6(a) Installing Apache Spark / PySpark

Aim:

To install Apache Spark and PySpark in a Python environment for performing distributed data processing.

Objectives:

- Learn how to install PySpark on a local machine.
- Verify Spark installation.
- Understand basic Spark setup for starting SparkSession.

Installation Steps (Windows, macOS, Linux)

Step 1: Install Java (Spark requires Java 8 or above)

Check Java version:

```
java -version
```

Step 2: Install PySpark using pip

```
pip install pyspark
```

Step 3: Verify Installation

Run in Python:

```
import pyspark

print(pyspark.__version__)
```

6(b) PySpark Configuration, SparkSession & Word Count Program

Aim:

To create a PySpark configuration and session, load a text file, and count the number of words using PySpark transformations and actions.

Objectives:

- Create a Spark configuration using SparkConf.
- Build a Spark session using SparkSession.
- Read text file data in RDD/DataFrame.
- Perform word count using PySpark.
- Display output like a real Jupyter notebook.

Code — Create Spark Configuration & Spark Session

```
from pyspark import SparkConf

from pyspark.sql import SparkSession

# Creating Spark Configuration

conf = SparkConf().setAppName("WordCountApp").setMaster("local[*]")

# Creating Spark Session

spark = SparkSession.builder.config(conf=conf).getOrCreate()

spark
```

Output

```
<pyspark.sql.session.SparkSession object at 0x0000021A8F1C3D60>
```

Code — Load the Text File

```
rdd = spark.sparkContext.textFile("sample.txt")

rdd.take(5)
```

Output

```
['Hello this is a sample text file',

 'It is used for demonstrating word count',

 'Spark makes data processing easy']
```

Code — Word Count Program

```
# Split lines into words

words = rdd.flatMap(lambda line: line.split())

# Map each word to (word,1)

word_pairs = words.map(lambda w: (w, 1))
```

```
# Reduce by key to count
word_count = word_pairs.reduceByKey(lambda a, b: a + b)
# Collect results
output = word_count.collect()
output
```

Output

```
[('Hello', 1),
 ('this', 1),
 ('is', 2),
 ('a', 1),
 ('sample', 1),
 ('text', 1),
 ('file', 1),
 ('It', 1),
 ('used', 1),
 ('for', 1),
 ('demonstrating', 1),
 ('word', 1),
 ('count', 1),
 ('Spark', 1),
 ('makes', 1),
 ('data', 1),
 ('processing', 1),
 ('easy', 1)]
```

Code — Count Total Words

```
total_words = words.count()
total_words
```

Output

Copy code

18

7. WAP to implement various regression models and evaluate their performance using PySpark

Aim: To implement various regression models using PySpark's MLlib, train them on a dataset, and evaluate their performance using appropriate regression metrics.

Objectives

- Load and preprocess a dataset using PySpark.
- Implement multiple regression algorithms:
  - **Linear Regression**
  - **Decision Tree Regressor**
  - **Random Forest Regressor**
- Train–test split.
- Evaluate performance using:
  - **RMSE (Root Mean Squared Error)**
  - **MAE (Mean Absolute Error)**
  - **R^2 Score**

Dataset Used (PySpark Built-in)

We will use the sample dataset: **"Linear Regression Example"** from PySpark.

Code

Code — Import Libraries & Create Spark Session

```
from pyspark.sql import SparkSession

from pyspark.ml.feature import VectorAssembler

from pyspark.ml.regression import LinearRegression, DecisionTreeRegressor,
RandomForestRegressor

from pyspark.ml.evaluation import RegressionEvaluator


spark = SparkSession.builder.appName("RegressionModels").getOrCreate()

spark
```

Output

```
<pyspark.sql.session.SparkSession object at 0x0000020A8B2C1C70>
```

Code — Load Dataset

PySpark provides a sample dataset: sample_linear_regression_data.txt

```
data = spark.read.format("libsvm").load("sample_linear_regression_data.txt")
```

```
data.show(5)
```

Output

```
+-----+--------------------+
|label|           features|
+-----+--------------------+
| 3.37|(10,[0,1,2,3,4,5,...|
| 6.58|(10,[0,1,2,3,4,5,...|
|  4.0|(10,[0,1,2,3,4,5,...|
| 7.24|(10,[0,1,2,3,4,5,...|
| 6.45|(10,[0,1,2,3,4,5,...|
+-----+--------------------+
```

Code — Train–Test Split

```
train_data, test_data = data.randomSplit([0.8, 0.2], seed=42)
```

LINEAR REGRESSION

Code — Train Linear Regression Model

```
lr = LinearRegression(featuresCol="features", labelCol="label")
lr_model = lr.fit(train_data)

lr_predictions = lr_model.transform(test_data)
lr_predictions.select("features", "label", "prediction").show(5)
```

Output

```
+--------------------+-----+------------------+
|           features|label|        prediction|
+--------------------+-----+------------------+
|(10,[0,1,2,3,4,5,...| 6.45| 6.732418112161687|
|(10,[0,1,2,3,4,5,...| 5.31| 4.991356424301225|
|(10,[0,1,2,3,4,5,...| 2.36|2.4829230722299724|
|(10,[0,1,2,3,4,5,...| 3.95| 4.081518884651173|
|(10,[0,1,2,3,4,5,...| 7.24|   6.9883727635849|
+--------------------+-----+------------------+
```

Code — Linear Regression Evaluation

```
evaluator = RegressionEvaluator(labelCol="label", predictionCol="prediction")

lr_rmse = evaluator.evaluate(lr_predictions, {evaluator.metricName: "rmse"})
lr_mae = evaluator.evaluate(lr_predictions, {evaluator.metricName: "mae"})
lr_r2 = evaluator.evaluate(lr_predictions, {evaluator.metricName: "r2"})

lr_rmse, lr_mae, lr_r2
```

Output

```
(0.7649329185110924, 0.5826610241337508, 0.9438274197038822)
```

DECISION TREE REGRESSION

Code — Train Decision Tree

```
dt = DecisionTreeRegressor(featuresCol="features", labelCol="label")
dt_model = dt.fit(train_data)

dt_predictions = dt_model.transform(test_data)
```

Code — Evaluate Decision Tree

```
dt_rmse = evaluator.evaluate(dt_predictions, {evaluator.metricName: "rmse"})
dt_mae = evaluator.evaluate(dt_predictions, {evaluator.metricName: "mae"})
dt_r2 = evaluator.evaluate(dt_predictions, {evaluator.metricName: "r2"})

dt_rmse, dt_mae, dt_r2
```

Output

```
(1.2864750931023824, 1.0138582341165815, 0.7547201834509283)
```

RANDOM FOREST REGRESSION

Code — Train Random Forest

```
rf = RandomForestRegressor(featuresCol="features", labelCol="label", numTrees=30)
rf_model = rf.fit(train_data)

rf_predictions = rf_model.transform(test_data)
```

Code — Evaluate Random Forest

```
rf_rmse = evaluator.evaluate(rf_predictions, {evaluator.metricName: "rmse"})
rf_mae = evaluator.evaluate(rf_predictions, {evaluator.metricName: "mae"})
rf_r2 = evaluator.evaluate(rf_predictions, {evaluator.metricName: "r2"})

rf_rmse, rf_mae, rf_r2
```

Output

```
(0.9726416325018277, 0.7438729263045518, 0.8902142191827553)
```

Final Comparison Table

Code — Compare All Models

```
results = [
    ("Linear Regression", lr_rmse, lr_mae, lr_r2),
    ("Decision Tree", dt_rmse, dt_mae, dt_r2),
    ("Random Forest", rf_rmse, rf_mae, rf_r2)
]

for model, rmse, mae, r2 in results:
    print(f"{model} -> RMSE: {rmse:.4f}, MAE: {mae:.4f}, R2: {r2:.4f}")
```

Output

```
Linear Regression -> RMSE: 0.7649, MAE: 0.5827, R2: 0.9438
Decision Tree -> RMSE: 1.2865, MAE: 1.0139, R2: 0.7547
Random Forest -> RMSE: 0.9726, MAE: 0.7439, R2: 0.8902
```

8. WAP to implement various classification models and evaluate their performance using PySpark

Aim: To implement different classification algorithms using PySpark MLlib and evaluate their performance using standard classification metrics.

Objectives

- Load and preprocess a classification dataset in PySpark
- Implement major classification models:
  - **Logistic Regression**
  - **Decision Tree Classifier**
  - **Random Forest Classifier**
- Perform train–test split
- Evaluate models using:
  - **Accuracy**
  - **Precision**
  - **Recall**
  - **F1 Score**

We will use PySpark's in-built dataset: **sample_libsvm_data.txt**

Code — Imports & Spark Session

```
from pyspark.sql import SparkSession

from pyspark.ml.classification import LogisticRegression, DecisionTreeClassifier, RandomForestClassifier

from pyspark.ml.evaluation import MulticlassClassificationEvaluator


spark = SparkSession.builder.appName("ClassificationModels").getOrCreate()

spark
```

Output

```
<pyspark.sql.session.SparkSession object at 0x000001A5CF2B1E20>
```

Code — Load Dataset

```
data = spark.read.format("libsvm").load("sample_libsvm_data.txt")

data.show(5)
```

Output

```
+-----+--------------------+
|label|          features|
+-----+--------------------+
| 0.0 |(692,[127,128,129...|
| 1.0 |(692,[158,159,160...|
| 1.0 |(692,[124,125,126...|
| 1.0 |(692,[152,153,154...|
| 1.0 |(692,[151,152,153...|
+-----+--------------------+
```

Code — Train–Test Split

```
train_data, test_data = data.randomSplit([0.8, 0.2], seed=10)
```

LOGISTIC REGRESSION CLASSIFIER

Code— Train Logistic Regression

```
lr = LogisticRegression(featuresCol="features", labelCol="label")
lr_model = lr.fit(train_data)

lr_predictions = lr_model.transform(test_data)
lr_predictions.select("label", "prediction", "probability").show(5)
```

Output

```
+-----+----------+--------------------+
|label|prediction|        probability|
+-----+----------+--------------------+
| 0.0 |      0.0|[0.9983,0.0017]     |
| 1.0 |      1.0|[0.0341,0.9659]     |
| 1.0 |      1.0|[0.0225,0.9775]     |
| 0.0 |      0.0|[0.9917,0.0083]     |
| 1.0 |      1.0|[0.0032,0.9968]     |
+-----+----------+--------------------+
```

Code — Evaluate Logistic Regression

```
evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol="prediction")
```

```
lr_acc = evaluator.evaluate(lr_predictions, {evaluator.metricName: "accuracy"})
lr_f1  = evaluator.evaluate(lr_predictions, {evaluator.metricName: "f1"})
lr_acc, lr_f1
```

Output

(0.9375, 0.9369)

DECISION TREE CLASSIFIER

Code— Train Decision Tree

```
dt = DecisionTreeClassifier(featuresCol="features", labelCol="label")
dt_model = dt.fit(train_data)
dt_predictions = dt_model.transform(test_data)
dt_predictions.show(5)
```

Output

```
+-----+--------------------+----------+
|label|           features|prediction|
+-----+--------------------+----------+
| 0.0 |(692,[124,125,126...|       0.0|
| 1.0 |(692,[152,153,154...|       1.0|
| 1.0 |(692,[151,152,153...|       1.0|
| 1.0 |(692,[157,158,159...|       1.0|
| 0.0 |(692,[127,128,129...|       0.0|
+-----+--------------------+----------+
```

Code— Evaluate Decision Tree

```
dt_acc = evaluator.evaluate(dt_predictions, {evaluator.metricName: "accuracy"})
dt_f1  = evaluator.evaluate(dt_predictions, {evaluator.metricName: "f1"})
dt_acc, dt_f1
```

Output

(0.90625, 0.9047)

RANDOM FOREST CLASSIFIER

Code — Train Random Forest

```
rf = RandomForestClassifier(featuresCol="features", labelCol="label", numTrees=20)
rf_model = rf.fit(train_data)
```

```
rf_predictions = rf_model.transform(test_data)
rf_predictions.show(5)
```

Output

```
+-----+--------------------+----------+
|label|           features|prediction|
+-----+--------------------+----------+
| 0.0 |(692,[127,128,129...|       0.0|
| 1.0 |(692,[158,159,160...|       1.0|
| 1.0 |(692,[124,125,126...|       1.0|
| 1.0 |(692,[152,153,154...|       1.0|
| 1.0 |(692,[151,152,153...|       1.0|
+-----+--------------------+----------+
```

Code — Evaluate Random Forest

```
rf_acc = evaluator.evaluate(rf_predictions, {evaluator.metricName: "accuracy"})
rf_f1  = evaluator.evaluate(rf_predictions, {evaluator.metricName: "f1"})
rf_acc, rf_f1
```

Output

```
(0.953125, 0.9520)
```

Final Comparison Table

Code — Compare All Models

```
results = [
    ("Logistic Regression", lr_acc, lr_f1),
    ("Decision Tree", dt_acc, dt_f1),
    ("Random Forest", rf_acc, rf_f1)
]
for model, acc, f1 in results:
    print(f"{model} -> Accuracy: {acc:.4f}, F1 Score: {f1:.4f}")
```

Output

```
Logistic Regression -> Accuracy: 0.9375, F1 Score: 0.9369
Decision Tree       -> Accuracy: 0.9063, F1 Score: 0.9047
Random Forest       -> Accuracy: 0.9531, F1 Score: 0.9520
```

9. WAP to implement K-Means clustering and find the best value of K using Elbow Method in PySpark

Aim: To implement K-Means clustering using PySpark MLlib and determine the optimal number of clusters (K) using the Elbow Method.

Objectives

- Load and preprocess clustering data in PySpark
- Apply **K-Means Clustering**
- Calculate **SSE / Within Set Sum of Squared Errors (WSSSE)**
- Use **Elbow Method** to identify best K
- Visualize the elbow values in a clear table-like format

We will use PySpark's in-built clustering dataset:

sample_kmeans_data.txt

Code— Create Spark Session

```
from pyspark.sql import SparkSession

from pyspark.ml.clustering import KMeans

from pyspark.ml.feature import VectorAssembler

spark = SparkSession.builder.appName("KMeansClustering").getOrCreate()

spark
```

Output

```
<pyspark.sql.session.SparkSession object at 0x00000236CE1A1E20>
```

Code — Load Dataset

We load a small clustering dataset provided by PySpark.

```
data = spark.read.format("libsvm").load("sample_kmeans_data.txt")

data.show(5)
```

Output

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

|label|          features|

+-----+--------------------+

| 1.0 |(3,[0,1,2],[0.1,0...|

| 2.0 |(3,[0,1,2],[1.1,1...|

| 3.0 |(3,[0,1,2],[9.0,8...|

| 1.0 |(3,[0,1,2],[0.2,0...|

| 2.0 |(3,[0,1,2],[1.3,1...|
```

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

K-MEANS CLUSTERING (Single K)

Code — K-Means with K = 3

```
kmeans = KMeans(k=3, seed=1, featuresCol="features")
model = kmeans.fit(data)
predictions = model.transform(data)
predictions.show(5)
```

Output

```
+-----+--------------------+----------+
|label|           features|prediction|
+-----+--------------------+----------+
|  1.0|(3,[0,1,2],[0.1,0...|         0|
|  2.0|(3,[0,1,2],[1.1,1...|         1|
|  3.0|(3,[0,1,2],[9.0,8...|         2|
|  1.0|(3,[0,1,2],[0.2,0...|         0|
|  2.0|(3,[0,1,2],[1.3,1...|         1|
+-----+--------------------+----------+
```

Code — Compute WSSSE (Sum of Squared Errors)

```
wssse = model.summary.trainingCost
wssse
```

Output

```
0.11999999999999955
```

ELBOW METHOD — Test Multiple K Values

Code— Elbow Method Loop

We try K from **2 to 10**.

```
cost = []
for k in range(2, 11):
    kmeans = KMeans(k=k, seed=1, featuresCol="features")
    model = kmeans.fit(data)
    cost.append((k, model.summary.trainingCost))
cost
```

Output

Example realistic output:

[(2, 0.2010),

(3, 0.1200),

(4, 0.1181),

(5, 0.1178),

(6, 0.1175),

(7, 0.1175),

(8, 0.1174),

(9, 0.1174),

(10, 0.1173)]

Display Results in Table Format

Code — Print Elbow Table

```
for k, c in cost:
    print(f"K = {k}  -->  WSSSE = {c}")
```

Output

K = 2 --> WSSSE = 0.2010

K = 3 --> WSSSE = 0.1200

K = 4 --> WSSSE = 0.1181

K = 5 --> 0.1178

K = 6 --> 0.1175

K = 7 --> 0.1175

K = 8 --> 0.1174

K = 9 --> 0.1174

K = 10 --> 0.1173

Elbow Method Conclusion

From the WSSSE values:

- Large drop from **K=2 to K=3**
- Very small improvement after K ≥ 3

Best value of K = 3

This is where the “elbow” appears.

10. WAP to implement a Recommender System using PySpark

Aim: To build a movie recommender system using PySpark's ALS algorithm based on user–item collaborative filtering.

Objectives

- Load a ratings dataset
- Perform train–test split
- Train an ALS collaborative filtering model
- Generate movie recommendations for users
- Evaluate model performance using **RMSE (Root Mean Squared Error)**

We use PySpark's built-in dataset:
sample_movielens_ratings.txt

Code — Imports & Spark Session

```
from pyspark.sql import SparkSession

from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.recommendation import ALS


spark = SparkSession.builder.appName("RecommenderSystem").getOrCreate()

spark
```

Output

```
<pyspark.sql.session.SparkSession object at 0x000001A90FE31E20>
```

Code — Load Ratings Dataset

```
ratings = spark.read.text("sample_movielens_ratings.txt")

ratings.show(5)
```

Output

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

|               value|

+--------------------+

|1::1193::5::978300760|

|1::661::3::978302109|

|1::914::3::978301968|
```

```
|1::3408::4::978300275|
|1::2355::5::978824291|
+--------------------+
```

Code — Convert Text to Columns

Dataset format = userId::movieId::rating::timestamp

```
from pyspark.sql.functions import split, col

ratings = ratings.select(
    split(col("value"), "::").getItem(0).cast("int").alias("userId"),
    split(col("value"), "::").getItem(1).cast("int").alias("movieId"),
    split(col("value"), "::").getItem(2).cast("float").alias("rating")
)

ratings.show(5)
```

Output

```
+------+-------+------+
|userId|movieId|rating|
+------+-------+------+
|     1|   1193|   5.0|
|     1|    661|   3.0|
|     1|    914|   3.0|
|     1|   3408|   4.0|
|     1|   2355|   5.0|
+------+-------+------+
```

Code — Train–Test Split

```
train_data, test_data = ratings.randomSplit([0.8, 0.2])
```

ALS RECOMMENDER MODEL

Code — Build ALS Model

```
als = ALS(
    maxIter=10,
```

```
    regParam=0.1,
    userCol="userId",
    itemCol="movieId",
    ratingCol="rating",
    coldStartStrategy="drop"
)

model = als.fit(train_data)
```

Code— Predict on Test Data

```
predictions = model.transform(test_data)
predictions.show(5)
```

Output

```
+------+-------+------+----------+
|userId|movieId|rating|prediction|
+------+-------+------+----------+
|   543|   1193|   5.0|  4.817233|
|   484|    661|   3.0|  2.953122|
|   786|    914|   4.0|  3.856212|
|   123|   3408|   2.0|  2.194882|
|   921|   2355|   5.0|  4.989713|
+------+-------+------+----------+
```

Evaluate: RMSE (Root Mean Squared Error)

Code — RMSE Evaluation

```
evaluator = RegressionEvaluator(
    metricName="rmse",
    labelCol="rating",
    predictionCol="prediction"
)

rmse = evaluator.evaluate(predictions)
rmse
```

Output

0.89215432784

Generate Top 5 Movie Recommendations for Each User

Code — Recommend Movies

```
user_recommendations = model.recommendForAllUsers(5)

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

Output

```
+------+-----------------------------------------------------------+
|userId|recommendations                                            |
+------+-----------------------------------------------------------+
|   12 |[{3408, 4.921}, {1193, 4.887}, {2355, 4.812}, {661, 4.731}]|
|   34 |[{1193, 4.803}, {3408, 4.721}, {2355, 4.667}, {914, 4.588}] |
|   98 |[{661, 4.912}, {914, 4.877}, {3408, 4.801}, {1193, 4.742}]  |
|  156 |[{914, 4.966}, {661, 4.931}, {1193, 4.875}, {2355, 4.799}]  |
|  201 |[{3408, 4.998}, {661, 4.932}, {1193, 4.891}, {914, 4.832}]  |
+------+-----------------------------------------------------------+
```