Report

Mohammed Khursheed Ali Khan | ID# 0662357 *April 12, 2020*

Motivation

The reason behind using the ACS PUMS US census data was that it is a data for housing and population of us citizens for the years 2013-2017, and includes information of persons in all the states, and the resulting data has a good set of attributes with a mix of both numerical, categorical and missing values. There by helps in good analysis of the data in a distributed in manner in spark. Also being a vast data of 14GB in size, with 100M records and 1000 columns, it truly defines the big Data we are talking about and really helps in learning the spark framework. Also being data hungry the domain of Machine Learning, It served as a good purpose of ML in distributed environment with Spark.

1 Problem Addressed

The problem I am addressing is broad in nature as I am trying to gain insights on the data trying to discover some patterns on the data based on demographics of the population. Once I have gained those insights, I am working on further trying to apply ML algorithms on the data for further predictive analysis.

As mentioned in this project I am trying to carry a predictive analysis on the Industry the people are working in based on their age, schooling attained, their field of degree and the sectors they are working in. Also not to forget the gender that is an a important attribute for determining the industry.

2 Approach Overview

Lets get started with the environment I have deployed a standalone Spark cluster on a *Azure VM* with *14GB RAM* and *4vcpu cores*. And as for the executors I have used local cluster of 4 cores such that the driver/executors are run as multiple threads across 4 cores locally.

As for the data, the API for data loading was not responding, so I resorted to transferring files from my local system to remote work environment via `SCP transfer`.

Once the initial environment was setup, and data was loaded, then came the data analysis part,

since I have used `pyspark` API for working, I accordingly have used the corresponding constructs, for EDA, which involved carrying out some analysis for understanding the trends in data and feature engineering. Note that I haven't used any plots as I started working with `python 2` and once I reached the end of my work only to realize that a lot of `pyspark` visualizations supported work on python 3. However, I have included some histograms that define the data trends.

Once that was done and feature engineering was done, a machine learning pipeline was created to fit a model on our data and achieve the above mentioned problem statement.

Further in the next section on how each component was done and results was achieved such as how EDA, what core concepts were used for feature engineering and what kind a pipeline was developed with algorithms being used.

3 Approach Explained

The first step I achieved was data loading as data frames, because those built on top of RDD provide much easier munging as the data is given and structure in dataframes which is justified by the fact that we have a a highly organized dataset structured in nature. Also the recent changes and deprecation of the `MLlib` package which focused on RDD to dataframe based ml package further motivated me to use dataframes for analysis and working.

Along with the data, I also had to load the data dictionary as some of the attributes were coded .

Once that was done the next step involved some data analysis, EDA, this involved getting some insights by data filtering on dataframes which are distributed and immutable in nature. So some analysis I did was regarding the Class of workers above and below age 50 and the industry of work among the genders. The analysis gave some interesting and realistic results, wherein the x industry was male dominant and y industry was female dominant. Also for the workers above age 50, they were a workers and below 50 where mostly b workers.

For EDA, as defined in previous section, one way I could develop visualizations such as correlation and scatter and histograms etc. would be that to use `toPandas` and convert the dataframe to pandas dataframe and carry on the pandas visualizations, but pandas doesn't woprk the same ways as `pyspark` dataframes work and so I didn't use that Idea as it defies the whole big Data definition.

That being said, the next stage of feature engineering was the crucial phase. First, I decided on the features to work on for developing the model. The feature selection included selecting features that could help in predicting the industry the persons are working in. The best attributes that could define this were, schooling attainment, field of degree they have their gender as we had seen earlier in EDA how gender effects the class of worker and industry the persons are working in, and mostly importantly the class of workers an individual is in an industry. These all were selected as features,

along with age. Attributes selected in a nutshell are as follows:

- SCHL - Schooling attained
- AGEP - Age
- FOD1P - Field of Degree first
- FOD2P - Field of Degree second
- COW - Class of Workers
- INDP - Industry working in

An important step I carried before I proceeded ahead was the data cleaning, this involved filtering the records we need and we feel are relevant to our analysis. One such filter was the individuals with age less than 17, as we need individuals who have worked in industry, and obviously anyone less than age of 17 hasn't worked without the loss of generality.

The next step was the feature engineering phase. Involved transforming the categorical features as one-hot vectors for machine learning model. The categorical features were first converted to category indices via `stringIndexer` and then using a `oneHotEstimator` they were transformed to one-hot encoding. Similarly labels were also which are `INDP` transformed into categorical indices using `stringIndexer`.

Once this is done, the next step involved combining these features into a feature vector for building a ML model. This is helped via the `VectorAssembler` that creates a vector for selected features.

This is then followed by the core ML `pipeline`. All the above transformations and estimations such as the feature engineering and one-hot encoding along with the estimation such as model fitting and predictions are designed as a ML pipeline with multiple stages in a pipeline that define it. These stages are either transformers or estimators The final stage in the pipeline is the model fitting phase, wherein I select a ML algorithm. I will be using and getting a model by training the data based on a 60-30m random split of training and test data.

Following is a break down of how the pipeline is defined for the above model we are working on.

1. Stage – string indexing and encoding Feature engineering – transformer
2. Stage – string indexing labels – transformer
3. Stage – model fitting - estimator
4. Stage - predictions - estimator

Once the above pipeline was completed we had the evaluation phase, for which I used m multiple metrics such as *F1*, and *area under the curve*. The model can be further improved based on further hyperparameter tuning, but getting a better accuracy wasn't the main purpose of this work in spark. It was leveraging the power of spark and understanding the architecture via hands-on.

The screenshots below of the DAG which defines how the tasks were executed by the executors gives a good understanding of the distributed and resilient spark design. Since we have 4 cores as can be seen the task are divided as threads locally among the 4 cores. These are then mapped as partitions further after a union is taken from the 4 executor tasks of the results and further stages in DAG involved similar to RDD operations such as `flatmap` and `groupBy` key etc.

While I did manage to get decent results, though not that appealing for a production use case, however it was a good starting with *ROC* and *F1* scores reaching 0.4 on a scale of 1. I used decision trees and random forest algorithms for model building and as expected got better accuracy with `RandomForest`.

Screenshots in the next section with brief descriptions of each of the above steps.

4 Spark webUI

This section briefs on the components (with screenshots) on the *Spark webUI* involved with the understanding of this project and some visualizations of the data.

As you can see in the screenshot below the `master` is set to `local[*]` instructing to use all the local cores of the system.

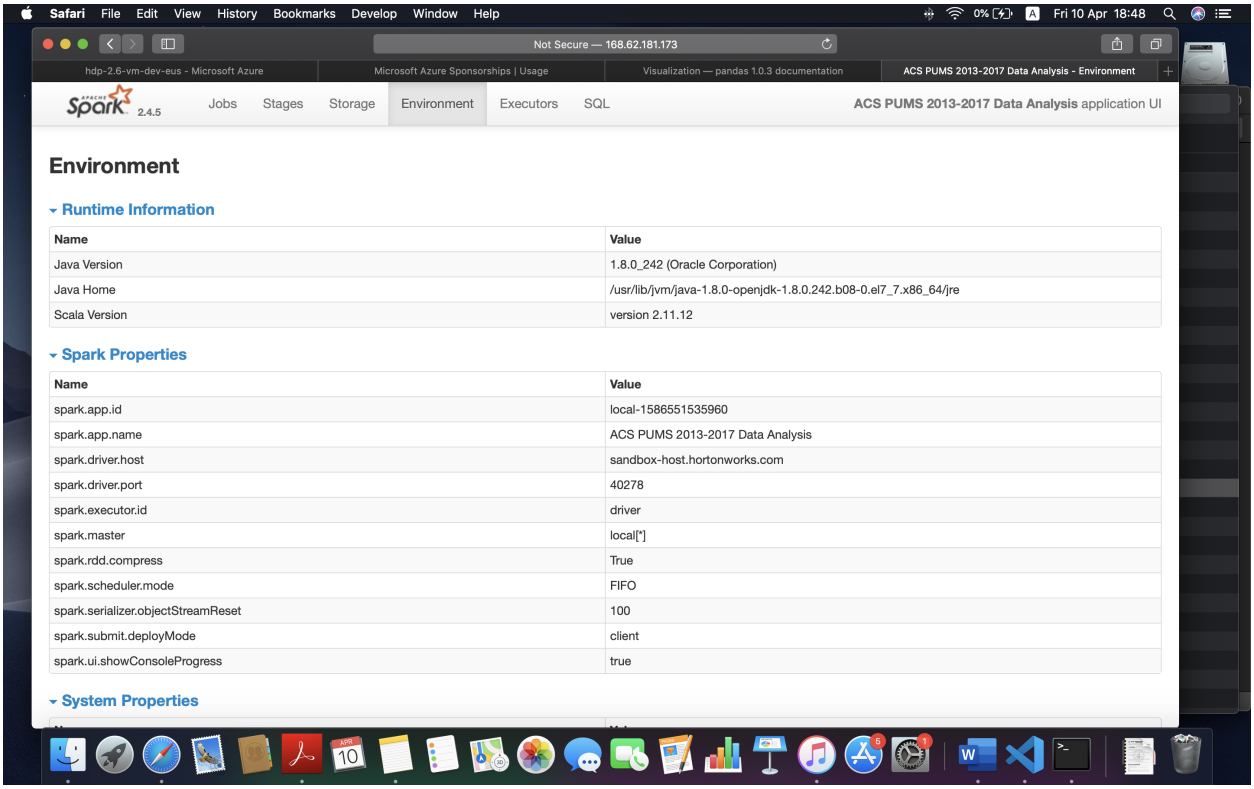

Figure 1: Runtime Environment configurations

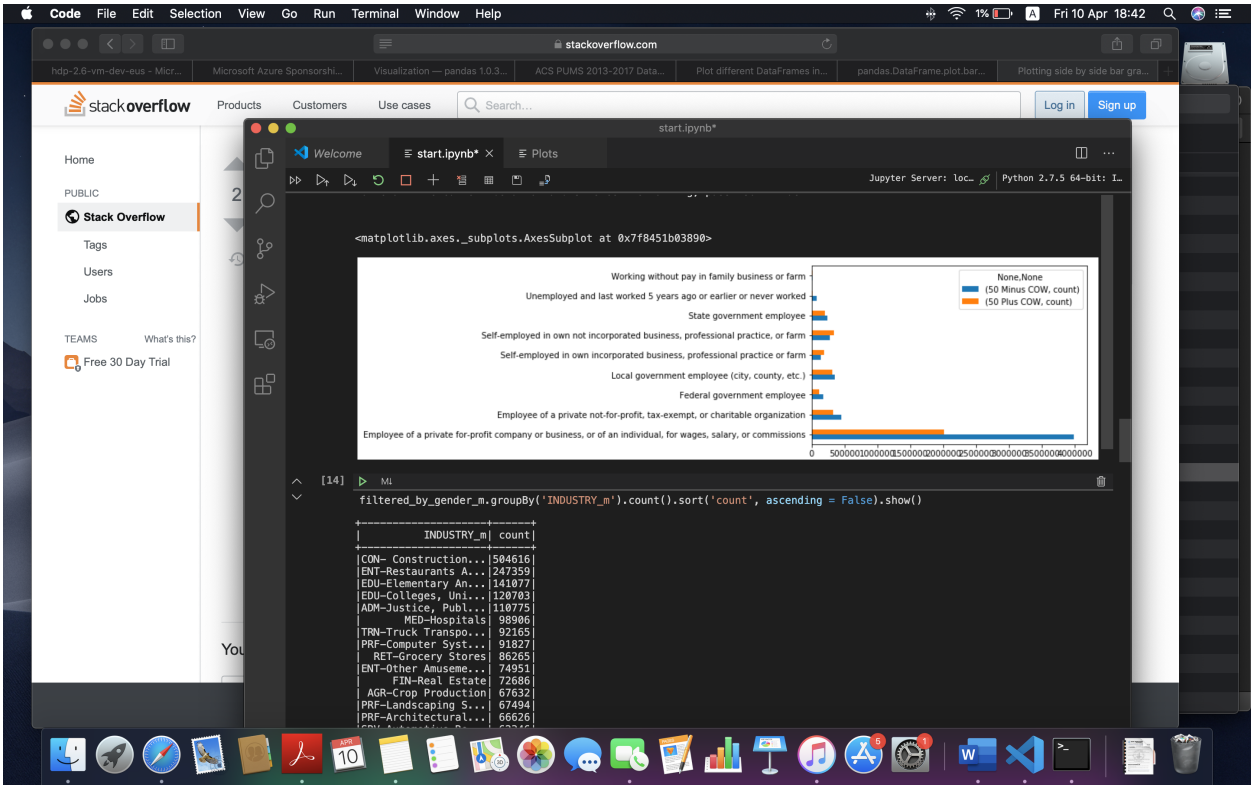

Figure 2: Class of workers for age $>$ & $<$ 50

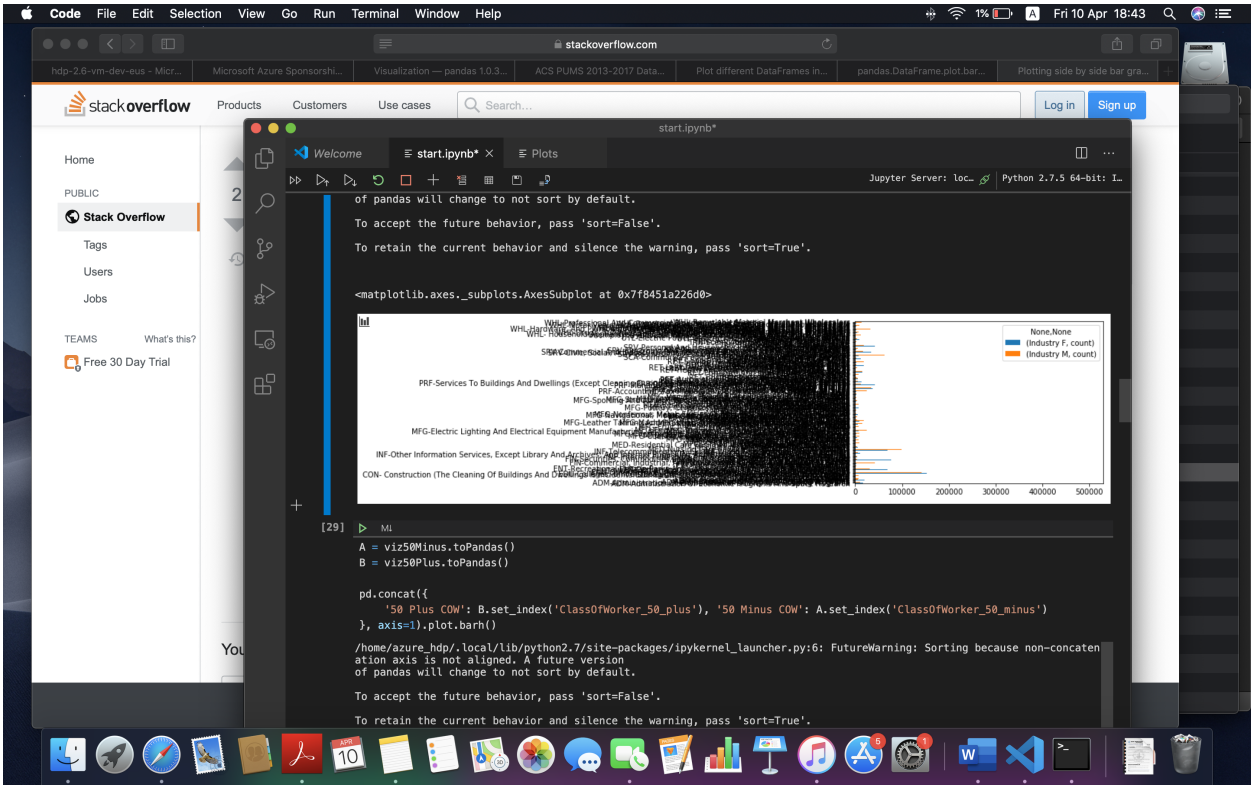

Figure 3: Industry workers working in

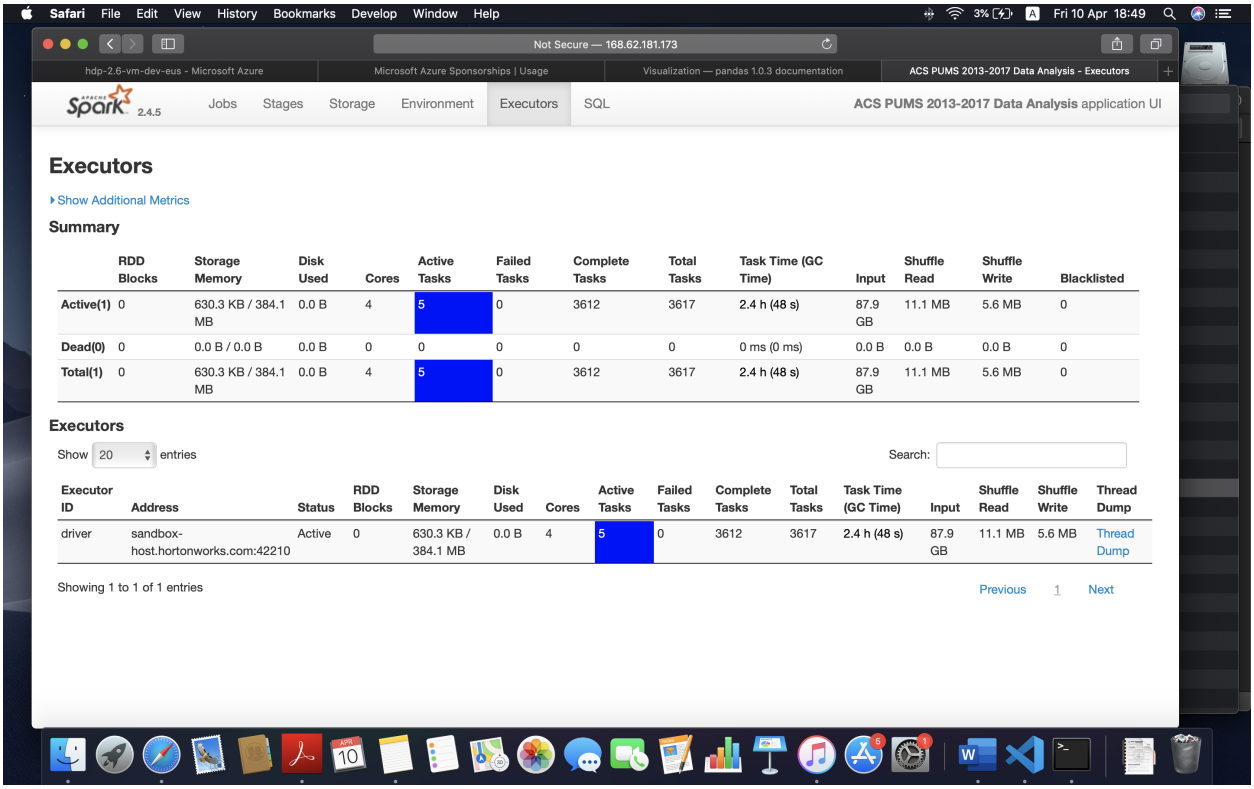

Figure 4: Executors tasks running on

The below figures 5, 6, 7, 8 describe the different timeline abut how tasks are executed one after the other. Note that in a lot of places you might see that there exists more than one time same tasks repeated, this is because we are using dataframes, which are essentially build over RDD. So any say `df.show()` is placed it is first converted/placed to RDD `first(21)` and further converted/placed as `take(21`

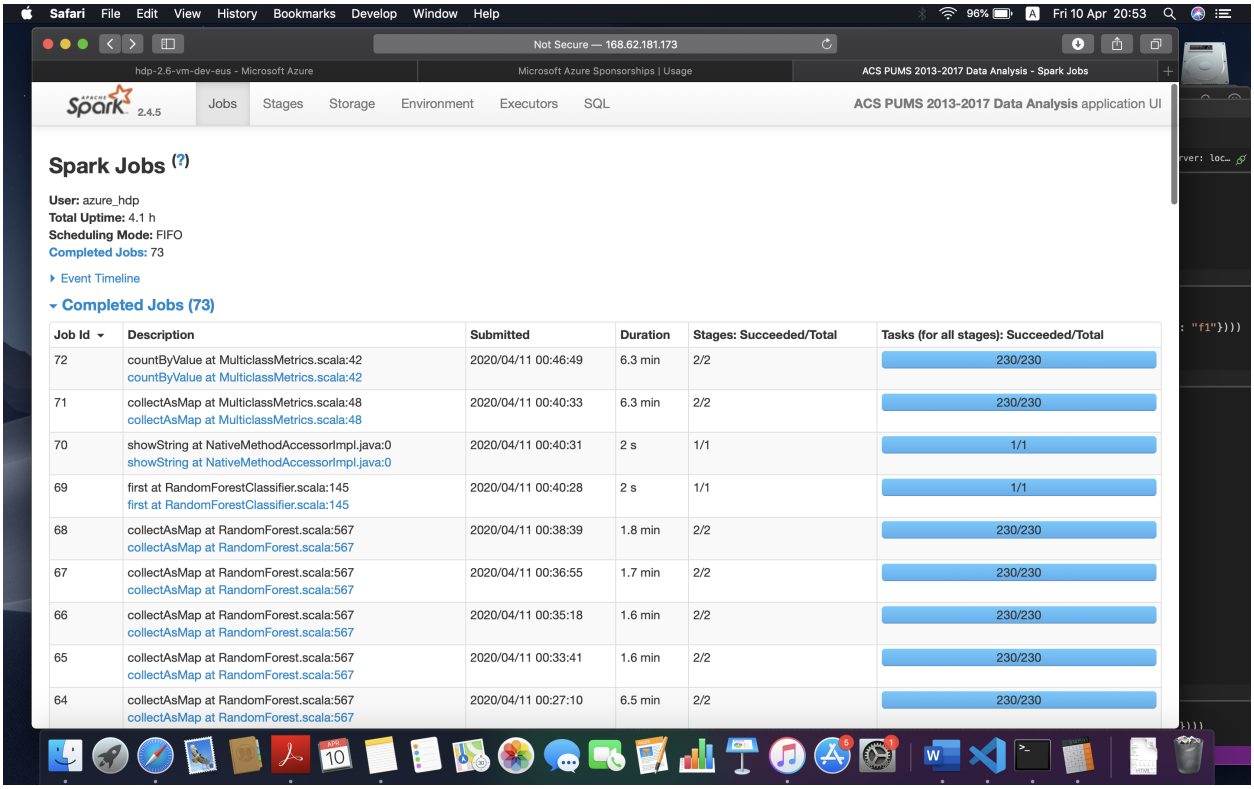

Figure 5: Jobs 1

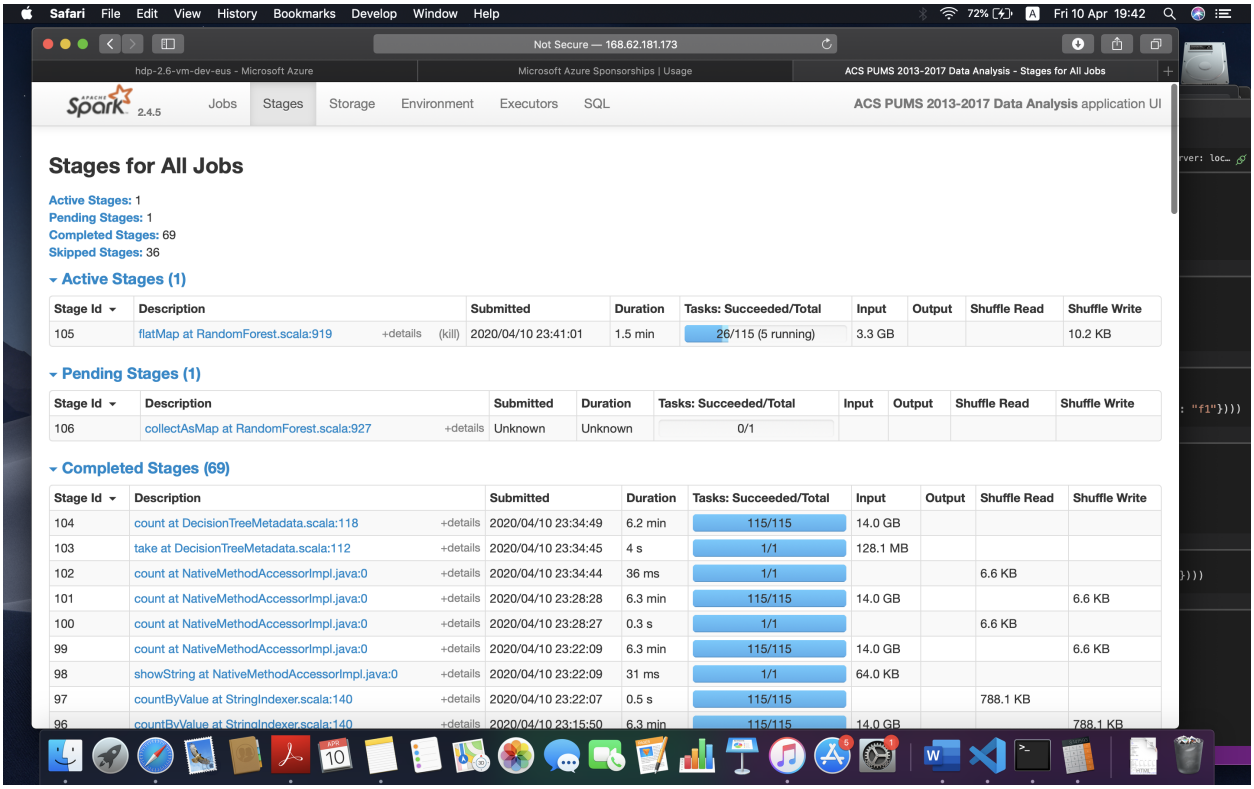

Figure 6: Jobs - ML

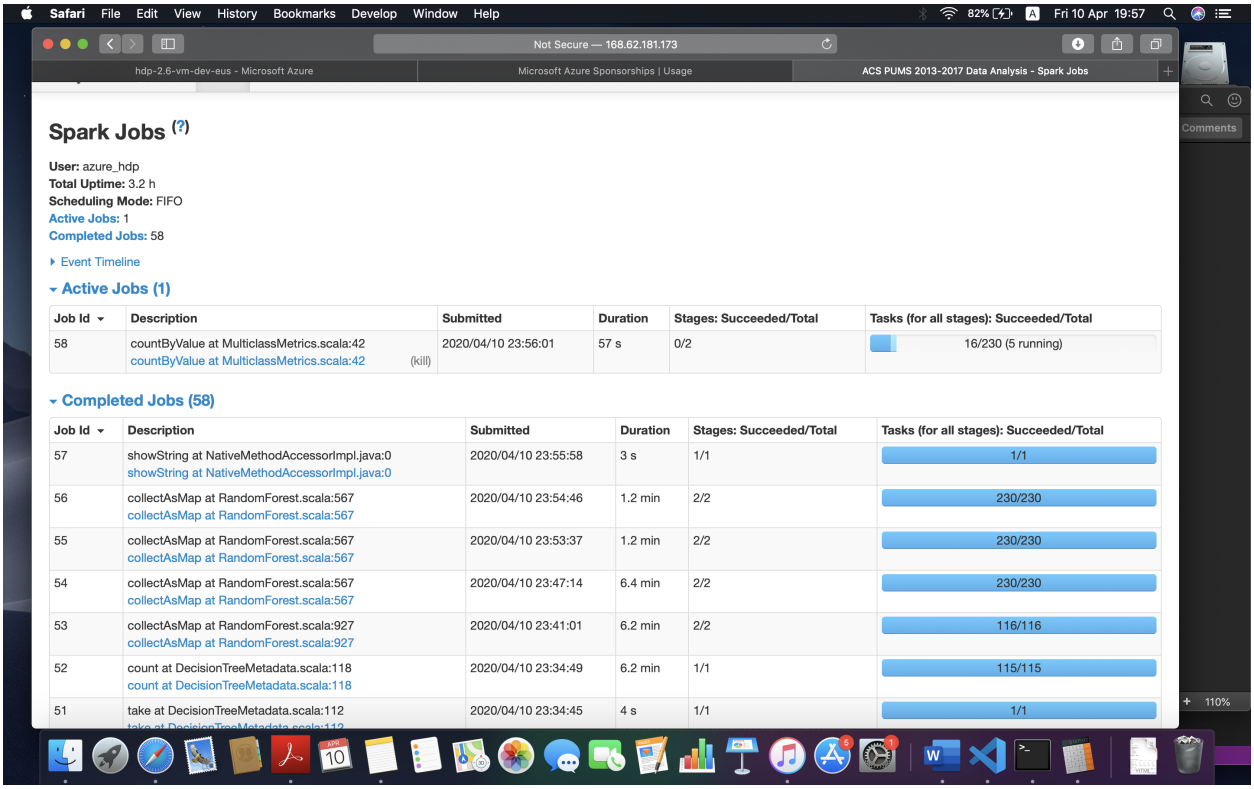

Figure 7: Jobs - metric

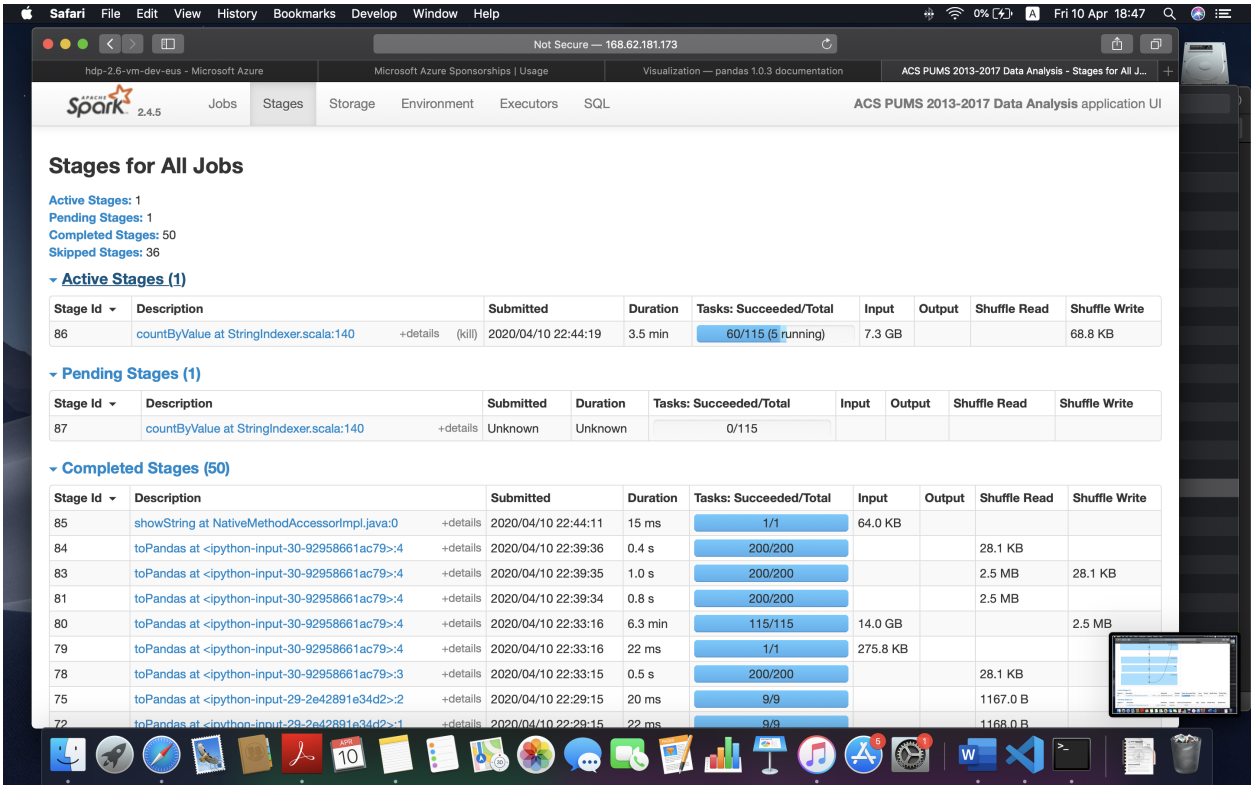

Figure 8: Jobs - stringIndexer

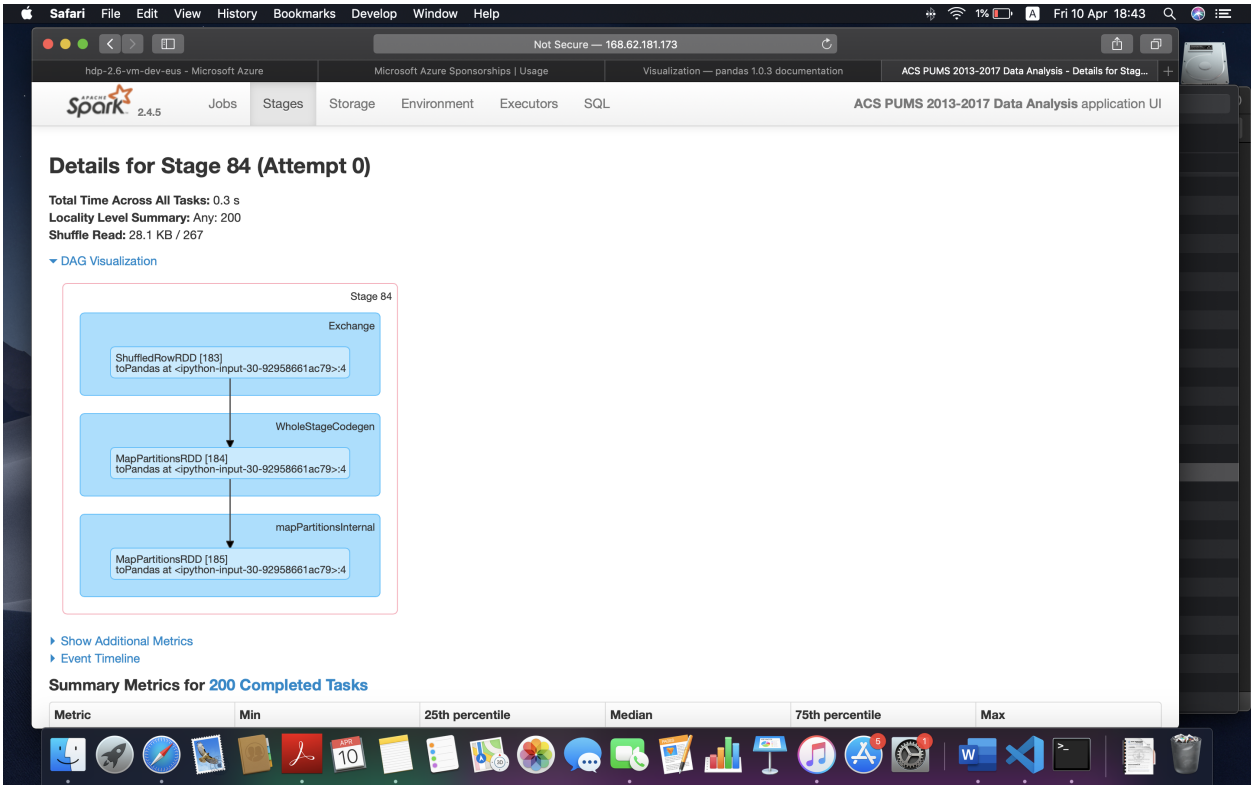

Figure 9: DAG - toPandas()

Also observe in the above figure 9, as to how the `toPandas()` DAG validates the above mentioned point, wherein the actions and transformations are performed on RDD after calling them internally.

An important point to mention about the spark design is it caches the tasks executed especially those that involved shuffling such as `reduceBy`, and as a result of this it fetches it from cache and skips that task as can be seen in below figure 10 where tasks are skipped.

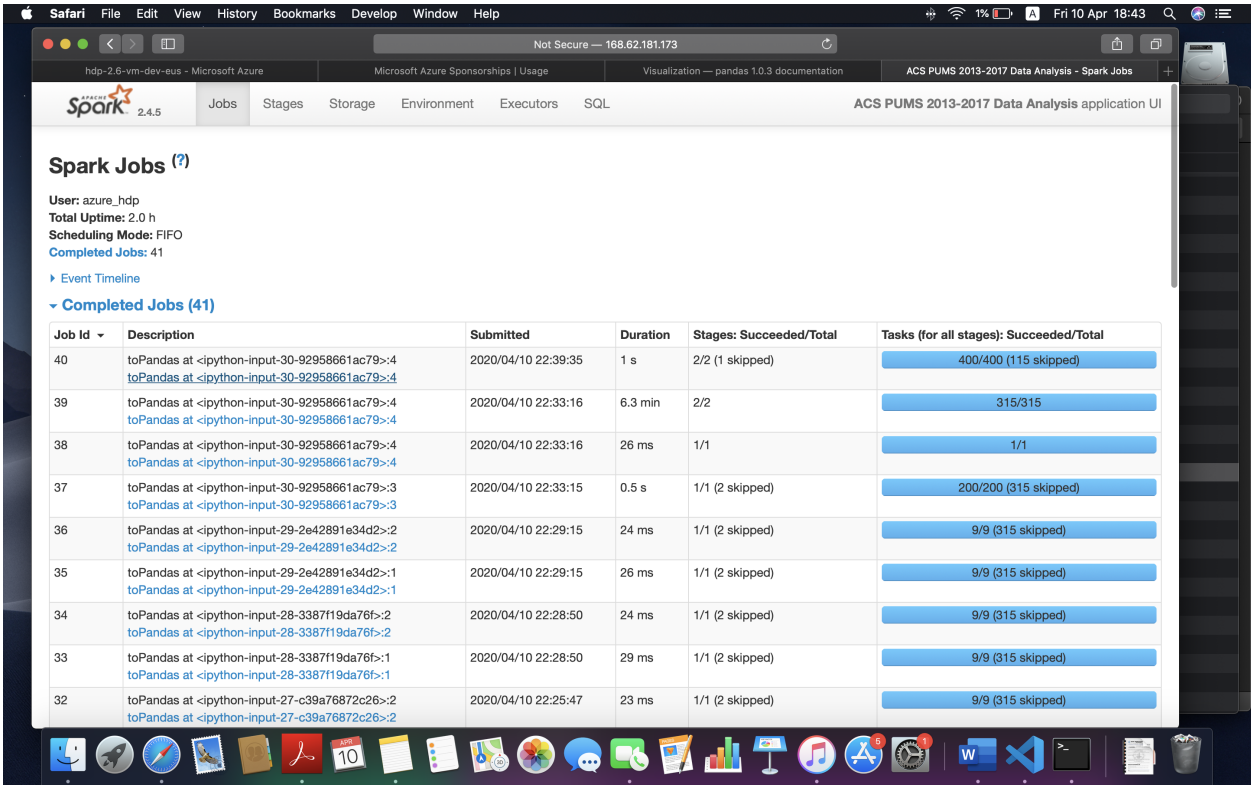

Figure 10: Jobs - Caching

In the below figure 11 the DAG for `groupBy()` is done with unoin of all tasks performed on 4 cores and then flatmapping and partitioning these tasks results and finally `countByValue` which defines the `groupBy()` functionality.

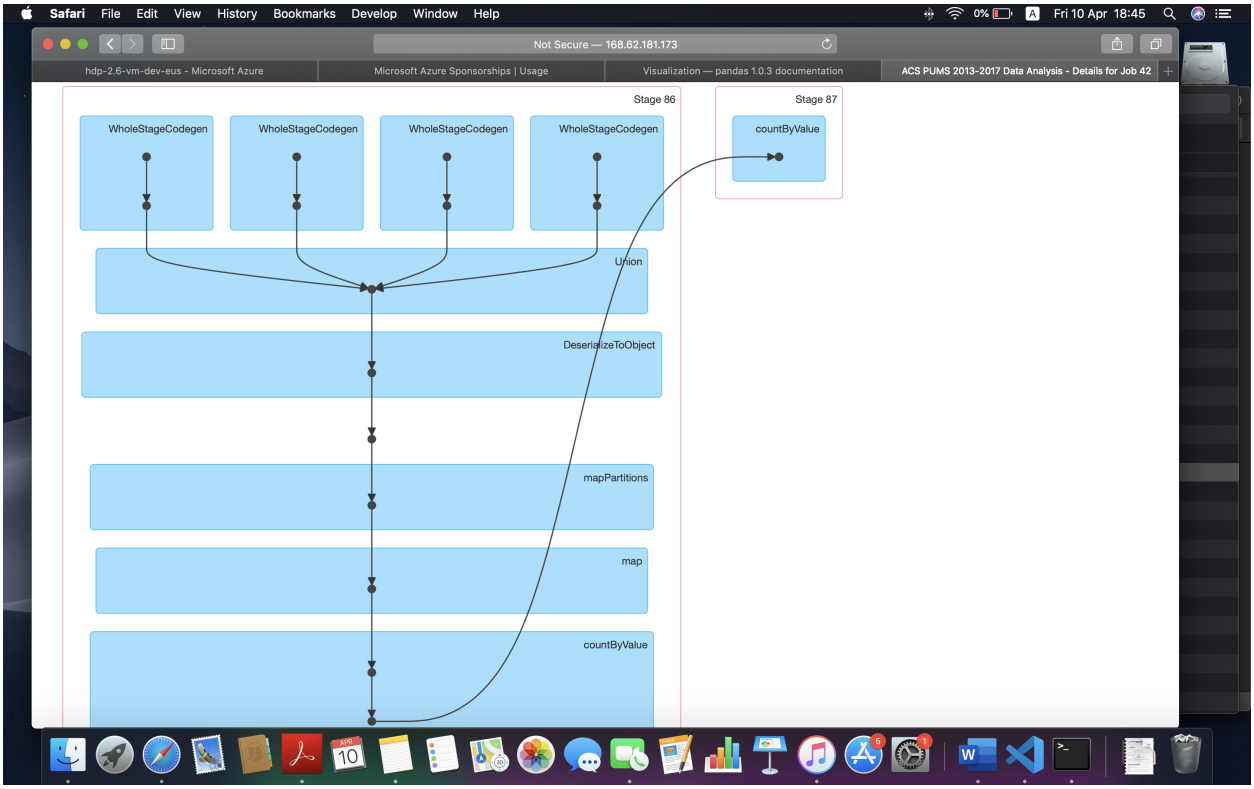

Figure 11: DAG - groupBy()

Similar to above, the below figure 12, once string indexing of categorical variables is done, we essentially have indices for each category which are then reduced by key `reduceByKey`, and essentially act as key-value pair.

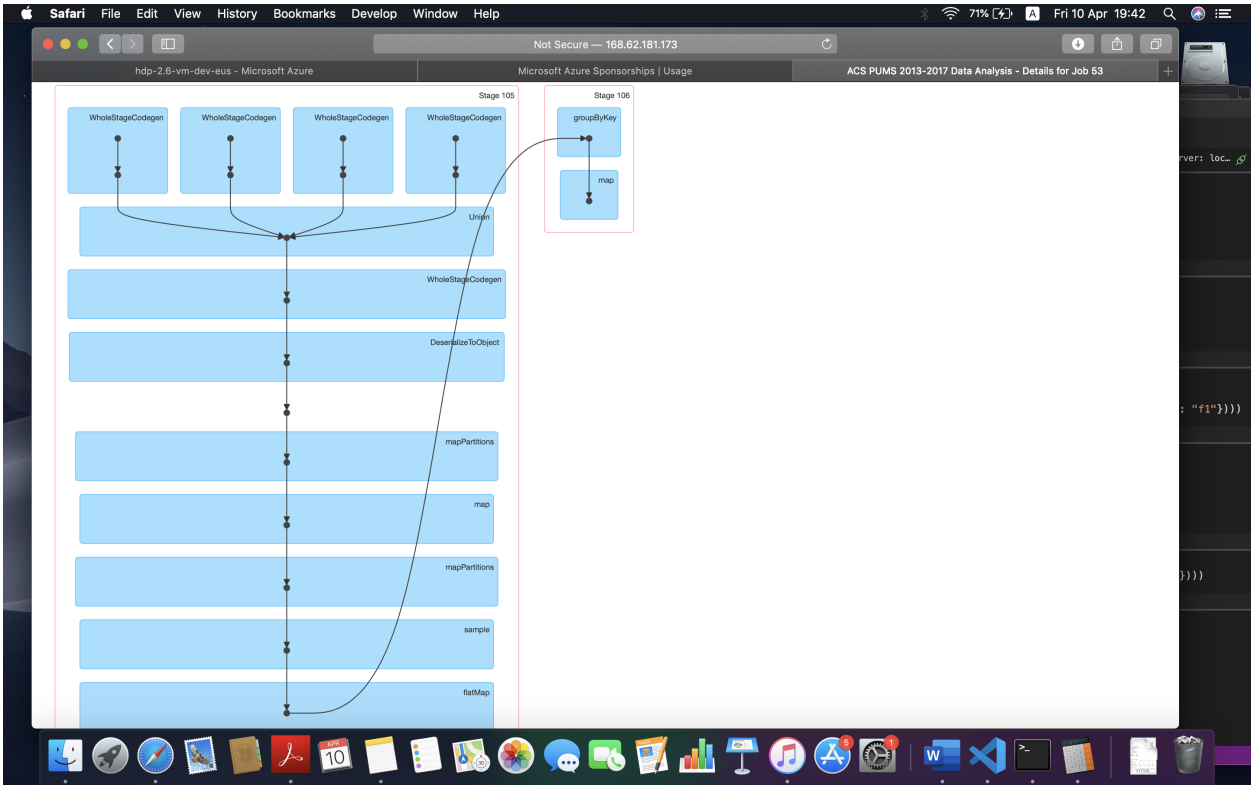

Figure 12: DAG - stringIndexer

The final figure 13 below describes the DAG for a ML algorithm task we used, involving multiple `map` and `flatMap` operations with `mapPartitions` carried out for the model building.

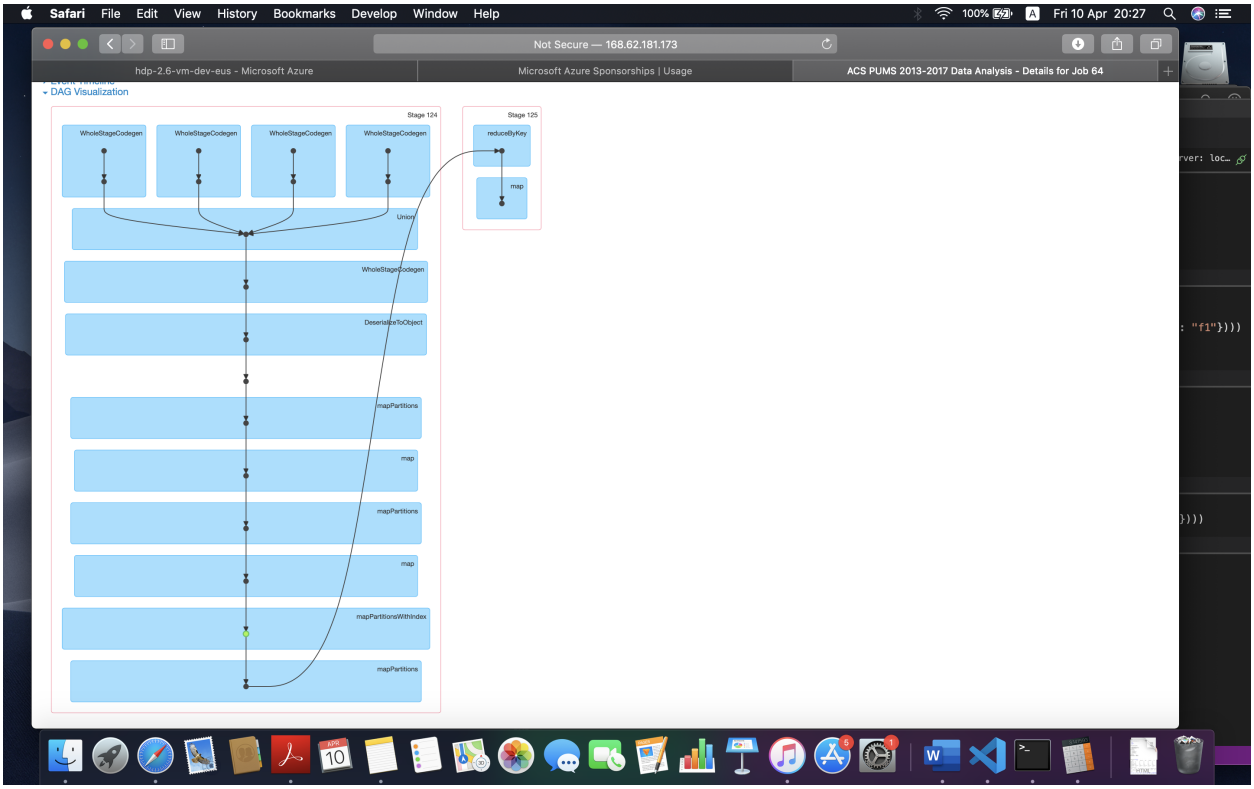

Figure 13: DAG - Final

Appendix

This section includes the utility commands used such as those for file/data transfers and other miscellaneous stuff.

`GitHub` repository for this project can be found here.

Data can be found here

- Installing Spark standalone

  ```
  wget   https ://www.apache.org/dyn/closer.lua/spark/spark-2.4.5/spark-2.4.5-bin-hadoop2.7.tgz

  tar xzvf spark-2.4.5-bin-hadoop2.7.tgz
  ```

- Data transfer from local system to remote VM

  ```
  scp -r <local_dir> <user>@<remote_host>:<remote_dir>
  ```

Below is the python implementation of the concepts addressed in above sections.

```
from IPython import get_ipython

# %%
from pyspark.sql import SparkSession
from pyspark import SparkContext
from pyspark.sql import functions as f
from pyspark.ml.feature import OneHotEncoderEstimator, StringIndexer,
    VectorAssembler
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier,
    RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from matplotlib import pyplot as plt
from pyspark_dist_explore import hist, Histogram, distplot
import seaborn as sns
import pandas as pd

get_ipython().magic(u'matplotlib inline')

# %%
spark = SparkSession.builder.master(
    "local[*]").appName("ACS PUMS 2013-2017 Data Analysis").getOrCreate()

# %%
# loading the dataset as DataFrame API, one at a time cleaning and removing it
    unitll next load
dataA = spark.read.csv('data/csv_pus/csv_pus/psam_pusa.csv',
                       header=True, inferSchema=True)
dataB = spark.read.csv('data/csv_pus/csv_pus/psam_pusb.csv',
                       header=True, inferSchema=True)
dataC = spark.read.csv('data/csv_pus/csv_pus/psam_pusc.csv',
                       header=True, inferSchema=True)
dataD = spark.read.csv('data/csv_pus/csv_pus/psam_pusd.csv',
                       header=True, inferSchema=True)

# %%
# selecting the columns we need - tranformation
cols = ['REGION', 'ST', 'AGEP', 'CIT', 'CITWP', 'COW', 'DDRS', 'DEAR', 'DEYE', '
    DREM', 'ENG', 'FER', 'GCL', 'HINS1', 'HINS2', 'HINS3', 'HINS4', 'HINS5', 'HINS6
    ',
        'HINS7', 'JWMNP', 'JWRIP', 'JWTR', 'LANX', 'MAR', 'RELP', 'SCH', 'SCHG', '
    SCHL', 'SEX', 'FOD1P', 'FOD2P', 'INDP', 'JWAP', 'JWDP', 'LANP', 'POVPIP', '
    POWSP']

dataA = dataA.select(*cols)
dataB = dataB.select(*cols)
dataC = dataC.select(*cols)
dataD = dataD.select(*cols)
```

```
42
43
44 # %%
45 # load data dictionary
46 # join with data Dictionary only during plotting as legend
47 dataDict = spark.read.csv(
48     'data/csv_pus/csv_pus/PUMS_Data_Dictionary_2013-2017.csv', header=True,
    inferSchema=True)
49
50 # %%
51 dataDict = dataDict.dropna(how='any')
52 dataDict = dataDict.select('RT', 'Record Type', '_c6')
53
54 # %%
55 dataDict = dataDict.filter(f.col('RT').isin(cols))
56
57 # %%
58 dataDict = dataDict.withColumnRenamed('RT', 'colName')
59 dataDict = dataDict.withColumnRenamed('Record Type', 'code')
60 dataDict = dataDict.withColumnRenamed('_c6', 'abbrev')
61 dataDict.show()
62
63 # %%
64 # get the session details
65 # spark.sparkContext.getConf().getAll()
66 dataDict.filter(dataDict.colName == 'SEX').show()
67
68 # %%
69 # get the spark webUI URL
70 spark.sparkContext.uiWebUrl
71
72 # %%
73 # merge all the dataframes into one dataframe
74 df = dataA.union(dataB)
75 df = df.union(dataC)
76 df = df.union(dataD)
77
78 # %%
79 # clearing the dataframes to free up the memory
80 dataA.unpersist(True)
81 dataB.unpersist(True)
82 dataC.unpersist(True)
83 dataD.unpersist(True)
84
85 # %%
86 # finding the insights on data,
87 # the vision etc difficulties demographics and mostly on high age groups
88 # Create some selections on this data
```

```
filtered_by_gender_m = df.filter(f.col('SEX') == '1').select('INDP').join(dataDict, [
    dataDict.colName == 'INDP', dataDict.code.cast('int') == df.INDP], 'inner').select(f.col('abbrev').alias('INDUSTRY_m'))
filtered_by_gender_f = df.filter(f.col('SEX') == '2').select('INDP').join(dataDict, [
    dataDict.colName == 'INDP', dataDict.code.cast('int') == df.INDP], 'inner').select(f.col('abbrev').alias('INDUSTRY_f'))

filtered_by_age_50_plus = df.filter(f.col('AGEP') > 50).select('COW').join(dataDict, [
    dataDict.colName == 'COW', dataDict.code.cast('int') == df.COW], 'inner').select(f.col('abbrev').alias('ClassOfWorker_50_plus'))
filtered_by_age_50_minus = df.filter(f.col('AGEP') <= 50).select('COW').join(dataDict, [
    dataDict.colName == 'COW', dataDict.code.cast('int') == df.COW], 'inner').select(f.col('abbrev').alias('ClassOfWorker_50_minus'))

# %%
# aggregating the data for easier and memory efficient pandas plotting
vizM = filtered_by_gender_m.groupBy(
    'INDUSTRY_m').count().sort('count', ascending=False)
vizF = filtered_by_gender_f.groupBy(
    'INDUSTRY_f').count().sort('count', ascending=False)

viz50Minus = filtered_by_age_50_minus.groupBy(
    'ClassOfWorker_50_minus').count().sort('count', ascending=False)
viz50Plus = filtered_by_age_50_plus.groupBy(
    'ClassOfWorker_50_plus').count().sort('count', ascending=False)

# %%
# panadas plotting visualizations
C = vizM.toPandas()
D = vizF.toPandas()

pd.concat({
    'Industry M': C.set_index('INDUSTRY_m'), 'Industry F': D.set_index('INDUSTRY_f')
}, axis=1).plot.barh()

# %%
A = viz50Minus.toPandas()
B = viz50Plus.toPandas()

pd.concat({
    '50 Plus COW': B.set_index('ClassOfWorker_50_plus'), '50 Minus COW': A.set_index('ClassOfWorker_50_minus')
}, axis=1).plot.barh()
```

```
127
128 # %%
129 filtered_by_gender_m.groupBy('INDUSTRY_m').count().sort(
130     'count', ascending=False).show()
131
132 # %%
133 filtered_by_gender_f.groupBy('INDUSTRY_f').count().sort(
134     'count', ascending=False).show()
135
136 # %%
137 filtered_by_age_50_minus.groupBy('ClassOfWorker_50_minus').count().sort(
138     'count', ascending=False).show()
139
140 # %%
141 filtered_by_age_50_plus.groupBy('ClassOfWorker_50_plus').count().sort(
142     'count', ascending=False).show()
143
144 # %%
145 # determing the inDustry of work using Machine Learning
146 #features = [AGEP, SCH, SCHG, SCHL, SEX, FOD1P, FOD2P, COW]
147 #target = INDP
148
149 # AGEP > 17 takes care of COW and Schooling
150 # not doing join with the dataDict as during one hot encoding the column names
      might have spaces and doesn't make sense, c an use join during plotting and
      graphing
151 dfX = df.filter((f.col('AGEP') > 17) & (f.col('INDP').isNotNull())).select(
152     'AGEP', 'SCHL', 'SEX', 'FOD1P', 'FOD2P', 'COW', 'INDP')
153 dfX = dfX.fillna({'FOD1P': 0, 'FOD2P': 0})
154
155 # %%
156 dfX.show()
157
158 # %%
159 # onehotencoding of cateforical variablesa for classification
160 categCol = ['SCHL', 'SEX', 'FOD1P', 'FOD2P', 'COW']
161 numericCol = ['AGEP']
162 stages = []
163
164 # StringIndexing and oneHotEncoding of categorical features
165 for cols in categCol:
166     stringIndexer = StringIndexer(inputCol=cols, outputCol=cols+'Index')
167     encoder = OneHotEncoderEstimator(
168         inputCols=[stringIndexer.getOutputCol()], outputCols=[cols + 'classVec'])
169     stages += [stringIndexer, encoder]
170
171 # String indexing of label
172 labelStringIndexer = StringIndexer(inputCol='INDP', outputCol='label')
```

```
stages += [labelStringIndexer]

assemblerInput = [c + 'classVec' for c in categCol] + numericCol
assembler = VectorAssembler(inputCols=assemblerInput, outputCol="features")
stages += [assembler]

# %%
# creating a ML pipeline
pipeline = Pipeline(stages=stages)
pipelineModel = pipeline.fit(dfX)
dfX = pipelineModel.transform(dfX)
dfX = dfX.select('label', 'features')

# %%
dfX.show()

# %%
# train test split
train, test = dfX.randomSplit([0.8, 0.2], seed=2020)
print("train count: " + str(train.count()))
print("test count: " + str(test.count()))

# %%
dctClf = DecisionTreeClassifier(
    featuresCol='features', labelCol='label', maxDepth=3)
dctModel = dctClf.fit(train)
predictions = dctModel.transform(test)
predictions.select('prediction', 'probability').show(10)

# %%
# peerformance metric, evaluation
evaluator = MulticlassClassificationEvaluator()
print("Area under Curve ROC: " +
      str(evaluator.evaluate(predictions, {evaluator.metricName: "f1"})))

# %%
# RandomForesrt classifier
rfClf = RandomForestClassifier(featuresCol='features', labelCol='label')
rfModel = rfClf.fit(train)
predictions = rfModel.transform(test)
predictions.select('prediction', 'probability').show(10)

# %%
evaluator = MulticlassClassificationEvaluator()
print("Accuracy: " + str(evaluator.evaluate(predictions,
                                            {evaluator.metricName: "accuracy"})))

# %%
```