**2202 CS 6500 5001/MSA 6500 5001**

**Spring 2020**

**Project 3**

**Chicago Parking Tickets**

**Group 2**

**Group Members:**

Nolan Gormley

## Ravali Bongoni

## 

## Plan Content

* **Plan 1 MapReduce**

**Task0:**

Using a map reduce paradigm for this task would be fairly simple. As the data is already in a CSV format, there would be no need to modify the original data in order to work with the map reduce platform, MRJob. For this task, we would map each line with the whole value of the line as the key and a value of 1 for the value. For the reduce portion, we would only return the values that sum greater than 1. These rows would be the full duplicates in the dataset. This can be modified to check duplicates within specific columns by instead of mapping the whole line, just mapping the portion you wish to check. For example, this can be done on a column such as ‘license plate’ to see the records that have the same license plate listed. For an optional step that would give you a more consolidated result, you could add a second reducer that simply counts the number of duplicated rows coming from the first reducer, and returns it as a single number. This would give you a value that is just the rows of duplicated data based on the input, while the first solution would print out the actual rows that are duplicated.

If you choose to look into specific columns of this data for duplicate values, you will need to modify how you read in the data. Instead of reading in the singular rows, you will need to split the values on the comma and read the specific value in the list it creates. For this, you would use the built in python string function, split. To this you would pass a parameter of ‘,’ to tell the function that you would like to split the input string on commas. This would give you a list of values for the row of the data.

There are some difficulties when using map reduce for this sort of work as it is sometimes hard to visualize your data within map reduce jobs. This is different from a tool such as pyspark as you can simply print out a few rows, or look at the data types within your dataframe. This sort of thing does not really exist within a map reduce job without creating a new job to do specifically those things.

**Task1:**

Task 1 would not be very difficult within a map reduce paradigm either. These analyses would heavily reply on the split operation that was described in task 0. This is because each analysis, like most analyses, rely on the values of specific columns.

* Task 1a: time vs tickets: do certain months have more ratings?

This task would likely be the most difficult task. This is because it involved dates. Dates are tricky within map reduce as you need to decide how you want to parse them. First off, you need to find out how the dates are formatted and whether or not that formatting is consistent with the whole dataset. If it is, you have one of two options, the good option where you convert the date column to a datetime object within python and then grab the necessary data. In this case, that would be the month. The other option, which is not nearly as safe, would be to grab the substring of where the month is according to the datetime scheme in the dataset. This is a risky method as it will break with different date formats and will not give an error message, it will just grab incorrect data. The first method will fail, but it will give an error message.

After you have the month, you simply map the month and a value of one, then reduce by summing each key value. This will give you a list of tickets per month over the period of the entire dataset.

This method would be more difficult than pyspark or a similar tool as within those tools you can cast a datetime column to a datetime object, then simply get the month from it. We are essentially doing the same thing here, but it is a bit more low level and is usually a bit more tricky. Also, since the typecasting is always during runtime, it is likely that the computation time for map reduce would be higher in the case that you cast the datetime before running this query in other tools.

* Task 1b: ticket distribution: how many times does each kind of ticket appear? what are the total and average revenues by ticket?

This task would be more simple than the first. For the first part, you would map the violation code and the value 1, then sum by each key. This would give you the amount of times each violation occurred. For the second part, you would need to map the violation code and the value of the revenue from the ticket. This portion would have two reduce functions. One to sum the values and returns a sum. This function could also either by default return the number of tickets that it summed, or be modified in order to do so. This would allow for another reducer to return an average value.

This task would be much more tedious than doing it in a higher level tool, such as pyspark. This is because both parts of this could likely be done in one line in pyspark, while in map reduce, it would take 3 functions.

* Task 1c: who are the top-10 cars with the most number of tickets?

As we have learned by doing an initial look into the data, license plate numbers are not unique in themselves. This is because two license plates can be duplicated between states. Interestingly enough, there is also a discrepancy in the data between license plate numbers after factoring in the state when we look at the license plate type. Because of this, we believe that the true unique key for a single car is the license plate number + license plate state + license plate type. Because of this, we will map this as the key and Null for the 1 for the value. Interestingly enough, the most difficult part about this task is limiting the list to 10. The first reducer will sum the tickets for each car. The next part of the task is likely the hardest part of any of these tasks. In order to get the top 10 cars, the list would have to be sorted. This is not a simple thing to do within map reduce as it is a distributed system. For this, likely we would likely use an implementation of Terasort, which is a Hadoop sorting method. After the data is sorted, we would have another reducer that uses a global variable to control how many rows it has grabbed.

This task would be much more difficult than in a language like pyspark, as sorting in pyspark is very easy. It is also easy to just grab n rows of data through a function like head in Pandas.

* Task 1d: who are the top-10 cars with the largest total fines across all their tickets?

This task will be very similar to task 1c, but instead of just reducing and sorting based on the count of the tickets for each car, we will count the fine for each ticket. By adding this into the mapper, we can use the same two reducers from task 1c.

This task has the same limitations because of map reduce as task 1c.

**Task2.1: Cluster Analysis**

As map reduce is a very low level paradigm, it would not be sensible to cluster within map reduce alone. Because of this, we would reduce the data as much as possible, by grabbing only the rows we need. This would shrink the size of the data by a large amount as the majority of the data is textual data that is also already encoded separately. We would do this by mapping only the features, ‘officer’, ‘violation-code’, ‘location’. We would likely not have a reducer in this script as we need all of the data in the set in order to cluster the data appropriately. After this, we would export the data to a smaller CSV. With this, we would load it into python and run our cluster analysis through Sci Kit Learn’s package.

This is much less computationally effective than using a tool like pyspark. This is because pyspark also has a ML library built in, allowing you to run clustering algorithms while keeping the highly effective backend of a big data tool.

**Task2.2: Predictive Analysis**

This task would be much more difficult to use in the same way we used the cluster analysis. This is because we are not sure what features we would want to use in this predictive analysis. In order to find out, we would likely run an algorithm such as penalized regression that would give us feature importances in order to grab only the most important features. This is something that you can do within a map reduce paradigm, but it’s not an easy method and implementing these methods aren’t usually as computationally favorable as a packaged method. Because of this, we would likely reduce the dataset by taking only the textual fields out that are already encoded through other features, then load it into python, much like the cluster analysis. We would first run the Lasso method before mentioned, then fit a random forest or simple neural network to the data. This would likely require a system with a lot of main memory in order to run a model on such a large dataset. After the model is fit, we would predict whether or not the person paid their tickets and collect metrics on how well the model fit.

This would require a very nice computer in order to run as a dataset this large would take a lot of memory and computation time. On pyspark, you could run similar algorithms with a big data backend which is built for datasets such as this. Pyspark would run this much better, but each analysis would likely return similar results.

* **Plan 2 Pyspark**

**Task0:**

By using pyspark, we could import pyspark.sql. Filtering would become fairly easy as well. First, we can select columns and count distinct rows. It will return a number. We can compare the number with the row number of original data. If these two numbers are not equal, we can look for the duplicates then drop them. We want to check if the ticket\_number is unique. Additionally, we want to pick a composite key that can represent the ticket\_number and check if there does not have any duplicates in the original data.

**Task1:**

By using pyspark, we would build up a data frame first, and count, group, and order the variables needed. We need to modify the format of some variables, such as time(we can split it to year and month for easier filtering), and group up some variables, such as license plate number since the same license number can have different states and types. For analyzing the data, we would see the frequencies of month appeared, ticket numbers appeared, and license plate numbers appeared, also calculate the average and sum of the revenues and fines to see the distributions. The potential problem could be whether we state the conditions learly for showing the correct answer. When running the codes, the errors could be various, especially in the step of calculating the avg and sum for grouping variables. Since the data is much larger than last time, the time taken in processing could be much longer than before. The time of analyzing data for grouping them and finding the average and total number would be the most concerning to me.

* Task 1a: time vs tickets: do certain months have more ratings?

At first we needed to split the time variables in the data frame from task 0 to year and month, this could be similar to the step of splitting the timestamp that I used to do on project 2. By importing time and datetime, we can make the format to %Y %m, and then seperate it to year variable and month variable. And then, we can see the frequency of each month to find which month has the most records, which means we have more ratings on that month. We can use cube function or group by, count, and order by functions to reach the goal.

* Task 1b:ticket distribution: how many times does each kind of ticket appear? What are the total and average revenues by ticket?

To see the distribution of the tickets, we can also use the cube function or group by, count, and order by functions to show the frequencies of violation code since it asks about the type of tickets. The types of tickets are defined by violation codes. And then, grouping each ticket type by violation code, we need to sum and avg the total payments, which is the revenues, to find the total and average revenues of each ticket.

* Task 1c:who are the top-10 cars with the most number of tickets?

Since we found that some license plate numbers come from different states and different types by distinct function, so when we group up the records by license plate number, we also need to take license plate state, and license plate type into consideration. After creating a new data frame which selects the distinct license plate numbers, we can use count to show the frequency of each license plate. Then use order by function to find the top 10 license plate numbers that have the most tickets.

* Task 1d:who are the top-10 cars with the largest total fines across all their tickets?

Using the same data frame that groups license plate numbers from each state and each type, we need to sum up the total payments of each license plate number to get the total fine. And then ordering the total fine to get the top 10 license plate numbers have the most total fines.

**Task2.1 Cluster Analysis:**

We would do cluster analysis using Jupyter notebook in python. We would focus on retrieving the officers who fines similar tickets based on the violation code using Sci Kit Learn’s package. It seems like all the variables are not of the same data type in the data. The data is to be modified in python as required for the analysis. First we would take 10000 observations of data and then find a utility matrix to find the officers with similar tickets. We believe it takes almost a day to figure out the data and do analysis and obtain the expected results without any issues. The possible issues we would face are with data preparation. We would do a k-means cluster analysis in R studio to find better clusters in the data.

**Task2.2 Predictive Analysis:**

By using Pyspark, we can use the data frame from task 0 to predict whether a person is paying his/her on time. First, we need to find the variables which can affect the probability of delay payment. For this process, we need to list the importance of impact of each variable to the delay payment, and extract the significant variables using some sort of penalized regression, such as Lasso. After finding only important variables, we can reduce the data frame for fitting models. For predictions, a random forest classifier is needed for reducing overfitting. At last, we can choose the best model, collecting the coefficients, and see whether the prediction is accurate enough.

**Which plan do you prefer - and why?**

The plan using Pyspark, which is plan 2 is preferred by us from the beginning. Compared with MapReduce, Pyspark could be easier for some of our members who do not have a CS background. The processes and outputs of using Pyspark could be clearer, and show enough messages we want. Beyond this, Pyspark allows us to keep all of our work in a single atmosphere. If we went with our map reduce plan, we would have to change our tool in order to do our analyses for task 2. Beyond this, many of the tasks are significantly more complicated in map reduce which is time that can be saved by using Pyspark.

**Which plan did we choose for this project and why?**

For this project, we did not strictly follow either one of these plans. With that being said, we followed plan 2 much closer as we used pyspark for most tasks within this project. There were a few times using different tools was just simply easier and more effective with working with the data. For example, when attempting to filter down the original, 22 GB dataset to get unseen data for the predictive model, we needed to use a map reduce script along with some other basic shell commands in order to split up the data and gather the information we were looking for. Part of this could be done through pyspark, but it would have taken longer and been more likely to run out of memory as it was doing when we attempted to load the entire dataset. We also used Pandas and Sci-Kit learn in this project for cluster analysis. This is because it is easier to visualize data through Pandas and we were more familiar and comfortable with these python tools than we were with the clustering tools that are build into pyspark.

### 

### **Task 0 - Filtering/Cleanup**

**Step 1:**

We collected chicago\_parking\_tickets.zip from docker image. After unzipping the file, we got the parking\_tickets.csv, the schema, and the unit\_key.csv. The data we need is parking\_tickets.csv which is 2.3 Gigabyte.

**Step 2:**

We built a spark environment on our laptops, and processed the data in a jupyter notebook. Then we imported SQLContext from pyspark.sql, and we were ready to clean up the data.

**Step 3:**

There are 8334543 records in this dataset. We first checked all ticket numbers, and the result was 8334543 tickets. So, we found there were not any duplicates.

input.select("ticket\_number").distinct().count()

**Problems Occured:**

We picked a few columns that can identify a ticket number in our thoughts, which are issue data, violation location, license plate number, violation code, and officer. The result was 8326001, so we lost 8532 records from this count.

input.select("issue\_date","violation\_location","license\_plate\_number","violation\_code","officer").distinct().count()

Then we tried to figure out why we lost 8532 records. Then we found there were 2764216 license plate numbers but 2899185 license plate numbers with their license plate state and license plate type. We realized that some vehicles might have the same license plate number. The schema said “ license\_plate\_number: contains a hash, making it possible to determine whether tickets were issued to the same vehicle, while anonymizing the actual license plate numbers.” We all agreed and assumed that the license plate numbers were able to identify a vehicle. 8532 records are a small part of this dataset. We decided to ignore these 8532 records and use the full dataset to do all the following tasks.

### **Task1 - Exploratory Analysis Tasks**

For this task, we use pyspark to analyze our data as we talked about at our plan. On pyspark, we mainly use a relatively simple way for us, which is define a relevant function for getting the data and features we want from the original data, and then analyse it as needed. After having the outputs we want, I simply gathered the outputs to tables and graphs by Excel. So, at first, we use the following code to set up and prepare the data for the next steps.

*sc.stop()*

*sc = SparkContext(appName="test")*

*data=sc.textFile('hdfs:///p3/parking\_tickets.csv')*

*def header\_filter(item):*

*if "ticket\_number" in item:*

*return False*

*else:*

*return True*

1. time vs tickets: do certain months have more ratings?

For this part, we planned to split the date to year, month and day. But for convenience, if we do not need to analyze another part of the date such as year and day, it is fine to not split them into 3 features. At last, I just count the frequency of months in each year because this year range is 2007 to 2009. As a result, we can see in July, 2007, there were most ticket records. Summing up the records by each month, between 2007-2009, April has more records than others.

It takes 3-5 minutes to filter the date from the data, count the frequency of each month and sort them by order.

Code and part of output:

def time\_vs\_tickets(item):

item = item.split(",")

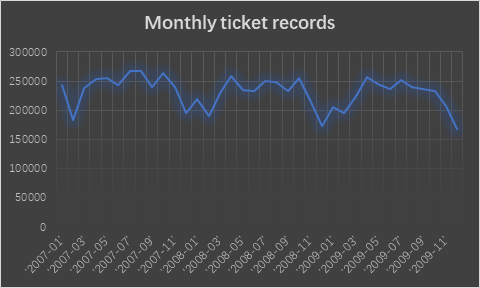
issue\_date = "-".join(item[1].split(' ')[0].split('-')[:-1])

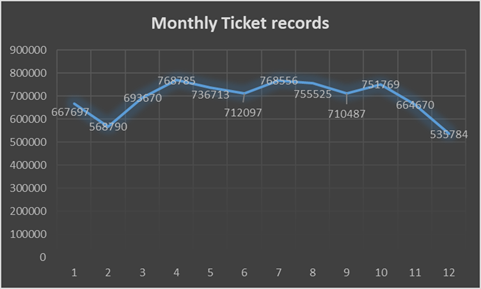
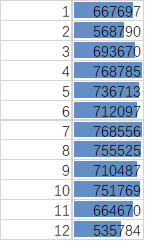
return issue\_date,1

frequecy\_of\_months = data.filter(header\_filter).map(time\_vs\_tickets).reduceByKey(lambda x,y:x+y).collect()

sorted(frequecy\_of\_months,key = lambda x:x[1],reverse=True)

[('2007-07', 267413),('2007-08', 266785), ('2007-10', 263316),('2008-04', 259614),('2009-04', 256206), ('2007-05', 255659), ('2008-10', 254727), ('2007-04', 252965), ('2009-07', 251198), ('2008-07', 249945), ('2008-08', 248696),('2009-05', 245637), ('2007-06', 243078), ('2007-01', 242614), ('2007-09', 240335), ('2009-08', 240044), ('2007-11', 239129), ('2007-03', 238501), ('2009-09', 236807), ('2009-06', 236019), ('2008-05', 235417), ('2009-10', 233726), ('2008-09', 233345), ('2008-06', 233000), ('2008-03', 230374), ('2009-03', 224795), ('2008-01', 219049), ('2008-11', 217378), ('2009-11', 208163), ('2009-01', 206034), ('2009-02', 195622), ('2007-12', 194940), ('2008-02', 190070), ('2007-02', 183098), ('2008-12', 172557), ('2009-12', 168287)]





1. ticket distribution: how many times does each kind of ticket appear? What are the total and average revenues by ticket?

To find how many times each kind of ticket appears, we count the frequency of each violation code. Part of the results are shown below, violation code ‘0976160F’ has most tickets. From the graph of the frequency of each violation code, about 75% of them are having frequency lower than 10.

The process of filtering data and counting the frequency of each violation code, and ordering them takes about 4-6 minutes since violation codes have over hundred groups. The process of grouping them and calculating the average and total revenues takes 5-7miniutes to show the all output.

Codes and part of output:

def ticket\_distribution(item):

item = item.split(",")

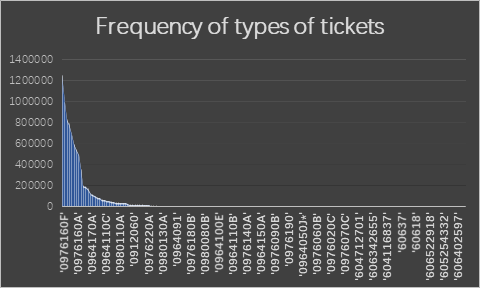
violation\_code = item[7]

return violation\_code,1

frequency\_of\_violation = data.filter(header\_filter).map(ticket\_distribution).reduceByKey(lambda x,y:x+y).collect()

sorted(frequency\_of\_violation,key = lambda x:x[1],reverse=True)

[('0976160F', 1249723), ('0964040B', 985104), ('0964190', 824807), ('0964090E', 785689), ('0964125', 676528), ('0964150B', 591569), ('0976160A', 544984), ('0964080A', 487447), ('0964190A', 357457), ('0964140B', 199949), ('0964080B', 191656), ('0964100A', 170171), ('0964170A', 117976), ('0964190B', 115501), ('0976220B', 98901),('0964130', 83932), ('0964110A', 81692), ('0976160D', 67218), ('0964110C', 62236).......]



To find the total and average revenues for each ticket, since there is no duplicate in ticket number, the total and average revenue of each ticket would be payment itself. So, we mainly analyze the total and average revenue of each kind of ticket, which is based on violation code, the part of the result is shown. The distribution of total revenue is not as balanced as we already know that the frequency of each violation code is not so balanced. Most of them are having low total revenue because they have lower frequency. But on average, the revenue of each violation code is relatively distributed even by each violation code, few of them have much higher payments than others. The average revenue also has a lower range for revenues, which makes more sense.

Codes and part of outputs:

def avg\_total\_revenues(item):

try:

item = item.split(",")

ticket\_number = item[0]

total\_payments = item[15]

return ticket\_number,(float(total\_payments),1)

except:

pass

avg\_total\_revenues\_num = data.filter(header\_filter).map(avg\_total\_revenues).filter(lambda x:x).reduceByKey(lambda x,y:(x[0]+y[0],x[1]+y[1])).map(lambda x:(x[0],x[1][0],x[1][0]/x[1][1])).collect()

sorted(avg\_total\_revenues\_num,key = lambda x:x[1],reverse=True)[:10]

[(u'53251012', 2000.0, 2000.0), (u'57340766', 1970.0, 1970.0), (u'52136636', 1457.0, 1457.0), (u'55122862', 1335.05, 1335.05), (u'54316578', 1252.26, 1252.26), (u'9177486685', 1208.0, 1208.0), (u'56584567', 1202.0, 1202.0).......]

def avg\_total\_revenues(item):

try:

item = item.split(",")

violation\_code = item[7]

total\_payments = item[15]

return violation\_code,(float(total\_payments),1)

except:

pass

avg\_total\_revenues\_num = data.filter(header\_filter).map(avg\_total\_revenues).filter(lambda x:x).reduceByKey(lambda x,y:(x[0]+y[0],x[1]+y[1])).map(lambda x:(x[0],x[1][0],x[1][0]/x[1][1])).collect()

for violation\_code,total\_payments,avg\_payments in avg\_total\_revenues\_num:

print("violation\_code:",violation\_code,"total\_payments:",total\_payments,"avg\_payments:",avg\_payments)

violation\_code: 0976140A total\_payments: 27352.430000000004 avg\_payments: 79.05326589595377

violation\_code: 0964030B total\_payments: 223077.84000000003 avg\_payments: 40.493345434743155

violation\_code: 0976210B total\_payments: 350415.48 avg\_payments: 20.20384455719557

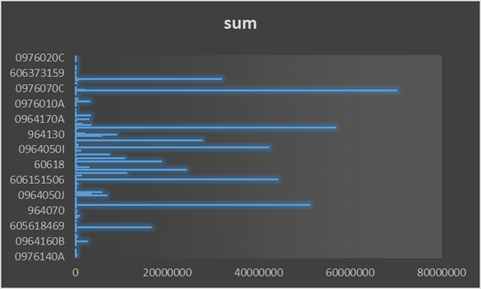
violation\_code: 0980220 total\_payments: 500.0 avg\_payments: 500.0

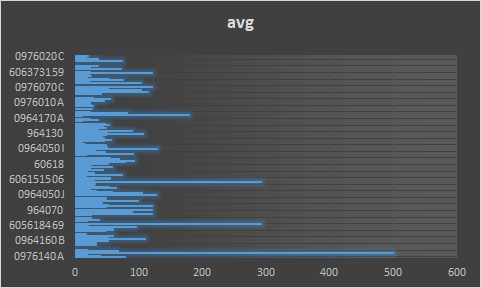
violation\_code: 0964210 total\_payments: 21972.0 avg\_payments: 39.732368896925855

violation\_code: 0964110H total\_payments: 20949.71 avg\_payments: 67.57970967741936

violation\_code: 0976120 total\_payments: 21850.42 avg\_payments: 20.23187037037037

……..





1. Who are the top-10 cars with the most number of tickets?

At first, we try to find the top 10 cars with the most tickets based on different states and types of license plates. Because we found that the ticket records on only license plate numbers and on state and types are different. But the top 10 cars, which is license plate numbers with most tickets based on different states and types, the 9 of them have the same plate number but different state and plate. Among 9 of them, even 6 of them are from the same state but different types. So, we know that it wouldn’t be meaningful if we take license plate state and types into consideration. Therefore, we count out the top 10 cars which have most tickets. The top of the top, the car with license plate number ‘603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d’ has the most ticket no matter what situation.

This part and part d take the longest time. License plate numbers have much more groups than the violation code and months. In total, I took about half a day to get the output of the last 2 questions. Actual process to running code doesn’t take much time, the main problem is the error of showing when the docker memory is not enough. With the limit condition of my computer, setting the memory to the largest, I need to rerun the initial codes to get into the pyspark, and reprocess the data and try to get out the outputs. When I have some errors on the codes, I need to redo the whole process again to try if the changed one is correct and showing the correct output. On my computer, the error of unable to connect to java server would show again and again unless I reenter the pyspark. That would be the most annoying part.

def top\_10\_cars(item):

item = item.split(",")

license\_plate\_number = item[3]

license\_plate\_state = item[4]

license\_plate\_type = item[5]

return (license\_plate\_number,license\_plate\_state,license\_plate\_type),1

top\_10\_cars\_count = data.filter(header\_filter).map(top\_10\_cars).reduceByKey(lambda x,y:x+y).collect()

sorted(top\_10\_cars\_count,key = lambda x:x[1],reverse=True)[:10]

[((u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', u'IL', u'PAS'), 124290), ((u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', u'IL', u'""'), 15289), ((u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', u'""', u'NON'), 5805), ((u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', u'IL', u'TRK'), 1799), ((u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', u'""', u'PAS'), 1378), ((u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', u'IL', u'TMP'), 1027), ((u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', u'IL', u'NON'), 850), ((u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', u'""', u'OTH'), 715), ((u'e9da919cc452d86e9eef0021d53660d97af0222726b6b91a5bed124d129aceed', u'IL', u'PAS'), 647), ((u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', u'IN', u'PAS'), 447)]

def top\_10\_cars(item):

item = item.split(",")

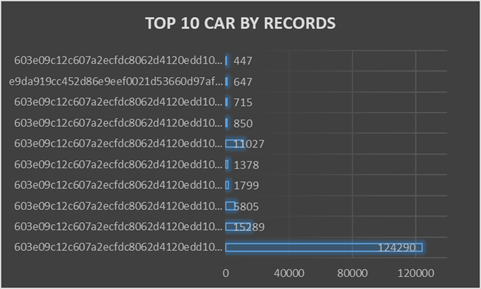
license\_plate\_number = item[3]

return license\_plate\_number,1

top\_10\_cars\_count = data.filter(header\_filter).map(top\_10\_cars).reduceByKey(lambda x,y:x+y).collect()

sorted(top\_10\_cars\_count,key = lambda x:x[1],reverse=True)[:10]

[(u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', 154741), (u'e9da919cc452d86e9eef0021d53660d97af0222726b6b91a5bed124d129aceed', 651), (u'a865795339abe6481e1d7ff0045bf85da19df648f60e2829ae6b4066130efe53', 473), (u'8298fb7281dd7e9b37c8ac5cd60b34c987511ecc04efcee872232bde2fdb26a6', 384), (u'3271411212d41ce14c94d123b71b40c3dced2de3300106abcd973ed1db33e008', 378), (u'17708b2238277b0174cc14bcff02d23b58700d8ee863672d5def039a47a83d32', 365), (u'b1b2c46d45086d1ff33be06e086114fdd71a458a94bd31220824232ad4a7045a', 333), (u'66830a0de93c5c86b69703d03e78c50e82f3b14012f3bd70aff9a75425077e1c', 326), (u'bb37fb760ccc0197fa6019b7de94f76e5da87f55a59030d5c990ee4a8cb62402', 320), (u'ba6af903e9e298b15a113b015b11f853c1ba9e530ae9ea29501e94d47bf1eb09', 301)]



1. Who are the top-10 cars with the largest total fines across all their tickets?

Based on what we found from part c, we decided to focus on only the license plate number. I used a similar way with part b to calculate the total payments of each license plate number and found the following 10 cars with most total payments across all their tickets. The winner still is '603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d' with total payment of 257525.25 across 154741 tickets.

As I mentioned above, this is the most time cost part, and because we need to sum the total payment, I tried several ways to get the appropriate output.

def top\_10\_cars\_fines(item):

try:

item = item.split(",")

license\_plate\_number = item[3]

total\_payments = item[15]

return license\_plate\_number,(float(total\_payments),1)

except:

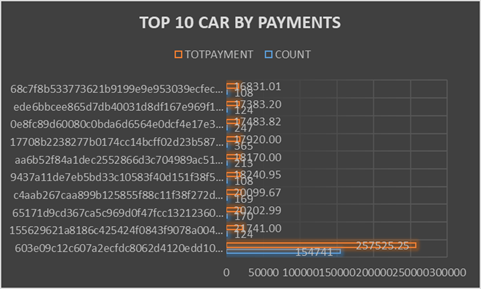
pass

top\_10\_cars\_fines\_num = data.filter(header\_filter).map(top\_10\_cars\_fines).filter(lambda x:x).reduceByKey(lambda x,y:(x[0]+y[0],x[1]+y[1])).collect()

sorted(top\_10\_cars\_fines\_num,key = lambda x:x[1],reverse=True)[:10]

[(u'603e09c12c607a2ecfdc8062d4120edd10b2f5499d76fb4cc5d7a8ec73f9e04d', (257525.25, 154741)), (u'155629621a8186c425424f0843f9078a004f1b5b8fc864ff378b284c75aa74e9', (21741.0, 124)), (u'65171d9cd367ca5c969d0f47fcc13212360dcfd87cb38e2af0b2bd7aa2b9e817', (20202.990000000002, 170)), (u'c4aab267caa899b125855f88c11f38f272d1e88fd59e9a024de7e682ec234040', (20099.670000000002, 169)), (u'9437a11de7eb5bd33c10583f40d151f38f5b5c80d5de2ff2155a8111892f373a', (18240.949999999997, 108)), (u'aa6b52f84a1dec2552866d3c704989ac519d494351da70b6ab00dab462430189', (18170.0, 213)), (u'17708b2238277b0174cc14bcff02d23b58700d8ee863672d5def039a47a83d32', (17920.0, 365)), (u'0e8fc89d60080c0bda6d6564e0dcf4e17e39d14676dbdc83fdc2594d401c234b', (17483.82, 247)), (u'ede6bbcee865d7db40031d8df167e969f176abd839ebec05cc78fb128a0ebe3b', (17383.200000000004, 124)), (u'68c7f8b533773621b9199e9e953039ecfec5378b385ff90bbbd51e4c048288e5', (16831.010000000002, 108))]





## 

### **Task 2-1 - Predictive Analysis:**

In this predictive analysis, we set out to find out whether or not we could predict whether or not the ticket recipient has paid the ticket or not. Our base assumptions with this analysis was that some of the variables in this dataset would help in predicting whether or not someone would leave a parking ticket unpaid. Some of these factors we thought would show predictive power was the price of the ticket, whether or not the recipient was from out of state, and where in the city the ticket was given. This analysis was done completely through PySpark from beginning to end. We went through many steps before modeling to help ensure that we made the best decisions possible for our predictive model. Firstly, we did an initial inspection of the data to find anything we may want to clean. Next, we attempted some feature engineering to create features that will be significant to the model. After this, we ran statistical tests on these features to attempt to find their predictive power in the model before the modeling phase. Lastly, we choose the best features and begin the modeling phase. For the models, we will be running a random forest classifier and a multilayer perceptron classifier, otherwise known as a neural network.

In our initial inspection of the data, we found a few problems. First off, there are some missing values scattered across the dataset. We initially allowed these values to stay in the model and cast them as NaN values in our final vector. Later on, we decided to remove these values as we believed them to have a negative effect on our model. The zip codes in this dataset are also non-standardized. Sometimes they are given as a five digit zip code, others they include the extra four digits that are included on more specific zip codes. In this dataset, we also found that if we wanted to use location data outside of the zip code, we would have to strip it from the ‘address’ column. Lastly, to find out whether or not the ticket has been paid, we would need to check whether or not the ‘ticket\_queue’ column has the value ‘paid’.

In the feature engineering phase, we wanted to attempt to capture a few aspects of this data that we do not believe was captured well with the more structured variables. First off, we created a regular expression to get the street from the address. The code is as follows:

def getStreet(address):

expr = r"(\d\*)(.\*)"

street = re.search(expr, address)

return street.group(2)

We then cast this function into a PySpark user defined function. From there, we apply it across the address column and save the outcome into a new column. Next, we created a regular expression to get individual components out of the date column. The regular expression we created allows for us to grab the year, month, and day by specifying the group that you want. The set of functions is as follows:

def getPartialDate(date, group\_num):

expr = r"(\d\d\d\d)-(\d\d)-(\d\d)"

part = re.search(expr, date)

return part.group(group\_num)

def getYear(date):

return int(getPartialDate(date, 1))

def getMonth(date):

return int(getPartialDate(date, 2))

def getDay(date):

return int(getPartialDate(date, 3))

Just as before, we create user defined functions from each of these functions, then run them across the date column and save the result in their own column. Next, we wanted to clean the zip code column as it varied between five digit and nine digit. To do this, we created a function just like before, but the string sliced the first 5 characters and casted the value to an integer. This function had to be placed into a try except statement in order to catch some errors that it had. This is because some of the values in this column were not completely numeric. The code looks as follows:

def shortenZip(zip):

try:

return int(zip[0:5])

except:

return None

Next, we created our final predictive feature to analyse. This was a feature to hold the value of whether the recipient was out of state or not. This function simply checked the state column to see whether or not it was the value ‘IL’. The code is as follows:

def notIllinois(state):

if state == 'IL':

return 0

else:

return 1

Lastly, we created our label variable. This was whether or not the ticket was paid or not. To do this, we created a function to check the ‘ticket\_queue’ column to see if the value was paid or not. The code is as follows:

def isPaid(value):

if value == 'Paid':

return 1

else:

return 0

After these features were created, we ran statistical tests to see the predictive power of these features against the outcome. Firstly, we created a correlation matrix with all of the features in question. As some of these features are categorical, we had to encode them before this. From this correlation matrix, we found that nearly all of these features have an extremely low correlation to the outcome variable. We settled on a subset of features with a correlation coefficient greater than .05. This was an arbitrary heuristic we created. The fact is that .05 is a terribly low correlation coefficient and likely won’t have a significant impact on the model. These features are, “Unit Description, Unit, Officer, Fine level 1, and Fine level 2”. Not all of these were used in the predictive model as Unit Description and Unit were very correlated with a coefficient of .824. Officer and Unit were also fairly correlated with a coefficient of -.786. We decided to draw out cutoff at a correlation of .8, so the Unit Description was removed from the subset, but Officer and Unit stayed in.

After we decided on the subset of features we will include in our model, we started the modeling phase. We quickly learned that this dataset was fairly imbalanced with ~70% of the observations being of paid tickets and ~30% of unpaid. Because of this, our initial models were simply always predicting that the ticket was paid. To counteract this, we had to balance the dataset. To do this, we oversampled the unpaid samples of the dataset to create a perfect split of data. This will allow the model to create an unbiased decision about the dataset. To do this we used the following PySpark code:

dfTrue = dfTemp[dfTemp['paid'] == 1]

dfFalse = dfTemp[dfTemp['paid'] == 0]

ratio = dfTrue.count() / dfFalse.count()

dfFalse = dfFalse.sample(True, ratio)

dfTemp = dfTrue.unionAll(dfFalse)

This code calculates the ratio of the two classes, then oversamples the unpaid samples with the ratio that it calculated. This essentially duplicates rows of the unpaid data in order to balance the data.

After we balanced the data, we were ready to create models for the data. We started with a random forest classifier and trained the model with 80% of the data. After the model was trained, we predicted the outcome of the test set and ran our metrics. With this model, we got an accuracy score of 58%. This is 8% higher than a random guess, which shows that our model provided some predictive power, but not much. The code is the following:

from pyspark.ml.classification import RandomForestClassifier

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

(dfTrain, dfTest) = dfFeat.randomSplit([0.8, 0.2])

rf = RandomForestClassifier(labelCol="paid", featuresCol="features")

model = rf.fit(dfTrain)

predictions = model.transform(dfTest)

evaluator = MulticlassClassificationEvaluator(labelCol="paid", predictionCol="prediction", metricName="accuracy")

accuracy = evaluator.evaluate(predictions)

print("Accuracy =", accuracy)

We repeated this process on a multilayer perceptron classifier with 3 hidden layers of varying size, not including the input and output layer. Our results were significantly lower with an accuracy score of 57%. The code is as follows:

from pyspark.ml.classification import MultilayerPerceptronClassifier

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

(dfTrain, dfTest) = dfFeat.randomSplit([0.8, 0.2])

layers = [4, 12, 8, 3, 2]

trainer = MultilayerPerceptronClassifier(labelCol="paid", featuresCol="features", maxIter=100, layers=layers, blockSize=128, seed=42)

model = trainer.fit(dfTrain)

predictions = model.transform(dfTest)

evaluator = MulticlassClassificationEvaluator(labelCol="paid", predictionCol="prediction", metricName="accuracy")

accuracy = evaluator.evaluate(predictions)

print("Accuracy =", accuracy)

After this, we wanted to run the random forest model on an unseen portion of the data. In order to do this, we downloaded the full dataset for the chicago parking data and got a subsample of only samples from the year 2012. Getting this subsample was not a simple process as the full dataset was 22 GB. This meant that the data was not able to be loaded into RAM on our computer, so we could not run the filter in a normal fashion. To fix this issue, we split the dataset into 22, 1GB chunks, then ran a map-reduce task, which grabbed only the lines that were from the year 2010, on each file. We then piped the output of these tasks to a single file. The map reduce task code is the following:

class MRActivity2(MRJob):

def steps(self):

return [

MRStep(mapper=self.mapper)

]

def mapper(self, \_, line):

line\_data = line.split(',')

try:

year = str(line\_data[1][0:4])

except:

year = ''

if year == '2010':

yield line, None

def year\_reducer(self, year, data):

if year == '2010':

yield data, None

if \_\_name\_\_ == '\_\_main\_\_':

MRActivity2.run()

This simply grabs the lines that have the year 2010 and ignores the rest. After this, we load the output file into a Python notebook to clean the data coming from the map reduce job. This removes the null key from each line and the quotations that come from this job as well. This code is the following:

in\_file = open('./mr\_output/2010\_parking.csv').read()

out\_file = open('./mr\_output/2010\_parking\_fixed.csv', 'a')

data = in\_file.split('\n')

out\_file.write(cols + '\n')

for line in data:

out\_file.write(line[1:-6] + '\n')

out\_file.close()

After this, the data was loaded into pandas, ignoring the bad lines that came from the dataset. We then filtered the data down to only the five columns that we needed, and output the data to a CSV. This CSV was loaded into the Collaboratory that we ran the original model in. We then ran the trained model on this data and resulted in an accuracy score of 64%. This was on an inbalanced dataset, much like the original dataset that we worked with. For comparison, the original dataset ran through the pretrained model came out with an accuracy score of 59%. The distribution of the classes between these datasets is also very similar with both datasets being 70% paid. This was an unexpected result as the unseen data did better than the data that this model was trained on.

## 

## **Problems Occured:**

There were many problems we ran into with this analysis that we did not suspect. First of all, many of the categorical variables had so many classes that they could not be used within a standard decision tree based model without one hot encoding them or modifying the decision tree to allow more classes per predictor. We attempted the first option without much increase in the predictive power, and the second option created a model that was extremely slow to run. On the topic of speed of these models, all of them were extremely slow to run. It also took a very long time to encode any data into a vector when we were still considering some of the categorical variables. Figuring out how to filter the complete 22 GB file was also very difficult and I had to email Dr. Dyer in order to get guidance on how to do it. Even after splitting the file and reducing it, the data had random bad characters. This made extra columns and the header did not match up. We went through the data and found the 5 variables we needed to run the model and sliced them out of the data. After this, we were finally able to save the csv and run it through the pretrained model.

### **Task 2-2 - Cluster Analysis:**

Cluster analysis aims to classify a sample of subjects on the basis of a set of measured variables into a number of different groups such that similar subjects are placed in the same group. There is no prior information about the group or cluster membership for any of the objects.In the Chicago Parking tickets data, we are concerned with finding the officers who issue similar tickets based on several factors available in the data. We downloaded the data from project pdf provided with the link,

“<https://www.propublica.org/datastore/dataset/chicago-parking-ticket-data>”

It took around 20 minutes to download the data into my google drive and then import into google colab. The number of records when downloaded the data was 562880 with 23 variables in it. As the data is huge, we faced issues loading the data directly into Jupyter Notebook or R studio. So we chose to analyze the data in Google Colab.

Ticketing officers would issue tickets mostly based on the violation reason. In this analysis we are interested in grouping the officers who issue similar tickets,what are the characteristics of these vehicles and how would people react to the fine amount charged with the tickets in particular year- 2007. As there are no characteristics of an officer other than ID, we couldn’t group the officers and determine which specific officers are ticketing simillary. So we planned to group all the attributes of tickets which indirectly cluster the officers ticketing them.

We imported several packages like pandas, matplotlib, sklearn and seaborn to use them in our analysis. First we checked if there is any missing data to drop them. Interestingly there are no missing values in the data. We checked the datatype of each variable in the data. Other than numeric, all variables are object type. We need to modify the data. The issue date is changed to “datetime” datatype and then filtered the data which has only records of 2007. There are around 180741 records in the new dataframe now. We considered the variables, violation\_code', 'license\_plate\_type', 'vehicle\_make',' fine\_level1\_amount', 'ticket\_queue', 'notice\_level', 'hearing\_disposition' for the analysis. Except fine level amount, all other variables are object type. The dataframe is modified to categorical variables from object type.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

parking\_tickets=pd.read\_csv('/content/drive/My Drive/parking\_tickets.csv')

parking\_tickets = parking\_tickets.dropna()

parking\_tickets.shape

(562880, 23)

df= pd.DataFrame(parking\_tickets)

Df.dtypes

df['year'] = pd.DatetimeIndex(df['issue\_date']).year

df2=df.loc[df['year'] == 2007]

df2.shape

#(180741, 24)

df2=df2[['violation\_code','license\_plate\_type',"vehicle\_make",'fine\_level1\_amount',"ticket\_queue","notice\_level","hearing\_disposition"]]

df2 = df2.astype('category')

df2['fine\_level1\_amount']=pd.to\_numeric(df2['fine\_level1\_amount'])

Next to perform k-means cluster analysis, these categorical variables are transformed into numerical variables based on their levels. Now the datatype of every variable would be int64. All the variables are now normalized by minmax scaler function.

from sklearn.preprocessing import LabelEncoder

lb = LabelEncoder()

df2['violation\_code'] = lb.fit\_transform(df2['violation\_code'])

df2['license\_plate\_type'] = lb.fit\_transform(df2['license\_plate\_type'])

df2['vehicle\_make'] = lb.fit\_transform(df2['vehicle\_make'])

df2['ticket\_queue'] = lb.fit\_transform(df2['ticket\_queue'])

df2['notice\_level'] = lb.fit\_transform(df2['notice\_level'])

df2['hearing\_disposition'] = lb.fit\_transform(df2['hearing\_disposition'])

df2.head()

from sklearn import preprocessing

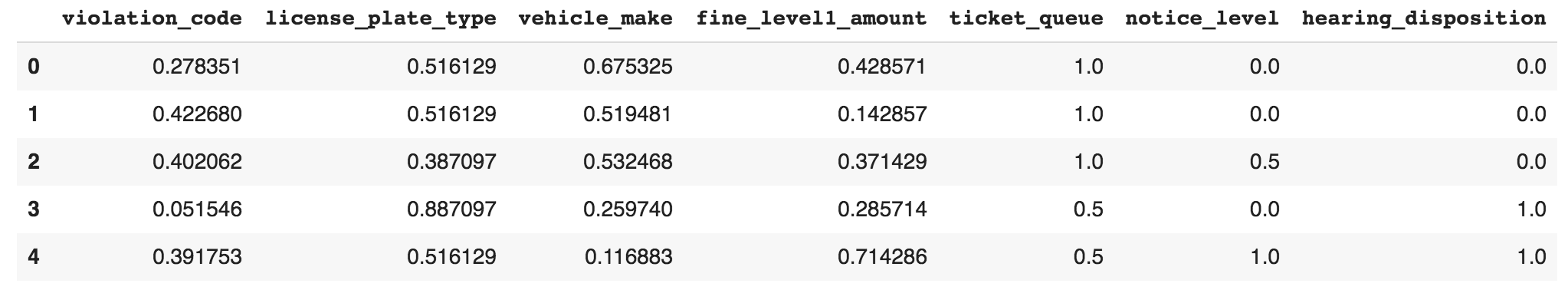
x = df2.values #returns a numpy array

min\_max\_scaler = preprocessing.MinMaxScaler()

x\_scaled = min\_max\_scaler.fit\_transform(x)

df2 = pd.DataFrame(x\_scaled)

Normalizing the variables helps the model to consider all the variables to be on the same range which is between 0 and 1. Now the datatype of variables would be float64. The output after normalization would look like,



From Python's Scikit-Learn library, we import the PCA function to perform PCA analysis. It is a popular [dimensionality reduction](https://en.wikipedia.org/wiki/Dimensionality_reduction) technique used to avoid “the curse of dimensionality”. As our data have 7 variables, it is difficult to visualize the clusters in 7D with 7 variables present. Hence PCA can be employed to select the optimal number of features that design our significant model.

from sklearn.decomposition import PCA

pca= PCA()

pca.fit(df2)

pca.explained\_variance\_ratio\_

array([0.58321522, 0.13293702, 0.11760486, 0.09702364, 0.04653128,

0.01817134, 0.00451664])

pca.explained\_variance\_ratio\_.cumsum()

array([0.58321522, 0.71615224, 0.83375709, 0.93078074, 0.97731202,

0.99548336, 1])

These PCA components can be selected considering the variance of the components. From PCA variance, we can see there are 7 components and each variance is explained. Together these 7 components explain 100% of the variability of the data. we’ll keep almost 80% of the initial variability for better performance of the model. From the above PCA cumulative variance, nearly 83% of the variance is expected from the first 3 components only. Now our data is fit with these 3 PCA components and the PCA scores are obtained for these 3 components. These scores represent our data with 3 variables but there is no significant meaning of these scores.

pca=PCA(n\_components=3)

pca.fit(df2)

scores\_pca= pca.transform(df2)

scores\_pca

array([[ 0.8308166 , -0.02511124, 0.20486481],

[ 0.81719588, -0.05489042, 0.07672566],

[ 0.52001389, 0.35442838, 0.05734115],

...,

[-0.62218384, 0.16636603, 0.41734079],

[ 0.85365116, 0.00302633, 0.41570924],

[-0.67452829, 0.07299362, -0.16221886]])

We will now incorporate the obtained PCA scores in K-means algorithm. This way we can perform segmentation based on principal components scores instead of the original features. We run the k-means algorithm with a different number of clusters and select the optimal k number of cluster values with least “Within Cluster Sum of Squares” or “WCSS” for each solution.

from sklearn.cluster import KMeans

wcss = []

for i in range(1,20):

kmeans\_pca = KMeans(n\_clusters=i,init ='k-means++',random\_state=42)

kmeans\_pca.fit(scores\_pca)

wcss.append(kmeans\_pca.inertia\_)

plt.figure(figsize=(10,8))

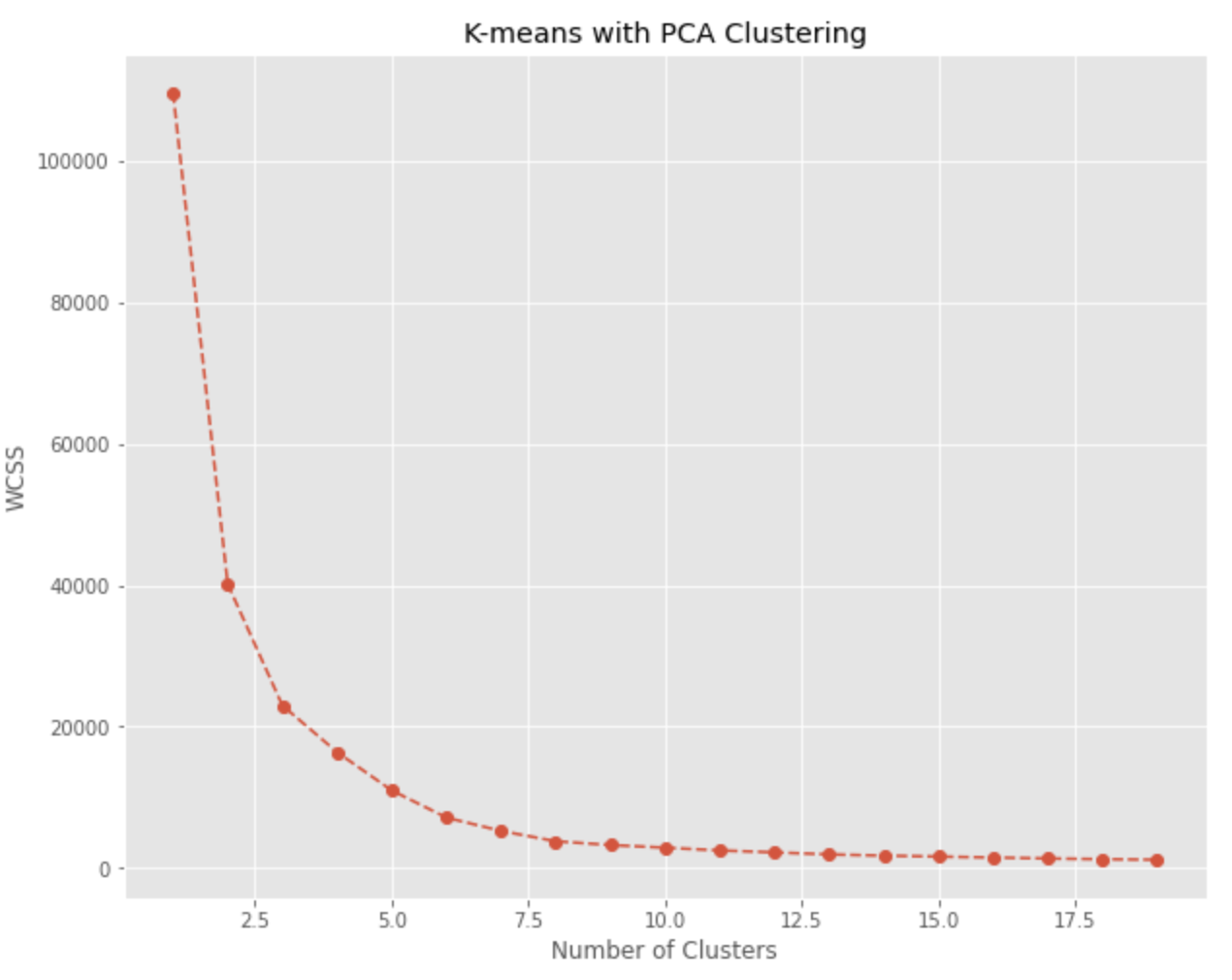
plt.plot(range(1,20),wcss,marker="o",linestyle = "--")

plt.xlabel("Number of Clusters")

plt.ylabel("WCSS")

plt.title("K-means with PCA Clustering")

plt.show()



From the above “Elbow method”, we are looking for a kink or elbow in the WCSS graph. Usually, the part of the graph before the elbow would be steeply declining, while the part after it – much smoother. In this instance, the kink/elbow comes at the 4 clusters mark. So, we will be keeping a four-cluster solution.

kmeans\_pca= KMeans(n\_clusters=4,init="k-means++",random\_state=42)

kmeans\_pca.fit(scores\_pca)

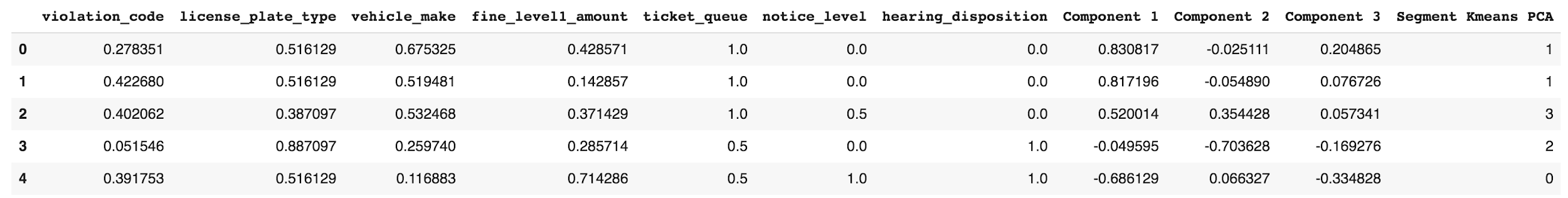
Here, we use initializer and random state to fit the model with the principal component scores. Next we created a new dataframe “segment”, in which we concat the scores of PCA components to our original dataframe. Now, the segment data frame would look like,

segment = pd.concat([df2,pd.DataFrame(scores\_pca)],axis=1)

segment.columns.values[-3: ] = ["Component 1",'Component 2','Component 3']

segment["Segment Kmeans PCA"]=kmeans\_pca.labels\_

segment.head()



segment['Segment']=segment['SegmentKmeansPCA'].map({0:'first',1:'second',2:'Third',3:'Fourth'})

import seaborn as sns

x\_axis= segment['Component 2']

y\_axis= segment['Component 1']

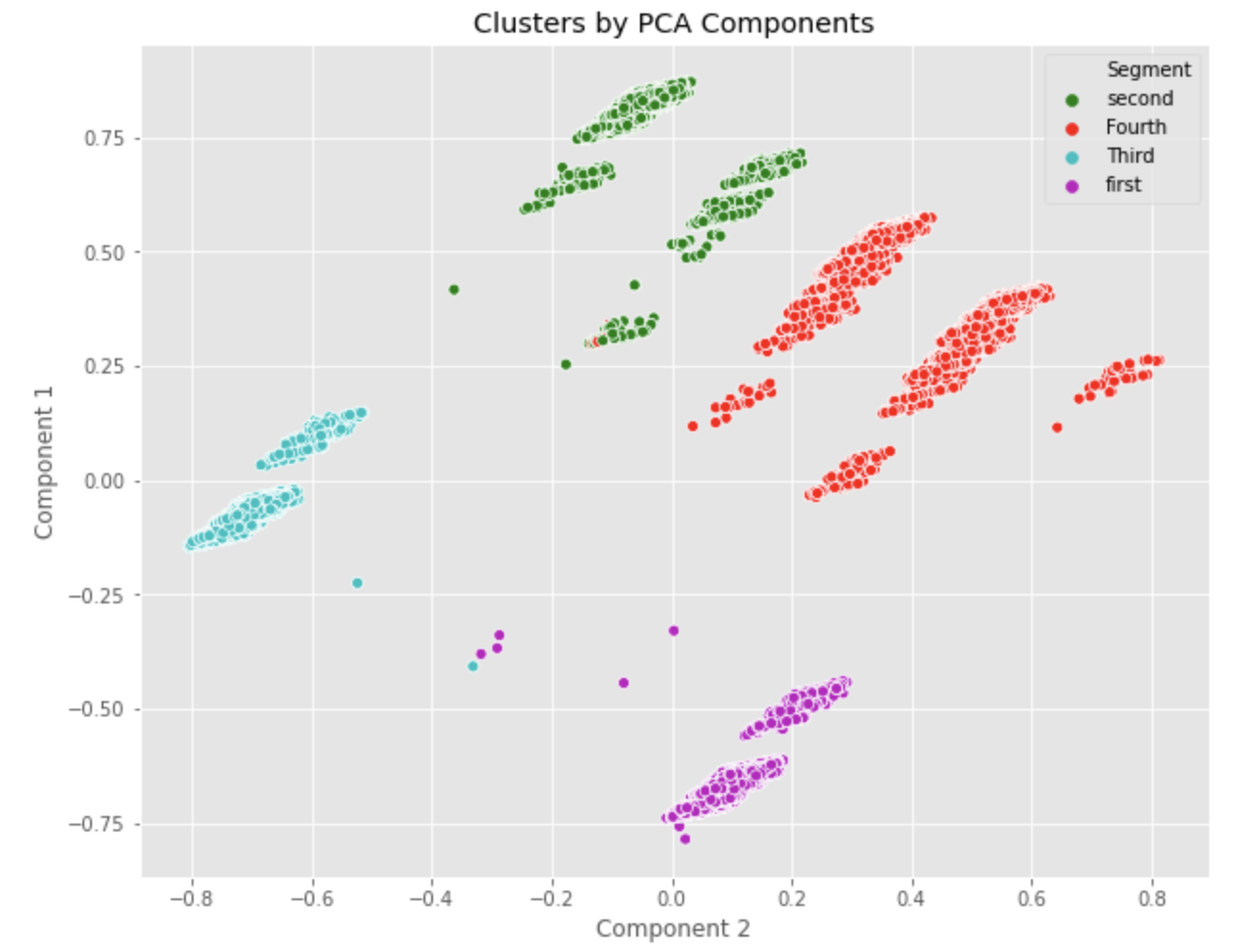
plt.figure(figsize=(10,8))

sns.scatterplot(x\_axis,y\_axis,hue=segment['Segment'],palette=['g','r','c','m'])

plt.title('Clusters by PCA Components')

plt.show()

Next we labelled the clusters and this can be visualized in the plot using Python’s Seaborn library. We used the first two components to view the clusters in 2D plot. We could see 4 clusters of data in different colors grouped by the PCA components.



The green and red clusters seem to have more correlation compared to others with the two principal components. These principal components are calculated only from features and no information from actual variables are considered. So PCA is an unsupervised method and it’s difficult to interpret the two axes as they are some complex mixture of the original features. We can make a heat-plot to see how the features mixed up to create the components.

plt.matshow(pca.components\_,cmap='viridis')

plt.yticks([0,1,2],['Component1','Component 2','Component 3'],fontsize=10)

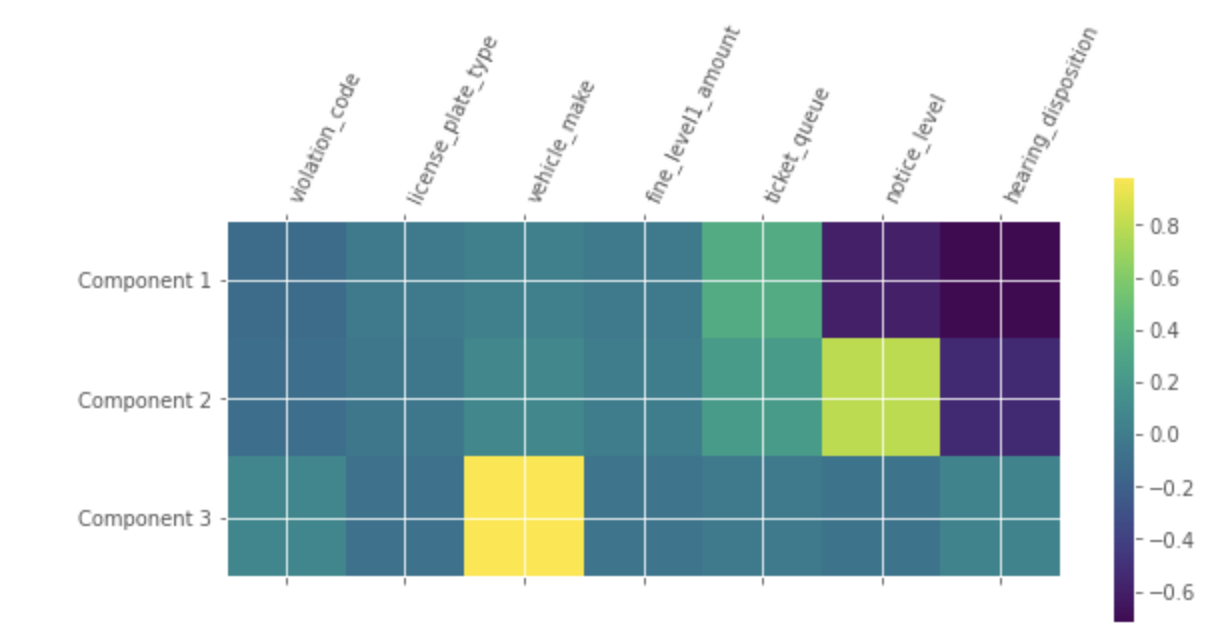
plt.colorbar()

plt.xticks(range(len(df2.columns)),df2.columns,rotation=65,ha='left')

plt.tight\_layout()

plt.show()

From Python’s matplotlib, matshow function is used to plot a heat plot to recognise the features which are more responsible in the respective principal components.



From the above plot, we could observe that, yellow-green portion indicates a high correlation close to 1. So Vehicle\_make is more significant in component 3. Relatively, there is a high percentage of ticket\_queue observations in component 1. In component 2, notice\_level variable is responsible for high variance.

**Total time taken**:

It took around 2 days for me to complete the analysis with code and obtain the results.

**Problems occured:**

As the data is huge and mixed with categorical and numeric observations, we were not sure how to perform k-means clustering as it is applicable only to numeric variables. On browsing we got to know that the k-prototype method needs to be performed with mixed data. But we faced issues in using k-prototype as it was very messy. Then we figured out that k-means clustering could be used after converting all the categorical variables to numeric. This way we solved our issue.

### **Summary**

Task 0 was to check the duplicates. There are no duplicates on ticket numbers. Task 0 is quick work, but it is the foundation of the project. In this project, it occured something we could not explain. We all got an agreement through discussion and consideration.

Task 1 summarizes the monthly distribution of ticket records among 2007 and 2009; the distribution of each kind of tickets and its total and average revenues; and top 10 cars with most tickets or payments. We got a conclusion that generally from April to October, the records are relatively high. In April, we have the most tickets among those years. The distribution of each kind of ticket is based on the violation code, and violation code 0976160F, which refers to expired plates or temporary registration has the most records. The total revenues of each kind of tickets is not distributed evenly since the difference between the numbers of tickets of each kind. So, the total revenue has really big differences among different violation codes. But on average, the payments would be relatively balanced. Comparing the outputs of top 10 cars with most tickets by only license plate number with different states and types, we found out it is not that informational if we take license plate states and types into consideration. As result, we got the license plate number ‘603e09c12c607a2ecfdc8062d4120edd10b2f549 9d76fb4c c5d7a8ec73f9e04d’ has most tickets and payments. The main problem we had for Task 1 is that when we try to run the calculation of total payment and average payment, docker memory is not enough for the calculation. We just solve it by increasing the memory. There are no other unanticipated problems.

Task 2 predictive analysis attempts to create a predictive model to predict whether or not a parking ticket was paid or not. There were many difficulties throughout this analysis including dealing with the imbalance of the data, large numbers of classes within categorical variables, and dirty data. After solving each of these problems individually, we were able to generate a predictive model that was able to predict whether or not someone had paid a ticket 8% higher than a random guess. After looking at the complete dataset that this came from, this model could likely be improved with some of the extra features that it holds such as the geocoded locations. Geocoded locations work much better in models than something such as zip code as they are a continuous variable where the value directly correlates to the location on a map, unlike zip codes. Zip codes show some of this correlation, but not always enough.

Task 2 Cluster analysis groups the set of data objects into clusters. Here our main objective was to find the cluster of officers who issue similar tickets. We were aware that there would be issues with data importing and data preprocessing as the datafile is too huge which is around 2GB. We could not load the data into jupyter notebook or Rstudio which is expected, so we changed our plan to do analysis in Google Collabs platform. Then the unanticipated issue we encountered was clustering “officer” data based on “violation code”. We haven’t noticed that officer ID values are just nominal values which aren’t either numerical or categorical to modify for the analysis. And there is no much information regarding officer other than ID like, Age or salary so that officers could be grouped. So to avoid these, we changed the plan to find clusters in the data based on all variables so that indirectly officers would be grouped with respective to other columns. We modified the plan by filtering data for the year 2007 and did not consider random 10,000 observations. We wanted to be more specific in this analysis by choosing 2007. There were around 180741observations in this filtered data. Another unanticipated issue was using k-means cluster analysis method to our data. K-means method could be applied to only numeric variables. But our data is mixed with categorical and numeric variables. On browsing, we found that the K- prototype method could be used for clustering mixed data, but it was too complex. We faced issues in implementing all the functions related to the k-prototype method. Then we decided to encode the categorical data into numeric data so that, k-means method could be applied for clustering. This way we were able to figure out the data limitations and solve them.