```
"""BigData_FinalProject.ipynb

# K-Beans Clustering and Classification
##by Purvi Contractor, Swapna Kumar, Niko Laohoo, Terisha Prax </h2>

#### Python and Spark Set Up
"""

!rm -rf spark-3.1.1-bin-hadoop3.2

!apt-get install openjdk-8-jdk-headless -qq > /dev/null
!pip install -q findspark pyspark

import os
os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"

import findspark
findspark.init()
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .config("spark.jars", "/usr/local/lib/python3.10/dist-packages/
pyspark/jars/graphframes-0.8.2-spark3.3.2-s_2.11.jar") \
    .getOrCreate()

spark.conf.set("spark.sql.repl.eagerEval.enabled", True)  # Property
used to format output tables better\

sc = spark.sparkContext

#here we are actually building a spark session. In DB we already had
the session available to us. In the Spark Session, we're making sure
that the appropriate jar file is loaded.

"""# Part 1: Exploratory Data Analysis and Pre-Processing #

**Dataset Description:**
This dataset consists of dry beans features describing the shape of
the bean. We'll classify the most well-known 7 types of beans -
Barbunya, Bombay, Cali, Dermason, Horoz, Seker and Sira.

Loading the Dataset
"""

import pandas as pd

# Reads in Dry Beans Dataset
file_path = 'https://raw.githubusercontent.com/tkolencherry/
bigData_Project/main/Dry_Bean_Dataset.csv'
df_pandas = pd.read_csv(file_path)
```

```
df_pandas_final = df_pandas # Used to find finalized dataset in
original scale
df_pandas["index"] = range(0, len(df_pandas))
df = spark.createDataFrame(df_pandas)

# Standardizing dataset
bean_types = df_pandas['Class']
df_pandas = df_pandas.loc[:, df_pandas.columns != "Class"]
df_pandas = (df_pandas - df_pandas.mean()) / df_pandas.std()
df_pandas["index"] = range(0, len(df_pandas))
df_pandas['Class'] = bean_types
df_scaled = spark.createDataFrame(df_pandas)

# Printing the schema to see the data types and column names
df.printSchema()

#viewimg the summary statistics of the dataset

df.describe().show()

"""Data Cleaning"""

#Checking for missing values or data inconsistencies
#count the number of missing or null entries in each column

from pyspark.sql.functions import col, isnan, when, count
df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c
in df.columns]).show()

"""Data Visualization:
Converting the Spark Dataframe to pandas Dataframe to vizualize
dataset using Matplotlib or Seaborn
"""

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Convert Spark DataFrame to Pandas DataFrame
pandas_df = df.toPandas()

"""## 1- Count and distribution of all beans categories:"""

# Histogram of features
pandas_df["Class"].value_counts().plot(kind='bar',title= "Bean type
counts in the training data",xlabel= 'Bean type',ylabel= 'Frequency');

"""*   We noticed a disparity between the count of each class.
Dermason is the most frequent while Bombay being the least frequent.
*   The Big difference between the two classes will be taken care of
```

```
when building a model, if required.

## 2- Univariate Analysis
"""

# Distributuon of Numerical features
fig, axes = plt.subplots(4, 4, figsize=(15, 20))
for feature, ax in zip(pandas_df.columns.drop("Class"),
axes.flatten()):
    sns.histplot(data=pandas_df[feature],ax=ax)
    median = pandas_df[feature].median()
    ax.set_title( f'{feature} ,Median : {median:0.1f}')
    ax.axvline(median,
               color ='red',
               lw=2,
               alpha=0.5)
plt.show()

"""*   Some distributions have long tails and most are bi-modal which
shows that some bean classes are quite distinct from others.
*   Many features show skewness and outliers in their distributuon
which may resemble a unique class of dry beans.

## 3- Boxplot of numerical features for each type of bean
"""

fig, ax = plt.subplots(8, 2, figsize=(15, 25))

for variable, subplot in zip(pandas_df.columns.drop("Class"),
ax.flatten()): sns.boxplot(x=pandas_df['Class'], y=
pandas_df[variable], ax=subplot)
plt.tight_layout()

"""*   Bombay class differs significantly from other classes, has
large area and perimeter.
*   Dermason class is similar to Seker class in some features, and
Sira class in other features.
*   Both Barbunya class and Cali class have similar distributions and
values in many features (area, minor axis length, equivalent diameter,
extent, shape factor1).

## 4- Bivariate/Multivariate Analysis
"""

sns.pairplot(pandas_df, hue="Class");

"""*   We see a linear trend between many features
*   There are some clusters which overlap mainly between Dermason and
Sira class.
*   Bombay is the most separated class from others in some features.
```

```
# Part 2: K-Means Clustering Algorithm via MapReduce

Preparing RDD from Dataframe: remove class, randomize observations,
set index as key
"""

# Creates RDD from Beans dataframe

n = df_scaled.count()       # Finds number of observations
p = len(df_scaled.columns)  # Finds number of columns

# Dataset comes with types of beans grouped together so first
observations are randomized to ensure intial centers are randomized
beans = df_scaled.rdd.map(lambda x: (x[p-2],
x[0:p-2])).takeSample(withReplacement=False , num = n, seed = 415224)
# Index is used as the key
beans = sc.parallelize(beans)

"""Functions to cluster observations into groups using MapReduce"""

# Function assigns each point to cluster with the closest cluster
center
from math import dist
def assign_clusters(x, centers): # Functions requires current cluster
centers and point in question (x)
  min = float("inf")      # Initialize min distance to infinity
  assigned_cluster = -1   # Initialize cluster to -1
  d = 0                   # Initialize distance to cluster center to 0
  for i in range(len(centers)):
    d = dist(x, centers[i])     # For each cluster find the distance
from the point to cluster center
    if (d < min):               # If distance is less than current min
distance:
      min = d                   # Re-assign min distance
      assigned_cluster = i      # Assign point to new cluster
  return(assigned_cluster, x) # Return cluster assignment and original
data


# Function takes data in RDD format and uses k-means clustering,
through MapReduce, to group data into clusters
def kmeans_clustering(data, k, maxiter): # Function requires data and
value of k (number of clusters)

  p = len(data.take(1)[0][1])   # Finds number of columns of the
dataset
  i = 0 # Iteration counter

  centers = data.map(lambda x:
```

```
x[1]).takeSample(withReplacement=False , num = k, seed = 4192024) #
Randomly selects intial centers

  data = data.map(lambda x: (x[0], -1, x[1:p-1])) # Initializes every
point to cluster -1

  old_assigned = data.map(lambda x: (x[1])).collect() # Saves old
cluster assignments, which is all -1 (used in while loop for stopping
condition)
  data = data.map(lambda x: (x[0], assign_clusters(x[2][0],
centers))).map(lambda x: (x[0], x[1][0], x[1][1])) # First iteration
of kmeans-clustering
  new_assigned = data.map(lambda x: x[1]).collect()   # Saves first
cluster assigments (used in while loop for stopping condition)

  # Iterates through kmeans algorithm until previous cluster
assignments are the same as the current iteration, or until max number
ot iterationis reached
  while (old_assigned != new_assigned and i <= maxiter):

    old_assigned = new_assigned # Save previous iteration's clusters
assignments

    # Update the centers to the mean/center of the current cluster
assignments (finds average of all points in that cluster using
MapReduce)
    centers = data.map(lambda x: (x[1], x[2])).reduceByKey(lambda x,y:
((x[0]+y[0])/2, (x[1]+y[1])/2, (x[2]+y[2])/2, (x[3]+y[3])/2,
(x[4]+y[4])/2,

(x[5]+y[5])/2, (x[6]+y[6])/2, (x[7]+y[7])/2, (x[8]+y[8])/2,
(x[9]+y[9])/2, (x[10]+y[10])/2,

(x[11]+y[11])/2, (x[12]+y[12])/2, (x[13]+y[13])/2, (x[14]+y[14])/2,
(x[15]+y[15])/2 )).sortBy(lambda x: x[0]).map(lambda x:
x[1]).collect()

    # Applies k-means algorithm with new centers
    data = data.map(lambda x: (x[0], assign_clusters(x[2],
centers))).map(lambda x: (x[0], x[1][0], x[1][1]))

    new_assigned = data.map(lambda x: x[1]).collect() # Saves new
cluster assignments
    i = i + 1 # Update iteration counter

  return data # Return data with final cluster assignments

"""## Testing for optimal value of k using silhouette width"""

# Testing for optimal value for k using silhouette width
```

```
from pyspark.ml.evaluation import ClusteringEvaluator
from pyspark.ml.feature import VectorAssembler

vectors_assemble = VectorAssembler(inputCols=["Area", "Perimeter",
"MajorAxisLength", "MinorAxisLength", "AspectRation",
                                   "Eccentricity", "ConvexArea",
"EquivDiameter", "Extent", "Solidity", "roundness",
                                   "Compactness", "ShapeFactor1",
"ShapeFactor2", "ShapeFactor3", "ShapeFactor4"],
                                   outputCol= "features")

evaluator = ClusteringEvaluator(predictionCol = "cluster", featuresCol
= "features")

"""k = 2, silhouette width = 0.5547"""

# k = 2
clustered_beans = kmeans_clustering(beans, 2, maxiter=500)
cluster_assignments = clustered_beans.map(lambda x: (x[0],
x[1])).sortBy(lambda x: x[0])
cluster_assignments = cluster_assignments.toDF()

df_scaled2 = df_scaled.join(cluster_assignments, df_scaled.index ==
cluster_assignments._1).drop("_1").withColumnRenamed(existing = "_2",
new = "cluster")

clustered_beans_df = vectors_assemble.transform(df_scaled2)

evaluator.evaluate(clustered_beans_df)

"""k = 3, silhouette width = 0.4812"""

# k = 3
clustered_beans = kmeans_clustering(beans, 3, maxiter=500)
cluster_assignments = clustered_beans.map(lambda x: (x[0],
x[1])).sortBy(lambda x: x[0])
cluster_assignments = cluster_assignments.toDF()

df_scaled3 = df_scaled.join(cluster_assignments, df_scaled.index ==
cluster_assignments._1).drop("_1").withColumnRenamed(existing = "_2",
new = "cluster")

clustered_beans_df = vectors_assemble.transform(df_scaled3)

evaluator.evaluate(clustered_beans_df)

"""k = 4, silhouette width = 0.4953"""

# k = 4
```

```
clustered_beans = kmeans_clustering(beans, 4, maxiter=500)
cluster_assignments = clustered_beans.map(lambda x: (x[0],
x[1])).sortBy(lambda x: x[0])
cluster_assignments = cluster_assignments.toDF()

df_scaled4 = df_scaled.join(cluster_assignments, df_scaled.index ==
cluster_assignments._1).drop("_1").withColumnRenamed(existing = "_2",
new = "cluster")

clustered_beans_df = vectors_assemble.transform(df_scaled4)

evaluator.evaluate(clustered_beans_df)

"""k = 5, silhouette width = 0.4480


"""

# k = 5
clustered_beans = kmeans_clustering(beans, 5, maxiter=500)
cluster_assignments = clustered_beans.map(lambda x: (x[0],
x[1])).sortBy(lambda x: x[0])
cluster_assignments = cluster_assignments.toDF()

df_scaled5 = df_scaled.join(cluster_assignments, df_scaled.index ==
cluster_assignments._1).drop("_1").withColumnRenamed(existing = "_2",
new = "cluster")

clustered_beans_df = vectors_assemble.transform(df_scaled5)

evaluator.evaluate(clustered_beans_df)

"""k = 6, silhouette width = 0.4216

"""

clustered_beans = kmeans_clustering(beans, 6, maxiter=500)
cluster_assignments = clustered_beans.map(lambda x: (x[0],
x[1])).sortBy(lambda x: x[0])
cluster_assignments = cluster_assignments.toDF()

df_scaled6 = df_scaled.join(cluster_assignments, df_scaled.index ==
cluster_assignments._1).drop("_1").withColumnRenamed(existing = "_2",
new = "cluster")

clustered_beans_df = vectors_assemble.transform(df_scaled6)

evaluator.evaluate(clustered_beans_df)

"""## Running kmeans clustering algorithm with optimal value k = 2 to
```

```
get final clusters and final dataframe"""

# Running kmeans clustering algorithm on beans rdd with optimal value
k = 2

clustered_beans = kmeans_clustering(beans, 2, maxiter=500)
cluster_assignments = clustered_beans.map(lambda x: (x[0],
x[1])).sortBy(lambda x: x[0])
cluster_assignments = cluster_assignments.toDF()

# Adding index to dataframe in original scale to join with cluster
assignments
df_pandas_final["index"] = range(0, len(df_pandas_final))
df_final = spark.createDataFrame(df_pandas_final)

# Adding cluster assignments to original dataframe
df_final = df_final.join(cluster_assignments, df_final.index ==
cluster_assignments._1).drop("_1").withColumnRenamed(existing = "_2",
new = "cluster")

# Adding cluster assignments to standardized dataframe
df_scaled = df_scaled.join(cluster_assignments, df_scaled.index ==
cluster_assignments._1).drop("_1").withColumnRenamed(existing = "_2",
new = "cluster")

df_scaled.show()

df_final.show()

"""## Investigation into members of the two groups"""

# Investigating characteristics of the two groups
df_final.groupBy("cluster").count()

df_final.groupBy("cluster", "Class").count().orderBy("cluster").show()

df_final.groupBy("cluster").avg()

df_scaled.groupBy("cluster").avg()

"""# Part 3: Logistic Regression with CrossValidator

## 3A. Logistic Regression with CrossValidator with scaled dataset to
predict the clusters

**Importing Required Libraries**
"""

from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
```

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.mllib.evaluation import MulticlassMetrics

# Create ML pipeline for pre-processsing using stages

# create array of all features
vectors_assemble = VectorAssembler(inputCols=["Area", "Perimeter",
"MajorAxisLength", "MinorAxisLength", "AspectRation",
                                   "Eccentricity", "ConvexArea",
"EquivDiameter", "Extent", "Solidity", "roundness",
                                   "Compactness", "ShapeFactor1",
"ShapeFactor2", "ShapeFactor3", "ShapeFactor4"],
                                   outputCol= "features")

# Create the logistic regresion model and pipeline using the model

lr = LogisticRegression(featuresCol = "features", labelCol =
"cluster")

# Create the pipeline for the LR model with above stages
pipeline = Pipeline(stages = [vectors_assemble, lr])

# Create classification model for LR for tuning
paramGrid = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.005, 0.01]) \
    .addGrid(lr.maxIter, [5, 10]) \
    .build()

# Using crossvalidator to find the best param for the model
# Set evaluator
evaluator = MulticlassClassificationEvaluator(labelCol ="cluster",
predictionCol ="prediction", metricName = "accuracy")

crossvalidator = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=evaluator,
                          numFolds=3)

cvModel = crossvalidator.fit(df_scaled)

#get y_hat
lrmodel_predict = cvModel.transform(df_scaled)

# Save y_hat and y
predictionAndLabels = lrmodel_predict.select("prediction",
"cluster").rdd.map(lambda lp: (float(lp.prediction),
float(lp.cluster)))
```

```
# Assign metrics object so we can get confusion mtx + accuracy,
precision, and recall
metrics = MulticlassMetrics(predictionAndLabels)

metrics.confusionMatrix().toArray()

metrics.accuracy

metrics.precision(1.0)

metrics.recall(1.0)

"""## 3B. Performing Logistic Regression from original dataset to
predict the class of beans (using original seven classes)

#### Import Required Libraries
"""

# Import required libraries

from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import FeatureHasher, StringIndexer,
IndexToString, VectorAssembler
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.mllib.evaluation import MulticlassMetrics

"""#### Create Data Frame for Logistic Regression

#### Split data into train/test
"""

# create dataframe for logistic regression (LR)
beansDF = spark.createDataFrame(df_pandas)
beansDF.show()

# Splitting the dataset into test and train sub-datasets.
# Training sample is 80%, testing is 20%.
train, test = beansDF.randomSplit([0.80, 0.20], seed = 2015)

"""#### Create ML Pipeline for pre-processing"""

# Create ML pipeline for pre-processsing using stages

# Vectorize all feature columns (all cols are numeric)

vctrassembler =  VectorAssembler(inputCols=["Area", "Perimeter",
"MajorAxisLength", "MinorAxisLength", "AspectRation",
                                   "Eccentricity", "ConvexArea",
```

```
"EquivDiameter", "Extent", "Solidity", "roundness",
                                   "Compactness", "ShapeFactor1",
"ShapeFactor2", "ShapeFactor3", "ShapeFactor4"],
                        outputCol="features")

# Label Conversion: Convert label in the "Class" name to a number
classindexer = StringIndexer(inputCol = "Class", outputCol =
"label" ).fit(beansDF)

# Convered prediction labels back to the "Class" name
labelConverter  = IndexToString(inputCol="prediction",
outputCol="predictClass", labels=classindexer.labels)

# Create the logostic regresion model
lr = LogisticRegression(featuresCol = "features", labelCol = "label")

"""#### Create the logistic regresion model and pipeline using the
stages"""

# Create the pipeline for the LR model with above stages
pipeline = Pipeline(stages = [vctrassembler, classindexer, lr,
labelConverter])

# Create classification model for LR for tuning
paramGrid = ParamGridBuilder() \
    .addGrid(lr.regParam, [0.0, 0.005, 0.01]) \
    .addGrid(lr.maxIter, [5, 10]) \
    .build()

# Using crossvalidator to find the best param for the model
# Set evaluator
evaluator = MulticlassClassificationEvaluator(labelCol ="label",
predictionCol ="prediction", metricName = "accuracy")

crossvalidator = CrossValidator(estimator=pipeline,
                          estimatorParamMaps=paramGrid,
                          evaluator=evaluator,
                          numFolds=3)

"""#### Executing the model by running cross-validation and fine-
tuning hyper parameters"""

# Run cross-validation, and choose the best set of parameters.
cvmodel = crossvalidator.fit(train)

"""#### Predict bean classes using the trained model
Predicted Bean Classes:
0 - DERMASON, 1 - SIRA, 2 - SEKER, 3 - HOROZ, 4 - CALI, 5 - BARBUNYA,
6 - BOMBAY
"""
```

```
# Predict bean classes from the model
lrPredict = cvmodel.transform(test)
lrPredict.show()

"""#### Get metrics for all iterations, ordered by best params for
predicting beans class"""

l_params = [{param.name: val for param, val in pmap.items()} for pmap
in cvmodel.getEstimatorParamMaps()]
#print(l_params)
import pandas as pd
import numpy as np

metricDF = pd.DataFrame.from_dict([
    {**hyper_param, cvmodel.getEvaluator().getMetricName(): metric}
    for  hyper_param, metric in zip( l_params, cvmodel.avgMetrics)
])

metricDF = metricDF.sort_values("accuracy", ascending=False)
print(metricDF)

# Best model metrics and params
print(" ")
print("Best model accuracy %s: " % np.max(cvmodel.avgMetrics))
cvmodel.getEstimatorParamMaps()[ np.argmax(cvmodel.avgMetrics) ]

"""#### Display/print statistics and classification metrics for beans
class prediction"""

# Compute raw scores
predictLabels = lrPredict.select("prediction", "label").rdd.map(lambda
lp: (float(lp.prediction), float(lp.label)))

# Overall statistics

# Instantiate metrics object
metrics = MulticlassMetrics(predictLabels)

print("======================================================================
========")
print(" ")
print("Classification Metrics Summary for label Prediction using
Scaled Data")
print("======================================================================
========")
print(" ")
print("Accuracy = %s" % metrics.accuracy)
print(" ")
print("======================================================================
```

```
========")
print(" ")
# Statistics by class
print("Metrics Statistics By Class")
labels = lrPredict.rdd.map(lambda lp: lp.label).distinct().collect()

for l_label in sorted(labels):

print("======================================================================
====")
    print("Class %s precision = %s" % (l_label,
metrics.precision(l_label)))
    print("Class %s recall = %s" % (l_label, metrics.recall(l_label)))
    print("Class %s F1 Measure = %s" % (l_label,
metrics.fMeasure(float(l_label), beta=1.0)))
    print("Class %s True positive rate = %s" % (l_label,
metrics.truePositiveRate(float(l_label))))
    print("Class %s False positive rate = %s" % (l_label,
metrics.falsePositiveRate(float(l_label))))
    print(" ")

print("======================================================================
========")
print(" ")

# Weighted stats
print("WEIGHTED STATISTICS")
print("======================================================================
========")
print(" ")
print("Weighted Recall = %s" % metrics.weightedRecall)
print("Weighted Precision = %s" % metrics.weightedPrecision)
print("Weighted F(1) Score = %s" % metrics.weightedFMeasure())
print("Weighted F(0.5) Score = %s" %
metrics.weightedFMeasure(beta=0.5))
print("Weighted True Positive Rate = %s" %
metrics.weightedTruePositiveRate)
print("Weighted False Positive Rate = %s" %
metrics.weightedFalsePositiveRate)

"""#### Join the predicted bean class results with original DF to
compare results"""

# create df with selected columns for the join with kmeans df
lrPredict_final = lrPredict.select(["index", "prediction",
"predictClass"]).withColumnRenamed(existing = "index", new =
"index_1")

lrDF_scaled_class = beansDF.join(lrPredict_final, beansDF.index ==
lrPredict_final.index_1).drop("index_1")
```

```
lrDF_scaled_class.show()

beansdf_orig = spark.createDataFrame(df_pandas_final)
lrDF_class = beansdf_orig.join(lrPredict_final, beansdf_orig.index ==
lrPredict_final.index_1).drop("index_1")
lrDF_class.show()

"""## 3C. Performing Logistic Regression on scaled data received after
K-means clustering and using clusters as target variable (This model
was only used for comparison to test model in 3A)"""

# create dataframe for logistic regression (LR)

scaled_beansDF = df_scaled
scaled_beansDF.show()
# Splitting the dataset into test and train sub-datasets.
# Training sample is 80%, testing is 20%.

scaled_train, scaled_test = scaled_beansDF.randomSplit([0.80, 0.20],
seed = 2015)

"""#### Create ML Pipeline for pre-processing, including class as a
feature"""

# Create ML pipeline for pre-processsing using stages

# Label Conversion: Convert label in the "Class" name to a number
classindexer_s = StringIndexer(inputCol = "Class", outputCol =
"ClassLabel" ).fit(scaled_beansDF)

# use VectorAssembler for featurizing all columns

vctrassembler_s =  VectorAssembler(inputCols=["Area", "Perimeter",
"MajorAxisLength", "MinorAxisLength", "AspectRation",
                                   "Eccentricity", "ConvexArea",
"EquivDiameter", "Extent", "Solidity", "roundness",
                                   "Compactness", "ShapeFactor1",
"ShapeFactor2", "ShapeFactor3", "ShapeFactor4", "ClassLabel"],
                        outputCol="features")

# Create the logostic regresion model and pipeline using the model
lr_s = LogisticRegression(featuresCol = "features", labelCol =
"cluster")

"""#### Create the pipeline using the stages and parameter grid"""

# Create the pipeline for the LR model with above stages
pipeline_s = Pipeline(stages = [classindexer_s, vctrassembler_s,
lr_s])
```

```
# Create classification model for LR for tuning
paramGrid_s = ParamGridBuilder() \
    .addGrid(lr_s.regParam, [0.0, 0.005, 0.01]) \
    .addGrid(lr_s.maxIter, [5, 10]) \
    .build()

# Using crossvalidator to find the best param for the model
# Set evaluator
evaluator_s = MulticlassClassificationEvaluator(labelCol ="cluster",
predictionCol ="prediction", metricName = "accuracy")

crossvalidator_s = CrossValidator(estimator=pipeline_s,
                          estimatorParamMaps=paramGrid_s,
                          evaluator=evaluator_s,
                          numFolds=3)

"""#### Executing the scaled LR model by running cross-validation and
fine-tuning hyper parameters"""

# Run cross-validation, and choose the best set of parameters.
cvmodel_s = crossvalidator_s.fit(scaled_train)

"""#### Predict clusters using the scaled trained model"""

# Predict bean classes from the model
lrPredict_s = cvmodel_s.transform(scaled_test)
lrPredict_s.show()

"""#### Get metrics for all iterations, ordered by best params"""

l_params = [{param.name: val for param, val in pmap.items()} for pmap
in cvmodel_s.getEstimatorParamMaps()]
#print(l_params)
import pandas as pd
import numpy as np

metricDF_s = pd.DataFrame.from_dict([
    {**hyper_param, cvmodel_s.getEvaluator().getMetricName(): metric}
    for  hyper_param, metric in zip( l_params, cvmodel_s.avgMetrics)
])

metricDF_s = metricDF_s.sort_values("accuracy", ascending=False)
print(metricDF_s)

# Best model metrics and params

print(" ")
print("Best model accuracy %s: " % np.max(cvmodel_s.avgMetrics))
cvmodel_s.getEstimatorParamMaps()[ np.argmax(cvmodel_s.avgMetrics) ]
```

```
"""#### Display/print statistics and classification metrics

"""

# Compute raw scores
predictLabels_s = lrPredict_s.select("prediction",
"cluster").rdd.map(lambda lp: (float(lp.prediction),
float(lp.cluster)))

# Overall statistics

# Instantiate metrics object
metrics_s = MulticlassMetrics(predictLabels_s)

print("=================================================================
========")
print(" ")
print("Classification Metrics Summary for Cluster Prediction using
Scaled Data")
print("=================================================================
========")
print(" ")
print("Accuracy = %s" % metrics_s.accuracy)
print(" ")
print("=================================================================
========")
print(" ")
# Statistics by class
print("Metrics Statistics By Class")
labels_s = lrPredict_s.rdd.map(lambda lp:
lp.cluster).distinct().collect()

for l_label in sorted(labels_s):

print("=================================================================
====")
    print("Class %s precision = %s" % (l_label,
metrics_s.precision(l_label)))
    print("Class %s recall = %s" % (l_label,
metrics_s.recall(l_label)))
    print("Class %s F1 Measure = %s" % (l_label,
metrics_s.fMeasure(float(l_label), beta=1.0)))
    print("Class %s True positive rate = %s" % (l_label,
metrics_s.truePositiveRate(float(l_label))))
    print("Class %s False positive rate = %s" % (l_label,
metrics_s.falsePositiveRate(float(l_label))))
    print(" ")

print("=================================================================
========")
```

```
print(" ")

# Weighted stats
print("WEIGHTED STATISTICS")
print("=================================================================
========")
print(" ")
print("Weighted Recall = %s" % metrics_s.weightedRecall)
print("Weighted Precision = %s" % metrics_s.weightedPrecision)
print("Weighted F(1) Score = %s" % metrics_s.weightedFMeasure())
print("Weighted F(0.5) Score = %s" %
metrics_s.weightedFMeasure(beta=0.5))
print("Weighted True Positive Rate = %s" %
metrics_s.weightedTruePositiveRate)
print("Weighted False Positive Rate = %s" %
metrics_s.weightedFalsePositiveRate)

"""#### Join with scaled K-means output and original with K-means"""

# create df with selected columns for the join with kmeans df
lrPredict_s_final = lrPredict_s.select(["index",
"prediction"]).withColumnRenamed(existing = "index", new =
"index_1").withColumnRenamed(existing = "prediction", new =
"predictCluster")
lrDF_s_final = scaled_beansDF.join(lrPredict_s_final,
scaled_beansDF.index == lrPredict_s_final.index_1).drop("index_1")
lrDF_s_final.show()

lrDF_cluster = df_final.join(lrPredict_s_final, df_final.index ==
lrPredict_s_final.index_1).drop("index_1")
lrDF_cluster.show()

"""# Part 4: Performing Linear Discriminant Analysis

**Importing Necessary libraries**
"""

import numpy as np
import pandas as pd
from sklearn import metrics
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import train_test_split, cross_val_score,
GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report,
accuracy_score, ConfusionMatrixDisplay
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt
import seaborn as sns

# checking the scaled dataset
```

```
df_pandas.head()

# Preparing the dataset by separating it into target and feature
variables.

X = df_pandas.drop(columns= ['Class', 'index'],  axis=1)
Y = df_pandas['Class']

# Encode categorical data
encoder = LabelEncoder()
y_encoded = encoder.fit_transform(Y)

#Splitting the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded,
test_size=0.3, random_state=42)

"""**The class labels are : Seker-0, Barbunya-1, Bombay-2, Cali-3,
Dermosan-4, Horoz-5, Sira-6.**"""

#Perform LDA

lda = LinearDiscriminantAnalysis()
lda.fit(X_train, y_train)
y_pred = lda.predict(X_test)

#Evaluate the model

# Accuracy
print("Accuracy:", accuracy_score(y_test, y_pred))

# Confusion Matrix
print("\nConfusion Matrix:\n\n", confusion_matrix(y_test, y_pred))

# Classification Report
print("\nClassification Report:\n\n", classification_report(y_test,
y_pred))

# Visualizing confusion matrix

cm = confusion_matrix(y_test, y_pred)
class_names = ["Seker", "Barbunya", "Bombay"," Cali", "Dermosan",
"Horoz", "Sira"]

plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
xticklabels=class_names, yticklabels=class_names)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()
```

```
"""**Using Cross-Validation to evalute performance of the model**

"""

# Performing 10-fold cross-validation

scores = cross_val_score(lda, X, y_encoded, cv=10)
print("Cross-validated scores:", scores)
print("Average score:", np.mean(scores))

"""**Hyperparameter Tuning Using GridSearchCV**"""

# Defining parameter grid
param_grid = {
    'solver': ['svd', 'lsqr', 'eigen'],
    'shrinkage': [None, 'auto', 0.1, 0.5, 0.9]
}

# Setting up the GridSearchCV object
grid_search = GridSearchCV(lda, param_grid, cv=2, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Best parameters and best score
print("Best parameters:", grid_search.best_params_)
print("Best cross-validated score:", grid_search.best_score_)

# Evaluate on the test set
y_pred = grid_search.predict(X_test)
print("Test Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test,
y_pred))

#evaluate the model fitted with the best parameters found by
GridSearchCV

# Check performance on the test data
print("Test Accuracy:", accuracy_score(y_test, y_pred))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n\n", classification_report(y_test,
y_pred))

"""## **Performing LDA on the dataset received after K-means
clustering using clusters as target variable**"""

# converting spark datafrome to pandas dataframe

df = df_scaled.toPandas()
df.head()
```

```
from sklearn.preprocessing import LabelEncoder

# Create a label encoder object
encoder = LabelEncoder()

# converting categorical Class to Numeric
df['Class'] = encoder.fit_transform(df['Class'])

# Preparing the dataset by separating it into target and feature
variables.

X = df.drop(columns=['cluster'],  axis=1)
Y = df['cluster']


#Splitting the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, Y,
test_size=0.3, random_state=42)

#Perform LDA

lda = LinearDiscriminantAnalysis()
lda.fit(X_train, y_train)
y_pred = lda.predict(X_test)

# Performing 3-fold cross-validation

scores = cross_val_score(lda, X, Y, cv=2)
print("Cross-validated scores:", scores)
print("Average score:", np.mean(scores))

# Defining parameter grid

param_grid = {
    'solver': ['svd', 'lsqr', 'eigen'],
    'shrinkage': [None, 'auto', 0.1, 0.5, 0.9]
}

# Setting up the GridSearchCV object
grid_search = GridSearchCV(lda, param_grid, cv=2, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Best parameters and best score
print("Best parameters:", grid_search.best_params_)
print("Best cross-validated score:", grid_search.best_score_)

# Evaluate on the test set
y_pred = grid_search.predict(X_test)
print("Test Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test,
```

```
y_pred))

#evaluate the model fitted with the best parameters found by
GridSearchCV

# Check performance on the test data
print("Test Accuracy:", accuracy_score(y_test, y_pred))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n\n", classification_report(y_test,
y_pred))
```