FIT5202 Assignment 2 - 31125301

Analysing pedestrian traffic across the city of Melbourne and predicting activity using Apache Spark Streaming.

We have two sets of data, one with sensor locations and the other with data captured by those sensors. And aim to find out the *peak* activity hours and thus the assocaiated locations for the street art performers to maximise their audience/reach.

Use Case -

1. **Binary Classification** : To predict if the hourly count goes above 2000 steps between 9 AM and Midnight
2. **Regression** : To estimate the hourly step count between 9 AM and Midnight

Section 1 - Data Loading and Exploration

1.1 Data Loading

1.1.1 Spark Configuration

We begin by specifying the number of cores to be utilised, name of the application and the time zone.

In [1]:
```
# Import SparkConf class into program
from pyspark import SparkConf
# run Spark in local mode with as many working processors as logica
master = "local[*]"
# to be shown on the Spark cluster UI page
app_name = "FIT5202 Assignment 2 - 31125301"
# configuration parameters for Spark
spark_conf = SparkConf().setMaster(master).setAppName(app_name).set
```

Creating Spark Session using the Configurations defined above.

In [2]:
```
# Import SparkContext classes
from pyspark import SparkContext # Spark
from pyspark.sql import SparkSession # Spark SQL

# Using SparkSession to instantitate a SparkContext
spark = SparkSession.builder.config(conf=spark_conf).getOrCreate()
sc = spark.sparkContext
sc.setLogLevel('ERROR')
```

1.1.2 Data Schema

Refering the metadata file, we define schema for two empty dataframes.

In [3]:
```
from pyspark.sql.types import StructType, StructField, StringType,
# schema for pedestrian count
schema_ped = StructType([
  StructField('ID', IntegerType(), True),
  StructField('Date_Time', TimestampType(), True),
  StructField('Year', IntegerType(), True),
  StructField('Month', StringType(), True),
  StructField('Mdate', IntegerType(), True),
  StructField('Day', StringType(), True),
  StructField('Time', IntegerType(), True),
  StructField('Sensor_ID', IntegerType(), True),
  StructField('Sensor_Name', StringType(), True),
  StructField('Hourly_Counts', IntegerType(), True),
  ])

# schema for sensor locations
schema_sensor = StructType([
  StructField('sensor_id', IntegerType(), True),
  StructField('sensor_description', StringType(), True),
  StructField('sensor_name', StringType(), True),
  StructField('installation_date', DateType(), True),
  StructField('status', StringType(), True),
  StructField('note', StringType(), True),
  StructField('direction_1', StringType(), True),
  StructField('direction_2', StringType(), True),
  StructField('latitude', FloatType(), True),
  StructField('longitude', FloatType(), True),
  StructField('location', StringType(), True),
  ])
```

1.1.3 Loading Data

Now we import the data from the CSV file using the predefined schema(s).

In [4]:
```
# load data from file
df_ped = spark.read.csv("Pedestrian_Counting_System_-_Monthly__coun
df_sensor = spark.read.csv("Pedestrian_Counting_System_-_Sensor_Loc
```

And inspect the schema after transformation as:

In [5]:
```
df_ped.printSchema()
```

```
root
 |-- ID: integer (nullable = true)
 |-- Date_Time: timestamp (nullable = true)
 |-- Year: integer (nullable = true)
 |-- Month: string (nullable = true)
 |-- Mdate: integer (nullable = true)
 |-- Day: string (nullable = true)
 |-- Time: integer (nullable = true)
 |-- Sensor_ID: integer (nullable = true)
 |-- Sensor_Name: string (nullable = true)
 |-- Hourly_Counts: integer (nullable = true)
```

In [6]:
```
df_sensor.printSchema()
```

```
root
 |-- sensor_id: integer (nullable = true)
 |-- sensor_description: string (nullable = true)
 |-- sensor_name: string (nullable = true)
 |-- installation_date: date (nullable = true)
 |-- status: string (nullable = true)
 |-- note: string (nullable = true)
 |-- direction_1: string (nullable = true)
 |-- direction_2: string (nullable = true)
 |-- latitude: float (nullable = true)
 |-- longitude: float (nullable = true)
 |-- location: string (nullable = true)
```

1.1.4 Data Transformation

Refering to the first use case, we compute if the hourly count for sensor is above the threshold (2000 steps). Thus we create a new column that prints labels:

- 0 for Count $<$ 2000
- 1 for Count $\geq$ 2000

In [7]:
```
from pyspark.sql.functions import when
# add new column with label
df_ped = df_ped.withColumn('above_threshold', when(df_ped['Hourly_C
```

Inspecting data after modifacations :

In [8]:
```
# print first 5 records
df_ped.take(5)
```

Out[8]:
```
[Row(ID=2887628, Date_Time=datetime.datetime(2019, 11, 2, 4, 0), Y
ear=2019, Month='November', Mdate=1, Day='Friday', Time=17, Sensor
_ID=34, Sensor_Name='Flinders St-Spark La', Hourly_Counts=300, abo
ve_threshold=0),
 Row(ID=2887629, Date_Time=datetime.datetime(2019, 11, 2, 4, 0), Y
ear=2019, Month='November', Mdate=1, Day='Friday', Time=17, Sensor
_ID=39, Sensor_Name='Alfred Place', Hourly_Counts=604, above_thres
hold=0),
 Row(ID=2887630, Date_Time=datetime.datetime(2019, 11, 2, 4, 0), Y
ear=2019, Month='November', Mdate=1, Day='Friday', Time=17, Sensor
_ID=37, Sensor_Name='Lygon St (East)', Hourly_Counts=216, above_th
reshold=0),
 Row(ID=2887631, Date_Time=datetime.datetime(2019, 11, 2, 4, 0), Y
ear=2019, Month='November', Mdate=1, Day='Friday', Time=17, Sensor
_ID=40, Sensor_Name='Lonsdale St-Spring St (West)', Hourly_Counts=
627, above_threshold=0),
 Row(ID=2887632, Date_Time=datetime.datetime(2019, 11, 2, 4, 0), Y
ear=2019, Month='November', Mdate=1, Day='Friday', Time=17, Sensor
_ID=36, Sensor_Name='Queen St (West)', Hourly_Counts=774, above_th
reshold=0)]
```

1.2 Exploring the Data

1.2.1 Column Statistics

Here we print descriptive stats for the numeric columns, wherein we first filter the required columns and then compute the results.

In [9]:
```
df_ped['ID', 'Year', 'Mdate', 'Time', 'Sensor_ID', 'Hourly_Counts']
```

```
+-------+-----------------+------------------+------------------+-
-----------------+------------------+-----------------+
|summary|ID               |Year              |Mdate             |T
ime              |Sensor_ID         |Hourly_Counts    |
+-------+-----------------+------------------+------------------+-
-----------------+------------------+-----------------+
|count  |3435106          |3435106           |3435106           |3
435106           |3435106           |3435106          |
|mean   |1717553.5        |2016.0032330880038|15.751918863639142|1
1.459955238644746|22.978422791028866|560.7805942524044|
|stddev |991629.8312350252|3.1237869143646275|8.79918757461428  |6
.943473866829414 |16.229792156265397|809.9942576353371|
|min    |1                |2009              |1                 |0
|1                 |0                 |
|max    |3435106          |2020              |31                |2
3                 |71                |15979             |
+-------+-----------------+------------------+------------------+-
-----------------+------------------+-----------------+
```

1.2.2 Class Distribution

Now we print the number of records for each label to study the spread of data.

In [10]:
```
df_ped.groupBy('above_threshold').count().show()
```

```
+---------------+-------+
|above_threshold|  count|
+---------------+-------+
|              1| 250942|
|              0|3184164|
+---------------+-------+
```

As evident from the output table above, there are approximately 12 times more sensors with the step count less than 2000 when compared to sensors with counts greater than (or equal) to 2000.

$$\frac{3184164}{250942} = 12.68 \approx 13$$

As we can see the data is distributed in the ratio 13:1, thus there's a significant class imbalance. And this may lead to skewed predictions. For instance, I propose an algorithm that predicts label as 0 irrespective of the the features, I still achieve an accuracy of 92%. While one may suggest use of other metric like Precision, Recall or even F1 (for worst case scenario), the training phase may still be highly biased or skewed towards a specific label (1 in my example).

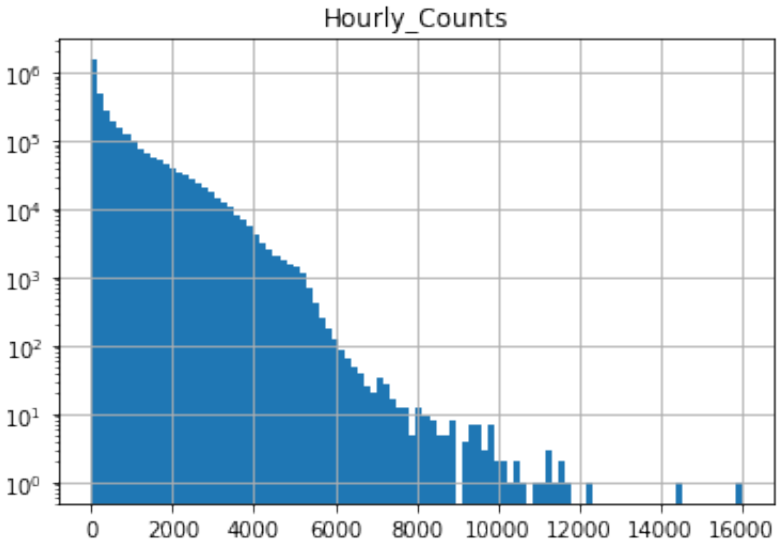
All in all, such a significant imbalance makes it difficult for the model to learn charactersistic for the minority class (label 1 in our case) as majority of models assume a balanced distribution of the target variable.

1.2.3 Histogram

Visualising the distribution of the Hourly Counts with log-scale for the frequency.

In [11]:
```
df_ped.select('Hourly_Counts').toPandas().hist(bins = 100, log='y')
```

Out[11]:
```
array([[<AxesSubplot:title={'center':'Hourly_Counts'}>]], dtype=object)
```



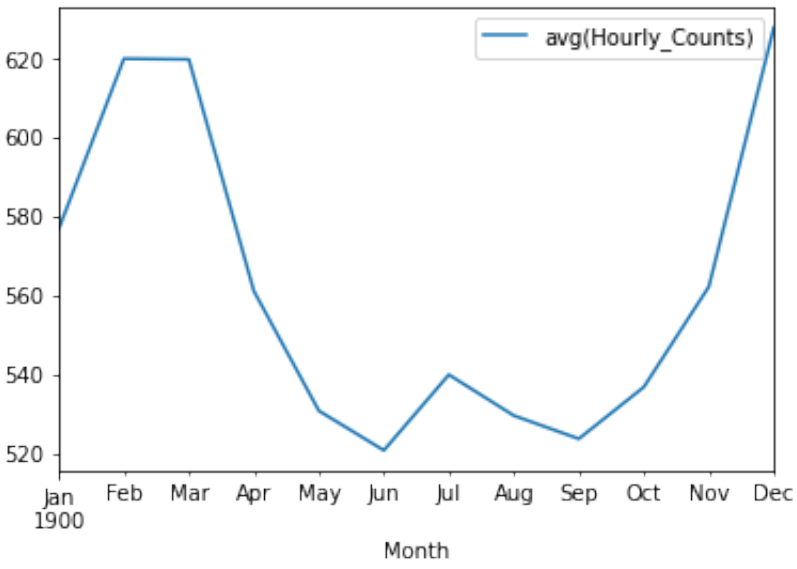
We can see the distribution is far from normal and there's an unusually large number of sensors with a count of absolute Zero. Considering the threshold is set quite low, the cummulative distributions below and above 2000 seems fine.

1.2.3 Line Plot

Visualising the trend of average daily counts across all the months in the given time-frame.

In [12]:
```
# import library to handle dates
from datetime import datetime
# group data by month
df = df_ped.groupBy('Month').mean('Hourly_Counts').toPandas()
# transform months from string to datetime
df['Month']= df.apply(lambda row: datetime.strptime(row['Month'], "
# order data according to month
df.sort_values('Month', inplace=True)
# create line plot
df.plot(x='Month', y='avg(Hourly_Counts)')
```

Out[12]:
```
<AxesSubplot:xlabel='Month'>
```



Looking at the line plot, we conclude that the average activity hits a peak twice a year. It starts increasing in September up till March and then starts declining. Similarly, the count increases during July and continues to fall till September.

Well, firstly, there's high activity during Summer months i.e. Decmber through March; plus during the month of July potentially due to winter break for students.

1.2.4 Addtional Plots

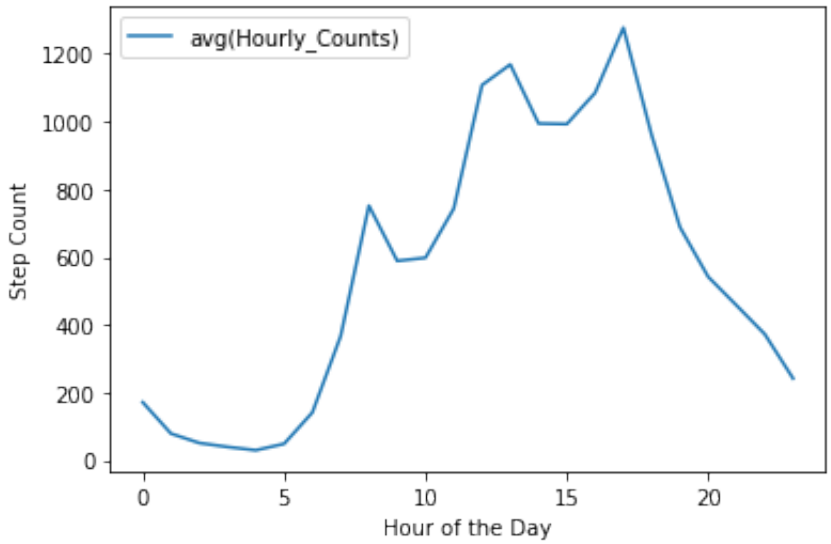
In the last sub-section, we looked at the overall count distriution (skewed) as well as the monthly change of trend. This motivated me to further visualise the data based on "Time of the Day". Quite oviosuly, we can expect high activity during business hours. So we plot the step frequency according to the Hour of the day.

To do so, we first need to mould data into an optimal format. Begining with aggregation, I group the data based on Time and then calculate the mean hourly count for each hour. Followed by sorting of data based on the Hour and then finally plotting a line curve.

In [13]:
```
import matplotlib.pyplot as plt
# aggregation
hourly_counts = df_ped.groupBy('Time').mean('Hourly_Counts').toPand
# ordering data
hourly_counts.sort_values('Time', inplace=True)

# plotting a line curve
hourly_counts.plot(x='Time', y='avg(Hourly_Counts)')
plt.xlabel('Hour of the Day')
plt.ylabel('Step Count')
```

Out[13]:
```
Text(0, 0.5, 'Step Count')
```



As expected, the step count is significantly high around office hours i.e. 8 AM and 5 PM. Addtionally, the midday rise around noon may reflect lunch time or people working half day shifts.

Similarly, we can say that day of the week also affects step count. Precisely speaking we want to see the contrast between weekdays and weekends. Just like in last plot, we process the data by

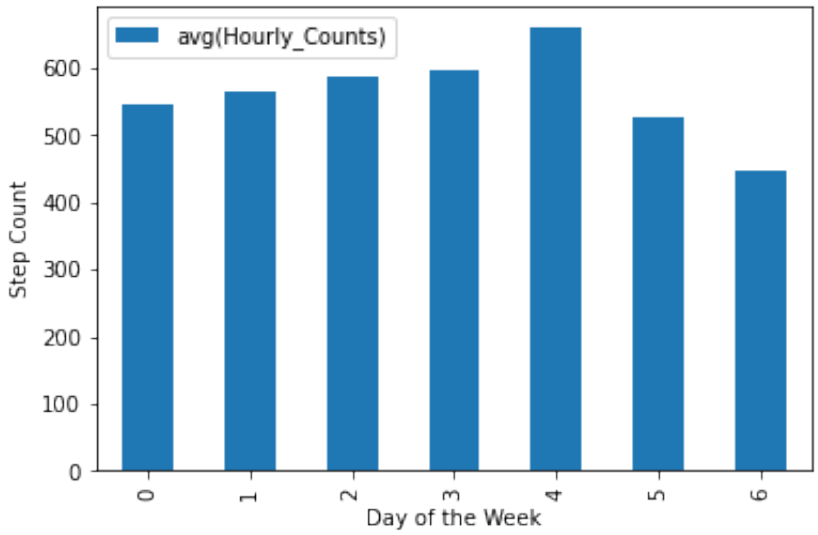
1. Aggregation
2. Sorting
3. Plotting

One complexity that we may come across is the ordering of Days. So we change the string values into categorical variables (0 for Monday, 1 for Tuesday ..... 6 for Sunday) and then plot the data.

In [14]:
```
import pandas as pd
# aggregate data on Day
daily_counts = df_ped.groupBy('Day').mean('Hourly_Counts').toPandas
# ordering data
daily_counts.sort_values('Day', inplace=True)

# cast days into categorical variable
cats = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Sa
daily_counts['Day'] = pd.Categorical(daily_counts['Day'], categorie
# sort data
daily_counts = daily_counts.sort_values('Day')
# reset index
daily_counts.reset_index(inplace=True, drop=True)

# plot data
fig, ax = plt.subplots()
daily_counts[['Day', 'avg(Hourly_Counts)']].plot.bar(ax=ax)
plt.xlabel('Day of the Week')
plt.ylabel('Step Count')
plt.show()
```



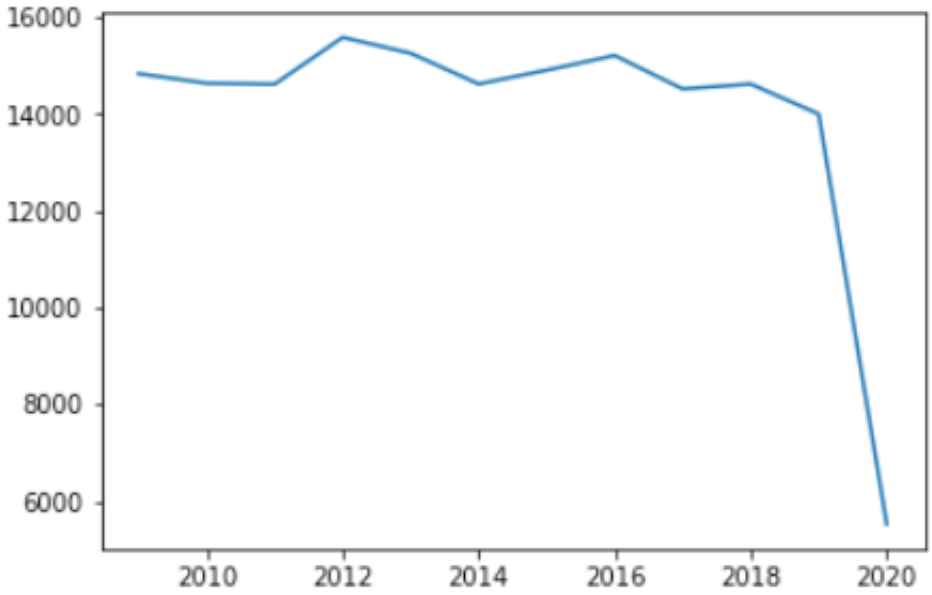
We see there's high pedestrian traffic on the Friday of each Week and a considerable low count on the Weekends (Saturday and Sunday).

Thus Day seems to directly affect the label and hence the predictions.

Section 2 - Feature Extraction & ML Training

2.1.1 Theoretical Aspect

Looking at the exploration results, I propose the use of the columns *Time, Day, Month and Sensor_ID* as features for Model training.



Referring the Yearly trend (image above) from last assignment, we already know that Year doesn't play a crucial role when looking at **average** Hourly Count. Plus the data from 2020 has been excluded from predictions, thus I decided to not include this column in training. Similarily, as Sensor ID (primary key of sensor locations dataset) encompasses the location, direction as well status of the sensor, I feel we do not need to include any columns from second dataset.

Considering the column "Time", not only it directly influences the label class but is also requied to filter data based on given conditions (9 AM to 12 AM). As the data follows 24 hour format and is in numeric form, we simply exclude the hours between 1 and 8. Similarly we also pick the Year to filter data for training and testing (section 2.3)as per the assignment requirement.

The columns Day and Month are in String Format, so we would use a StringIndexer to typecast it into categorical form.

Now we have all three columns as categorical variables but with numeric values. And this might confuse the machine about their oridinality. Thus we further implement One Hot Encoding to create Binary Vectors to avoid such a situation.

Lastly we use a Vector Assembler to put together all the features into one column which would be used to train the model.

2.1.2 Feature Engineering

Now we implement the above stated methodology via code.

To begin with, we filter rows with Time between 9 AM & Midnight and extract the required columns.

In [15]:
```
# filter data based on Time
df_ped = df_ped.filter(df_ped.Time >= 9)

# list of input columns
inputCols = ['Time','Day', 'Month', 'above_threshold']
# list comprehension for output columns
outputCols = [x+"_index" for x in inputCols]

inputCols_OHE = [x for x in outputCols if x != 'above_threshold_ind
outputCols_OHE = [x+"_vec" for x in inputCols_OHE]

inputCols_assembler = [x for x in outputCols_OHE] + ['Sensor_ID']
```

2.2.1 Transformers and Estimators

First we use the transformers *StringIndexer, OneHotEncoder and Vector Assembler* to create a new dataframe from the existing one.

Next we use the estimators Decision Tree and Gradient Boosted Tree for predicting data. At this stage we are focussing on defining these transformers/estimators and not actually implementing them.

In [16]:
```
# import relevant libraries
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import OneHotEncoder
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml.classification import GBTClassifier
from pyspark.ml.regression import GBTRegressor

# Transformer 1 - String Indexer
indexer = StringIndexer(inputCols=inputCols, outputCols=outputCols)
# Transformer 2 - One Hot Encoder
encoder = OneHotEncoder(inputCols=inputCols_OHE, outputCols=outputC
# Transformer 3 - Vector Assemble
assembler = VectorAssembler(inputCols=inputCols_assembler, outputCo
# Estimator 1 - Decision Tree Classifier
dtc = DecisionTreeClassifier(featuresCol = "features", labelCol = '
# Estimator 2 - Decision Tree Regressor
dtr = DecisionTreeRegressor(featuresCol="features", labelCol="Hourl
# Estimator 3 - Gradient Boosted Tree Classifier
gbtc = GBTClassifier(featuresCol="features", labelCol="above_thresh
# Estimator 4 - Gradient Boosted Tree Regressor
gbtr = GBTRegressor(featuresCol="features", labelCol="Hourly_Counts
```

2.2.2 Pipeline API

Now that we have defined all the transformers and estimators, we organise them in the form of sequenced stages via a Pipeline.

In [17]:
```
# USE CASE 1
# Decision Tree Classifier
pipeline1 = Pipeline(stages=[indexer, encoder, assembler, dtc])
# Gradient Boosted Tree Classifier
pipeline2 = Pipeline(stages=[indexer, encoder, assembler, gbtc])

# USE CASE 2
# Decision Tree Regressor
pipeline3 = Pipeline(stages=[indexer, encoder, assembler, dtr])
# Gradient Boosted Tree Regressor
pipeline4 = Pipeline(stages=[indexer, encoder, assembler, gbtr])
```

2.2.3 Decision Tree Classification

Hyperparameters refers to the settings that can not be learned by the model itself. And need to be specified by the user for optimal performance. In our case, we talk about the two most important hyperparameters namely maxDepth and maxBins.

MaxDepth refers to the depth of the tree. And the higher its value, lower would be the training error but may also lead to oevrfitting. And maxBin refers to the number of bins used when discretizing continous features. Having a large value for maxBins allows greater splits and thus fine grained results but significantly increases the computation cost.

Here we use K Fold cross validation to test an array of maxBin and maxDepth values and found out 10 to be the optimal value for both while maintaining a manageable run time cost.

2.3 Preparing Data

We pick entries between the years 2014 and 2018 for training our model and use the entries for the year 2019 for testing phase. To do so we first filter data based on Year and put them in separate dataframes.

In [18]:
```
# training data
train = df_ped.filter(df_ped.Year >= 2014).filter(df_ped.Year<=2018
# test data
test = df_ped.filter(df_ped.Year == 2019)
# caching data
train = train.cache()
test = test.cache()
```

Use Case 1 - Classification

2.4.1 Training Models

Now that the data has been split accordingly, we use the Pipelines defined above to learn from test dataframe and make predictions using test dataframe:

In [19]:
```
# decision tree
# training phase
pipelineModel1 = pipeline1.fit(train)
# perform predictions
predictions1 = pipelineModel1.transform(test)
```

In [20]:
```
# gradient based tree
# training phase
pipelineModel2 = pipeline2.fit(train)
# perform predictions
predictions2 = pipelineModel2.transform(test)
```

2.4.2 Evaluating Models

To check how our models perform, we print a column wise table to compar ethe expected labels and the one predicted by our model.

In [21]:
```
# results from Decision Tree
predictions1.groupBy('above_threshold_index', 'prediction').count()
```

```
+--------------------+----------+------+
|above_threshold_index|prediction| count|
+--------------------+----------+------+
|                 1.0|       1.0| 18133|
|                 0.0|       1.0|  9216|
|                 1.0|       0.0| 12908|
|                 0.0|       0.0|245275|
+--------------------+----------+------+
```

In [22]:
```
# results from Gradient Boosted Tree
predictions2.groupBy('above_threshold_index', 'prediction').count()
```

```
+--------------------+----------+------+
|above_threshold_index|prediction| count|
+--------------------+----------+------+
|                 1.0|       1.0| 10374|
|                 0.0|       1.0|  1913|
|                 1.0|       0.0| 20667|
|                 0.0|       0.0|252578|
+--------------------+----------+------+
```

While the table does give us an overview of correct predictions, we would like to scrutinise the performance metrics in a little more depth. To begin with, let's have a look at the AOC or Area Under the Curve:

In [23]:
```
# import evaluator
from pyspark.ml.evaluation import BinaryClassificationEvaluator
evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPred

# rename columns to use evaluator
predictions1 = predictions1.withColumnRenamed('above_threshold_inde
predictions2 = predictions2.withColumnRenamed('above_threshold_inde

# AOC for Decision Tree
auc_dt = evaluator.evaluate(predictions1)
# AOC for Gradient Boosted Tree
auc_gbt = evaluator.evaluate(predictions2)

print(evaluator.getMetricName(), "for Decision Tree: ", auc_dt)
print(evaluator.getMetricName(), "for Gradient Boosted Tree ", auc_
```

```
areaUnderROC for Decision Tree:  0.6849436670047464
areaUnderROC for Gradient Boosted Tree  0.8675267775433219
```

Well, AOC is defined as the value ranging between 0 and 1 which tells how well can the model differentiate between the class labels which in our case is above_threhold. The higher the score, the better. As can be seen from the results above, the Decision tree has lower AOC (0.69) when compared to that of Gradient Boosted Tree (0.87).

This simply tells us that Gradient Boosted Tree can better explain the differnce between given labels, while the simpler Decision Tree seems to have considerable overlapping.

Next up, we create a function that takes in the dataframe after predictions and returns metric values like Accuracy, Precision, Recall and F1 score for the input dataframe.:

In [24]:
```
# function to calculate performance metrics
def compute_metrics(predictions):
    # true negative
    TN = predictions.filter('prediction = 0 AND above_threshold_ind
    # true negative
    TP = predictions.filter('prediction = 1 AND above_threshold_ind
    # false negative
    FN = predictions.filter('prediction = 0 AND above_threshold_ind
    # false positive
    FP = predictions.filter('prediction = 1 AND above_threshold_ind

    # calculation logic
    accuracy = (TP + TN)/(TP + FP + FN + TN)
    precision = TP/(TP + FP)
    recall = TP /(TP + FN)
    f1 =  (2 * (precision*recall))/(precision + recall)

    return accuracy, precision, recall, f1
```

Now we use the function to print performance metrics for both the models.

In [25]:
```
dt_output = compute_metrics(predictions1)
print(f"** Decision Tree **")
print(f"accuracy: {dt_output[0]}")
print(f"precision: {dt_output[1]}")
print(f"recall: {dt_output[2]}")
print(f"f1: {dt_output[3]}")

print("\n\n")
gbt_output = compute_metrics(predictions2)
print(f"** Gradient Boosted Tree **")
print(f"accuracy: {gbt_output[0]}")
print(f"precision: {gbt_output[1]}")
print(f"recall: {gbt_output[2]}")
print(f"f1: {gbt_output[3]}")
```

```
** Decision Tree **
accuracy: 0.9225165655688329
precision: 0.6630224139822297
recall: 0.584162881350472
f1: 0.621099503339613



** Gradient Boosted Tree **
accuracy: 0.9209195466707759
precision: 0.8443069911288353
recall: 0.33420315067169226
f1: 0.47885893648449046
```

Looking at the results, both the models seems to have an equivalent accuracy but considering the imabalanced distribution of training data labels, it's very likely for the machine to predict everything to be zero. A hypothetical example would be:

I porpose an algorithm which prints 0 irrespective of the features.

This way I am bound to get high accuracy but it does not imply that approach/algorithm was correct. Thus we look at other metrics like Precision and Recal. And similar to the last scenario, it is very easy to cheat the reuslts by manipulating the distribution. So to avoid potential loopholes, I suggest the use of F1 Score.

Analysing the models in consideration, we can say that Decision Tree seems to have better F1 score (0.62 vs 0.49) and reduces the chance of false predictions. Finally, we persist this (better) model on to our machine:

In [26]:
```
pipelineModel1.save('classification_31125301')
```

2.4.3 Printing Splitting Criteria

In [27]:
```
pipelineModel1.stages[-1].extractParamMap()
```

Out[27]:
```
{Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='featuresCol', doc='features column name.'): 'features',
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='labelCol', doc='label column name.'): 'above_threshold_index',
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='predictionCol', doc='prediction column name.'): 'prediction',
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='probabilityCol', doc='Column name for predicted class conditional probabilities. Note: Not all models output well-calibrated probability estimates! These probabilities should be treated as confidences, not precise probabilities.'): 'probability',
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='rawPredictionCol', doc='raw prediction (a.k.a. confidence) column name.'): 'rawPrediction',
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='seed', doc='random seed.'): -9178414489278363 54,
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='cacheNodeIds', doc='If false, the algorithm will pass trees to executors to match instances with nodes. If true, the algorithm will cache node IDs for each instance. Caching can speed up training of deeper trees. Users can set how often should the cache be checkpointed or disable it by setting checkpointInterval.'): False,
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='checkpointInterval', doc='set checkpoint interval (>= 1) or disable checkpoint (-1). E.g. 10 means that the cache will get checkpointed every 10 iterations. Note: this setting will be ignored if the checkpoint directory is not set in the SparkContext.'): 10,
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='impurity', doc='Criterion used for information gain calculation (case-ins
```

```
y', doc='Criterion used for information gain calculation (case ins
ensitive). Supported options: entropy, gini'): 'gini',

 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='leafCol
', doc='Leaf indices column name. Predicted leaf index of each ins
tance in each tree by preorder.'): '',
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='maxBins
', doc='Max number of bins for discretizing continuous features.
Must be >=2 and >= number of categories for any categorical featur
e.'): 100,
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='maxDept
h', doc='Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 l
eaf node; depth 1 means 1 internal node + 2 leaf nodes.'): 10,
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='maxMemo
ryInMB', doc='Maximum memory in MB allocated to histogram aggregat
ion. If too small, then 1 node will be split per iteration, and it
s aggregates may exceed this size.'): 256,
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='minInfo
Gain', doc='Minimum information gain for a split to be considered
at a tree node.'): 0.0,
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='minInst
ancesPerNode', doc='Minimum number of instances each child must ha
ve after split. If a split causes the left or right child to have
fewer than minInstancesPerNode, the split will be discarded as inv
alid. Should be >= 1.'): 1,
 Param(parent='DecisionTreeClassifier_3fb9d13a319d', name='minWeig
htFractionPerNode', doc='Minimum fraction of the weighted sample c
ount that each child must have after split. If a split causes the
fraction of the total weight in the left or right child to be less
than minWeightFractionPerNode, the split will be discarded as inva
lid. Should be in interval [0.0, 0.5).'): 0.0}
```

2.4.4

As discussed the class imbalance leads to skewed predictions. And to fix that I propose two methods to improve the performance:

Considering we have ample amount of data, we can try **sampling** the data with equal distribution of above_threhold labels. Thus having balanced data (equivalent number of both labels) in the training phase may allow the model to learn better.

Secondly, we should use **Hyperparmeter Tuning** to check the best value for our various paramaters like depth, bins and number of iterations. To find out the best combination, we must use ParamGridBuilder as manual selection ad checking may take very long. Plus this tuning must go in parallel with **Cross Validation** as it will allow the machine to learn on K number sets and avoids bias.

A code snippet example for the same:

```
from pyspark.ml.tuning import ParamGridBuilder,
CrossValidator,CrossValidatorModel

from pyspark.ml.evaluation import BinaryClassificationEvaluator

dtparamGrid = (ParamGridBuilder()
             .addGrid(dt.maxDepth, [2, 5, 10, 20, 30])
             .addGrid(dt.maxBins, [10, 20, 40, 80, 100])
             .build())

dtevaluator =
BinaryClassificationEvaluator(rawPredictionCol="rawPrediction")

dtcv = CrossValidator(estimator = pipeline,
                      estimatorParamMaps = dtparamGrid,
                      evaluator = dtevaluator,
                      numFolds = 3)
```

Use Case 2 - Regression

2.4.5 Training Models

Having completed the Classification Part for above_threshold labels, we moove towards predicting the Hourly_Count for each record in the test dataframe. Similar to the previosuly stated process, we beign by training the model and then perform predictions on test data.

In [28]:
```
# decision tree
# training phase
pipelineModel3 = pipeline3.fit(train)
# perform predictions
predictions3 = pipelineModel3.transform(test)
```

In [29]:
```
# gradient based tree
# training phase
pipelineModel4 = pipeline4.fit(train)
# perform predictions
predictions4 = pipelineModel4.transform(test)
```

2.4.6 Evaluating Models

To compare the perforance of both the models, we compute the Root Mean Squared Error and R^2 values for each model:

In [30]:
```
from pyspark.ml.evaluation import RegressionEvaluator

evaluator1 = RegressionEvaluator(labelCol="Hourly_Counts", predicti
evaluator2 = RegressionEvaluator(labelCol="Hourly_Counts", predicti

rmse1 = evaluator1.evaluate(predictions3)
r1 = evaluator2.evaluate(predictions3)

rmse2 = evaluator1.evaluate(predictions4)
r2 = evaluator2.evaluate(predictions4)

print("Root Mean Squared Error (RMSE) on test data for Decion Tree =
print("R-squared on test data for Decion Tree = ", r1)
print("Root Mean Squared Error (RMSE) on test data for Gradient Boo
print("R-squared on test data for Gradient Boosted Tree = ", r2)
```

```
Root Mean Squared Error (RMSE) on test data for Decion Tree =  635
.0554253816476
R-squared on test data for Decion Tree =  0.47153750414937
Root Mean Squared Error (RMSE) on test data for Gradient Boosted T
ree =  698.114078107275
R-squared on test data for Gradient Boosted Tree =  0.361378254653
94224
```

We know that RMSE is an absolute value and R^2 represents Goodness of Fit o our model. Thus RMSE should be lower while R^2 ranges from 0 to 1 and must be higher.

Looking at the results, Decision Tree has lower RMSE and higher R^2 when compared to Gradient Boosted Tree. Thus we conclude that Decision Tree is the better model and we persist it using the save() function.

In [31]:
```
pipelineModel1.save('regression_31125301')
```

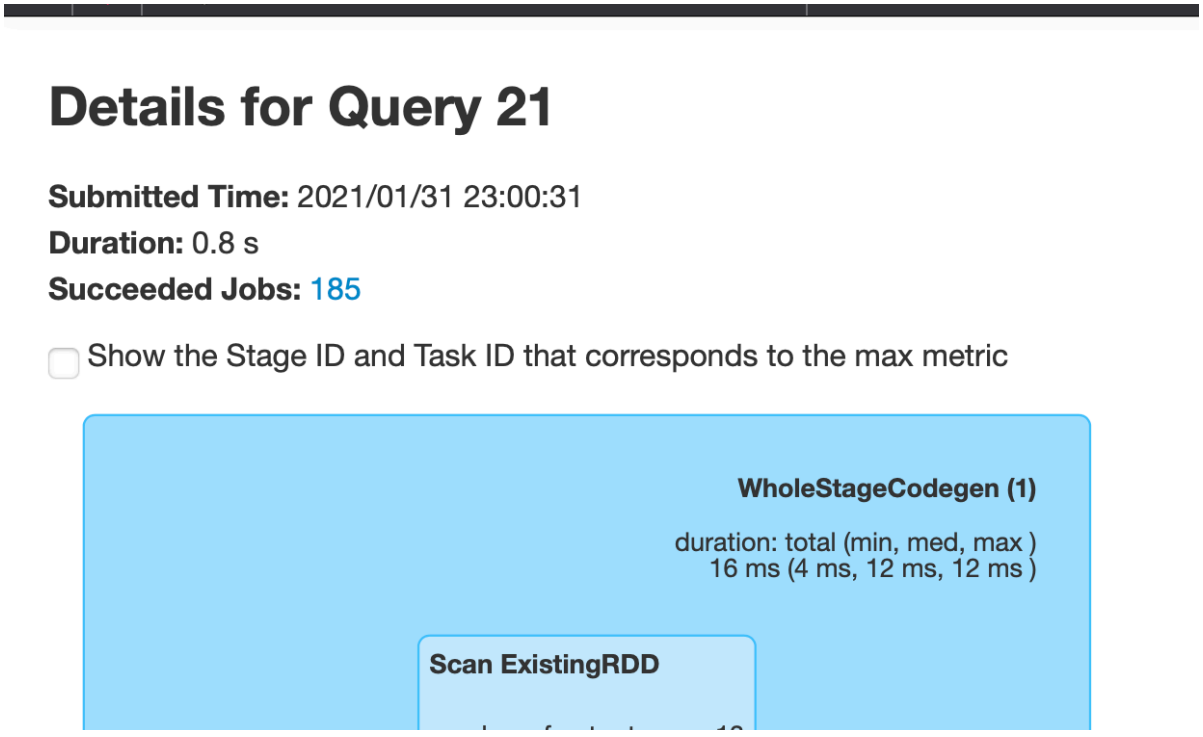
Section 3 - Clustering

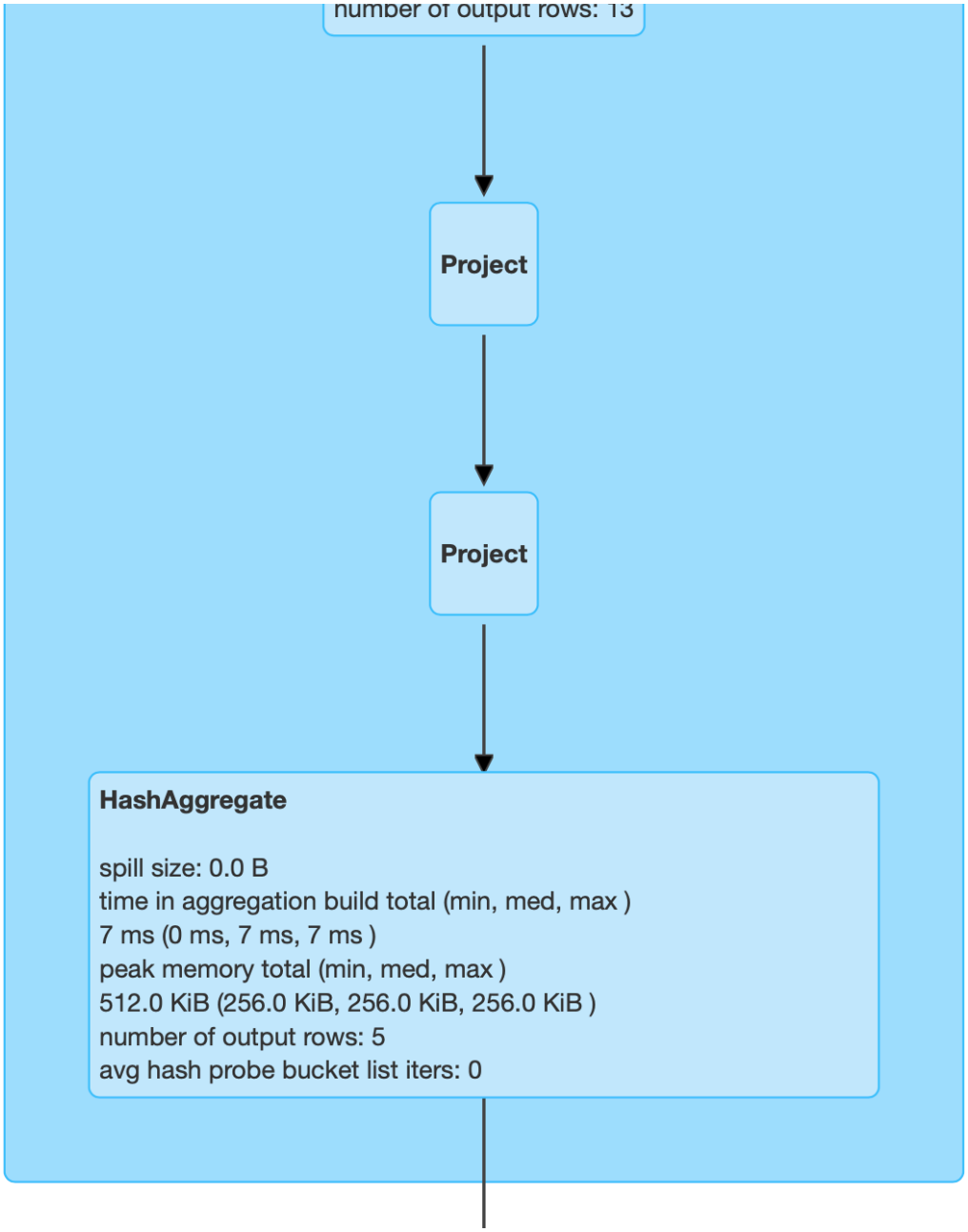
3.1 Jobs Observed

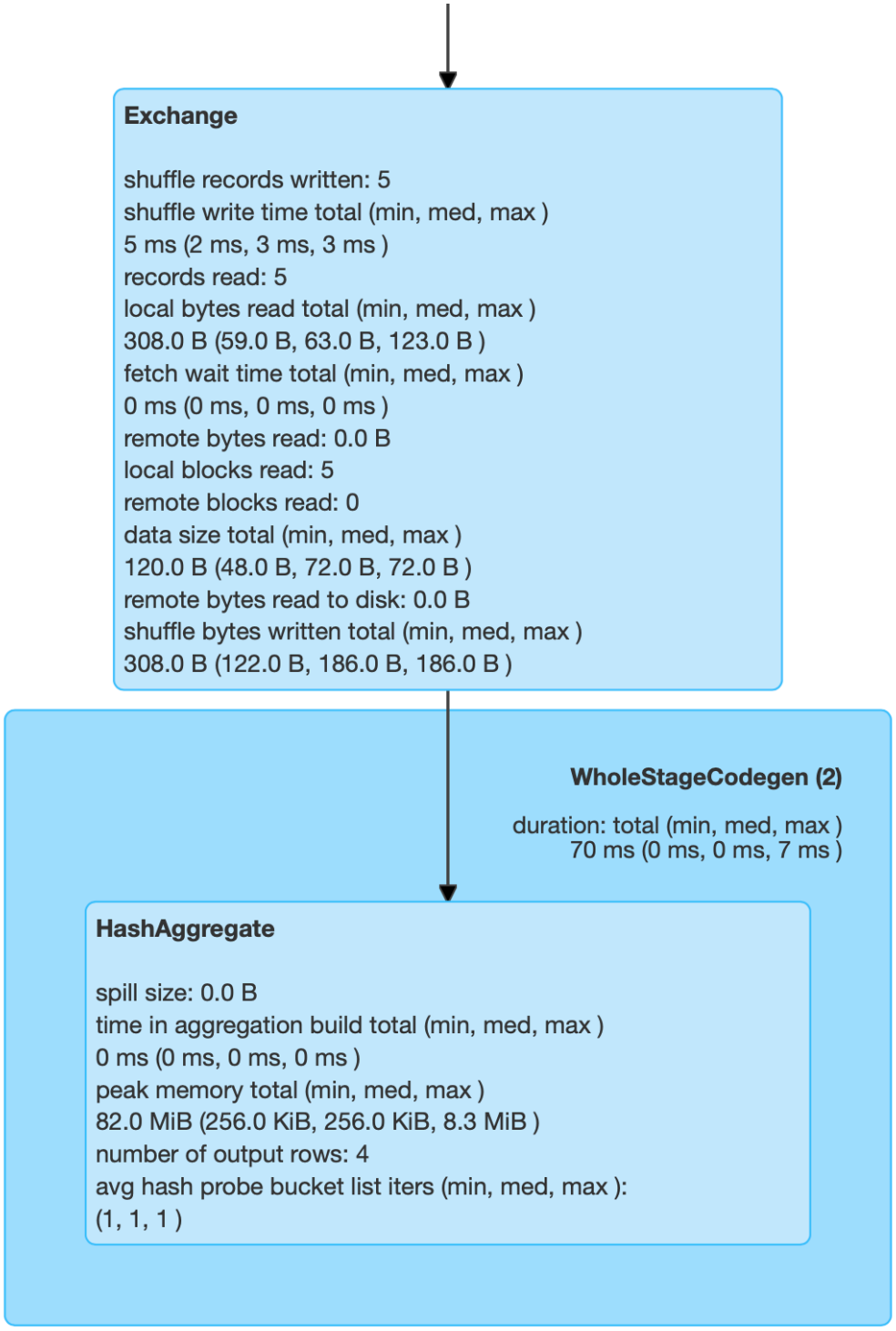
In [32]:
```
from pyspark.ml.clustering import KMeans

customer_df = spark.createDataFrame([ (0,19,15,39),
(0,21,15,81),
(1,20,16,6),
(1,23,16,77),
(1,31,17,40),
(1,22,17,76),
(1,35,18,6),
(1,23,18,94),
(0,64,19,3),
(1,30,19,72),
(0,67,19,14),
(1,35,19,99),
(1,58,20,15)],
['gender', 'age', 'annual_income', 'spending_score'])
assembler = VectorAssembler(
inputCols=['gender', 'age', 'annual_income', 'spending_score'], out
kmeans = KMeans(k=4).fit(assembler.transform(customer_df))
```

We can see 185 jobs were observed in training the KMeans Clustering. The screenshot is attahed for reference:




number of output rows: 13
Project
Project
HashAggregate
spill size: 0.0 B
time in aggregation build total (min, med, max)
7 ms (0 ms, 7 ms, 7 ms)
peak memory total (min, med, max)
512.0 KiB (256.0 KiB, 256.0 KiB, 256.0 KiB)
number of output rows: 5
avg hash probe bucket list iters: 0


Exchange
shuffle records written: 5
shuffle write time total (min, med, max)
5 ms (2 ms, 3 ms, 3 ms)
records read: 5
local bytes read total (min, med, max)
308.0 B (59.0 B, 63.0 B, 123.0 B)
fetch wait time total (min, med, max)
0 ms (0 ms, 0 ms, 0 ms)
remote bytes read: 0.0 B
local blocks read: 5
remote blocks read: 0
data size total (min, med, max)
120.0 B (48.0 B, 72.0 B, 72.0 B)
remote bytes read to disk: 0.0 B
shuffle bytes written total (min, med, max)
308.0 B (122.0 B, 186.0 B, 186.0 B)
WholeStageCodegen (2)
duration: total (min, med, max)
70 ms (0 ms, 0 ms, 7 ms)
HashAggregate
spill size: 0.0 B
time in aggregation build total (min, med, max)
0 ms (0 ms, 0 ms, 0 ms)
peak memory total (min, med, max)
82.0 MiB (256.0 KiB, 256.0 KiB, 8.3 MiB)
number of output rows: 4
avg hash probe bucket list iters (min, med, max):
(1, 1, 1)

Details

```
== Parsed Logical Plan ==
'Project [unresolvedalias('prediction, None), unresolvedalias('count, None)]
+- Aggregate [prediction#6704], [prediction#6704, count(1) AS count#6714L]
   +- Project [prediction#6704]
      +- Project [gender#6679L, age#6680L, annual_income#6681L, spending_score#6682L, features#6691, UDF(features#6691) AS prediction#6704]
         +- Project [gender#6679L, age#6680L, annual_income#6681L, spending_score#6682L, UDF(struct(gender_double_VectorAssembler_cb769a2cb5ac, cast(gender#6679L as double),
age_double_VectorAssembler_cb769a2cb5ac, cast(age#6680L as double), annual_income_double_VectorAssembler_cb769a2cb5ac, cast(annual_income#6681L as double),
spending_score_double_VectorAssembler_cb769a2cb5ac, cast(spending_score#6682L as double))) AS features#6691]
            +- LogicalRDD [gender#6679L, age#6680L, annual_income#6681L, spending_score#6682L], false

== Analyzed Logical Plan ==
prediction: int, count: bigint
Project [prediction#6704, count#6714L]
+- Aggregate [prediction#6704], [prediction#6704, count(1) AS count#6714L]
   +- Project [prediction#6704]
      +- Project [gender#6679L, age#6680L, annual_income#6681L, spending_score#6682L, features#6691, UDF(features#6691) AS prediction#6704]
         +- Project [gender#6679L, age#6680L, annual_income#6681L, spending_score#6682L, UDF(struct(gender_double_VectorAssembler_cb769a2cb5ac, cast(gender#6679L as double),
age_double_VectorAssembler_cb769a2cb5ac, cast(age#6680L as double), annual_income_double_VectorAssembler_cb769a2cb5ac, cast(annual_income#6681L as double),
spending_score_double_VectorAssembler_cb769a2cb5ac, cast(spending_score#6682L as double))) AS features#6691]
            +- LogicalRDD [gender#6679L, age#6680L, annual_income#6681L, spending_score#6682L], false

== Optimized Logical Plan ==
Aggregate [prediction#6704], [prediction#6704, count(1) AS count#6714L]
+- Project [UDF(features#6691) AS prediction#6704]
   +- Project [UDF(struct(gender_double_VectorAssembler_cb769a2cb5ac, cast(gender#6679L as double), age_double_VectorAssembler_cb769a2cb5ac, cast(age#6680L as double),
annual_income_double_VectorAssembler_cb769a2cb5ac, cast(annual_income#6681L as double), spending_score_double_VectorAssembler_cb769a2cb5ac, cast(spending_score#6682L as double))) AS features#6691]
      +- LogicalRDD [gender#6679L, age#6680L, annual_income#6681L, spending_score#6682L], false

== Physical Plan ==
*(2) HashAggregate(keys=[prediction#6704], functions=[count(1)], output=[prediction#6704, count#6714L])
+- Exchange hashpartitioning(prediction#6704, 200), true, [id=#994]
   +- *(1) HashAggregate(keys=[prediction#6704], functions=[partial_count(1)], output=[prediction#6704, count#6720L])
      +- *(1) Project [UDF(features#6691) AS prediction#6704]
         +- *(1) Project [UDF(struct(gender_double_VectorAssembler_cb769a2cb5ac, cast(gender#6679L as double), age_double_VectorAssembler_cb769a2cb5ac, cast(age#6680L as double),
annual_income_double_VectorAssembler_cb769a2cb5ac, cast(annual_income#6681L as double), spending_score_double_VectorAssembler_cb769a2cb5ac, cast(spending_score#6682L as double))) AS features#6691]
            +- *(1) Scan ExistingRDD[gender#6679L,age#6680L,annual_income#6681L,spending_score#6682L]
```

3.2 Data Parallelism

I believe *Data Parallelism* is adopted in the implementation of KMeans Clustering in Spark. Wherein the data is split among the processors (2 in our case) then they are separately clustered in each processor. Lastly the results for each cluster from the individual processors are combined at the end by a manager node.

In [ ]:

FIT5202 Assignment 2B - 31125301

Section 1 : Producing the Data

To begin with, we are given two CSV files -

1. Pedestrian data collected by electronic sensors in December 2020
2. Geographic (and other) attributes of each sensor collecting data

Here, we read the first file using a *dict reader* which returns a list of dictionaries.

Next, we need to send these dictionary records in batches wherein each batch contains data for 1 specific day. Thus we iteratively run a script to extract records depending upon the **Mdate** value. Now this Mdate attribute refers to the Day of the Month which ranges from 1 to 31 (number of days in December).

Finally, an **array** of dict-based records for **each day** is sent to the broker. And this marks the successful creation of a Kafka Producer. This data will be consumed by a streaming service defined in another notebook.

In [28]:
```
# import statements
from time import sleep
from json import dumps
from kafka import KafkaProducer
import datetime as dt
import csv

# function to read data from file and return it as a list
def readCSVFile(fileName):
    # create empty list
    data_list=[]
    # open file
    with open(fileName, 'rt') as f:
        # read file in a dictionary format
        reader = csv.DictReader(f)
        # traverse through each row
        for row in reader:
            # pick value for required keys and add it to the empty
            data_list.append({'ID':str(row['ID']),'Date_Time':str(r
                              'Year':str(row['Year']),'Month':str(r
                             'Mdate':str(row['Mdate']), 'Day':str(r
                             'Time':str(row['Time']),'Sensor_ID':st
                             'Sensor_Name':str(row['Sensor_Name']),
    # return a list of dictionaries
    return data_list

# function to publish message to broker
def publish_message(producer_instance, topic_name, data):
```

```
    try:
        producer_instance.send(topic_name, data)
        print('Message published successfully. Data : \n' + str(dat
    except Exception as ex:
        print('Exception in publishing message.')
        print(str(ex))

# function to connect to producer
def connect_kafka_producer():
    _producer = None
    try:
        _producer = KafkaProducer(bootstrap_servers=['localhost:909
                                  value_serializer=lambda x: dumps(
                                  api_version=(0, 10))
    except Exception as ex:
        print('Exception while connecting Kafka.')
        print(str(ex))
    finally:
        return _producer

if __name__ == '__main__':

    # name of the topic to publish
    topic = 'pedstream'
    # name of the file containing data
    file = 'Streaming_Pedestrian_December_counts_per_hour.csv'
    # importing data from file using function defined above
    cRows = readCSVFile(file)

    print('Publishing records..')
    producer = connect_kafka_producer()

    # loop through each day in the month of december
    for day in range(1,32):
        # create empty list
        data = []
        # traverse through each records
        for row in cRows:
            # check if day matches the upper loop
            if row['Mdate']==str(day):
                # add this records to a list
                data.append(row)
        # call function to publish data to broker
        publish_message(producer, topic, data)
        # send records after 5 seconds
        sleep(5)
```

```
Publishing records..
Message published successfully. Data :
[{'ID': '3435630', 'Date_Time': '12/01/2020 08:00:00 AM', 'Year':
'2020', 'Month': 'December', 'Mdate': '1', 'Day': 'Tuesday', 'Time
': '8', 'Sensor_ID': '39', 'Sensor_Name': 'Alfred Place', 'Hourly_
Counts': '83'}, {'ID': '3435798', 'Date_Time': '12/01/2020 11:00:0
0 AM', 'Year': '2020', 'Month': 'December', 'Mdate': '1', 'Day': '
Tuesday', 'Time': '11', 'Sensor_ID': '13', 'Sensor_Name': 'New Qua
```

FIT5202 Assignment 2B - 31125301

Section 2 : Streaming Application using Spark Structured Streaming

First we import all the required libraries or modules for our application:

In [1]:
```
# import required libraries
import os
from pyspark import SparkConf
from pyspark import SparkContext # Spark
from pyspark.sql import SparkSession # Spark SQL
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
from pyspark.sql import functions as F
from pyspark.sql.types import *
from pyspark.ml.pipeline import PipelineModel
```

2.1 Next we create a Spark Session by configuring the number of cores, application name and the time zone for our processing.

In [2]:
```
# set os environment
os.environ['PYSPARK_SUBMIT_ARGS'] = '--packages org.apache.spark:sp

# to run Spark in local mode with as 2 logical cores
master = "local[2]"
# application name to be shown on the Spark cluster UI page
app_name = "FIT5202 Assignment 2B - 31125301"
# configuration parameters for Spark
spark_conf = SparkConf().setMaster(master).setAppName(app_name).set
# Using SparkSession to instantitate a SparkContext
spark = SparkSession.builder.config(conf=spark_conf).getOrCreate()
sc = spark.sparkContext
sc.setLogLevel('ERROR')
```

2.2 Refering the metadata file, we define the schema for the file conatining sensors' location data.

In [3]:
```
from pyspark.sql.types import StructType, StructField, StringType,
# schema for sensor locations
schema_sensor = StructType([
  StructField('sensor_id', IntegerType(), True),
  StructField('sensor_description', StringType(), True),
  StructField('sensor_name', StringType(), True),
  StructField('installation_date', DateType(), True),
  StructField('status', StringType(), True),
  StructField('note', StringType(), True),
  StructField('direction_1', StringType(), True),
  StructField('direction_2', StringType(), True),
  StructField('latitude', FloatType(), True),
  StructField('longitude', FloatType(), True),
  StructField('location', StringType(), True),
  ])
```

Now load the file into a dataframe variable using the schema defined above.

In [4]:
```
df_sensor = spark.read.csv("Pedestrian_Counting_System_-_Sensor_Loc
```

2.3 Here we want to ingest the data generated by the Kafka Producer. So we set the topic name and host-address to be same as that of Producer. And create a stream for incoming data.

In [5]:
```
# topic name
topic = "pedstream"
# spark streaming
df = spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "127.0.0.1:9092") \
  .option("subscribe", topic) \
  .load()
```

2.4 As a general approach to a streaming application, we store the data streamed above in its raw for using a *parquet* sink.

In [6]:
```
# persist streaming data
query_file_sink = df.writeStream.format("parquet")\
        .outputMode("append")\
        .option("path", "parquet/pedstream_df")\
        .option("checkpointLocation", "parquet/pedstream_df/checkpo
        .start()
```

In [7]:
```
# stop query
query_file_sink.stop()
```

2.5 As we mention raw data, we want to transform the attributes to specific data types like datetime and integer so that we can process the same. Thus we refer the metadata and convert each column into its desired format. So we pick the key and value columns and transform them into string type:

In [8]:
```
df = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
```

We crosscheck the transformation by printing the schema:

In [9]:
```
df.printSchema()
```

```
root
 |-- key: string (nullable = true)
 |-- value: string (nullable = true)
```

2.6 Now we use the metadata file to create a schema.

Note: The data as published by producer is in string format. So we define each attribute as StringType for bug free ingestion and later typecast them.

In [10]:
```
# schema for December data
schema_ped = ArrayType(StructType([
  StructField('ID', StringType(), True),
  StructField('Date_Time', StringType(), True),
  StructField('Year', StringType(), True),
  StructField('Month', StringType(), True),
  StructField('Mdate', StringType(), True),
  StructField('Day', StringType(), True),
  StructField('Time', StringType(), True),
  StructField('Sensor_ID', StringType(), True),
  StructField('Sensor_Name', StringType(), True),
  StructField('Hourly_Counts', StringType(), True)
  ]))
```

Hence the schema is used to import the data as *json array* from the broker.

In [11]:
```
df = df.select(F.from_json(F.col("value").cast("string"), schema_pe
```

Again we cross check the structure of our dataframe:

In [12]:
```
df.printSchema()
```

```
root
 |-- parsed_value: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- ID: string (nullable = true)
 |    |    |-- Date_Time: string (nullable = true)
 |    |    |-- Year: string (nullable = true)
 |    |    |-- Month: string (nullable = true)
 |    |    |-- Mdate: string (nullable = true)
 |    |    |-- Day: string (nullable = true)
 |    |    |-- Time: string (nullable = true)
 |    |    |-- Sensor_ID: string (nullable = true)
 |    |    |-- Sensor_Name: string (nullable = true)
 |    |    |-- Hourly_Counts: string (nullable = true)
```

As we are dealing with JSON Array, each published batch consists of a list of records and is thus ingested in a nested dataframe. Thus we manually flatten it for further processing.

In [13]:
```
# to flatten
df = df.select(F.explode(F.col("parsed_value")).alias('unnested_val
```

After flatteing, we rename the columns as:

In [14]:
```
df_formatted = df.select(
                    F.col("unnested_value.ID").alias("ID"),
                    F.col("unnested_value.Date_Time").alias("Date_T
                    F.col("unnested_value.Year").alias("Year"),
                    F.col("unnested_value.Month").alias("Month"),
                    F.col("unnested_value.Mdate").alias("Mdate"),
                    F.col("unnested_value.Day").alias("Day"),
                    F.col("unnested_value.Time").alias("Time"),
                    F.col("unnested_value.Sensor_ID").alias("Sensor_
                    F.col("unnested_value.Sensor_Name").alias("Sens
                    F.col("unnested_value.Hourly_Counts").alias("Ho
                )
```

In [15]:
```
df_formatted.printSchema()
```

```
root
 |-- ID: string (nullable = true)
 |-- Date_Time: string (nullable = true)
 |-- Year: string (nullable = true)
 |-- Month: string (nullable = true)
 |-- Mdate: string (nullable = true)
 |-- Day: string (nullable = true)
 |-- Time: string (nullable = true)
 |-- Sensor_ID: string (nullable = true)
 |-- Sensor_Name: string (nullable = true)
 |-- Hourly_Counts: string (nullable = true)
```

Finally the data is transformed into the desired structure but still requires the type conversion. So we look up the metadat file for each column:

In [16]:
```
df_formatted = df_formatted.selectExpr("CAST(ID AS INT)", \
                                       "TO_DATE(CAST(UNIX_TIMESTAMP
                                       "CAST(Year AS INT)", "CAST(M
                                       "CAST(Time AS INT)", "CAST(S
```

2.6 Following the assignment specification for the pretrained Model, we pick the columns:

1. Sensor ID
2. Date_Time
3. Hourly_Counts

to create a new dataframe.

In [17]:
```
df_next = df_formatted.selectExpr("Sensor_ID",\
                                  "date_add(to_date(Date_Time,'MM-d
                                  "Time",\
                                  "Hourly_Counts AS prev_count")
```

Yet, the dataframe needs addtional attributes derived from the next_date column to be fed into the given Model.

In [18]:
```
df_final = df_next.selectExpr("Sensor_ID" ,\
                              "weekofyear(next_date) AS next_day_we
                              "dayofmonth(next_date) AS next_Mdate"
                              "dayofweek(next_date) AS next_day_of_
                              "Time",\
                              "prev_count")
```

2.7 Now the data is in the form as required by the Model but we only need predictions for sensor activity after 9 AM. Thus we filter the rows accordingly.

In [19]:
```
# remove records before 9 am
df_final = df_final.filter(df_final.Time >= 9)
```

Finally, the model is loaded into our Jupyter Notebook as :

In [20]:
```
model = PipelineModel.load('count_estimation_pipeline_model/')
```

And the model is fit on the dataframe created in the last to last cell. The predictions from the same are persisted in the parquet format.

In [21]:
```
# predict data using model
predictions = model.transform(df_final)
```

In [22]:
```
# perist the predictions
query_file_sink = df_final.writeStream.format("parquet")\
        .outputMode("append")\
        .option("path", "parquet/pedstream_predictions")\
        .option("checkpointLocation", "parquet/pedstream_prediction
        .start()
```

In [23]:
```
query_file_sink.stop()
```

2.8a As we want the number of hours for each sensor where the step count was greater than 2000, we first discard the rows with count less than 2000.

In [24]:
```
predictions_filtered = predictions.filter(predictions.prediction>20
```

Now we group by the ID and Day and use count to retrieve the total number of hours.

In [25]:
```
windowedCounts = predictions_filtered \
    .groupBy("Sensor_ID", "next_Mdate")\
    .agg(F.count("Sensor_ID").alias("total"))\
    .orderBy("Sensor_ID")\
    .select("Sensor_ID","next_Mdate","total")
```

In [32]:
```
query = windowedCounts \
    .writeStream \
    .outputMode("complete") \
    .format("memory") \
    .trigger(processingTime='5 seconds') \
    .queryName("tableName")\
    .start()
```

In [30]:
```
spark.sql("select * from tableName").show()
```

```
+---------+----------+-----+
|Sensor_ID|next_Mdate|total|
+---------+----------+-----+
|        1|        25|    3|
|        1|        20|    8|
|        1|        22|    7|
|        1|        23|    7|
|        1|        21|    8|
|        1|        24|    5|
|        2|        22|    7|
|        2|        21|    7|
|        2|        20|    8|
|        2|        25|    4|
|        2|        24|    5|
|        2|        23|    6|
|        4|        22|    8|
|        4|        24|   12|
|        4|        23|    7|
|        4|        20|   12|
|        4|        21|   10|
|        4|        25|    7|
|        4|        19|    5|
|        5|        22|    6|
+---------+----------+-----+
only showing top 20 rows
```

In [31]:
```
query.stop()
```

2.8b As we have already filtered the rows with step count more than 2000, we create another 2 dataframes which just has :

1. Sensor ID as key and Predicted Step Count as Value
2. Sensor ID as key and location co-ordinates as Value

And join them to create a stream to be written to Kafka.

In [36]:
```
predictions_df = predictions_filtered.selectExpr("CAST(Sensor_ID AS
location_df = df_sensor.selectExpr("CAST(Sensor_ID AS STRING) AS ke
```

In [37]:
```
joined_df = predictions_df.join(location_df,F.expr("""key_pred == k
            .selectExpr("key_pred AS key","value_loc AS value")
```

This combined stream with an inner join is sent to Kafka with a new topic name "predictionStream".

In [40]:
```
ds = joined_df \
  .selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)") \
  .writeStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "localhost:9092") \
  .option("checkpointLocation", "parquet/pedstream_df/checkpoint")\
  .option("topic", "predictionStream") \
  .start()
```

In [41]:
```
# query.stop()
```

FIT5202 Assignment 2B - 31125301

Section 3 - Consuming Data using Kafka

In this notebook we consume the data from Kafka Stream and display the sensor locations on a map. To do so, we first install the library **Folium** which has been commented out for now.

In [1]:
```
# !pip install folium
```

Once the libarary is installed we configure a consumer to retrieve data from the Stream, process the messages published in each batch and then plot them on a map.

In [1]:
```
# import required libraries
from time import sleep
from kafka import KafkaConsumer
from json import loads
import datetime as dt
import folium

# topic name
topic = 'predictionStream'

# function to subscribe to broker
def connect_kafka_consumer():
    _consumer = None
    try:
         _consumer = KafkaConsumer(topic,
                                   consumer_timeout_ms=10000, # sto
                                   auto_offset_reset='latest', # \t
                                   bootstrap_servers=['localhost:90
                                   api_version=(0, 10))

    except Exception as ex:
        print('Exception while connecting Kafka')
        print(str(ex))
    finally:
        return _consumer

# function to process incoming stream
def consume_messages(consumer):
    # create empty list
    lat, long = [], []
    print('Waiting for messages')
    # loop through each message in batch
    for message in consumer:
        # extract the location co-ordinates
        data = message.value.decode('ascii').split(',')
```

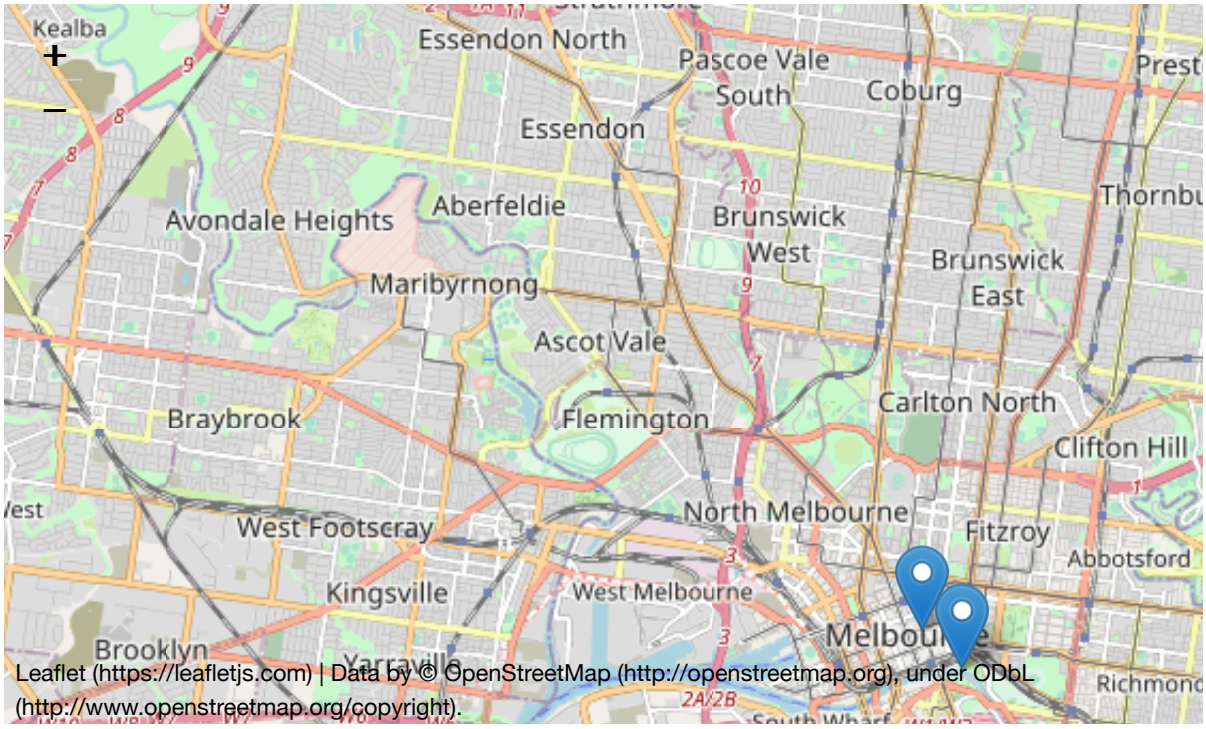
```
        data = message.value.decode('ascii').split(',')
        # add longitude and lattitude to separate lists
        lat.append(float(data[0][1:]))
        long.append(float(data[1][:-1]))
#         print(lat,long)

        # plot on map if more than 10 co-ordinates
        if len(lat) > 10:
            # intiate an empty map
            m = folium.Map(location=[20,0], zoom_start=2)
            # add all co-oridates in list to map one by one
            for i in range(0,len(data)):
                folium.Marker([lat[i], long[i]]).add_to(m)
            lat.pop()
            long.pop()
            # print map
            display(m)


if __name__ == '__main__':

    # sunscribe to broker
    consumer = connect_kafka_consumer()
    # process and plot the incoming data
    consume_messages(consumer)
```

Waiting for messages



FIT 5202 Assignment 2B Feedback Sheet					
Student Name: SIMRAN SINGH GULATI					
Marked By: David C.					
Part B: Building Streaming applications					
Tasks	**Criteria**	**Yes**	**Partial**	**No**	**Comments**
1 Producing the data	Read data from file	☑	☐	☐	
	Create producer	☑	☐	☐	
	Extract each day's data	☑	☐	☐	
	All sensor ids present in each batch	☑	☐	☐	
	Data sent in String format	☑	☐	☐	
	Data being sent every 5 seconds	☑	☐	☐	
2.1 Spark Session	SparkSession using SparkConf, with two cores, appropriate name, and spark.sql. session.timeZone	☑	☐	☐	
2.2 Load sensor location data	Define sensor location dataset schema correctly	☑	☐	☐	
	Sensor location CSV file loaded using schema	☑	☐	☐	
2.3 Ingest streaming data	Streaming count data load from Kafka	☑	☐	☐	
2.4 Persist raw streaming data	Persist the raw streaming data into parquet file sink	☑	☐	☐	
2.5 Transform streaming data	Value transformed properly accordingly metadata file	☑	☐	☐	
2.6 Prepare data for ML prediction	Create "next_date" column correctly	☑	☐	☐	
	Create "next_Mdate" column correctly	☑	☐	☐	
	Create "next_day_week" column correctly	☑	☐	☐	
	Create "next_day_of_week" column correctly	☐	☐	☑	- First day of week should be Monday, if using dayofweek function, need to do more transformation for this
	Rename "Hourly_Count" column	☑	☐	☐	

2.7 Predict next day hourly counts	Load machine learning models	☑	☐	☐	
	Predict next day's hourly counts	☑	☐	☐	
	Persist the prediction result in parquet file sink	☑	☐	☐	
2.8a Monitor above threshold sensors daily	Filter the rows	☑	☐	☐	
	Groupby Sensor and time window	☑	☐	☐	
	Get the number of hours	☑	☐	☐	
	Output the streaming result to notebook	☑	☐	☐	
2.8b Output prediction to Kafka	Filter the rows	☑	☐	☐	
	Join the stream based on sensor ID	☑	☐	☐	
	Create value column for writing to Kafka	☑	☐	☐	
	Write the stream back to Kafka sink (instead of using Kafka producer)	☑	☐	☐	
3 Visualise location on Map	Create consumer	☑	☐	☐	
	Read data from Kafka (from Q 2.8b)	☑	☐	☐	
	Extract the location of each data point	☑	☐	☐	
	Create the plot showing the sensor location on a map	☑	☐	☐	
	Continuously update the plot for every new batch of data	☐	☐	☑	- Implementation does not allow to update the plot as batch of data arrives
Qualitative Aspect for Notebook	Organization of tasks in jupyter notebook Adherance to python standards Use of appropriate comments, documentations, code reusability, output readability, reference	☑	☐	☐	
Interview	Satisfatory answers for interview questions				**Excellent Understanding**
	Final Grade	Late Submission		0	**HD**