part2

September 22, 2022

1 Initializations

```
[ ]: !apt-get install openjdk-8-jdk-headless -qq > /dev/null
     !wget -q https://archive.apache.org/dist/spark/spark-2.4.4/spark-2.4.
      ↪4-bin-hadoop2.7.tgz
     !tar xf spark-2.4.4-bin-hadoop2.7.tgz
     !pip install -q findspark

     import os

     os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"
     os.environ["SPARK_HOME"] = "/content/spark-2.4.4-bin-hadoop2.7"
```

```
[ ]: import findspark
     findspark.init()
     from pyspark.sql import SparkSession
     from time import time
     from dill.source import getfile

     def init_spark(app_name: str):
       spark = SparkSession.builder.appName(app_name).getOrCreate()
       sc = spark.sparkContext
       return spark, sc

     spark, sc = init_spark('proj2')
     print(sc.version)
     !unzip /content/Static\ data.zip
```

```
2.4.4
Archive:  /content/Static data.zip
  inflating: Static data/data.json
```

```
[ ]: from pyspark.sql.types import *
     SCHEMA = StructType([StructField("Arrival_Time",LongType(),True),
                          StructField("Creation_Time",LongType(),True),
                          StructField("Device",StringType(),True),
                          StructField("Index", LongType(), True),
                          StructField("Model", StringType(), True),
                          StructField("User", StringType(), True),
                          StructField("gt", StringType(), True),
                          StructField("x", DoubleType(), True),
                          StructField("y", DoubleType(), True),
                          StructField("z", DoubleType(), True)])
     df = spark.read.json('/content/Static\ data/data.json',schema=SCHEMA)
```

2 Data pre-processing

Functions to be used

```
[ ]: from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder
from pyspark.sql.types import DoubleType, ArrayType, StringType
import pyspark.sql.functions as f
from datetime import datetime
import numpy as np
from pyspark.ml.linalg import Vectors, VectorUDT

# These functions ideally could have been wrote using spark, however, I kept
# running into many errors without a soultion so I had to use udf

norm = f.udf(lambda x : sum([i ** 2 for i in x]), DoubleType())

seconds_in_day = 24 * 60 * 60

# Explained later
def sec_sin_aux(ts):
  time  = datetime.fromtimestamp(ts/1000)
  seconds_since_midnight = (time - time.replace(hour=0, minute=0, second=0,␣
 ↪microsecond=0)).total_seconds()
  return (seconds_since_midnight/seconds_in_day) * 2 * np.pi

def sec_cos_aux(ts):
  time  = datetime.fromtimestamp(ts/1000)
  seconds_since_midnight = (time - time.replace(hour=0, minute=0, second=0,␣
 ↪microsecond=0)).total_seconds()
  return (seconds_since_midnight/seconds_in_day) * 2 * np.pi


seconds_sin = f.udf(lambda x: sec_sin_aux(int(x)))
seconds_cos = f.udf(lambda x: sec_cos_aux(int(x)))
```

The following pipeline performs transformations on the data, in the end we recieve a dataframe that is ready to be used for the model.

The pipeline can be broken up into 3 stages:

1. **Time Manipulation:**

   First, we used the column ”Arrival_Time”, we converted the column from timestamp to human readable time, which came in the form of 2 columns, first represents time (H,M,S) and the second represents date.

   Since we are working with time-series data, we wanted to combat a known problem in the field, representing time as it is currently doesn't portray the cyclicity and periodicity of time, in other words we interpert `23:59:59` and `00:00:00` as very distant times when in fact they are almost the same.

   To solve this problem we first converted the time to seconds since midnight then used the following transformation

$$T(t) = (t_{cos}, t_{sin}) = (\cos(\frac{t * 2\pi}{24 * 60 * 60}), \sin(\frac{t * 2\pi}{24 * 60 * 60})) \tag{1}$$

source

This solution helped us solve the original problem, and also converts time to a format readable by the model.

Second, as discussed in the first part of the project we divided the the day into 3 segments morning, noon and night which helped us get a rough estimation of when each user's info is being logged. That was utilized using dummy variables which also was fed to the model

2. **Position Manipulation:**

We also discussed the behavior of the norm of the position of each user in relation with the activity being done, so we added another column representing the norm to the data.

3. **Final Touches:**

Since the model doesn't accept categorical values we used StringIndex to get numerical values for the columns ”gt” and ”user”.

Lastly, the model accept only features as vector and labels we simply combined the features into one vector

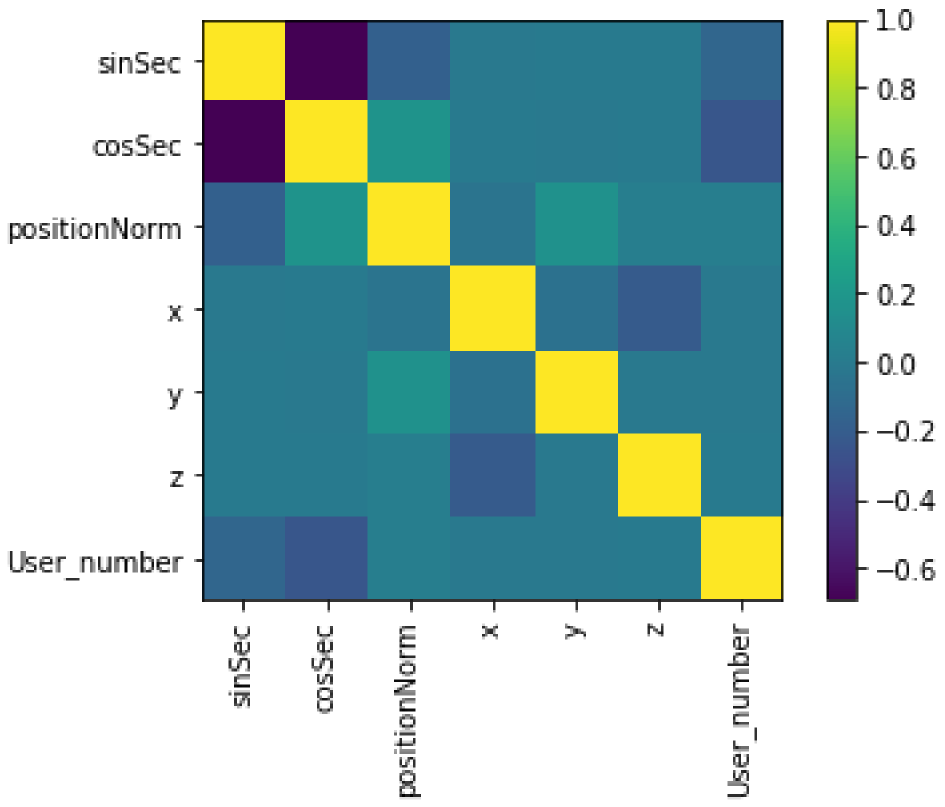
[ ]:
```python
from pyspark.sql.functions import when
position_label_transformer = Pipeline(stages=[
    VectorAssembler(inputCols=["x", "y", "z",], outputCol="position"),
    StringIndexer(inputCol = 'gt', outputCol = 'label'),
])

pday_transformer = Pipeline(stages=[
    VectorAssembler(inputCols=["Morning", "Noon", "Night"],
                    outputCol="pdayVec")
])
```

```
feature_transformer = Pipeline(stages=[
    VectorAssembler(inputCols=["sinSec", "cosSec", "positionNorm", "position",
                               "User_number"],
                    outputCol="features")
])

user_transformer = Pipeline(stages=[
    StringIndexer(inputCol='User', outputCol='User_number')
])

def process_data(df):

  # Time manipulation

  df = df.withColumn("Arrival_Time1", f.from_unixtime(f.col("Arrival_Time")/
↪1000))
  df = df.withColumn("Arrival_Date", f.split("Arrival_Time1", " ").getItem(0))
  df = df.withColumn("Arrival_Hour", f.split("Arrival_Time1", " ").getItem(1))


  df = df.withColumn("p_day", when((6 <= f.hour("Arrival_Hour")) & (f.
↪hour("Arrival_Hour") <= 12),"Morning")
                              .when((12 < f.hour("Arrival_Hour")) & (f.
↪hour("Arrival_Hour") <= 19),"Noon")
                              .otherwise("Night"))
  categories = ["Morning", "Noon", "Night"]

  exprs = [f.when(f.col("p_day") == category, 1).otherwise(0).alias(category) \
          for category in categories]

  df = df.select('*', *exprs)

  df = pday_transformer.fit(df).transform(df)
  df = df.drop(*categories)


  df = df.withColumn("sinSec", seconds_sin("Arrival_Time"))
  df = df.withColumn("sinSec", f.sin("sinSec"))

  df = df.withColumn("cosSec", seconds_cos("Arrival_Time"))
  df = df.withColumn("cosSec", f.cos("cosSec"))


  # Position manipulation
  model = position_label_transformer.fit(df)
  df = df.withColumn("positionNorm", norm(f.array("x", "y", "z")))
  df = model.transform(df)
```

```
    df = df.drop("x", "y", "z")

    # User manipulation
    df = user_transformer.fit(df).transform(df)

    # Final composition
    df = feature_transformer.fit(df).transform(df)
    df = df.select("features", "label")

    return df

data = process_data(df)

test, train = data.randomSplit([0.7, 0.3])
```

After we performed the transformations on our data and picked the features to use for our model, we want to make sure that there aren't any redundant features to save memory and computation time for our model

[ ]:
```
import matplotlib.pyplot as plt
from pyspark.ml.stat import Correlation
matrix = Correlation.corr(data, "features")
corr = matrix.collect()[0]["pearson({})".format("features")].values
corr = corr.reshape([7,7])

fig, ax = plt.subplots()

def heatmap2d(arr: np.ndarray):
    plt.imshow(arr, cmap='viridis')
    plt.colorbar()
    plt.show()

cols = ["sinSec", "cosSec", "positionNorm",
        "x", "y", "z","User_number"]

ax.set_xticks(list(range(0,len(cols))))
ax.set_xticklabels(cols)
plt.xticks(rotation=90)

ax.set_yticks(list(range(0,len(cols))))
ax.set_yticklabels(cols)

heatmap2d(corr)
```



We can see that the features aren't correlated, so we are ready to start working with a ML model.

3 Random Forest

```
[ ]: %%time
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

rf = RandomForestClassifier(featuresCol = 'features', labelCol = 'label',
                            maxDepth=10, numTrees=50)
rfModel = rf.fit(train)

# Evaluating our model

predictions = rfModel.transform(train)
evaluator = MulticlassClassificationEvaluator(labelCol="label",␣
 ↪predictionCol="prediction")
accuracy = evaluator.evaluate(predictions)
print("Training accuracy = %s" % (accuracy))
```

```
predictions = rfModel.transform(test)
evaluator = MulticlassClassificationEvaluator(labelCol="label",␣
 ↪predictionCol="prediction")
accuracy = evaluator.evaluate(predictions)
print("Testing ccuracy = %s" % (accuracy))
```

```
Training accuracy = 0.7329869116587655
Testing:
Testing ccuracy = 0.7275385787682436
CPU times: user 1.07 s, sys: 146 ms, total: 1.22 s
Wall time: 3min 35s
```

4 Cross Validation Random Forest

[ ]:
```
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
# Create an initial RandomForest model.
rf = RandomForestClassifier(labelCol="label", featuresCol="features")

# Evaluate model
rfevaluator = MulticlassClassificationEvaluator(predictionCol="prediction",
                                                labelCol="label",␣
 ↪metricName="accuracy")

# Create ParamGrid for Cross Validation
rfparamGrid = (ParamGridBuilder()
             #.addGrid(rf.maxDepth, [2, 5, 10, 20, 30])
               .addGrid(rf.maxDepth, [2, 5, 10])
             #.addGrid(rf.numTrees, [5, 20, 50, 100, 500])
               .addGrid(rf.numTrees, [5, 20, 50])
             .build())

# Create 5-fold CrossValidator
rfcv = CrossValidator(estimator = rf,
                      estimatorParamMaps = rfparamGrid,
                      evaluator = rfevaluator,
                      numFolds = 5)

# Run cross validations.
rfcvModel = rfcv.fit(train)
print(rfcvModel)

# Use test set here so we can measure the accuracy of our model on new data
```

```
rfpredictions = rfcvModel.transform(test)

# cvModel uses the best model found from the Cross Validation
# Evaluate best model
print('accuracy:', rfevaluator.evaluate(rfpredictions))
```

```
CrossValidatorModel_3ed515b251ca
accuracy: 0.7160608080728861
```

Here we get the parameters of the optimal model.

[ ]:
```
rfcvModel.getEstimatorParamMaps()[ np.argmax(rfcvModel.avgMetrics) ]
```

[ ]:
```
{Param(parent='RandomForestClassifier_bde47c0b5f36', name='maxDepth',
doc='Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node; depth 1
means 1 internal node + 2 leaf nodes.'): 10,
 Param(parent='RandomForestClassifier_bde47c0b5f36', name='numTrees',
doc='Number of trees to train (>= 1).'): 50}
```

[ ]:
```
!jupyter nbconvert --to html /content/part2.ipynb
```

```
[NbConvertApp] Converting notebook /content/part2.ipynb to html
[NbConvertApp] Writing 334522 bytes to /content/part2.html
```