30/1/24

Lab - 4

Q1. Demonstrate how to load a dataset for recommendation systems into a Pyspark DataFrame

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as f
import findspark
findspark.init()
spark = SparkSession.builder.getOrCreate()

ratings = spark.read.json("moves 1.json").select("user_id",
          "product_id","score").cache()
ratings = ratings.head(10000)
ratings = spark.createDataFrame(ratings)
```

Q2. Script to split the data and train a recommendation model

```
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator
from pyspark.ml.feature import StringIndexer
from pyspark.ml import Pipeline

indexers = [StringIndexer(inputCol = column, outputCol = column+
           "_index").fit(ratings)
           for column in ["user_id","product_id"]]

pipline = Pipeline(stages = indexers)
ratings_indexed = pipeline.fit(ratings).transform(ratings)

training_data, validation_data = ratings_indexed.randomSplit
                                 ([8.0, 2.0])

als = ALS(usersCol = "user_id_index", itemCol = "product_id_index",
      ratingCol = "score", rank = 10, maxIters = 5, regParam = 0.
      coldStartStrategy = "drop")

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

model = als.fit(training_data)
predictions = model.transform(validation_data)
predictions.show(10, False)
```

Q3. ALS algorithm for collaborative filtering

```
user1 = validation_data.filter(validation_data["user_id_index"]
        == 1.0).select('product_id','product_id_index',
        'user_id','user_id_index')
recommendations = model.transform(user1)
recommendations.orderBy('prediction',ascending=False).show()
```

Q4. Implement code to Evaluate the performance using appropriate metrics

```
rmse = evaluator.evaluate(predictions)
print(f"RMSE = {rmse}")

mae_eval = RegressionEvaluator(metricName="mae",
           labelCol = "score", predictionCol = "prediction")

mae = mae_evaluator.evaluate(predictions)
print(f"MAE = {mae}")
```

Output

```
RMSE = 4.8775213
MAE = 3.7637585
```

RDD

1. loading a dataset

```
import json
from pyspark.sql import ~~SparkConf~~ SQLContext
from pyspark import SparkConf

conf = SparkConf()
SC = pyspark.SparkContext(conf=conf)

reviews_raw = sc.textFile('movies1.json')
reviews = reviews_raw.map(lambda line: json.loads(line))
                .filter(validate)
```

2&3.

```
from pyspark.mllib.recommendation import ALS
from numpy import array
import hashlib
import math
def get_hash(s):
    return int(hashlib.sha1(s).hexdigest(),16)%(10**8)

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_data = ratings.filter(lambda entry: ~~tuple~~
        ((entry[0]+entry[1])%10)>=2)
test_data = ratings.filter(lambda entry:
        ((entry[0]+entry[1])%10)<2)
```

4.

```
from math import sqrt

rank = 10
numIterations = 10
model = ALS.train(train_data, rank, numIterations)

def convertToFloat(lines):
    returnedLine = []
    for x in lines:
        returnedline.append(float(x))
    return returnedline

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)
MSE = true_and_predictions.map(lambda r:
    (int(r[1][0])-int(r[1][1])**2).reduce(
    lambda x,y: x+y)/true_and_predictions.count())
```

6/2/24

13/2/24

Lab-6

1) Load dataset and display info about dataset

2) Pyspark script to handle missing values & categorical fea

```
import pyspark
import os
from pyspark.sql.. import SparkSession
from pyspark.sql.types import DoubleType
from pyspark.sql.functions import col

spark = SparkSession.builder.getOrCreate()
data_without_header = spark.read.option("inferSchema","True")
        .option("header", False).csv("data/covtype.data")

colnames=["Elevation", "Aspect", .... ]+
    [f"Wilderness_Area_{i}" for i in range(4)] +
    [f"Soil_Type_{i}" for i in range(40)] + ["Cover_Type"]

data = data_without_header.toDF(*colnames).\
        withColumn("Cover_Type", col("Cover_Type").cast(DoubleType))
train_data, test_data = data.randomSplit([0.9, 0.1])
```

3) Develop a script that trains a Decision Tree Model on the training dataset

```
from pyspark.ml.feature import VectorAssembler
input_cols = colnames[:-1]
vector_assembler = VectorAssembler(inputCols=input_cols,
                                   outputCol="featureVector")
assembled_train_data = vector_assembler.transform(
                                   train_data)
assembled_train_data.select("featureVector").show(truncate=
                                   False)
classifier = DecisionTreeClassifier(seed=1234, labelCol="Cover_Type",
        featuresCol="featureVector", predictionCol="prediction")
model = classifier.fit(assembled_train_data)
print(model.toDebugString)
```

4) Code to evaluate decision tree model

```
from pyspark.ml.evaluation import MulticlassClassification
                                   Evaluator

predictions = model.transform(assembled_train_data)
predictions.select("Cover_Type","prediction","probability").\
                                   show(10, truncate=False)
evaluator = MulticlassClassificationEvaluator(labelCol="Cover_Type",
                                   predictionCol="prediction")

print(evaluator.setMetricName("accuracy").evaluate(predictions))
print(evaluator.setMetricName("f1").evaluate(predictions))
```

Output

Accuracy : 0.7022

F1 score : 0.6865

13/2/24

36 20/12/24

Lab-7

```
import pyspark
from pyspark.sql import SparkSession

spark= SparkSession.builder.getOrCreate()

data_without_header =spark.read.option("inferSchema",
                    .option("header","False").csv("kddcup.data")

column_names = ["duration","protocol_type","service",
                "src_bytes",...]
data = data_without_header.toDF(*column_names)

from pyspark.sql.functions import col
data.select("label").groupBy("label").count().orderBy(
    col("count").desc()).show()

from pyspark.ml.feature import VectorAssembler
from pyspark.ml.clustering import KMeans,KMeansModel
from pyspark.ml import Pipeline

numeric_only = data.drop("protocol_type","service",
                         ).cache()

assembler = VectorAssembler().setInputCols(numeric_only.
                columns[:-1]).setOutputCol("featureVector")
kmeans = KMeans().setPredictionCol("cluster").
                setFeaturesCol("featureVector")
pipeline = Pipeline().setStages([assembler,kmeans])
pipeline_model = pipeline.fit(numeric_only)
kmeans_model = pipeline_model.stages[1]

from pprint import pprint
pprint(kmeans_model.clusterCenters())
  #There are only 2 clusters
```



```
with_cluster = pipeline_model.transform(numeric_only)
with_cluster.select("cluster","label").groupBy("cluster",
    "label").count().orderBy("cluster",col("count").desc()).show(25)

#Choosing a good k value by checking training cost

from pyspark.sql import DataFrame
from random import randint

def clustering_score1(input_data,k):
    input_numeric_only = input_data.drop("protocol_type",
                         "service","flag")
    assembler = VectorAssembler().setInputCols(input_numeric_only.
                columns[:-1]).setOutputCol("featureVector")
    kmeans = KMeans().setSeed(randint(100,100000)).setK(k).
             setPredictionCol("cluster").setFeaturesCol("featureVector")
    pipeline = Pipeline().setStages([assembler,kmeans])
    pipeline_model = pipeline.fit(input_numeric_only)
    kmeans_model = pipeline_model.stages[-1]
    training_cost = kmeans_model.summary.trainingCost
    return training_cost

for k in list(range(20,100,20)):
    print(clustering_score1(numeric_only,k))

3452668119878 1.35
296653471320 96.52
7464324642470.006
5140760167652.76

from pyspark.ml.feature import OneHotEncoder,StringIndexer
from pyspark.ml.feature import StandardScaler

def one_hot_pipeline(input_col):
    indexer = StringIndexer().setInputCol(input_col).setOutputCol(
              input_col+"_indexed")
```

```
encoder = OneHotEncoder().setInputCol(input_col + "_indexed")
                         .setOutputCol(input_col + "_vec")
pipeline = Pipeline().setStages([indexer, encoder])
return pipeline, input_col + "_vec"

def fit_pipeline(data, k):
    proto_type_pipeline, proto_type_vec_col = one_hot_pipeline("protocol_type")
    service_pipeline, service_vec_col = one_hot_pipeline("service")
    flag_pipeline, flag_vec_col = one_hot_pipeline("flag")
    assemble_cols = set(data.columns) - {"label", "protocol_type",
        "service", "flag"} | {proto_type_vec_col, service_vec_col,
        flag_vec_col}
    assembler = VectorAssembler(inputCols=list(assemble_cols),
                                outputCol="featureVector")
    scaler = StandardScaler(inputCol="featureVector",
                            outputCol="scaledFeatureVector")
    kmeans = KMeans(seed=randint(100,100000), k=k,
        predictionCol="cluster", featuresCol="scaledFeatureVector",
        maxIter=40, tol=1.0e-5)
    pipeline = Pipeline(stages=[proto_type_pipeline,
        service_pipeline, flag_pipeline, assembler, scaler, kmeans])
    return pipeline.fit(data)

pipeline_model = fit_pipeline(data, 100)
count_by_cluster = pipeline_model.transform(data).select("cluster",
    "label").groupBy("cluster", "label").count().
    orderBy("cluster", "label")
```

27/2/24

Lab-9

Estimating Risk Through Monte Carlo simulation

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.config("spark.driver.memory", "4g")
        .appName('MCS').getOrCreate()
stocks = spark.read.csv(["data/stocks/ABAX.csv", ",", "],
        header='true', inferSchema='true')

from pyspark.sql import functions as fun
stocks = stocks.withColumn("Symbol", fun.input_file_name())
        .withColumn("Symbol", fun.element_at(fun.split("Symbol",
        "/"), -1))
stocks.show(2)
```

Date	Open	High	Low	Close	Volume	Symbol
31-Dec-13	52.94	54.37	52.25	52.83	79429	AEPI
30-Dec-13	50.36	54.10	50.36	52.95	131095	AEPI

```
factors = spark.read.csv(["data/stocks/ABAX.csv", "NAME", "AEPI"],
        header="true", inferSchema="true")
factors = factors.withColumn("Symbol", fun.input_file_name()).
        withColumn("Symbol", fun.element_at(fun.split("Symbol", "\."), 1))

from pyspark.sql import Window

stocks = stocks.withColumn('count', fun.count('Symbol').
        over(Window.partitionBy('Symbol'))).filter(
        fun.col('count') > 260*5 + 10)
spark.sql("set spark.sql.legacy.timeParserPolicy=LEGACY")
stocks = stocks.withColumn('Date', fun.to_date(fun.to_timestamp(
        fun.col('Date'), 'dd-MMM-yy')))
```

```
from datetime import datetime
stocks = stocks.filter(fun.col('Date') >= datetime(2009,10,23)).
        filter(fun.col('Date') <= datetime(2014,10,23))

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))
stocks_pd_df = stocks.toPandas()
factors_pd_df = factors.toPandas()
factors_pd_df.head(5)

n_steps = 10
def my_fun(x):
    return ((x.iloc[-1] - x.iloc[0]) / x.iloc[0])
stocks_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)
stock_returns = stock_returns.reset_index().sort_values
        ('level_1').reset_index()
factors_returns = factors_returns.reset_index().sort_values
        ('level_1'),reset_index()
stocks_pd_df_with_returns = stocks_pd_df.assign(stock_returns
        = stock_returns['Close'])
factors_pd_df_with_returns=factors_pd_df.assign(factors_returns=
        factors_returns['Close'], factors_returns_squared=
        factors_returns['Close'] ** 2)
factors_pd_df_with_returns=factors_pd_df_with_returns.pivot(
        index='Date',columns='Symbol',values=['factors_returns',
        'factors_returns_squared'])
factors_pd_df_with_returns.columns=factors_pd_with_returns.colum
        to_serils().str.join('_').reset_index()[0]
factors_pd_df_with_returns = factors_pd_df_with_returns.reset_
        index()
```

```
import pandas as pd
from sklearn.linear_model import LinearRegression
stocks_factors_combined_df = pd.merge(stocks_pd_df_with_returns,
        factors_pd_df_with_returns, how="left", on="Date")
feature_columns = list(stocks_factors_combined_df.columns[4:])

with pd.option_context('mode.use_inf_as_na', True):
    stocks_factors_combined_df = stocks_factors_combined_df.
        dropna(subset=feature_columns + ['stock_returns'])

def find_els_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])

coefs_per_stock = stocks_factors_combined_df.groupby(
        'Symbol').apply(find_els_coef)
coefs_per_stock = pd.DataFrame(coefs_per_stock).reset_index()
coefs_per_stock.columns = ['symbol', 'factor_coef_list']
coefs_per_stock = pd.DataFrame(coef_per_stock.factor_coef_list.tolist(),
        index=coefs_per_stock.index, columns=['Symbol']
        +feature_columns)

samples = factors_returns.loc[factors_returns.Symbol ==
        factors_returns.Symbol.unique()[0]]['Close']
samples.plot.kde()

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.   +1      [1]]['Close']
```

```
pd.DataFrame({'f1': list(f_1)[1:1040], 'f2': list(f_2)[1:
              'f3': list(f_3)}).corr()
```

	f1	f2	f3
f1	1.000000	0.275057	-0.015415
f2	0.275057	1.000000	0.031370
f3	-0.015415	0.031370	1.000000

```
factors_returns_cov = pd.DataFrame({'f1': list(f_1)[1:10
    'f2': list(f_2)[1:1040], 'f3': list(f_3)}).cov().to_numpy
factors_returns_mean = pd.DataFrame({'f1': list(f_1)[1:1040],
    'f2': list(f_2)[1:1040], 'f3': list(f_3)}).mean()

from numpy.random import multivariate_normal
multivariate_normal(factors_returns_mean, factors_return

array([0.03773615, 0.00190841, 0.05145688])

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

from pyspark.sql.types import IntegerType

parallelism = 1000
num_trials = 1000000
base_seed = 1496
seeds = [b for b in range(base_seed, base_seed + parallelism)
seedsDF = spark.createDataFrame(seeds, IntegerType)
seedsDF = seedsDF.repartition(parallelism)

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

```
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)                                    mean_values
        random_factors = multivariate_normal(b_factors_returns
                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

udf_return = udf(calculate_trial_return, ArrayType(DoubleType()))

def calculate 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'))
trials.cache()
trials.approxQuantile('trialsreturn', [0.05], 0.0)
```

-0.010437