

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
Навчально-науковий інститут Інформаційних технологій
Кафедра штучного інтелекту

Чичкарьов Є.А.

Методичні рекомендації
щодо проведення практичних занять з дисципліни «Аналіз і обробка великих
даних» для студентів спеціальності 122 Комп'ютерні науки всіх форм
навчання

Частина 1 – Форми представлення даних, типи і види даних. Пакети Pandas і
pySpark

Розглянуто на засіданні кафедри ШІ
Протокол № 1 від «29» серпня 2022 р

Київ
2022

УДК 004.658

Чичкарьов Є.А. Методичні рекомендації щодо проведення практичних занять з дисципліни «Основи Big data» для студентів спеціальності 122 Комп'ютерні науки всіх форм навчання. Частина 1 – Форми представлення даних, типи і види даних. Пакети Pandas і pySpark. Київ: ДУТ, 2022. 58 с.

Методичні рекомендації призначені для ознайомлення студентів спеціальності 122 Комп'ютерні науки всіх форм навчання з різними аспектами обробки великих даних з використанням Pandas і pySpark та набуття ними практичного досвіду вирішення різних задач обробки великих обсягів даних за допомогою Pandas і pySpark, використання методів первинної обробки і завантаження даних при виконанні практичних занять з дисципліни «Аналіз і обробка великих даних».

Рецензенти:

Рекомендовано
на засіданні кафедри штучного інтелекту,
протокол № 1 від 29 серпня 2022 р.

ЗМІСТ

ПРАКТИЧНА РОБОТА №1. Робота з Apache Spark і pySpark	4
1. Мета та зміст.....	4
2. Теоретичне обґрунтування.....	4
2.1 Основи Apache Spark та pySpark	4
2.2 Використання PySpark через Google Colab	5
2.3 Деякі механізми Spark	8
2.4 Приклад машинного навчання з PySpark	9
Крок 1) Завантаження даних за допомогою pySpark	10
2.4 Деякі методи pySpark.....	13
Крок 2) Попередня обробка даних	17
Крок 3) Створення конвеєра обробки даних.....	19
Крок 4) Побудова логістичного класифікатора	21
Крок 5) Навчання та оцінка моделі	23
Крок 6) Налаштування гіперпараметрів.....	26
3 Завдання до практичної роботи	28
4 Контрольні питання	29
ПРАКТИЧНА РОБОТА №2. Робота з даними за допомогою бібліотеки Pandas.....	30
1. Мета та зміст.....	30
2. Теоретичне обґрунтування.....	30
2.1 Встановлення та використання бібліотеки Pandas	30
2.2 Введення до об'єктів Pandas	31
2.3 Об'єкт Series Pandas.....	31
2.4 Об'єкт DataFrame	35
2.5 Побудова об'єктів DataFrame	37
2.6 Об'єкт Pandas Index	38
2.7 Індиксація та вибір даних.....	40
2.8 Операція з даними в Pandas	47
3 Завдання	51
4 Контрольні запитання.....	57
СПИСОК ЛІТЕРАТУРИ.....	58

ПРАКТИЧНА РОБОТА №1. Робота з Apache Spark і pySpark

1. Мета та зміст

Мета практичної роботи: знайомство з можливостями і особливостями Apache Spark.

2. Теоретичне обґрунтування

2.1 Основи Apache Spark та pySpark

Що таке Apache Spark?

Spark - це рішення для обробки великих даних, яке виявилось простіше і швидше, ніж Hadoop MapReduce . Spark це програмне забезпечення з відкритим вихідним кодом, розроблене лабораторією RAD Каліфорнійського університету в Берклі в 2009 році. З того часу, як він був випущений для широкої публіки в 2010 році, популярність Spark зростає і використовується в галузі в безпрецедентних масштабах.

В епоху Big Data практикам як ніколи потрібні швидкі та надійні інструменти для обробки потокової передачі даних. Більш ранні інструменти, такі як MapReduce були улюбленими, але працювали повільно. Щоб вирішити цю проблему, Spark пропонує рішення, одночасно швидке та універсальне. Основна відмінність між Spark і MapReduce полягає в тому, що Spark виконує обчислення у пам'яті, а потім на жорсткому диску. Це забезпечує високошвидкісний доступ та обробку даних, скорочуючи час із годин до хвилин.

Що таке PySpark ?

PySpark – це інструмент, створений спільнотою Apache Spark для використання Python із Spark. Він дозволяє працювати з RDD(Resilient Distributed Dataset) в Python. Він також пропонує PySpark Shell для зв'язку API Python із ядром Spark для запуску контексту Spark. Spark – це механізм імен для реалізації кластерних обчислень, а PySpark – це бібліотека Python для використання Spark.

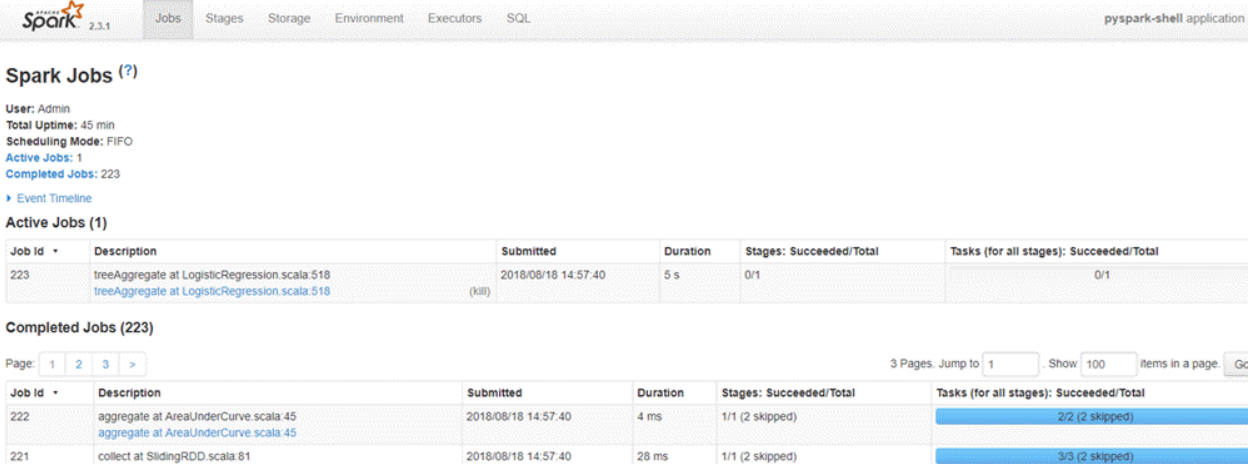
Як працює Spark?

Spark заснований на обчислювальному механізмі, що означає, що він відповідає за планування, розповсюдження та моніторинг програм. Кожне завдання виконується на різних робочих машинах, які називаються обчислювальним кластером. Обчислювальний кластер відноситься до поділу завдань. Одна машина виконує одне завдання, тоді як інші роблять внесок у кінцевий результат за допомогою іншого завдання. Зрештою, всі завдання об'єднуються для отримання результату. Адміністратор Spark надає 360-градусний огляд різних завдань Spark.

Spark призначений для роботи з

- Python
- Java
- Scala
- SQL

Важливою особливістю Spark є величезний обсяг вбудованої бібліотеки, у тому числі MLlib для машинного навчання. Spark також призначений для роботи з кластерами Hadoop і може читати файли широкого типу, включаючи дані Hive, CSV, JSON, Casandra та інші.



The screenshot shows the 'Spark Jobs' page in the Databricks interface. It includes a top navigation bar with tabs for Jobs, Stages, Storage, Environment, Executors, and SQL. Below the navigation bar, there's a summary section for 'Spark Jobs' with statistics like 'User: Admin', 'Total Uptime: 45 min', 'Scheduling Mode: FIFO', 'Active Jobs: 1', and 'Completed Jobs: 223'. The main content area is divided into two sections: 'Active Jobs (1)' and 'Completed Jobs (223)'. The 'Active Jobs' section shows a single job (ID 223) with a description 'treeAggregate at LogisticRegression scala 518', submitted on 2018/08/18 at 14:57:40, with a duration of 5 s and 0/1 stages. The 'Completed Jobs' section shows two jobs (IDs 222 and 221) with descriptions 'aggregate at AreaUnderCurve scala 45' and 'collect at SlidingRDD scala 81' respectively, both submitted on 2018/08/18 at 14:57:40. Job 222 has a duration of 4 ms and 1/1 (2 skipped) stages. Job 221 has a duration of 28 ms and 1/1 (2 skipped) stages. The 'Completed Jobs' section also includes a pagination bar showing '3 Pages', 'Jump to 1', 'Show 100', and 'Items in a page'.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
223	treeAggregate at LogisticRegression scala 518 treeAggregate at LogisticRegression scala 518 (kill)	2018/08/18 14:57:40	5 s	0/1	0/1

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
222	aggregate at AreaUnderCurve scala 45 aggregate at AreaUnderCurve scala 45	2018/08/18 14:57:40	4 ms	1/1 (2 skipped)	2/2 (2 skipped)
221	collect at SlidingRDD scala 81	2018/08/18 14:57:40	28 ms	1/1 (2 skipped)	3/3 (2 skipped)

Spark може працювати автономно, але найчастіше працює поверх інфраструктури кластерних обчислень, такої як Hadoop. Однак під час тестування та розробки спеціаліст за даними може ефективно використовувати Spark для своєї розробки комп'ютери чи ноутбуки без кластера. Однією з основних переваг Spark є створення архітектури, яка включає управління потоковою передачею даних, плавні запити даних, прогнозування машинного навчання і доступ в реальному часі до різних аналізів. Spark тісно співпрацює з мовою SQL, тобто із структурованими даними. Це дозволяє вимагати дані в режимі реального часу.

Маніпулювання даними має бути надійним і таким самим простим у використанні. Spark - відповідний інструмент завдяки своїй швидкості та багатому API.

2.2 Використання PySpark через Google Colab

Переваги налаштування PySpark через Google Colab

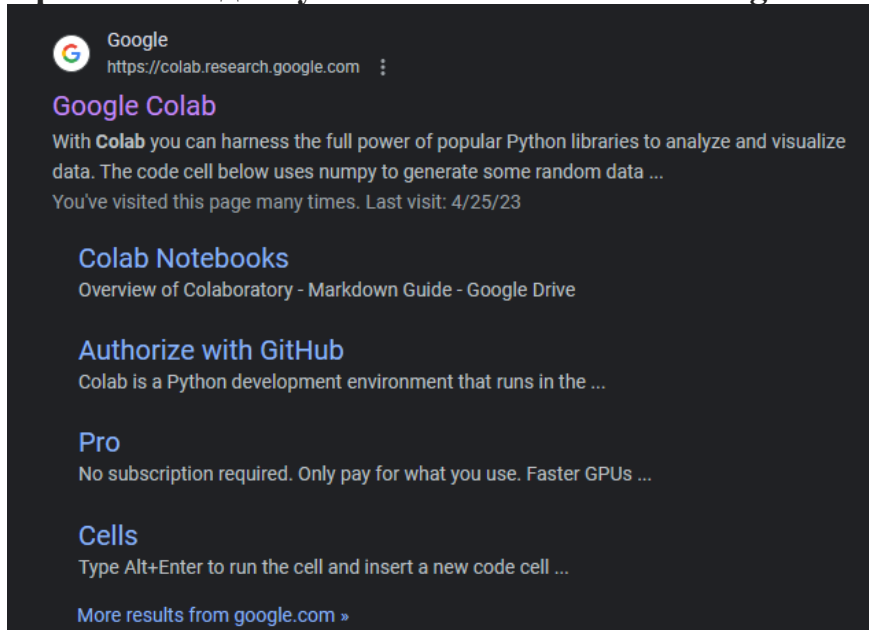
Використання PySpark в середовищі Google Colab має багато переваг. Ось кілька найпомітніших:

- **Безкоштовне використання:** Google Colab є безкоштовною службою, тому вам не потрібно турбуватися про будь-які попередні витрати.
- **Простота у використанні:** Google Colab — це веб-програма, тож ви можете використовувати її будь-де, де є підключення до Інтернету.
- **Доступ до графічних процесорів:** Google Colab пропонує безкоштовний доступ до графічних процесорів, що може значно пришвидшити роботу PySpark.
- **Підтримка спільноти:** Google Colab має велику та активну спільноту користувачів і розробників, які можуть допомогти вам у вирішенні будь-яких проблем, з якими ви зіткнетеся.

- **Можливість спільної роботи:** ви можете ділитися своїми блокнотами Colab з іншими, що полегшує співпрацю над проектами.
- **Можливість перенесення даних:** ви можете зберігати свої блокноти Colab і отримувати до них доступ із будь-якого пристрою з підключенням до Інтернету.
- **Розширюваність:** ви можете додавати власні бібліотеки та розширення до своїх блокнотів Colab, що надає вам ще більшої гнучкості.

Кроки для налаштування PySpark через Google Colab

Крок 1. Увійдіть у свій обліковий запис Google Colab



Крок 2. Виконайте наступний блок коду, щоб завантажити необхідні бібліотеки та почати сеанс Sparks

```
!apt-get update # Оновити репозиторій apt-get.
!apt-get install openjdk- 8 -jdk-headless -qq > /dev/null # Встановити Java.
!wget -q http://archive.apache.org/dist/spark/spark- 3.1 .1 /spark- 3.1 .1 -
bin -hadoop3 .2 .tgz # Завантажити Apache Sparks.
!tar xf spark- 3.1 .1 - bin -hadoop3 .2 .tgz # Розпакуйте файл tgz.
!pip install -q findspark # Встановити findspark. Додає PySpark до системного
шляху під час виконання.
# Встановити змінні середовища
import os
os.environ[ "JAVA_HOME" ] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ[ "SPARK_HOME" ] = "/content/spark-3.1.1- bin-hadoop3.2"
!ls
# Ініціалізувати findspark
import findspark
findspark.init()
# Створити сеанс PySpark
is pyspark.sql import SparkSession
spark = SparkSession.builder.master( "local[*]" ).getOrCreate()
spark
```

Після виконання ви отримаєте приблизно такий результат:

```
!apt-get update # Update apt-get repository.
!apt-get install openjdk-8-jdk-headless -qq > /dev/null # Install Java.
!wget -q http://archive.apache.org/dist/spark/spark-3.1.1-bin-hadoop3.2.tgz # Download Apache Sparks.
!tar xf spark-3.1.1-bin-hadoop3.2.tgz # Unzip the tgz file.
!pip install -q findspark # Install findspark. Adds PySpark to the System path during runtime.

# Set environment variables
import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
os.environ["SPARK_HOME"] = "/content/spark-3.1.1-bin-hadoop3.2"

!ls

# Initialize findspark
import findspark
findspark.init()

# Create a PySpark session
from pyspark.sql import SparkSession
spark = SparkSession.builder.master("local[*]").getOrCreate()
spark

Get:1 https://cloud.r-project.org/bin/linux/ubuntu focal-cran40/ InRelease [3,622 B]
Get:2 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64 InRelease [1,581 B]
Get:3 https://cloud.r-project.org/bin/linux/ubuntu focal-cran40/ Packages [76.4 kB]
Get:4 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu focal InRelease [18.1 kB]
Hit:5 http://archive.ubuntu.com/ubuntu focal InRelease
Get:6 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:7 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64 Packages [995 kB]
Get:8 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Hit:9 http://ppa.launchpad.net/cran/libgit2/ubuntu focal InRelease
Hit:10 http://ppa.launchpad.net/deadsnakes/ppa/ubuntu focal InRelease
Get:11 http://archive.ubuntu.com/ubuntu focal-backports InRelease [108 kB]
Get:12 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [2,632 kB]
Hit:13 http://ppa.launchpad.net/graphics-drivers/ppa/ubuntu focal InRelease
Get:14 http://archive.ubuntu.com/ubuntu focal-updates/universe amd64 Packages [1,331 kB]
Hit:15 http://ppa.launchpad.net/ubuntugis/ppa/ubuntu focal InRelease
Get:16 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu focal/main Sources [2,499 kB]
Get:17 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages [3,114 kB]
Get:18 http://security.ubuntu.com/ubuntu focal-security/universe amd64 Packages [1,035 kB]
Get:19 http://security.ubuntu.com/ubuntu focal-security/restricted amd64 Packages [2,145 kB]
Get:20 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu focal/main amd64 Packages [1,182 kB]
Get:21 http://archive.ubuntu.com/ubuntu focal-updates/restricted amd64 Packages [2,283 kB]
Fetched 17.7 MB in 7s (2,570 kB/s)
Reading package lists ... Done
sample_data spark-3.1.1-bin-hadoop3.2.tgz
spark-3.1.1-bin-hadoop3.2 spark-3.1.1-bin-hadoop3.2.tgz.1

SparkSession - in-memory
SparkContext
Spark UI
Version
v3.1.1
Master
local[*]
AppName
pyspark-shell
```

Крок 3. Завантажте зразок файлу CSV і прочитайте його у фрейм даних

The screenshot shows a PySpark Jupyter Notebook interface. On the left, a file explorer sidebar displays a directory structure. Under a folder named 'sample_data', the file 'Sample_Data.csv' is highlighted with a red box. Other files in the directory include 'README.md', 'anscombe.json', 'california_housing_test.csv', 'california_housing_train.csv', 'mnist_test.csv', 'mnist_train_small.csv', and a subdirectory 'spark-3.1.1-bin-hadoop3.2' containing 'spark-3.1.1-bin-hadoop3.2.tgz' and 'spark-3.1.1-bin-hadoop3.2.tgz.1'. The main area of the notebook shows the output of the previous code cell, which includes the same terminal output as the first image, with the 'SparkSession - in-memory' status and file download progress visible.

```
df = spark.read.format("csv").option("header", "true").load("sample_data/Sample_Data.csv")
df.show(5)
```

id	first_name	last_name	gender	city	job_title	latitute
1	Gert	Vernall	Female	Gouxaria	Clinical Specialist	39.4763332
2	Janella	Martyntsev	Female	Bugene	Human Resources A...	-1.5923262
3	Tadd	Starmont	Male	Carolina	Geological Engineer	-26.0730802
4	Greggory	Capron	Male	Sufālat Samā'il	Administrative Of...	23.303506
5	Morganne	Elsy	Female	Katumba	Engineer III	-9.2263237

only showing top 5 rows

2.3 Деяки механізми Spark

SparkContext

SparkContext – це внутрішній механізм, що забезпечує з'єднання із кластерами. Якщо ви хочете запустити операцію, вам знадобиться SparkContext. Створення SparkContext виконується за допомогою імпорту необхідного модуля і створення відповідного об'єкта:

```
import pyspark
from pyspark import SparkContext
sc = SparkContext()
```

Тепер, коли SparkContext готовий, ви можете створити колекцію даних під назвою RDD (Resilient Distributed Dataset). Обчислення в RDD автоматично розпаралелюються у кластері.

```
nums = sc.parallelize ([1,2,3,4])
```

Ви можете отримати доступ до першого рядка за допомогою take:

```
nums.take (1)
```

Результат

```
[1]
```

Ви можете застосувати перетворення даних за допомогою лямбда-функції. У наведеному нижче прикладі PySpark ви повертаєте квадрат чисел. Це трансформація за допомогою лямбда-функції і map:

```
squared = nums.map(lambda x: x*x).collect()
for num in squared:
    print('%i'%(num))
```

Результат:

```
1
4
9
16
```


SQLКонтекст

Більш зручний спосіб - використовувати DataFrame. SparkContext вже встановлено, ви можете використовувати його для створення dataframe . Вам також необхідно оголосити SQLContext

SQLContext дозволяє підключати двигун до різних джерел даних. Він використовується для запуску функцій Spark SQL.

```
from pyspark.sql import Row
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

Приклад створення структури даних Spark

Тепер давайте створимо список кортежів. Кожен кортеж міститиме імена людей та їх вік. Потрібно чотири кроки:

Крок 1) Створіть список кортежів з інформацією

```
[('John', 19), ('Smith', 29), ('Adam', 35), ('Henry', 50)]
```

Крок 2) Створення rdd

```
rdd = sc.parallelize(list_p)
```

Крок 3) Перетворення кортежів

```
rdd.map(lambda x: Row(name = x[0], age = int(x[1])))
```

Крок 4) Створіть контекст DataFrame

```
sqlContext.createDataFrame(ppl)
list_p = [('John', 19), ('Smith', 29), ('Adam', 35), ('Henry', 50)]
rdd = sc.parallelize(list_p)
ppl = rdd.map(lambda x: Row(name = x[0], age = int(x[1])))
DF_ppl = sqlContext.createDataFrame(ppl)
```

Якщо ви хочете отримати доступ до типу кожного елемента даних, можна використовувати printSchema().

```
DF_ppl.printSchema()
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

2.4 Приклад машинного навчання з PySpark

Тепер, коли у вас є коротке уявлення про Spark і SQLContext , ви готові створити свою першу програму машинного навчання.

Ось кроки зі створення програми машинного навчання за допомогою PySpark :

Крок 1) Завантаження даних за допомогою pySpark

Крок 2) Попередня обробка даних

Крок 3) Створення конвеєра обробки даних

Крок 4) Побудова логістичного класифікатора

Крок 5) Навчання та оцінка моделі

Крок 6) Налаштування гіперпараметрів

Spark призначено для обробки значного обсягу даних. Продуктивність Spark збільшується в порівнянні з іншими бібліотеками машинного навчання, коли набір даних збільшується.

Крок 1) Завантаження даних за допомогою pySpark

Перш за все, вам необхідно ініціалізувати SQLContext, який ще не запущений.

```
# from pyspark.sql import SQLContext
url = "https://raw.githubusercontent.com/guru99-edu/R-Programming/master/adult_data.csv"
from pyspark import SparkFiles
sc.addFile(url)
sqlContext = SQLContext(sc)
```

потім ви можете прочитати файл .csv з за допомогою sqlContext.read.csv. Передбачено використання inferSchema, для якої встановлено значення True, щоб Spark автоматично вгадував тип даних, за замовчуванням встановлено значення False.

```
df = sqlContext.read.csv(
    SparkFiles.get("adult_data.csv"), header = True,
    inferSchema = True)
```

Давайте подивимося на тип даних

```
df.printSchema()
```

Результат

root

```
-- age: integer (nullable = true) |
-- workclass : string (nullable = true) |
-- fnlwgt : integer (nullable = true) |
-- education: string (nullable = true) |
-- education_num : integer (nullable = true) |
-- marital: string (nullable = true)
-- occupation: string (nullable = true)
-- relationship: string (nullable = true) |
-- race: string (nullable = true)
-- sex: string (nullable = true)
-- capital_gain : integer (nullable = true)
-- capital_loss : integer (nullable = true) |
-- hours_week : integer (nullable = true)
```

```
|-- native_country : string (nullable = true)
|-- label : string(nullable = true )
```

Ви можете переглянути дані за допомогою show.

```
df.show(5, truncate = False)
```

```
+---+-----+-----+-----+-----+-----+---+
+-----+-----+-----+-----+-----+-----+---+
+-----+-----+-----+-----+-----+-----+---+
+-----+-----+
| age | workclass | fnlwgt | education | education_num |
marital | occupation | relationship | race | sex |
capital_gain|capital_loss|hours_week|native_country|lab
el |
+---+-----+-----+-----+-----+-----+---+
+-----+-----+-----+-----+-----+-----+---+
+-----+-----+-----+-----+-----+-----+---+
+-----+-----+
|39 |State-gov |77516 |Bachelors|13 |Never-married |Adm-
clerical | Not-in-family | White | Male | 2174 |0 |40
|United-States |<=50K|
|50 |Self-emp-not-inc|83311 |Bachelors|13 | Married-civ-
spouse | Exec- managerial | Husband | White|Male |0 |0
|13 |United-States |<=50K|
|38 |Private |215646|HS- grad | 9 |Divorced | Handlers-
cleaners|Not-in-family|White|Male |0 |0 |40 |United-
States |<=50K|
|53 |Private |234721|11th |7 | Married-civ-
spouse|Handlers-cleaners|Husband | Black| Male | 0 |0 |40
|United-States |<=50K|
|28 |Private |338409|Bachelors|13 | Married-civ-
spouse|Prof-specialty |Wife |Black|Female|0 |0 |40 |Cuba
|<=50K|
+---+-----+-----+-----+-----+-----+---+
+-----+-----+-----+-----+-----+-----+---+
+-----+-----+-----+-----+-----+-----+---+
+-----+-----+
only showing top 5 rows
```

Встановивши inferSchema=true, Spark автоматично переглядатиме файл csv і виводитиме схему кожного стовпця. Це вимагає додаткового проходження файлу, що призведе до повільнішого читання файлу з inferSchema, встановленим у значення true. Але натомість фрейм даних, швидше за все, матиме правильну схему, враховуючи його вхідні дані. Приклад:

```
df_string =
sqlContext.read.csv(SparkFiles.get("adult.csv"),
header=True, inferSchema = False)
df_string.printSchema()
```

Результат

```
root
|-- age: string (nullable = true)
|-- workclass: string (nullable = true) |
|-- fnlwt: string (nullable = true)
|-- education: string (nullable = true) |
|-- education_num: string (nullable = true)
|-- marital: string (nullable = true)
|-- occupation: string (nullable = true)
|-- relationship: string (nullable = true) |
|-- race: string (nullable = true)
|-- sex: string (nullable = true)
|-- capital_gain: string (nullable = true)
|-- capital_loss: string (nullable = true)
|-- hours_week: string (nullable = true)
|-- native_country: string (nullable = true)
|-- label: string(nullable = true )
```

Щоб перетворити безперервну змінну на потрібний формат, можна використовувати перетворення стовпців. Ви можете використовувати `withColumn`, щоб вказати Spark, в якому стовпці виконувати перетворення.

```
# Import all from `sql.types`
from pyspark.sql.types import *

# Write a custom function to convert the type data of
DataFrame columns
def convertColumn(df, names, newType):
    for name in names:
        df = df.withColumn(name, df[name].cast(newType))
    return df
# List of continuous features
CONTI_FEATURES = ['age', 'fnlwt', 'capital_gain',
'education_num', 'capital_loss', 'hours_week']
# Convert the type
df_string = convertColumn(df_string, CONTI_FEATURES,
FloatType())
# Check the dataset
df_string.printSchema()
root
```

```

|-- age: float (nullable = true)
|-- workclass : string (nullable = true) |
|-- fnlwgt : float (nullable = true)
|-- education: string (nullable = true) |
|-- education_num : float (nullable = true)
|-- marital: string (nullable = true)
|-- occupation: string (nullable = true)
|-- relationship: string (nullable = true) |
|-- race: string (nullable = true)
|-- sex: string (nullable = true)
|-- capital_gain : float (nullable = true)
|-- capital_loss : float (nullable = true)
|-- hours_week : float (nullable = true)
|-- native_country : string (nullable = true)
|-- label: string (nullable = true)

from pyspark.ml.feature import StringIndexer
#stringIndexer = StringIndexer(inputCol = "label",
outputCol = "newlabel")
#model = stringIndexer.fit(df)
#df = model.transform(df)
df.printSchema()

```

2.4 Деякі методи pySpark

Select columns

Ви можете вибрати та відобразити рядки за допомогою вибору та назв об'єктів. Нижче вибрано age і fnlwgt.

```
df.select('age', 'fnlwgt').show(5)
```

```

+---+-----+
| age | fnlwgt |
+---+-----+
| 39  | 77516  |
| 50  | 83311  |
| 38  | 215646 |
| 53  | 234721 |
| 28  | 338409 |
+---+-----+

```

only showing top 5 rows

Count by group

Якщо ви хочете підрахувати кількість входжень по групах, ви можете створити ланцюжок:

- groupBy()
- count()

разом. У наведеному нижче прикладі PySpark ви підраховуєте кількість рядків за рівнем освіти.

```
df.groupBy("education").count().sort("count", ascending=True).show()
```

```
+-----+-----+
| education|count |
+-----+-----+
|  Preschool|  51|
|  1st-4th  | 168 |
|  5th-6th  | 333 |
|  Doctorate | 413 |
|  12th     | 433 |
|  9th      | 514 |
|  Prof-school | 576 |
|  7th-8th  | 646 |
|  10th     | 933 |
|  Assoc - acdm | 1067 |
|  11th     | 1175 |
|  Assoc- voc | 1382 |
|  Masters  | 1723 |
|  Bachelors | 5355 |
|  Some-college| 7291 |
|  HS-grad  | 10501 |
+-----+-----+
```

Describe the data

Щоб отримати зведену статистику даних, можна використати метод `describe()`. Він вирахає:

- `count`
- `mean`
- `stddev` (стандартне відхилення)
- `min`
- `max`

```
df.describe().show()
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+
|summary|  age  |  workclass  |  fnlwgt  |  освіта|
education_num | marital| occupation|relationship | race|
sex|      capital_gain      |      capital_loss      |
hours_week|native_country|label |
+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+
```

```

-----+-----+-----+-----+-----+-----+-----+
-----+-----+
| count | 32561 | 32561 | 32561 | 32561 | 32561 | 32561 |
| 32561 | 32561 | 32561 | 32561 | 32561 | 32561 | 32561 |
| 32561 | 32561 |
| mean | 38.58164675532078 | null | 189778.36651208502 |
null | 10.0806793403151 | null | null | null | null |
null | 1077.6488437087312 | 87.303829734959 |
40.437455852092995 | null | null |
| stddev | 13.640432553581356 | null | 105549.97769702227 |
| null | 2.572720332067397 | null | null | null | null |
null |
7385.292084840354 | 402.960218649002 | 12.347428681731838 |
null | null |
| min | 17 | ? | 12285 | 10th | 1 | Divorced | ? | Husband | Amer-
Indian-Eskimo | Female | 0 | 0 | 1 | ? | <=50K |
| max | 90 | Without-pay | 1484705 | Some-college | 16 |
Widowed | Transport-moving | Wife | White | Male | 99999
| 4356 | 99 | Югославія | >50K |
+-----+-----+-----+-----+-----+-----+-----+
----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+
-----+-----+

```

Якщо вам потрібна зведена статистика лише для одного стовпця, додайте ім'я стовпця всередині методу опису().

```
df.describe('capital_gain').show()
```

```

+-----+-----+
|summary| capital_gain |
+-----+-----+
| count | 32561 |
| mean | 1077.6488437087312 |
| stddev | 7385.292084840354 |
| min | 0 |
| max | 99999 |
+-----+-----+

```

Crosstab computation

У деяких випадках може бути цікаво побачити описову статистику між двома парними стовпцями. Наприклад, ви можете порахувати кількість людей з доходом нижчим або вищим за 50 тисяч за рівнем освіти. Ця операція називається перехресною таблицею.

```
df.crosstab ('age', 'label').sort("age_label").show()
```

```
+-----+-----+-----+
```

	age_label	<=50K	>50K
17	395	0	
18	550	0	
19	710	2	
20	753	0	
21	717	3	
22	752	13	
23	865	12	
24	767	31	
25	788	53	
26	722	63	
27	754	81	
28	748	119	
29	679	134	
30	690	171	
31	705	183	
32	639	189	
33	684	191	
34	643	243	
35	659	217	
36	635	263	

only showing top 20 rows

Ви можете бачити, що жодна людина не має доходу вище 50 тисяч, коли вона молода.

Drop column

Існує два інтуїтивно зрозумілі API для видалення стовпців:

- `drop()`: видалити стовпець
- `dropna()`: видалити NA

Нижче ви опускаєте стовпець `education_num`

```
df.drop('education_num').columns
```

```
['age',
 'workclass',
 'fnlwgt',
 'education',
 'marital',
 'occupation',
 'relationship',
 'race',
 'sex',
 'capital_gain ',
 'capital_loss ']
```



```
'hours _week',
'native _country',
'label']
```

Filter data

Ви можете використовувати `filter()` для описової статистики до підмножини даних. Наприклад, ви можете порахувати кількість людей, старших за 40 років.

```
df.filter(df.age>40).count()
13443
```

Descriptive statistics by group

Нарешті, ви можете групувати дані групи і виконувати статистичні операції, такі як середнє значення.

```
df.groupby('marital').agg({'capital_gain':
'mean'}).show()
+-----+-----+
| marital| avg(capital_gain ) |
+-----+-----+
| Separated| 535.5687804878049 |
| Never-married|376.58831788823363 |
|Married-spouse-ab. ..| 653.9832535885167 |
| Divorced| 728.4148098131893 |
| Widowed | 571.0715005035247 |
| Married-AF-spouse| 432.6521739130435 |
| Married -civ-spouse | 1764.8595085470085 |
+-----+-----+
```

Крок 2) Попередня обробка даних

Попередня обробка даних - найважливіший крок у машинному навчанні. Вона передбачає видалення помилкових або непотрібних даних, або створення додаткових.

Щоб додати новий стовпчик, вам потрібно:

1. Виберіть стовпець
2. Застосуйте перетворення і додайте його до DataFrame.

```
from pyspark.sql.functions import *
```

```
# 1 Select the column
age_square = df.select(col("age")**2)
```

```
# 2 Apply the transformation and add it to the DataFrame
df = df.withColumn("age_square", col("age")**2)
```

```
df.printSchema()
root
|-- age: integer (nullable = true)
```

```
|-- workclass: string (nullable = true)
|-- fnlwgt: integer (nullable = true)
|-- education: string (nullable = true)
|-- education_num: integer (nullable = true)
|-- marital: string (nullable = true)
|-- occupation: string (nullable = true)
|-- relationship: string (nullable = true)
|-- race: string (nullable = true)
|-- sex: string (nullable = true)
|-- capital_gain: integer (nullable = true)
|-- capital_loss: integer (nullable = true)
|-- hours_week: integer (nullable = true)
|-- native_country: string (nullable = true)
|-- label: string (nullable = true)
|-- age_square: double (nullable = true)
```

Ви можете бачити, що `age_square` був успішно доданий у кадр даних. Ви можете змінити порядок змінних за допомогою `select`. Нижче ви вказуєте `age_square` одразу після `age`.

```
COLUMNS = ['age', 'age_square', 'workclass', 'fnlwgt',
            'education', 'education_num', 'marital',
            'occupation', 'relationship', 'race', 'sex',
            'capital_gain', 'capital_loss',
            'hours_week', 'native_country', 'label']
df = df.select(COLUMNS)
df.first()
```

```
Row(age=39, age_square=1521.0, workclass='State-gov',
fnlwgt=77516, education='Bachelors', education_num=13,
marital='Never-married', occupation='Adm-clerical',
relationship='Not-in-family', race='White', sex='Male',
capital_gain=2174, capital_loss=0, hours_week=40,
native_country='United-States', label='<=50K')
```

Коли група всередині об'єкта має лише одне спостереження, це не приносить жодної інформації модель. Навпаки, це може спричинити помилку під час перехресної перевірки.

Давайте перевіримо походження домогосподарства

```
df.filter(df.native_country == 'Holand-
Netherlands').count()
df.groupby('native_country').agg({'native_country':
'count'}).sort(asc("count(native_country)")).show()
```

```
+-----+-----+
```

native_country	count(native_country)
Holland-Netherlands	1
Scotland	12
Hungary	13
Honduras	13
Outlying-US (Guam-...	14
Yugoslavia	16
Thailand	18
Laos	18
Cambodia	19
Trinidad&Tobago	19
Hong	20
Ireland	24
Ecuador	28
Greece	29
France	29
Peru	31
Nicaragua	34
Portugal	37
Iran	43
Haiti	44

only showing top 20 rows

В об'єкті Native_country вказано лише одне домогосподарство з Нідерландів. Його можна видалити наступною командою:

```
df_remove = df.filter(df.native_country != 'Holland-Netherlands')
```

Крок 3) Створення конвеєра обробки даних

Подібно до scikit-learn, PySpark має конвеєрний API.

Конвеєр дуже зручний для підтримки структури даних. Ви надсилаєте дані в конвеєр. У середині конвеєра виконуються різні операції, вихідні дані використовуються живлення алгоритму.

Розглянемо приклад побудови конвеєру для перетворення об'єктів та додавання їх до остаточного набору даних. У конвеєрі буде чотири операції, але ви можете додавати стільки операцій скільки захочете. Перелік дій:

1. Кодування категоріальні дані
2. Індекссування функцію мітки
3. Додавання безперервної змінної
4. Побудова pipeline.

Кожен крок зберігається у списку під назвою «етапи». Цей список повідомить VectorAssembler, яку операцію слід виконати всередині конвеєра.

1. Кодування категоріальних даних

Нижче наведено приклад коду:

```

from pyspark.ml import Pipeline
from pyspark.ml.feature import OneHotEncoderEstimator
CATE_FEATURES = ['workclass', 'education', 'marital',
'occupation', 'relationship', 'race', 'sex',
'native_country']
stages = []
#stages in our Pipeline для категоричних колів в
#CATE_FEATURES:
stringIndexer = StringIndexer(inputCol = categoricalCol,
outputCol = categoricalCol + "Index")
encoder = OneHotEncoderEstimator(inputCols =
[stringIndexer.getOutputCol()],
outputCols=[categoricalCol + "classVec"])
stages += [stringIndexer, encoder]

```

2. Індекссування міток

Spark, як і багато інших бібліотек, не приймає рядкові значення для мітки. Ви конвертуєте функцію мітки за допомогою StringIndexer та додаєте її на етапи списку.

```

# Convert label in label indices using the StringIndexer
label_stringIdx = StringIndexer(inputCol = "label",
outputCol = "newlabel")
stages += [label_stringIdx]

```

3. Додавання безперервної змінної

InputCols VectorAssembler являє собою список стовпців. Ви можете створити новий список, що містить нові стовпці. Код нижче заповнює список закодованими категоріальними ознаками та безперервними ознаками.

```

assemblerInputs = [c + " classVec " for c in
CATE_FEATURES] + CONTI_FEATURES

```

4. Побудова pipeline з використанням VectorAssembler.

Нарешті, ви проходите всі кроки у VectorAssembler.

```

assembler = VectorAssembler(inputCols = assemblerInputs,
outputCol = "features")
stages += [assembler]

```

Тепер, коли всі кроки готові, ви надсилаєте дані в конвеєр.

```
# Create a Pipeline.
```

```

pipeline = Pipeline (stages = stages)
pipelineModel = pipeline.fit(df_remove )
model = pipelineModel.transform(df_remove )

```

Якщо ви перевірите новий набір даних, ви побачите, що він містить усі об'єкти, перетворені та не перетворені. Вас цікавлять тільки нова мітка та стовпчики. Об'єкти включають всі перетворені об'єкти і безперервні змінні.

```
model.take(1)
```

```
[Row(age=39, age_square =1521.0, workclass ='State-gov',
fnlwgt =77516, education='Bachelors', education_num =13,
marital='Never-married', occupation='Adm-clerical',
relationship ='Not-in-family', race='White', sex='Male',
capital_gain =2174, capital_loss =0, hours_week =40,
native_country ='United-States', label='<=50K',
workclassIndex =4.0, workclassclassVec = SparseVector
(8, {4: 1.0}), educationIndex =2.0, educationclassVec =
SparseVector (15, {2: 1.0}), maritalIndex =1.0,
maritalclassVec = SparseVector (6, {1: 1.0}),
occupationIndex =3.0, occupationclassVec = SparseVector
(14, {3: 1.0}), relationshipIndex =1.0,
relationshipclassVec = SparseVector (5, {1: 1.0}),
raceIndex =0.0, raceclassVec = SparseVector (4, {0: 1}).
, sexIndex =0.0, sexclassVec = SparseVector (1, {0:
1.0}), native_countryIndex =0.0, native_countryclassVec
= SparseVector (40, {0: 1.0}), newlabel =0.0, features=
SparseVector (99, {4: 10: 1.0, 24: 1.0, 32: 1.0, 44: 1.0,
48: 1.0, 52: 1.0, 53: 1.0, 93: 39.0, 94: 77516.0, 95:
2174.0. ]
```

Крок 4) Побудова логістичного класифікатора

Щоб прискорити обчислення, треба конвертувати модель DataFrame. Для цього необхідно вибрати нову мітку та об'єкти з моделі за допомогою map і лямбда-функції.

```
from pyspark.ml.linalg import DenseVector
input_data = model.rdd.map(lambda x: (x[" newlabel "],
DenseVector (x["features"])))
```

Створимо тренувальні дані у вигляді DataFrame за допомогою sqlContext

```
df_train = sqlContext.createDataFrame(input_data,
["label", "features"])
```

Перевіримо другий рядок

```
df_train.show (2)
+-----+-----+
|label| features|
+-----+-----+
| 0.0 | [0.0,0.0,0.0,0.0, ... |
| 0.0 | [0.0,1.0,0.0,0.0, ... |
+-----+-----+
only showing top 2 rows
```

Створення наборів train/test

Розділимо набір даних 80/20 за допомогою randomSplit.

```
# Split the data in train and test sets
train_data, test_data =
df_train.randomSplit([.8, .2], seed=1234)
```

Нижче наведено приклад розрахунку, скільки людей із доходом нижче/вище 50 тис. є в навчальній та в тестовій вибірці.

```
train_data.groupby('label').agg({'label':
'count'}).show()
+-----+-----+
| label | count (label) |
+-----+-----+
| 0.0   | 19698         |
| 1.0   | 6263          |
+-----+-----+
test_data.groupby('label').agg({'label':
'count'}).show()
+-----+-----+
| label | count (label) |
+-----+-----+
| 0.0   | 5021          |
| 1.0   | 1578          |
+-----+-----+
```

Побудова логістичної регресії

Розглянемо створення класифікатору. PySpark має API під назвою LogisticRegression для виконання логістичної регресії.

Розглянемо приклад.

Ініціалізуємо lr, вказуючи стовпець мітки та стовпці функцій. Встановлюємо максимум 10 ітерацій та додаємо параметр регуляризації зі значенням 0.3. Далі буде використано перехресна перевірка з сіткою параметрів для налаштування моделі.

```
# Import `LinearRegression`
from pyspark.ml.classification import LogisticRegression

# Initialize `lr`
lr = LogisticRegression(labelCol = "label",
                        featuresCol = "features",
                        maxIter = 10,
                        regParam = 0.3)

# Fit the data to the model
linearModel = lr.fit(train_data)
```

Ви можете побачити коефіцієнти регресії

```
# Print the coefficients and intercept for logistic
# regression
print("Coefficients: " + str(linearModel.coefficients))
print("Intercept:" + str( linearModel.intercept))
Coefficients:      [-0.0678914665262,-0.153425526813,-
0.0706009536407,-0.164057586562,-
0.120655298528,0.162922330862,0.149176870438,-
0.626836362611,-0.193483661541,-
0.0782269980838,0.222667203836,0.399571096381,-
0.0222024341804,-0.311925857859,-0.0434497788688,-
0.306007744328,-0.41318209688,          0.547937504247,-
0.395837350854,-0.23166535958,0.618743906733,-
0.344088614546,-0.385266881360,6038      ,-0.201335923138,-
0.232878560088,-0.13349278865,-
0.119760542498,0.17500602491,-0.04809681011984422442
,0.0524163478063,-0.300952624551,-0.22046421474      ,-
0.16557996579,-0.114676231939,-0.311966431453,-
0.344226119233,0.105530129507,0.15224304781432432      3,-
0.199951374076,-0.30329422583,-
0.231087515178,0.418918551,-0.0565930184279,-
0.1778180730482-6.23. 2,0.168491215697,-0.12181255723, -
0.385648075442,-0.202101794517,0.0469791640782,-
0.00842850210625,-0.00373211448629,-0.2592961433
4409756,-0.11048086026,0.0280647963877,-
0.204187030092,-0.414392623536,-
0.252806580669,0.1433263465          27370849,-
0.301949286524,0.0878249035894,          -0.210951740965,-
0.621417928742,-0.099445190784,-0.232671473401,-
0.1077745606,-0.360429419703,306,209          9,-
0.395186242741,0.0826401853838,-
0.280251589972,0.187313505214,-0.20295228799,-
0.4311770646129      4,-0.319314858424,0.0028450133235,      -
0.651220387649,-0.327918792207,-
0.143659581445,0.00691075160413,8.38517628783e-
08,2.188567173762200 1075966823,0.00893832698698]
Intercept : -1.9884177974805692
```

Крок 5) Навчання та оцінка моделі

Щоб створити прогноз для вашого набору тестів, можна використовувати LinearModel з Transform() для test_data.

```
# Make predictions on test data з допомогою методу
# transform().
predictions = linearModel.transform(test_data)
```

Ви можете роздрукувати елементи у прогнозах

```
predictions.printSchema ()
root
|-- label: double (nullable = true)
|-- features: vector (nullable = true)
|-- rawPrediction : vector (nullable = true) |
|-- probability: vector (nullable = true) |
|-- prediction: double (nullable = false)
```

Виведемо мітку, прогноз та ймовірність

```
selected = predictions.select("label", "prediction",
"probability")
```

```
selected.show (20)
```

```
+-----+-----+-----+
| label | prediction | probability|
+-----+-----+-----+
| 0.0   | 0.0|[0.91560704124179...|
| 0.0   | 0.0|[0.92812140213994...|
| 0.0   | 0.0|[0.92161406774159...|
| 0.0   | 0.0|[0.96222760777142...|
| 0.0   | 0.0|[0.66363283056957...|
| 0.0   | 0.0|[0.65571324475477...|
| 0.0   | 0.0|[0.73053376932829...|
| 0.0   | 1.0|[0.31265053873570...|
| 0.0   | 0.0|[0.80005907577390...|
| 0.0   | 0.0|[0.76482251301640...|
| 0.0   | 0.0|[0.84447301189069...|
| 0.0   | 0.0|[0.75691912026619...|
| 0.0   | 0.0|[0.60902504096722...|
| 0.0   | 0.0|[0.80799228385509...|
| 0.0   | 0.0|[0.87704364852567...|
| 0.0   | 0.0|[0.83817652582377...|
| 0.0   | 0.0|[0.79655423248500...|
| 0.0   | 0.0|[0.82712311232246...|
| 0.0   | 0.0|[0.81372823882016...|
| 0.0   | 0.0|[0.59687710752201...|
```

```
+-----+-----+-----+
```

only showing top 20 rows

Оцінка моделі

Вам потрібно подивитися на показник точності, щоб побачити наскільки добре (або погано) працює модель. В даний час у Spark немає API для обчислення показника точності. Значенням за умовчанням є ROC, крива

робочої характеристики приймача. Це інші показники, які враховують рівень хибно-позитивних результатів.

Створимо DataFrame з позначкою та прогнозом.

```
cm = predictions.select ("label", "prediction")
```

Ви можете перевірити номер класу в позначці та прогнозі.

```
cm.groupby('label').agg({'label': 'count'}).show()
```

```
+-----+-----+
| label | count (label) |
+-----+-----+
| 0.0   | 5021          |
| 1.0   | 1578          |
+-----+-----+
```

```
cm.groupby('prediction').agg({'prediction':
'count'}).show()
```

```
+-----+-----+
| prediction | count(prediction) |
+-----+-----+
| 0.0        | 5982              |
| 1.0        | 617               |
+-----+-----+
```

Наприклад, у тестовому наборі 1578 домогосподарств із доходом вище 50 тис. та 5021 домогосподарство нижче. Проте класифікатор передбачив 617 домогосподарств із доходом понад 50 тисяч.

Ви можете оцінити точність, обчисливши кількість випадків, коли мітки правильно класифіковані за кількістю рядків.

```
cm.filter(cm.label== cm.prediction ).count()/cm.count ()
0.8237611759357478
```

Ви можете об'єднати все разом та написати функцію для обчислення точності.

```
def accuracy_m (model):
    predictions = model.transform(test_data)
    cm = predictions.select("label", "prediction")
    acc = cm.filter(cm.label==
cm.prediction).count()/cm.count()
    print( "Model accuracy: %.3f%%" % (acc * 100))
    accuracy_m(model = linearModel)
```

Model accuracy: 82.376%

ROC- метрики

Модуль BinaryClassificationEvaluator включає заходи ROC. Крива робочих характеристик приймача - ще один поширений інструмент, який використовується при двійковій класифікації. Вона дуже схожа на криву точності/відкликання, але замість побудови графіка залежності точності від

повноти крива ROC показує співвідношення справжнього позитивного результату (тобто відгуку) і рівня хибнопозитивного результату. Рівень хибно-позитивних результатів - це співвідношення негативних випадків, які помилково класифікуються як позитивні. Він дорівнює одиниці мінус істинно негативна ставка. Істинно негативний показник також називається специфічністю. Отже, крива ROC відображає чутливість(нагадуваність) залежно від 1 – специфічність.

```
### Use ROC
from pyspark.ml.evaluation import
BinaryClassificationEvaluator

# Evaluate model
evaluator =
BinaryClassificationEvaluator(rawPredictionCol = "
rawPrediction ")
print( evaluator.evaluate (predictions))
print( evaluator.getMetricName ())
0.8940481662695192 областьПід ROC
print( evaluator.evaluate (predictions))
0.8940481662695192
```

Крок 6) Налаштування гіперпараметрів

Щоб скоротити час обчислень, можна налаштувати параметр регуляризації лише з двома значеннями.

```
from pyspark.ml.tuning import ParamGridBuilder,
CrossValidator
```

```
# Create ParamGrid for Cross Validation
paramGrid =(ParamGridBuilder()
            .addGrid(lr.regParam, [0.01, 0.5])
            .build())
```

Оцінимо модель, використовуючи метод перехресної перевірки із 5 згинами. Навчання займає декілька хвилин.

```
from time import *
start_time = time()

# Create 5-fold CrossValidator
cv = CrossValidator(estimator = lr ,
                    estimatorParamMaps = paramGrid,
                    evaluator=evaluator, numFolds =5)
```

```
# Run cross validations
cvModel = cv.fit(train_data )
# likely take a fair amount of time
end_time = time( )
elapsed_time = end_time - start_time
print( "Time to train model: %.3f seconds" % elapsed_time
)
```

Час навчання моделі: 978.807 секунд.

Найкращий гіперпараметр регуляризації - 0.01 з точністю 85.316 відсотка.

```
accuracy_ m(model = cvModel)
```

```
Model accuracy: 85.316%
```

Ви можете вилучити рекомендований параметр, зв'язавши cvModel.bestModel з ExtractParamMap().

```
bestModel = cvModel.bestModel
```

```
bestModel.extractParamMap ( )
```

```
{Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' aggregationDepth ', doc='suggested depth for
treeAggregate (>= 2)'): 2,
  Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' elasticNetParam ', doc='The ElasticNet mixing
parameter, in range [0, 1]. Для alpha = 0, penalty is an
L2 penalty. Для alpha = L1 penalty'): 0.0,
  Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name='family', doc='The name of family which is
description of label distribution be used in the model.
Supported options: auto, binomial, multinomial.'): ',
  Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' featuresCol ', doc='features column name'):
'features',
  Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' fitIntercept ', doc='whether to fit intercept
term'): True,
  Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' labelCol ', doc='label column name'): 'label',
  Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' maxIter ', doc='maximum number of iterations (>=
0)'): 10,
```

```

    Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' predictionCol ', doc='prediction column name'):
'prediction',
    Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' probabilityCol ', doc='Column name for predicted
class conditional probabilities. Опис: Немає всіх
моделей, які можуть бути впевнені, що існують, що
існують, впевнені, що існують, якби не було ніяких
проблем! probabilities') : 'probability',
    Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' rawPredictionCol ', doc='raw prediction (aka
confidence) column name'): ' rawPrediction ',
    Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' regParam ', doc='regularization parameter (>=
0)'): 0.01,
    Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name='standardization', doc='whether to standardize the
training features before fitting the model'): True,
    Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name='threshold', doc='threshold in binary
classification prediction, in range [0, 1]'): 0.5,
    Param(
parent='LogisticRegression_4d8f8ce4d6a02d8c29a0',
name=' tol ', doc='the convergence tolerance for
iterative algorithms (>= 0)'): 1e-06}

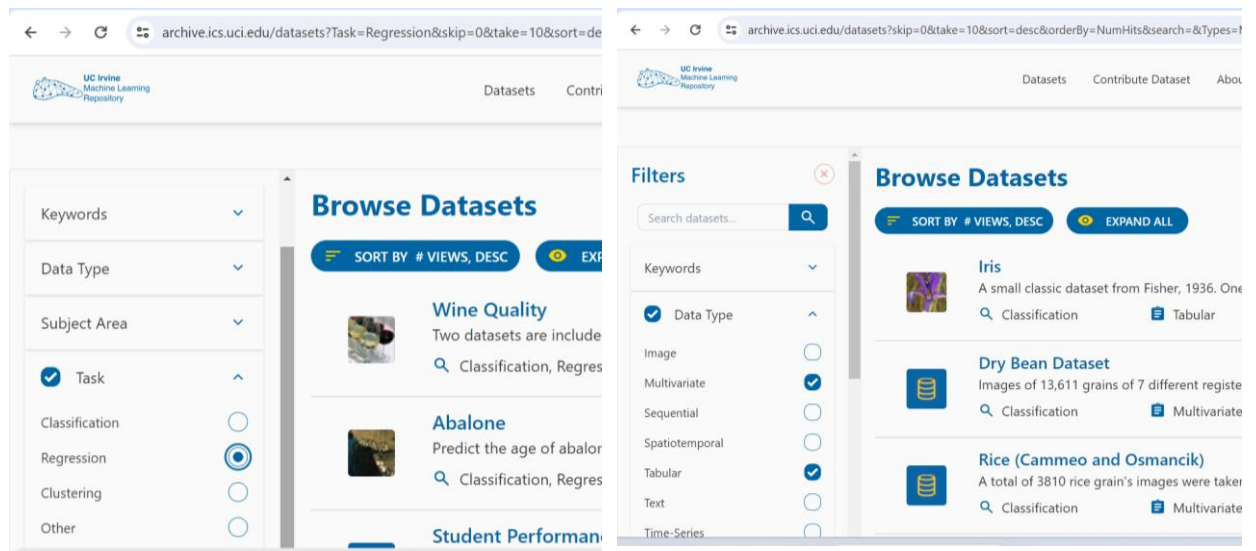
```

Таким чином, Spark - це фундаментальний інструмент для фахівця за даними. Це дозволяє практикуючому спеціалісту підключати додаток до різних джерел даних, без перешкод виконувати аналіз даних або додавати прогнозну модель.

3 Завдання до практичної роботи

Побудувати модель логістичної регресії за даними із зовнішнього джерела, використовуючи pySpark.

Дані завантажити з сайту <https://archive.ics.uci.edu/datasets>. Обрати дані для вирішення задач класифікації (встановити фільтр, див. рисунок нижче).



4 Контрольні питання

1. Що таке Spark і для яких цілей він використовується?
2. Що таке pySpark і для яких цілей він використовується?
3. Яким чином завантажити дані у пам'ять за допомогою Spark?
4. Які можливості Spark для конвеєрної обробки даних?

ПРАКТИЧНА РОБОТА №2. Робота з даними за допомогою бібліотеки Pandas

1. Мета та зміст

Мета практичної роботи: отримати базові знання та навички з аналізу та обробки даних за допомогою бібліотеки Pandas.

2. Теоретичне обґрунтування

Pandas – пакет, побудований поверх NumPy, він забезпечує ефективну реалізацію розвиненої структури даних DataFrame. DataFrame – це, по суті, багатовимірні масиви з прикріпленими мітками рядків і стовпців, і часто з неоднорідними типами та/або відсутніми даними.

Pandas – це швидкий, потужний, гнучкий і простий у використанні інструмент аналізу та обробки даних з відкритим кодом, побудований за допомогою мови програмування Python. Pandas є корисним інструментом при роботі з табличними даними, що зберігаються в електронних таблицях або базах даних. Ця бібліотека допомагає досліджувати, очищати та обробляти дані. В Pandas таблиці даних називаються DataFrame. Pandas підтримує інтеграцію з багатьма форматами даних, такими як csv, excel, sql, json тощо.

Окрім зручного інтерфейсу зберігання маркованих даних, Pandas реалізує ряд потужних операцій з даними, знайомих користувачам як фреймворки баз даних, так і програм електронних таблиць. Як ми бачили, структура даних ndarray NumPy забезпечує важливі особливості для типу чистих, добре впорядкованих даних, що зазвичай спостерігаються в числових обчислювальних завданнях. Незважаючи на те, що ndarray слугує цій меті дуже добре, його обмеження стають зрозумілими, коли нам потрібна більша гнучкість (наприклад, прикріплення міток до даних, робота з відсутніми даними тощо) та при спробах операцій, які погано відображаються на поелементному бродкастингу (наприклад, групування, зведені таблиці тощо), кожна з яких є важливою частиною аналізу менш структурованих даних, доступних у багатьох формах у навколишньому світі. Pandas, зокрема його об'єкти Series і DataFrame, базується на структурі масиву NumPy і забезпечує ефективний доступ до таких видів завдань «зміни даних», які займають більшу частину часу дата аналітика. У цій частині курсу ми зупинимось на механіці ефективного використання Series, DataFrame та суміжних структур. Ми будемо використовувати приклади, отримані з реальних наборів даних, де це є доречним.

2.1 Встановлення та використання бібліотеки Pandas

Для встановлення Pandas у вашій системі потрібно встановити NumPy. Детально про встановлення можна ознайомитись у Pandas documentation [2]. Якщо ви слідували порадам, викладеним у попередніх лекціях, і використовували стек Anaconda, ви вже встановили Pandas. Після встановлення Pandas ви можете імпортувати його та перевірити версію [3]:

```
In [1]: import pandas
        Pandas.__version__
Out[1]: '2.1.1'
```

Подібно до того, як ми зазвичай імпортуємо NumPy під псевдонімом `np`, ми будемо імпортувати Pandas під псевдонімом `pd`:

```
In [2]: import pandas as pd
```

2.2 Введення до об'єктів Pandas

На самому базовому рівні об'єкти Pandas можна розглядати як вдосконалені версії структурованих масивів NumPy, в яких рядки та стовпці ідентифікуються мітками, а не простими цілочисельними індексами. Pandas пропонує велику кількість корисних інструментів, методів та функціональних можливостей поверх базових структур даних, але майже все, що буде використані, вимагатиме розуміння того, що це за структури. Отже, перед тим, як йти далі, давайте представимо ці три основні структури даних Pandas: `Series`, `DataFrame` та `Index`.

Розпочнемо роботу зі стандартного імпортування NumPy та Pandas:

```
In [3]: import numpy as np
        import pandas as pd
```

2.3 Об'єкт Series Pandas

Pandas `Series` – це одновимірний масив індексованих даних. Його можна створити зі списку або масиву наступним чином:

```
In [4]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
        data
Out[4]: 0 0.25
        1 0.50
        2 0.75
        3 1.00
        dtype: float64
```

Як ми бачимо в `Out[4]`, `Series` являє собою і послідовність значень, і послідовність індексів, до яких ми можемо отримати доступ за значеннями та атрибутами індексу. Ці значення – це просто звичний масив NumPy:

```
In [5]: data.values
Out[5]: array([0.25, 0.5 , 0.75, 1.  ])
```

`index` – це масивноподібний об'єкт типу `pd.Index`, про який ми поговоримо докладніше

```
In [6]: data.index
Out[6]: RangeIndex(start=0, stop=4, step=1)
```

Як і в масиві NumPy, доступ до даних можна отримати за допомогою відповідного індексу через звичну нотацію квадратних дужок Python:

```
In [7]: data[1]
Out[7]: 0.5
In [8]: data[1:3]
Out[8]: 1 0.50
        2 0.75
        dtype: float64
```

Однак, як ми побачимо, об'єкт **Series** Pandas набагато більш загальний та гнучкий, ніж одновимірний масив NumPy, який він емулює.

2.3.1 Series як узагальнений масив NumPy

З того, що ми тільки що побачили, може виглядати, що об'єкт **Series** в основному взаємозамінний з одновимірним масивом NumPy. Суттєвою відмінністю є наявність індексу: в той час як масив NumPy має *неявно визначений* цілочисельний індекс, що використовується для доступу до значень, Pandas **Series** має *явно визначений* індекс, пов'язаний зі значеннями. Це явне визначення індексу надає об'єкту **Series** додаткові можливості. Наприклад, індекс не обов'язково повинен бути цілим числом, але він може складатися зі значень будь-якого бажаного типу. Наприклад, якщо ми хочемо, ми можемо використовувати рядки як індекс:

```
In [9]: data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a',
'b', 'c', 'd'])
data
Out[9]: a 0.25
        b 0.50
        c 0.75
        d 1.00
        dtype: float64
```

```
In [10]: data.index
Out[10]: Index(['a', 'b', 'c', 'd'],
              dtype='object')
```

І доступ до елементів працює як і раніше:

```
In [11]: data['b']
Out[11]: 0.5
```

Ми можемо використовувати навіть несуміжні або непослідовні індекси:


```
In [12]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
index=[2, 5, 3, 7])
data
Out[12]: 2 0.25
          5 0.50
          3 0.75      7 1.00
          dtype: float64
```

2.3.2 Series як спеціалізований словник

Таким чином, ви можете уявити собі **Series** Pandas, трохи схожим на особливу реалізацію словника Python. Словник – це структура, яка відображає довільні ключі до набору довільних значень, а **Series** – це структура, яка ставить у відність типизовані ключі до набору типизованих значень. Ця типизація важлива: так само, як скомпільований за типом код масиву NumPy робить його ефективнішим, ніж список Python для певних операцій, інформація про тип Pandas **Series** робить його набагато ефективнішим, ніж словники Python для певних операцій.

Аналогію **Series** як словник можна зробити ще більш зрозумілою, побудувавши об'єкт **Series** безпосередньо зі словника Python:

```
In [13]: population_dict = {'Kharkiv': 1419036,
                             'Kherson': 289697,
                             'Mariupol': 446103,
                             'Lviv': 721301,
                             'Kyiv': 2884359}
population = pd.Series(population_dict)
population
Out[13]: Kharkiv    1419036
         Kherson    289697
         Mariupol   446103
         Lviv       721301
         Kyiv       2884359
         dtype: int64
```

За замовчуванням буде створено **Series**, де індекс виводиться з ключів. Звідси можна здійснити типовий доступ до елементів у стилі словника:

```
In [14]: population['Kharkiv']
Out[14]: 1419036
```

Однак, на відміну від словника, **Series** також підтримує операції в стилі масиву, такі як слайсинг:

```
In [14]: population['Kherson':'Lviv']
Out[14]: Kherson    289697
         Mariupol   446103
         Lviv       721301
         dtype: int64
```

2.3.3 Побудова об'єктів Series

Ми вже бачили кілька способів побудови Pandas **Series** з нуля; всі вони є деякою версією наступного:

```
In[]: pd.Series(data, index=index),
```

де **index** є необов'язковим аргументом, а **data** можуть бути однією з багатьох сутностей.

Наприклад, **data** можуть бути списком або масивом NumPy, у цьому випадку за замовчуванням **index** має цілочисельну послідовність:

```
In [15]: pd.Series([2, 4, 6])
Out[15]: 0    2
         1    4
         2    6
         dtype: int64
```

data може бути скаляром, який повторюється для заповнення вказаного індексу:

```
In [16]: pd.Series(5, index=[100, 200, 300])
Out[16]: 100    5
         200    5
         300    5
         dtype: int64
```

data може бути словником, в якому **index** за замовчуванням використовує ключі словника:

```
In [17]: pd.Series({'2':'a', '1':'b', '3':'a'})
Out[17]: 2    a
         1    b
         3    a
         dtype: object
```

У кожному випадку індекс може бути явно встановлений, якщо бажано отримати інший результат:

	population	area
Kharkiv	1419036	350
Kherson	289697	145
Mariupol	446103	135
Lviv	721301	149
Kyiv	2884359	856

Як і об'єкт `Series`, `DataFrame` має атрибут `index`, що надає доступ до міток індексу:

```
In [21]: cities.index
Out[21]: Index(['Kharkiv', 'Kherson', 'Mariupol',
               'Lviv', 'Kyiv'], dtype='object')
```

Крім того, `DataFrame` має атрибут `columns`, який є об'єктом `Index`, що містить мітки стовпців:

```
In [22]: cities.columns
Out[22]: Index(['population', 'area'], dtype='object')
```

Таким чином, `DataFrame` можна розглядати як узагальнення двовимірного масиву NumPy, де і рядки, і стовпці мають узагальнений індекс для доступу до даних.

2.4.2 DataFrame як спеціалізований словник

Подібним чином ми можемо розглядати `DataFrame` як спеціалізацію словника. Там, де словник ставить у відповідність ключ до значення, `DataFrame` ставить у відповідність ім'я стовпця до `Series` даних стовпців. Наприклад, запит атрибуту `area` повертає об'єкт `Series`, що містить площі, які ми бачили раніше:

```
In [23]: cities['area']
Out[23]: Kharkiv    350
         Kherson    145
         Mariupol   135
         Lviv       149
         Kyiv       856
         Name: area, dtype: int64
```

Зверніть увагу на потенційну можливість плутанини: у двовимірному масиві NumPy `data[0]` поверне перший рядок. Для `DataFrame`, `data['col0']` поверне перший стовець. Через це, мабуть, краще думати про `DataFrame` як про узагальнені словники, а не як про узагальнені масиви, хоча обидва способи розгляду ситуації можуть бути корисними

2.5 Побудова об'єктів DataFrame

Pandas DataFrame може бути побудовано різними способами. Тут ми наведемо кілька прикладів.

2.5.1 З одного об'єкта Series

DataFrame – це колекція Series об'єктів, і DataFrame з одним стовпчиком може бути побудований з одного Series:

```
In[24]:pd.DataFrame(population,columns=['population'])  
Out[24]:
```

population	
Kharkiv	1419036
Kherson	289697
Mariupol	446103
Lviv	721301
Kyiv	2884359

2.5.2 Зі списку словників

З будь-якого списку словників можна створити DataFrame. Ми використаємо просте заповнення списку для створення даних:

```
In [25]: data = [{ 'a': i, 'b': 2 * i} for i in  
range(5)] pd.DataFrame(data) Out[25]:
```

	a	b
0	0	0
1	1	2
2	2	4
3	3	6
4	4	8

Навіть якщо в словнику відсутні деякі ключі, Pandas заповнить їх значеннями NaN (тобто "not a number"):

```
In [26]: pd.DataFrame([{'a': 1, 'b': 2},  
{'b': 3, 'c': 4}]) Out[26]:
```

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

2.5.3 Зі словника об'єктів Series

Як ми бачили раніше, `DataFrame` також може бути побудований зі словника об'єктів `Series`:

```
In [27]: pd.DataFrame({'population': population,
'area': area}) Out[27]:
```

	population	area
Kharkiv	1419036	350
Kherson	289697	145
Mariupol	446103	135
Lviv	721301	149
Kyiv	2884359	856

2.5.4 З двовимірною масиву NumPy

Маючи двовимірний масив даних, ми можемо створити `DataFrame` з будь-якими вказаними іменами стовпців та індексів. Якщо їх не вказувати, для кожного із стовпців буде використовуватися цілочисельний індекс:

```
In [28]: pd.DataFrame(np.random.rand(3, 2),
columns=['foo', 'bar'], index=['a', 'b', 'c'])
Out[28]:
```

	foo	bar
a	0.780121	0.846270
b	0.166138	0.396992
c	0.846922	0.827634

2.6 Об'єкт Pandas Index

Ми бачили, що об'єкти `Series` і `DataFrame` містять явний індекс, що дозволяє посилатися та змінювати дані. Цей об'єкт `Index` сам по собі є цікавою структурою, і його можна сприймати і як незмінний масив, так і як упорядковану множину (технічно мультимножину, оскільки об'єкти `Index` можуть містити повторювані значення). Це має деякі цікаві наслідки в операціях, доступних над об'єктами `Index`. Як простий приклад, давайте побудуємо `Index` зі списку цілих чисел:

```
In [29]: ind = pd.Index([2, 3, 5, 5, 7, 11]) ind
Out[29]: Int64Index([2, 3, 5, 5, 7, 11], dtype='int64') In
[30]: df_ex=pd.DataFrame(['a','b','c','d','e','f'],
index=ind)
df_ex Out[30]:
```

0	
2	a
3	b
5	c
5	d
7	e
11	f

2.6.1 Index як незмінний масив

Index багато в чому працює як масив. Наприклад, ми можемо використовувати стандартну нотацію індексації Python для отримання значень або фрагментів:

```
In [31]: ind[1]
Out[31]: 3
In [32]: ind[:2]
Out[32]: Int64Index([2, 5, 7], dtype='int64')
```

Index об'єкти також мають багато атрибутів, знайомих з масивів NumPy:

```
In [33]: print(ind.size, ind.shape, ind.ndim,
ind.dtype) Out[33]: 6 (6,) 1 int64
```

Однією з різниць між об'єктами Index та масивами NumPy є те, що індекси незмінні – тобто їх не можна змінювати звичайними засобами:

```
In [34]: try:
    ind[1] = 0 except TypeError as e:
    print (f'TypeError: ',e)
Out[34]: TypeError: Index does not support mutable
operations
```

Ця незмінність робить більш безпечним обмін індексами між кількома DataFrame та масивами, без потенційних побічних ефектів від ненавмисної модифікації індексу.

2.6.2 Index як множина

Об'єкти Pandas призначені для полегшення таких операцій, як об'єднання наборів даних, які залежать від багатьох аспектів арифметики множин. Об'єкт Index дотримується багатьох домовленостей, використовуваних вбудованою структурою даних set Python, так що об'єднання, перетини, відмінності та інші комбінації можуть бути обчислені звичним способом:

```
In [35]: indA = pd.Index([1, 3, 5, 7, 9])      indB =  
pd.Index([2, 3, 5, 7, 11])  
In [36]: indA.intersection(indB)  
Out[36]: Int64Index([3, 5, 7], dtype='int64')  
In [37]: indA.union(indB)  
Out[37]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')  
In [38]: indA.symmetric_difference(indB)  
Out[38]: Int64Index([1, 2, 9, 11], dtype='int64'))
```

2.7 Індексція та вибір даних

Раніше ми детально розглядали методи та засоби доступу, встановлення та модифікації значень у масивах NumPy. Сюди входили індексція (наприклад, `arr[2, 1]`), слайсінг (наприклад, `arr[:, 1:5]`), маски (наприклад, `arr[arr > 0]`), розширене індексування (fancy indexing) (наприклад, `arr[0, [1, 5]]`) та їх комбінації (наприклад, `arr[:, [1, 5]]`). Зараз ми розглянемо подібні засоби доступу та модифікації значень у об'єктах Pandas Series і DataFrame. Якщо ви використовували шаблони NumPy, відповідні шаблони в Pandas будуть відчуватися дуже знайомими, хоча є кілька відмінностей, про які слід пам'ятати.

Почнемо з простого випадку одновимірного об'єкта Series, а потім перейдемо до більш складного двовимірного об'єкта DataFrame.

2.7.1 Вибір даних у Series

Як ми бачили в попередньому розділі, об'єкт Series багато в чому діє як одновимірний масив NumPy і багато в чому як стандартний словник Python. Будемо пам'ятати про ці дві аналогії, що перекриваються, і це допоможе нам зрозуміти закономірності індексації та відбору даних у цих масивах.

Як і словник, об'єкт Series забезпечує співставлення набору ключів до набору значень:

```
In [39]: data = pd.Series([0.25, 0.5, 0.75, 1.0],  
index=['a', 'b', 'c', 'd'])  
data  
Out[39]: a    0.25  
         b    0.50  
         c    0.75  
         d    1.00  
         dtype: float64  
In [40]: data['b']  
Out[40]: 0.5
```

Ми також можемо використовувати словникові вирази та методи Python для дослідження ключів/індексів та значень:


```
In [41]: 'a' in data
Out[41]: True
In [42]: data.keys()
Out[42]: Index(['a', 'b', 'c', 'd'], dtype='object')
In [43]: list(data.items())
Out[43]: [('a', 0.25), ('b', 0.5), ('c', 0.75),
('d', 1.0)]
```

`Series` об'єкти можна навіть модифікувати за допомогою словникового синтаксису. Подібно до того, як ви можете розширити словник, призначивши новий ключ, ви можете продовжити `Series`, призначивши нове значення індексу:

```
In [44]: data['e'] = 1.25      data
Out[44]: a    0.25
         b    0.50
         c    0.75
         d    1.00
         e    1.25
         dtype: float64
```

Ця легка зміна об'єктів є зручною особливістю: під капотом `Pandas` приймає рішення щодо розміщення пам'яті та копіювання даних, які можуть знадобитися; користувачеві, як правило, не потрібно турбуватися про ці проблеми.

`Series` базується на словниковому інтерфейсі та забезпечує вибір елементів у стилі масиву за допомогою тих самих основних механізмів, що й масиви `NumPy` – тобто зрізи, маски та *fancy* індексування. Приклади:

```
In [45]: # Зрізи по явному індексу data['a':'c']
Out[45]: a    0.25
         b    0.50
         c    0.75
         dtype: float64
In [46]: # Зрізи по неявному індексу data[0:2]
Out[46]: a    0.25
         b    0.50
         dtype: float64
In [47]: # Використання маски data[(data > 0.3) & (data <
0.8)]
Out[47]: b    0.50
         c    0.75
         dtype: float64
```

```
In [48]: # Розширене індексування data[['a', 'e']]
```

```
Out[48]: a    0.25
         e    1.25
         dtype: float64
```

Серед всіх цих інструментів, зрізи можуть спричинити найбільшу плутанину. Зверніть увагу, що при слайсингу з *явним індексом* (тобто `data ['a': 'c']`), кінцевий індекс *включається* в зріз, тоді як при зрізах з *неявним індексом* (тобто `data[0:2]`), остаточний індекс *виключається* зі зрізу.

2.7.2 Індиксатори: `loc` і `iloc`

Правила слайсингу та індексування можуть викликати плутанину. Наприклад, якщо у вашому `Series` є явний цілочисельний індекс, операція індексації, така як `data[1]`, використовуватиме явні індекси, тоді як операція слайсингу, як `data[1: 3]` буде використовувати неявний індекс у стилі Python.

```
In [49]: data = pd.Series(['a', 'b', 'c'],                      index=[1,
3, 5])
         data Out[49]: 1    a
                     3    b    5    c    dtype: object
In [50]: # Використання явного індексу для доступу
data[1]
Out[50]: 'a'
In [51]: # Використання неявного індексу для зрізів
data[1:3]
Out[51]: 3    b
         5    c
         dtype: object
```

Через цю потенційну плутанину у випадку цілочисельних індексів, Pandas надає деякі спеціальні атрибути `indexer`, які явно розкривають певні схеми індексації. Це не функціональні методи, а атрибути, які надають певний інтерфейс нарізки даним у `Series`.

По-перше, атрибут `loc` дозволяє індексувати та нарізати, з посиланням на *явний* індекс:

```
In [52]: data.loc[1]
Out[52]: 'a'
In [53]: data.loc[1:3]
Out[53]: 1    a
         3    b
         dtype: object
```

`iloc` атрибут дозволяє індексувати та нарізати, з посиланням на

неявний індекс у стилі Python:

```
In [52]: data.iloc[1]
Out[52]: 'b'
In [53]: data.iloc[1:3]
Out[53]: 3    b
         5    c
         dtype: object
```

Одним із керівних принципів Python, є «явне краще, ніж неявне» (“Explicit is better than implicit”). Явна природа `loc` та `iloc` робить їх дуже корисними для підтримки чистого та читабельного коду, особливо у випадку цілочисельних індексів. Тому використання цих атрибутів є рекомендованим як для полегшення читання та розуміння коду, так і для запобігання дрібним помилкам.

2.7.3 Вибір даних у DataFrame

Нагадаємо, що `DataFrame` з одного боку діє як двовимірний або структурований масив, з іншого – як словник структур `Series`, що мають один і той же індекс. Ці аналогії можуть бути корисними, коли ми досліджуємо відбір даних у цій структурі.

Першою аналогією, яку ми розглянемо, є `DataFrame` як словник пов'язаних об'єктів `Series`. Повернемося до нашого прикладу площі і населення міст України:

```
In [54]: area = pd.Series({'Kharkiv': 350,
                          'Kherson': 145,
                          'Mariupol': 135,
                          'Lviv': 149,
                          'Kyiv': 856})
pop = pd.Series({'Kharkiv': 1419036,
                'Kherson': 289697,
                'Mariupol': 446103,
                'Lviv': 721301,
                'Kyiv': 2884359})
data=pd.DataFrame({'area':area, 'pop':pop})
data
```

Out[54]:

	area	pop
Kharkiv	350	1419036
Kherson	145	289697
Mariupol	135	446103
Lviv	149	721301
Kyiv	856	2884359

Окремі `Series`, що складають стовпці `DataFrame`, можна отримати за допомогою індексації назви стовпця у стилі словника:

```
In [55]: data['area']
Out[55]: Kharkiv    350
         Kherson    145
         Mariupol   135
         Lviv       149
         Kyiv       856
         Name: area, dtype: int64
```

Крім того, ми можемо використовувати доступ до стовпчиків як до атрибутів `DataFrame`

```
In [56]: data.area
Out[56]: Kharkiv    350
         Kherson    145
         Mariupol   135
         Lviv       149
         Kyiv       856
         Name: area, dtype: int64
```

Цей доступ до стовпця як до атрибутів насправді отримує такий самий об'єкт, як і доступ у стилі словника:

```
In [57]: data.area is data['area']
Out[57]: True
```

Хоча це корисне скорочення, майте на увазі, що воно працює не у всіх випадках! Наприклад, якщо імена стовпців не є рядками, або якщо імена стовпців конфліктують з методами `DataFrame`, такий тип доступу неможливий. Наприклад, у `DataFrame` є метод `pop()`, тому `data.pop` вказуватиме на метод, а не на стовпець `pop`:

```
In [58]: data.pop is data['pop']
Out[58]: False
```

Як і для обговорених раніше об'єктів `Series`, синтаксис у стилі словника також може бути використаний для модифікації об'єкта, в цьому випадку додаючи новий стовпець:

```
In [59]: data['density'] = data['pop']/data['area'] data
Out[59]:
```

	area	pop	density
Kharkiv	350	1419036	4054.388571
Kherson	145	289697	1997.910345
Mariupol	135	446103	3304.466667
Lviv	149	721301	4840.946309
Kyiv	856	2884359	3369.578271

Як вже згадувалося раніше, ми також можемо розглядати `DataFrame` як вдосконалений двовимірний масив. Ми можемо перевірити базовий масив даних, використовуючи атрибут `values`:

```
In [60]: data.values[3,2] #щільність населення у Львові
Out[60]: 4840.946308724832
```

Маючи це на увазі, багато відомих дій, за аналогією з масивом, можна зробити на самому `DataFrame`. Наприклад, ми можемо транспонувати повний `DataFrame`, щоб поміняти місцями рядки та стовпці:

```
In [61]: data.T Out[61]:
```

	Kharkiv	Kherson	Mariupol	Lviv	Kyiv
area	3.500000e+02	145.000000	135.000000	149.000000	8.560000e+02
pop	1.419036e+06	289697.000000	446103.000000	721301.000000	2.884359e+06
density	4.054389e+03	1997.910345	3304.466667	4840.946309	3.369578e+03

Що стосується індексації об'єктів `DataFrame`, то очевидно, що індексація стовпців у стилі словника виключає можливість просто розглядати це як масив `NumPy`. Зокрема, передача одного індексу масиву отримує доступ до рядка:

```
In [62]: data.values[0]
Out[62]: array([3.50000000e+02, 1.41903600e+06,
                4.05438857e+03])
```

Передача одного `index` в `DataFrame` дає доступ до стовпця:

```
In [63]: data['area']
Out[63]: Kharkiv    350
         Kherson    145
         Mariupol   135
         Lviv      149
         Kyiv      856
         Name: area, dtype: int64
```

Таким чином, для індексації в стилі масиву нам потрібна інша умова. Тут `Pandas` знову використовує згадані раніше індексатори `loc`, `iloc`. Використовуючи індексатор `iloc`, ми можемо індексувати базовий масив так,

ніби це простий масив NumPy (з використанням неявного індексу), але в результаті зберігаються мітки індексів рядка та стовпця DataFrame:

```
In [64]: data.iloc[:3, :2] Out[64]:
```

	area	pop
Kharkiv	350	1419036
Kherson	145	289697
Mariupol	135	446103

Подібним чином, використовуючи індексатор loc, ми можемо індексувати дані у стилі, подібному до масиву, але використовуючи явні імена індексів та стовпців:

```
In [65]: data.loc[:, 'Mariupol', : 'pop'] Out[65]:
```

	area	pop
Kharkiv	350	1419036
Kherson	145	289697
Mariupol	135	446103

Будь-який із звичних шаблонів доступу до даних у стилі NumPy можна використовувати в цих індексаторах. Наприклад, в індексаторі loc ми можемо поєднувати використання масок та розширеного індексування, як показано нижче:

```
In [66]: data.loc[data.density>3000, ['pop', 'density']] Out[66]:
```

	pop	density
Kharkiv	1419036	4054.388571
Lviv	721301	4840.946309
Kyiv	2884359	3369.578271

Будь-яка з цих домовленостей щодо індексування також може використовуватися для встановлення або модифікації значень; це робиться стандартним способом, до якого ви могли б звикнути працювати з NumPy:

```
In [67]: data.iloc[2, 2] = 900 data Out[67]:
```

	area	pop	density
Kharkiv	350	1419036	4054.388571
Kherson	145	289697	1997.910345
Mariupol	135	446103	900.000000
Lviv	149	721301	4840.946309
Kyiv	856	2884359	3369.578271

Існує кілька додаткових правил щодо індексування, які можуть здатися розбіжними з попереднім обговоренням, але тим не менше можуть бути дуже корисними на практиці.

По-перше, тоді як *індексація* відноситься до стовпців, *слайсинг* відноситься до рядків:

```
In [68]: data['Kharkiv':'Mariupol'] Out[68]:
```

	area	pop	density
Kharkiv	350	1419036	4054.388571
Kherson	145	289697	1997.910345
Mariupol	135	446103	900.000000

Такі фрагменти також можуть посилатися на рядки за номером, а не за індексом:

```
In [69]: data[1:3] Out[69]:
```

	area	pop	density
Kherson	145	289697	1997.910345
Mariupol	135	446103	900.000000

Подібним чином, операції застосування маски також інтерпретуються по рядках, а не по стовпцях:

```
In [70]: data[data.density > 4000] Out[70]:
```

	area	pop	density
Kharkiv	350	1419036	4054.388571
Lviv	149	721301	4840.946309

Якщо ж необхідно отримати доступ до кількох стовпчиків, то можна скористатися наступними методами: In [71]: data[['area', 'pop']] Out[71]:

	area	pop
Kharkiv	350	1419036
Kherson	145	289697
Mariupol	135	446103
Lviv	149	721301
Kyiv	856	2884359

```
In [72]: data.loc[:, 'area':'pop'] Out[72]:
```

	area	pop
Kharkiv	350	1419036
Kherson	145	289697
Mariupol	135	446103
Lviv	149	721301
Kyiv	856	2884359

2.8 Операція з даними в Pandas

Однією з найважливіших частин NumPy є здатність виконувати швидкі елементарні операції, як з базовою арифметикою (додавання, віднімання, множення тощо), так і з більш складними операціями (тригонометричні

функції, експоненційні та логарифмічні функції тощо). Pandas успадковує більшу частину цієї функціональності від NumPy.

Pandas включає в себе кілька корисних функцій: для унарних операцій, таких як заперечення та тригонометричні функції, ці універсальні функції будуть зберігати мітки індексів рядків та стовпців на виході, а для бінарних операцій, таких як додавання та множення, Pandas автоматично «вирівнює» індекси, при передачі об'єктів до універсальних функцій. Це означає, що збереження вмісту даних та комбінування даних з різних джерел – що є потенційно схильними до помилок при роботі з необробленими масивами NumPy – стають по суті надійними із Pandas. Додатково ми побачимо, що існують чітко визначені операції між одновимірними структурами `Series` та двовимірними структурами `DataFrame`.

2.8.1 Універсальні функції: збереження індексу

Оскільки Pandas призначений для роботи з NumPy, будь-які універсальні функції NumPy будуть працювати у Pandas `Series` та `DataFrame` об'єктах. Почнемо з визначення простих `Series` та `DataFrame`, на яких це можна продемонструвати:

```
In [73]: rnd = np.random.RandomState(10)
         a = pd.Series(rnd.randint(0, 10, 4))
         a
Out[73]: 0    9
         1    4
         2    0
         3    1
         dtype: int32
```

Якщо ми застосуємо універсальні функції NumPy цього об'єкта, результатом буде інший об'єкт Pandas із збереженими індексами:

```
In [74]: np.exp(a)
Out[74]: 0    8103.083928
         1    54.598150
         2     1.000000
         3     2.718282
         dtype: float64
```

Аналогічний результат отримаємо і з об'єктом `DataFrame`:

```
In [75]: df = pd.DataFrame(rnd.randint(0, 10, (3, 4)),
columns=['a', 'b', 'c', 'd']) df Out[75]:
```


	a	b	c	d
0	6	8	1	8
1	4	1	3	6
2	5	3	9	6

In [76]: np.square(df) Out[76]:

	a	b	c	d
0	36	64	1	64
1	16	1	9	36
2	25	9	81	36

2.8.2 Універсальні функції: Index Alignment

Для бінарних операцій над двома об'єктами `Series` або `DataFrame`, Pandas вирівнює індекси в процесі виконання операції. Це дуже зручно під час роботи з неповними даними.

Почнемо розгляд з об'єктів `Series`.

Як приклад, припустимо, ми поєднуємо два різні джерела даних і знаходимо лише чотири міста України за площею і чотири міста за населенням:

```
In [77]: area = pd.Series({'Kharkiv': 350,
                           'Mariupol': 135,
                           'Lviv': 149,
                           'Kyiv': 856})
population= pd.Series({'Kharkiv': 1419036,
                       'Kherson': 289697,
                       'Mariupol': 446103,
                       'Kyiv': 2884359})
```

Давайте подивимось, що відбувається, коли ми розділимо ці значення для обчислення щільності населення:

```
In [78]: population / area
Out[78]: Kharkiv    4054.388571
         Kherson    NaN
         Kyiv      3369.578271
         Lviv      NaN
         Mariupol   3304.466667
         dtype: float64
```

Отриманий масив містить об'єднання індексів двох вхідних масивів, які можна визначити, використовуючи стандартну арифметику для множин Python для цих індексів:

```
In [79]: area.index.union(population.index)
```

```
Out[79]: Index(['Kharkiv', 'Kherson', 'Kyiv',  
              'Lviv', 'Mariupol'], dtype='object')
```

Будь-який елемент, для якого той чи інший не має запису, позначається NaN, або Not a Number (не число), саме так Pandas позначає відсутні дані. Цей збіг індексів реалізовано таким чином для будь-якого вбудованого арифметичного виразу Python; будь-які відсутні значення за замовчуванням заповнюються NaN:

```
In [80]: A = pd.Series([1, 3, 5], index=[1, 2, 3])  
        B = pd.Series([2, 4, 6], index=[0, 1, 2])  
        A + B  
Out[80]: 0    NaN  
        1    5.0  
        2    9.0  
        3    NaN  
        dtype: float64
```

Якщо використання значень NaN не є бажаним, можна задати значення для заповнення, використовуючи відповідні об'єктні методи замість операторів. Наприклад, виклик `A.add(B)` еквівалентно виклику `A + B`, але дозволяє необов'язкову явну вказівку значення заливки для будь-яких елементів в A або B які можуть бути відсутніми:

```
In [81]: A.add(B, fill_value=0)  
Out[81]: 0    2  
        1    5.0  
        2    9.0  
        3    5  
        dtype: float64
```

При виконанні операцій між `DataFrame` та `Series` аналогічно підтримується вирівнювання індексу та стовпця. Операції між `DataFrame` та `Series` схожі на операції між двовимірним та одновимірним масивами NumPy. Розглянемо одну типову операцію, коли ми знаходимо різницю двовимірного масиву та одного з його рядків:

```
In [82]: A=pd.DataFrame(rnd.randint(0, 10, (2, 2)),  
                        columns=list('AB'))  
        A
```

```
Out[82]:
```

	A	B
0	9	1
1	9	4

```
In [83]: B=pd.DataFrame(rnd.randint(0, 10, (3, 3)),
```

```
columns=list('BAC'))
```

B

```
Out[83]:
```

	B	A	C
0	7	9	3
1	5	2	4
2	7	6	8

```
In [84]: A + B
```

```
Out[84]:
```

	A	B	C
0	18.0	8.0	NaN
1	11.0	9.0	NaN
2	NaN	NaN	NaN

Зверніть увагу, що індекси правильно вирівняні незалежно від їх порядку в двох об'єктах, до того ж індекси в результаті сортуються. Як і у випадку із **Series**, ми можемо використовувати відповідні арифметичні методи та передати будь-яке бажане **fill_value**, яке буде використано замість відсутніх значень. Тут ми заповнимо середнім арифметичним всіх значень в A (обчислюється шляхом складання рядків A):

```
In [85]: fill = A.stack().mean()
```

```
A.add(B, fill_value=fill)
```

```
Out[85]:
```

	A	B	C
0	18.00	8.00	8.75
1	11.00	9.00	9.75
2	11.75	12.75	13.75

3 Завдання

1. Завантажте набір даних про дитячі імена США з веб-сайту [kaggle.com](https://www.kaggle.com/kaggle/us-babynames?select=NationalNames.csv) (<https://www.kaggle.com/kaggle/us-babynames?select=NationalNames.csv>)
2. Виконайте завдання по варіантах.

Варіант	Номери вправ
1	1, 2, 3, 5, 10, 11, 12, 13, 14, 15, 16, 17, 18, 21, 22, 23, 24, 26
2	3, 4, 5, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19, 20, 22, 23, 24, 27
3	1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 18, 19, 20, 21, 23, 25, 27
4	1, 3, 6, 7, 8, 12, 13, 14, 15, 16, 17, 19, 20, 22, 24, 25, 26, 27
5	2, 4, 6, 7, 9, 10, 15, 16, 17, 18, 20, 21, 22, 23, 24, 25, 26, 27
6	1, 2, 3, 5, 10, 11, 12, 13, 14, 15, 16, 17, 18, 21, 22, 23, 24, 26
7	3, 4, 5, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19, 20, 22, 23, 24, 27
8	1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 18, 19, 20, 21, 23, 25, 27
9	1, 3, 6, 7, 8, 12, 13, 14, 15, 16, 17, 19, 20, 22, 24, 25, 26, 27
10	2, 4, 6, 7, 9, 10, 15, 16, 17, 18, 20, 21, 22, 23, 24, 25, 26, 27

Варіанти завдання:

1. Виведіть перші 8 рядків набору даних.

Очікуваний результат:

Out[3]:

	Id	Name	Year	Gender	Count
0	1	Mary	1880	F	7065
1	2	Anna	1880	F	2604
2	3	Emma	1880	F	2003
3	4	Elizabeth	1880	F	1939
4	5	Minnie	1880	F	1746
5	6	Margaret	1880	F	1578
6	7	Ida	1880	F	1472
7	8	Alice	1880	F	1414

2. Вивести останні 8 рядків набору даних.

Очікуваний результат:

Out[4]:

	Id	Name	Year	Gender	Count
1825425	1825426	Zo	2014	M	5
1825426	1825427	Zyeir	2014	M	5
1825427	1825428	Zyel	2014	M	5
1825428	1825429	Zykeem	2014	M	5
1825429	1825430	Zymeer	2014	M	5
1825430	1825431	Zymiere	2014	M	5
1825431	1825432	Zyran	2014	M	5
1825432	1825433	Zyryn	2014	M	5

3. Отримайте імена стовпців набору даних Очікуваний результат:

Out[4]: Index(['Id', 'Name', 'Year', 'Gender', 'Count'], dtype='object')

4. Отримайте загальну інформацію про дані у наборі даних.

Очікуваний результат:

Out[5]:

	Id	Year	Count
count	1.825433e+06	1.825433e+06	1.825433e+06
mean	9.127170e+05	1.972620e+03	1.846879e+02
std	5.269573e+05	3.352891e+01	1.566711e+03
min	1.000000e+00	1.880000e+03	5.000000e+00
25%	4.563590e+05	1.949000e+03	7.000000e+00
50%	9.127170e+05	1.982000e+03	1.200000e+01
75%	1.369075e+06	2.001000e+03	3.200000e+01
max	1.825433e+06	2.014000e+03	9.968000e+04

5. Знайдіть кількість унікальних імен у наборі даних Очікуваний результат:

Out[33]:

93889

6. Обчисліть кількість унікальних жіночих та чоловічих імен у цілому наборі даних *Очікуваний результат:*

Out[37]:

Name	
Gender	
F	64911
M	39199

7. Знайдіть 5 найпопулярніших чоловічих імен у 2010 році *Очікуваний результат:*

Out[45]:

	Id	Name	Year	Gender	Count
1677392	1677393	Jacob	2010	M	22082
1677393	1677394	Ethan	2010	M	17985
1677394	1677395	Michael	2010	M	17308
1677395	1677396	Jayden	2010	M	17152
1677396	1677397	William	2010	M	17030

8. Знайдіть найпопулярніше ім'я за результатами одного року (ім'я, для якого Count максимальне) *Очікуваний результат:*

The name is 'Linda' in 1947

9. Підрахуйте кількість записів, для яких Count - мінімальне у наборі. *Очікуваний результат:*

Out[10]: 254615

10. Підрахуйте кількість унікальних імен у кожному році *Очікуваний результат:*

Out[26]:

Name	
Year	
1880	1889
1881	1830
1882	2012
1883	1962
1884	2158

11. Знайдіть рік із найбільшою кількістю унікальних імен. *Очікуваний результат:*

Out[32]:

Name	
Year	
2008	32488

12. Знайдіть найпопулярніше ім'я в році з найбільшою кількістю унікальних імен (тобто у 2008 році) *Очікуваний результат:*

Out[24]:

'Jacob'

13. Знайдіть рік, коли ім'я "Jacob" було найпопулярнішим серед жіночих імен

Очікуваний результат:

	Id	Name	Year	Gender	Count
1455556	1455557	Jacob	2004	F	171

14. Знайти рік із найбільшою кількістю гендерно нейтральних імен (однакові чоловічі та жіночі імена) *Очікуваний результат:*

Out[19]:

Gender_neutral_names	
Year	
2008	2557

15. Знайдіть загальну кількість народжень за рік.

Очікуваний результат для перших 5 рядків:

Out[56]:

Count	
Year	
1880	201484
1881	192699
1882	221538
1883	216950
1884	243467

16. Знайдіть рік, коли народилося найбільше дітей *Очікуваний результат:*

Out[49]:

1957

17. Знайдіть кількість дівчаток та хлопчиків, які народились кожного року

Очікуваний результат для перших 5 рядків:

Out[50]:

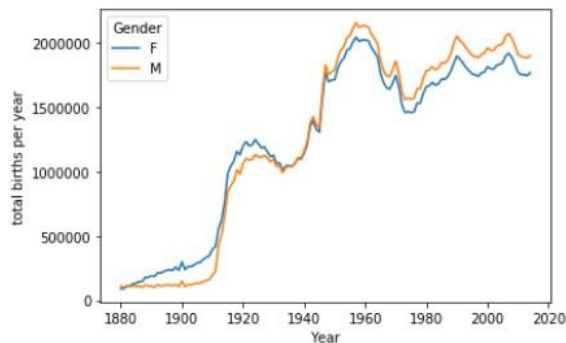
Gender	F	M
Year		
1880	90993	110491
1881	91954	100745
1882	107850	113688
1883	112321	104629
1884	129022	114445

18. Підрахуйте кількість років, коли дівчаток народжувалось більше, ніж хлопчиків. *Очікуваний результат:*

Out[64]: 54

19. Накресліть графік загальної кількості народжень хлопчиків та дівчаток на рік.

Очікуваний результат:



20. Підрахуйте кількість гендерно-нейтральних імен (однакових для дівчат та хлопців) *Очікуваний результат:*

Out[85]: 10221

21. Порахуйте, скільки разів хлопчиків називали Barbara *Очікуваний результат:*

Out[99]: 4139

22. Підрахуйте скільки років проводилось спостереження *Очікуваний результат:*

Out[218]: 'Спостереження проводилось 135 років'

23. Знати найпопулярніші гендерно-нейтральні імена (ті, що присутні кожного року)

Очікуваний результат:

Out[219]:

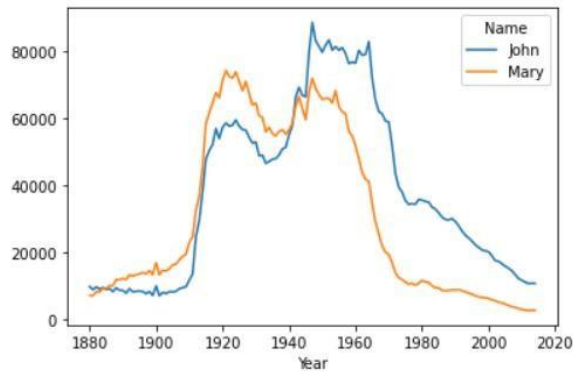
	0
0	James
1	Leslie
2	Joseph
3	Jessie
4	Jesse
5	Sidney
6	John
7	Robert
8	Tommie
9	Jean
10	Johnnie
11	William
12	Lee
13	Marion
14	Francis
15	Ollie

24. Знайти найпопулярніше серед непопулярних імен (непопулярне ім'я, яким називали дітей найбільшу кількість разів) *Очікуваний результат:*

Out[94]: 'Наиболее популярное из непопулярных имен - это Celester. Им называли 160 раз'

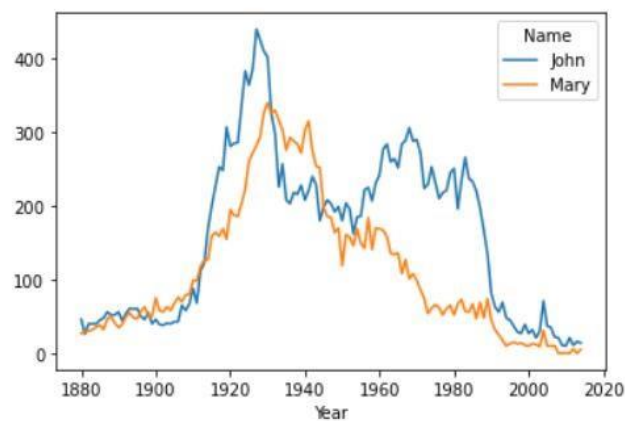
25. Побудувати графіки розподілення кількості імен John та Mary по роках без залежності до статі.

Очікуваний результат:



26. Побудувати графіки розподілення кількості жіночих імен John та чоловічих імен Mary по роках.

Очікуваний результат:



27. Знайти найпопулярніші імена в кожному році.

Очікуваний результат:

Out[214]:

	Name	Count
Year		
1880	John	9655
1881	John	8769
1882	John	9557
1883	John	8894
1884	John	9388
...
2010	Isabella	22883
2011	Sophia	21816
2012	Sophia	22267
2013	Sophia	21147
2014	Emma	20799

4 Контрольні запитання

1. Чи потребує бібліотека Pandas окремого встановлення?
2. Назвіть та охарактеризуйте структури даних Pandas.
3. В чому збіжність та відмінність Pandas Series від одновимірного масиву ndarray?
4. В чому збіжність та відмінність Pandas DataFrame від двовимірного масиву ndarray?
5. Які властивості Pandas Series та DataFrame походять від стандартних словників Python?
6. Охарактеризуйте методи доступу до елементів з використанням явного та неявного індексів.
7. В чому полягає перевага використання атрибутів loc та iloc?
8. В чому сутність операції збереження індексу?
9. В чому сутність вирівнювання індексу?
10. Для яких об'єктів Pandas застосовуються операції вирівнювання та збереження індексів?

СПИСОК ЛІТЕРАТУРИ

1. Jules DamjiDenny LeeBrooke WenigTathagata Das. Learning Spark // O'Reilly Media, 2020. – 300 p.
2. Bill Chambers. Spark: The Definitive Guide: Big Data Processing Made Simple// O'Reilly Media, 2018. – 608 p.
3. Документація з Pandas. Режим доступу – <http://pandas.pydata.org/>
4. User Guide. Режим доступу – https://pandas.pydata.org/docs/user_guide/index.html
5. McKinney W. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and Ipython / W. McKinney. – 2nd. Ed. – O'Reilly Media, 2017. – 550 p.