Одеський національний університет імені І. І. Мечникова
Факультет математики, фізики та інформаційних технологій
Кафедра оптимального керування та економічної кібернетики

Дипломна робота

бакалавра

на тему: **«Аналіз підходів до оптимізації гіперпараметрів у моделях машинного навчання»**

«Analysis of approaches to optimization of hyperparameters in machine learning models»

Виконала: студентка денної форми навчання
спеціальності 113 Прикладна математика
Ткачова Таїсія Максимівна

Керівник: канд. фіз.-мат. наук, доц. Страхов Є. М.
Рецензент: доктор фіз.-мат. наук, доц. Скрипник Н. В.

Рекомендовано до захисту:
Протокол засідання кафедри
№ ___ від «_____» _____ р.
Завідувач кафедри

_____ _____

Захищено на засіданні ЕК № _____
Протокол № ____ від «____» ____ р.
Оцінка _____ / _____ / _____
Голова ЕК

_____ _____

Одеса — 2022 р.

ЗМІСТ

ВСТУП

Оптимізація моделі є однією з найскладніших проблем у пошуку рішень задач машинного навчання. Етапом розв'язку задач машинного навчання, завдяки якому досягається оптимальність моделі називають налаштуванням гіперпараметрів (або оптимізацією гіперпараметрів).

Гіперпараметри – це параметри, що встановлюються перед початком процесу навчання та визначають архітектуру моделі. Префікс «гіпер» говорить про те, що це параметри «верхнього рівня», тобто контролюють процес навчання та параметри моделі, що випливають із неї.

Налаштування гіперпараметрів є важливою частиною керування поведінки моделі машинного навчання. Цей процес полягає у знаходженні такої комбінації значень гіперпараметрів, яка найкраще максимізує продуктивність моделі, мінімізуючи попередньо визначену функцію втрат.

Дослідження останніх років демонструють, що автоматизовані методи налаштування гіперпараметрів можуть прискорити процес оптимізації та знайти конфігурації гіперпараметрів, які призводять до кращих моделей. Нині перенесення знань з попередніх експериментів у нові представляє особливий інтерес, оскільки було показано, що це дозволяє додатково покращити оптимізацію гіперпараметрів.

Практичне значення цього дослідження складається в тому, що попри існування тереоретичних причини віддавати перевагу тому, чи іншому алгоритму при вирішенні проблеми пошуку оптимальних гіперпараметрів, поточне розуміння машинного навчання не дозволяє заздалегідь передбачити, чи буде один метод працювати краще, ніж інший.

Метою дипломної роботи є вивчення та дослідження методів налаштування гіперпараметрів для різноманітних моделей машинного навчання.

Предмет: застосування автоматизованих підходів налаштування гіперпараметрів для вирішення задачі пошуку оптимальних гіперпараметрів.

Об'єкт: класичні алгоритми оптимізації гіперпараметрів та їх особливості.

Методи дослідження: проведення експериментів із використанням

бібліотек та фреймворків для налаштування гіперпараметрів.

В рамках дипломної роботи було розглянуто основні методи налаштуванням гіперпараметрів на задачі класифікації. Результати було порівняно та проаналізовано за допомогою розрахування метрик та фіксації часових витрат.

РОЗДІЛ 1

ТЕОРЕТИЧНА ЧАСТИНА

1.1 Алгоритми машинного навчання

Машинне навчання зазвичай має справу з трьома типами задач, а саме: класифікація, регресія та кластеризація. Залежно від типу задачі можна визначити відповідний алгоритм. Алгоритмом машинного навчання називають метод за допомогою якого система штучного інтелекту передбачає вихідні значення з заданого вхідного набору даних. У цій роботі будуть розглянуті алгоритми розв'язування задач класифікації.

1.1.1 Метод k-найближчих сусідів

Одним із найпростіших та широко використовуваних алгоритмів навчання є алгоритм k-найближчих сусідів (k-nearest neighbors, kNN).

Класифікація k найближчого сусіда була розроблена з необхідності виконання дискримінантного аналізу, коли надійні параметричні оцінки густин ймовірності невідомі або їх важко визначити. У 1951 році Фікс і Ходжес представили непараметричний метод класифікації шаблонів, який від того часу став відомий як правило найближчого сусіда k. [1]

Цей метод заснований на принципі, що екземпляри в наборі даних часто дуже близькі до інших екземплярів з подібними властивостями. Якщо екземпляри позначені міткою класифікації, то значення мітки некласифікованого екземпляра можна визначити, спостерігаючи за класом його найближчих сусідів. Метод найближчих сусідів знаходить k екземплярів, найближчих до екземпляра запиту, і визначає його клас, ідентифікуючи найпоширенішу мітку класу.

Перевагою KNN є простота реалізації. Зазначений метод легко реалізувати, тому що єдине, що потрібно обчислити, це відстань між різними точками на основі даних різних об'єктів, і цю відстань можна легко розрахувати за допомогою формули відстані, таких як – Евклідіова або

Манхеттенська.

Основним недоліком цього методу є те, що при обчисленні відстані використовуються всі функції, що робить метод обчислювально інтенсивним, особливо коли розмір навчального набору зростає. Крім того, точність класифікації k-найближчого сусіда значно погіршується через наявність шуму або невідповідних ознак. [2]

1.1.2 Наївний баєсівський класифікатор

Наївний класифікатор Баєса (naive Bayes classifier) - це простий імовірнісний класифікатор, заснований на застосуванні теореми Баєса з сильними припущеннями щодо незалежності ознак. [2] Більш описовим терміном для базової ймовірнісної моделі буде незалежна модель ознак. Припущення про незалежність роблять порядок ознак нерелевантним, тому наявність однієї ознаки не впливає на інші в задачах класифікації.

Наївні баєсівські класифікатори часто показують набагато кращі результати в багатьох складних реальних ситуаціях, ніж можна було б очікувати. Вважається, що наївні класифікатори Баєса працюють напрочуд добре для багатьох задачах класифікації реального світу за певних умов.

До переваг цього метода відносяться відсутність необхідності великої кількості навчальних даних для оцінки параметрів; можливість кодування взаємозалежностей між змінними та прогнозування подій; а також можливість включати попередні знання та дані. [3]

Недоліком використання наївного баєсівського підходу полягає в тому, що ознаки є умовно незалежними, що не завжди так. Однак слід зазначити, що незважаючи на це, баєсівські класифікатори дають задовільні результати, оскільки вони зосереджені на ідентифікації класу екземпляра, а не на точних ймовірностях.

1.1.3 Логістична регресія

Логістична регресія (logistic regression) - це тип регресії, який передбачає ймовірність виникнення події шляхом підгонки даних до логістичної

функції. [4] Незалежні змінні в цій моделі виступають у ролі предикторів залежної змінної та можуть бути виміряні за номінальною, порядковою, інтервальною або шкалою відношень, тоді як залежна змінна має двійковий формат. Зв'язок між залежною та незалежними змінними є нелінійним.

Перевага логістичної регресії полягає в тому, що вона вимірює не тільки кореляцію предикторів, а й напрям їх асоціації (позитивний чи негативний). Також алгоритм добре себе проявляє у випадку лінійно розділеного набору даних.

Основним обмеженням логістичної регресії є припущення про лінійність між залежними та незалежними змінними. Тому що у реальному світі дані рідко можна розділити лінійно, у більшості випадків дані будуть безладними. Іншим недоліком є те, що використання логістичної регресії може призвести до перенавчання моделі, якщо кількість спостережень буде менша за кількість ознак.

1.1.4 Метод опорних векторів

Метод опорних векторів (support vector machines, SVM) є одним із дискримінаційних методів класифікації, які зазвичай вважаються більш точними. [2] Зазначений метод класифікації будується на принципі мінімізації структурного ризику з теорії обчислювального навчання. Ідея цього принципу полягає в тому, щоб знайти гіпотезу, яка буде гарантувати найменшу ймовірність помилки, тоді як традиційні методи навчання засновані на мінімізації емпіричного ризику, який намагається оптимізувати продуктивність навчального набору.

До переваг SVM відносяться здатність моделювання нелінійних меж рішення та нижчий рівень схильності до перенавчання, ніж в інших методах. До того ж, метод опорних векторів у порівнянні з логістичною регресією є менш схильний до викидів, оскільки він дбає лише про точки, найближчі до межі рішення або векторів підтримки.

Недоліком методу є його висока алгоритмічна складність і великі вимоги до пам'яті. [3] Через це, швидкість як під час навчання, так і при тестуванні – повільна.

1.1.5 Дерево рішень

Дерево рішень (decision tree) – це методика прогнозного моделювання, яка найчастіше використовується для розв'язання проблем класифікації. [3] Цей принцип ґрунтується на основі статистичних оцінок і серії перевірок логічних умов.

В теорії графів структура дерева описується як орієнтований граф без циклів. Це означає, що дерево складається з вершин і ребер, що їх з'єднують, кожна вершина має одну вхідну і дві або більше вихідних ребер до сусідніх вершин. Корінь дерева відповідає початковому стану процесу класифікації. Кожна вершина пов'язана з логічною функцією, що обчислює умову розгалуження. Здебільшого поділ спирається на одну з характеристик вхідної величини або на попередньо визначеному наборі правил поділу. Вхідний стан відповідає першій логічній перевірці, яку потрібно виконати. Вихідним результатом є вершини, так звані листки, що представляють отримані класи. В результаті отримаємо графічне рішення, а саме дерево, при обході якого, починаючи з кореня, кожен вузол ділить набір рішень, допоки не досягне кінцевої вершини, яка визначає клас.

Перевагами дерева рішень є те, що метод інтуїтивно зрозумілий та надає візуальний розв'язок. Ще одна перевага це здатність навчатися на даних, які містять помилки або відсутні значення.

Основний недолік використання дерева рішень полягає в схильності до перенавчання, що може призвести до великих і надмірно складних деревоподібних структур, у тому разі коли набір даних має велику кількість записів. [2]

1.1.6 Випадковий ліс

Алгоритми випадкового лісу (random forest) утворюють сімейство методів класифікації, які спираються на комбінацію кількох дерев рішень. Особливістю таких ансамблів класифікаторів є те, що їх деревоподібні компоненти на основі випадкових лісів вирощуються з певної кількості випадковості. Виходячи з цієї ідеї, випадковий ліс визначається як загальний

принцип рандомізованих ансамблів дерев рішень. [5]

Основною перевагою класифікатора випадкових лісів перед іншими методами дерева рішень є те, що кожного разу дерево вирощується до максимальної глибини на нових навчальних даних за допомогою комбінації функцій. Так би мовити повністю дорослі дерева не обрізаються. [6]

Недоліком методу є висока дисперсія. На практиці нерідкі випадки, коли невелика зміна навчального набору даних призводить до зовсім іншого результативного дерева. Причина полягає в ієрархічній структурі класифікатора, інакше кажучи, якщо виникає помилка у вершині близькій до кореня дерева, то вона поширіться аж до листків [7]

1.2 Налаштування гіперпараметрів

Як зазначалося вище, важливість гіперпараметрів полягає в їх здатності безпосередньо керувати поведінкою алгоритму навчання. Іншими словами, вибір відповідних гіперпараметрів відіграє значну роль у знаходженні оптимальної моделі. Традиційно пошук найкращих значень гіперпараметрів для певного набору даних виконується вручну. Однак, щоб встановити ці значення, дослідники зазвичай покладаються на свій минулий досвід використання того чи іншого алгоритму машинного навчання. Проблема полягає в тому, що найкращі гіперпараметри моделі для одного конкретного набору даних не будуть найкращими для іншого. Очевидно, що лише на основі попереднього досвіду важко визначити оптимальні значення гіперпараметрів, для цього потрібен більш автоматизований та контрольований підхід. Найпоширенішими алгоритмами оптимізації являються пошук у сітці, випадковий пошук, баєсівська оптимізація та оптимізація рою частинок.

Перед тим як перейти до детального розгляду алгоритмів оптимізації важливо пояснити процес, який називається перехресною перевіркою, тому що він вважається ключовим кроком у процесі встановлення гіперпараметрів. Перехресна перевірка (cross-validation, CV) – це статистичний метод оцінки та порівняння алгоритмів навчання шляхом поділу даних на два сегменти: один використовується для навчання моделі, а інший – для перевірки моделі. [8] Основним підходом цього методу є k-кратна перехресна

перевірка (k-fold CV). Насамперед набір даних розбивається на k підмножин, далі кожний з k підмножин навчається і перевіряється на решті даних. Остаточна оцінка моделі отримується за допомогою усереднення моделі по кожній з підмножин. [9]

1.2.1 Пошук у сітці

Класичним та найпростішим підходом для налаштування гіперпараметрів є вичерпний пошук у сітці (Grid Search). Сітка пошуку базується на основі декартового добутку усіх можливих комбінацій гіперпараметрів. Потім будуються моделі для цих комбінацій, і вибираються найкращі гіперпараметри завдяки методики перехресної перевірки. [1]

Перевагою такого пошуку є його ретельність. Оскільки оцінюється кожна можлива комбінація гіперпараметрів, пропустити найкращу майже неможливо. Недоліком є те, що час обробки цілих наборів параметрів може бути величезним, а отже, кількість параметрів для дослідження має практичні обмеження.

1.2.2 Випадковий пошук

Як випливає з назви, випадковий пошук - це алгоритм, в якому використовуються випадкові комбінації гіперпараметрів для пошуку найкращого рішення для побудованої моделі. На відміну від фіксованих значень у методі пошуку у сітці, числові параметри можна вказувати у вигляді діапазону.

Дослідники вважають, що випадковий пошук більш практичний, ніж пошук у сітці, через те що його можна застосовувати навіть при використанні кластера, який може вийти з ладу. До того ж метод дозволяє змінювати роздільну здатність на льоту, а саме: додавати нові випробування до набору або ігнорувати невдалі. [10]

Основна перевага полягає в тому, що не треба перейматися про час виконання, тому що можна контролювати кількість ітерацій. Вагомим недоліком алгоритму випадкового пошуку є те, що він не використовує інформацію з попередніх ітерацій для вибору наступного набору, а також

не використовує стратегію для прогнозування наступного випробування. [11] Крім того, оскільки вибір комбінацій повністю випадковий, то пошук дає високу дисперсію під час обчислень.

1.2.3 Баєсівська оптимізація

Алгоритм баєсівської оптимізації побудован таким чином, що кожна ітерація вивчає попередню, а поточний результат допомагає створити наступну ітерацію. Наведена оптимізація нагадує метод випадкового пошуку в тому сенсі, що він обирає підмножину комбінацій гіперпараметрів, втім вони відрізняються за способом вибору кожної комбінації.

Баєсівська оптимізація розглядає процес налаштування гіперпараметрів як оптимізацію функції чорного ящика. [12] Функція, яку потрібно оптимізувати, — це точність прогнозування моделі на конкретному тестовому наборі. Для мінімізації цієї функції можна застосувати будь-яку глобальну систему оптимізації. Найпоширенішою моделлю, яка використовується для апроксимації цільової функції, є процес Гаусса.

Основною відмінністю методу баєсівської оптимізації від вище згаданих методів полягає в тому, що процес налаштування розглядає лише комбінації гіперпараметрів, які передбачено повинні дати хороші результати, а не всі можливі комбінації в зазначеному діапазоні.

Перевага баєсівського підходу оптимізації полягає в ефективності пошуку, тому що не розглядається кожна комбінація в просторі пошуку, як це відбувається в пошук у сітці, і водночас метод виконується більш систематично, ніж у разі випадкового пошуку. До недоліків алгоритму належить можливість застрягання на локальному оптимальному рівні.

1.2.4 Оптимізація рою частинок

Оптимізація рою частинок (particle swarm optimization, PSO) - це типовий алгоритм сімейства ройового інтелекту, вперше запропонований Кеннеді та Еберхартом у 1995 році. [13] Метод черпає натхнення з того, як зграя птахів у пошуках джерел їжі змінює своє положення, виходячи з

їхнього індивідуального колишнього положення та положення їхнього рою. [14]

Оптимізація рою частинок вирішує проблему, намагаючись оптимізувати рішення ітераційним способом щодо певної міри якості, дозволяючи групі частинок (рою) сканувати простір пошуку напіввипадковим чином. [15] Алгоритм знаходить оптимальне рішення шляхом обміну інформації та співпраці між частинками в групі. У PSO рій має набір частинок, кожна з яких представлена вектором, що містить поточне положення, швидкість і можливе найкраще положення частинки. Після ініціалізації положення та швидкості для кожної частинки обчислюється показник оптимальності його поточного положення. На наступній ітерації швидкість кожної частинки змінюється відповідно до розрахункової інформації попередньої ітерації, тобто розташування частинки та поточного глобального оптимального ріншення. Далі частинки рухаються відповідно до свого нового вектора швидкості. Такі кроки повторюються до тих пір, поки не буде досягнутий певний критерій збіжності або завершення.

До переваг оптимізації рою частинок можна віднести можливість обходити великий багатовимірний простір пошуку за допомогою простої, але ефективної процедури. Недоліком цієї оптимізації є те, що алгоритм часто знаходить лише локальний оптимум, особливо це помітно при оптимізації дискретних гіперпараметрів.

1.3 Бібліотеки та пакети

Scikit-learn – це бібліотека на Python з відкритим вихідним кодом, що надає прості та ефективні інструменти для прогнозного аналізу даних. [16]. За допомогою scikit-learn, можна застосувати два найпоширеніші підходи до пошуку гіперпараметрів: GridSearchCV, що вичерпно розглядає всі комбінації параметрів, та RandomizedSearchCV, який може вибирати задану кількість кандидатів із простору параметрів із заданим розподілом. Обидва методи мають послідовні аналоги HalvingGridSearchCV і HalvingRandomSearchCV, які можуть бути набагато швидшими при пошуку хорошої комбінації параметрів. [17]

Bayes_opt – це обмежений пакет глобальної оптимізації, заснований на баєсівській інтерференції та гаусовому процесі, який намагається знайти максимальне значення невідомої функції за якомога меншу кількість ітерацій. [18] BayesianOptimization() зі згаданої бібліотеки може приймати будь-яку функцію чорного ящика як вхідні дані та максимізувати вихідне значення, що повертається цією функцією. Оптимізація працює шляхом побудови апостеріорного розподілу функцій (тобто гаусового процесу), який найкраще описує функцію, яку потрібно оптимізувати. З метою мінімізування кількісті кроків, метод використовує проблему оптимізації проксі (знаходження максимуму функції збору даних), яка, незважаючи на складну задачу, є дешевшою в обчислювальному сенсі.

PySwarms — це дослідницький набір інструментів для оптимізації роя частинок на Python. [19] Бібліотека PySwarms дає змогу використовувати алгоритм оптимізації роя частинок за допомогою інтерфейсу високого рівня. Крім того, цей інструмент дозволяє реалізувати і використовувати різноманітні методи для рою багатьох частинок та з легкістю відображувати графічне зображення рішення.

РОЗДІЛ 2

ЕКСПЕРИМЕНТИ ТА РЕЗУЛЬТАТИ

2.1 Обчислювальний експеримент

Головна мета дослідження полягає в тому, щоб проаналізувати найпоширеніші методи оптимізації гіперпараметрів для різноманітних моделей машинного навчання.

Не менш важливим кроком при створенні моделей є їх оцінка, оскільки вона визначає, чи було досягнуто поставленої мети, до того ж це дозволяє порівнювати різні підходи до моделювання та редагувати наступні дослідження.

Найбільш обгрунтованим показником ефективності вважають співвідношення між кількістю правильно класифікованих спостережень і загальною кількістю спостережень, ця метрика називається точністю (accuracy). Він призначений для багатокласових випадків і зазвичай є хорошим показником, але лише якщо набір даних збалансований, інакше точність не можна вважати надійним, оскільки він надає занадто оптимістичну оцінку здатності класифікатора.

Нині міра F1 також широко використовується в більшості прикладних областей машинного навчання, не тільки в бінарних випадках, а й у багатокласових. Оцінку F1 можна інтерпретувати як середньозважене значення між Precision (частка правильно класифікованих позитивних спостережень, поділена на загальну кількість позитивно передбачених спостережень) та Recall (частка правильно класифікованих позитивних спостережень, поділена на загальну кількість позитивно класифікованих одиниць), де оцінка F1 досягає найкращого значення при 1 і найгіршого результату при 0. Для мультикласових випадків передбачені процедури усереднення мікро/макро F1, які навіть можуть бути призначені для спеціальної оптимізації.

2.2 Опис набору даних

В цій дипломній роботі експерименти проводилися на відкритому датасеті "Bank Marketing". [20]

За допомогою вище загаданого датасету необхідно проаналізувати останню маркетингову кампанію банку та визначити закономірності, які допоможуть зробити висновки щодо розробки стратегій для покращення майбутніх маркетингових кампаній банку.

Набір даних являє собою класичну інформацію про маркетингову кампанію фінансової установи. Нам надають таку інформацію про клієнтів як:

1) age - вік
2) job - вид зайнятості, можливі значення: адміністратор, комірець, підприємець, покоївка, менеджер, пенсіонер, самозайнятий, надання послуг, студент, технік, безробітний або невідомий (тобто інформація відсутня)
3) marital - сімейний стан: розлучений, одружений, неодружений, або невідомий
4) education - освіта: базова, середня, безграмотний, професійна, вища освіта або невідомий
5) default - чи має клієнт прострочений кредит
6) balance - баланс на рахунку
7) housing - наявність житлового кредиту
8) loan - наявність позик
9) contact - контактний тип зв'язку
10) day - день тижня, коли був останній контакт з клієнтом
11) month - місяць року, коли був останній зв'язок з клієнтом
12) duration - тривалість останнього спілкування в секундах
13) campaign - кількість контактів до клієнта, здійснених під час цієї кампанії
14) previous - кількість дзвінків до клієнта, виконаних до поточної кампанії
15) pdays - кількість днів, що минули після останнього зв'язку з клієнтом під час попередньої кампанії

16) poutcome - результат попередньої маркетингової кампанії, можливі значення: невдача, успіх, неіснуючий (мається на увазі, що клієнт новий і до цієї кампанії його не було в базі)

Саме аналіз цих показників і повинен вказати чи зацікавлений клієнт в депозиті (строковій інвестиції, що включає внесення грошей на рахунок у фінансовій установі).

2.3 Попередня обробка

Зважаючи на те, що в датасеті присутні категоріальні показники, виділимо трішки уваги попередній обробці даних.

Одночасне кодування (one-hot encoding) - це процес перетворення категоріальних ознак у форму, яка може бути подана в алгоритми машинного навчання. Такий метод попередньої обробки категоріальних змінних створює нову двійкову ознаку для кожного можливого класу та призначає значення 1 ознакові кожного зразка, що відповідає його вихідному класу. Бібліотека pandas містить функцію get_dummies(), яка і допомагає перетворювати категоріальні ознаки в фіктивні/індикаторні змінні. [21] Ця техніка була використана для перекодування ознак, що містять велику кількість значень, таких як job, marital та education.

Ознаки default, housing та loan, які приймають значення «так» чи «ні» було перекодовано у 1 і 0 за допомогою функції map() з тієї ж бібліотеки. Ця функція зіставляє значення числового ряду відповідно до вхідних значень ознаки. [22]

2.4 Аналіз результатів

Початкові оцінки моделей зі значеннями за замовчуванням до використання методів оптимізації гіперпараметрів зобразимо у вигляді таблиці.

Алгоритм	Accuracy	F1
Метод k-найближчих сусідів	0.7485	0.7483
Наївний баєсівський класифікатор	0.7123	0.71
Логістична регресія	0.8015	0.8012
Метод опорних векторів	0.7402	0.7372
Дерево рішень	0.8004	0.8003
Випадковий ліс	0.8484	0.8485

Для алгоритму k-найближчих сусідів у ролі гіперпараметра, який необхідно оптимізувати, виступає n_neighbors - кількість сусідів для запитів. Отримані результати представлені нижче.

Метод	Параметри	Accuracy	F1	Час (с)
GridSearchCV	n_neighbors=29	0.7718	0.7714	542.253
HalvingGridSearchCV	n_neighbors=15	0.7632	0.7629	378.366
RandomizedSearchCV	n_neighbors=24	0.7692	0.7689	87.057
HalvingRandomSearchCV	n_neighbors=17	0.7621	0.7618	108.189
BayesianOptimization	n_neighbors=43	0.7628	0.7623	62.732
PSO	n_neighbors=17	0.7621	0.7618	1756.19

В цьому випадку найякісніший результат продемонстрував метод пошуку у сітці, при цьому він має не найбільші часові затрати проти методу оптимізації рою частинок.

В наївному баєсівський класифікаторі будемо оптимізувати гіперпараметр var_smoothings - частка найбільшої дисперсії всіх ознак, яка додається до дисперсій для стабільності обчислень.

Метод	Параметри	Accuracy	F1	Час (с)
GridSearchCV	var_smoothing=1e-08	0.7442	0.7434	8.026
HalvingGridSearchCV	var_smoothing=1e-08	0.7442	0.7434	9.369
RandomizedSearchCV	var_smoothing=1e-08	0.7442	0.7434	6.658
HalvingRandomSearchCV	var_smoothing=1e-08	0.7442	0.7434	7.322
BayesianOptimization	var_smoothing=1e-08	0.7442	0.7434	4.882
PSO	var_smoothing=1e-3	0.6912	0.6735	78.574

Оскільки майже всі методи знайшли однакове значення гіперпараметру, то на цьому прикладі наочно видно перевагу у швидкості послідовних аналогів методів випадкового пошуку та пошуку у сітці над їх витоками. Але все ж таки першість посідає метод баєсівської оптимізації. У цьому разі метод рою частинок не продемонстрував гарні результати як за якістю моделі, так і за швидкістю знаходження оптимального гіперпарамметра.

Для логістичної регресії обрали наступні параметри:

1) penalty - норма стягнення, можливі значення l1, l2, elasticnet або ніякий;
2) C - обернена сила регуляризації, менші значення визначають сильнішу регуляризацію;
3) solver - алгоритм для використання в задачі оптимізації.

Метод	Параметри	Accuracy	F1	Час (с)
GridSearchCV	C=0.0335981828628378, penalty='l2', solver='liblinear'	0.8069	0.8066	106.097
HalvingGridSearchCV	C=206.913808111479, penalty='l2', solver='liblinear'	0.8094	0.8091	37.449
RandomizedSearchCV	C=0.0127427498570313, penalty='l2', solver='liblinear'	0.8047	0.8044	26.479
HalvingRandomSearchCV	C=78.47599703514607, penalty='l1', solver='liblinear'	0.8090	0.8087	21.328
BayesianOptimization	C=11.2883789, penalty='l2', solver='liblinear'	0.8090	0.8087	12.246
PSO	C=3.54697711, penalty='l2', solver='liblinear'	0.8097	0.8094	539.043

Найшвидкішим методом підбору виявися алгоритм баєсівської оптимізації, при чому якість знайденого рішення не сильно поступається найкращому за точністю знайденому методом скороченого пошуку у сітці чи оптимізації рою часток.

Для методу опорних векторів у ролі параметрів визначили значення оберненої сили регуляризації C та коефіцієнт ядра gamma.

Метод	Параметри	Accuracy	F1	Час (с)
GridSearchCV	C=1, gamma=0.0001	0.7689	0.7686	974.775
HalvingGridSearchCV	C=1, gamma=0.0001	0.7689	0.7686	584.071
RandomizedSearchCV	C=1, gamma=0.0001	0.7689	0.7686	1047.453
HalvingRandomSearchCV	C=10, gamma=0.0001	0.7506	0.749	26.189
BayesianOptimization	C=1, gamma=0.0001	0.7689	0.7686	986.545
PSO	C=1000, gamma=0.0001	0.7212	0.7211	5289.64

На відміну від попередніх досліджень в поточному випадку оптимізація методом Баєса, не є найкращим за швидкістю, тут перевагу має послідовний аналог методу пошуку у сітці.

Для дерева рішення оптимізувати будемо наступні параметри:

1) max_leaf_nodes - максимальна кількість листових вершин;
2) min_samples_leaf - мінімальна кількість зразків, яка повинна бути у вузлі листка;
3) min_samples_split - мінімальна кількість вибірок, необхідних для розділення внутрішнього вузла.

Метод	Параметри	Accuracy	F1	Час (с)
GridSearchCV	max_leaf_nodes=18, min_samples_leaf=2, min_samples_split=2	0.8169	0.817	221.019
HalvingGridSearchCV	max_leaf_nodes=10, min_samples_leaf=6, min_samples_split=8	0.8140	0.8140	147.259
RandomizedSearchCV	max_leaf_nodes=18, min_samples_leaf=4, min_samples_split=10	0.8169	0.817	8.654
HalvingRandomSearchCV	max_leaf_nodes=4, min_samples_leaf=6, min_samples_split=12	0.7861	0.7839	145.164
BayesianOptimization	max_leaf_nodes=18, min_samples_leaf=6, min_samples_split=2	0.8169	0.817	8.671
PSO	max_leaf_nodes=10, min_samples_leaf=3, min_samples_split=2	0.8029	0.8029	1016.15

Перше на що слід звернути увагу так, це часові витрати алгоритмів випадкового пошуку та баєвсівської оптимізації, незважаючи на складність задачі, методи дуже швидко впоралися. Найкращі результат продемонстрували методи випадкового пошуку, пошуку у сітці та баєвсівської оптимізації, при цьому вибрана комбінація гіперпараметрів в кожного алгоритма різна.

У методі випадкового лісу оптимізувати будемо такі параметри:

1) n_estimators - кількість дерев у лісі;
2) min_samples_leaf - мінімальна кількість зразків, яка повинна бути у вузлі листка;
3) min_samples_split - мінімальна кількість вибірок, необхідних для розділення внутрішнього вузла.

Метод	Параметри	Accuracy	F1	Час (с)
GridSearchCV	min_samples_leaf=2, min_samples_split=10, n_estimators=150	0.8488	0.8488	1037.61
HalvingGridSearchCV	min_samples_leaf=2, min_samples_split=10, n_estimators=150	0.8488	0.8488	466.232
RandomizedSearchCV	min_samples_leaf=3, min_samples_split=8, n_estimators=100	0.8527	0.8527	40.188
HalvingRandomSearchCV	min_samples_leaf=2, min_samples_split=8, n_estimators=50	0.8527	0.8528	278.205
BayesianOptimization	min_samples_leaf=4, min_samples_split=8, n_estimators=150	0.8517	0.8517	59.314
PSO	min_samples_leaf=3, min_samples_split=6, n_estimators=100	0.8502	0.8502	5239.93

Як і в минулому випадку часові витрати алгоритмів випадкового пошуку та баєвсівської оптимізації найменші. Найкращі результат продемонстрували метод випадкового пошуку та його послідовний аналог.

ВИСНОВКИ

Під час виконання дипломної роботи було вивчено загальні підходи налаштування гіперпараметрів для різноманітних моделей машинного навчання.

За допомогою бібліотеки scikit-learn та bayes_opt було побудовано чотири основні підходи пошуку оптимальних гіперпараметрів. Найякісніші комбінації гіперпараметрів переважно знаходили методи пошуку у сітці та баєсівської оптимізації. Найменші часові витрати, з відносно хорошою якістю моделі, продемонстрував алгоритм баєсівської оптимізації. Щодо методу оптимізації рою часток, то він демонстрував середні значення якості моделі, але його недоліком виявилася швидкість пошуку. Метод випадкового пошуку з його послідовним аналогом також поступається іншим алгоритмам по показникам якості та часовим витратам.

На закінчення, я вважаю, що є ще багато засобів для вдосконалення поточного дослідження. Наприклад, у якості підходів для налаштування гіперпараметрів розглянути метод оптимізації градієнтного спуску або ж генетичні алгоритми. А також поставити експеримент для іншого типу задач і, як наслідок, інших алгоритмів машинного навчання.

СПИСОК ЛІТЕРАТУРИ

1. Swamynathan M. Mastering Machine Learning with Python in Six Steps: A Practical Implementation Guide to Predictive Data Analytics Using Python, 2017
2. Fadi A. Journal of Advances in Information Technology, 2010
3. Adebowale A., Idowu S. A., Amarachi A. A. Comparative Study of Selected Data Mining. Algorithms Used For Intrusion Detection, 2013
4. Caraciolo M. Machine Learning with Python - Logistic Regression, 2011 *http://aimotion.blogspot.com/2011/11/machine-learning-with-python-logistic.html*
5. Breiman L. Random forests. Machine Learning Journal Paper, 2001 *https://link.springer.com/content/pdf/10.1023/A:1010933404324.pdf*
6. Pal M. Random forest classifier for remote sensing classification. International Journal of Remote Sensing, 2005
7. Azar A. T., Elshazly H. I., Hassanien A. E., Elkorany A. M. A random forest classifier for lymph diseases. Computer Methods and Programs in Biomedicine, 2014
8. Refaeilzadeh P., Tang L., Liu H. Cross-Validation, 2009 *https://link.springer.com/referenceworkentry/10.1007/978-0-387-39940-9_565*
9. Scikit-learn. Cross-validation: evaluating estimator performance *https://scikit-learn.org/stable/modules/cross_validation.html*
10. Bergstra J., Bengio Y. Random Search for Hyper-Parameter Optimization, 2012 *https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf*
11. Elgeldawi E., Sayed A., Galal A. R., Zaki A. M. Hyperparameter Tuning for Machine Learning Algorithms. Used for Arabic Sentiment Analysis, 2021 *https://www.mdpi.com/2227-9709/8/4/79*
12. Nguyen V. Bayesian Optimization for Accelerating Hyper-Parameter Tuning, 2019
13. Kennedy J. The particle swarm: social adaptation of knowledge, 1995
14. Esmin Ahmed A. A., Lambert-Torres G., Zambroni de Souza A. C. A hybrid

particle swarm optimization applied to loss power minimization, 2005

15. Chuan L., Quanyuan F. The Standard Particle Swarm Optimization Algorithm Convergence Analysis and Parameter Selection, 2007
16. Scikit-learn documentation *https://scikit-learn.org/stable/*
17. Scikit-learn: Tuning the hyper-parameters of an estimator *https://scikit-learn.org/stable/modules/grid_search.html*
18. Bayesian Optimization, 2020 *https://github.com/fmfn/BayesianOptimization*
19. PySwarms documentation *https://pyswarms.readthedocs.io/en/latest/*
20. Bank Marketing UCI, 2018 *https://www.kaggle.com/c/bank-marketing-uci*
21. Pandas: get_dummies *https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html*
22. Pandas: Series.map *https://pandas.pydata.org/docs/reference/api/pandas.Series.map.html*

ДОДАТОК

В цьому додатку наведен приклад програмної реалізації налаштування гіперпараметрів для випадку побудови моделі k-найближчих сусідів.

```
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
print("Accuracy:", accuracy_score(y_valid, y_pred), "\nF1:", f1_score(y_valid, y_pred, average='weighted'))

kf = KFold(n_splits = 5, shuffle = True, random_state = 42)
knn = KNeighborsClassifier(n_jobs = -1, weights='distance')
knn_params = {"n_neighbors": np.arange(1, 51)}

start_time = time.time()
search_1 = GridSearchCV(knn, knn_params, cv = kf, scoring="f1")
search_1.fit(X_train, y_train)
print("GridSearchCV\nBest_params: ", search_1.best_params_, " Best_estimator: ", search_1.best_estimator_,
"Best_score: ", search_1.best_score_)
print("Time costs:", "--- %s seconds ---" % (time.time() - start_time))

start_time = time.time()
search_2 = HalvingGridSearchCV(knn, knn_params, cv = kf, scoring="f1")
search_2.fit(X_train, y_train)
print("\nHalvingGridSearchCV\nBest_params: ", search_2.best_params_, " Best_estimator: ", search_2.best_estimator_,
"Best_score: ", search_2.best_score_)
print("Time costs:", "--- %s seconds ---" % (time.time() - start_time))

start_time = time.time()
search_3 = RandomizedSearchCV(knn, knn_params, cv = kf, scoring="f1")
search_3.fit(X_train, y_train)
print("\nRandomizedSearchCV\nBest_params: ", search_3.best_params_, " Best_estimator: ", search_3.best_estimator_,
"Best_score: ", search_3.best_score_)
print("Time costs:", "--- %s seconds ---" % (time.time() - start_time))

start_time = time.time()
search_4 = HalvingRandomSearchCV(knn, knn_params, cv = kf, scoring="f1")
search_4.fit(X_train, y_train)
print("HalvingRandomSearchCV\nBest_params: ", search_4.best_params_, " Best_estimator: ", search_4.best_estimator_,
"Best_score: ", search_4.best_score_)
print("Time costs:", "--- %s seconds ---" % (time.time() - start_time))

best_search_1 = KNeighborsClassifier(n_jobs=-1, n_neighbors=29, weights='distance') # n_neighbors=13
y_pred_1 = best_search_1.fit(X_train, y_train).predict(X_valid)
print("GridSearchCV\naccuracy:", accuracy_score(y_valid, y_pred_1), "\nf1:", f1_score(y_valid, y_pred_1,
average='weighted'))

best_search_2 = KNeighborsClassifier(n_jobs=-1, n_neighbors=15, weights='distance')
y_pred_2 = best_search_2.fit(X_train, y_train).predict(X_valid)
print("\nHalvingGridSearchCV\naccuracy:", accuracy_score(y_valid, y_pred_2), "\nf1:", f1_score(y_valid, y_pred_2,
average='weighted'))

best_search_3 = KNeighborsClassifier(n_jobs=-1, n_neighbors=24, weights='distance')
y_pred_3 = best_search_3.fit(X_train, y_train).predict(X_valid)
print("\nRandomizedSearchCV\naccuracy:", accuracy_score(y_valid, y_pred_3), "\nf1:", f1_score(y_valid, y_pred_3,
average='weighted'))

best_search_4 = KNeighborsClassifier(n_neighbors=17)
y_pred_4 = best_search_4.fit(X_train, y_train).predict(X_valid)
```

```
print("\nHalvingRandomSearchCV\naccuracy:", accuracy_score(y_valid, y_pred_4), "\nf1:", f1_score(y_valid, y_pred_4,
average='weighted'))

# Define the black box function to optimize
def black_box_function(n_neighbors):
    model = KNeighborsClassifier(n_neighbors=int(n_neighbors), n_jobs=-1, weights='distance')
    model.fit(X_train, y_train)
    y_score = model.predict(X_valid)
    f = roc_auc_score(y_valid, y_score)
    return f

pbounds = {'n_neighbors':(1, 51)}

start_time = time.time()
optimizer = BayesianOptimization(f = black_box_function,
                                 pbounds = pbounds,
                                 verbose = 2,
                                 random_state = 0)
optimizer.maximize(init_points = 5, n_iter = 10)
print("Best result: {}; f(x) = {}.".format(optimizer.max["params"], optimizer.max["target"]))
print("Time costs:", "--- %s seconds ---" % (time.time() - start_time))

best_search_5 = KNeighborsClassifier(n_neighbors=43) # n_neighbors=17
y_pred_5 = best_search_5.fit(X_train, y_train).predict(X_valid)
print("BayesianOptimization \naccuracy:", accuracy_score(y_valid, y_pred_5), "\nf1:", f1_score(y_valid, y_pred_5,
average='weighted'))

def choice(x):
    return int(x)
def uniform(x):
    return x
def loguniform(x):
    return 10**x
def ErrorDistribs(y_true,y_pred):
    return abs(y_true-y_pred)/y_true
def tpr_weight_funtion(y_true,y_predict):
    d = pd.DataFrame()
    d['prob'] = list(y_predict)
    d['y'] = list(y_true)
    d = d.sort_values(['prob'], ascending=[0])
    y = d.y
    PosAll = pd.Series(y).value_counts()[1]
    NegAll = pd.Series(y).value_counts()[0]
    pCumsum = d['y'].cumsum()
    nCumsum = np.arange(len(y)) - pCumsum + 1
    pCumsumPer = pCumsum / PosAll
    nCumsumPer = nCumsum / NegAll
    TR1 = pCumsumPer[abs(nCumsumPer-0.001).idxmin()]
    TR2 = pCumsumPer[abs(nCumsumPer-0.005).idxmin()]
    TR3 = pCumsumPer[abs(nCumsumPer-0.01).idxmin()]
    return 0.4 * TR1 + 0.3 * TR2 + 0.3 * TR3
auc_scorer = make_scorer(tpr_weight_funtion)
class model():
    def __init__(self,n_particles=10,c1=0.5,c2=0.5,w=0.9,verbose=2,cv=5,scoring=auc_scorer):
        self.n_particles=n_particles
        self.c1=c1
        self.c2=c2
        self.w=w
        self.options={'c1':self.c1,'c2':self.c2,'w':self.w}
        self.verbose=verbose
        self.cv=cv
```

```
        self.scoring=auc_scorer
    def train(self, X_train, y_train, clf, param_distribs):
        start_time = time.time()
        self.X_train = X_train
        self.y_train = y_train
        self.clf = clf
        self.param_distribs=param_distribs
        self.dimensions = len(param_distribs)
        upper=np.zeros(self.dimensions)
        lower=np.zeros(self.dimensions)
        for count, (key, value) in enumerate(self.param_distribs.items()):
            lower[count] = value[1]
            upper[count] = value[2]
        bounds=(lower,upper)
        optimizer=ps.single.GlobalBestPSO(n_particles=self.n_particles,dimensions=self.dimensions,
            options=self.options, bounds=bounds)
        best_cost,best_pos=optimizer.optimize(self.search, iters = 25,verbose = self.verbose) #,print_step=10)
        self.best_params={}
        for count, (key, value) in enumerate(self.param_distribs.items()):
            if value[0].__name__=='choice':
                index=value[0](best_pos[count])
                self.best_params[key]=value[3][index]
            else:
                self.best_params[key]=value[0](best_pos[count])
        self.final_model=self.clf(**self.best_params)
        self.final_model.fit(self.X_train,self.y_train)
        print("Time costs:", "--- %s seconds ---" % (time.time() - start_time))
        joblib.dump(self.final_model,'{}.pkl'.format(self.clf.__name__))
    def search(self,param):
        score_array=np.zeros((self.n_particles,self.cv))
        fit_params={}
        for i in range(self.n_particles):
            for count, (key, value) in enumerate(self.param_distribs.items()):
                if value[0].__name__=='choice':
                    index=value[0](param[i,count])
                    fit_params[key] = value[3][index]
                else:
                    fit_params[key] = value[0](param[i,count])
            kf = KFold(n_splits = 5, shuffle = True, random_state = 42)
            score_array[i,:]=cross_val_score(self.clf(**fit_params), self.X_train, self.y_train,
                scoring = self.scoring, cv = kf)
        return 1-np.mean(score_array, axis=1)
if __name__=='__main__':
    model_list = ['KNeighborsClassifier']
    param_dict = {}
    param_dict['KNeighborsClassifier']={
        'n_neighbors': [uniform, 1, 10]
    }
    model_dict={}
    model_dict['KNeighborsClassifier'] = KNeighborsClassifier
    for model_name in model_list:
        clf_model = model()
        clf_model.train(X_train, y_train, model_dict[model_name], param_dict[model_name])
best_search_6 = KNeighborsClassifier(n_neighbors=17)
y_pred_6 = best_search_6.fit(X_train, y_train).predict(X_valid)
print("PSO \nAccuracy:", accuracy_score(y_valid, y_pred_6), "\nf1:", f1_score(y_valid, y_pred_6,
    average='weighted'))
```