0+\beta_1 x_1+\beta_2 x_2}}{b^{\beta_0+\beta_1 x_1+\beta_2 x_2+1}} = \frac{1}{1+b^{-(\beta_0+\beta_1 x_1+\beta_2 x_2)}}$$

Наведена вище формула показує, що один раз β_i є фіксованими, ми можемо легко обчислити або log-шанси, де *Y=1* для даного спостереження, або ймовірність того, що Y=0 для даного спостереження. Основним випадком використання логістичної моделі має бути спостереження (x_1, x_2) та оцінка ймовірністі *p*, така що *Y=1*.

Логістична регресія вимірює взаємозв'язок між категоріально залежною змінною та однією або кількома незалежними змінними шляхом оцінки ймовірностей за допомогою логістичної функції, яка є кумулятивною функцією розподілу логістичного розподілу. Таким чином, він розглядає той самий набір проблем, що і регресія пробіта, використовуючи подібні методи, причому останні використовують замість цього кумулятивну криву нормального розподілу. Еквівалентно, при інтерпретації прихованих змінних

цих двох методів перший передбачає стандартний логістичний розподіл помилок, а другий - стандартний нормальний розподіл помилок [7].

Логістична регресія може розглядатися як приватний випадок узагальненої лінійної моделі і, таким чином, аналогічна лінійній регресії. Однак модель логістичної регресії базується на зовсім інших припущеннях (про взаємозв'язок між залежними та незалежними змінними) від тих, що стосуються лінійної регресії. Зокрема, ключові відмінності між цими двома моделями можна побачити в наступних двох особливостях логістичної регресії. По-перше, умовний розподіл $y \mid x$ є розподілом Бернуллі, а не розподілом Гауса, оскільки залежна змінна є двійковою. По-друге, прогнозовані значення є ймовірностями і тому обмежуються *(0,1)* через функцію логістичного розподілу, оскільки логістична регресія передбачає ймовірність конкретних результатів, а не значення самих результатів.

Логістична регресія є альтернативою методу Фішера (лінійний дискримінантний аналіз). Якщо виконуються припущення лінійного дискримінантного аналізу, росподіл може бути обернено, щоб отримати логістичну регресію. Однак зворотнє не відповідає дійсності, оскільки логістична регресія не вимагає багатовимірного нормального припущення про дискримінантний аналіз.

1.6. Дерево рішень

Навчання на дереві рішень - це метод, який зазвичай використовується в аналізі даних. Метою є створення моделі, яка передбачає значення цільової змінної на основі кількох вхідних змінних.

Дерево рішень - це просте представлення для класифікації даних. Для цього припустимо, що всі вхідні функції мають кінцеві дискретні домени, і існує одна цільова ознака, яка називається "класифікація". Кожен елемент області класифікації називається класом. Дерево рішень або дерево класифікації - це дерево, в якому кожен внутрішній (нелистовий) вузол позначений вхідною ознакою. Дуги, що надходять від вузла, позначеного

вхідною ознакою, позначаються кожним із можливих значень цільової ознаки, або дуга веде до підлеглого вузла прийняття рішень на іншій вхідній ознаці. Кожен лист дерева позначений класом або розподілом ймовірностей за класами, що означає, що набір даних класифікований деревом або до певного класу, або до певного розподілу ймовірностей (що, якщо дерево рішень добре конструйоване, перекошений до певних підмножин класів).

Дерево будується шляхом розбиття набору джерел, що становить кореневий вузол дерева, на підмножини, які становлять діти-наступники. Розподіл базується на наборі правил розбиття на основі ознак класифікації. Цей процес повторюється на кожній похідній підмножині рекурсивно, що називається рекурсивним секціонуванням. Рекурсії завершуються, коли підмножина в вузлі має те ж значення цільових змінних, або, коли розділення більше не підвищує цінність прогнозів. Цей процес індукції зверху вниз дерев рішень (TDIDT) є прикладом жадібного алгоритму, і це, безумовно, найпоширеніша стратегія навчання дерев рішень на основі даних. На рисунку 1.4 можна побачити принцип роботи логістичної регресії.

У процесі видобутку даних дерева рішень можна описати також як поєднання математичних та обчислювальних методів для полегшення опису, категоризації та узагальнення даного набору даних. На рисунку 1.5 можна побачити приклад побудованого дерева рішень.

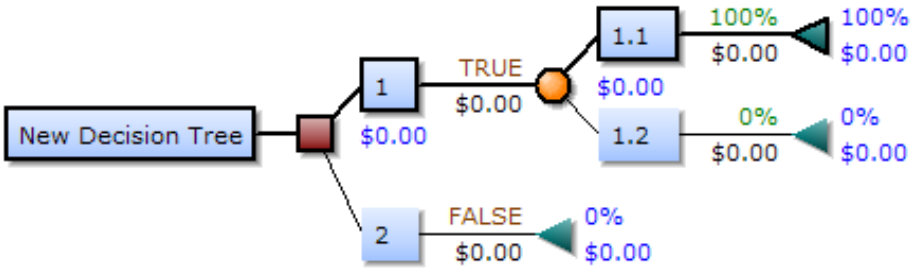


Рисунок 1.5 – Принцип роботи дерева класифікації.

Прикладом класифікаційного дерева рішень є так званий алгоритм ID3 (Ітеративний Dichotomiser 3).

Алгоритм ID3 починається з вихідного набору *S* як кореневий вузол[9]. На кожній ітерації алгоритму він перебирає кожен невикористаний атрибут

набору *S* і обчислює ентропію *H(S)* або отримання інформації *IG(S)* цього атрибута. Потім він вибирає атрибут, що має найменше значення ентропії (або найбільший приріст інформації). Набір *S* потім розділяється обраним атрибутом для отримання підмножин даних. (Наприклад, вузол може бути розділений на дочірні вузли на основі підмножин сукупності, вік яких менше 50, від 50 до 100 і більше 100.) Алгоритм продовжує повторюватися для кожної підмножини, враховуючи лише атрибути, які ніколи не були вибрані раніше.

Рекурсія на підмножині може зупинитися в одному з таких випадків:

1) Кожен елемент у підмножині належить до одного класу; у цьому випадку вузол перетворюється на листовий вузол і позначається класом прикладів.
2) Більше немає атрибутів для вибору, але приклади все одно не належать до одного класу. У цьому випадку вузол робиться листовим вузлом і маркується найпоширенішим класом прикладів у підмножині.
3) Немає ніяких прикладів в підмножині , яке відбувається , коли не було знайдено ні одного прикладу в вихідному наборі , щоб відповідати конкретне значення обраного атрибута. Прикладом може бути відсутність людини серед населення віком понад 100 років. Потім створюється листовий вузол, який позначається найпоширенішим класом прикладів у наборі батьківського вузла.

Протягом алгоритму дерево рішень будується з кожним нетермінальним вузлом (внутрішнім вузлом), що представляє вибраний атрибут, на якому були розділені дані, і кінцевими вузлами (листовими вузлами), що представляють мітку класу остаточної підмножини цієї гілки.

Отже дерева рішень мають наступні переваги:

- Легко впорюються з якісними (категоричними) особливостями.
- Добре працюють з межами рішення, паралельними осі об’єкта.

- Дерево рішень - це дуже швидкий алгоритм як навчання, так і прогнозування.

І недоліки:

- Проблеми з діагональними межами рішень.
- Можуть бути легко перенавчити.
- Надзвичайно чутливі до невеликих перемішень даних.
- Проблеми з прогнозуванням поза вибіркою.

Також одним із найросповсюдженіших алгоритмів на основі дерева рішень є алгоритм Random forest (випадковий ліс). Це суттєва модифікація беггінгу, яка створює колекцію некорельованих дерев, а потім їх усереднює.

Беггінг – це мета-алгоритм машинного навчання, призначений для підвищення стабільності та точності алгоритмів машинного навчання.

Основна ідея беггінгу полягає в тому, щоб усередити багато великих, але приблизно неупереджених моделей, і таким чином зменшити дисперсію. Дерева є ідеальними кандидатами для беггінгу, оскільки вони можуть фіксувати в даних складні структури взаємодії, і якщо вони ростуть досить глибоко, вони мають відносно низький ухил. Оскільки дерева можуть мати великі розміри, вони отримують велику користь від усереднення. На рисунку 1.6 можна побачити приклад побудованого випадкового лісу.

Кожне дерево будується з використанням наступного алгоритму:

1. Нехай N - кількість тестових випадків, M - кількість змінних у класифікаторі.
2. Нехай m - кількість вхідних змінних, які будуть використані для визначення рішення в даному вузлі; m має бути набагато менше M.
3. Вибирається навчальний набір для цього дерева та використовується решта тестових кейсів для оцінки помилки.
4. Для кожного вузла у дереві випадковим чином вибирається m змінних, на основі яких буде прийнято рішення. Обчисліть найкращий розділ навчального набору з m змінних.

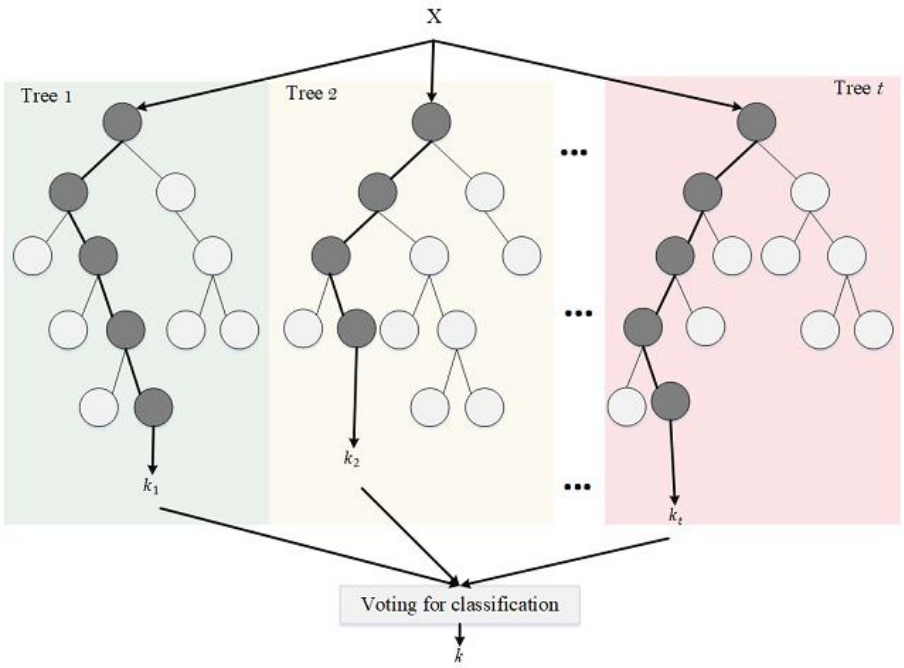


Рисунок 1.6 – Структура Random forest.

Для прогнозування новий випадок зрушується з дерева. Потім йому присвоюється мітка термінального вузла, де він закінчується. Цей процес повторюється по всіх деревах у збірці, і мітка, яка отримує найбільше випадків, повідомляється як передбачення.

Отже, Random forest мають такі переваги:

- Ансамблі дерев рішень дуже швидко піддаються навчанню порівняно з іншими техніками.
- Зменшена дисперсія (щодо звичайних дерев).
- Не вимагає підготовки та попередньої обробки вхідних даних.

І недоліки:

- Досить повільно створюються прогнози після навчання.
- Більше дерев у лісі збільшує складність часу на етапі прогнозування.
- Не просто візуально інтерпретувати.
- Легко може бути надмірність.
- Потрібно вибрати кількість дерев у лісі.

1.7. Метод опорних векторів

Опорні векторні машини (support vector machines, SVM) - це набір керованих алгоритмів навчання, розроблених Володимиром Вапником та його командою в лабораторіях AT&T.

Ці методи належним чином пов'язані з проблемами класифікації та регресії. Враховуючи набір навчальних прикладів, можемо позначити класи та навчити SVM будувати модель, яка передбачає клас нової вибірки. Інтуїтивно SVM - це модель, яка представляє вибіркові точки в просторі, розділяючи класи на 2 простори, наскільки це можливо, за допомогою гіперплощини розділення, визначеної як вектор між 2 точками, з 2 класів, найближчий до якого називається опорним вектором. Коли нові зразки приводяться у відповідність зі згаданою моделлю, залежно від просторів, до яких вони належать, їх можна віднести до одного або іншого класу [10].

Більш формально, SVM будує гіперплощину або набір гіперплощин у дуже високому (або навіть нескінченному) розмірному просторі, який може бути використаний у задачах класифікації або регресії. Хороший розподіл між класами дозволить правильно класифікувати.

Враховуючи набір точок, підмножину більшої множини (пробілу), в якій кожна з них належить до однієї з двох можливих категорій, алгоритм на основі SVM будує модель, здатну передбачити, чи нова точка (категорію якої ми не знаємо) належить до тієї чи іншої категорії.

Як і в більшості контрольованих методів класифікації, вхідні дані (точки) розглядаються як p-мірний вектор (упорядкований список p чисел).

SVM шукає гіперплощину, яка оптимально відокремлює точки одного класу від точки іншого, який, зрештою, раніше міг бути спроектований у більш вимірний простір.

У цій концепції "оптимального розділення" полягає основна характеристика SVM: цей тип алгоритму шукає гіперплощину, яка має максимальну відстань (запас) з найближчими точками. Ось чому SVM іноді називають класифікаторами основних націнок. Таким чином, векторні точки,

позначені однією категорією, будуть з одного боку гіперплану, а випадки, що входять до іншої категорії, будуть з іншого боку.

Алгоритми SVM належать до сімейства лінійних класифікаторів. Їх також можна вважати особливим випадком регуляризації Тихонова.

У літературі SVM змінну предиктора називають атрибутом, а перетворений атрибут, який використовується для визначення гіперплощини, - характеристикою. Вибір найбільш відповідного подання досліджуваного простору здійснюється за допомогою процесу, який називається вибором ознак.

Вектор, утворений точками, найближчими до гіперплощини, називається вектором опори.

Моделі на основі SVM тісно пов'язані з нейронними мережами. Використовуючи функцію ядра, вони дають альтернативний метод навчання поліноміальних класифікаторів, радіальних базисних функцій та багатошарового персептрона.

У наступному ідеалізованому прикладі для 2-виміру подання даних (рисунок 1.7), що підлягають класифікації, виконується в площині *xy*. Алгоритм SVM намагається знайти одновимірну гіперплощину (у розглянутому прикладі це пряма лінія), яка приєднує змінні предикатора і становить межу, що визначає, чи належить елемент введення до однієї категорії чи іншої.

Існує нескінченна кількість можливих гіперплощин (ліній), які виконують класифікацію, але найкращим рішенням є те, що допускає максимальний запас між елементами двох категорій.

Опорними векторами називаються точки, що складають дві прямі, паралельні гіперплощині, при цьому відстань між ними (поле) є якомога більшою.

В ідеалі, модель, заснована на SVM, повинна створювати гіперплощину, яка повністю відокремлює дані від досліджуваного простору на дві категорії.

Однак ідеальне розділення не завжди можливо, і якщо воно є, вихід моделі не можна узагальнити на інші дані. Це відомо як надмірність.

Для того, щоб забезпечити деяку гнучкість, SVM обробляють параметр *C*, який контролює компроміс між навчальними помилками та жорсткими полями, створюючи таким чином м'який запас, який допускає деякі помилки в класифікації, одночасно виправляючи їх.

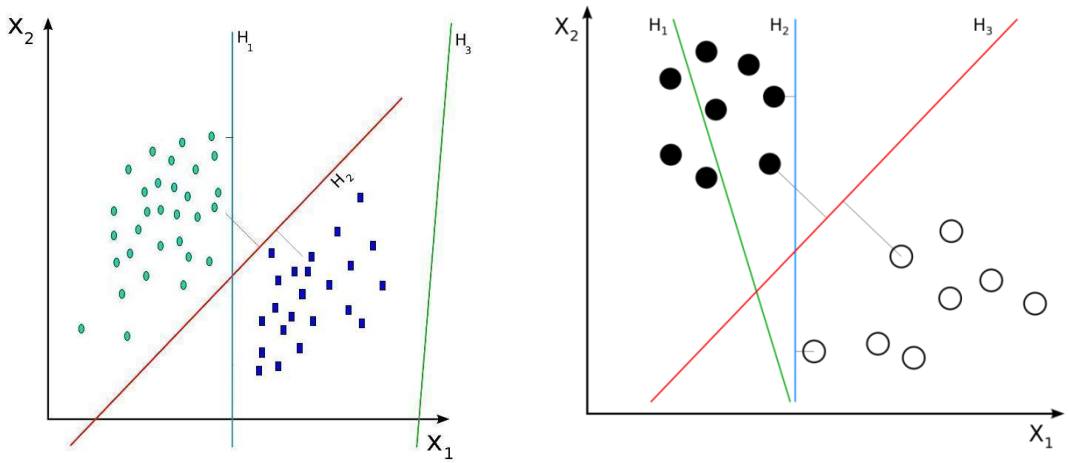


Рисунок 1.7 – Принцип роботи SVM. *H1* не розділяє класи. *H2* розділяє їх, але лише з невеликим запасом. *H3* розділяє їх з максимальним запасом.

В ідеалі, модель, заснована на SVM, повинна створювати гіперплощину, яка повністю відокремлює дані від досліджуваного простору на дві категорії. Однак ідеальне розділення не завжди можливо, і якщо воно є, вихід моделі не можна узагальнити на інші дані. Це відомо як надмірність.

Для того, щоб забезпечити деяку гнучкість, SVM обробляють параметр *C*, який контролює компроміс між навчальними помилками та жорсткими полями, створюючи таким чином м'який запас, який допускає деякі помилки в класифікації, одночасно виправляючи їх.

Найпростіший спосіб відокремити - пряма лінія, пряма площина або *N*-мірна гіперплощина.

Через обчислювальні обмеження лінійних машин навчання вони не можуть бути використані в більшості реальних програм. Представлення за допомогою функцій ядра пропонує вирішення цієї проблеми, проектуючи інформацію на більший простір характеристик, що збільшує обчислювальну

здатність лінійних машин навчання. Тобто ми відобразимо вхідний простір X до нового простору ознак вищої розмірності (Гільберта):

$F = \{\varphi(x) | x \in X\}$

$x = \{x1, x2, \cdots, xn\} \rightarrow \varphi(x) = \{\varphi1(x), \varphi2(x), \cdots, \varphi n(x)\}$

Отже, переваги SVM наступні:

- SVM може моделювати нелінійні межі прийняття рішень.
- Виконується аналогічно логістичній регресії при лінійному розділенні.
- Надійна проти проблем із надмірністю.

А недоліки такі:

- недостатня прозорість результатів, спричинена великою розмірністю.
- Вибір ефективної функції ядра важкий (сприйнятливий до надмірності / проблем із навчанням залежно від ядра).
- Складність пам'яті.

1.8. Нейронні мережі

Штучні нейронні мережі - це обчислювальна модель, натхненна поведінкою, що спостерігається у відповідному біологічному аналозі. Вона складається з набору одиниць, які називаються штучними нейронами, з'єднаних між собою для передачі сигналів. Вхідна інформація проходить через нейронну мережу (де вона зазнає різних операцій), виробляючи вихідні значення.

Кожен нейрон пов’язаний з іншими за допомогою зв’язків, де вихідне значення попереднього нейрона множиться на вагове значення. Ці ваги зв’язку можуть збільшувати або гальмувати стан активації сусідніх нейронів. Подібним чином, на виході нейрона може існувати обмежувальна або порогова функція, яка модифікує отримане значення або встановлює межу, яку не можна перевищувати перед розповсюдженням на інший нейрон. Ця функція відома як функція активації.

Ці системи навчаються самі, а не чітко програмуються, і досягають успіху в областях, де пошук рішень або особливостей важко виразити за допомогою звичайного програмування. Для цього машинне навчання, як правило, намагається мінімізувати функцію втрат, яка оцінює мережу в цілому. Значення ваг нейронів оновлюються з метою зменшення значення функції втрат. Цей процес здійснюється шляхом розповсюдження назад. На рисунку 1.8 можна побачити узагальнену структуру нейронної мережі.

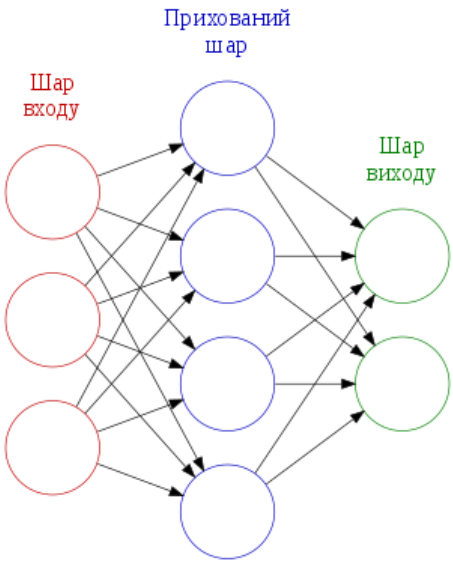


Рисунок 1.8 – Штучна нейронна мережа.

Метою нейронної мережі є вирішення проблем таким же чином, як і мозку людини, хоча нейронні мережі є більш абстрактними. Сучасні нейронні мережі, як правило, містять від декількох тисяч до кількох мільйонів нейронних одиниць.

Нові дослідження мозку часто стимулюють створення нових зразків у нейронних мережах. Новий підхід - використання далекосяжних зв’язків та шарів обробки зв’язків, а не завжди локалізація на сусідніх нейронах. Інші дослідження вивчають різні типи сигналів з часом, які поширюються аксони, оскільки глибоке навчання інтерполює більшу складність, ніж набір булевих змінних, які просто ввімкнені чи вимкнені.

Нейронні мережі використовувались для вирішення найрізноманітніших завдань, таких як комп’ютерний зір та розпізнавання мови, які важко

вирішити за допомогою звичайного програмування на основі правил. Історично використання моделей нейронних мереж позначило зміну напрямку в кінці вісімдесятих з високого рівня, що характеризується експертними системами зі знаннями, вбудованими в правила тоді, і на машинне навчання низького рівня, що характеризується знання, включені в параметри когнітивної моделі з якоюсь динамічною системою.

Існує багато варіантів нейронних мереж, які використовуються в Data Mining, розглянемо деякі з них:

Періодичні нейронні мережі (Recurrent Neural Networks, RNN). RNN призначає більше ваг попереднім точкам послідовності даних. Тому цей прийом є потужним методом класифікації рядків та послідовних даних. Більше того, цю техніку можна використовувати для класифікації зображень. У RNN нейронна мережа розглядає інформацію попередніх вузлів дуже витонченим методом, який дозволяє покращити семантичний аналіз структур у наборі даних.

Довга короткочасна пам'ять - (Long Short-Term Memory, LSTM) була введена С. Хохрейтером та Дж. Шмідхубером і розроблена багатьма вченими-дослідниками. Для вирішення цих проблем довга короткочасна пам’ять (LSTM) - це особливий тип RNN, який зберігає довгострокову залежність більш ефективним способом порівняно з базовими RNN. Це особливо корисно для подолання проблеми зникнення градієнта. Незважаючи на те, що LSTM має ланцюгову структуру, подібну до RNN, LSTM використовує безліч шлюзів для ретельного регулювання обсягу інформації, яка буде допущена до кожного стану вузла.

Ще однією архітектурою глибокого навчання, яка використовується для ієрархічної класифікації документів, є згорткові нейронні мережі (Convolutional Neural Networks, CNN). Хоча спочатку вони були побудовані для обробки зображень з архітектурою, подібною до зорової кори, CNN також ефективно використовуються для класифікації даних. У базовій CNN для обробки зображень тензор зображення складається з набором ядер розміром *d*

на *d* . Ці шари згортки називаються функціональними картами, і їх можна скласти, щоб забезпечити кілька вхідних фільтрів. Щоб зменшити обчислювальну складність, CNN використовують об'єднання, яке зменшує розмір вихідних даних від одного рівня до наступного в мережі. Для зменшення результатів роботи, зберігаючи важливі особливості, використовуються різні методи об’єднання.

Найпоширенішим методом об'єднання є максимальне об'єднання, де максимальний елемент вибирається у вікні об'єднання. Для того, щоб об'єднати вихідні дані зі складених карт на наступний шар, карти згладжуються в одну колонку. Кінцеві шари в CNN, як правило, повністю з'єднані щільними шарами. Загалом, на етапі зворотного розповсюдження згорткової нейронної мережі регулюються не тільки ваги, а й фільтри детектора характеристик.

Повторювані згорткові нейронні мережі (Recurrent Convolutional Neural Networks, RCNN) також використовуються для класифікації тексту. Основною ідеєю цієї техніки є збір контекстної інформації з повторюваною структурою та побудова даних за допомогою згорткової нейронної мережі. Ця архітектура є поєднанням RNN та CNN для використання переваг обох методів у моделі.

1.9. Алгоритм перцептрон

Персептрон - спрощена модель біологічного нейрона . Хоча складність моделей біологічних нейронів часто потрібна для повного розуміння нейронної поведінки, дослідження показують, що персептрон-подібна лінійна модель може спричинити деяку поведінку, яка спостерігається у реальних нейронах [8].

У сучасному розумінні персептрон - це алгоритм вивчення двійкового класифікатора, який називається пороговою функцією: функцією, яка відображає його вхідні дані *x* (дійсний вектор) до вихідної величини *f(x)* (одне двійкове значення):

$$f(x) = \begin{cases} 1, & if\ w \cdot x + b > 0 \\ 0, & else \end{cases}$$

Де w - вектор реальних ваг, де $w \cdot x$ – скалярний добуток $\sum_{i=1}^{m} w_i x_i$, де m - кількість входів в персептрон, а b - зміщення. Упередженість зміщує межу прийняття рішення від початку координат і не залежить від будь-якого вхідного значення.

Значення *f(x)* (0 або 1) використовується для класифікації *x* як позитивна, так і негативна міра, у випадку двійкової проблеми класифікації. Якщо b від'ємне, то зважена комбінація вхідних даних повинна давати позитивне значення більше ніж $|\ b\ |$ для того, щоб перенести нейрон класифікатора через поріг 0. Просторово зміщення змінює положення (хоча і не орієнтацію) межі рішення. Алгоритм навчання персептрону не припиняється, якщо навчальний набір не можна лінійно розділяти. Якщо вектори не є лінійно відокремленими, навчання ніколи не досягне точки, коли всі вектори класифікуються належним чином. Найвідомішим прикладом нездатності персептрона розв'язувати задачі з лінійно не відокремленими векторами є булева ексклюзивна задача.

У контексті нейронних мереж персептрон - це штучний нейрон, що використовує функцію ступеня Хевісайда як функцію активації. Алгоритм персептрона також називають одношаровим персептроном, щоб відрізнити його від багатошарового персептрона, що є неправильним терміном для більш складної нейронної мережі. Будучи лінійним класифікатором, одношаровий персептрон є найпростішою нейронною мережею прямого зв'язку.

Нижче наведено приклад алгоритму навчання одношарового персептрона. Для багатошарових персептронів, де існує прихований шар, слід використовувати більш складні алгоритми, такі як зворотне розповсюдження. Якщо функція активації або основний процес, що моделюється персептроном, є нелінійним, можна використовувати альтернативні алгоритми навчання, такі як правило дельта, доки функція активації є диференційованою. Тим не

менше, алгоритм навчання, описаний у кроках нижче, часто буде працювати, навіть для багатошарових перцептронів з нелінійними функціями активації.

Коли декілька персептронів об'єднані у штучній нейронній мережі, кожен вихідний нейрон працює незалежно від усіх інших; таким чином, вивчення кожного результату можна розглядати окремо.

Спочатку визначимо деякі змінні:

- r - швидкість навчання персептрону. Швидкість навчання становить від 0 до 1, більші значення роблять зміни ваги більш нестабільними.
- $y=f(z)$ позначає вихід з персептрона для вхідного вектора z.

$D=\{(x_1,d_1),\ldots,(x_s,d_s)\}$ – тренувальний набір для s зразків, де:

- x_j – n-вимірний вхідний вектор
- d_j– бажане вихідне значення персептрона для цього входу.

Показуємо значення функцій наступним чином:

- $x_{(j,i)}$ – значення i-го особливості j-го навчального вхідного вектора.
- $x_{(j,i)}=1$

Для подання ваг:

- w_j - це i-те значення у векторі ваг, яке помножується на i-те значення вхідної функції.
- Оскільки $x_{(j,i)}=1$, w_0 фактично є зміщенням, яке використовується замість константи зміщення b.

Щоб показати часову залежність w, використовується:

- $w_i(t)$ це вага i за час t.

Отже алгоритм можна поділити на наступні кроки:

1. Ініціалізуємо ваги та поріг. Ваги можуть бути ініціалізовані до 0 або до невеликого випадкового значення. У наведеному нижче прикладі використовується 0.
2. Для кожного прикладу j у навчальному наборі D виконуємо наступні кроки над введенням x_j і бажаний результат d_j:
   - Обчислимо фактичний випуск:

$$y_j(t) = f\left|w(t) \cdot x_j\right| =$$

$$= f(w_0(t)x_{j,0} + w_1(t)x_{j,1} + w_2(t)x_{j,2} + \cdots + w_n(t)x_{j,n})$$

- Оновлюємо ваги (приклади застосування та оновлення ваг можна побачити на рисунках 1.8 і 1.9 відповідно):
  $w_i(t+1) = w_i(t) + r \cdot (d_j - y_j(t))x_{j,i}$, для всіх функцій $0 \leq i \leq n, r$ – це швидкість навчання

3. Для офлайн-навчання другий крок може повторюватися до помилки ітерації $\frac{1}{S}\sum_{j=1}^{S}\left|d_j - y_j(t)\right|$ менше заданого користувачем порогу помилки γ, або було виконано заздалегідь визначену кількість ітерацій, де s знову є розміром набору вибірки.

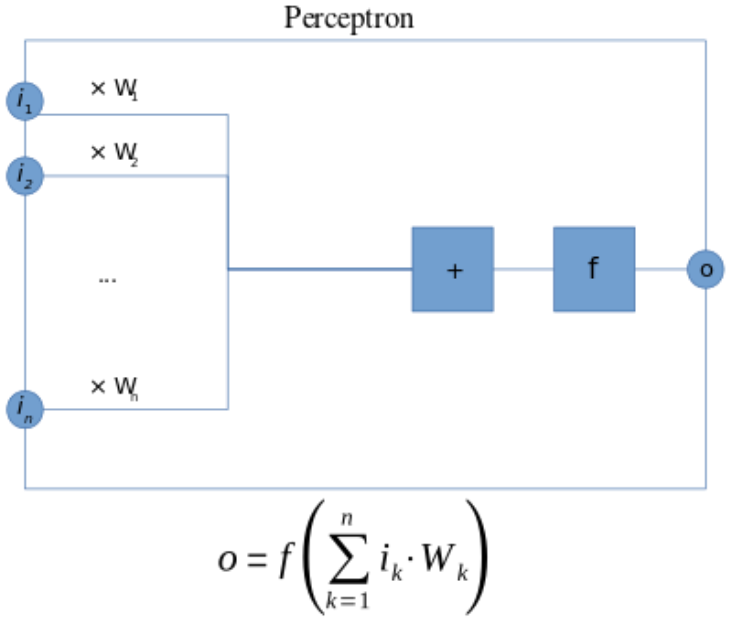


Рисунок 1.8 – Відповідні ваги застосовуються до вхідних даних, а отримана зважена сума передається функції.

Персептрон є лінійним класифікатором, тому він ніколи не дійде до стану з усіма вхідними векторами, класифікованими правильно, якщо навчальний набір D не лінійно відокремлюваний, тобто якщо позитивні приклади не можна відокремити від негативних прикладів гіперплощиною. У цьому випадку до стандартного алгоритму навчання не буде поступово наближатись жодне «приблизне» рішення, а навпаки, навчання повністю

провалиться. Отже, якщо лінійна відокремлюваність навчального набору апріорі невідома, слід використовувати один із варіантів навчання нижче.

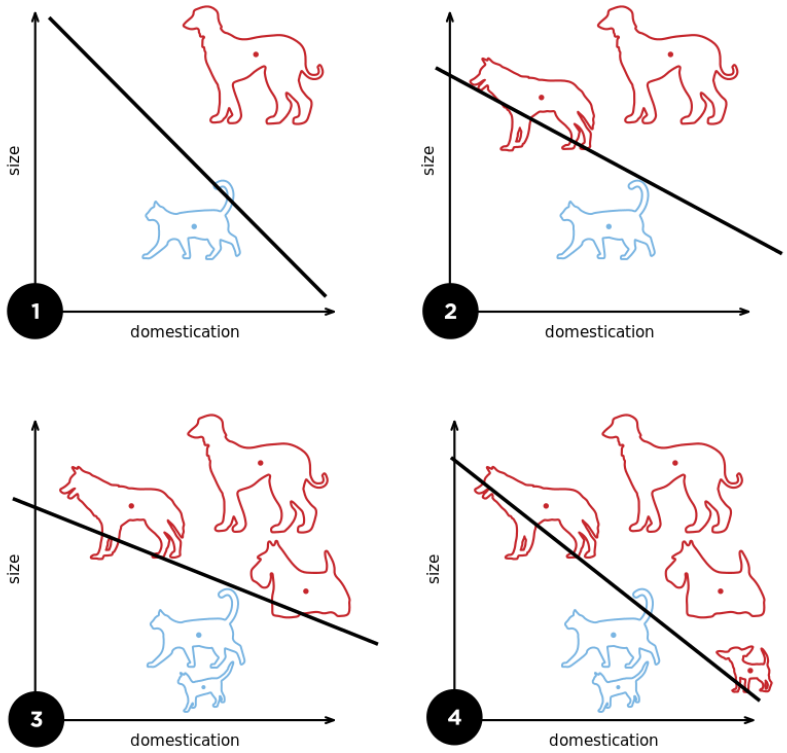


Рисунок 1.9 – Персептрон, що оновлює свою лінійну межу, коли додається більше прикладів навчання.

Якщо навчальний набір є лінійно нероздільні, то персептрон гарантовано сходитися. Крім того, існує верхня межа кількості випадків, коли персептрон буде регулювати свою вагу під час тренування.

Припустимо, що вхідні вектори з двох класів можуть бути розділені гіперплощиною з полем γ, тобто існує ваговий вектор $w, \|w\| = 1$, а термін упередження b такий, що $w \cdot x_j > \gamma$ для усіх j з $d_j = 1$ і $w \cdot x_j < -\gamma$ для усіх j з $d_j = 0$, де d_j - бажане вихідне значення персептрона для введення j. Також нехай R позначає максимальну норму вхідного вектора. Доведено, що в цьому випадку алгоритм персептрона збігається після створення $O(R^2/\gamma^2$ оновлення. Ідея доказу полягає в тому, що вектор ваги завжди коригується обмеженою величиною в напрямку, з яким він має від'ємне крапкове добуток, і, отже, може бути обмежений вище $O(\sqrt{t}\,)$, де t - кількість змін вектор ваги.

Однак він також може бути обмежений нижче *O(t)*, оскільки якщо існує (невідомий) вектор задовільної ваги, то кожна зміна прогресує в цьому (невідомому) напрямку на позитивну величину, яка залежить лише від вхідного вектора.

Два класи точок і два з нескінченно багатьох лінійних меж, що розділяють їх. Незважаючи на те, що межі знаходяться майже під прямим кутом одна до одної, алгоритм персептрону не може вибрати між ними.

Хоча алгоритм персептрону гарантовано збігається на якомусь рішенні у випадку лінійно відокремлюваного навчального набору, він все одно може вибрати будь-яке рішення, і проблеми можуть допускати безліч рішень різної якості. Персептрон оптимальної стабільності, в даний час більш відомий як лінійні опорні вектори, був розроблений, щоб вирішити цю проблему.

1.10. Самоорганізаційна карта Кохонена

Самоорганізаційна карта Кохонена (self-organized map, SOM) - це повністю зв’язана одношарова лінійна штучна нейронна мережа, де вихід, як правило, організований у двовимірному розташуванні вузлів (рисунок 1.11). Основою SOM є м’яка конкуренція між вузлами в вихідний рівень; оновлюється не лише один вузол (переможець), а й його сусіди.

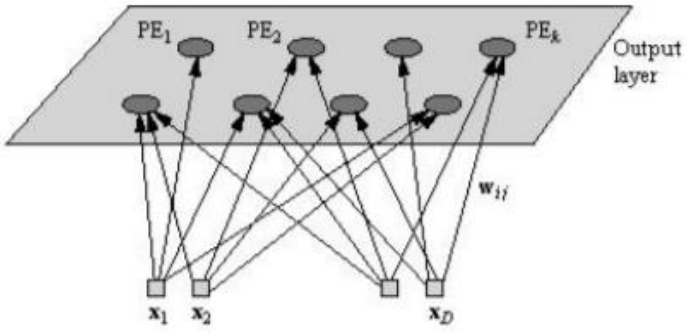


Рисунок 1.11 – Схема SOM.

Мережі, що самоорганізуються, мають можливість вивчати та виявляти закономірності та кореляції на входах, а також передбачати відповіді на вхідні дані. Нейрони в конкурентній мережі вчаться розпізнавати групи подібних

вхідних векторів, тоді як самоорганізуючі карти (SOM) вчаться розпізнавати групи подібних вхідних векторів таким чином, що нейрони фізично поруч один з одним у нейронному шарі реагують на подібні вхідні вектори. Карти, що самоорганізуються, вивчають як розподіл, так і топологію вхідних векторів, на яких вони навчаються; в той час як нейрон-переможець визначається аналогічно для конкурентних шарів та SOM, замість оновлення лише одного вузла SOM оновлює сусідство вузлів навколо вузла-переможця.

SOM визначає відображення із вхідного простору даних R^n на регулярний двовимірний масив вузлів (сітка карти). Параметричний опорний вектор $m_i \in R^n$ асоціюється з кожним вузлом і на карті. Масив вузлів можна проектувати на прямокутну або шестикутну решітку. Кожен вхідний вектор *x* порівнюється з $m_i : s$ і найкращий збіг зазначається, потім вхідні дані відображаються на цьому місці. SOM можна розглядати як "нелінійну проекцію" функції щільності ймовірності вхідних даних великого розміру на двовимірну площину. Кожен вхідний вектор $x \in R^n$ можна порівняти з $m_i : s$ в будь-якій метриці. Потім виграшний вузол *c* обчислюється за:

$$\|x - m_c\| = min_i\{\|x - m_i\|\}, \text{ або}$$

$$c = argmin_i\{\|x - m_i\|\}$$

і *x* відображається на *c* відносно значень параметрів m_i.

Вузли, топографічно близькі до інших в масиві, навчатимуться з того самого вводу. Формулу оновлення можна записати:

$$m_i(t+1) = m_i(t) + h_{c,i}(t)[x(t) - m_i(t)]$$

Де *t* - координата дискретного часу, $h_{c,i}$ - функція, що визначає околиці. Початкові значення $m_i : s$ можуть бути випадковими. Функція сусідства визначається над точками решітки або картою і певним чином визначає стійкість або еластичність, що відноситься до точок даних.

Структура SOM складається з наступних елементів:

1. Картографічна сітка. Зазвичай вхідні дані відображаються на 1- або 2-мірній карті. Нанесення на вищі розміри також можливо, але ускладнює візуалізацію. Розмір сітки (кількість нейронів) можна визначити вручну або

автоматично витягнути з вхідних даних. Нейрони, пов'язані з сусідніми нейронами, завдяки взаємозв'язку сусідства визначають структуру карти. Двома найпоширенішими двовимірними сітками є гексагональна сітка та прямокутна сітка. Іншими поширеними (3-мірними) формами карт є форми циліндра та тороїда.

2. Функція сусідства. Сусідство визначає кореляцію між нейронами. Найпростіша функція сусідства називається бульбашкою; вона постійна над сусідством нейрона-переможця, інакше нуль. Більш вигідним визначенням є функція гауссового сусідства:

$e^{-\frac{\|r_c - r_i\|^2}{2\sigma^2(t)}}$, де r_c це координата модуля c у сітці, а $\sigma(t)$ – радіус сусіда у час t. Сусідство з часом зменшується.

Алгоритм SOM для класифікації складається з наступних етапів:

1. Розмір і форма. Необхідно вказати кількість нейронів, розміри сітки карти, решітку та форму карти. Чим більше нейронів у нас є, тим вигіднішим стає відображення, але збільшується обчислювальна складність, а також фаза навчання.

2. Ініціалізація. Вагові вектори можна ініціалізувати різними способами: випадкова ініціалізації, ініціалізація вибірки (початкові вагові вектори беруться із випадкових вибірок на вході), лінійна ініціалізація (охоплюється власними векторами набору вхідних даних).

3. Навчання. На кожному етапі навчання вибірковий вектор x вибирається з набору вхідних даних випадковим чином і обчислюється схожість між ним та усіма векторами ваги на карті. Найбільш схожий вектор називається BMU (best-matching unit). Як метрику подібності зазвичай використовується евклідова відстань. Весові вектори на карті поблизу BMU оновлюються таким чином, щоб вони наближались до зразкового вектора.

4. Параметри тренінгу. Швидкість навчання є спадною функцією в інтервалі [0, 1]. Швидкість навчання може лінійно зменшуватися або визначатися якоюсь нелінійною функцією. Радіус сусідства також

зменшується з часом. Великі області, які роблять SOM більш жорстким, використовуються на початку навчання, а згодом зменшуються під час тренувань для налаштування SOM.

5. Пакетна підготовка. Пакетний алгоритм проходить весь навчальний набір один раз, і лише після цього оновлює ваги за допомогою мережевого впливу всіх зразків. Весові вектори на карті замінені вектором-прототипом із середньозваженим середнім для зразків, де ваговими коефіцієнтами є значення функції сусідства. Таким чином, функція оновлення для вагових векторів на карті стає:

$$m_i(t+1) = \frac{\int_{j=1}^{n} h_{i,c(j)}(t) x_j}{\int_{j=1}^{n} h_{i,c(j)}(t)}$$

Де m вектор ваг карти, $c(j)$ це BMU екземпляру вектора x_j, $h_{i,c(j)}$ – функція сусідства, а n – кількіть еземплярів.

Отже алгоритми на основі нейронних мереж мають такі переваги:

- Гнучкість завдяки дизайну функцій (зменшує потребу в розробці функцій)
- Архітектура, яка може бути адаптована до нових проблем.
- Зв’язані зі складними відображеннями вводу-виводу
- Легко піддаються онлайн-навчанню.
- Можливість паралельної обробки.

А також і недоліки:

- Потрібен великий обсяг даних.
- Навчати надзвичайно обчислювально.
- Інтерпретабельність моделі - найважливіша проблема глибокого навчання.
- Пошук ефективної архітектури та структури.

1.11. Метаевристичні алгоритми

Метаевристика - це процедура вищого рівня, призначена для пошуку, генерування або вибору (частковий алгоритм пошуку), яка може забезпечити досить хороше рішення проблеми оптимізації, особливо з неповною або недосконалою інформацією або обмеженою обчислювальною здатністю[11]. Метаевристика відбирає підмножину рішень, яка в іншому випадку занадто велика, щоб бути повністю переліченою або дослідженою іншим чином. Метаевристика може зробити порівняно мало припущень щодо розв'язуваної задачі оптимізації, і тому може бути використана для різних проблем.

Метаевристики знаходять широке застосування для вирішення складних оптимізаційних задач, машинного навчання, розпізнавання образів і т. ін. Вони забезпечують пошук оптимальних або близьких до оптимальних рішень. При цьому обсяг обчислень може виявитися великим, але швидкість його зростання при збільшенні розмірності задачі зазвичай буває менше, ніж у інших відомих методів. Зі зростанням продуктивності комп'ютерних систем і зменшенні їх вартості, метаевристики перетворилися на самий популярний інструмент пошуку оптимальних рішень задач, які раніше вважалися нерозв'язними.

Розглянемо дві найбільш відомі на даний момент реалізації метаевристичних алгоритмів для задачі класифікації: метод рою часток і мурашиний алгоритм.

Метод рою часток (particle swarm optimization, PSO) надихається розумною поведінкою істот як частину спільноти, що ділиться досвідом, на відміну від ізольованої індивідуальної реакції на навколишнє середовище. Процес адаптації має в основі три принципи: оцінювати, порівнювати та наслідувати.

Оцінка - це здатність кваліфікувати екологічні стимули та необхідна умова соціального навчання. Сама оцінка одночасно марна і неможлива без можливості порівняння; всі наші показники - це лише порівняння з добре відомою одиницею, і одне значення стає безглуздим без значень своїх сусідів.

Нарешті, імітація є найновішою формою обміну досвідом з точки зору приймача; це передбачає не тільки спостереження, але й усвідомлення адекватності мети та часу.

В алгоритмах PSO частка вирішує, куди рухатися далі, враховуючи власний досвід, який є пам’яттю про її найкращу минулу позицію та досвід свого найуспішнішого сусіда.

Для сусідства можуть існувати різні концепції та цінності; його можна розглядати як просторове сусідство, де воно визначається евклідовою відстанню між положеннями двох частинок, або як соціометричне сусідство (наприклад: індексне положення в масиві зберігання). Останній найчастіше використовується з двох основних мотивів:

- Якби координати представляли розумові здібності чи навички, два дуже схожі індивіди можуть ніколи не прийти назустріч за своє життя, що стосується елементів однієї сім'ї, які можуть суттєво відрізнятися один від одного, але тим не менш, вони завжди будуть сусідами.
- Обчислювальні зусилля, необхідні для обробки евклідової відстані, коли стикаються з великою кількістю частинок або розмірами - у кожній ітерації потрібно було б розрахувати відстань між кожними двома частинками, і для кожної частинки потрібно було б відсортувати найближчі k сусідів.

Кількість сусідів (*k*), як правило, розглядається або *k = 2*, або *k = all*.

Хоча деякі дії відрізняються від одного варіанту PSO до іншого, його псевдокод такий [12]:

```
Initiate_Swarm()
    Loop
    For p = 1 to number of particles
        Evaluate(p)
        Update_past_experience(p)
        Update_neighbourhood_best(p,k)
        For d = 1 to number of Dimensions
```

Move(p,d)

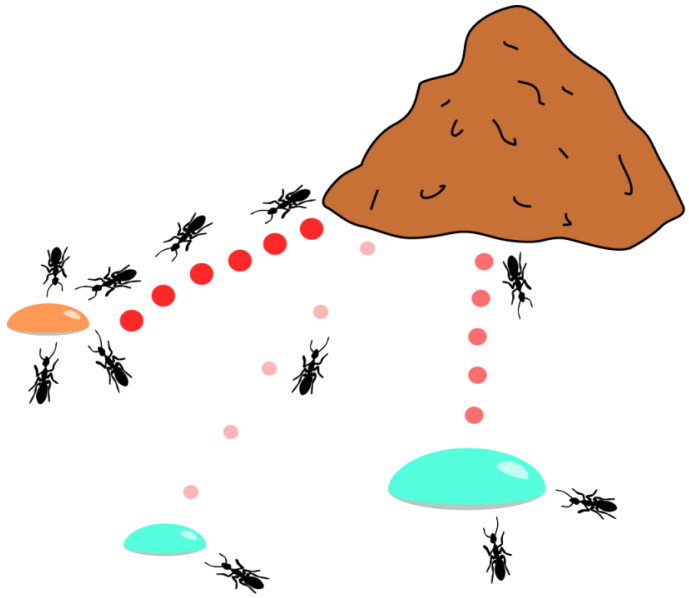
Until Criterion.

Існує необхідність підтримувати та оновлювати попереднє найкраще положення частинки (P_{id}) та найкраще положення в околиці (P_{gd}). Існує також швидкість (V_{id}), пов'язана з кожним виміром, що є приростом, який слід зробити в кожній ітерації до асоційованого виміру, таким чином змушуючи частинку змінювати своє положення в просторі пошуку:

$$\begin{cases} \upsilon_{id}(t) = \chi\big(\upsilon_{id}(t-1)\big) + \varphi_{1id}\big(P_{id} - x_{id}(t-1)\big) + \varphi_{2id}\left(P_{gd} - x_{id}(t-1)\right) \\ x_{id}(t) = x_{id}(t-1) + \upsilon_{id}(t) \end{cases}$$

Де φ_1та φ_2 - випадкові ваги, визначені верхньою межею, χ - коефіцієнт звуження. Загальний ефект рівняння полягає в тому, що кожна частинка коливається в просторі пошуку між своїм попереднім найкращим положенням та найкращим положенням свого найкращого сусіда, сподіваючись знайти нові найкращі точки під час своєї траєкторії. Якби швидкості частинки дозволялося змінюватись без обмежень, рій ніколи не сходився б до оптимуму, оскільки коливання частинок зростали. Тому зміни швидкості обмежуються χ - коефіцієнтом звуження, що змушує рій сходитися. Значення цього параметра та верхніх меж φ_1та φ_2 обираються так, щоб гарантувати збіжність.

Мурашиний алгоритм (Ant colony optimization, ACO) натхненний соціальними комахами, такими як мурахи. Хоча кожна особа має лише обмежені можливості, повний рій виявляє складну загальну поведінку. Тому розумна поведінка може розглядатися як нова характеристика рою. Зосереджуючись на колоніях мурашок, можна помітити, що мурахи спілкуються лише опосередковано - через оточення - відкладаючи речовину, яка називається феромон. Шляхи з більш високим рівнем феромону будуть швидше обрані і, таким чином, посилені, тоді як інтенсивність феромонів, які не обрані, зменшується випаровуванням. Ця форма непрямого спілкування відома як стигмергія та забезпечує можливість пошуку найкоротшого шляху (рисунок 1.12).



1.12 – Принцип роботи мурашиного алгоритму.

В ACO працюють штучні мурахи, які співпрацюють, щоб знайти кращі рішення для дискретних задач оптимізації. Ці програмні агенти імітують харчову поведінку своїх біологічних аналогів у пошуку найкоротшого шляху до джерела їжі. Першим алгоритмом, що слідує принципам метаевристики ACO, є Ant System [14], де мурахи ітеративно будують розчини і додають феромон до шляхів, що відповідають цим рішенням. Вибір шляху - це стохастична процедура, що базується на двох параметрах - феромоні та евристичних значеннях. Значення феромону вказує на кількість мурах, які нещодавно обрали шлях, тоді як евристичне значення є мірою якості, яка залежить від проблеми. Коли мураха досягає точки прийняття рішення, швидше за все обирає слід із вищими феромонними та евристичними значеннями. Після того, як мураха прибуває до місця призначення, оцінюється розчин, що відповідає шляху, яким рухався мураха, і відповідно збільшується значення феромонів шляху. Крім того, випаровування призводить до того, що рівень феромонів у всіх слідах поступово зменшується. Отже, неармовані стежки поступово втрачають феромон, і в свою чергу матимуть меншу ймовірність бути обраними наступними мурахами.

Перша реалізіція ACO для задачі класифікації була запропонована в 2001 році і називалась AntMiner [13]. Псевдокод цього алгоритму представлено нижче.

TrainingSet = {усі тренувальні зразки};
DiscoveredRuleList = []; / список правил */*
WHILE (TrainingSet ≥ Max_Uncovered_Cases)
i = 1; / індекс мурахи */*
No_Ants_Converg = 1; / індекс збіжоісті тесту */*
Ініціалізуються всі стежки однаковою кількістю феромону;
REPEAT
Ant_i починається з порожнього правила і поступово створює правило класифікації R_i, додаючи по одному елементу до поточного правила; Ініціалізується равило обрізання R_i;
Оновіть феромони на всіх слідах, збільшуючи кількість феромоу на шляху, відповідно до Ant_i (пропорційно якості R_i) і зменшуючи кількість феромону для інших шляхів (імітація випаровування феромонів);
IF ($R_i = R_{i-1}$) / оновлення умови збіжності */*
THEN No_Ants_Converge = No_Ants_Converge + 1;
ELSE No_Ants_Converge = 1;
END IF
i = i + 1;
UNTIL (i ≥ No_of_Ants) OR (No_Ants_Converg ≥ No_Rules_Converg)
Обираємо найкраще правило R_{best} серед усіх правил R_i побудованих мурахами;
Додаємо правило R_{best} у DiscoveredRuleList;
TrainingSet = TrainingSet - {сукупність зразків охоплених відповідно до R_{best}};
END WHILE

Коли кількість випадків, що залишились у навчальному наборі, менше *Max_uncovered_cases*, пошук правил зупиняється. На даний момент система виявила кілька правил. Виявлені правила зберігаються в упорядкованому списку правил (у порядку відкриття), який буде використовуватися для класифікації нових випадків, небачених під час навчання. Система також додає правило за замовчуванням до останньої позиції списку правил. Правило за замовчуванням має порожній антецедент (тобто жодної умови) і, як

наслідок, передбачає більшість класів у наборі навчальних екземплярів, на які не поширюється жодне правило. Це правило за замовчуванням застосовується автоматично, якщо жодне з попередніх правил у списку не охоплює новий випадок, який слід класифікувати.

Після того, як список правил буде заповнений, система нарешті готова класифікувати новий тест, невидимий під час навчання. Для цього система намагається застосовувати виявлені правила по порядку. Застосовується перше правило, яке охоплює новий випадок - тобто екземпляру присвоюється клас, передбачений відповідним правилом.

Висновок за проведеним аналізом

Проаналізувавши існуючи рішення для задачі класифікації, були визначені основні переваги і недоліки кожного з них.

На основі переваг та перспектив, які мають метаевристичні алгоритми на основі ройового інтелекту, було обрано для дослідження новий алгоритм бджолиної колонії для його імплементації у сфері інтелектуального аналізу даних, а саме в задачі класифікації.

2. АЛГОРИТМ БДЖОЛИНОЇ КОЛОНІЇ ДЛЯ ІНТЕЛЕКТУАЛЬНОГО АНАЛІЗУ ДАНИХ

2.1. Алгоритм бджолиної колонії

Алгоритм блджолиної колонії (artificial bee colony, ABC) - це оптимізаційна техніка, що імітує поведінку медоносних бджіл на корм і успішно застосовується до різних практичних проблем. ABC належить до групи алгоритмів ройового інтелекту і був запропонований Карабогою в 2005 році [2].

Він складається з трьох важливих компонентів: джерел їжі, зайнятих фуражирів та безробітних фуражирів. Існує дві основні форми поведінки: вербування до джерела їжі та відмова від джерела їжі.

1) Джерела їжі: це позиція рішення проблеми оптимізації, прибутковість джерела їжі виражається як придатність рішення.

2) Безробітний фуражир: їх є два типи - розвідники та спостерігачі. Їх головним завданням є дослідження та експлуатація джерела їжі. На початку є два варіанти для безробітних фуражників: (i). він стає розвідником - хаотично шукає нові джерела їжі навколо гнізда; (ii). Він стає спостерігачем - визначає кількість нектару джерела їжі після перегляду танцювальних танців зайнятої бджоли та вибирає джерело їжі відповідно до прибутковості.

3) Зайняті фуражири: медоносні бджоли, які знайшли джерело їжі, дорівнюють кількості джерел їжі. Зайняті бджоли зберігають інформацію про джерело їжі та діляться з іншими з певною ймовірністю. Зайнята бджола стане розвідником, коли джерело їжі буде вичерпано.

Нижче приведені основні кроки алгоритму (схематично зображено на рисунку 2.1).

1. Початкові джерела їжі виробляються для всіх зайнятих фуражирів

REPEAT

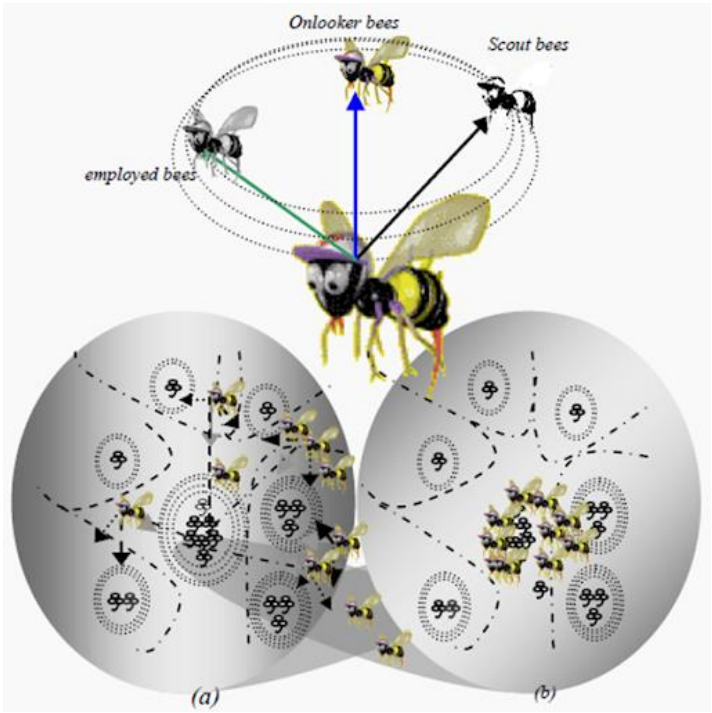
2. Кожна зайнята бджола переходить до джерела їжі в її пам’яті і визначає найближче джерело, потім оцінює кількість нектару і танцює у вулику.

3. Кожен спостерігач спостерігає за танцями зайнятих бджіл і вибирає одне з їх джерел залежно від танців, а потім переходить до цього джерела. 4. Вибравши навколо цього сусіда, вона оцінює кількість нектару.

5. Покинуті джерела їжі визначаються і замінюються новими джерелами їжі, виявленими розвідниками.

6. Зареєстровано найкраще джерело їжі, яке було знайдено.

UNTIL (умова зупинки алгоритму)



2.1 – Принцип роботи бджолиного алгоритму.

В алгоритмі ABC, заснованому на популяції, положення джерела їжі представляє можливе рішення проблеми оптимізації, а кількість нектару джерела їжі відповідає якості (придатності) відповідного рішення. Кількість зайнятих бджіл дорівнює кількості розчинів у популяції. На першому кроці формується випадково розподілена початкова популяція (позиції джерел їжі). Після ініціалізації популяція піддається повторення циклів процесів пошуку зайнятих, спостерігачів та бджіл-розвідників відповідно. Зайнята бджола виробляє модифікацію положення джерела в її пам’яті та виявляє нове положення джерела їжі. За умови, що кількість нектару в новому є більшим, ніж у попередньому джерелі, бджола запам’ятовує нове положення джерела і забуває старе. В іншому випадку вона зберігає позицію тієї, що

запам'ятовується. Після того, як усі зайняті бджоли завершують процес пошуку, вони діляться інформацією про місце розташування джерел з глядачами на так званій танцювальній зоні. Кожен спостерігач оцінює інформацію про нектар, взяту від усіх зайнятих бджіл, а потім вибирає джерело їжі залежно від кількості джерел нектару. Як і у випадку із зайнятою бджолою, вона вносить зміни в положення джерела в своїй пам’яті та перевіряє кількість нектару. За умови, що його нектар вище, ніж у попереднього, бджола запам’ятовує нове положення і забуває старе. Визначені покинуті джерела визначаються і випадковим чином створюються нові джерела, які замінюються на покинуті штучними розвідниками.

В алгоритмі ABC перша половина рою складається із зайнятих бджіл, а друга половина - бджіл-спостерігачів.

Кількість зайнятих бджіл або бджіл-спостерігачів дорівнює кількості розчинів у рої. ABC генерує випадково розподілену початкову популяцію розчинів *SN* (джерел їжі), де *SN* позначає розмір рою.

Нехай $X_i = \{x_{i,1}, x_{i,2}, \dots, x_{i,n}\}$ представляє собою *i*-тий розв’язок в рої, де n – розмірність.

Кожна робоча бджола X_i генерує нового кандидата для розв’зку V_i в області його теперішнього положення відповідно до рівняння:

$$u_{i,k} = x_{i,k} + \Phi_{i,k} \cdot \left(x_{i,k} - x_{j,k}\right) \quad (1.1)$$

Де X_j – це випадково обраний кандидат для розв’зку $(i \neq j)$, k – це індекс випадкової розмірності, вибраний з набору *{1, 2, ..., n}*, і $\Phi_{i,k}$ – випадкове число у інтервалі *[-1, 1]*. Коли новий кандидат розв’зку V_i сгенеровано, використовується жадібний відбір. Якщо значення функції пристосованості (фітнес) V_i більше ніж у батьківської X_i, оновлюємо X_i разом з V_i; інакше залишаємо X_i без змін. Після того, як усі зайняті бджоли завершують процес пошуку; вони діляться інформацією про свої джерела їжі з бджолами-спостерігачами за допомогою танців. Бджола-спостерігач оцінює інформацію про нектар, взяту від усіх зайнятих бджіл, і вибирає джерело їжі з

імовірністю, пов'язаною з кількістю нектару. Цей імовірнісний вибір насправді є механізмом вибору колеса рулетки, який описаний як рівняння нижче:

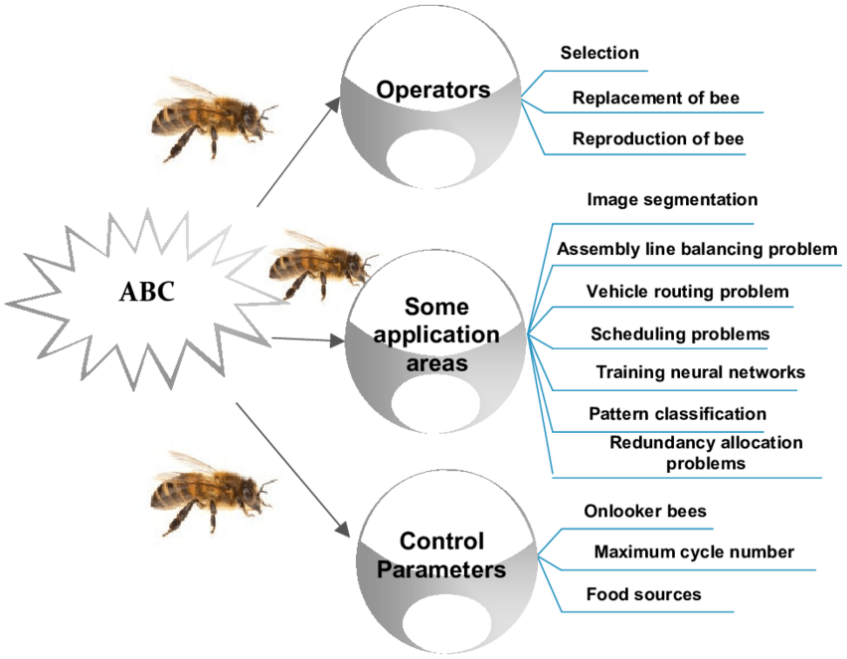
$$P_i = \frac{fit_i}{\sum_j fit_j} \quad (1.2)$$

Де fit_i це значення фітнесу i-того розв'зку в рої. Як видно, чим краще рішення i, тим вища ймовірність обраного i-того джерела їжі. Якщо положення неможливо покращити за заздалегідь визначену кількість циклів, це означає, що джерело їжі відмовляється. Припустимо, що покинутим джерелом є X_i, і тоді бджола-розвідник виявляє нове джерело їжі, яке слід замінити на i-те, як рівняння нижче:

$$x_{i,k} = lb_k + \Phi_{i,k} \cdot (ub_k - lb_k) \quad (1.3)$$

Де $\Phi_{i,k} = rand(0,1)$ – випадкове число в інтервалі *[0,1]* на базі нармального розподілу, і lb_k, ub_k – нижня і верхня межі k-тої розмірності відповідно.

Основні складові алгоритму ABC модна побачити на рисунку 2.2.



2.2 – Основні складові алгоритму ABC.

2.2. Застосування ABC для задачі класифікації

У методі класифікації правило класифікації для кожного атрибута містить дві частини: попередню та наступну. Наступний формат можна побачити на рисунку 2.3.

Feature 1		Feature 2		.....	Feature N	
Lower Bound	Upper Bound	Lower Bound	Upper Bound		Lower Bound	Upper Bound

Рисунок 2.3 – Формат правила класифікації.

Де функції (features на рисунку 2.2.1) від 1 до N - це всі атрибути набору даних. Кожен атрибут має нижню межу, яка є найнижчим значенням для цього правила, і верхня межа, яка є найвищим значенням для цього правила. З правилом класифікації пов’язані ще три значення: прогностичний клас (клас X), значення придатності та відсоток покриття правила. Ці три числа мають тісний взаємозв'язок з функцією фітнесу та стратегією прогнозування.

З наведеного вище пояснення, кожне правило класифікації розроблено як структура нижче (версія на мові C):

```
Struct RuleSet{
        Double * lowb; // нижня межа для всіх атрибутів
        Double * upb; // верхня межа для всіх атрибутів
        Char cName[50]; // передбачувальний класс X
        Double prec; // відсоток покриття
        Double * fitvalue; // значення fitness-функції
};
```

Для оцінки величини придатності для класифікації буде використана fitness-функція замість вимірювання кількості нектару. Її представлення визначено, як показано нижче:

$$fit = \frac{TP}{TP + FN} \times \frac{TN}{TN + FP} \quad (2.1)$$

Де TP, FN, FP та TN - це кількість різних типів записів, що представляють справжні позитивні результати, помилкові негативи,

помилкові позитиви та справжні негативи, пов'язані з правилом відповідно. Перш ніж вводити ці чотири значення, буде пояснено дві важливі концепції:

- Коли алгоритм перевіряє тип запису, він вимірює кожну функцію в записі. Якщо значення ознаки знаходиться між нижньою та верхньою межею для цієї ознаки, це означає, що ознака може бути охоплена правилом. Якщо правило може охоплювати всі функції запису, це означає, що правило може охоплювати запис.
- Якщо клас оцінюваного запису за правилом дорівнює класу передбачення, це означає, що запис має клас, передбачений правилом.

Істинно позитивні дані (TP): кількість записів, охоплених правилом, які мають клас, передбачений правилом;

Хибно негативні (FN): кількість записів, на які не поширюється правило, але вони мають клас, передбачений правилом;

Хибно позитивні (FP): кількість записів, охоплених правилом, але їх клас не передбачається правилом;

Істинно негативні (TN): кількість записів, на які не поширюється правило, і які не мають класу, передбаченого правилом.

Коли зайнята бджола не відповідає вимогам або не досягає максимального числа циклів, їй потрібно перейти до нового джерела їжі з дотриманням місцевої стратегії пошуку. Оригінальний алгоритм ABC використовує рівняння 2.1 для реалізації методу локального пошуку. Однак це забирає величезну кількість часу, коли класифікований набір даних містить велику кількість даних і не підходить для класифікаційних програм. Зважаючи на те, що час є важливим фактором класифікації та підвищення точності, пропонується нову простішу стратегію локального пошуку, щоб замінити початкову стратегію локального пошуку.

$$V_{i,j} = X_{k,j} \quad (2.2)$$

Де $V_{i,j}$ представляє позицію нового джерела їжі та $X_{k,j}$ означає сусіда попереднього джерела їжі. Число i та k знаходяться між 1 і SN, проте k

повинен бути різним з i. Крім того, j – величина розмірності. У задачі класифікації розмірність набору даних дорівнює кількості об'єктів набору даних. $k \in [1,2,\dots,SN]$ і $j \in [1,2,\dots,D]$ – випадково вибрані параметри. Коли поміщаємо цю стратегію в область класифікації, рівняння 2.2 змінюється на такі функції:

$$V_{i,j}(lb) = X_{k_1,j}(lb) \quad (2.3)$$

$$V_{i,j}(ub) = X_{k_2,j}(ub) \quad (2.4)$$

Де k_1 і k_2 – два видакових числа, які не дорівнюють i.

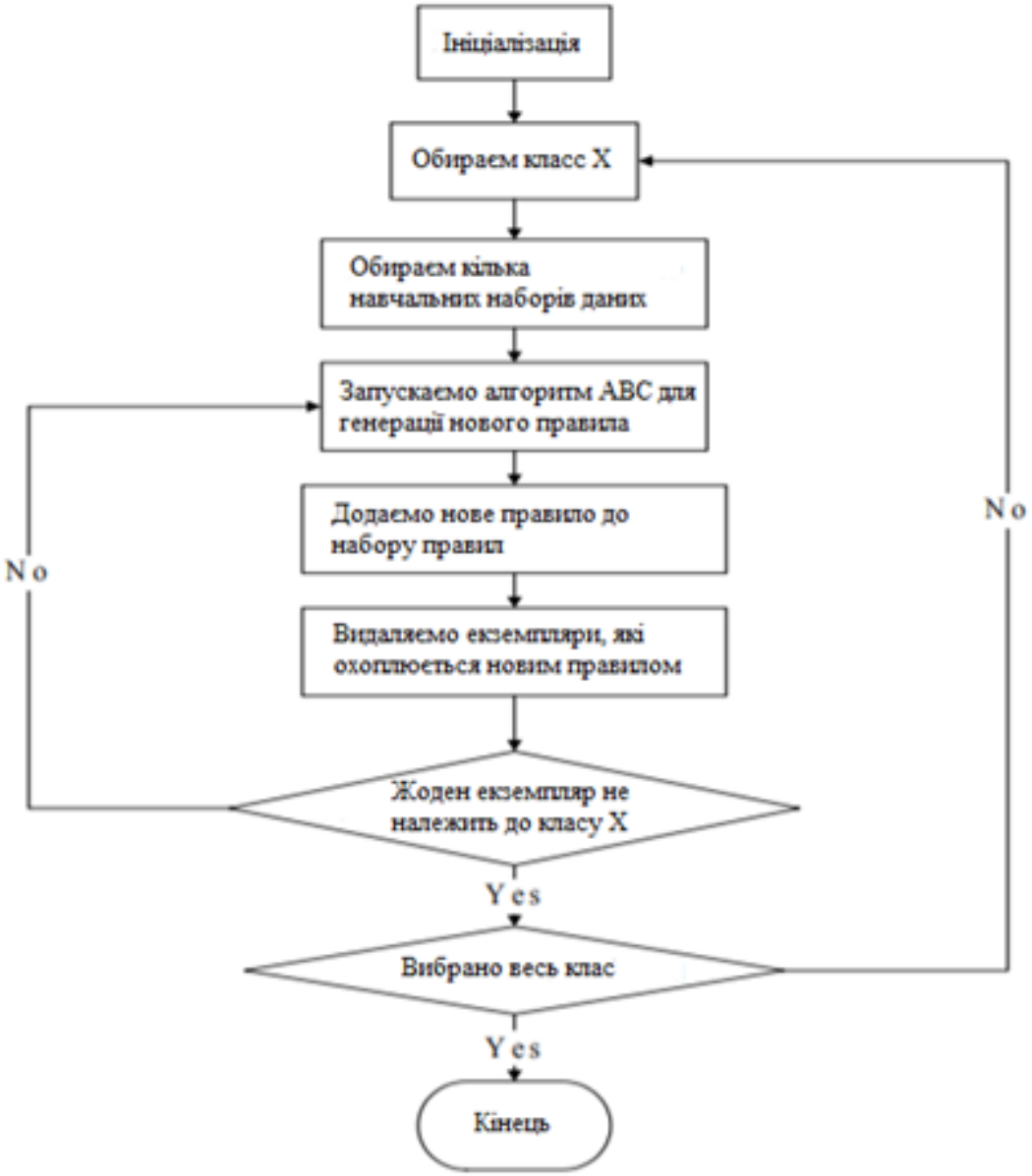


Рисунок 2.4. – Виявлення правил в запропонованому алгоритмі.

Метою видобутку правил класифікації є пошук набору правил, які можуть ідентифікувати конкретний клас з різних груп. Таким чином, фаза виявлення правил є найважливішою для класифікаційного алгоритму, оскільки набори правил є результатом цієї фази. Виявлення правил щодо алгоритму ABC показано на блок-схемі на рисунку 2.4.

На етапі ініціалізації ми встановлюємо значення нижньої межі та значення верхньої межі для кожного атрибута. Процедура визначена у рівняннях 2.5 та 2.6, як показано нижче:

$$lb = f - k_1 x(F_{max} - F_{min}) \quad (2.5)$$

$$ub = f + k_2 x(F_{max} - F_{min}) \quad (2.6)$$

Для цих двох рівнянь F_{max} та F_{min} є, відповідно, максимальним значенням та мінімальним значенням ознаки. Різниця між ними означає діапазон ознаки. f - початкове значення ознаки. k_1 і k_2 - це два випадкові значення між *0* і *1*.

Алгоритм видобутку правил класифікації може автоматично виявляти правила для кожного класу. Для вибраного класу він буде знаходити правила ітеративно, поки набір правил не охоплює всі екземпляри, що належать до цього класу. Кожне окреме правило дотримується структури, і набір складається з багатьох правил.

Після обробки всіх класів і генерування всіх наборів правил кожне правило буде включено в процедуру обрізки. Основною метою обрізки правил є усунення зайвого обмеження функцій, яке могло б бути непотрібним, включеним до наборів правил. Оскільки деякі відносні ознаки негативно вплинуть на результат класифікації, правило обрізки може підвищити точність. Процес обрізки показаний на рисунку 2.5. Цей процес буде повторюватися, поки не будуть оцінені всі правила з набору правил.

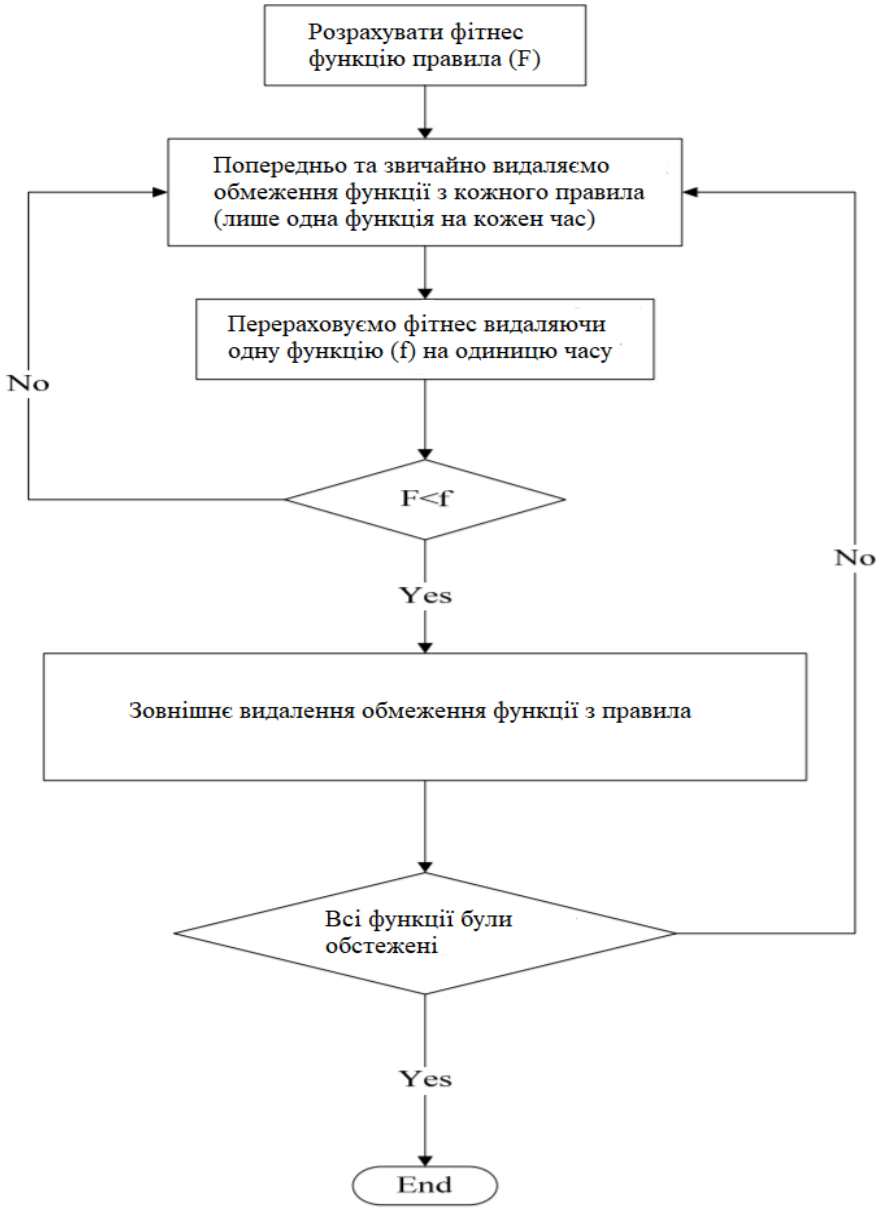


Рисунок 2.5 – Схема процедури виявлення правила.

Остаточний набір правил буде використаний для прогнозування нових даних, для яких їх класи невідомі. Але іноді один запис даних тестування охоплюватиме не одне правило для іншого класу. Коли це сталося, стратегія передбачення визначатиме, який клас слід передбачати. Існує три основні кроки для підходу прогнозування, які вказані наступним чином:

1) Обчислити значення прогнозу для всіх правил, які охоплюють запис даних тесту.

2) Накопичити ці значення прогнозування відповідно до різних можливих класів.

3) Виберіть клас, який має найвище значення прогнозування, як кінцевий клас.

Після процедури стратегії передбачення ядром є функція передбачення, яка використовується для обчислення значення передбачення для кожного правила. Це визначено в рівнянні 2.7 наступним чином:

$$prediction = (\alpha \times rule\ fitness) + (\beta \times rule\ cover) \quad (2.7)$$

Де α і β – два зважені параметри, пов'язані зі значенням придатності та відсотком покриття правила, $\alpha \in [0,1]$ і $\beta = (1 - \alpha)$. Рівняння 2.1 може обчислити значення придатності для кожного правила. Відсоток покриття визначає, що частка записів, які охоплюються правилом, мають клас, передбачений правилом (TP). Він обчислюється за виразом, показаним у рівнянні 2.8:

$$Cover\ percentage = \frac{TP}{N}$$

Де N - загальна кількість записів, які належать до передбачуваного класу за правилами.

Стратегія прогнозування врівноважила ефект значення фітнесу та відсотка покриття для остаточного прогнозованого класу. Потрібно ретельно вибирати значення α та β, оскільки вони вплинуть на точність класифікації.

Новий запропонований підхід забезпечив новий механізм видобутку правил класифікації на основі алгоритму оптимізації ABC. Короткий зміст основної процедури запропонованого алгоритму аналізу даних ABC показано на рисунку 2.6.

На рисунку 2.6 кожен прямокутник представляє один із етапів у процесі класифікації даних, а ромб вказує структуру даних або набір даних, створених попереднім етапом. На етапі ініціалізації встановлюються всі параметри управління, такі як кількість колоній, максимальний цикл та обмежене значення.

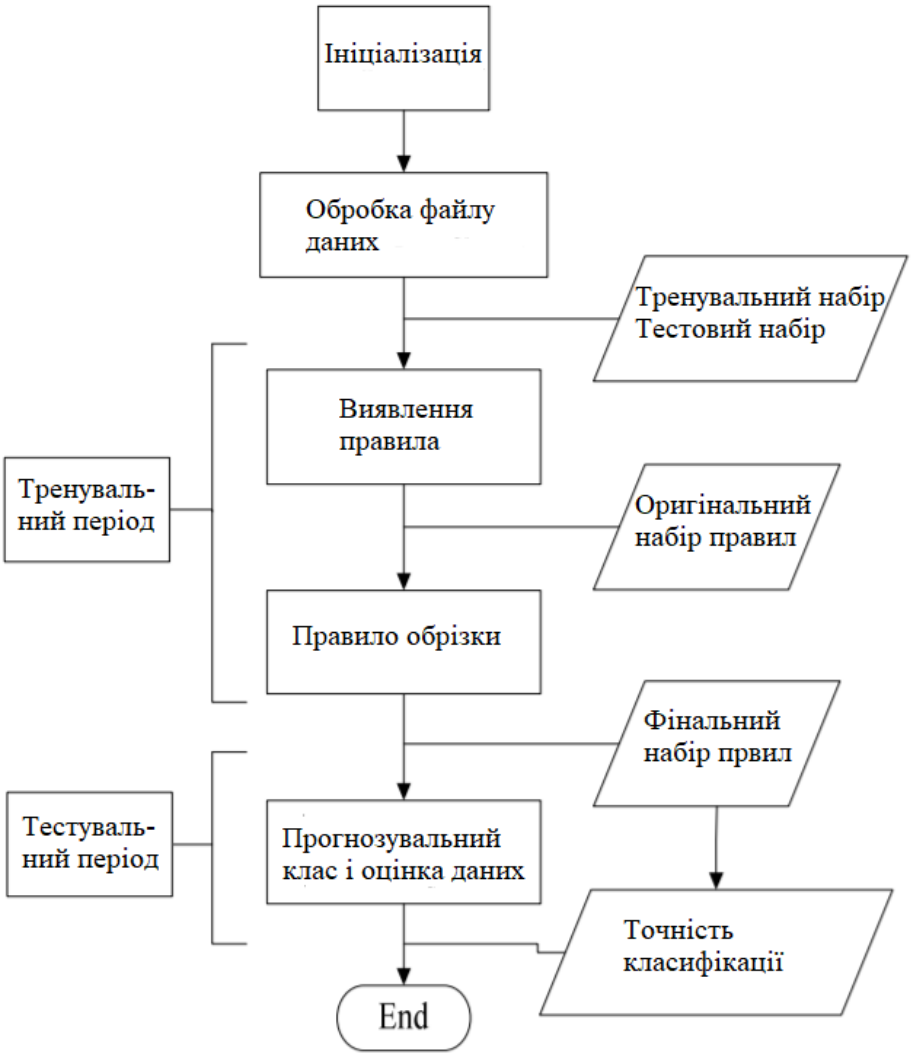


Рисунок 2.6 – Схема класифікації.

Висновок за другим розділом

У цьому розділі було проаналізовано оригінальний алгоритм бджолиного рою, і на основі нього була створена адаптація ABC для задачі клвсифікації. Також було запропоновано нову простішу стратегію локального пошуку задля зменшення кількості обчислень при класифікації.

3. ОПИС РОЗРОБЛЕНОЇ ПРОГРАМИ

3.1. Середовище та компоненти розробки

Мовою програмування, якою написано розроблений алгоритм, було обрано Python, а також відповідну бібліотеку машинного навчання scikit-learn.

Python (найчастіше вживане прочитання — «Пайтон», запозичено назву з британського шоу Монті Пайтон) — інтерпретована об'єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією[17]. Розроблена в 1990 році Гвідо ван Россумом. Структури даних високого рівня разом із динамічною семантикою та динамічним зв'язуванням роблять її привабливою для швидкої розробки програм, а також як засіб поєднування наявних компонентів. Python підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. Інтерпретатор Python та стандартні бібліотеки доступні як у скомпільованій, так і у вихідній формі на всіх основних платформах. В мові програмування Python підтримується кілька парадигм програмування, зокрема: об'єктно-орієнтована, процедурна, функціональна та аспектно-орієнтована.

Серед основних її переваг можна назвати такі:

- чистий синтаксис (для виділення блоків слід використовувати відступи);
- переносність програм (що властиве більшості інтерпретованих мов);
- стандартний дистрибутив має велику кількість корисних модулів (включно з модулем для розробки графічного інтерфейсу);
- можливість використання Python в діалоговому режимі (дуже корисне для експериментування та розв'язання простих задач);
- стандартний дистрибутив має просте, але разом із тим досить потужне середовище розробки, яке зветься IDLE і яке написане мовою Python;
- зручний для розв'язання математичних проблем (має засоби роботи з комплексними числами, може оперувати з цілими числами довільної

величини, у діалоговому режимі може використовуватися як потужний калькулятор);

- відкритий код (можливість редагувати його іншими користувачами).

На рисунку 3.1 можна побачити ієрархію структур даних в python.

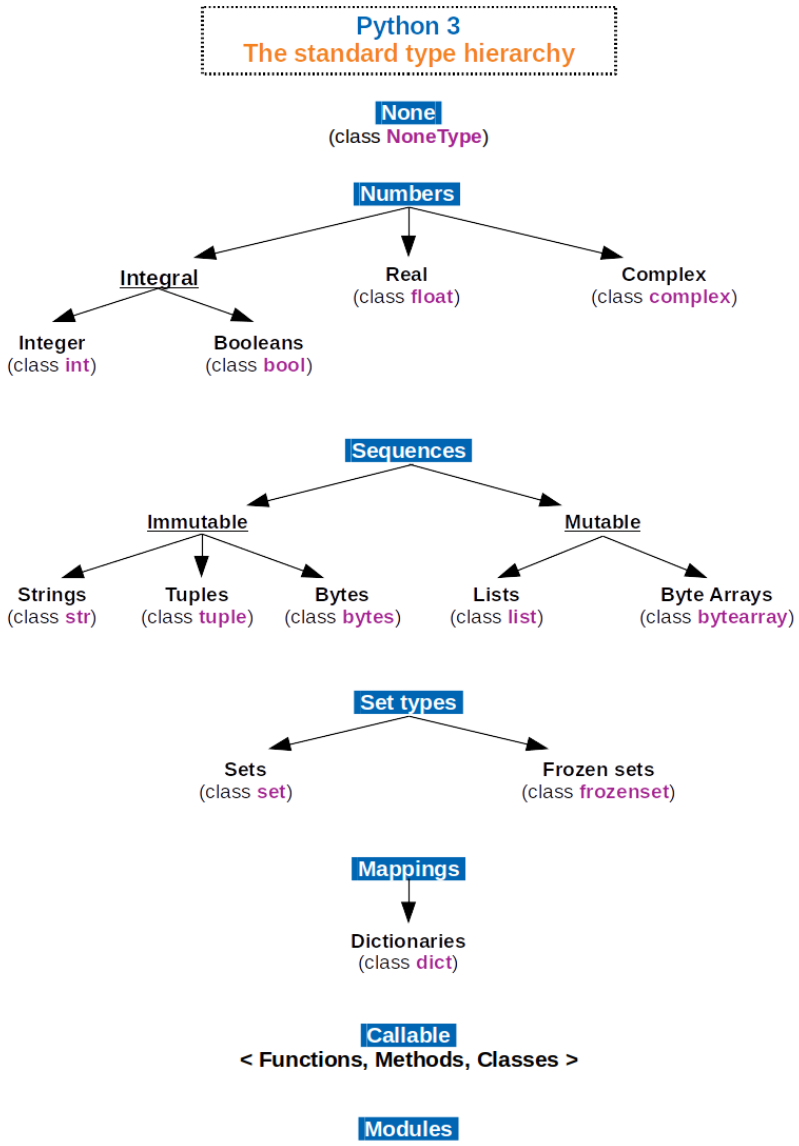


Рисунок 3.1 – Структури даних в python.

Python має ефективні структури даних високого рівня та простий, але ефективний підхід до об'єктно-орієнтованого програмування. Елегантний синтаксис Python, динамічна обробка типів, а також те, що це інтерпретована мова, роблять її ідеальною для написання скриптів та швидкої розробки прикладних програм у багатьох галузях на більшості платформ.

Інтерпретатор мови Python і багата Стандартна бібліотека (як вихідні тексти, так і бінарні дистрибутиви для всіх основних операційних систем) можуть бути отримані з сайту Python www.python.org, і можуть вільно розповсюджуватися. Цей самий сайт має дистрибутиви та посилання на численні модулі, програми, утиліти та додаткову документацію.

Інтерпретатор мови Python може бути розширений функціями та типами даних, розробленими на C чи C++ (або на іншій мові, яку можна викликати із C). Python також зручна як мова розширення для прикладних програм, що потребують подальшого налагодження.

Python підтримує динамічну типізацію, тобто, тип змінної визначається лише під час виконання. З базових типів слід зазначити підтримку цілих чисел довільної довжини і комплексних чисел. Python має багату бібліотеку для роботи з рядками, зокрема, кодованими в юнікоді.

З колекцій Python підтримує кортежі (tuples), списки (масиви), словники (асоціативні масиви) і від версії 2.4, множини.

Система класів підтримує множинне успадкування і метапрограмування. Будь-який тип, включаючи базові, входить до системи класів, й за необхідності можливе успадкування навіть від базових типів.

Дизайн мови Python побудований навколо об'єктно-орієнтованої моделі програмування. Реалізація ООП в Python є елегантною, потужною та добре продуманою, але разом з тим, достатньо специфічною в порівнянні з іншими об'єктно-орієнтованими мовами.

Можливості та особливості:

- Класи є одночасно об'єктами з усіма нижче наведеними можливостями.
- Успадкування, в тому числі множинне.
- Поліморфізм (всі функції віртуальні).
- Інкапсуляція (два рівні — загальнодоступні та приховані методи і поля). Особливість — приховані члени доступні для використання та помічені як приховані лише особливими іменами.

- Спеціальні методи, що керують життєвим циклом об'єкта: конструктори, деструктори, розподільники пам'яті.
- Перевантаження операторів (усіх, крім is, '.', '=' і символьних логічних).
- Властивості (імітація поля за допомогою функцій).
- Управління доступу до полів (емуляція полів і методів, частковий доступ тощо).
- Методи для управління найпоширенішими операціями (істиннісне значення, len(), глибоке копіювання, серіалізація, ітерація по об'єкту).
- Метапрограмування (управління створенням класів, тригери на створення класів, та ін).
- Повна інтроспекція.
- Класові та статичні методи, класові поля.
- Класи, вкладені у функції та інші класи.

Python підтримує парадигму функціонального програмування, зокрема:

- функція є об'єктом;
- функції вищих порядків;
- рекурсія;
- розвинена обробка списків (спискові вирази, операції над послідовностями, ітератори);
- аналог замикань (closures);
- часткове застосування функції;
- можливість реалізації інших засобів на самій мові (наприклад, каррінг).

Багата стандартна бібліотека є однією з привабливостей мови Python. Тут є засоби для роботи з багатьма мережевими протоколами та форматами Інтернету, наприклад, модулі для написання HTTP-серверів та клієнтів, для розбору та створення поштових повідомлень, для роботи з XML, тощо. Набір модулів для роботи з операційною системою дозволяє писати крос-

платформні застосунки. Існують модулі для роботи з регулярними виразами, текстовими кодуваннями, мультимедійними форматами, криптографічними протоколами, архівами, серіалізацією даних, юніт-тестуванням та ін.

Scikit-learn (раніше scikits.learn, а також відомий як sklearn) - це безкоштовна бібліотека машинного навчання для мови програмування Python. Вона містить різні реалізації алгоритмів класифікації, регресії та кластеризації, включаючи SVM, Random forest, k-найближчих сусідів та інші, і призначений для взаємодії з числовими та науковими бібліотеками Python NumPy та SciPy [18].

Проект scikit-learn розпочався як scikits.learn, проект Google Summer of Code від Девіда Курно.

Scikit-learn добре інтегрується з багатьма іншими бібліотеками Python, такими як matplotlib та plotly для побудови графіків, numpy для векторизації масивів, pandas для фреймів даних, scipy та багато іншого.

3.2. Опис розробленої програми

Розглянемо структуру розробленої програми, основою якої є три класи: *OnlookerBee, EmployedBee*, і *ArtificialBeeColony*.

ArtificialBeeColony описує відповідно взаємодію класів *EmployedBee* та *OnlookerBee* і має такі атрибути:

1. *food_sources* – двомірний масив, який представляє список джерел їжі для бджіл, *food_sources* складається з одиниць і нулів, де одиниця відповідає значимому атрибуту для пошуку, а нулі відповідно не значимому;
2. *features* – масив, який складається з атрибутів тестового набору даних;
3. *data* – двовимірний масив, де стовпці це атрибути, а строки це екземпляри даних; матриця представляє собою набір даних для тренування алгоритму;
4. *test_data* – двовимірна матриця, яка представляє собою набір даних для тестування алгоритму;

5. *labels* – масив міток тренувального набору даних, де кожному зразку присвоюється клас, якому він відповідає;
6. *test_labels* – масив міток класів тестового набору даних;
7. *fitness* – найкраще значення фукнції пристосованості для бджіл-розвідників на даній ітерації;
8. *fitnesses* – масив значень функцій пристосованості для кожної бджоли на даній ітерації;
9. *modification_rate* – параметр алгоритму в інтервалі [0, 1], який відповідає за долю розв'язків які модифікуються на кожній ітерації (*modification_rate* є параметром, аналогічним α, описаним у попередньому розділі);
10. *food_sources_num* – кількість джерел їжі;
11. *selected_features* – список обраних атрибутів даних для кожної ітерації;
12. *features_bounds* – масив, що складається з нижньої та верхньої межі (найменшого і найбільшого значень атрибуту) для кожного атрибута;
13. *clf* – дерево класифікації, на якій і відбувається пошук правил;

I наступні методи:

1. *initialize_food_source(food_source_size, food_source_num)* – метод, який ініціалізує нулями *food_sources*, де *food_source_size* – розмір масиву що представляє собою джерело їжі, а *food_source_num* – кількість таких джерел, тобто *food_source_size* і *food_source_num* це параметри, які відповідають за розміри матриці *food_sources*; при ініціалізації кожен рядок має одну одиницю, щоб всі робочі бджоли були зайняті відповідним джерелом їжі;
2. *execute(cycle, target)* – метод, що запускає алгоритм, параметр *cycle* – це кількість ітерації, після яких алгоритм зупиняється, а *target* – це значення точності (accuracy) класифікації, при досягненні якого алгоритм зупиняється; повертає в результаті поточне значення fitness, *selected_features*, *onlooker_bees.best_employed_bee* і *fitnesses*, де *onlooker_bees* – це екземпляр класу *OnlookerBee* на даній ітерації.

Клас *EmployedBee* описує діяльність робочіх (зайнятих) бджіл і складається з таких атрибутів:

1. *current_limit* – лічильник, який збільшується на одиницю при кожній оцінці фітнес-функції;
2. *max_limit* – максимальна кількість розрахувань функції пристосованості до того, як буде змінено джерело їжі;
3. *dataset* – набір даних для тренування алгоритму;
4. *labels* – масив міток класів для тренувального набору даних;
5. *test_data* – набір даних для тестування алгоритму;
6. *test_labels* – масив міток класів тестового набору даних;
7. *features* – список атрибутів тестового набору даних;
8. *current_food_source* – одномірний масив, який представляє собою джерело їжі для бджоли, тобто набір з одиниць і нулів, індекс представляє собою відповідний атрибут вхідного набору даних;
9. *modification_rate* – параметр, відповідаючий за кількіть оновлюємих розв’язків;
10. *clf* – дерево класифікації, на якій і відбувається пошук правил;
11. *y_pred* – набір передбачувальних класів, тобто список, розмір якого відповідає розміру тестових даних, де кожному рядку передбачується деякий клас;
12. *current_fitness* – точність класифікації для даної зайнятої бджоли (тобто значення функції пристосованості);

А також наступні методи:

1. *calculate_fitness()* – функція, яка вираховує значення пристосованності current_fitness;
2. *generate_new_food_source()* – метод, який генерує новий набір current_food_source, тобто оновлює джерело їжі;

Клас *OnlookerBee* описує процесс пошуку для бджіл-спостерігачів складається з таких полей даних:

1. *employed_bees* – масив представляючий зайнятих бджіл (масив екземплярів класу *EmployedBee*);
2. *bees_distribution* – розподіл бджіл, який представляє собою масив екземплярів класу *EmployedBee,* із розрахованою для кожної бджоли ймовірністю відбору у наступний набір бджіл;
3. *food_source_size* – довжина масиву джерела їжі;
4. *best_fitness* – найкраще значення фітнес-функції серед бджіл;
5. *best_food_source* – одномірний масив, який представляє найкраще значеннє джерела їжі;
6. *best_employed_bee* – найкраща зайнята бджола;

І методів:

1. *evaluates_nectar()* – функція, яка оцінює значення фітнес-функції для кожної бджоли за допомогою функції roulette_wheel();
2. *roulette_wheel()* – метод, який оновлює бджіл у списку *employed_bees*, відповідно до значень функції пристосованості;

Отже, для того щоб запустити алгоритм необхідно створити екземпляр класу *ArtificialBeeColony*, передавши параметри *features, data, test_data, labels, test_labels, modification_rate,* і *food_sources_num* після чого запустити метод *execute* з параметрами *cycle, target* і *max_limit*.

Висновок за третім розділом

В даному розділі було описано розроблений програмний додаток, який реалізує запропонований алгоритм ABC для задачі класифікації. Програма була написана на мові програмування python, яка є найбільш пристосованою для задач, зв’язаних з Data Mining і Machine Learning, за рахунок свого простого синтаксису та поширеною системою бібліотек, таких як pandas, numpy, scipy і scikit-learn.

4. РЕЗУЛЬТАТИ ТЕСТУВАННЯ ЗАПРОПОНОВАНОГО АЛГОРИТМУ ТА ПОРІВНЯЛЬНИЙ АНАЛІЗ

4.1. Аналіз параметрів алгоритму

Для аналізу параметрів запропонованого алгоритму використаємо набір даних *german*, який складається з 1000 рядків та 24 атрибутів, який необхідно розділити на 2 класи (русунок 4.1).

	Atribut1	Atribut2	Atribut3	Atribut4	Atribut5	...	Atribut21	Atribut22	Atribut23	Atribut24	Keterangan
0	1	6	4	12	5	...	1	0	0	1	1
1	2	48	2	60	1	...	1	0	0	1	2
2	4	12	4	21	1	...	1	0	1	0	1
3	1	42	2	79	1	...	0	0	0	1	1
4	1	24	3	49	1	...	0	0	0	1	2
..	...	...	...	...	...	...	...	...	...	...	...
995	4	12	2	17	1	...	1	0	1	0	1
996	1	30	2	39	1	...	1	0	0	0	1
997	4	12	2	8	1	...	1	0	0	1	1
998	1	45	2	18	1	...	0	0	0	1	2
999	2	45	4	46	2	...	1	0	0	1	1

[1000 rows x 25 columns]

Рисунок 4.1 – Тестовий набір даних для аналізу параметрів.

	precision	recall	f1-score	support
1	0.79	0.86	0.82	350
2	0.59	0.45	0.51	150
accuracy			0.74	500
macro avg	0.69	0.66	0.67	500
weighted avg	0.73	0.74	0.73	500

Рисунок 4.2 – Розширений результат запуску алгоритму.

Результатом роботи алгоритму можуть бути як звичайний, так і розширений вивід (це задається відповідним параметром при запуску алгоритму). Звичайним є значення точності класифікації (accuracy) відповідно до формули 2.1, яка представлена у 2 розділі даної роботи. На рисунку 4.2 можна побачити розширений вивід, який складається з значень:

- *precision* – це прогностична значущість позитивних результатів, тобто відношення істинно позитивних результатів класифікації до всіх класифікованих даних;

- *recall* – це частка загального числа позитивних зразків, яку було дійсно знайдено;
- *f1-score* – це середнє гармонійне значень *precision* і *recall*;
- *support* – це кількість даних що потребують класифікації відповідно до кожного класу, а також загальна кількість таких даних.

Наприклад з рисунка 4.2, кількість рядків, які мають клас 1, дорівнює 350, а відповідно клас 2 – 150. Загальна кількість даних, що потребують класифікації дорівнює 500 (ця кількість задається параметром *test_size* при запуску), отже тренувальний набір складається з 1000-500=500 записів. Також виводиться макро-середні та сважено-середні для параметрів *precision, recall* та *f1-score*.

Спочатку визначимо залежність точності від розміру тестового і тренувального наборів (параметр *test_size*).

Отже *test_size* представляє собою частку тестового набору від всього набору даних, тобто якщо *test_size=0.5* це означає що набір даних поділений навпіл, де одна половина – тренувальний набір, інша – тестовий.

```
Test Size: 0.1
              precision    recall  f1-score   support

           1       0.81      0.95      0.87        75
           2       0.67      0.32      0.43        25

    accuracy                           0.79       100
   macro avg       0.74      0.63      0.65       100
weighted avg       0.77      0.79      0.76       100

Test Size: 0.9
              precision    recall  f1-score   support

           1       0.77      0.85      0.81       634
           2       0.53      0.39      0.45       266

    accuracy                           0.72       900
   macro avg       0.65      0.62      0.63       900
weighted avg       0.70      0.72      0.70       900
```

Рисунок 4.3 – Розширений вивід для test_size=0.1 і test_size=0.9

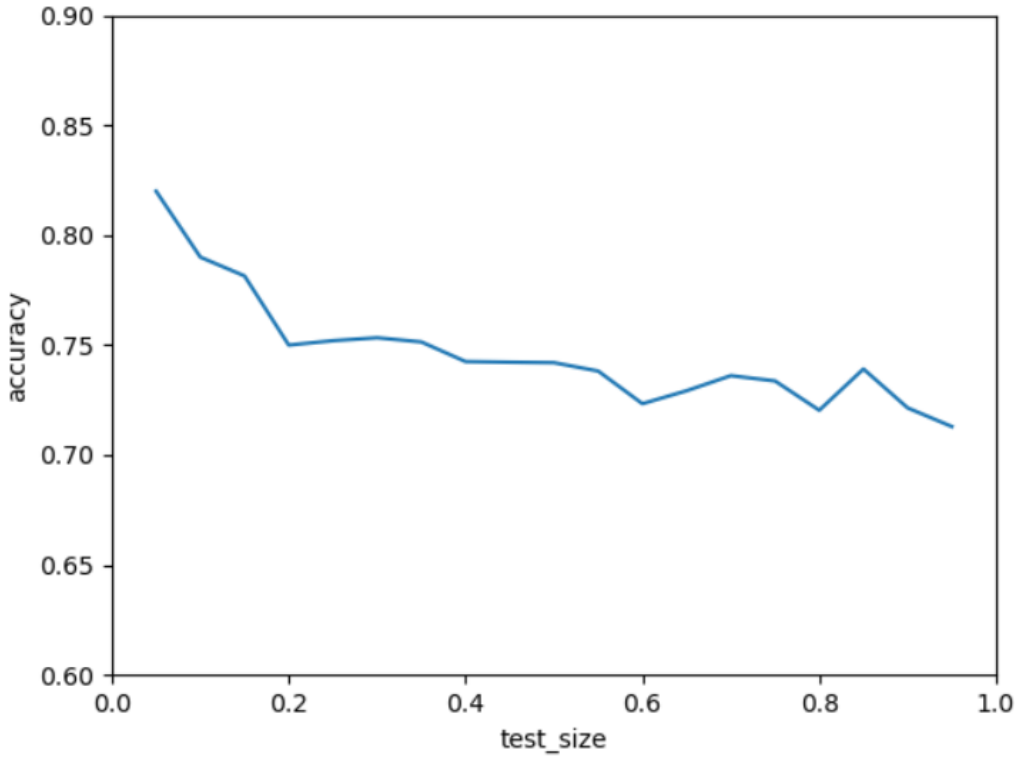


Рисунок 4.4 – Графік залежності точності від *test_size*.

З рисунків 4.3 і 4.4 видно, що залежність точності від *test_size* є обернено пропорційним, тобто чим більше відношення розміру тренувального набору до тестового, тим більша точність класифікації.

Для оцінки точності результатів тестування також використовується *K-fold* перехресне тестування. У *K*-кратному перехресному тестуванні вихідний набір даних випадковим чином поділяється на *K* підмножини. Період навчання та тестування триватиме *K* разів. Для кожного разу в якості даних перевірки для тестування використовується одна підмножина (тестуючий набір), а решта (*K-1*) підмножини зберігаються як дані навчання. Крім того, кожен з підмножин *K* використовується лише один раз як дані перевірки. Перехресна перевірка є найбільш часто використовуваним алгоритмом для перевірки точності класифікації [16]. Після розділення двох наборів даних навчальний набір буде використовуватися під час період навчання і метою є знайти набір правил класифікації. Нарешті, для оцінки алгоритму, обчислюється точність для кожного набору даних тесту та обчислюється середнє значення *K* тестувань як остаточну точність (приклад на рисунку 4.5).

```
Score Fold-1: 0.68
Score Fold-2: 0.65
Score Fold-3: 0.71
Score Fold-4: 0.68
Score Fold-5: 0.64
Score Fold-6: 0.69
Score Fold-7: 0.72
Score Fold-8: 0.74
Score Fold-9: 0.72
Score Fold-10: 0.72
Score Average-: 0.695
```

Рисунок 4.5 – Перехресне тестування для *K =10*, де *Score Fold* – точність для кожного тестування, *Average* – середнє.

Проаналізуємо залежність точності, а також сталості (середнє відхилення результатів точності) алгоритму від параметрів *cycle*, *max_limit*, *modification_rate* і *food_sources_num* використовуючи *K-fold* перехресне тестування зі значенням *K =10*. Тут і надалі буде використовуватись саме середнє значення за *10* тестувань.

1) Для того щоб проаналізувати параметр *cycle*, було виконано 7 тестувань з такими значеннями [1, 2, 3, 4, 6, 9, 14], інші параметри залишались незмінними і були встановлені наступні значення *max_limit=3*, *modification_rate=0.3* і *food_sources_num=40*.

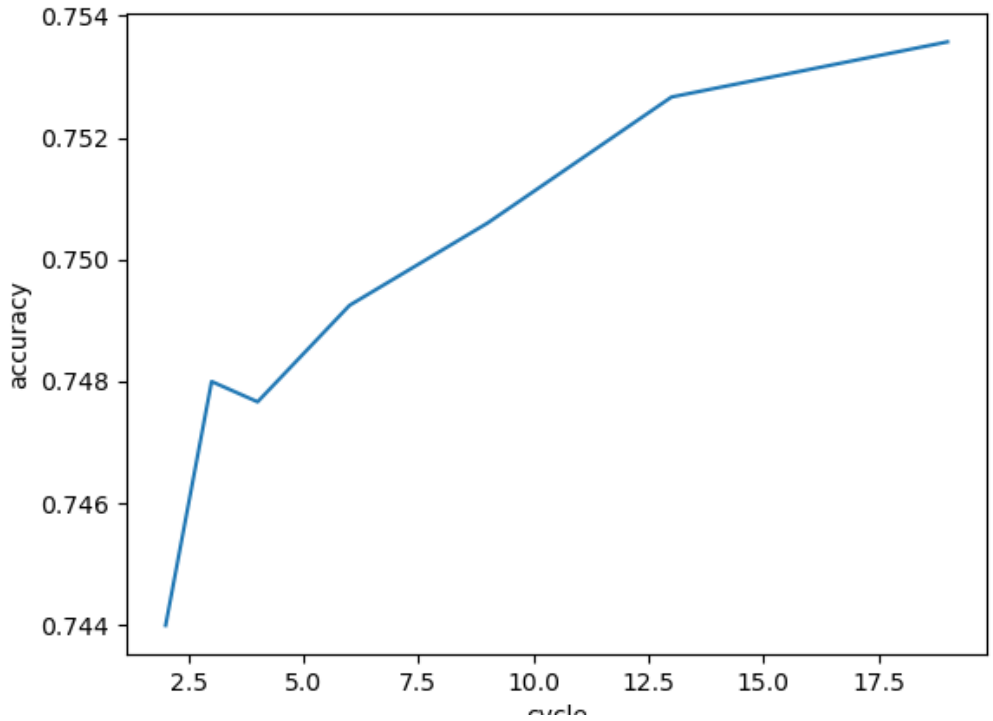


Рисунок 4.6 – Графік залежності точності від *cycle*.

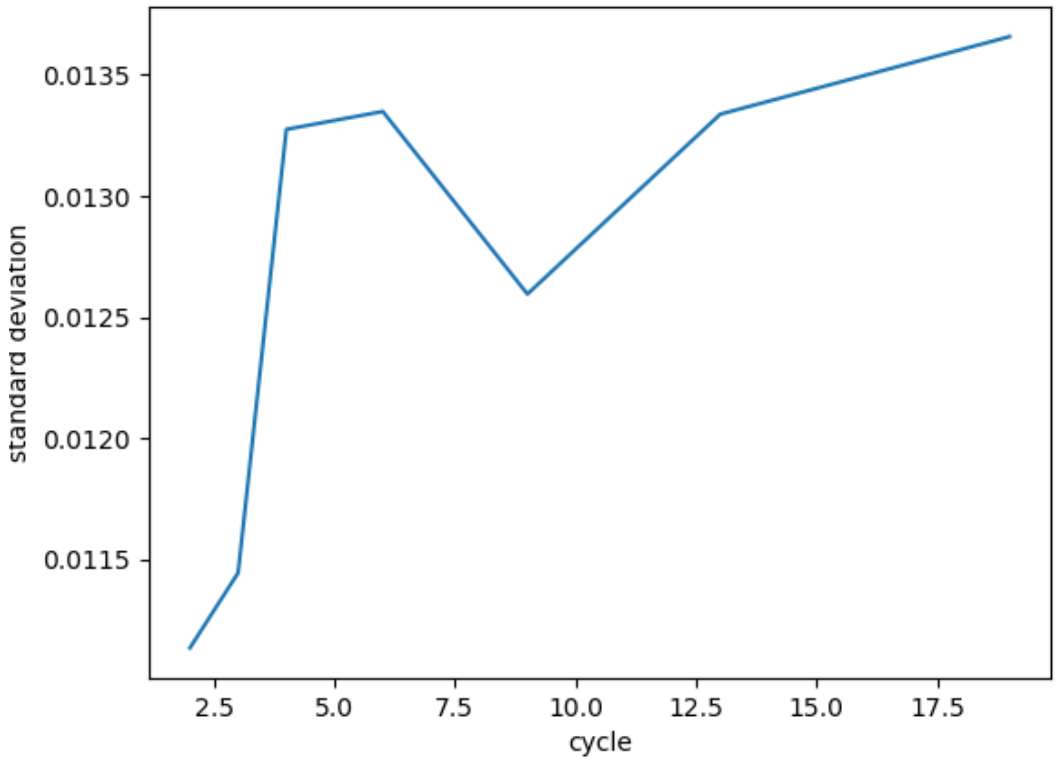


Рисунок 4.7 – Графік залежності стандартного відхилення від *cycle*.

Точність запропонованого алгоритму збільшується при збільшенні *cycle* (рисунок 4.6), проте зі збільшенням значення *cycle*, швидкість зростання точності зменшується. Стандартне відхилення збільшується по мірі збільшення *cycle* (рисунок 4.7). Найкращими показники досягаються при значеннях *cycle* від 7 до 10.

2) Для того щоб проаналізувати параметр *max_limit*, було виконано 6 тестувань з значеннями [1, 2, 3, 4, 6, 9] , інші параметри залишались незмінними і були встановлені наступні значення *cycle=2*, *modification_rate=0.3* і *food_sources_num=40*.

Як можна побачити з рисунка 4.8, *accuracy* прямо пропорційно залежить від *max_limit*. З графіка на рисунку 4.9 видно, що найменше значення відхилення досягається при *max_limit*=6, проте близькими до мінімума є і знначення 2 та 3. Отже оскільки при *max_limit*=6, кількість обчислень зростає приблизно в 2 рази в порівнянні з *max_limit*=3, а точність збільшилась лише на 0.002, отже оберемо значення *max_limit*=3 для подальших тестувань.

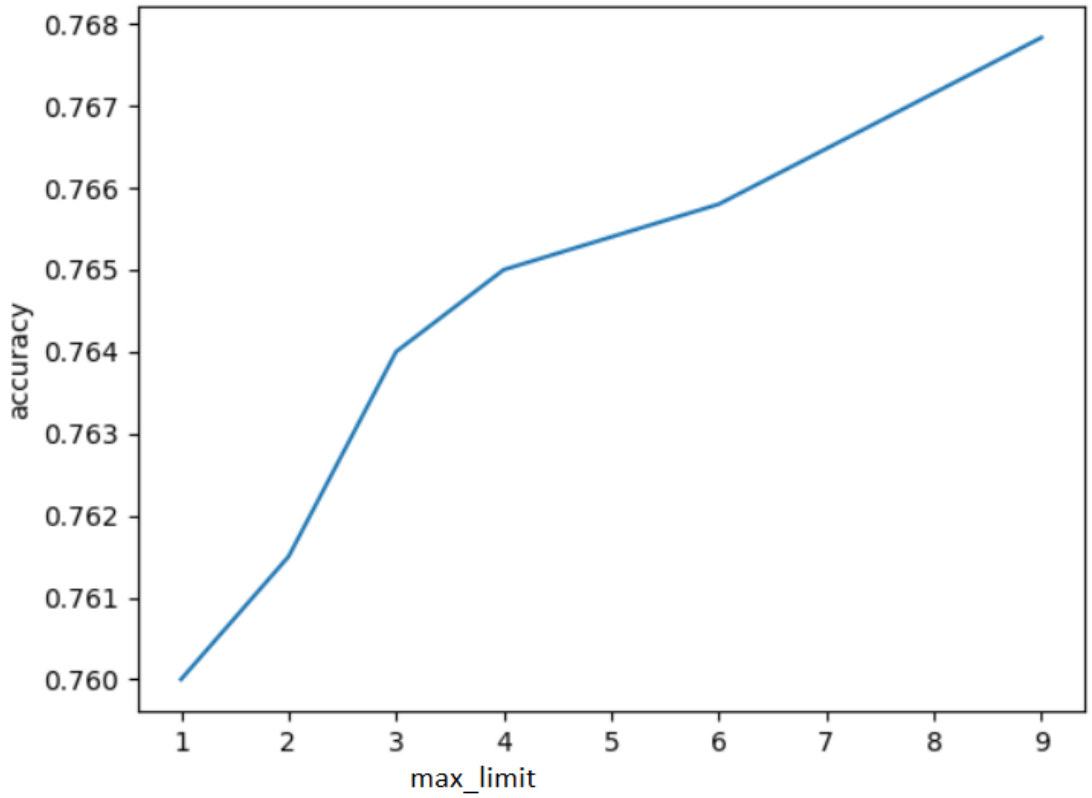


Рисунок 4.8 – Графік залежності точності від *max_limit*.

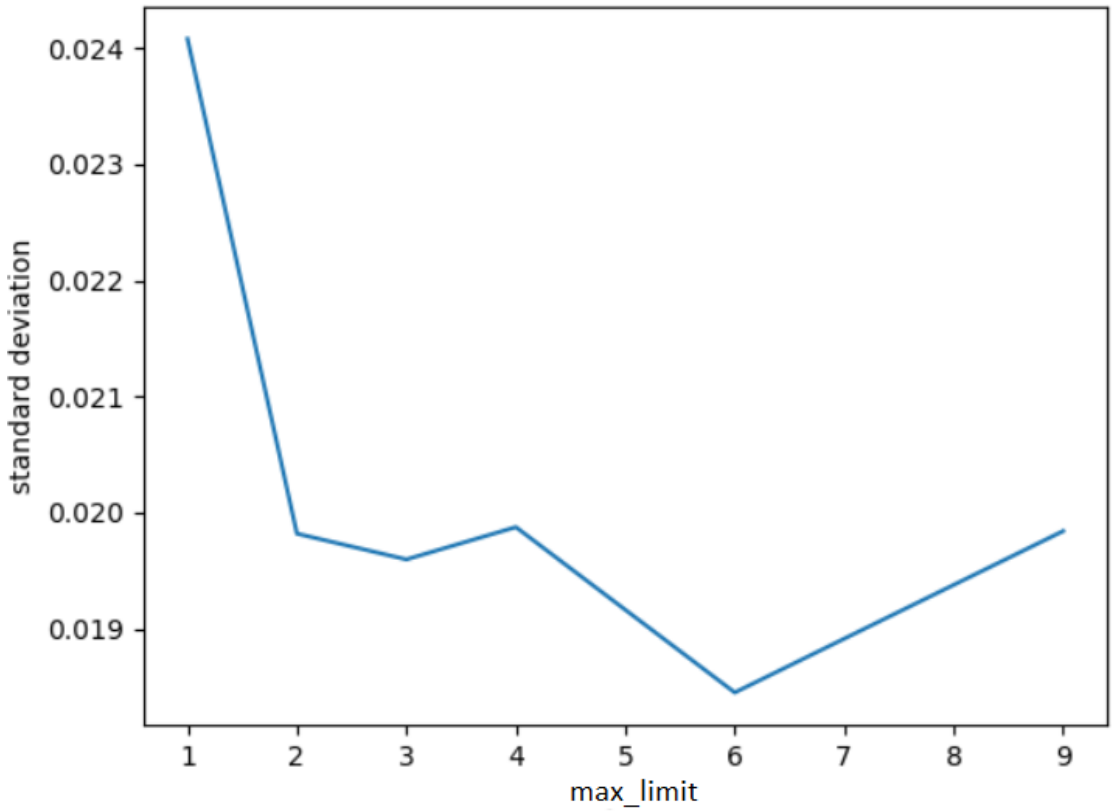


Рисунок 4.9 – Графік залежності відхилення від *max_limit*.

3) Для того щоб проаналізувати параметр *modification_rate*, було виконано 19 тестувань зі значеннями в інтерівалі [0.05, 0.95] і з кроком 0.05, інші параметри залишались незмінними і були встановлені наступні значення *max_limit=3*, *modification_rate=0.3* і *food_sources_num=40*.

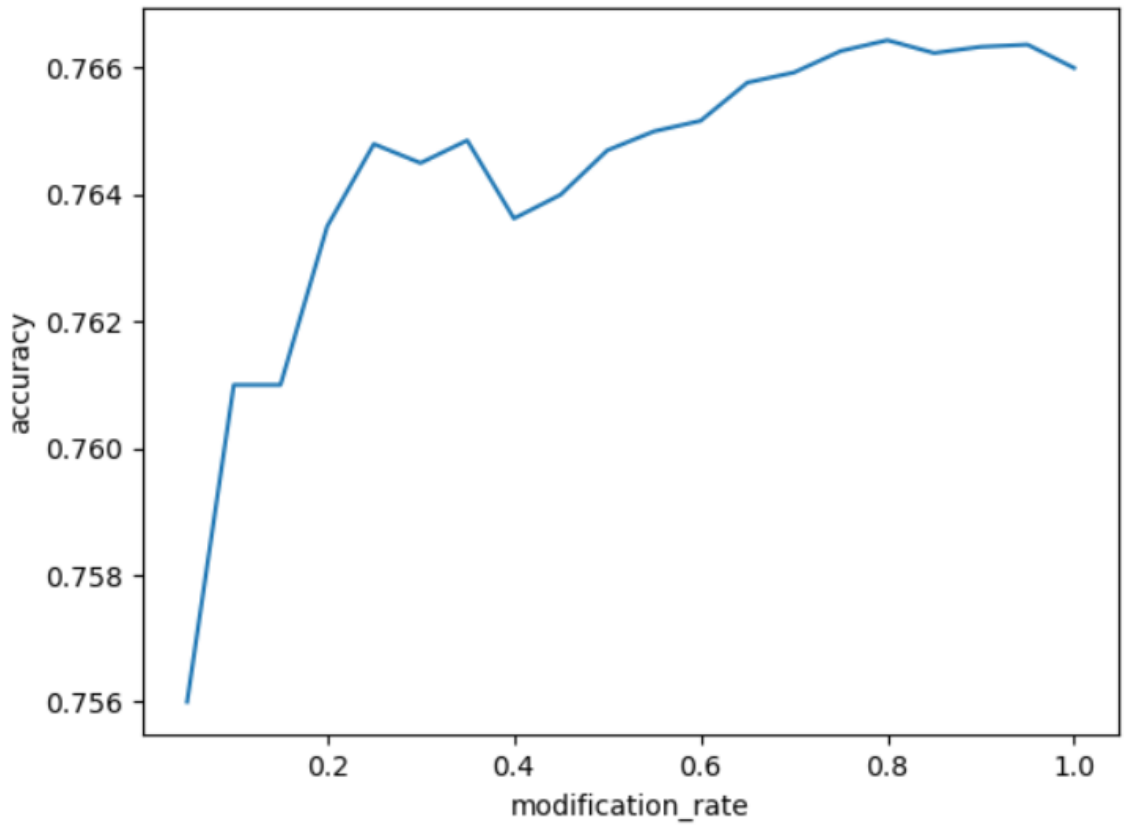


Рисунок 4.10 – Графік залежності точності від *modification_rate*, де $modification\_rate \in [0, 1]$.

Як можна побачити з графіка на рисунку 4.10, найбільше значення точності досягається при *modification_rate* в інтервалі [0.7, 1]. Для знаходження точного оптимального значення *modification_rate*, було зроблено ще одне тестування зі значеннями в інтерівалі [0.7, 1] і з кроком 0.02, результати якого можна побачити на рисунку 4.12, з якого видно, що найбільша точність досягається при *modification_rate=0.85*. При тому найкраща сталість алгоритму досягаєтсья при *modification_rate=0.4* (графік на рисунку 4.11). Отже для точності обираємо значення 0.85, для сталості 0.4.

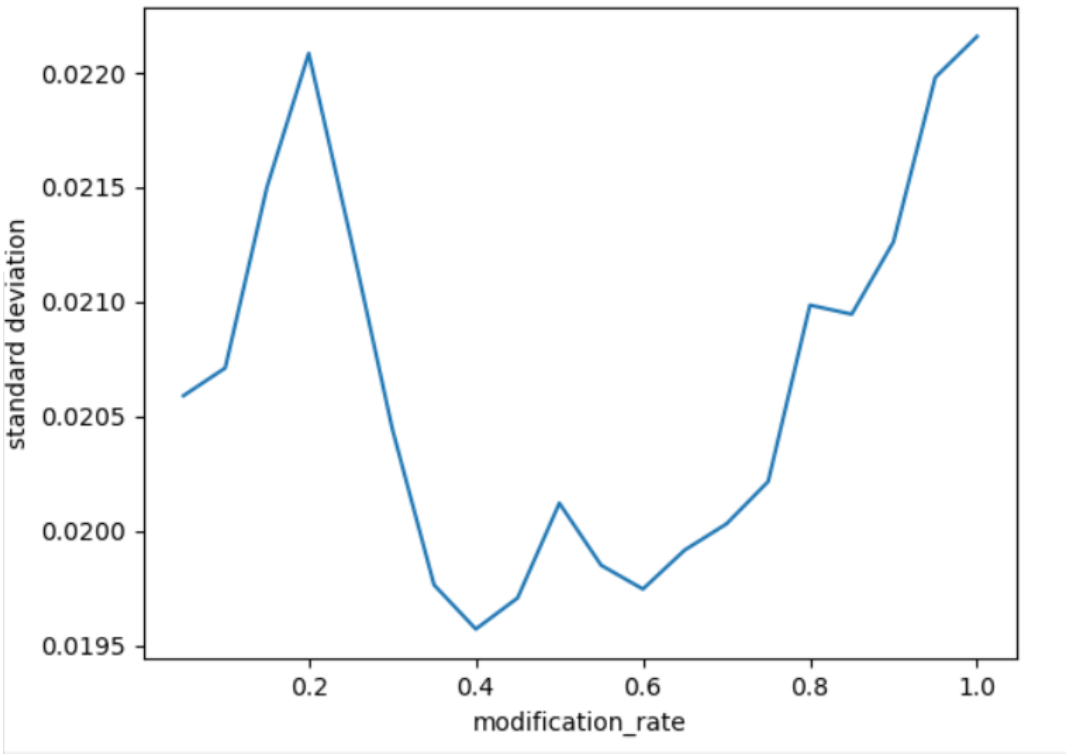


Рисунок 4.11 – Графік залежності відхилення від *modification_rate*.

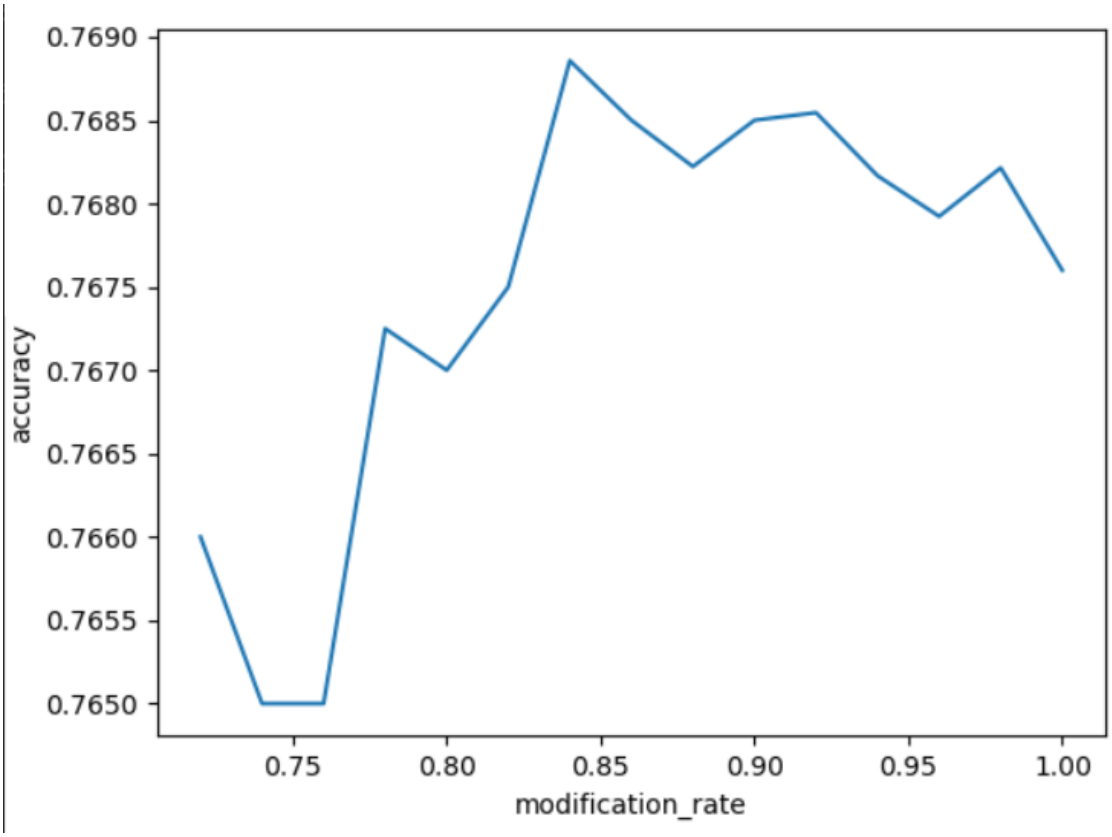


Рисунок 4.12 – Графік залежності точності від *modification_rate*, де $modification\_rate \in [0.7, 1]$.

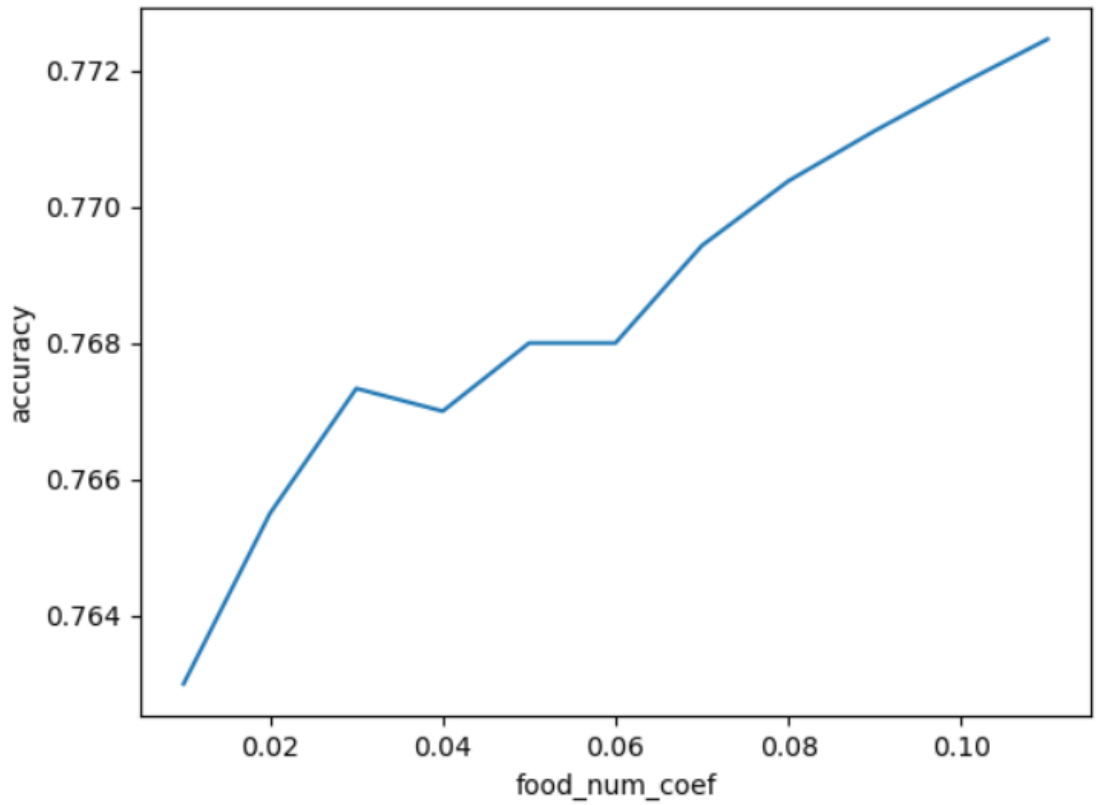


Рисунок 4.13 – Графік залежності точності від *food_num_coef*.

4) Для того щоб проаналізувати параметр *food_sources_num*, було визначено коефіцієнт *food_num_coef*, який показує відношення *food_sources_num* / *train_num*, де *train_num* – кількість тренувальних даних. Виконано тестування з значеннями *food_num_coef* в інтервалі [0, 0.1] з кроком 0.01, інші параметри залишались незмінними і були встановлені наступні значення *cycle=2*, *max_limit=3* і *modification_rate =0.85*.

Як можна побачити з графіка на рисунку 4.13 залежність *food_num_coef* і точності алгоритму носить прямо пропорційний характер, тобто чим більше джерел їжі, тим більша точність, проте збільшується і кількість обчислень алгоритму. Згідно цього, значення *food_sources_num* треба обирати опираючись на обчилювальні потужності комп’ютера, на якому виконується тестування. Такий висновок можна зробити і для параметру *max_limit*.

Середнє відхилення для цього тестування було оптимальним при значенні *food_num_coef = 0.09*.

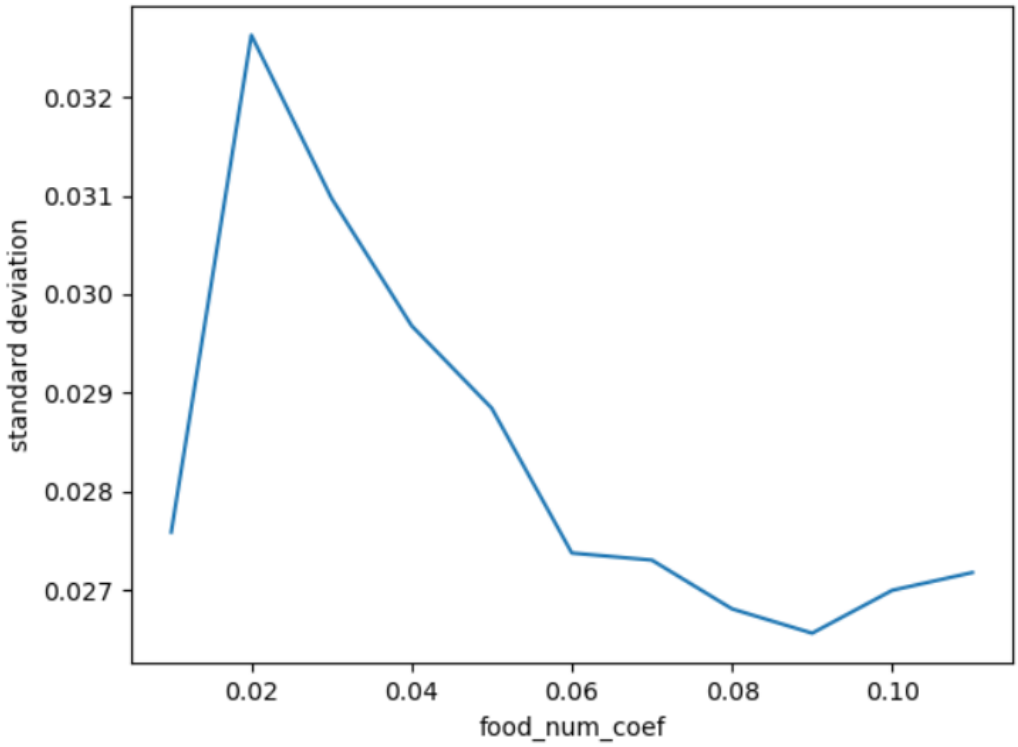


Рисунок 4.14 – Графік залежності відхилення від *food_num_coef*.

Отже, проаналізувавши залежність точності і сталості запропонованого алгоритму від параметрів *cycle*, *max_limit*, *modification_rate* і *food_sources_num*, були визначені характеристики цих залежностей, і згідно цих характеристик маємо такі висновки для параметрів:

- збільшення *cycle* дає суттєве збільшення точності в діапазоні [1, 3], після чого точність практично не міняється;
- *max_limit* прямо пропорційно впливає на точність, проте збільшення цього параметру сильно впливає на час обчислень алгоритму;
- для *modification_rate* були знайденні оптимальні значення для цього тестовогу набору;
- точність алгоритму прямо пропорційно залежить від *food_sources_num*, проте так само від нього залежить і час обчислень, *max(food_sources_num)=train_num*.

4.2. Порівняльний аналіз

Для порівняння були взяті наступні алгоритми з бібліотеки scikit-learn: k-nearest neighbor (метод k-найближчих сусідів, KNN), decision tree (дерево рішень, DT), SVM (метод опорних векторів) і random forest (випадковий ліс, RF).

Спочатку запустимо ці алгоритми на наборі даних german використовуючи k-fold перехресне тестування. Для алгоритму ABC були обрані наступні параметри:

- *cycle=2*
- *max_limit=4*
- *modification_rate=0.85*
- *food_num_coef=0.1*

	DT	KNN	RF	SVM	ABC
Fold-1	0.680000	0.70000	0.810000	0.760000	0.820000
Fold-2	0.630000	0.63000	0.710000	0.720000	0.730000
Fold-3	0.700000	0.75000	0.780000	0.450000	0.780000
Fold-4	0.650000	0.70000	0.780000	0.710000	0.770000
Fold-5	0.620000	0.72000	0.770000	0.740000	0.760000
Fold-6	0.690000	0.73000	0.770000	0.760000	0.790000
Fold-7	0.720000	0.63000	0.740000	0.410000	0.770000
Fold-8	0.710000	0.71000	0.800000	0.580000	0.820000
Fold-9	0.690000	0.65000	0.720000	0.750000	0.790000
Fold-10	0.750000	0.70000	0.800000	0.710000	0.780000
Std	0.040607	0.04158	0.034254	0.131694	0.026854
Average	0.684000	0.69200	0.768000	0.659000	0.781000

Рисунок 4.15 – Порівняльна таблиця для k-fold тестування набору даних german.

Як можна побачити з рисунку 4.15 алгоритм ABC має кращу точність класифікації майже для всіх наборів даних, і найкращу середню точність. На рисунку 4.16 можна побачити результати k-fold тестування алгоритмів для другого набору даних, де можна побачити які саме показники впливають на результати класифікації відповідних алгоритмів.

```
DT Fold-2
              precision    recall  f1-score   support

           1       0.75      0.70      0.73        70
           2       0.40      0.47      0.43        30

    accuracy                           0.63       100
   macro avg       0.58      0.58      0.58       100
weighted avg       0.65      0.63      0.64       100

KNN Fold-2
              precision    recall  f1-score   support

           1       0.71      0.80      0.75        70
           2       0.33      0.23      0.27        30

    accuracy                           0.63       100
   macro avg       0.52      0.52      0.51       100
weighted avg       0.60      0.63      0.61       100

RF Fold-2
              precision    recall  f1-score   support

           1       0.75      0.89      0.81        70
           2       0.53      0.30      0.38        30

    accuracy                           0.71       100
   macro avg       0.64      0.59      0.60       100
weighted avg       0.68      0.71      0.68       100

SVM Fold-2
              precision    recall  f1-score   support

           1       0.73      0.97      0.83        70
           2       0.71      0.17      0.27        30

    accuracy                           0.73       100
   macro avg       0.72      0.57      0.55       100
weighted avg       0.73      0.73      0.67       100

ABC Fold-2
              precision    recall  f1-score   support

           1       0.83      0.79      0.81        70
           2       0.56      0.63      0.59        30

    accuracy                           0.74       100
   macro avg       0.70      0.71      0.70       100
weighted avg       0.75      0.74      0.74       100
```

Рисунок 4.16 – Розширений вивід результатів для набору даних german при *K-Fold=2*.

Використаємо набір даних iris для порівняння, який складається з 150 зразків і 3 класів, де кожному класу відповідають по 50 рядків. Параметри для алгоритму ABC оберемо аналогічні попередньому тесту.

	DT	KNN	RF	SVM	ABC
Fold-1	1.000000	1.000000	1.000000	1.000000	1.000000
Fold-2	0.933333	0.933333	0.933333	1.000000	1.000000
Fold-3	1.000000	1.000000	1.000000	1.000000	1.000000
Fold-4	0.933333	1.000000	0.933333	1.000000	1.000000
Fold-5	0.933333	0.866667	0.933333	0.933333	0.933333
Fold-6	0.866667	0.933333	0.933333	0.933333	0.866667
Fold-7	0.933333	0.933333	0.866667	0.800000	0.933333
Fold-8	0.933333	1.000000	1.000000	1.000000	1.000000
Fold-9	1.000000	1.000000	1.000000	1.000000	1.000000
Fold-10	1.000000	1.000000	1.000000	1.000000	1.000000
Std	0.044997	0.047140	0.046614	0.064788	0.046614
Average	0.953333	0.966667	0.960000	0.966667	0.973333

Рисунок 4.17 – Порівняльна таблиця тестування набору даних iris.

Як можна побачити з рисунку 4.17 алгоритм ABC має кращу точність класифікації майже для всіх наборів даних, і найкращу середню точність також і для набору даних iris.

Використаємо набір даних digits для порівняння, який складається з 1797 зразків і 10 класів, де кожному класу відповідають приблизно по 180 рядків. Параметри для алгоритму ABC оберемо наступні (збільшено *cycle*, *max_limit* і *food_num_coef*):

- *cycle=10*
- *max_limit=6*
- *modification_rate=0.85*
- *food_num_coef=0.5*

	DT	KNN	RF	SVM	ABC
Fold-1	0.811111	0.927778	0.911111	0.894444	0.952242
Fold-2	0.861111	0.983333	0.983333	0.950000	0.974004
Fold-3	0.838889	0.977778	0.938889	0.872222	0.975503
Fold-4	0.816667	0.955556	0.950000	0.861111	0.961750
Fold-5	0.805556	0.972222	0.961111	0.927778	0.964857
Fold-6	0.883333	0.972222	0.966667	0.950000	0.976996
Fold-7	0.900000	0.988889	0.977778	0.972222	0.976996
Fold-8	0.815642	0.983240	0.960894	0.949721	0.972337
Fold-9	0.826816	0.983240	0.932961	0.854749	0.972337
Fold-10	0.821229	0.966480	0.944134	0.938547	0.956774
Std	0.032610	0.018098	0.021835	0.042676	0.008905
Average	0.838035	0.971074	0.952688	0.917079	0.968380

Рисунок 4.18 – Порівняльна таблиця тестування набору даних digits.

Для набору даних digits алгоритм ABC не завжди дає найкращу точність (рисунок 4.18) і має середню точність нижчу за алгоритм KNN, оскільки digits має багато класів рівномірно розподілених по набору і саме стратегія найближчого сусіда є кращою для такого випадку. Незважаючи на це, за показником середнього відхилення саме алгоритм ABC показує найкращі результати.

Використаємо набір даних wine для порівняння, який складається з 178 зразків і 3 класів, де кожному класу відповідають по [59,71,48] рядків. Параметри для алгоритму ABC оберемо аналогічні попередньому тесту.

	DT	KNN	RF	SVM	ABC
Fold-1	0.888889	0.666667	0.944444	0.777778	1.0
Fold-2	0.888889	0.666667	1.000000	0.722222	1.0
Fold-3	0.722222	0.611111	0.944444	0.666667	1.0
Fold-4	0.888889	0.611111	0.944444	0.722222	1.0
Fold-5	0.833333	0.611111	1.000000	0.777778	1.0
Fold-6	0.888889	0.611111	1.000000	0.722222	1.0
Fold-7	1.000000	0.722222	1.000000	0.722222	1.0
Fold-8	0.888889	0.666667	1.000000	0.722222	1.0
Fold-9	0.941176	0.823529	1.000000	0.941176	1.0
Fold-10	0.764706	0.764706	1.000000	0.882353	1.0
Std	0.080288	0.073689	0.026836	0.084295	0.0
Average	0.870588	0.675490	0.983333	0.765686	1.0

Рисунок 4.19 – Порівняльна таблиця тестування набору даних wine.

	DT	KNN	RF	SVM	ABC
Fold-1	0.912281	0.912281	0.982456	0.877193	0.991189
Fold-2	0.859649	0.877193	0.894737	0.859649	0.964274
Fold-3	0.929825	0.894737	0.947368	0.929825	0.973329
Fold-4	0.859649	0.964912	0.964912	0.929825	0.973329
Fold-5	0.964912	0.947368	0.982456	0.947368	1.000000
Fold-6	0.894737	0.929825	0.982456	0.929825	0.991189
Fold-7	0.877193	0.964912	0.964912	0.947368	0.991189
Fold-8	0.947368	0.929825	0.982456	0.929825	1.000000
Fold-9	0.929825	0.912281	0.964912	0.912281	0.991189
Fold-10	0.928571	0.964286	1.000000	0.946429	1.000000
Std	0.036402	0.030866	0.029181	0.030014	0.012755
Average	0.910401	0.929762	0.966667	0.920959	0.987569

Рисунок 4.20 – Порівняльна таблиця тестування набору даних breast cancer.

Отже, для набору даних wine запропонований алгоритм завжди правильно визначає усі класи на відміну від інших досліджуємих алгоритмів, що можна побачити на рисунку 4.19.

Використаємо набір даних breast cancer для порівняння (рисунок 4.20), який складається з 569 зразків і 2 класів, де кожному класу відповідають по [212, 357] рядків відповідно. Параметри для алгоритму ABC оберемо аналогічні попередньому тесту.

Як видно з рисунку 4.20 ABC показує найкращі результати за показником точності і сталості для усіх тестових випадків у наборі даних breast cancer.

Висновок за четвертим розділом

В даному розділі були розроблені параметричні та порівняльні тестування і продемонстровані їх результати за допомогою таблиць та графіків. За результатом цих тестувань були визначені оптимальні параметри для розробленого алгоритму, після чого проведений порівняльний аналіз алгоритму ABC з іншими поширеними класифікаторами.

За результатами порівняльного аналізу, запропонований алгоритм ABC має найкращі результати за показником сталості у всіх тестових випадках, які були представлені у даному розділі, а за показником точності у всіх, окрім набору даних digits, де алгоритм KNN є кращим..

ВИСНОВКИ

В даній магістерській дисертації були проаналізовані алгоритми інтелектуального аналізу даних, які вирішують задачу класифікації. Оскільки метаевристичні алгоритми на основі ройового інтелекту з кожним днем мають все ширше застосування у сфері Data Mining, було обрано застосувати алгоритм бджолиної колонії для задачі класифікації.

На основі класичного алгоритму бджолиної колонії, який раніше застосовувався тільки для задачі глобального пошуку, було розроблено алгоритм ABC для задачі класифікації, основним елементом якого є «обмінена» стратегія пошуку правила.

Також було розроблено програмний додаток, який реалізує запропонований алгоритм. Програма була написана на мові програмування python, яка є найбільш пристосованою для задач, зв’язаних з Data Mining і Machine Learning, за рахунок свого простого синтаксису та поширеною системою бібліотек, таких як pandas, numpy, scipy і scikit-learn.

Розроблена програма дає змогу тестувати алгоритм безпосередньо задаючи параметр розміру тестового та тренувального наборів, та за допомогою методики k-fold. На екран можна виводити як і розширену інформацію з тесту, так і лише точність.

Були проаналізовані всі параметри алгоритму, та визначена залежність точності та сталості від кожного з них. Згідно цих результатів були обрані параметри для порівняльних тестів, де запропонований алгоритм було порівняно з такими популярними класифікаторами, як дерево рішень, k-найближчих сусідів, метод SVM та випадковий ліс (Random forest). За результатами порівняльного аналізу було доведено ефективність застосування запропонованого алгоритму ABC для задачі класифікації, оскільки він показав кращі результати в сенсі точності і сталості для більшості тестових наборів даних.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Alpaydin, Ethem (2010). Introduction to Machine Learning. ISBN 978-0-262-01243-0. MIT Press. p. 9.
2. Karaboga, D. (2005). An Idea Based on Honey Bee Swarm for Numerical Optimization (TECHNICAL REPORT-TR06).
3. Cover, Thomas M.; Hart, Peter E. (1967). "Nearest neighbor pattern classification". IEEE Transactions on Information Theory. 13 (1): 21–27.
4. Venables, W. N.; Ripley, B. D. (2002). Modern Applied Statistics with S (4th ed.). Springer Verlag.
5. Narasimha Murty, M.; Susheela Devi, V. (2011). Pattern Recognition: An Algorithmic Approach.
6. Tolles, Juliana; Meurer, William J (2016). "Logistic Regression Relating Patient Characteristics to Outcomes". JAMA. 316 (5): 533–4.
7. Rodríguez, G. (2007). Lecture Notes on Generalized Linear Models. pp. Chapter 3, page 45, URL: http://data.princeton.edu/wws509/notes/ (дата звернення 01.11.2020).
8. Freund, Y.; Schapire, R. E. (1999). "Large margin classification using the perceptron algorithm". Machine Learning. 37 (3): 277–296.
9. Quinlan, J. R. 1986. Induction of Decision Trees. Mach. Learn. 1, 1 (Mar. 1986), 81–106.
10. Nello Cristianini, John Shawe-Taylor. An Introduction to Support Vector Machines and Other Kernel-based Learning Methods. – Cambridge University Press, 2000.
11. Bianchi, Leonora; Marco Dorigo; Luca Maria Gambardella; Walter J. Gutjahr (2009). "A survey on metaheuristics for stochastic combinatorial optimization". Natural Computing. 8 (2): 239–287.
12. Sousa, T., Silva, A., & Neves, A. (2004). Particle Swarm based Data Mining Algorithms for classification tasks. Parallel Computing, 30(5-6), 767-783.

13. R. S. Parpinelli, H. S. Lopes, and A. A. Freitas, “Data mining with an ant colony optimization algorithm,” IEEE Trans. Evol. Comput., vol. 6, no. 4, pp. 321–332, Aug. 2002.
14. M. Dorigo, V. Maniezzo, and A. Colorni, Positive feedback as a search strategy Dipartimento di Elettronica e Informatica, Politecnico di Milano, Milano, Italy, Tech. Rep. 91016, 1991.
15. Kaski, Samuel (1997). Data Exploration Using Self-Organizing Maps. Acta Polytechnica Scandinavica. Mathematics, Computing and Management in Engineering Series No. 82. Espoo, Finland: Finnish Academy of Technology.
16. McLachlan, G. J., Kim-Anh, D., & Christophe, A. (2004). Analyzing Microarray gene expression data.
17. Документація мови python : URL: https://www.python.org/doc/ (дата звернення: 01.11.2020).
18. Документація бібліотеки scikit-learn : URL: https://scikit-learn.org/stable/ (дата звернення: 01.11.2020).

ДОДАТКИ

Додаток 1. Покроковий вивід розробленої програми

Init ArtificialBeeColony
Created 23 employed_bees
Evaluates nectar start
Init OnlookerBee
Evaluates nectar end
1 bee: food source - [1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0];
1 bee: fitness - 0.6833; importance - 0.0000
2 bee: food source - [0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1];
2 bee: fitness - 0.7167; importance - 0.0901
3 bee: food source - [1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0];
3 bee: fitness - 0.7033; importance - 0.0000
4 bee: food source - [1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1];
4 bee: fitness - 0.6867; importance - 0.0000
5 bee: food source - [1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0];
5 bee: fitness - 0.7133; importance - 0.0186
6 bee: food source - [0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0];
6 bee: fitness - 0.7133; importance - 0.0186
7 bee: food source - [0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1];
7 bee: fitness - 0.7133; importance - 0.0186
8 bee: food source - [1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1];
8 bee: fitness - 0.7167; importance - 0.0901
9 bee: food source - [1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1];
9 bee: fitness - 0.7133; importance - 0.0186
10 bee: food source - [0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0];
10 bee: fitness - 0.7167; importance - 0.0901
11 bee: food source - [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
11 bee: fitness - 0.7200; importance - 0.1615
12 bee: food source - [1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1];
12 bee: fitness - 0.7200; importance - 0.1615
13 bee: food source - [1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1];
13 bee: fitness - 0.7133; importance - 0.0186
14 bee: food source - [1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1];
14 bee: fitness - 0.7133; importance - 0.0186
15 bee: food source - [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1];
15 bee: fitness - 0.7300; importance - 0.3758
16 bee: food source - [1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0];

16 bee: fitness - 0.7133; importance - 0.0186
17 bee: food source - [1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1];
17 bee: fitness - 0.7133; importance - 0.0186
18 bee: food source - [1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1];
18 bee: fitness - 0.7133; importance - 0.0186
19 bee: food source - [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1];
19 bee: fitness - 0.7133; importance - 0.0186
20 bee: food source - [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1];
20 bee: fitness - 0.7200; importance - 0.1615
21 bee: food source - [0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0];
21 bee: fitness - 0.7133; importance - 0.0186
22 bee: food source - [0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0];
22 bee: fitness - 0.7133; importance - 0.0186
23 bee: food source - [0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1];
23 bee: fitness - 0.7133; importance - 0.0186
Best fitness 0.73
Best food source [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Evaluates nectar start
Init OnlookerBee
Evaluates nectar end
1 bee: food source - [1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0];
1 bee: fitness - 0.6833; importance - 0.0000
2 bee: food source - [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0];
2 bee: fitness - 0.7300; importance - 0.2762
3 bee: food source - [1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0];
3 bee: fitness - 0.7033; importance - 0.0000
4 bee: food source - [1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1];
4 bee: fitness - 0.6867; importance - 0.0000
5 bee: food source - [1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0];
5 bee: fitness - 0.7133; importance - 0.0000
6 bee: food source - [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0];
6 bee: fitness - 0.7133; importance - 0.0000
7 bee: food source - [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0];
7 bee: fitness - 0.7167; importance - 0.0409
8 bee: food source - [0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1];
8 bee: fitness - 0.7167; importance - 0.0409
9 bee: food source - [0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0];
9 bee: fitness - 0.7133; importance - 0.0000
10 bee: food source - [0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0];

10 bee: fitness - 0.7167; importance - 0.0409
11 bee: food source - [1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1];
11 bee: fitness - 0.7200; importance - 0.0997
12 bee: food source - [1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1];
12 bee: fitness - 0.7200; importance - 0.0997
13 bee: food source - [0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0];
13 bee: fitness - 0.7133; importance - 0.0000
14 bee: food source - [1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0];
14 bee: fitness - 0.7133; importance - 0.0000
15 bee: food source - [1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0];
15 bee: fitness - 0.7300; importance - 0.2762
16 bee: food source - [0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0];
16 bee: fitness - 0.7133; importance - 0.0000
17 bee: food source - [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1];
17 bee: fitness - 0.7133; importance - 0.0000
18 bee: food source - [0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0];
18 bee: fitness - 0.7133; importance - 0.0000
19 bee: food source - [1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0];
19 bee: fitness - 0.7133; importance - 0.0000
20 bee: food source - [0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1];
20 bee: fitness - 0.7200; importance - 0.0997
21 bee: food source - [1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0];
21 bee: fitness - 0.7133; importance - 0.0000
22 bee: food source - [0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0];
22 bee: fitness - 0.7133; importance - 0.0000
23 bee: food source - [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0];
23 bee: fitness - 0.7400; importance - 0.4527
Best fitness 0.74
Best food source [1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0]

	precision	recall	f1-score	support
1	0.81	0.83	0.82	214
2	0.55	0.52	0.54	86
accuracy			0.74	300
macro avg	0.68	0.68	0.68	300
weighted avg	0.74	0.74	0.74	300

Додаток 2. Код розробленої програми

Class ArtificialBeeColony

```
from algorithms.employee_bee import EmployedBee
from algorithms.onlooker_bee import OnlookerBee
import numpy as np
import random

class ArtificialBeeColony:

    def __init__(self, clf, features, X_train, X_test, y_train, y_test, modification_rate, food_sources_num):
        self.food_sources = np.array([])
        self.features = features
        self.data = X_train
        self.test_data = X_test
        self.labels = y_train
        self.test_labels = y_test
        self.fitness = 0.0
        self.fitnesses = np.array([])
        self.modification_rate = modification_rate
        self.selected_features = self.features
        self.features_bounds = np.array([[self.test_data.iloc[:, i].min(axis=0), self.test_data.iloc[:, i].max(axis=0)] for i in range(len(self.features))])
        self.food_sources_num = food_sources_num
        self.clf = clf
        print("Init ArtificialBeeColony")

    def initialize_food_source(self, food_source_size, food_source_num):
        self.food_sources = np.empty((0, food_source_size), np.int)
        for i in range(food_source_num):
            self.food_sources = np.append(self.food_sources, np.array([[1 if j == i % food_source_size else 0 for j in range(food_source_size)]]), axis=0)

    def execute(self, cycle, target, max_limit):
        self.initialize_food_source(len(self.features), self.food_sources_num)

        best_food_source = np.array([random.choice((0, 1)) for _ in range(len(self.features))])

        employed_bees = np.array([])
        food_source_counter = 0
        for food_source in self.food_sources:
            employed_bees = np.append(employed_bees, EmployedBee(self.clf, self.features, self.data, self.labels, self.test_data, self.test_labels, food_source, self.modification_rate, max_limit))
```

```
            food_source_counter+=1

        print(f'Created {len(employed_bees)} employed_bees')

        for _ in range(cycle):

            print('Evaluates nectar start')
            onlooker_bees = OnlookerBee(employed_bees)
            print('Evaluates nectar end')
            onlooker_bees.evaluates_nectar()

            self.fitness = onlooker_bees.get_best_fitness()
            print(f'Best fitness {self.fitness}')
            self.fitnesses = np.append(self.fitnesses, self.fitness)
            best_food_source = onlooker_bees.get_best_food_source()
            print(f'Best food source {best_food_source}')
            self.selected_features = [f for i, f in enumerate(self.features) if best_food_source[i] == 1]

            # print(best_food_source)

            # if self.fitness >= target:
            #     return self.fitness, self.selected_features, onlooker_bees.get_best_employed_bee(),
self.fitnesses

        return np.max(self.fitnesses), self.selected_features,
onlooker_bees.get_best_employed_bee(), self.fitnesses
```

class EmployedBee

```
from sklearn.metrics import accuracy_score
import numpy as np
import random

class EmployedBee:

    def __init__(self, clf, features, dataset, labels, test_data, test_labels, starter_food_source,
modification_rate, MAX_LIMIT):
        self.current_limit = 0
        self.MAX_LIMIT = MAX_LIMIT
        self.dataset = dataset
        self.labels = labels
        self.test_data = test_data
        self.test_labels = test_labels
        self.features = features
        self.current_food_source = starter_food_source
```

```
        self.modification_rate = modification_rate
        self.clf = clf

        selected_features = [f for i, f in enumerate(
            self.features) if self.current_food_source[i] == 1]

        self.clf.fit(self.dataset[selected_features], self.labels)
        self.y_pred = self.clf.predict(self.test_data[selected_features])
        # self.y_pred = get_mknn_predicted(self.dataset[selected_features].to_numpy(), self.labels.to_numpy(
        # ), self.test_data[selected_features].to_numpy(), self.test_labels.to_numpy(), k)
        self.current_fitness = accuracy_score(
            self.test_labels.to_numpy(), self.y_pred)

    def calculate_fitness(self):
        neighbor = [(int(not bit) if random.uniform(0, 1) < self.modification_rate else bit) for bit in self.current_food_source]
        if (sum(neighbor) == 0):
            rand_index = random.randrange(0, len(neighbor))
            neighbor[rand_index] = 1

        selected_features = [f for i, f in enumerate(
            self.features) if neighbor[i] == 1]

        self.clf.fit(self.dataset[selected_features], self.labels)
        self.y_pred = self.clf.predict(self.test_data[selected_features])
        fitness = accuracy_score(self.test_labels, self.y_pred)

        if fitness > self.current_fitness:
            self.current_limit = 0
            self.current_food_source = neighbor
            self.current_fitness = fitness
        else:
            self.current_limit += 1
            if self.current_limit != self.MAX_LIMIT:
                self.calculate_fitness()
            else:
                self.current_limit = 0
                self.current_food_source = [random.choice((0, 1)) for _ in range(len(self.current_food_source))]

    def get_y_pred(self):
        return self.y_pred

    def get_current_fitness(self):
```

```
        return self.current_fitness

    def get_current_food_source(self):
        return self.current_food_source

    def generate_new_food_source(self):
        self.current_food_source = np.array(
            [random.choice((0, 1)) for _ in range(len(self.current_food_source))])
```

class OnlookerBee

```
import numpy as np
import random

class OnlookerBee:

    def __init__(self, employed_bees):
        self.employed_bees = employed_bees
        self.bees_distribution = np.array([])
        self.food_source_size = len(self.employed_bees[0].get_current_food_source())
        self.best_fitness = 0
        self.best_food_source = np.array([])
        self.best_employed_bee = None
        print("Init OnlookerBee")

    def get_best_food_source(self):
        return self.best_food_source

    def get_best_employed_bee(self):
        return self.best_employed_bee

    def get_best_fitness(self):
        return self.best_fitness

    def evaluates_nectar(self):
        for i in range(len(self.employed_bees)):
            self.employed_bees[i].calculate_fitness()

        self.roulette_wheel()

    def roulette_wheel(self):
        self.bees_distribution = np.array([])
        num_of_bees = len(self.employed_bees)
        total_fitness = 0
        count = 0
```

```
        total_fitness = np.array([])

        for bee in self.employed_bees:
            fitness = bee.get_current_fitness()
            if fitness > self.best_fitness:
                self.best_food_source = bee.get_current_food_source()
                self.best_fitness = fitness
                self.best_employed_bee = bee
            total_fitness = np.append(total_fitness, fitness)

        avg_fitness = np.mean(total_fitness)
        v_fitness = np.max(total_fitness) - np.min(total_fitness)
        var_fitness = np.std(total_fitness)

        for bee in self.employed_bees:
            count += 1
            freq_d = (bee.get_current_fitness() - avg_fitness) / v_fitness
            if freq_d < var_fitness:
                freq_d = 0
            freq = int(freq_d * len(self.employed_bees))
            print(f'{count} bee: food source - {bee.get_current_food_source()};')
            print(f'{count} bee: fitness - {"{:.4f}".format(bee.get_current_fitness())}; importance -
{"{:.4f}".format(freq_d)}')
            for i in range(freq):
                self.bees_distribution = np.append(self.bees_distribution, bee)

        num_of_dist = len(self.bees_distribution)

        for i in range(len(self.employed_bees)):
            freq_d = (bee.get_current_fitness() - avg_fitness) / v_fitness
            if freq_d < var_fitness:
                self.employed_bees[i] = self.bees_distribution[random.randrange(
                    0, num_of_dist)]
```

Додаток 3. Копії тез доповіді на ПМК-2019

УДК 004.021

К.т.н., доцент Зорін Ю.М., студент Абдураімов Т.З.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

МОДИФІКОВАНИЙ МУРАШИНИЙ АЛГОРИТМ ДЛЯ РОЗВ'ЯЗАННЯ ЗАДАЧІ ГЛОБАЛЬНОЇ ОПТИМІЗАЦІЇ

Abstract

Yuri Zorin, Assoc. Prof., PhD; Tair Abduraimov, student
A modified Ant Colony Optimization Algorithm for global optimization.

Paper presents modification of the ant colony optimization algorithm which is robust to variables domain. The proposed algorithm applies self-adaptive approaches in terms of domain adjustment, pheromone increment, domain division, and ant colony size without any major conceptual changes to ACO's framework. When using this algorithm, there is no need to estimate proper initial domains for practical continuous optimization problems in expert and intelligent systems.

Вступ

Мурашиний алгоритм (Ant Colony Optimization, ACO) представляє собою метаевристичний алгоритм, концепція якого полягає в моделюванні поведінки колонії мурах в природі. Даний алгоритм демонструє якісні показники при розв'язанні задач комбінаторної оптимізації [1], навчанні нейронних мереж, оптимізації систем зв'язку [2].

Перший алгоритм на основі ACO, який був розроблений К. Соча та М. Доріго в 2008 р. для розв'язання задачі оптимізації неперервних функцій (ACO_R [3]). Принцип ACO_R полягає в використанні зважувальної гауссової функції (probability density function, PDF) для генерації розв'язків, що зберігаються в архіві, й оновлення феромонів шляхом заміни найгірших розв'язків новими.

В роботі запропоновано модифікацію мурашиного алгоритму, позначену ACO*, яка дозволяє знаходити близький до оптимального розв'язок навіть при некоректній оцінці початкової області пошуку. Якщо ж простір для пошуку оцінений вірно, то даний алгоритм показує кращу швидкість збіжності до оптимуму ніж ACO_R.

Постановка задачі

Метою роботи є розробка модифікованого мурашиного алгоритму для розв'язання задачі глобальної оптимізації, який перевищує оригінальний в сенсі в точності результатів і сталості, а також є стійким до області пошуку.

Основні принципи модифікованого алгоритму

Запропонована модифікація має багато спільного з оригінальним ACO_R. По-перше, інформація про феромони й розв'язки зберігаються в архіві, по-друге, використовується PDF, по-третє, кількість феромонів збільшується для найкращих розв'язків

Основними відмінностями запропонованого алгоритму є метод сітки [4] для простору пошуку, а також само-адаптивні механізми для формування нових розв'язків.

Метод сітки полягає в розділенні початкового простору пошуку на рівні проміжки

$$h_i = \frac{x^i_{\max} - x^i_{\min}}{k}, (i = 1,2,...,n),$$

де k – кількість таких проміжків. Ймовірність вибору i-того розв'язку

$$p_{ij} = \tau_{ij} \Big/ \sum_{i=1}^{k+1} \tau_{ij},$$

де τ_{ij} – величина випару феромонів *j*-тої змінної *i*-того розв'язку. Кожен мураха додає деякий приріст феромонів $\Delta\tau_{ij}$ на кожному розв'язку

$$\Delta\tau_{ij} = Q / f ,$$

де f – значення цільової функції, Q – величина, що регулює рівень феромонів. Рівень феромонів τ_{ij} змінюється за наступною формулою

$$\tau_{ij}^{new} = (1-\rho)\cdot\tau_{ij}^{old} + \Delta\tau_{ij}$$

В даній модифікації пропонується система адаптивного регулювання області пошуку. Введемо параметр θ як поріг для вирішення питання про те, чи запускати механізм зміни області пошуку. Позначимо r_i *(i=1,2,…,n)* – номер рядка найбільшого елемента в кожному стовпчику архіва розв'язків.

Якщо $r_i \le \theta(k+1)$ або $r_i \ge (1-\theta)(k+1)$, то зрозуміло, що точка розв'язку з найбільшою кількістю феромонів знаходиться поблизу границі області пошуку, і це означає, що оптимум знаходиться скоріш за все за межами області, отже мурахам треба надати нову область пошуку з центром в r_i

$$x_{\min}^{i} = r_i - (k/2 + \Delta_1)\cdot h_i$$
$$x_{\max}^{i} = r_i + (k/2 + \Delta_1)\cdot h_i ,$$

де Δ_1 – параметр, який регулює збільшення області пошуку.

Якщо $\theta(k+1) \le r_i \le (1-\theta)(k+1)$, то точка з найбільшою кількістю феромонів знаходиться далеко від границі, тобто оптимум скоріш за все всередині заданої області пошуку, отже мурахам треба звузити область для зменшення похибки пошуку. Границі обновлюються наступним чином

$$x_{\min}^{i} = x_{\min}^{i} + (x_{\max}^{i} - x_{\min}^{i})\cdot\Delta_2$$
$$x_{\max}^{i} = x_{\max}^{i} + (x_{\max}^{i} - x_{\min}^{i})\cdot\Delta_2 ,$$

де Δ_2 – параметр, який регулює зміну розміру області пошуку.

Необхідність саморегулювання кількості феромонів пояснюється тим, що під час роботи алгоритму порядок величини Q може ставати невідповідним значенням функції, що призводить до незначних змін приросту феромонів. Отже, адаптивність кількості феромонів досягається за допомогою зміни значення параметру Q

$$Q = 10^{OM_{\min}+1},$$

де OM_{min} – порядок величини мінімального значення цільової функції на даній ітерації. Застосування даного принципу призводить до підвищення точності результатів алгоритму.

Також важливим є адаптивність розділу області пошуку й кількості розв'язків (мурах), оскільки різна кількість рівних частин, на які поділений простір пошуку, може давати зовсім різні результати. Як було визначено, k – це кількість таких рівних проміжків. Спочатку присвоюємо k невелике значення для прискорення швидкості збіжності. Якщо мурахи не можуть знайти кращого результату після кількох ітерацій, збільшуємо k на одиницю. Таким чином, структура простору пошуку динамічно змінюється, що дозволяє мурахам шукати інші значення й уникати застрягання в локальному екстремумі. Отже, коли збільшується k, кількість мурах m повинне відповідати цьому, і це досягається наступним перетворенням $m = m + \Delta m$, де Δm – мале додатне число, завдяки чому m є трохи більшим за k, що запобігає збільшенню кількості мурах до занадто великих розмірів.

Опис алгоритму

Псевдокод запропонованого алгоритму.

```
Initialize parameters
Do
Divide the domain into k equivalent shares
      Generate random routes, give τ heuristic information
```

```
Update ant size m and pheromone amount Q
Do
        Each ant finishes its route
        Find the best route
                Update τ elements corresponding to the best one
While (nc ≤ nc_max)
Find the minimum function value f_min
If (f_min < F_min)
        Update the global minimum function value F_min
        Else
                If (f_min ≥ F_min) for t times, k=k+1
        Adjust the variables domain
While (max(h_1,…, h_2))
Output F_min
```

Параметри алгоритму й результати тестування

В результаті тестування запропонованого алгоритму були знайдені такі оптимальні параметри: $k = 11$, $\theta = 0.2$, $\Delta m = 2$, $\Delta_1 = 1.25$, $\Delta_2 = 0.05$ і $nc\_max = 50$ - кількість ітерацій, коли мурахи не знаходять кращого розв'язку, після чого k збільшується на 1.

Для порівняння результатів запропонованого алгоритму з оригінальним ACO_R, обидва алгоритми запускалися 50 разів на тестових функціях. Умовою зупинки було досягнення похибки *1e-10*. Характеристиками порівняння є MNFE (median number of function evaluation) – середня кількості обчислень функцій, а також AE (average error) – середня похибка.

Таблиця 1. Порівняння результатів алгоритмів ACOR і ACO*.

Function	MNFE ACO_R	MNFE ACO*	AE ACO*
Sphere x:[-3,7]^n, n=10	1371.1	192.5	8.11e-11
Ellipsoid x:[-3,7]^n, n=10	4452.6	218	8.96e-11
Cigar x:[-3,7]^n, n=10	3841.4	235	8.19e-11
Tablet x:[-3,7]^n, n=10	2567	203	8.74e-11
Rosenbrock x:[-5,5]^n, n=10	7191.1	1256	9.91e-11

Як бачимо, кількість обчислень функції в запропонованому алгоритмі на порядок менше ніж в стандартному, що досягається за рахунок застосування зменшення області пошуку й само-адаптації кількості випару феромонів.

Розглянемо як модифікований алгоритм здатен шукати оптимум, який знаходиться поза заданою початково областю пошуку, для чого на чотирьох різних областях пошуку при $n = 2$, які не містять оптимум, запустимо алгоритм 50 разів для тестових функцій і отримаємо значення ANFE, а також success rate – відсоток успішних запусків алгоритму.

Таблиця 2. Тестування алгоритму на різних початкових областях

	x_1:[100,200] x_2:[50,80]	x_1:[-300,-180] x_2:[-600,-50]	x_1:[1,2] x_2:[-3,-1]	x_1:[100,110] x_2:[-300,-190]	Function
ANFE	155	171	108	163	Zakharov
Succes rate	100%	100%	100%	100%	
ANFE	188	220	108	249	Griewank
Succes rate	70%	80%	100%	25%	
ANFE	156	189	111	159	Rastrigin
Succes rate	100%	100%	100%	100%	
ANFE	253	242	163	423	Rosenbrock
Succes rate	100%	100%	100%	100%	
ANFE	171	x	108	x	Ackley
Succes rate	100%	0%	100%	0%	

Як видно, для більшості тестів даний алгоритм здатен адаптувати простір пошуку і знаходити глобальний мімінум. Лише для функцій griewank і ackley, відсоток успішних запусків є недостатнім, особливо в випадку дуже далекого від оптимуму початкового значення області пошуку.

Висновки

Основною особливістю даної модифікації мурашиного алгоритму, підтвердженою тестами, є можливість так званого широкостороннього пошуку, тобто незалежного від правильно заданого початкового простору. Це дуже важливо для функцій, що належать до реальних проблем, де важко оцінити розташування оптимуму. Така можливість була досягнута за допомогою кількох ключових само-адаптивних нововведень: налаштування області пошуку та його розділення, зміни кількості мурах і приросту феромонів.

Література

1. Dorigo, M., & Gambardella, L. M. Ant colony system: A cooperative learning approach to the traveling salesman problem.// IEEE Transaction on Evolutionary Computation 1(1). – 1999. – PP. 53–66.
2. Dorigo, M., & Stützle, T. (2010). Ant colony optimization: Overview and recent advances. In Handbook of metaheuristics). – 2010. – PP. 227–263.
3. Socha, K., & Dorigo, M. (2008). Ant colony optimization for continuous domains. //European Journal of Operational Research, 185(3). – 2008. – PP. 1155–1173.
4. Gao, S., Zhong, J., & Mo, S. J.Research on ant colony algorithm for continuous optimization problem. //Microcomputer Development, 13(1). – 2003. – PP. 21–22.

Додаток 4. Копії тез доповіді на ПМК-2020

УДК 004.021

К.т.н., доцент Зорін Ю.М., студент Абдураімов Т.З.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

МОДИФІКОВАНИЙ МУРАШИНИЙ АЛГОРИТМ ДЛЯ ВИРІШЕННЯ ЗАДАЧІ КОМІВОЯЖЕРА

Abstract

Yuri Zorin, Assoc. Prof., PhD; Tair Abduraimov, student

Modified Ant Colony Optimization for solving Travelling salesman problem.

It is known that ant colony optimization (ACO) methods are effective for combinatorial optimization. The traveling salesman problem (TSP) is a benchmark used to test new combinatorial optimization algorithms. This paper reviews the application of ACO methods to TSP and discusses some general aspects of ACO that were not previously considered. In fact, it is observed that the length of the solution does not accurately reflect the quality of a particular edge belonging to the solution, but it is used only to relatively assess whether this edge is good or bad in the process of reinforcement training. Based on this observation, we propose modified algorithm - a three-tier Ant system - which are not only easy to implement but also provide better performance compared to the well-known Max-Min Ant system. Performance is assessed using numerical simulations using baseline datasets.

Вступ

Мурашиниий алгоритм (Ant Colony Optimization, ACO) - це евристичний метод пошуку для вирішення NP-складних проблем у комбінаторній оптимізації. Принцип алгоритма полягає в тому, що він імітує біологічну поведінку справжньої колонії мурах, коли мурахи намагаються знайти найкоротший шлях між своїм гніздом та джерелом їжі. Під час пошуку їжі мурахи відкладають на землю речовину, яку називають феромоном. У будь-який час мураха має тенденцію йти шляхом, по якому проходять феромонні стежки, залишені попередніми мурахами. Зрештою, мурахи зможуть знайти найкоротший шлях від джерела їжі до гнізда, і вся колонія піде цим шляхом, щоб транспортувати їжу назад до свого гнізда.

Розроблено багато варіантів ACO для вирішення добре відомої проблеми мандрівного продавця (traveling salesman problem, TSP), серед яких система колоній мурашок (ACS) [1] та система Max-Min Ant (MMAS) [2] є двома найпопулярнішими алгоритмами. Також проводились різні теоретичні дослідження збіжності та інших характеристик алгоритмів ACO [3]. Ці дослідження дають вказівки щодо вибору відповідних параметрів, корисних для підвищення ефективності алгоритму.

Отже, на основі цих досліджень було розроблено модифікований мурашиний алгоритм для задачі комівояжера і порівняно його результати з оригінальним ACO.

Постановка задачі

Метою роботи є розробка модифікованого мурашиного алгоритму для вирішення задачі комівояжера, який перевищує стандартний в сенсі в точності результатів і сталості.

ACO для задачі комівояжера

Натхненний біологічною поведінкою справжньої колонії мурах, ACO розробляється для «штучних мурах». Це робиться шляхом формулювання вихідної задачі оптимізації до задачі найкоротшого шляху у графі, пов’язаним із попередньою задачею. ACO поєднує в собі два типи інформації: евристичну інформацію, яка пов’язана з довжиною ребра графа, та інформаційну навчальну інформацію, яка пов’язана з місцевою інформацією,

представленою інтенсивністю феромонових слідів на цьому ребрі. Інтенсивність феромонів змінюється з часом. Наприклад, його можна або зменшити (випаровувати), якщо жодні інші мурахи не відвідують ребро, або посилити, якщо інші мурахи залишають більше феромону на ребрі. Таким чином, встановлюється правило або набір правил для оновлення інтенсивності феромонів. Правила оновлення в конкретному алгоритмі ACO представляють стратегію пошуку алгоритму і, отже, впливають на його ефективність.

Алгоритм ACO має наступні кроки:

1) ***Ініціалізація***. Визначається кількість мурах *m*, кількість ітерацій *T*, та інші параметри. Інтенсивність феромону встановлюється: $\tau_{ij} = \tau_0$, для всіх $e_{ij} \in \varepsilon$, де ε – набір ребер, які повністю з'єднують вершини.
2) ***for*** *t = 1 : T* ***do***
3) Будуємо *m* розв'язків (для *m* мурах), використовуючи випадкові шляхи.
4) Регулюємо інтенсивність феромонів, використовуючи правила оновлення. Для MMAS маємо:

$$\Delta\tau_{ij} = \begin{cases} (1-\rho)\times 1/L_{gb}, & if\,(e_{ij} \in s^*(t)) \\ 0, & else \end{cases}$$

, де L_{gb} – довжина *s*(t)*

5) ***end for***
6) Отримаємо найкраще рішення *s*(t).*

Опис запропонованого алгоритму

Для модифікації алгоритму використані наступні вдосконалення. По-перше, кількість розв'язків (шляхів) не використовується для оновлення кількості феромонів. По-друге, нижня і верхня межі, τ_{max} та τ_{min}, не визначені в абсолютний спосіб; натомість вони визначаються у формі відношення τ_{max} / τ_{min}. Отже, ці співвідношення встановлюються наступним чином:

$$\frac{\tau_{\max}}{\tau_{\min}} = Nk \quad \frac{\tau_{mid}}{\tau_{\min}} = k$$

, де

$$k = \begin{cases} (N+50)/100, & if\,(N \geq 50) \\ 1, & else \end{cases}$$

Вибір значення співвідношення базується на спеціальному врахуванні складності алгоритму. Маємо τ_{max} / τ_{min} пропорційною до можливої кількості шляхів, тоді як τ_{mid} / τ_{min} - ця пропорційна константа k. Крім того, кількість вершин у кожному екземплярі більше або дорівнює 50, отже, наша найкраща лінійна оцінка для *k* така, як наведена вище.

Як і в роботі Stützle та Hoos [2] запропоновано спосіб згладжування інтенсивності феромонів з метою збільшення розвідки MMAS, коли алгоритм знаходиться поблизу збіжності. Побудова рішення в модифікованому така ж, як і в MMAS. Покращення, порівняно з MMAS, здійснюється за допомогою наступної модифікації правила оновлення для випарювання феромонів:

$$\Delta\tau_{ij} = \begin{cases} (1-\rho)\times\tau_{\max}, & if\,(e_{ij} \in s^*(t)) \\ (1-\rho)\times\tau_{\min}, & else \end{cases}$$

Разом із попередньо визначеним співвідношенням τ_{max} / τ_{min}, це правило оновлення спрощує реалізацію алгоритму та уникає труднощів у визначенні абсолютних значень для τ_{min} та τ_{max}. У порівнянні з MMAS, інтенсивність феромонів, оновлена в запропонованому алгоритмі, змінюється повільніше і завжди залишається в інтервалі $[\tau_{min}, \tau_{max}]$, отже, в алгоритмі виконується менше обчислень.

Результати тестування алгоритму

Щоб оцінити ефективність запропонованого алгоритму (ACO* в таблиці 1), ми чисельно порівнюємо його з MMAS, використовуючи 12 наборів даних тестів, вилучених з TSPLIB95 [4]: 8 наборів даних для TSP (ei51, kroA100, kroB150, d198, kroA200, lin318, att532 і rat783). Експериментальні дослідження для MMAS проводились програмним пакетом ACOTSP (версія 1.0), розробленим Stützle [5], для перших 8 наборів даних.

Кожен алгоритм запускався 25 разів із кількістю рішень S = 10000 × N. Інші параметри: α = 1 та β = 2, m = N / 2 та ρ = 0,02. Результати експерименту представлені в таблиці 1.

Для конкретного алгоритму та набору даних верхнє число відображає середню довжину найкращих знайдених рішень. Поруч із ним - відсотковий показник у дужці, що вказує на відхилення від оптимального значення (позначене як “opt” у таблиці 1). Нижній лівий та правий показники представляють найкращі та найгірші рішення за весь час роботи. Жирні цифри вказують на найкращі результати впроваджених алгоритмів для певного набору даних.

Таблиця 1. Порівняння результатів алгоритмів.

	ACO*		**MMAS**	
eil51 opt: 426	426.46 (0.10%)		**426 (0%)**	
	426	428	426	426
kroA100 opt: 21282	21304.3 (0.11%)		**21293.44 (0.05%)**	
	21282	21378	21282	21379
kroB150 opt: 26130	26315.72 (0.71%)		**26142.28 (0.05%)**	
	26176	26438	26130	26181
dl98 opt: 15780	**15950.96 (1.08%)**		15954.04 (1.1%)	
	15875	16034	15884	16005
kroA200 opt: 29368	29665.84 (1.01%)		**29436.56 (0.23%)**	
	29422	29843	29394	29540
lin318 opt: 42029	42956.96 (2.21%)		**42260.48 (0.55%)**	
	42837	43324	42136	42373
att532 opt: 27686	28767.1 (3.9%)		**28113.08 (1.54%)**	
	28636	28920	27885	28683
rat783 opt: 8806	9283.6 (5.42%)		**8949 (1.62%)**	
	9249	9336	8901	8998

З таблиці 1 можна побачити, що середні значення правильно відображають ефективність алгоритму, тоді як найкращі та найгірші значення використовуються як еталон для динамічного діапазону рішень, знайдених алгоритмом.

Що стосується середніх значень, можна побачити, що ACO* дає кращі результати, ніж MMAS, за винятком тестів dl98. Враховуючи найкращі значення, результати показують, що всі алгоритми дають оптимальні значення для тестів eil51 та kroA100.

Висновки

Алгоритми ACO ефективно використовуються у комбінаторній оптимізації, в якій TSP є важливою проблемою. Проведені різні дослідження щодо характеристик алгоритмів

та розроблені різні правила оновлення феромонів. Визначення верхньої та нижньої меж інтенсивності феромонів не повинно залежати від функції якості. Натомість це повинно залежати лише від стратегії пошуку - інтенсифікація чи диверсифікація - що потім відображається відношенням верхньої межі до нижньої межі. Ця стратегія спрощує реалізацію алгоритму, оскільки нам не потрібно знати абсолютні значення цих меж.

На основі вищезазначеного спостереження було запропоновано модернізований мурашиний алгоритм, який не тільки може бути легко впроваджений, але також забезпечує кращу продуктивність та сталість для більшості тестувань порівняно з найбільш розвинуним на даний момент алгоритмом MMAS, що було показано в результаті порівняльного аналізу за допомогою чисельного моделювання з використанням 8 наборів даних базових показників.

Література

1. M. Dorigo and L. M. Gambardella, “Ant colony system: A cooperative learning approach to the traveling salesman problem,” IEEE Transactions on Evolutionary Computation, vol. 1, no. 1. – 1997. – PP. 53-66.
2. T. Stützle and H. H. Hoos, “MAX-MIN ant system” Future Generation Computer Systems, vol. 16, no. 9. – 2000. – PP. 889-914.
3. Y. Zhou, “Runtime analysis of an ant colony optimization algorithm for TSP instances,” IEEE Transactions on Evolutionary Computation, vol. 13, no. 5. – 2009. – PP. 1083-1092.
4. G. Reinelt. Tsplib. Heidelberg University. Germany. [Online]. Available: http://comopt.ifi.uniheidelberg.de/software/TSPLIB95/
5. T. Stützle. Acotsp. (Software package). [Online]. Available: http://www.aco-metaheuristic.org/acocode/public-software.html

Додаток 5. Копії слайдів презентації