National Technical University of Athens
Department of Electrical Engineering

Δ.Π.Μ.Σ. Επιστήμης Δεδομένων
και Μηχανικής Μάθησης

Big Data Management
A semester asssignment

Φίλιππος Μαυρεπής (03400098)
Ευάγγελος Τσόγκας (03400120)

July 23, 2021

1 Introduction

The purpose of this lab exercise was becoming familiar with tools such as Spark(PySpark), SparkML and Hadoop filesystem through a set of tasks. The first focuses on the creation and execution of pipelines leading to meaningful results and the second implements an end to end machine learning pipeline on text data. In this sense, we experimented with the RDD API provided by Spark as also as with the combination of Spark SQL / DataFrames API. Further on, we tested different file formats such as comma separated values (.csv) and parquet files to compare them with respect to their efficiency and performance. The data employed for the first part of this project are a subset of the famous 2015 - NYC taxi data, whilst the data for the second part are a subset of a consumer complaints dataset which can be found Complaints dataset.

2 Parquet File Format

Firstly and for us to work in a distributed environment it was necessary to load the data files into the HDFS. This was done with the set of commands below:

```
# Create a new directory in the HDFS
hadoop fs -mkdir /input
# Move files from /home to hdfs /input
hadoop fs -put data/yellow_tripdata_1m.csv /input
```

Afterwards, we needed to transform the csv files to parquet. Parquet is a data file format provided by Apache Spark to increase efficiency and performance. It uses record shredding and assembly algorithm which differs from simple record flattening. Parquet is optimized to work with complex data in bulk and features different ways for efficient data compression and encoding types. This approach is best especially for those queries that need to read certain columns from a large table as parquet minimizes the need for IO due to the fact that it can read only the needed columns.
An indicative representation of the transformation-storage process is presented below:

```
# Read data
tripvendors = spark.read.option("header","false"
    ). \
option("inferSchema","true"). \
csv("hdfs://master:9000/input/
    yellow_tripvendors_1m.csv")

# Rename columns
tripvendors = tripvendors.withColumnRenamed("_c0
    ","ID") \
.withColumnRenamed("_c1","Vendor")

# Store to HDFS as parquet
```

```
tripvendors.write. \
parquet("hdfs://master:9000/input/
    yellow_tripvendors_1m.parquet")
```

The process of converting the the csv file to parquet took 151 seconds to complete.

3 Data Analysis

A thorough data analysis took place to identify possible misbehaviour and erroneous values, a quite common phenomenon while working with GPS data. The only suggested preprocessing step was to remove records with Latitude and Longitude values of **0**. Building on that idea and since we already know that the data corresponds to New York routes we decided to *create a bounding box* and dispose all other records. Later on, it occurred to us that another *possible mistake would be that the starting and ending position of the vehicle is the same* so we removed those records as well. Finally, another approach to pruning some records was *limiting the top speed* for taxis since we observed some impossible speed values in some cases. To accomplish that we found the haversine distance of the route and divided by its duration.
In summary we:

I. Applied NYC Bounding Box

II. Deleted records with no vehicle movement

III. Deleted records with extraordinary speed values.

By applying the aforementioned steps we managed to get rid of approximately *350 thousand* problematic records. It is rather obvious that applying different thresholds to the permitted speed values results in different sets of data. That is the reason why we experimented with those thresholds to keep most of the records without having problems in our queries.

4 Analytical Queries

Q_1: What is the average value of the longitude and latitude of boarding per hour of its start route? Sort the result based on the start time in ascending order.
Q_2: For each vendor, considering the distance to be Haversine distance, find the maximum distance of a route as well as its duration.

4.1 RDD

We will begin with the RDD API. We defined some helper functions (udf) in order to make the code more readable and maintainable.
Due RDDs being able to process one line at a time and since the input is a

string we need to split the columns based on ‘,’ as well as cast strings such as latitudes and longitudes to floats.
Also as we need to do the average based only on the start time of each route we need to define a function that extracts the hour part from the timestamp.
Last but not least we need to implement a function which will be used in a filter call for the removal of records outside the NYC bounding box and records where the starting posistion is identical to the ending position. All aforementioned functions can be found in the okeanos VM in the corresponding files and are omitted for the sake of readability.
Now the requested query can be broken down to a series of map, filter and reduce calls as seen below.

```
q1 = data. \
    map(Split Columns). \
    filter(Preprocess records). \
    map(Hour, (Lat,Lon,1)). \
    reduceByKey(Calculate sum of Lat,Lon per
    hour). \
    sortByKey(). \
    mapValues(Calculate average)
```

The time taken for RDD query 1 is 45 seconds and the results are presented below:

```
('00', (40.74352462039714, -73.9755281254266))
('01', (40.74155155672377, -73.97836875291003))
('02', (40.74112335355328, -73.98113822693551))
('03', (40.74156763219687, -73.98207117798593))
('04', (40.7449419280885, -73.97787033470128))
('05', (40.74822795121568, -73.96818277110464))
('06', (40.7511800022321, -73.96997305271658))
('07', (40.75415805511449, -73.97128179377935))
('08', (40.75421535610294, -73.9735225193056))
('09', (40.754110748081075, -73.97482514478928))
('10', (40.75457773649443, -73.97371626426579))
('11', (40.75455059649643, -73.97425922095346))
('12', (40.75430944248196, -73.97465170323727))
('13', (40.753735076367185, -73.97438301940515))
('14', (40.75343831711661, -73.97319851659698))
('15', (40.75336360638272, -73.97145951796949))
('16', (40.75297000087204, -73.96992380653482))
('17', (40.75333226074053, -73.97155232805113))
('18', (40.752786758877576, -73.9739598179012))
('19', (40.75143526278723, -73.97528093712585))
('20', (40.74969013538524, -73.97534866001912))
('21', (40.74880888384561, -73.97491833405356))
('22', (40.747713170851775, -73.97504014679001))
('23', (40.74577929909829, -73.97407217204909))
```

With regards to the second query we needed to implement some new udfs and edit accordingly some others. The most crucial function for this query was the calculation of the *haversine* distance.

Having a distance metric as well as the starting and ending time of each route allowed us to create a function to calculate an approximation of the vehicle speed. As already mentioned in section 3 this was done to remove some erroneous records and we considered a speed of 200 km/h to be a reasonable cutoff point. The question at hand requires information from both the vendors and the routes tables which means that a join of some sort is necessary.
The table containing the routes data needs to be processed with a series of map, filter operations as presented below:

```
p1 = routes. \
    map(Split Columns). \
    filter(Preprocess records). \
    map(ID, (duration,distance)). \
    filter(Speed_filter)
```

On the other hand the table with vendors' information need minimal to none preprocessing which is shown below:

```
p2 = vendors. \
    map(Split columns). \
    map(ID,vendor)
```

For the completion of this query the selected join is an inner join on the only shared column, namely the ID. Therefore now we have all the information needed, but we need to sort according to the distance for each vendor.
To this end we devised a function accepting two tuples of (distance, duration), compares them based on distance and returns the tuple containing the maximum value. This function was implemented in this way since we found that the 'reduceByKey' call can accept custom functions for reducing and none of the existing ones matched our purpose to the best of our knowledge.

```
joined = p1.join(p2). \
    map(Vendor_ID, (Distance, Duration)). \
    reduceByKey(Sort by distance)
```

Collecting the results we get tuples of the form:

$$(Vendor\_ID, (Distance(km), Duration(h)))$$

```
('2', (166.408, 2.277))
('1', (132.563, 2.324))
```

Regarding the execution time of this query it was around 455 seconds.

4.2 Spark SQL

About the Spark SQL implementation regardless of the type of file that we will be reading which in this case will be either a csv or a parquet file the process is as follows.

I. Read the data into dataframes.

II. Register temporary tables.

III. Construct necessary query.

IV. Execute query and gather results.

For Q_1 we create the following sql query with A, B, C, D, E, F being conditions corresponding to bounding box and the difference of starting and ending positions :

```
sqlString = "SELECT hour(Start) as HourOfDay,
                    AVG(S_Lat) as Latitude,
                    AVG(S_Lon) as Longitude FROM ROUTES " + \
                    "WHERE (A AND B AND C AND D AND E AND F" + \
                    "GROUP BY HourOfDay " + \
                    "ORDER BY HourOfDay ASC"
```

For Q_2 we define the *haversine* distance as a udf and then assign a new column to our routes table.

```python
haversine = udf(haversine_dist, DoubleType())
routes = routes.withColumn("Distance", haversine
    (phi1,phi2,l1,l2))
```

Afterwards, we need to create a joint table on both routes and vendors in a similar fashion as in the RDD case. Then the query we execute for Q_2 is:

```
sqlString = "SELECT Vendor, Di, T FROM " + \
                "(SELECT Vendor, Time as T, distance as Di, " + \
                "RANK() OVER
                    (PARTITION BY Vendor ORDER BY Di DESC)
                    as Rank FROM JOINED " + \
                    "WHERE (A AND B AND C AND D AND E AND F)" + \
                    "AND Di/T < 200
            WHERE Rank=1"
```

In the query above a lot of simplifications took place in order to present it in an orderly manner. First and foremost Time is defined as:

$$(\text{unix\_timestamp}(End) - \text{unix\_timestamp}(Start))/3600$$

Also A, B, C, D, E, F are the same as in Q_1 above. Of course as the same preprocessing has been implemented in both the csv and parquet we expect (and get) the same results and that is why we include them only once below.

Q_1 Results:

```
+---------+------------------+------------------+
|HourOfDay|          Latitude|         Longitude|
+---------+------------------+------------------+
|        0| 40.74352391432558|-73.97542119644494|
|        1| 40.74144873105477|-73.98015550551409|
|        2| 40.74096648053428|-73.98106246096083|
|        3| 40.74170628140122|-73.98179513007679|
|        4| 40.74445128274248|-73.97842361879847|
|        5| 40.74785017767986|-73.97273846432589|
|        6| 40.75116285983248|-73.96963386968942|
|        7| 40.75413501865128|-73.97124155969401|
|        8|40.754149591596814| -73.9734171341711|
|        9| 40.75406090873702|-73.97479352068423|
|       10| 40.75453335756495|-73.97366927085308|
|       11| 40.75432492996547|-73.97430727880621|
|       12|40.754283962218224| -73.9746343408384|
|       13| 40.75370263890706|-73.97432216233956|
|       14|40.753469749883244|-73.97310395038653|
|       15|40.753364065771095|-73.97143887498935|
|       16|40.752952755391675|-73.96993496643499|
|       17| 40.75330164005164|-73.97154585281723|
|       18|40.752760254901176|-73.97387697738458|
|       19|40.751279338230944| -73.9751888472364|
|       20|40.749653423341215| -73.9760667030382|
|       21|40.748726699352076|-73.97491147877938|
|       22| 40.74776911907728|-73.97503320838544|
|       23| 40.74569797772432|-73.97406554954225|
+---------+------------------+------------------+
```

Q_2 Pesults:

```
+------+-----------------+------------------+
|Vendor|         distance|              Time|
+------+-----------------+------------------+
|     1|132.563          |             2.324|
|     2|166.408          |             2.277|
+------+-----------------+------------------+
```

It is expected that the RDD implementation of the queries will be slower in execution time since it does not use the Catalyst Optimizer accessible through the Spark SQL and DataFrame API. Theoretically though, one can implement the same optimizations on the RDD and achieve comparable performances.

4.2.1 Spark SQL CSV

The time taken for the execution of the first query using csv input is approximately 44 seconds while for the second one is around 236 seconds. Even though we only captured the time needed for the queries there is also a significant difference in parsing time between csv and parquet files. Since we are using ‘infer_schema=True’ we are asking from Spark to try figuring out the types of each column of the csv. This can be a very time-consuming process so it is common to create the schema by hand and passing it to the read() call of Spark.

4.2.2 Spark SQL Parquet

On the other hand the execution time for the first query using parquet is around 12 seconds while for the second one is roughly 135 seconds. A complete comparison of execution times can be seen in Fig.1.

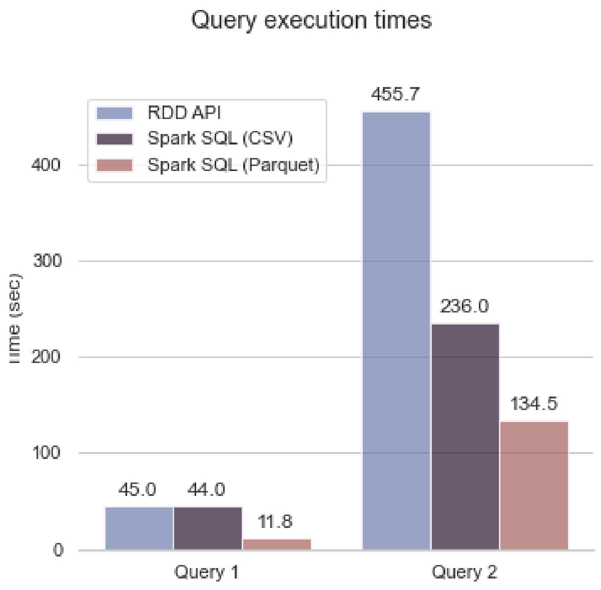

Figure 1: Barplot comparison of execution times for Q_1, Q_2 coloured by different implementations.

4.3 Join optimisation

In this step we are comparing both the performance and the physical execution plan produced by Spark when its join optimisation feature is enabled versus when it is disabled. The property we changed is ‘spark. sql. autoBroadcastJoinThreshold’ as it is responsible for configuring the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. By setting this property’s value to 0 or -1 we can effectively force Spark to never use BroadcastHashJoin and always use SortMergeJoin.
The full query execution pipeline from SQL through Dataset to RDD is seen in Fig. 2.

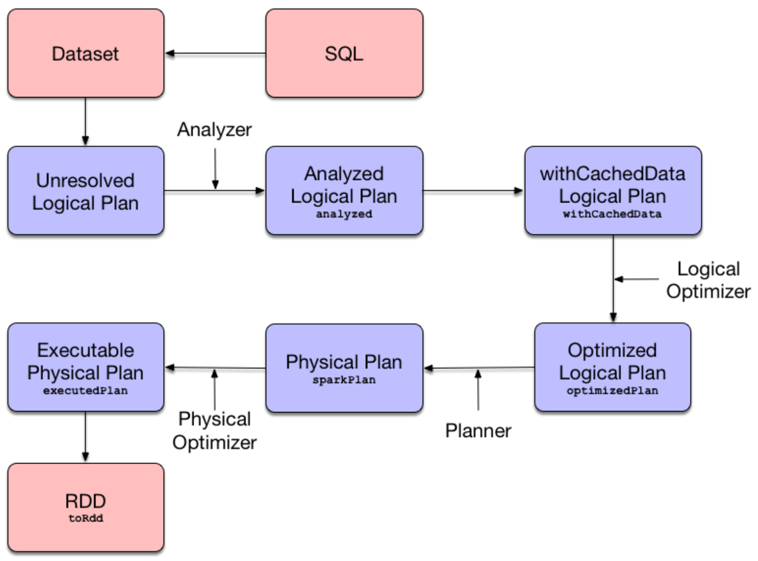

Figure 2: Query Execution — From SQL through Dataset to RDD

Below are presented the physical execution plans for the non-optimised and optimised cases. In the first as expected we see the usage of SortMergeJoin which is a map-side join. This kind of join works well if two datasets are co-partitioned and sorted based on the joint key. Co-grouping and co-partioning are strategies used to have performance improvement when multiple RDDs are to be joined. To avoid shuffling operations we can enforce co-grouping by using cogroup(rdd_1,rdd_2). In our example since the two rdds do not have the same type of data even though they are using the same partitioner they will not be partitioned in the same manner thus introducing shuffling in the case of joins. In the second case we can see the usage of BroadcastHashJoin which works well if $|rdd_1| << |rdd_2|$ and the first one can fit in RAM. The smaller rdd (or parts of it) is distributed to all the nodes mapped onto the other rdd and then for each row we probe on the join key.

```
Results for optimization disabled
== Physical Plan ==
*(6) SortMergeJoin [ID#16L], [ID#0L], Inner
:- *(3) Sort [ID#16L ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(ID#16L, 200)
:     +- *(2) Filter isnotnull(ID#16L)
:        +- *(2) GlobalLimit 100
:           +- Exchange SinglePartition
:              +- *(1) LocalLimit 100
:                 +- *(1) FileScan parquet [ID#16L,Vendor#17]
                    Batched:true,
                    Format: Parquet,
                    Location: InMemoryFileIndex
                    [hdfs://master:9000/input/yellow_tripvendors_1m.parquet],
                    PartitionFilters: [],
                    PushedFilters: [],
                    ReadSchema:
                    struct
                    <ID:bigint,Vendor:int>
+- *(5) Sort [ID#0L ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(ID#0L, 200)
      +- *(4) Project
        [ID#0L, Start#1, End#2, S_Lon#3, S_Lat#4, E_Lon#5, E_Lat#6, Cost#7]
         +- *(4) Filter isnotnull(ID#0L)
            +- *(4) FileScan parquet
            [ID#0L,Start#1,End#2,S_Lon#3,S_Lat#4,E_Lon#5,E_Lat#6,Cost#7]
            Batched: true,
            Format: Parquet,
            Location: InMemoryFileIndex
            [hdfs://master:9000/input/yellow_tripdata_1m.parquet],
            PartitionFilters: [],
            PushedFilters: [IsNotNull(ID)],
            ReadSchema:
            struct
            <ID:bigint, Start:timestamp,
            End:timestamp, S_Lon:double,
            S_Lat:double, E_Lon:double,
            E_Lat:double>

Results for optimization enabled
== Physical Plan ==
*(3) BroadcastHashJoin [ID#16L], [ID#0L], Inner, BuildLeft
:- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
:  +- *(2) Filter isnotnull(ID#16L)
:     +- *(2) GlobalLimit 100
:        +- Exchange SinglePartition
:           +- *(1) LocalLimit 100
:              +- *(1) FileScan parquet [ID#16L,Vendor#17]
                       Batched: true,
                       Format: Parquet,
                       Location: InMemoryFileIndex
                       [hdfs://master:9000/input/yellow_tripvendors_1m.parquet],
                       PartitionFilters: [],
                       PushedFilters: [],
                       ReadSchema:
                       struct
                       <ID:bigint,Vendor:int>
+- *(3) Project [ID#0L, Start#1, End#2, S_Lon#3, S_Lat#4, E_Lon#5, E_Lat#6, Cost#7]
   +- *(3) Filter isnotnull(ID#0L)
      +- *(3) FileScan parquet
              [ID#0L,Start#1,End#2,S_Lon#3,S_Lat#4,E_Lon#5,E_Lat#6,Cost#7]
              Batched: true,
              Format: Parquet,
              Location: InMemoryFileIndex
              [hdfs://master:9000/input/yellow_tripdata_1m.parquet],
              PartitionFilters: [],
              PushedFilters: [IsNotNull(ID)],
              ReadSchema:
              struct
              <ID:bigint, Start:timestamp,
              End:timestamp, S_Lon:double,
              S_Lat:double, E_Lon:double,
              E_Lat:double>
```

Finally, Fig.3 shows the experimental result of time taken with and without optimization.

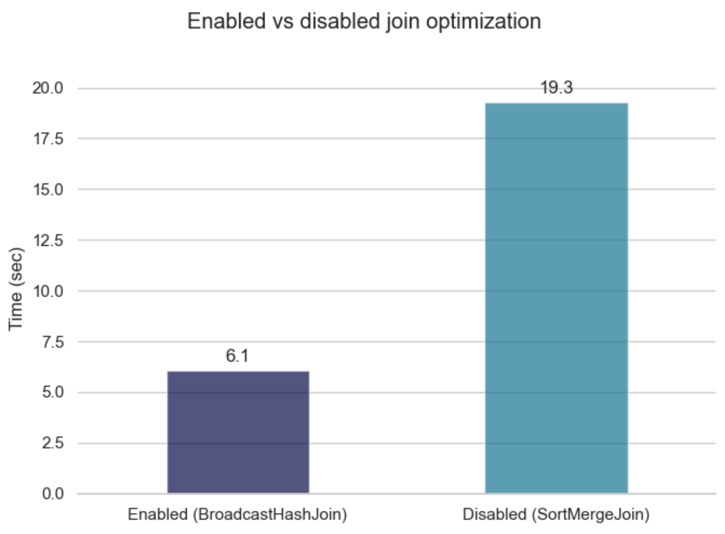

Figure 3: Time comparison between presence and absence of join optimization.

5 Machine Learning using Spark

For this part of the exercise we will be focusing on using Apache Spark for text classification. Before we begin we had to install the necessary libraries such as NLTK and numpy, but the VM provided by okeanos had python 3.5 pre-installed so we had to resort to older versions. The data used in this section are consumer complaints for products and services. In order to achieve better readability and performance we divided this task to three different files namely 'ml_preprocessing.py','tfidf_extraction.py' and 'mlp_trainer.py'. Since this is a machine learning problem the first and most crucial part of the process is the Exploratory Data Analysis. Initially we followed the proposed preprocessing steps which were:

I. Remove lines where date does not start with '201'

II. Remove empty lines

Besides that and after figuring out that there are still erroneous records we removed all rows where there were more than three columns meaning that there possibly existed some commas inside the text. Thus, having settled down to which rows we would use we moved on to data cleaning. We treated this dataset with the classic NLP cleaning pipeline which consists of the following:

1. Lowercase characters
2. Remove urls
3. Remove contraction
4. Tokenize text
5. Lemmatize using NLTK
6. Remove stop-words usign NLTK

All the aforementioned cleaning step are implemented using map and filter functions. Regarding the stop-words removal we discovered that inside the data every reference to naming and identifying elements has been converted to 'xxxx' and since we found no value in keeping these we considered them stop-words as well. When the process is finished we save the clean data to the HDFS to avoid doing this process for each experiment since the size of the dataset has been reduced to 490.996 records with 107302 unique words.

In the next step we needed to calculate the tf-idf for our data. To make our calculations realisable and due to the exercise's directives we defined a lexicon size of 500, meaning that our sentences will only contain the top 500 frequent words. Of course we did multiple experiments and evaluations and 500 provided a good trade-off between accuracy, memory needs and processing time.

Finding the most frequent words can be reduced to a set of operations as shown below with the initial data having the form (Category,Complaint):

```
most_common_words = complaints. \
    flatMap(Split words). \
    map(x,1). \
    reduceByKey(find sums). \
    sortBy(sum, ascending=False). \
    map(keep words). \
    take(lexicon_size)
```

This information concerning the most frequent words will be necessary to the next steps of the calculation so we need to broadcast it across all nodes. Next we need to remove from all the complaints the words that do not belong to the lexicon.

```
texts_with_words_in_lexicon = complaints. \
    map(Category, list_of_words). \
    map(Cateogry, list_of_words_in_lexicon). \
    filter(Remove records with empty lists)
```

So it is implied that our number of complaints is

$$N = \text{texts\_with\_words\_in\_lexicon}\,.count()$$

Now we need to find term counts. This is more complicated than the rest of our requests but it is implemented as such

```
# (Category, list_of_words_in_lexicon)
word_counts = texts_with_words_in_lexicon. \
    # ((Category,list_of_words_in_lexicon),
   sentence_id)
    zipWithIndex(). \
    # ((word,Category,sentence_id),1)
    flatMap(Create accumulator for each word in
   sentence). \
    # ((word,Category,sentence_id),
   word_count_in_sentence)
    reduceByKey(Find total word count in
   sentence). \
    # ((word,Category,sentence_id), (
   word_count_in_sentence,word_index_in_lexicon)
   ))
    map(Find word counts and indexes). \
    # ((sentence_id,Category), [word_index,
   word_count])
    map(Create lists of word counts and their
   indexes). \
    reduceByKey(Gather up list of words per
   sentence). \
    map(sentence_id , sorted list of words,index
   )
```

Having the term counts we can now calculate the number of texts containing each word. To accomplish that we created a list of 500 (#lexicon size) Spark accumulators and a special function named ('addition') which takes in a list of indices and updates the corresponding accumulators.

```
word_counts.foreach(addition(indexes of words of
     sentence))
```

After having calculated the appearances of each word in the collection we can finally calculate the tf-idf for each word.

```
complaints_tfidf = word_counts. \
    map(sentence_id,
    SparseVector(lexicon_size,
                 list_word_indexes,
                [term_frequency *
                 log(N/document_frequency)
                 ]))
```

Finally, we can transform this RDD to a DataFrame, use StringIndexer which is practically a LabelEncoder responsible for transforming categorical data to numerical and store it on the HDFS as requested in the form of:

$$(K, (ind1, ind2, \ldots, indM), (tfidf1, tfidf2, tfidfM))$$

Five rows from the tf-idf dataframe are shown below.

```
+--------------------+--------------------+-----+
|      Category_label|               tfidf|label|
+--------------------+--------------------+-----+
|     Debt collection|(500,[0,3,7,8,31,...|  1.0|
|     Debt collection|(500,[8,74,87,95,...|  1.0|
|     Debt collection|(500,[6,8,13,17,2...|  1.0|
|Credit reporting ...|(500,[0,1,3,7,8,3...|  0.0|
|         Credit card|(500,[0,1,3,8,12,...|  7.0|
+--------------------+--------------------+-----+
```

In order to perform a stratified train test split we created the training DataFrame using the samplebBy function provided by Spark and the test by subtracting the train from the initial data set.
Below are presented the counts per class for the training set which has a total of 343653 instances and testing set (with 125190 instances) accordingly.

```
+--------------------+------+ +--------------------+-----+
|      Category_label| count| |      Category_label|count|
+--------------------+------+ +--------------------+-----+
|     Debt collection| 74936| |     Debt collection|28668|
|    Virtual currency|     9| |    Virtual currency|    5|
|         Payday loan|  1204| |         Payday loan|  536|
|     Money transfers|  1046| |     Money transfers|  448|
|Checking or savin...| 13296| |Checking or savin...| 5747|
|Payday loan title...|  4600| |Payday loan title...| 1837|
|            Mortgage| 43214| |            Mortgage|18177|
|        Prepaid card|   991| |        Prepaid card|  456|
|Credit card or pr...| 22587| |Credit card or pr...| 9242|
|    Credit reporting| 22031| |    Credit reporting| 8226|
|Credit reporting ...|100441| |Credit reporting ...|26676|
|       Consumer Loan|  6593| |       Consumer Loan| 2832|
|         Credit card| 13246| |         Credit card| 5477|
|Bank account or s...| 10439| |Bank account or s...| 4387|
|Vehicle loan or l...|  5768| |Vehicle loan or l...| 2405|
|Money transfer vi...|  5487| |Money transfer vi...| 2429|
|Other financial s...|   204| |Other financial s...|   88|
|        Student loan| 17561| |        Student loan| 7554|
+--------------------+------+ +--------------------+-----+
```

The last step to this exercise is the creation and training of an MLP architecture capable of classifying texts to their corresponding Category. Even though we found pyspark.ml to be rather confining without much options regarding optimizers we experimented with multiple architectures and hyperparameters. In order to provide a fair comparison and understand whether our model is performing adequately or not, we set up a baseline model using NaiveBayes as provided by pyspark.ml. This model achieved accuracy = 56% which is definately not bad considering the requirements for this exercise was to surpass 50%.

The final architecture of our MLP is as follows:

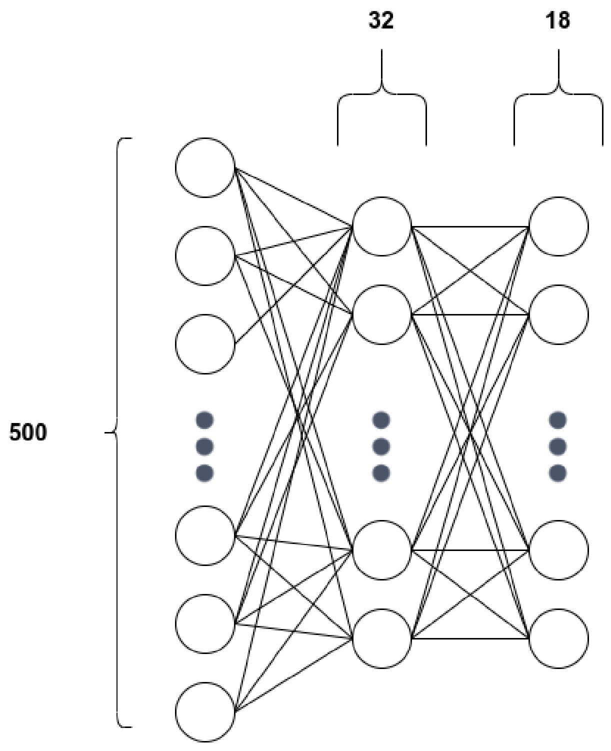

Figure 4: Our MLP architecture

We defined the max_iter parameter to be 200 and discovered that the default learning rate is the one providing the best results. With these settings our model was able to achieve 67% accuracy on the test set, while with corresponding implementations on sklearn, which were made for cross-reference, we were able to reach up to 70%.

With regards to caching/not caching the training set we receive some very counter-intuitive results. The expectation here was that caching the training set would be rather effective at reducing the time taken for the training process as the multiple fetches from the HDFS are avoided. But in our case experimental data show that the two times were very close with the best time being achieved when we were not using caching. This can be due to fact that we are using a rather large size of dictionary so maybe caching takes up a significant time to begin with or even our Sparse Vector is that big that it can't be fit in cache adding further overhead to this whole process.

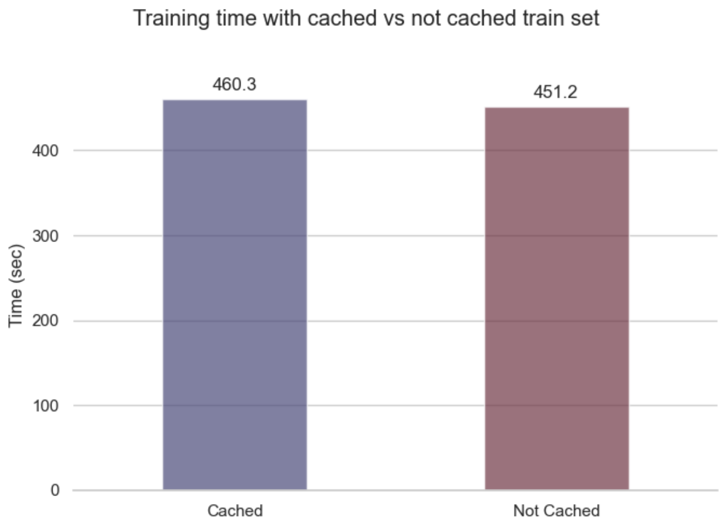

Figure 5: Time comparison for MLP training with and without the usage of caching on training data.