COMPILED BY-
ISHAAN NAGAL
210962058
ROLL NO. 31
CSE AI ML-B

DECISION TREE

```
# Import necessary libraries
import pyspark
import os
import sys
from pyspark import SparkContext
from pyspark.sql import SparkSession

# Set up PySpark environment
os.environ['PYSPARK_PYTHON'] = sys.executable  # Setting Python executable
for PySpark
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable  # Setting driver
Python executable for PySpark
spark = SparkSession.builder.config("spark.driver.memory",
"16g").appName('chapter_4').getOrCreate()  # Creating a SparkSession with
configuration

# Load the dataset without header
data_without_header = spark.read.option("inferSchema",
True).option("header", False).csv("data/covtype.data")  # Loading data
without header

# Define the column names
colnames = ["Elevation", "Aspect", "Slope",   # Defining column names for
the dataset
            "Horizontal_Distance_To_Hydrology",
            "Vertical_Distance_To_Hydrology",
"Horizontal_Distance_To_Roadways",
            "Hillshade_9am", "Hillshade_Noon", "Hillshade_3pm",
            "Horizontal_Distance_To_Fire_Points"] +
            [f"Wilderness_Area_{i}" for i in range(4)] +
            [f"Soil_Type_{i}" for i in range(40)] +
            ["Cover_Type"]

# Create a DataFrame with column names and cast the label column to
DoubleType
data = data_without_header.toDF(*colnames).withColumn("Cover_Type",
col("Cover_Type").cast(DoubleType()))  # Creating DataFrame with specified
column names and data types

# Split the data into train and test sets
(train_data, test_data) = data.randomSplit([0.9, 0.1])  # Splitting data
into train and test sets
train_data.cache()  # Caching train data for optimization
test_data.cache()  # Caching test data for optimization

# Feature engineering
```

```
input_cols = colnames[:-1]  # Defining input columns for feature engineering
vector_assembler = VectorAssembler(inputCols=input_cols, outputCol="featureVector")  # Creating VectorAssembler for feature engineering
assembled_train_data = vector_assembler.transform(train_data)  # Transforming train data using VectorAssembler

# Train a Decision Tree Classifier
classifier = DecisionTreeClassifier(seed=1234, labelCol="Cover_Type", featuresCol="featureVector", predictionCol="prediction")  # Creating DecisionTreeClassifier
model = classifier.fit(assembled_train_data)  # Training the model

# Print feature importances
print(model.toDebugString)  # Printing the debug string of the model
pd.DataFrame(model.featureImportances.toArray(), index=input_cols, columns=['importance']).sort_values(by="importance", ascending=False)  # Displaying feature importances

# Make predictions and evaluate the model
predictions = model.transform(assembled_train_data)  # Making predictions on train data
evaluator = MulticlassClassificationEvaluator(labelCol="Cover_Type", predictionCol="prediction")  # Creating evaluator for model evaluation
print(f"Accuracy: {evaluator.setMetricName('accuracy').evaluate(predictions)}")  # Calculating and printing accuracy
print(f"F1-score: {evaluator.setMetricName('f1').evaluate(predictions)}")  # Calculating and printing F1-score
confusion_matrix = predictions.groupBy("Cover_Type").pivot("prediction", range(1,8)).count().na.fill(0.0).orderBy("Cover_Type")  # Calculating confusion matrix
confusion_matrix.show()  # Displaying confusion matrix

# Calculate class probabilities
def class_probabilities(data):
    total = data.count()
    return data.groupBy("Cover_Type").count().orderBy("Cover_Type").select(col("count").cast(DoubleType())).withColumn("count_proportion", col("count")/total).select("count_proportion").collect()

train_prior_probabilities = class_probabilities(train_data)  # Calculating train prior probabilities
test_prior_probabilities = class_probabilities(test_data)  # Calculating test prior probabilities
train_prior_probabilities = [p[0] for p in train_prior_probabilities]  # Extracting train prior probabilities
test_prior_probabilities = [p[0] for p in test_prior_probabilities]  # Extracting test prior probabilities
print(f"Sum of train and test prior probabilities: {sum([train_p * cv_p for train_p, cv_p in zip(train_prior_probabilities, test_prior_probabilities)])}")  # Calculating and printing sum of train and test prior probabilities

# Hyperparameter tuning
```

```python
assembler = VectorAssembler(inputCols=input_cols,
outputCol="featureVector")  # Creating VectorAssembler for hyperparameter
tuning
classifier = DecisionTreeClassifier(seed=1234, labelCol="Cover_Type",
featuresCol="featureVector", predictionCol="prediction")  # Creating
DecisionTreeClassifier for hyperparameter tuning
pipeline = Pipeline(stages=[assembler, classifier])  # Creating pipeline
for hyperparameter tuning
paramGrid = ParamGridBuilder().addGrid(classifier.impurity, ["gini",
"entropy"]).addGrid(classifier.maxDepth, [1,
20]).addGrid(classifier.maxBins, [40, 300]).addGrid(classifier.minInfoGain,
[0.0, 0.05]).build()  # Creating parameter grid for hyperparameter tuning
multiclassEval =
MulticlassClassificationEvaluator().setLabelCol("Cover_Type").setPrediction
Col("prediction").setMetricName("accuracy")  # Creating evaluator for model
evaluation in hyperparameter tuning
validator = TrainValidationSplit(seed=1234, estimator=pipeline,
evaluator=multiclassEval, estimatorParamMaps=paramGrid, trainRatio=0.9)  #
Creating TrainValidationSplit for hyperparameter tuning
validator_model = validator.fit(train_data)  # Fitting TrainValidationSplit
on train data
best_model = validator_model.bestModel  # Extracting best model from
TrainValidationSplit
print(best_model.stages[1].extractParamMap())  # Printing best model
parameters
print(f"Best model accuracy on validation set: {metrics[0]}")  # Printing
best model accuracy on validation set
print(f"Best model accuracy on test set:
{multiclassEval.evaluate(best_model.transform(test_data))}")  # Calculating
and printing best model accuracy on test set

# One-hot encoding
def unencode_one_hot(data):
    wilderness_cols = ['Wilderness_Area_' + str(i) for i in range(4)]  #
Defining wilderness area columns
    wilderness_assembler =
VectorAssembler().setInputCols(wilderness_cols).setOutputCol("wilderness")
# Creating VectorAssembler for wilderness area
    unhot_udf = udf(lambda v: v.toArray().tolist().index(1))  # Creating
UDF for unencoding one-hot vectors
    with_wilderness =
wilderness_assembler.transform(data).drop(*wilderness_cols).withColumn("wil
derness", unhot_udf(col("wilderness")).cast(IntegerType()))  # Unencoding
wilderness area
    soil_cols = ['Soil_Type_' + str(i) for i in range(40)]  # Defining soil
type columns
    soil_assembler =
VectorAssembler().setInputCols(soil_cols).setOutputCol("soil")  # Creating
VectorAssembler for soil type
    with_soil =
soil_assembler.transform(with_wilderness).drop(*soil_cols).withColumn("soil
", unhot_udf(col("soil")).cast(IntegerType()))  # Unencoding soil type
    return with_soil  # Returning unencoded DataFrame

unenc_train_data = unencode_one_hot(train_data)  # Unencoding one-hot
vectors in train data
```

```python
unenc_test_data = unencode_one_hot(test_data)  # Unencoding one-hot vectors in test data

# Random Forest Classifier
cols = unenc_train_data.columns  # Extracting columns from unencoded train data
input_cols = [c for c in cols if c!='Cover_Type']  # Defining input columns for Random Forest Classifier
assembler = VectorAssembler().setInputCols(input_cols).setOutputCol("featureVector")  # Creating VectorAssembler for Random Forest Classifier
indexer = VectorIndexer().setMaxCategories(40).setInputCol("featureVector").setOutputCol("indexedVector")  # Creating VectorIndexer for Random Forest Classifier
classifier = RandomForestClassifier(seed=1234, labelCol="Cover_Type", featuresCol="indexedVector", predictionCol="prediction")  # Creating RandomForestClassifier
pipeline = Pipeline().setStages([assembler, indexer, classifier])  # Creating pipeline for Random Forest Classifier
paramGrid = ParamGridBuilder().addGrid(classifier.impurity, ["gini", "entropy"]).addGrid(classifier.maxDepth, [1, 20]).addGrid(classifier.maxBins, [40, 300]).addGrid(classifier.minInfoGain, [0.0, 0.05]).build()  # Creating parameter grid for Random Forest Classifier
multiclassEval = MulticlassClassificationEvaluator().setLabelCol("Cover_Type").setPredictionCol("prediction").setMetricName("accuracy")  # Creating evaluator for model evaluation in Random Forest Classifier
validator = TrainValidationSplit(seed=1234, estimator=pipeline, evaluator=multiclassEval, estimatorParamMaps=paramGrid, trainRatio=0.9)  # Creating TrainValidationSplit for Random Forest Classifier
validator_model = validator.fit(unenc_train_data)  # Fitting TrainValidationSplit on unencoded train data
best_model = validator_model.bestModel  # Extracting best model from TrainValidationSplit
forest_model = best_model.stages[2]  # Extracting Random Forest model from best model
feature_importance_list = list(zip(input_cols, forest_model.featureImportances.toArray()))  # Creating list of feature importances
feature_importance_list.sort(key=lambda x: x[1], reverse=True)  # Sorting feature importances
print("Feature importances:")  # Printing feature importances
pprint(feature_importance_list)  # Pretty-printing feature importances

# Make predictions on the test set using the best Random Forest model
best_model.transform(unenc_test_data.drop("Cover_Type")).select("prediction").show(1)  # Making predictions on test data using best Random Forest model and displaying predictions
```

ENTITY RESOLUTION

```
import pyspark  # Import PySpark library
import os  # Import os module for operating system functionalities
import sys  # Import sys module for system-specific parameters
from pyspark.sql import SparkSession  # Import SparkSession from PySpark

# Set up PySpark environment
os.environ['PYSPARK_PYTHON'] = sys.executable  # Setting Python executable
for PySpark
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable  # Setting driver
Python executable for PySpark
spark = SparkSession.builder.config("spark.driver.memory",
"16g").appName('chapter_2').getOrCreate()  # Creating a SparkSession with
configuration

# Read the CSV files
prev = spark.read.csv("data/linkage/donation/block_1/block_1.csv")  #
Reading a CSV file without header
prev.show(2)  # Displaying the first two rows of the DataFrame

# Read and parse CSV with options
parsed = spark.read.option("header", "true").option("nullValue",
"?").option("inferSchema",
"true").csv("data/linkage/donation/block_1/block_1.csv")  # Reading and
parsing a CSV file with options
parsed.printSchema()  # Printing the schema of the DataFrame
parsed.show(5)  # Displaying the first five rows of the DataFrame

# Count rows and cache DataFrame
parsed.count()  # Counting the number of rows in the DataFrame
parsed.cache()  # Caching the DataFrame for optimization

# Group by 'is_match' column and show counts
from pyspark.sql.functions import col  # Importing col function from
PySpark
parsed.groupBy("is_match").count().orderBy(col("count").desc()).show()  #
Grouping by 'is_match' column and showing counts in descending order

# Create temporary view for SQL queries
parsed.createOrReplaceTempView("linkage")  # Creating a temporary view for
SQL queries
spark.sql("""
SELECT is_match, COUNT(*) cnt
FROM linkage
GROUP BY is_match
ORDER BY cnt DESC
""").show()  # Executing a SQL query and displaying the result
```

```
# Summary statistics and operations
summary = parsed.describe()  # Generating summary statistics for the DataFrame
summary.select("summary", "cmp_fname_c1", "cmp_fname_c2").show()  # Selecting specific columns from the summary DataFrame and showing them

# Filtering and describing matches and misses
matches = parsed.where("is_match = true")  # Filtering rows where 'is_match' is true
match_summary = matches.describe()  # Generating summary statistics for matches

misses = parsed.filter(col("is_match") == False)  # Filtering rows where 'is_match' is false
miss_summary = misses.describe()  # Generating summary statistics for misses

# Converting summary to Pandas for operations
summary_p = summary.toPandas()  # Converting the summary DataFrame to Pandas DataFrame
summary_p.head()  # Displaying the first few rows of the Pandas DataFrame
summary_p.shape  # Displaying the shape of the Pandas DataFrame

# Transpose and convert back to Spark DataFrame
summary_p = summary_p.set_index('summary').transpose().reset_index()  # Transposing the Pandas DataFrame and resetting the index
summary_p = summary_p.rename(columns={'index':'field'})  # Renaming columns of the DataFrame
summary_p = summary_p.rename_axis(None, axis=1)  # Removing the axis name from the DataFrame
summary_p.shape  # Displaying the shape of the DataFrame

summaryT = spark.createDataFrame(summary_p)  # Creating a Spark DataFrame from the Pandas DataFrame
summaryT.printSchema()  # Printing the schema of the Spark DataFrame

# Convert metric columns to DoubleType
from pyspark.sql.types import DoubleType  # Importing DoubleType from PySpark
for c in summaryT.columns:  # Looping through columns of the DataFrame
    if c == 'field':  # Skipping the 'field' column
        continue
    summaryT = summaryT.withColumn(c, summaryT[c].cast(DoubleType()))  # Converting columns to DoubleType

summaryT.printSchema()  # Printing the schema of the DataFrame

# Define a function for pivoting and converting summary
def pivot_summary(desc):  # Defining a function for pivoting and converting summary
    desc_p = desc.toPandas()  # Converting Spark DataFrame to Pandas DataFrame
    desc_p = desc_p.set_index('summary').transpose().reset_index()  # Transposing and resetting index of the Pandas DataFrame
    desc_p = desc_p.rename(columns={'index':'field'})  # Renaming columns of the DataFrame
```

```
    desc_p = desc_p.rename_axis(None, axis=1)  # Removing the axis name
from the DataFrame
    descT = spark.createDataFrame(desc_p)  # Creating a Spark DataFrame
from the Pandas DataFrame
    for c in descT.columns:  # Looping through columns of the DataFrame
        if c == 'field':  # Skipping the 'field' column
            continue
        else:
            descT = descT.withColumn(c, descT[c].cast(DoubleType()))  #
Converting columns to DoubleType
    return descT  # Returning the converted DataFrame

match_summaryT = pivot_summary(match_summary)  # Pivoting and converting
match summary
miss_summaryT = pivot_summary(miss_summary)  # Pivoting and converting miss
summary

match_summaryT.createOrReplaceTempView("match_desc")  # Creating a
temporary view for match summary
miss_summaryT.createOrReplaceTempView("miss_desc")  # Creating a temporary
view for miss summary

# SQL query for comparing matches and misses
spark.sql("""
SELECT a.field, a.count + b.count total, a.mean - b.mean delta
FROM match_desc a INNER JOIN miss_desc b ON a.field = b.field
WHERE a.field NOT IN ("id_1", "id_2")
ORDER BY delta DESC, total DESC
""")  # Executing a SQL query for comparing matches and misses

# Good features and scoring
good_features = ["cmp_lname_c1", "cmp_plz", "cmp_by", "cmp_bd", "cmp_bm"]
# Defining good features for scoring
sum_expression = " + ".join(good_features)  # Joining good features with a
sum expression

from pyspark.sql.functions import expr  # Importing expr function from
PySpark
scored = parsed.fillna(0, subset=good_features).withColumn('score',
expr(sum_expression)).select('score', 'is_match')  # Filling NA values,
scoring, and selecting columns
scored.show()  # Displaying the scored DataFrame

# Cross-tabulation function
def crossTabs(scored: DataFrame, t: DoubleType) -> DataFrame:  # Defining a
function for cross-tabulation
    return scored.selectExpr(f"score >= {t} as above",
"is_match").groupBy("above").pivot("is_match", ("true", "false")).count()
# Performing cross-tabulation

crossTabs(scored, 4.0).show()  # Performing cross-tabulation with threshold
4.0
crossTabs(scored, 2.0).show()  # Performing cross-tabulation with threshold
2.0
```

K-Means

```python
  # Import necessary libraries
import pyspark  # Importing PySpark for distributed computing
import os  # Operating System module for environment setup
import sys  # System-specific parameters and functions
from pyspark import SparkContext  # SparkContext for controlling Spark 
functionality
from pyspark.sql import SparkSession  # SparkSession for working with 
structured data
from pyspark.sql.functions import col  # Column functions for data 
manipulation
from pyspark.ml.feature import VectorAssembler, StandardScaler, 
OneHotEncoder, StringIndexer  # ML feature engineering tools
from pyspark.ml.clustering import KMeans  # KMeans clustering algorithm 
from MLlib
from pyspark.ml import Pipeline  # Pipeline for ML workflow management
from random import randint  # Random number generator
from math import log  # Logarithmic function for entropy calculation
from pyspark.sql import functions as fun  # Alias for SQL functions in 
Spark
from pyspark.sql import Window  # Window functions for data aggregation and 
manipulation

# Set up Spark session
os.environ['PYSPARK_PYTHON'] = sys.executable  # Set Python executable for 
PySpark
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable  # Set Python 
executable for PySpark driver
spark = SparkSession.builder.config("spark.driver.memory", 
"16g").appName('chapter_5').getOrCreate()  # Configure Spark session

# Read data and define column names
data_without_header = spark.read.option("inferSchema", 
True).option("header", False).csv("data/kddcup.data_10_percent_corrected") 
# Read CSV data without inferring schema
column_names = [ "duration", "protocol_type", "service", "flag", 
"src_bytes", "dst_bytes", "land", "wrong_fragment", "urgent", "hot", 
"num_failed_logins", "logged_in", "num_compromised", "root_shell", 
"su_attempted", "num_root", "num_file_creations", "num_shells", 
"num_access_files", "num_outbound_cmds", "is_host_login", "is_guest_login", 
"count", "srv_count", "serror_rate", "srv_serror_rate", "rerror_rate", 
"srv_rerror_rate", "same_srv_rate", "diff_srv_rate", "srv_diff_host_rate", 
"dst_host_count", "dst_host_srv_count", "dst_host_same_srv_rate", 
"dst_host_diff_srv_rate", "dst_host_same_src_port_rate", 
"dst_host_srv_diff_host_rate", "dst_host_serror_rate", 
"dst_host_srv_serror_rate", "dst_host_rerror_rate", 
"dst_host_srv_rerror_rate", "label"]  # Define column names for the data
data = data_without_header.toDF(*column_names)  # Create DataFrame with 
specified column names
```

```python
# Display count of each label
data.select("label").groupBy("label").count().orderBy(col("count").desc()).
show(25)  # Group by label, count occurrences, and show top 25 counts

# Define functions for clustering and evaluation

# Function to calculate clustering score based on training cost
def clustering_score(input_data, k):
    # Prepare data
    input_numeric_only = input_data.drop("protocol_type", "service",
"flag")  # Exclude non-numeric columns
    assembler =
VectorAssembler().setInputCols(input_numeric_only.columns[:-
1]).setOutputCol("featureVector")  # Assemble feature vector
    kmeans =
KMeans().setSeed(randint(100,100000)).setK(k).setPredictionCol("cluster").s
etFeaturesCol("featureVector")  # Define KMeans algorithm
    pipeline = Pipeline().setStages([assembler, kmeans])  # Create ML
pipeline
    # Fit pipeline and get training cost
    pipeline_model = pipeline.fit(input_numeric_only)
    kmeans_model = pipeline_model.stages[-1]
    training_cost = kmeans_model.summary.trainingCost
    return training_cost

# Function to calculate clustering score with additional parameters
def clustering_score_1(input_data, k):
    # Prepare data
    input_numeric_only = input_data.drop("protocol_type", "service",
"flag")  # Exclude non-numeric columns
    assembler =
VectorAssembler().setInputCols(input_numeric_only.columns[:-
1]).setOutputCol("featureVector")  # Assemble feature vector
    kmeans =
KMeans().setSeed(randint(100,100000)).setK(k).setMaxIter(40).setTol(1.0e-
5).setPredictionCol("cluster").setFeaturesCol("featureVector")  # Define
KMeans algorithm with additional parameters
    pipeline = Pipeline().setStages([assembler, kmeans])  # Create ML
pipeline
    # Fit pipeline and get training cost
    pipeline_model = pipeline.fit(input_numeric_only)
    kmeans_model = pipeline_model.stages[-1]
    training_cost = kmeans_model.summary.trainingCost
    return training_cost

# Function to calculate clustering score with standard scaling
def clustering_score_2(input_data, k):
    # Prepare data
    input_numeric_only = input_data.drop("protocol_type", "service",
"flag")  # Exclude non-numeric columns
    assembler =
VectorAssembler().setInputCols(input_numeric_only.columns[:-
1]).setOutputCol("featureVector")  # Assemble feature vector
    scaler =
StandardScaler().setInputCol("featureVector").setOutputCol("scaledFeatureVe
ctor").setWithStd(True).setWithMean(False)  # Scale features
```

```
    kmeans =
KMeans().setSeed(randint(100,100000)).setK(k).setMaxIter(40).setTol(1.0e-
5).setPredictionCol("cluster").setFeaturesCol("scaledFeatureVector")  #
Define KMeans algorithm with scaled features
    pipeline = Pipeline().setStages([assembler, scaler, kmeans])  # Create
ML pipeline
    # Fit pipeline and get training cost
    pipeline_model = pipeline.fit(input_numeric_only)
    kmeans_model = pipeline_model.stages[-1]
    training_cost = kmeans_model.summary.trainingCost
    return training_cost
from pyspark.ml.feature import OneHotEncoder, StringIndexer
def one_hot_pipeline(input_col):

# Function to calculate clustering score with one-hot encoding
def clustering_score_3(input_data, k):
    # Prepare data
    proto_type_pipeline, proto_type_vec_col =
one_hot_pipeline("protocol_type")  # One-hot encode protocol_type column
    service_pipeline, service_vec_col = one_hot_pipeline("service")  # One-
hot encode service column
    flag_pipeline, flag_vec_col = one_hot_pipeline("flag")  # One-hot
encode flag column
    assemble_cols = set(input_data.columns) - {"label", "protocol_type",
"service", "flag"} | {proto_type_vec_col, service_vec_col, flag_vec_col}  #
Combine columns
    assembler =
VectorAssembler().setInputCols(list(assemble_cols)).setOutputCol("featureVe
ctor")  # Assemble feature vector
    scaler =
StandardScaler().setInputCol("featureVector").setOutputCol("scaledFeatureVe
ctor").setWithStd(True).setWithMean(False)  # Scale features
    kmeans =
KMeans().setSeed(randint(100,100000)).setK(k).setMaxIter(40).setTol(1.0e-
5).setPredictionCol("cluster").setFeaturesCol("scaledFeatureVector")  #
Define KMeans algorithm with scaled features
    pipeline = Pipeline().setStages([proto_type_pipeline, service_pipeline,
flag_pipeline, assembler, scaler, kmeans])  # Create ML pipeline
    # Fit pipeline and get training cost
    pipeline_model = pipeline.fit(input_data)
    kmeans_model = pipeline_model.stages[-1]
    training_cost = kmeans_model.summary.trainingCost
    return training_cost

# Function to calculate entropy
def entropy(counts):
    values = [c for c in counts if (c > 0)]  # Filter non-zero counts
    n = sum(values)  # Calculate total count
    p = [v/n for v in values]  # Calculate probabilities
    return sum([-1*(p_v) * log(p_v) for p_v in p])  # Calculate entropy

# Function to fit pipeline for clustering with one-hot encoding
def fit_pipeline_4(data, k):
    (proto_type_pipeline, proto_type_vec_col) =
one_hot_pipeline("protocol_type")  # One-hot encode protocol_type column
    (service_pipeline, service_vec_col) = one_hot_pipeline("service")  #
One-hot encode service column
```

```
    (flag_pipeline, flag_vec_col) = one_hot_pipeline("flag")  # One-hot
encode flag column
    assemble_cols = set(data.columns) - {"label", "protocol_type",
"service", "flag"} | {proto_type_vec_col, service_vec_col, flag_vec_col}  #
Combine columns
    assembler = VectorAssembler(inputCols=list(assemble_cols),
outputCol="featureVector")  # Assemble feature vector
    scaler = StandardScaler(inputCol="featureVector",
outputCol="scaledFeatureVector", withStd=True, withMean=False)  # Scale
features
    kmeans_model = KMeans(seed=randint(100,100000), k=k, maxIter=40,
tol=1.0e-5, predictionCol="cluster", featuresCol="scaledFeatureVector")  #
Define KMeans algorithm with scaled features
    pipeline = Pipeline(stages=[proto_type_pipeline, service_pipeline,
flag_pipeline, assembler, scaler, kmeans_model])  # Create ML pipeline
    pipeline_model = pipeline.fit(data)  # Fit pipeline
    return pipeline_model
# Define function to calculate clustering score based on training cost
def clustering_score_4(input_data, k):
    # Fit the ML pipeline for clustering
    pipeline_model = fit_pipeline_4(input_data, k)

    # Perform clustering and label analysis
    cluster_label = pipeline_model.transform(input_data).select("cluster",
"label")  # Transform data and select relevant columns
    df = cluster_label.groupBy("cluster",
"label").count().orderBy("cluster")  # Group by cluster and label, count
occurrences, and order by cluster

    # Calculate probabilities and entropy for each cluster
    w = Window.partitionBy("cluster")  # Define window for partitioning
    p_col = df['count'] / fun.sum(df['count']).over(w)  # Calculate
probabilities within each cluster
    with_p_col = df.withColumn("p_col", p_col)  # Add probability column to
DataFrame
    result = with_p_col.groupBy("cluster").agg(-fun.sum(col("p_col") *
fun.log2(col("p_col"))).alias("entropy"),  # Calculate entropy for each
cluster

fun.sum(col("count")).alias("cluster_size"))  # Calculate cluster size
    result = result.withColumn('weightedClusterEntropy', col('entropy') *
col('cluster_size'))  # Calculate weighted cluster entropy
    weighted_cluster_entropy_avg =
result.agg(fun.sum(col('weightedClusterEntropy'))).collect()  # Calculate
average weighted cluster entropy
    return weighted_cluster_entropy_avg[0][0] / input_data.count()  #
Return normalized clustering score

# Fit the ML pipeline for clustering with k=180
pipeline_model = fit_pipeline_4(data, 180)

# Perform clustering and label analysis
count_by_cluster_label = pipeline_model.transform(data).select("cluster",
"label").groupBy("cluster", "label").count().orderBy("cluster", "label")
count_by_cluster_label.show()
  # Display the result of clustering and label analysis
```

MONTE-CARLO

```
    # Import necessary libraries
import pyspark
import os
import sys
from pyspark import SparkContext

# Set environment variables for PySpark
os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable

# Import SparkSession from PySpark SQL module
from pyspark.sql import SparkSession

# Create a SparkSession with specified configuration
spark = SparkSession.builder.config("spark.driver.memory",
"16g").appName('chapter_8').getOrCreate()

# Read CSV files into a PySpark DataFrame
stocks =
spark.read.csv(["data/stocksA/ABAX.csv","data/stocksA/AAME.csv","data/stock
sA/AEPI.csv"], header='true', inferSchema='true')

# Show the first two rows of the DataFrame
stocks.show(2)

# Import SQL functions from PySpark
from pyspark.sql import functions as fun

# Add a column 'Symbol' to the DataFrame based on the file name
stocks = stocks.withColumn("Symbol", fun.input_file_name()).
withColumn("Symbol", fun.element_at(fun.split("Symbol", "/"), -1)).
withColumn("Symbol", fun.element_at(fun.split("Symbol", "."), 1))

# Show the first two rows of the updated DataFrame
stocks.show(2)

# Read the same CSV files into another DataFrame
factors =
spark.read.csv(["data/stocksA/ABAX.csv","data/stocksA/AAME.csv","data/stock
sA/AEPI.csv"], header='true', inferSchema='true')

# Add a column 'Symbol' to the DataFrame based on the file name
factors = factors.withColumn("Symbol", fun.input_file_name()).
withColumn("Symbol", fun.element_at(fun.split("Symbol", "/"), -1)).
withColumn("Symbol", fun.element_at(fun.split("Symbol", "."), 1))

# Import Window function from PySpark
```

```python
from pyspark.sql import Window

# Filter the stocks DataFrame based on a condition
stocks = stocks.withColumn('count', fun.count('Symbol').
over(Window.partitionBy('Symbol'))).
filter(fun.col('count') > 260*5 + 10)

# Set a configuration for parsing date/time strings
spark.sql("set spark.sql.legacy.timeParserPolicy=LEGACY")

# Convert the 'Date' column to date format
stocks = stocks.withColumn('Date',
fun.to_date(fun.to_timestamp(fun.col('Date'), 'dd-MMM-yy')))

# Print the schema of the DataFrame
stocks.printSchema()

# Import datetime module
from datetime import datetime

# Filter the stocks DataFrame based on date range
stocks = stocks.filter(fun.col('Date') >= datetime(2009, 10, 23)).
filter(fun.col('Date') <= datetime(2014, 10, 23))

# Convert the 'Date' column to date format and filter based on date range
factors = factors.withColumn('Date',
fun.to_date(fun.to_timestamp(fun.col('Date'), 'dd-MMM-yy')))
factors = factors.filter(fun.col('Date') >= datetime(2009, 10, 23)).
filter(fun.col('Date') <= datetime(2014, 10, 23))

# Convert PySpark DataFrames to pandas DataFrames
stocks_pd_df = stocks.toPandas()
factors_pd_df = factors.toPandas()

# Print the first 5 rows of the factors_pd_df DataFrame
factors_pd_df.head(5)

# Set the number of steps for rolling window calculation
n_steps = 10

# Define a function to calculate returns
def my_fun(x):
    return ((x.iloc[-1] - x.iloc[0]) / x.iloc[0])

# Calculate stock returns and factor returns using rolling window
stock_returns =
stocks_pd_df.groupby('Symbol').Close.rolling(window=n_steps).apply(my_fun)
factors_returns =
factors_pd_df.groupby('Symbol').Close.rolling(window=n_steps).apply(my_fun)

# Reset and sort the index of the returns DataFrames
stock_returns =
stock_returns.reset_index().sort_values('level_1').reset_index()
factors_returns =
factors_returns.reset_index().sort_values('level_1').reset_index()

# Add stock returns to the stocks DataFrame
```

```
stocks_pd_df_with_returns =
stocks_pd_df.assign(stock_returns=stock_returns['Close'])

# Add factor returns and squared factor returns to the factors DataFrame
factors_pd_df_with_returns =
factors_pd_df.assign(factors_returns=factors_returns['Close'],
factors_returns_squared=factors_returns['Close']**2)

# Pivot the factors DataFrame to have factors and squared factors as
columns
factors_pd_df_with_returns = factors_pd_df_with_returns.pivot(index='Date',
columns='Symbol', values=['factors_returns', 'factors_returns_squared'])

# Modify the column names of the pivoted DataFrame
factors_pd_df_with_returns.columns =
factors_pd_df_with_returns.columns.to_series().str.join('_').reset_index()[
0]
factors_pd_df_with_returns = factors_pd_df_with_returns.reset_index()

# Print the first row and column names of the pivoted DataFrame
print(factors_pd_df_with_returns.head(1))
print(factors_pd_df_with_returns.columns)

# Import necessary libraries
import pandas as pd
from sklearn.linear_model import LinearRegression

# Merge the stocks and factors DataFrames
stocks_factors_combined_df = pd.merge(stocks_pd_df_with_returns,
factors_pd_df_with_returns, how="left", on="Date")

# Get the feature column names
feature_columns = list(stocks_factors_combined_df.columns[-6:])

# Drop rows with NaN values in feature columns and stock returns
with pd.option_context('mode.use_inf_as_na', True):
    stocks_factors_combined_df =
stocks_factors_combined_df.dropna(subset=feature_columns +
['stock_returns'])

# Function to find OLS coefficients
def find_ols_coef(df):
    y = df[['stock_returns']].values
    X = df[feature_columns]
    regr = LinearRegression()
    regr_output = regr.fit(X, y)
    return list(df[['Symbol']].values[0]) + list(regr_output.coef_[0])

# Calculate OLS coefficients for each stock
coefs_per_stock =
stocks_factors_combined_df.groupby('Symbol').apply(find_ols_coef)
coefs_per_stock = pd.DataFrame(coefs_per_stock).reset_index()
coefs_per_stock.columns = ['symbol', 'factor_coef_list']
coefs_per_stock = pd.DataFrame(coefs_per_stock.factor_coef_list.tolist(),
index=coefs_per_stock.index, columns=['Symbol'] + feature_columns)

# Print the coefficients DataFrame
```

```python
coefs_per_stock

# Select a sample of factor returns
samples = factors_returns.loc[factors_returns.Symbol ==
factors_returns.Symbol.unique()[0]]['Close']

# Plot the kernel density estimate for the sample
samples.plot.kde()

# Select factor returns for the first three unique symbols
f_1 = factors_returns.loc[factors_returns.Symbol ==
factors_returns.Symbol.unique()[0]]['Close']
f_2 = factors_returns.loc[factors_returns.Symbol ==
factors_returns.Symbol.unique()[1]]['Close']
f_3 = factors_returns.loc[factors_returns.Symbol ==
factors_returns.Symbol.unique()[2]]['Close']

# Print the sizes of the factor return series
print(f_1.size, len(f_2), f_3.size)

# Calculate correlation between the factor return series
pd.DataFrame({'f1': list(f_1)[1:1040], 'f2': list(f_2)[1:1040], 'f3':
list(f_3)}).corr()
# Calculate covariance matrix for factor returns
factors_returns_cov = pd.DataFrame({'f1': list(f_1)[1:1040], 'f2':
list(f_2)[1:1040], 'f3': list(f_3)}).cov().to_numpy()

# Calculate mean for factor returns
factors_returns_mean = pd.DataFrame({'f1': list(f_1)[1:1040], 'f2':
list(f_2)[1:1040], 'f3': list(f_3)}).mean()

# Import multivariate_normal function from numpy.random
from numpy.random import multivariate_normal

# Generate random samples from the multivariate normal distribution
multivariate_normal(factors_returns_mean, factors_returns_cov)

# Broadcast the coefficients DataFrame and other variables across the
cluster
b_coefs_per_stock = spark.sparkContext.broadcast(coefs_per_stock)
b_feature_columns = spark.sparkContext.broadcast(feature_columns)
b_factors_returns_mean = spark.sparkContext.broadcast(factors_returns_mean)
b_factors_returns_cov = spark.sparkContext.broadcast(factors_returns_cov)

# Define parallelism and number of trials
from pyspark.sql.types import IntegerType
parallelism = 1000
num_trials = 1000000
base_seed = 1496

# Create a PySpark DataFrame with seeds
seeds = [b for b in range(base_seed, base_seed + parallelism)]
seedsDF = spark.createDataFrame(seeds, IntegerType())
seedsDF = seedsDF.repartition(parallelism)

# Import necessary libraries
import random
```

```python
from numpy.random import seed
from pyspark.sql.types import LongType, ArrayType, DoubleType
from pyspark.sql.functions import udf

# Define a function to calculate trial returns
def calculate_trial_return(x):
    trial_return_list = []
    for i in range(int(num_trials/parallelism)):
        random_int = random.randint(0, num_trials*num_trials)
        seed(x)
        random_factors = multivariate_normal(b_factors_returns_mean.value, 
b_factors_returns_cov.value)
        coefs_per_stock_df = b_coefs_per_stock.value
        returns_per_stock = (coefs_per_stock_df[b_feature_columns.value] * 
(list(random_factors) + list(random_factors**2)))
        trial_return_list.append(float(returns_per_stock.sum(axis=1).sum() 
/ b_coefs_per_stock.value.size))
    return trial_return_list

# Create a User Defined Function (UDF) from the calculate_trial_return 
function
udf_return = udf(calculate_trial_return, ArrayType(DoubleType()))

# Apply the UDF to a PySpark DataFrame to calculate trial returns
from pyspark.sql.functions import col, explode
trials = seedsDF.withColumn("trial_return", udf_return(col("value")))
trials = trials.select('value', 
explode('trial_return').alias('trial_return'))

# Cache the trials DataFrame for faster access
trials.cache()

# Find the 5th percentile of trial returns
trials.approxQuantile('trial_return', [0.05], 0.0)

# Calculate the average of the lowest 5% trial returns
trials.orderBy(col('trial_return').asc()).
limit(int(trials.count()/20)).
agg(fun.avg(col("trial_return"))).show()

# Convert trials DataFrame to pandas DataFrame and plot the distribution of 
returns
import pandas
mytrials=trials.toPandas()
mytrials.plot.line()
```

MOVIE RECOMMENDOR

```
# Import necessary libraries
import os
import sys
import pyspark as ps
import warnings
from pyspark.sql import SQLContext

# Set environment variables for PySpark
os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable

# Try to create a SparkContext, or print a warning if it already exists
try:
    # Create SparkContext using all available CPUs
    sc = ps.SparkContext('local[*]')
    # Uncomment the line below if you also need SQLContext
    # sqlContext = SQLContext(sc)
    print("Just created a SparkContext")
except ValueError:
    warnings.warn("SparkContext already exists in this scope")

# Import unittest library for writing and running tests
import unittest
import sys

# Define a test case class for testing RDD operations
class TestRdd(unittest.TestCase):
    # Define a test method for the 'take' operation on RDDs
    def test_take(self):
        # Create an RDD containing the numbers 1, 2, 3, 4
        input = sc.parallelize([1, 2, 3, 4])
        # Test if calling take(4) on the RDD returns [1, 2, 3, 4]
        self.assertEqual([1, 2, 3, 4], input.take(4))

# Define a function to run the tests
def run_tests():
    # Load the test case suite from the TestRdd class
    suite = unittest.TestLoader().loadTestsFromTestCase(TestRdd)
    # Run the tests and display results with verbosity level 1
    unittest.TextTestRunner(verbosity=1, stream=sys.stderr).run(suite)

# Call the run_tests function to execute the tests
run_tests()
import json

# Define the fields for different types of entries in the JSON data
fields = ['product_id', 'user_id', 'score', 'time']
fields2 = ['product_id', 'user_id', 'review', 'profile_name',
'helpfulness', 'score', 'time']
fields3 = ['product_id', 'user_id', 'time']
```

```python
fields4 = ['user_id', 'score', 'time']

# Function to validate if all required fields are present in an entry
def validate(line):
    for field in fields2:  # Check against fields2 as it seems to be the
most detailed set
        if field not in line:
            return False
    return True

# Load the raw reviews data from a file and filter it based on validation
reviews_raw = sc.textFile('data/movies.json')
reviews = reviews_raw.map(lambda line: json.loads(line)).filter(validate)
reviews.cache()  # Cache the RDD for efficiency

# Count the number of unique movies, users, and total entries in the
dataset
num_movies = reviews.groupBy(lambda entry: entry['product_id']).count()
num_users = reviews.groupBy(lambda entry: entry['user_id']).count()
num_entries = reviews.count()
print(str(num_entries) + " reviews of " + str(num_movies) + " movies by " +
str(num_users) + " different people.")

# Calculate average number of movies watched per user and top movies by
watch count
r1 = reviews.map(lambda r: ((r['product_id'],), 1))  # Map each movie to
count 1
avg3 = r1.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] +
y[0], x[1] + y[1]))  # Reduce to sum and count
avg3 = avg3.filter(lambda x: x[1][1] > 20)  # Filter movies with less than
20 views
avg3 = avg3.map(lambda x: ((x[1][0] + x[1][1],),
x[0])).sortByKey(ascending=False)  # Sort by view count

# Print top movies by watch count
for movie in avg3.take(10):
    print("http://www.amazon.com/dp/" + movie[1][0] + " WATCHED BY : " +
str(movie[0][0]) + " PEOPLE")

# Calculate average number of movies watched by each user and top users by
movie count
r2 = reviews.map(lambda ru: ((ru['user_id'],), 1))  # Map each user to
count 1
avg2 = r2.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] +
y[0], x[1] + y[1]))  # Reduce to sum and count
avg2 = avg2.filter(lambda x: x[1][1] > 20)  # Filter users with less than
20 reviews
avg2 = avg2.map(lambda x: ((x[1][0] + x[1][1],),
x[0])).sortByKey(ascending=False)  # Sort by review count

# Print top users by movie count
for user in avg2.take(10):
    print("User " + user[1][0] + " WATCHED : " + str(user[0][0]) + "
MOVIES")

# Filter entries with "George" in the profile name and print details
filtered = reviews.filter(lambda entry: "George" in entry['profile_name'])
```

```
print("Found " + str(filtered.count()) + " entries.n")
for review in filtered.collect():
    print("Rating: " + str(review['score']) + " and helpfulness: " +
review['helpfulness'])
    print("http://www.amazon.com/dp/" + review['product_id'])
    print(review['summary'])
    print(review['review'])
    print("n")
# Get best and worst rated movies
reviews_by_movie = reviews.map(lambda r: ((r['product_id'],), r['score']))

# Calculate average rating per movie
avg = reviews_by_movie.mapValues(lambda x: (x, 1))
    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
avg = avg.filter(lambda x: x[1][1] > 20)
avg = avg.map(lambda x: ((x[1][0] / x[1][1],), x[0]))
    .sortByKey(ascending=True)

# Print top movies by watch count
for movie in avg.take(10):
    print("http://www.amazon.com/dp/" + movie[1][0] + " Rating: " +
str(movie[0][0]))

# Convert RDD to Pandas DataFrame for time series analysis
from datetime import datetime
timeseries_rdd = reviews.map(lambda entry: {'score': entry['score'],
                                             'time':
datetime.fromtimestamp(entry['time'])})

# Sample the data for visualization
sample = timeseries_rdd.sample(withReplacement=False, fraction=20000.0 /
num_entries, seed=1134)
timeseries = pd.DataFrame(sample.collect(), columns=['score', 'time'])
print(timeseries.head(3))

# Set up time series analysis and plot
timeseries.score.astype('float64')
timeseries.set_index('time', inplace=True)
Rsample = timeseries.score.resample('Y').count()
Rsample.plot()
Rsample2 = timeseries.score.resample('M').count()
Rsample2.plot()
Rsample3 = timeseries.score.resample('Q').count()
Rsample3.plot()

# Visualize histograms for average rating of movies, number of movies
reviewed by users, and movies reviewed by number of users
for movie in avg.take(4):
    plt.bar(movie[1][0], movie[0][0])
    plt.title('Histogram of 'AVERAGE RATING OF MOVIE'')
    plt.xlabel('MOVIE')
    plt.ylabel('AVGRATING')

for movie in avg2.take(3):
    plt.bar(movie[1][0], movie[0][0])
    plt.title('Histogram of 'NUMBER OF MOVIES REVIEWED BY USER'')
    plt.xlabel('USER')
```

```python
    plt.ylabel('MOVIE COUNT')

for movie in avg3.take(4):
    plt.bar(movie[1][0], movie[0][0])
    plt.title('Histogram of 'MOVIES REVIEWED BY NUMBER OF USERS'')
    plt.xlabel('MOVIE')
    plt.ylabel('USER COUNT')

# Build recommendation model using ALS
from pyspark.mllib.recommendation import ALS
from numpy import array
import hashlib
import math

# Function to hash strings
def get_hash(s):
    return int(hashlib.sha1(s).hexdigest(), 16) % (10 ** 8)

# Prepare ratings data
ratings = reviews.map(lambda entry:
tuple([get_hash(entry['user_id'].encode('utf-8')),

get_hash(entry['product_id'].encode('utf-8')),
                                            int(entry['score'])]))

# Train ALS model
rank = 20
numIterations = 20
model = ALS.train(train_data, rank, numIterations)

# Evaluate the model on test data
unknown = test_data.map(lambda entry: (int(entry[0]), int(entry[1])))
predictions = model.predictAll(unknown).map(lambda r: ((int(r[0]),
int(r[1])), r[2]))
true_and_predictions = test_data.map(lambda r: ((int(r[0]), int(r[1])),
r[2])).join(predictions)

# Analyze word frequencies for sentiment analysis
min_occurrences = 10
good_reviews = reviews.filter(lambda line: line['score'] == 5.0)
bad_reviews = reviews.filter(lambda line: line['score'] == 1.0)

# Extract words and calculate frequencies
good_words = good_reviews.flatMap(lambda line: line['review'].split(' '))
num_good_words = good_words.count()
good_words = good_words.map(lambda word: (word.strip(), 1))
    .reduceByKey(lambda a, b: a + b)
    .filter(lambda word_count: word_count[1] > min_occurrences)

# Calculate word frequencies
frequency_good = good_words.map(lambda word: ((word[0],), float(word[1]) /
num_good_words))

# Join frequencies and analyze relative differences
joined_frequencies = frequency_good.join(frequency_bad)
result = joined_frequencies.map(lambda f: ((relative_difference(f[1][0],
f[1][1]),), f[0][0]))
```

```
    .sortByKey(ascending=False)

# Print top expressions for positive and negative reviews
for movie in result.take(50):
    plt.bar(movie[1], movie[0][0])
    plt.title('Histogram of 'SENTIMENT ANALYSIS'')
    plt.xlabel('WORD')
    plt.ylabel('NUMBER OF OCCURRENCES')
```