Prediction Evaluation on Non-emergency Response Requests

Jayanth Sivakumar

March 18, 2019

1 Introduction

The paper on NYCER lists the future work on evaluating other models with GCRF model to predict the response times of the non-emergency requests. In the future work, the paper suggests some time-series models to start with and extend to apply ARMA to compare the results generated from the GCRF for response and demand prediction. The dataset has 19 million rows from 2010 to present. In the paper, the period from 2015 to 2018 was filtered. Initially, to overcome computational challenges, Apache Spark with standalone cluster was used to load the dataset with the support of databrick's *spark-csv* tool to infer the schema from the dataset. From there, the initial preprocessing and pipelining was created to get started with prediction models. With the help of python pandas and scikit-learn, the dataset was filtered and split for prediction.On various aspects the python kernal crashed due to a very large dataset. Spark context was created for the environment and the necessary libraries were loaded. The datset was loaded using spark-csv. Dask, a python-pandas like dataframe tool to load and process on very big dataset came in handy for necessary summaries on the dataset like median, row value counts and aggregation. Only the agency type for NYPD was taken into consideration to reciprocate the results from the paper and going further with other linear models.

```
sc.stop()
sc = SparkContext()
sqlContext = SQLContext(sc)
```

```
import pyspark
from pyspark.sql import HiveContext
from pyspark.sql.types import *
from pyspark.sql import Row
from pyspark import SparkContext
from pyspark.sql.types import StructType,
        StructField, StringType, DoubleType,TimestampType,IntegerType
from pyspark.sql import SQLContext
from pyspark.sql import DataFrame
from pyspark.sql.types import *
from pyspark.sql.functions import *
from datetime import *
from dateutil.parser import parse
from pyspark.sql.functions import year,dayofmonth,
          dayofweek, month,unix_timestamp,to_timestamp
from pyspark.sql import window, Window
from pyspark.sql.window import Window
import calendar
from time import strftime
from datetime import datetime
import pandas as pd
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf,col
import pyspark.sql.functions as func
import pyspark.sql.functions as f
from pyspark.sql import functions as F
```

```
spark = SparkSession \
 .builder \
 .master("local") \
 .appName("NYCER") \
 .config("spark.some.config.option") \
 .getOrCreate()
```

2 Preprocessing

The datset was loaded and necessary preprocessing was done in the dataset. As the below coding goes on, the preprocessing phase will be explained in detail. Initially, the dataset was loaded. *format* field takes the spark-csv and infer the schema from the dataset. The dataset is very large that it approximately took 10-15 minutes to load the dataset. In preprocessing stage, the data was cleared of any values that look suspicious like negative response times, null values and filtered for the year 2015-2018. Then the dataset for NYPD was filtered to create a prediction model. Loading the data everytime in the notebook became a time consuming process. To overcome this, persisting the data after the necessary stages will help load everytime to resume. Some presist methods were spark's persist memory storage and panda's pickle file type. It was fast and efficient. It is simple and is very fast to save the file back for reusability.

```
data = sqlContext.read.load('311_Service_Requests_from_2010_to_Present.csv'
      ,format='com.databricks.spark.csv', header='true', inferSchema='true'
                                        )
```

The schema for the dataset is displayed in the below chunk.

```
data.printSchema()


    root
     |-- Unique Key: integer (nullable = true)
     |-- Created Date: string (nullable = true)
     |-- Closed Date: string (nullable = true)
     |-- Agency: string (nullable = true)
     |-- Agency Name: string (nullable = true)
     |-- Complaint Type: string (nullable = true)
     |-- Descriptor: string (nullable = true)
     |-- Location Type: string (nullable = true)
     |-- Incident Zip: string (nullable = true)
     |-- Incident Address: string (nullable = true)
     |-- Street Name: string (nullable = true)
     |-- Cross Street 1: string (nullable = true)
     |-- Cross Street 2: string (nullable = true)
     |-- Intersection Street 1: string (nullable = true)
     |-- Intersection Street 2: string (nullable = true)
     |-- Address Type: string (nullable = true)
     |-- City: string (nullable = true)
     |-- Landmark: string (nullable = true)
     |-- Facility Type: string (nullable = true)
     |-- Status: string (nullable = true)
     |-- Due Date: string (nullable = true)
     |-- Resolution Description: string (nullable = true)
     |-- Resolution Action Updated Date: string (nullable = true)
     |-- Community Board: string (nullable = true)
     |-- BBL: string (nullable = true)
     |-- Borough: string (nullable = true)
     |-- X Coordinate (State Plane): long (nullable = true)
     |-- Y Coordinate (State Plane): string (nullable = true)
     |-- Open Data Channel Type: string (nullable = true)
     |-- Park Facility Name: string (nullable = true)
     |-- Park Borough: string (nullable = true)
     |-- Vehicle Type: string (nullable = true)
     |-- Taxi Company Borough: string (nullable = true)
     |-- Taxi Pick Up Location: string (nullable = true)
     |-- Bridge Highway Name: string (nullable = true)
     |-- Bridge Highway Direction: string (nullable = true)
     |-- Road Ramp: string (nullable = true)
     |-- Bridge Highway Segment: string (nullable = true)
     |-- Latitude: double (nullable = true)
     |-- Longitude: double (nullable = true)
     |-- Location: string (nullable = true)
```

Resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. Since this is a standalone cluster, it still takes time to process any operation for this big of a dataset.

```
data_rdd = data.rdd
#data_rdd.persist(pyspark.StorageLevel.MEMORY_ONLY)
#data_rdd.getStorageLevel()
#print(data_rdd.getStorageLevel())
```

For date type, the schema was not infered. To create a spark dataframe, the schema is created for the fields that will be used all through the session. Here, the dataframe in spark is created.

```
schema = StructType([StructField("created_date", StringType(), True),
                     StructField("closed_date", StringType(), True),
                     StructField("agency", StringType(), True),
                     StructField("complaint_type", StringType(), True),
                     StructField("location", StringType(), True)])
df = sqlContext.createDataFrame(data_rdd.map(lambda d:(d[1],d[2],d[3],d[5],
                                        d[25])), schema=schema)
```

spark_shape is not the spark implementation but is just a simple python function. This computes the dimension for the dataset. For this very reason, dask dataframe type came useful to quickly compute the dimension.

```
def spark_shape(self):
    return (self.count(),len(self.columns))
pyspark.sql.dataframe.DataFrame.shape = spark_shape
```

NA's are removed from the dataset. To predict the response time, the fields without closing date is not useful.

```
df = df.na.drop(subset=["closed_date"])
```

```
df.shape()

    (19710524, 5)
```

```
df.printSchema()

    root
     |-- created_date: string (nullable = true)
     |-- closed_date: string (nullable = true)
     |-- agency: string (nullable = true)
     |-- complaint_type: string (nullable = true)
     |-- location: string (nullable = true)
     |-- parsed_created_date: string (nullable = true)
     |-- parsed_closed_date: string (nullable = true)
     |-- response_time: integer (nullable = true)
```

From the string type, the date format is parsed and renamed to create a column under different name. It is useful to keep the original fields since the time to load the dataset takes longer time.

```
df = df.withColumn("parsed_created_date",f.from_unixtime(f.unix_timestamp("
                                      created_date",'MM/dd/yyyy hh:mm:ss aa
                                      '),'MM/dd/yyyy HH:mm:ss'))
df = df.withColumn("parsed_closed_date",f.from_unixtime(f.unix_timestamp("
                                      closed_date",'MM/dd/yyyy hh:mm:ss aa'
                                      ),'MM/dd/yyyy HH:mm:ss'))
```

From created_Date and closed_Date, the response time is computed and renamed under a different column.

```
timefmt='MM/dd/yyyy HH:mm:ss'
timeDiff = (F.unix_timestamp("parsed_closed_date",format=timefmt) - F.
                                      unix_timestamp("parsed_created_date",
                                      format=timefmt))/3600
df = df.withColumn("response_time",timeDiff.cast(DoubleType()))
```

```
df.shape()

    (18849791, 8)
```

Then the dataset is filtered based on agency type, complaint type and location. As it was mentioned in the paper that 32 complaint types except for *Large Bulky Item Collection* was filtered, those complaint types are removed from processing.

```
df = df.where(col('response_time')>=0)
df = df.where((col('agency') == 'NYPD') |
              (col('agency') == 'HPD') |
              (col('agency') == 'DOT') |
              (col('agency') == 'DSNY') |
              (col('agency') == 'DEP') |
              (col('agency') == 'DOB') |
              (col('agency') == 'DPR') |
              (col('agency') == 'DOHMH'))
df = df.filter(~df["complaint_type"].like("%bulky%"))
df = df.where((col('location') == 'BROOKLYN') |
              (col('location') == 'QUEENS') |
              (col('location') == 'MANHATTAN') |
              (col('location') == 'BRONX') |
              (col('location') == 'STATEN ISLAND'))
```

The first two rows of the dataframe is printed below.

```
df.head(2)

[Row(created_date='12/05/2014 12:00:00 AM', closed_date='12/13/2014 12:00:
                                      00 AM', agency='HPD', complaint_type=
                                      'HEAT/HOT WATER', location='QUEENS',
                                      parsed_created_date='12/05/2014 00:00
                                      :00', parsed_closed_date='12/13/2014
                                      00:00:00', response_time=192, year=
                                      None),
Row(created_date='12/05/2014 12:00:00 AM', closed_date='12/10/2014 12:00:00
                                        AM', agency='HPD', complaint_type='
                                      HEAT/HOT WATER', location='QUEENS',
                                      parsed_created_date='12/05/2014 00:00
                                      :00', parsed_closed_date='12/10/2014
                                      00:00:00',response_time=120, year=
                                      None)]
```

As mentioned earlier, only the agency type NYPD was taken into consideration.

```
df_agency = df.where(df.agency =="NYPD")
df_agency_rdd = df_agency.select(df.parsed_created_date,df.
                                       parsed_closed_date,df.agency,df.
                                       response_time).rdd
```

```
df_agency = df_agency_rdd.map(lambda d:(parse(d[0]),parse(d[1]),d[2],d[3]))
                                       .toDF()
```

After parsing the date and computing the response time, the filtered dataset for agency type NYPD looks like this.

```
df_agency.head(2)

[Row(_1=datetime.datetime(2014, 8, 20, 20, 3, 57), _2=datetime.datetime(
                                       2014, 8, 20, 22, 32, 1), _3='NYPD',
                                       _4=2),
Row(_1=datetime.datetime(2014, 8, 21, 0, 29, 37), _2=datetime.datetime(2014
                                       , 8, 21, 4, 32, 28), _3='NYPD', _4=4)
                                       ]
```

The column names for the above filtered dataset is renamed.

```
newcolnames = ['created_date','closed_date','agency','response_time']
for c,n in zip(df_agency.columns,newcolnames):
    df_agency=df_agency.withColumnRenamed(c,n)
```

Now filtering the dataset for the year 2015-2018 as mentioned in the paper.

```
df_agency = df_agency.filter(year(df_agency.created_date) >= 2015)
df_agency = df_agency.filter(year(df_agency.created_date) <= 2018).orderBy(
                                       "created_date")
```

The dataset looks like how it should be. Below spark dataframe has all the needed columns. The next step would be to have an array of date which is the first day of the week to filter the three weeks window for prediction.

```
df_agency = df_agency.withColumn('created_dt',df_agency['created_date'].
                                       cast('date'))
```

```
df_agency.head(3)

[Row(created_date=datetime.datetime(2015, 1, 1, 0, 0, 50), closed_date=
                                       datetime.datetime(2015, 1, 1, 2, 47,
                                       50), agency='NYPD', response_time=2,
                                       created_dt=datetime.date(2015, 1, 1))
                                       ,
Row(created_date=datetime.datetime(2015, 1, 1, 0, 1, 29), closed_date=
                                       datetime.datetime(2015, 1, 1, 2, 42,
                                       22), agency='NYPD', response_time=2,
                                       created_dt=datetime.date(2015, 1, 1))
                                       ,
Row(created_date=datetime.datetime(2015, 1, 1, 0, 1, 30), closed_date=
                                       datetime.datetime(2015, 1, 1, 0, 20,
                                       33), agency='NYPD', response_time=0,
                                       created_dt=datetime.date(2015, 1, 1))
                                       ]
```

DayofWeek column is added to filter easily the days of week which has the value 1. From there, filtering the three week period will be easy.

```
#1 = Sunday, 2 = Monday, ..., 7 = Saturday).
df_agency = df_agency.withColumn('DayofWeek', dayofweek(df_agency.
                                       created_dt))
```

```
df_agency

[Row(created_date=datetime.datetime(2015, 1, 1, 0, 0, 50), closed_date=
                                       datetime.datetime(2015, 1, 1, 2, 47,
                                       50), agency='NYPD', response_time=2.
                                       783333333333333, created_dt=datetime.
                                       date(2015, 1, 1), DayofWeek=5),
Row(created_date=datetime.datetime(2015, 1, 1, 0, 1, 29), closed_date=
                                       datetime.datetime(2015, 1, 1, 2, 42,
                                       22), agency='NYPD', response_time=2.
                                       681388888888889, created_dt=datetime.
                                       date(2015, 1, 1), DayofWeek=5),
Row(created_date=datetime.datetime(2015, 1, 1, 0, 1, 30), closed_date=
                                       datetime.datetime(2015, 1, 1, 0, 20,
                                       33), agency='NYPD', response_time=0.
                                       3175, created_dt=datetime.date(2015,
                                       1, 1), DayofWeek=5)]
```

The dataframe with the first day of week will be saved in a dataframe to later filter from the main dataframe. As the rows keep reducing, pandas dataframe is easier to work with.

```
start_date = df_agency.select(df_agency.created_dt).where(df_agency.
                                        DayofWeek == 1).orderBy(df_agency.
                                        created_dt).distinct()
```

```
start_date = start_date.rdd
```

```
start_date = start_date.toDF().toPandas()
```

```
date = start_date.head(1)
date
```

The prediction is about median response time. The response time is aggregated based on the created date. The ordering is also based on the created date. The user defined function for median is shown below.

```
def median(values_list):
    med = np.median(values_list)
    return float(med)
udf_median = func.udf(median, FloatType())


df_grouped = df_agency.groupby("created_dt").agg(udf_median(func.
                                        collect_list(col('response_time'))).
                                        alias('response_time')).orderBy("
                                        created_dt")
```

Here, after creating a dataset with the median response time, the dataset is converted to pandas type and pickle file is saved to reuse and resume to reduce the time complexity.

```
grouped_median = df_grouped.toPandas()
```

```
train.to_pickle('train.pkl')
#to load the file back, use train = pd.read_pickle('train.pkl')
test.to_pickle('test.pkl')
#to load the file back, use test = pd.read_pickle('test.pkl')
start_date.to_pickle('start_date.pkl')
grouped_median.to_pickle('grouped_median.pkl')
```

```
start_date = pd.read_pickle('start_date.pkl')
df_grouped = pd.read_pickle('grouped_median.pkl')
```

```
df_agency_filtered = df_agency.where((df_agency.created_dt >= date) & (
                                        df_agency.created_dt <= date +
                                        timedelta(days=21))).orderBy("
                                        created_dt")
```

```
df_agency_filtered.select(df_agency_filtered.created_dt).distinct().show()

    +----------+
    |created_dt|
    +----------+
    |2015-01-04|
    |2015-01-05|
    |2015-01-06|
    |2015-01-07|
    |2015-01-08|
    |2015-01-09|
    |2015-01-10|
    |2015-01-11|
    |2015-01-12|
    |2015-01-13|
    |2015-01-14|
    |2015-01-15|
    |2015-01-16|
    |2015-01-17|
    |2015-01-18|
    |2015-01-19|
    |2015-01-20|
    |2015-01-21|
    |2015-01-22|
    |2015-01-23|
    +----------+
    only showing top 20 rows
```

3 Prediction

After the preprocessing, the dataset here is split to training and test set. The training set consists of two weeks data and the test set consist of the following week's data after the week in training set. Then using simple linear regression and lasso, the median response time is predicted. Using the test labels, RMSE for the predicted value is computed. The average RMSE for the whole four years is printed after averaging over the whole result. The prediction pipeline works as follows.

1. The *start_date* dataset is iterated over to get the start date of the week
2. Using panda's time series tools, the training data is filtered for two weeks from that date from the main dataset
3. The test data is filtered for two weeks from the start date of the week
4. The training and test labels are split as *train_X,test_X,train_Y,test_Y*

5. The prediction model with *train_X,train_Y* is created and fit to the training dataset
6. The RMSE for each iteration is computed for every *DayofWeek* until the end
7. The mean RMSE is computed and printed out

```
import math
import numpy as np
from sklearn import datasets,linear_model
from sklearn.linear_model import Lasso
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pylab as plt
from sklearn import preprocessing
from math import sqrt
from statistics import mean
import math
```

```
from dask import dataframe as ddf
```

```
dd = ddf.from_pandas(df_grouped, npartitions=8)
df_grouped
```

```
decimals = 2
df_grouped['response_time'] = df_grouped['response_time'].round(2)
df_grouped
```

The last three start dates are removed since the dates won't have the three weeks window for each date. The prediction won't be as accurate as other dates.

```
start_date.drop(start_date.tail(3).index,inplace=True)
```

The prediction procedure listed above is being implemented below by iterating over the rows of start_date dataframe.

```
RMSE = []
for index,row in start_date.iterrows():
    date = row['created_dt']
    df_agency_filtered = df_grouped.loc[(df_grouped['created_dt'] >= date)
                                          & (df_grouped['created_dt'] <
                                          date + timedelta(days=21))].
                                          sort_values('created_dt')

    train = df_agency_filtered.loc[(df_agency_filtered['created_dt'] >=
                                          date) & (df_agency_filtered['
                                          created_dt'] <= date + timedelta(
                                          days=13))] .sort_values('
                                          created_dt')
    test = df_agency_filtered.loc[(df_agency_filtered['created_dt'] > date
                                           + timedelta(days=13)) & (
                                           df_agency_filtered['created_dt']
                                           < date + timedelta(days=21)) ].
                                           sort_values('created_dt')

    test_X = test.index
    test_Y = test['response_time']

    test_Y = test_Y.values.reshape(-1, 1)
    test_X = test_X.values.reshape(-1, 1)
    test_X = test_X.astype('datetime64[D]').astype(float)

    train_X = train.index
    train_Y = train['response_time']

    train_X = train_X.values.reshape(-1, 1)
    train_Y = train_Y.values.reshape(-1, 1)
    train_X = train_X.astype('datetime64[D]').astype(float)

    #simple linear regression for predicting median response time
    lr = LinearRegression()
    lr.fit(train_X,train_Y)
    pred = lr.predict(test_X)
    pred = pred.round(2)

    #lasso for predicting median response time
    #lasso = Lasso(alpha=0.1, copy_X=True,fit_intercept=True, max_iter=1)
    #lasso.fit(train_X,train_Y)
    #pred = lasso.predict(test_X)'''
    err = sqrt(mean_squared_error(test_Y, pred))
    RMSE.append(err)
print("RMSE:",np.round(mean(RMSE),2))

    RMSE: 0.32
```

The simple ARIMA model was implemented for the two week window and the *(p,d,q)* values with the grid tuning is *(0,0,0)*. With that, the RMSE value is 0.28 for the univariate time-series data. The table in summary gives better comparison for simple linear regression, lasso and ARIMA.

```
for index,row in start_date.iterrows():
        date = row['created_dt']
        df_agency_filtered = df_grouped.loc[(df_grouped['created_dt'] >=
                                              date) & (df_grouped['
                                              created_dt'] < date +
                                              timedelta(days=21))].
                                              sort_values('created_dt')
        train = df_agency_filtered.loc[(df_agency_filtered['created_dt'] >=
                                                       date) & (df_agency_filtered[
                                                      'created_dt'] <= date +
                                                      timedelta(days=13))].
                                                      sort_values('created_dt')
        test = df_agency_filtered.loc[(df_agency_filtered['created_dt'] >
                                                      date + timedelta(days=13)) &
                                                      (df_agency_filtered['
                                                      created_dt'] < date +
                                                      timedelta(days=21)) ].
                                                      sort_values('created_dt')
        train = train.set_index('created_dt')
        train = train.T.squeeze()

        test = test.set_index('created_dt')
        test = test.T.squeeze()
        history = [x for x in train]
        # make predictions
        predictions = list()
        for t in range(len(test)):
            model = ARIMA(history, order=(0,0,0))
            model_fit = model.fit(disp=0)
            yhat = model_fit.forecast()[0]
            predictions.append(yhat)
            history.append(test[t])
        # calculate out of sample error
        error = sqrt(mean_squared_error(test, predictions))
        RMSE.append(error)

print("RMSE:",np.round(mean(RMSE),2))

      RMSE: 0.28
```

4 Summary

Based on the comparison with simple linear regression and lasso, the performance is relatively close. Although, ARIMA performes better compared to other two models in terms of RMSE.

Table for univariate time-series analysis on models

SLR	Lasso	ARIMA
0.32	0.30	0.28

5 Future Direction

Although the dataset is very large, dask for dataframe in python is a very useful tool to load the dataset and increase the window on an hourly basis. Using the historical data from a few years, predicting the response time of the upcoming year on an hourly basis every day could help understand the data better and how well the model can be improved. This method can be achieved through facebook's prophet for time-series analysis and prediction. It can be one of the evaluation criteria to understand the performance of the GCRF model. Also, some neural network techniques can be inculcated to predict the median response time. As a whole, using gradient boosting, the importance of the variable can be implemented and checked to see if the response time increases in Brooklyn or Manhattan compared to Queens and Bronx without splitting the dataset. Based on this information, a whole different splitting and pipelining can be implemented to improve the GCRF prediction. These benchmarks in comparison with GCRF model will allow us to improve the prediction in a more efficient and robust manner with fewer data.