Monash University

FIT5202 - Data processing for Big Data

Assignment 2A: Building models to predict future retail sales

Full Name: Syed Nazmul Kabir
Username : skab0005
ID: 32312016

Part 1: Data Loading and Exploration

1.1 Data Loading

1.1.1) Explain the SparkConf object that you have created, and how do you set the enable the maximum partition (1 mark)

Explanation:

SparkConf object is created using the following parameters or settings:

- setMaster is for setting the number of processors for partitioning. Here, all processors(*) are selected for locally.
- setAppName parameter for setting an appropriate application name. Here 'Predict Future Sales' is the name of the application.

```
# Import SparkConf and SparkSession class into program
from pyspark import SparkConf
from pyspark.sql import SparkSession

# local[*]: run Spark in local mode with as many working processors
master = "local[*]"

# The `appName` field is a name to be shown on the Spark cluster UI
app_name = "Predict Future Sales"

# Setup configuration parameters for Spark
spark_conf = SparkConf().setMaster(master).setAppName(app_name). \
                set('spark.sql.files.maxPartitionBytes', '10000000')


# Create Spark Configuration Object
spark = SparkSession.builder.config(conf=spark_conf).getOrCreate()

# Create SparkSession

sc = spark.sparkContext
sc.setLogLevel('ERROR')
```

- Third parameter is about enabling the maximum partition size. This is done in the following way:

Using the object of maxPartitionBytes() from spark.sql.files class, maximum partition size is confirmed by using the value of 10 mb where :

10 mb = 10000000 bytes (decimal)

1.1.2) Explain the schema of each DataFrame you have created (e.g., 1. spark objects do you use. 2. parameters) (2 marks)

Using StructType object from 'pyspark.sql.types' class, the schema of each DataFrame is created

The StructType() object is expanded with add() method which uses following parameters:

- Column name : a string and
- Column type using any of the following objects that is appropriate and aligned with metadata.
  StructField, StringType, IntegerType, DoubleType, DateType, BooleanType

A detailed view of the schema of each dataframe is shown below:

```
schema_features = StructType() \
      .add("Store",StringType(),True) \
      .add("Date",DateType(),True) \
      .add("Temperature",DoubleType(),True) \
      .add("Fuel_Price",DoubleType(),True) \
      .add("MarkDown1",DoubleType(),True) \
      .add("MarkDown2",DoubleType(),True) \
      .add("MarkDown3",DoubleType(),True) \
      .add("MarkDown4",DoubleType(),True) \
      .add("MarkDown5",DoubleType(),True) \
      .add("CPI",DoubleType(),True) \
      .add("Unemployment",DoubleType(),True) \
      .add("IsHoliday",BooleanType(),True)


# Creating schema for the sales dataframe
schema_sales = StructType() \
      .add("Store",StringType(),True) \
      .add("Dept",StringType(),True) \
      .add("Date",DateType(),True) \
      .add("Weekly_Sales",DoubleType(),True) \
      .add("IsHoliday",BooleanType(),True)


# Creating stores schema for the sales dataframe
schema_stores = StructType() \
      .add("Store",StringType(),True) \
      .add("Type",StringType(),True) \
      .add("Size",IntegerType(),True)
```

Here, Date datatype is initially set as stringtype to import data, later Date datatype is converted to date-type using withColumn function where to_date () object from pyspark.sql.functions is used to convert datatype.

1.1.3) Explain the spark object you used to load the data into one of a dataframe (e.g., 1. spark objects do you use. 2. parameters) (1 mark)

Data is imported as dataframe using ‘format’, ‘option’,’schema’, ‘load’ objects along with spark.read

- For format object, parameter is “csv”,
- For option object, parameter is set as True for ‘header’,
- For another option object, dateFormet is set as per provided data format; i.e dd/MM/yyyy
- For schema, created schema for each DataFrame is used as the parameter,
- For load, the local path is used as the parameter.
- Here is an example of the code:

```
df_features = spark.read.format("csv") \
      .option("header", True) \
      .option("dateFormat", "dd/MM/yyyy") \
      .schema(schema_features) \
      .load("data/features.csv")
```

1.2 Data Exploration

1.2.1) Explain the process in this task (e.g., the steps in a function you have created). Any method of a pyspark object used, the parameters, and the variables involved (1 mark)

col, isnan, when and count objects are used from sql.functions class to calculate total null values for each column of a dataframe.

A function named ‘counting_null’ is created in the following ways:

An iterator iterates on each column of a dataframe(function parameter) and counts the total number of null values for any of the values of ‘None’,’NULL’,empty space, nan. Or identifies any na values using isnan() method.

To implement this, contains() or isnan() method is used with the parameter value as [‘None’,’NULL’, ‘ ’, nan] inside a when() function where each of contain/isnan is joined with ‘or’ operator. Then finally count function do aggregation of all those values (None’,’NULL’,empty space, nan). Thus, the function returns total null values.

Here is the screenshot of this user defined function:

```
def counting_null(data):

    p = data.select([count(when(col(c).contains('None') | \
                            col(c).contains('NULL') | \
                            (col(c) == '' ) | \
                            col(c).isNull() | \
                            isnan(c), c
                           )).alias(c)
                     for c in data.columns])
    return p.show()
```

Finally counting_null function is used on each of the dataframe to count number of null values for each column. In this case, Date column for features and sales dataframe is converted to string type to avoid arising an error as isnan() does not work on Date data-type. Here is the screenshot of the implementation of counting_null function on a dataframe:

```
df_features_dateString = df_features.withColumn('Date', df_features.Date.cast('string'))
counting_null(df_features_dateString)
```

```
+-----+----+-----------+----------+---------+---------+---------+---------+---------+---+------------+---------+
|Store|Date|Temperature|Fuel_Price|MarkDown1|MarkDown2|MarkDown3|MarkDown4|MarkDown5|CPI|Unemployment|IsHoliday|
+-----+----+-----------+----------+---------+---------+---------+---------+---------+---+------------+---------+
|    0|   0|          0|         0|     4158|     5269|     4577|     4726|     4140|585|         585|        0|
+-----+----+-----------+----------+---------+---------+---------+---------+---------+---+------------+---------+
```

1.2.2) Explain the process in this task (e.g., the steps in a function you have created). Any method of a pyspark object used, the parameters, and the variables involved (1 mark)

Data_details() function is created which deals with the following activities:

- Summary method is used on each of the dataframe to show the basic statistics (including count, mean, stddev, min, max, 25 percentile, 50 percentile, 75 percentile) for each numeric column of a dataframe. toPandas() method with summary() provide better dataframe structure to read the statistics; however, this is avoided as conversion in pandas is discouraged in data-processing part.

- To identify non-numeric columns in each dataframe, a list comprehension is used where an iterator iterates over each field of schema of the data-frame and excludes the column which are either double/integer type or any of the column of Date, Store or Dept.

- Exclusion is confirmed using isinstance() function which takes two parameters: specific column data type and a specific datatype. So, the function is just confirming an instance for matching a column with specific datatype.

- A screenshot of the code shown below:

```
def data_details(data):
    print("Showing summary statistics of numeric columns")
    data.summary().show()

    print("Showing first five rows of non-numeric columns")

    str_cols_features = [f.name for f in data.schema.fields
                         if not isinstance(f.dataType, DoubleType)
                         and not isinstance(f.dataType, IntegerType)
                         and f.name not in {'Date','Store','Dept'}]
    data.select(str_cols_features).show(5)

    print("Showing counts of distinct values of non-numeric columns")

    data.select(str_cols_features).groupBy(str_cols_features).count().show()
```

- The identified non-numeric column is used with in select function as a parameter and then show() command is used to display five rows using '5' as the parameter of show command.

- The value count of each element of non-numeric column of any of the dataframe is shown using groupBy and count function where the parameter of the groupBy function is just the identified non-numeric column of the dataframe.

Finally, data_details() function is used on all data frames to meet the requirements of 1.2.2

1.2.3) Explain the process in this task (e.g., the steps in a function you have created). Any method of a pyspark object used, the parameters, and the variables involved (1 mark)

a) Histogram:

Seaborn and matplotlib packages are used to create the histogram plot.
At first, Weekly_Sales column is converted to an array using rdd, flatMap() and collect() method.
Then, the array is directly called in the histplot() object of seaborn class. Y axis is set to log scale using set_yscale() method where parameter is 'log'.
Then xlabel,ylabel, title objects and show methods with appropriate values are used from matplotlib class to do labelling, titling and displaying the histogram.

```
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.ticker import ScalarFormatter

x_array = df_sales.select('Weekly_Sales').rdd.flatMap(lambda x: x).collect()

fig, axs = plt.subplots(ncols=1, figsize=(10,6))
sns.despine(fig)
fig = sns.histplot(x_array).set_yscale('log')
plt.xlabel("Weekly Sales")
plt.ylabel("Frequency")
plt.title("Distribution of Weekly Sales")
plt.show(fig)
```

b) Line plot:

Weekly_Sales and Store are present in sales data frame. However, there is

no Month column. So, Month column is extracted from the Date using withColumn function where ‘month’ object from pyspark.sql.functions class is used as a parameter for withColumn along with column name as “Month”.

```
df1_sales = df_sales.withColumn('Month', f.month('Date'))
avg_weekly_sales = df1_sales.groupBy("Store","Month"). \
                            agg(f.mean("Weekly_Sales").alias("Average Sales"))
```

```
import seaborn as sns
x_array = avg_weekly_sales.select('Month'). \
                            rdd.flatMap(lambda x: x).collect()
y_array = avg_weekly_sales.select('Average Sales'). \
                            rdd.flatMap(lambda x: x).collect()
color = avg_weekly_sales.select('Store'). \
                            rdd.flatMap(lambda x: x).collect()
```

```
fig, axs = plt.subplots(ncols=1, figsize=(12, 8))
sns.despine(fig)
fig = sns.lineplot( x=x_array, y=y_array, hue=color)
plt.xlabel("Months")
plt.ylabel("Average Weekly Sales")
plt.title("Average Weekly Sales by Month and Store")
sns.move_legend(fig, "upper center",
                bbox_to_anchor=(.45, .995),
                ncol=8, title=None, frameon=False)
plt.show(fig)
```

Then average weekly sales is obtained from modified sales data frame using groupBy for Store and Month column because we need mean aggregation of weekly sales by month and by store. The aggregation is performed using mean object from sql.functions class as parameter of agg() object.
Next, average weekly sales, Month and Store column is converted to array following the same process as mentioned in 1.2.3 a.
Seaborn and matplotlib packages are used to create the line plot which used. At first, figure size and plot orientation are defined using subplots object from matplotlib. Then lineplot function from seaborn package is used to create the figure where parameters are x axis for Month, y-axis for average weekly sales and color for store.
At last xlabel, ylabel, title objects and show methods with appropriate values are used from matplotlib class to do labelling, titling and displaying the lineplot.

1.2.4) Describe the plot with 150 words max for each description. (3 marks)

Column Chart:
A column chart showing the relationship of department and IsHoliday with Cummulative Weekly Sales. The plot is generated using seaborn package by following the same methods as the previous two plots. Just in this case we used barplot() object.

The plot aims to find relationship between numerical data (weekly sales) and categorical values of two columns. The plot shows minimal relationships with the IsHoliday and Weekly Sales as the data is highly skewed towards non-holiday sales. On the other hand, weekly sales vary a lot from department to department. And that makes sense as sales varies a lot from different departments in practical. So, these

columns (Department and Weekly_Sales) have very high associations.

Correlation Matrix Diagram (Heat Map):

A correlation plot is created after combining all data frames which is generated by seaborn. The correlation metrics is calculated by corr() function after converting DataFrame to pandas dataframe using toPandas().
Heatmap object with the parameters correlation and annotation (True) is used.

The Heatmap clearly shows that Size (0.22) has the highest positive influence in weekly sales. MarkDown1,3,5 and Month have moderate level of positive correlations with weekly sales.

Unemployment has the highest negative impact on weekly sales (-0.39.

Temperature has hardly any influence on weekly sales.

Mark Down1 and 4 has the highest correlations (0.83) among all attributes. Other significant positive correlations are:

- Month vs MarkDown1 & MarkDown4(negative)
- Month & MarkDown3 (positive)
- Size & MarDown5 (positive)
- Temperature & Fuel Price (positive)
- Unemployment & Fuel Price (positive)
- CPI & Fuel Price (negative)
- Unemployment & CPI (negative)

Part2. Feature extraction and ML training

2.1 Discussion and creating columns.

2.1.1) Total 6 Marks in one paragraph

- Discussing the method used to show the feature importance (1 mark)
- Discussing which feature has a great impact on the label column, which should be transformed, and the reason (1 mark)
- You need to provide the reference (1 mark)
- The coherence of the explanation, compactness, completeness - 400 words (3 marks)

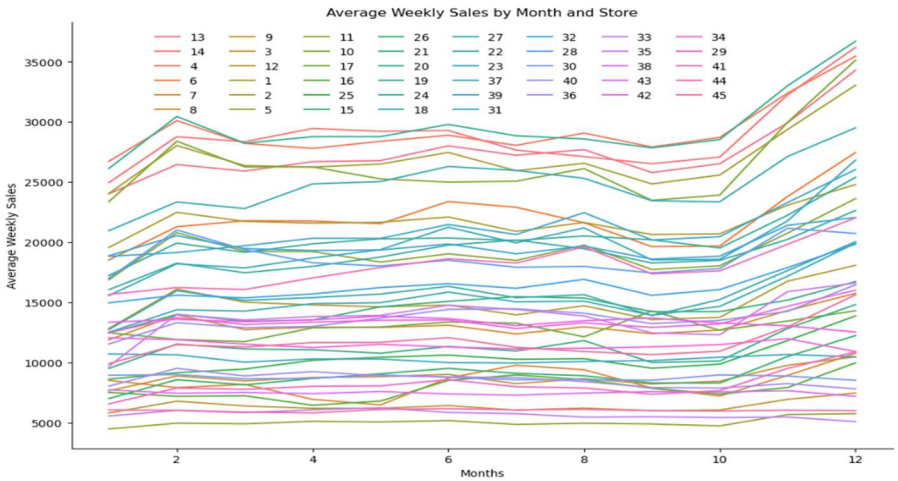
In both use cases, focus should be on weekly sales column while selecting features for the model. Because weekly sales is directly related to label ‘achieved goal’ for use case1 and will be used for prediction as the ‘previous weekly sales’ in use case 2.

Considerations of numeric features:

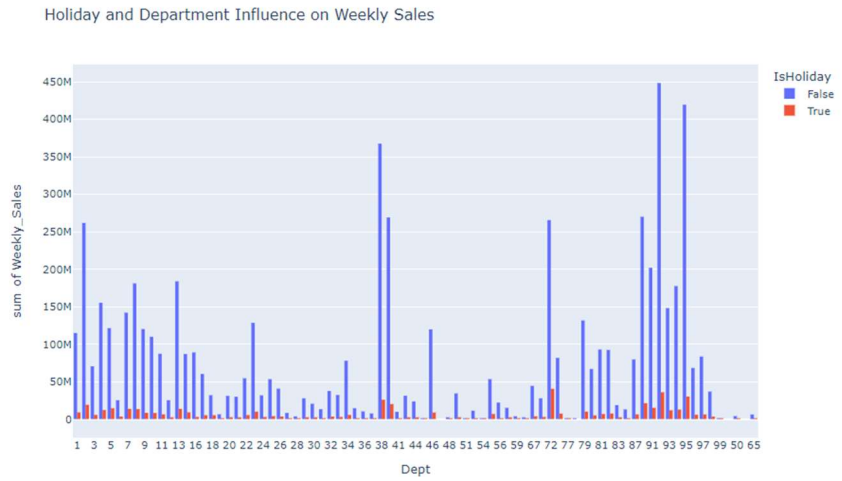
- Pearson's correlation coefficient, commonly called “correlation coefficient” is obtained by taking the ratio of the covariance of the two variables of numerical dataset, normalized to the square root of their variances.
- The values of correlation coefficient among target and other attributes determine the selection of the features for the model. That means the absolute values of higher correlation-coefficient of a feature with the target feature is the main selection criteria.
- Higher correlations among other features may impact the model performance. For example: MarkDown1 and 4 are highly correlated. But only MarkDown1 is selected for the model as it has higher correlations with target feature. Because, there is high probability that both features will bring same information to the model as those are highly correlated. This action will reduce model complexity too.

The considerations of Non-numeric features:

- As ‘Store’ is mainly an ID, so it should not be considered as a feature.
- Date itself can’t be used as the feature as it is a cumulative feature of day, month and year.
- The lineplot clearly shows that average weekly sales normally increases at the end of the year (may be due to Christmas and boxing day sales). So, extracted feature Month from ‘Date’ has specific impact on weekly sales.



- Column chart clearly shows that weekly sales vary from department to department significantly. So, Dept is an important feature. On the other hand, holiday buying is not significant enough to consider IsHoliday a prospective feature.



Selected Features:

'Size', 'Fuel_Price', 'CPI', 'Unemployment', 'MarkDown1', 'MarkDown3', 'MarkDown5', 'Month', 'Dept', 'Type', 'Previous_Weekly_Sales'(use case 2)

Required Transformation:

Categorical feature Type needs to be transformed using string indexing. It will enable encoding Type column of labels to a column of label indices.
Then one hot encoding is imposed on string indexed categorical data that helps to map categorical features represented as a label index to a binary vector thus avoiding any ordinal value impact on the model. Finally, all features will be passed through vector assembler.

Reference:

Wikipedia, *Correlation,* https://en.wikipedia.org/wiki/Correlation

Moodle, *FIT5202 Data processing for big data - Summer B 2023*. https://lms.monash.edu/course/view.php?id=149349

2.1.2) Explain the process in this task (e.g., the steps in a function you have created). Any method of a pyspark object used, the parameters, and the variables involved (2 marks)

Steps:

a) Creating Year and Month data from sales dataframe using withColumn function. It takes two parameter. First one is about naming of new column and second one is about extracting Month/Year data from Date column using month/year objects

from pyspark.sql.functions class.

b) Left outer joining sales and stores data using Store column as the key. Dropping out key column from right dataset, i.e., Store

c) In order to create 'Achieved Goal' column for use-case1, below procedures have been followed:

- At first, creating a dataframe consisting Total Sales column which is created using groupBy function on Store and Date columns. Then aggregating of Weekly_Sales using sum() object from pyspark.sql.functions class. This will create unique Total Sales for each store for each week for all departments.
- Joining Total Sales dataframe with SalesStores joined dataframe on the keys of Store and Date. This will bring total sales column within salesStores dataframe. Dropping out key columns for Total Sales data-set.
- Creating a new column named Achieved_Goal by dividing Total Sales with Size using withColumn function. Then the values of the Achieved_Goal is converted to 1 and 0 using the conditions mentioned in the requirements (Total Sales/ Size >8.5 = 1 else 0). In this case, when() method is used. Next, Total Sales column is dropped out. A screenshot of the code is given below:

```
df_salesStores  = df_salesStores. \
                        withColumn("Achieved_Goal",
                                   when((col("Total_Sales")/
                                         col("Size")) > 8.5, 1).otherwise(0)). \
                        drop(df_salesStores.Total_Sales)
```

- Finally salesStores dataframe (which contain newly created achieved goal) is joined with features dataframe to bring together all features of all dataset along with newly created label(achieved goal). The datasets are joined on Store and Date keys. Again, key columns from right data-set is dropped out.

d) To create 'Previous Weekly Sales' column for usecase2, next steps are followed:

- At first, a window is created using partitionBy object from sql.window class where partitioning columns are the parameters : Store and Dept. Then window will order DataFrame by Date column.
- Created the Previous_Weekly_Sales column from all joined dataset from the use case1. Here lag function is used along with over() method which uses the window as the parameter. These are all done as the parameters for withColum function that worked on sales dataset.
- All null values are dropped from the renovated all joined column obtained all the steps mentioned above.
- There are few impossible values in sales data as those are negative. Those are discarded.
- This is the dataset that will be used for creating test and train dataset.

2.2 Transformer/Estimators

2.2.1) Explain the process in this task (e.g., the steps in a function you have created). Any method of a pyspark object used, the parameters, and the variables involved (3 marks)

Steps:

a) StringIndexer object is used from ml.feature class to perform string indexing on categorical features. It takes two parameters: one is input columns which are categorical columns and another one is output columns which is set by adding '_index' suffix to all input columns except the Achieved_Goal column. For use case 1, Label column is appended for simplicity just prior to stringindexing.

b) The output columns of string indexing is the input column of one hot encoding where oneHotEncoder() object is used from ml.feature class to perform binary transformation of string indexing except 'label' column. Output columns is set by adding '_vec' suffix to all categorical columns.

c) Next, vectorAssembler object is used from ml.feature class on both one hot encoded categorical and all numeric columns. It assembles all vectors of a row as a single vector and placed it in a single column, here we named as features which is the outputCol parameter of vectorAssembler

```
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import OneHotEncoder
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.regression import GBTRegressor
from pyspark.ml.regression import DecisionTreeRegressor

#####################################################
categoryInputCols =['Type','Dept','Month']
numericInputCols1 = ['Size','Fuel_Price','CPI','Unemployment',
                     'MarkDown1','MarkDown3', 'MarkDown5']
numericInputCols2 = ['Size','Fuel_Price','CPI','Unemployment','MarkDown1',
                     'MarkDown3', 'MarkDown5','Previous_Weekly_Sales']
categoryOutputCol = 'Achieved_Goal'
categoryCols = categoryInputCols+[categoryOutputCol]

#################################################
outputCols1=[f'{x}_index' for x in categoryInputCols if x!='Achieved_Goal']
outputCols1.append('label')
inputIndexer1 = StringIndexer(inputCols=categoryCols, outputCols=outputCols1)

outputCols2=[f'{x}_index' for x in categoryInputCols]
inputIndexer2 = StringIndexer(inputCols=categoryInputCols, outputCols=outputCols2)

#################################################
inputCols_OHE1 = [x for x in outputCols1 if x!='label']
outputCols_OHE1 = [f'{x}_vec' for x in categoryInputCols]
encoder1 = OneHotEncoder(inputCols=inputCols_OHE1,outputCols=outputCols_OHE1)

inputCols_OHE2 = [x for x in outputCols2]# if x!='label']
outputCols_OHE2 = [f'{x}_vec' for x in categoryInputCols]
encoder2 = OneHotEncoder(inputCols=inputCols_OHE2,outputCols=outputCols_OHE2)

#####################################################
assemblerInputs1 = outputCols_OHE1 + numericInputCols1
assembler1 = VectorAssembler(inputCols = assemblerInputs1, outputCol="features1")

assemblerInputs2 = outputCols_OHE2 + numericInputCols2
assembler2 = VectorAssembler(inputCols = assemblerInputs2, outputCol='features2')

#########################################################
dt = DecisionTreeClassifier(featuresCol = 'features1', labelCol = 'label', maxDepth = 10)
gbt = GBTClassifier(featuresCol="features1",labelCol="label",  maxIter=10, maxDepth =10)

dtr = DecisionTreeRegressor(featuresCol = 'features2', labelCol = 'Weekly_Sales', maxDepth = 30)
gbtr = GBTRegressor(featuresCol = 'features2', labelCol = 'Weekly_Sales', maxDepth = 20, maxIter=10)
```

d) Next, a decision tree classifier is created for usecase-1 using DecisionTreeClassifier object from ml.classification class. The parameter is :

- featureCol which gets the value from the output column of vector

assembler,
- labelCol which takes the value of 'label' from the above operations
- maxDepth : maximum depth for decision tree. Value 10 is set.

e) In case of use case2, DecisionTreeRegressor() is used from ml.regressor class. Here labelCol is weekly sales.

f) A gradient boosting tree classifier is created for usecase-2 using GBTClassifier object from ml.classification class. The parameters are :
- featureCol which gets the value from the output column of vector assembler,
- labelCol which takes the value of 'label' from the above operations.
- maxIter : maximum iterations. Value 25 is set.

g) In case of use case2, GBTRegressor() is used from ml.regressor class. Here labelCol is weekly sales.

h) In use case2, numeric column is little bit different as it contains 'previous weekly sales' and there is no label.

2.2.2) Explain the process in this task (e.g., the steps in a function you have created). Any method of a pyspark object used, the parameters, and the variables involved (3 marks)

Steps :

a) Pipline object from pyspark.ml class is used to create a pipeline for decision tree model. The parameter is just the stages of the pipeline that are created in 2.2.1. For decision tree, the stages are : string indexing stage, next one hot encoding stage, next vector assembler and finally decision tree classifier.

```
from pyspark.ml import Pipeline

stage_1 = inputIndexer
stage_2 = encoder
stage_3 = assembler
stage_4 = dt
stage_5 = gbtr

stages_dt = [stage_1,stage_2,stage_3,stage_4]
stages_gbt = [stage_1,stage_2,stage_3,stage_5]

pipeline_dt = Pipeline(stages = stages_dt)
pipeline_gbt = Pipeline(stages = stages_gbt)
```

a) Same stages are used for creating pipeline for GBT model except the last stage. It will be GBT classifier instead of decision tree classifier.

2.3 Splitting the data.

2.3.1) Explain the process in this task (e.g., the steps in a function you have created). Any method of a pyspark object used, the parameters, and the variables involved (1

mark)

Training Data Set:
After performing feature selection, the dataset is splitted using stratified sampling for training dataset following the specifications. In this case sampleBy() object is used which takes three parameters:

- Splitting column : here it is 'Year'
- Splitting proportions based on column value :
  - for training: proportion is 2010:1, 2011:0.5, 2012:1
- Third parameter is for seed to make consistent results in sampling.

```
# Sampling the data to create training set.
train_case = df_salesStoresFeatures1.sampleBy("Year",{2010:1,2011:0.5,2012:1}, seed=234)
```

```
test_case = df_salesStoresFeatures1.join(train_case,
                                         ((df_salesStoresFeatures1.Date == train_case.Date) &
                                         (df_salesStoresFeatures1.Store == train_case.Store) &
                                         (df_salesStoresFeatures1.Dept == train_case.Dept)),
                                             how= "leftanti")
```

```
cached_train = train_case.cache()
cached_test = test_case.cache()
```

Testing Data Set:
Using left anti join, train data set is excluded from the main data set, thus test data set is obtained. Here original data set is joined with train dataset using 'leftanti' as the parameter value for how and keys are all of the Store, Date and Dept.

Caching Train and Test Set :
Cache() method is used to place the dataset in the cache memory and assigned to variable named cached_train and cached_test.

2.4 Training, Model creation, Prediction, and Evaluation

2.4.1) Briefly explain what the code does (1 mark)

For both the decision tree and GBT model,

- fit() method is used to train up the models using train data-set. So, fit() method is imposed on both models (named as model_dt and model_gbt) where parameter dataset is the training data set
- The trained up model is used to predict the label on testing dataset using transform() method. Here parameter dataset is testing dataset.

2.4.2) Total 6 Marks

- Discuss which metric is more proper for measuring the model performance (3

marks)

- Discuss which is the better model (3 marks)

Metric Selection:

Metric selection depends on business interests. Here, the business requirements are :

- predicting above-threshold events in order to give the performers good recommendations and
- reducing the chance of falsely recommending a location.

That means purpose is :

- Predicting TP more
- Reducing FP more

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

$$\text{Precision} = \frac{tp}{tp + fp}$$

$$\text{Recall} = \frac{tp}{tp + fn}$$

So, we should try to make higher true positive predictions out of all predicted positives. In that case, precision is the most important metrics for evaluating model performance. Because precision deals with both the TP and FP. That means:

- Precision will be higher if prediction of TP is more and
- Precision will be higher if FP is less also.

Whereas , if we only focus on predicting TP more (less prediction of FP is not important), then recall would have been a better choice which is not our business requirement.

On the other hand, forecasting lower threshold limit correctly is not our main priority, so accuracy is not the most appropriate metrics here. And AUC focuses on better separability between classes; however here we are focusing on accuracy of true predictions only i.e. precision.

Model Comparison:

Considering all metrics conditions GBT model outperform the Decision Tree model

Decision Tree normally suffers from being overfitted. As the number of maximum depths increases, model performance also increases by welcoming overfitting problem. Detailed description of decision tree problems is shown here:

- **Overfitting** happens for many reasons, including presence of noise and lack of representative instances. It's possible for overfitting with one large (deep) tree.
- **Bias error** happens when too many restrictions on target functions is placed. For example, restricting model result with a restricting function (e.g. a linear equation) or by a simple binary algorithm (like the true/false choices in the above tree) will often result in bias.

- **Variance error** Decision trees have high variance, which means that tiny changes in the training data have the potential to cause large changes in the final result.

On the other hand, Random Forest with GBT classifier does not suffer from overfitting model as much as decision tree model as GBT model is an ensemble model. Though it has limitation in response to noisy data, it shows less variance and GBT might be costly compared to decision tree for large dataset.

So, based on performance, consistency and overfitting issue, gradient boosting is a better model compared to decision tree model though decision tree can generate higher performance metrics at the cost of model generalization.

2.4.3) Briefly explain what the code does (2 marks)

Similar steps of 2.2.2 are followed to create models based on pipeline stages:

Steps for creating pipeline:

a) Pipline object from pyspark.ml class is used to create a pipeline for decision tree model. The parameter is just the stages of the pipeline that are created in 2.2.1. For decision tree, the stages are: string indexing stage, next one hot encoding stage, next vector assembler and finally decision tree classifier.

```
stage_1 = inputIndexer2
stage_2 = encoder2
stage_3 = assembler2
stage_4 = dtr
stage_5 = gbtr

stages_dtr = [stage_1,stage_2,stage_3,stage_4]
stages_gbtr = [stage_1,stage_2,stage_3, stage_5]

pipeline_dtr = Pipeline(stages = stages_dtr)
pipeline_gbtr = Pipeline(stages = stages_gbtr)
```

b) Same stages are used for creating pipeline for GBT model except the last stage. It will be GBT classifier instead of decision tree classifier.

Steps for using cache training data:

Training and testing set data has been cached in 2.3. To bring out the data from cache memory, unpersist() is used on cached variable as shown in the following screenshot:

```
#calling the cached train and test data
train_df = cached_train.unpersist()
test_df = cached_train.unpersist()
```

Steps for creating regression models:

Same steps of 2.4.1 are followed for creating regression models and predictions.

For both the decision tree and GBT model,

- fit() method is used to train up the models using non-cached train data-set. So, fit() method is imposed on both models (named as model_dtr and model_gbtr) where parameter for fit() is the training data set
- The trained up model is used to predict the weekly sales on testing dataset using transform() method. Here, parameter of transform() method is testing dataset.

2.4.4) Discuss which one is the better model among (2 marks)

R-squarred for both the decision tree regressor and gradient boosting tree regressor model is quite close. R-squarred for decision tree is higher than that of GBT model. So, based on R-squarred, decision tree regression model is preferable.

RMSE value for both the models are very high. The reason is that weekly sales vary from 0 to 693099 in the sales data. So, it is a big range of datum, hence the rmse value seems to be very big. If we do normalisation on rmse, it will be reduced significantly. [Ref: Stack Exchange]

Again, RMSE for decision tree regressor is relatively low (12316) compared to gradient boosting tree regressor (13189). Hence Decision Tree Regressor Model is better to perform prediction on Weekly Sales compared to Gradient Boosting Tree regressor.

2.4.5) Briefly explain what the code does (2 marks)

To get the features with each corresponding feature importance for the GBT model, following steps are followed:

- Extracting last stage from the model using stages method. The parameter is '-1' to select the last stage where the regressor itself is staying in the pipeline.

```
features = GbtrModel.stages[-1].featureImportances
```

- 'featureImportances' is used on classifier to get all the features along with importance as key value pair

For ranking the features:

- toArray() function is used to convert sparse vector to array
- pandas dataframe is used to create dataframe from the array where indexes are converted to column using reset_index() object from pandas.

- Then columns are renamed using rename() function.
- Finally sort_values on ‘Ranking’ column helps to sort Ranking in descending order by setting the function parameter as ascending = False

```
import pandas as pd
df = pd.DataFrame(features.toArray()).reset_index(). \
                    rename(columns={'index': 'Sparse Feature', 0: 'Ranking'})
```

```
df.sort_values("Ranking", ascending = False).head(10)
```

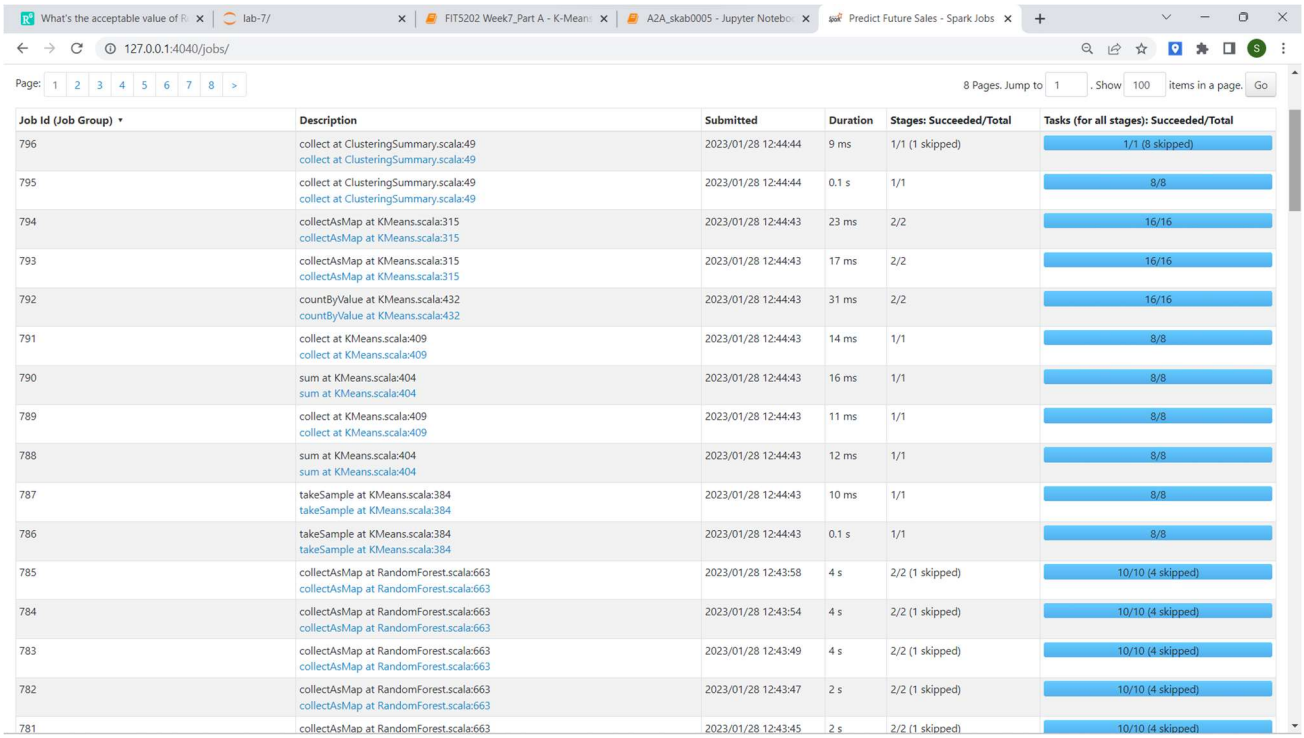
	Sparse Feature	Ranking
100	100	0.845941
93	93	0.017630
95	95	0.017381
96	96	0.014237
9	9	0.013237
97	97	0.012956
99	99	0.011205
98	98	0.010788
94	94	0.009929
37	37	0.007725

- Finally head() is called with 10 as the parameter value to select top 10 ranked features.

Part3: Knowledge Sharing

3.1.1) Discussion of the given instruction (3 marks)

A screenshot from Spark UI for running a simple Kmeans-model-training from the provided data is shown below:



11 jobs are (from job Id no. 786 to 796) observed when training the KMeans clustering model following the provided code.

3.1.2) Discussion of the given instruction (3 marks)

Main difference between data parallelism and result parallelism is about availability of one cluster in one processor or in multi-processor. In result parallelism, each processor will work on a particular target cluster data. On the other hand, in data parallelism, each processor will run k_Means locally.

So, in data parallelism, sum and count information is broadcasted to other processors to calculate centroid and there are no data movements. On the other hand, information is not shared with other partitions; rather data is transferred in result partition.

The spark UI jobs for the given codes shows that there is no data movement after placing data in initial partitions. So, it is not ‘result parallelism’.

Moreover, the runAlgorithmWithWeight() function from Spark source code of GitHub shows that in every iterations, ‘centers’ statistics are saved as a

```
// Execute iterations of Lloyd's algorithm until converged
while (iteration < maxIterations && !converged) {
  val bcCenters = sc.broadcast(centers)
  val stats = if (shouldComputeStats) {
    if (shouldComputeStatsLocally) {
      Some(distanceMeasureInstance.computeStatistics(centers))
    } else {
      Some(distanceMeasureInstance.computeStatisticsDistributedly(sc, bcCenters))
    }
  } else {
    None
  }
  val bcStats = sc.broadcast(stats)
```

broadcast variable so that the stats can be broadcasted to other nodes as long as centers are not converged. This is the characteristics of data parallelism as there is sharing of information for centroids among the nodes is happened.

So, the above discussion confirms that the given code for kmeans clustering is performed using the technique of data parallelism.

Reference:

Data Science Central, *Decision Tree vs Random Forest vs Gradient Boosting Machines: Explained Simply,* viewed on January 29, 2023
https://www.datasciencecentral.com/decision-tree-vs-random-forest-vs-boosted-trees-explained/

Moodle, *FIT5202 Data processing for big data - Summer B 2023*. Retrieved January 28, 2023
https://lms.monash.edu/course/view.php?id=149349

PojectPro, *How to plot a ROC Curve in Python?* Retrieved January 28, 2023
https://www.projectpro.io/recipes/plot-roc-curve-in-python

Stack Exchange, *What are good RMSE values?* viewed on January 29, 2023
https://stats.stackexchange.com/questions/56302/what-are-good-rmse-values

Wikipedia, *Correlation,* viewed on January 29, 2023
https://en.wikipedia.org/wiki/Correlation