The following explains some of the less obvious column headers:

**cosit:** The unique identifier for the traffic counter device. In conjunction with the site’s dataset, this can be used to determine the location and route of the counter, used to record the vehicle movement. **lane:** The Id of the lane in which the movement was recorded, which is specific

to each counter. **straddlelane:** If a value is present, this indicates that the vehicle may have been

changing lanes as it passed over the counter. **class/classname:** This indicates the category of vehicle that was recorded e.g. car, bus, etc.

**length:** The approximate length of the vehicle recorded. **headway:** The approximate distance between the front of the recorded vehicle

and the vehicle behind. **gap:** The approximate distance between the rear of the vehicle and the front

of the vehicle behind. **weight:** This is available on (Weigh-in-Motion) WIM sites only and indicates the

approximate weight of the vehicle. **temperature:** If available, this indicates the approximate surface temperature of

the road at the location of the device. **numberofaxles:** This is available on WIM sites only and indicates the number of

axles detected for the vehicle. **axleweights:** This is available on WIM sites only and expresses as an array of real

numbers, the weight over each axel in order. **axlespacing:** This is available on WIM sites only and expresses as an array of real

numbers, the distance between each of the axles.

STRUCTURE OF THE DATA

cosit year month day hour minute second millisecond \

0 998 2021 1 31 2 45 0 0

1 998 2021 1 31 2 45 1 0

2 998 2021 1 31 2 45 1 0

3 998 2021 1 31 2 45 2 0

4 998 2021 1 31 2 45 3 0

... ... ... ... ... ... ... ... ...

1106647 208001 2021 1 31 16 39 55 0

1106648 208001 2021 1 31 16 40 15 0

1106649 208001 2021 1 31 16 40 20 0

1106650 208001 2021 1 31 16 40 24 0

1106651 208001 2021 1 31 16 40 33 0

minuteofday lane ... headway gap speed weight temperature \

0 165 2 ... 1.07 1.13 71.0 0.0 0.0

1 165 2 ... 1.10 1.34 69.0 0.0 0.0

2 165 1 ... 1.17 1.11 69.0 0.0 0.0

3 165 1 ... 1.00 0.72 70.0 0.0 0.0

4 165 2 ... 1.01 0.72 71.0 0.0 0.0

... ... ... ... ... ... ... ... ...

1106647 999 2 ... 17.50 17.17 77.0 0.0 0.0

1106648 1000 2 ... 19.40 19.18 67.0 0.0 0.0

1106649 1000 1 ... 37.00 36.72 61.0 0.0 0.0

1106650 1000 1 ... 4.76 4.35 62.0 0.0 0.0

1106651 1000 2 ... 18.70 18.41 49.0 0.0 0.0

duration validitycode numberofaxles axleweights axlespacings

0 0 0 0 NaN NaN

1 0 0 0 NaN NaN

2 0 0 0 NaN NaN

3 0 0 0 NaN NaN

4 0 0 0 NaN NaN

... ... ... ... ... ...

1106647 0 0 0 NaN NaN

1106648 0 0 0 NaN NaN

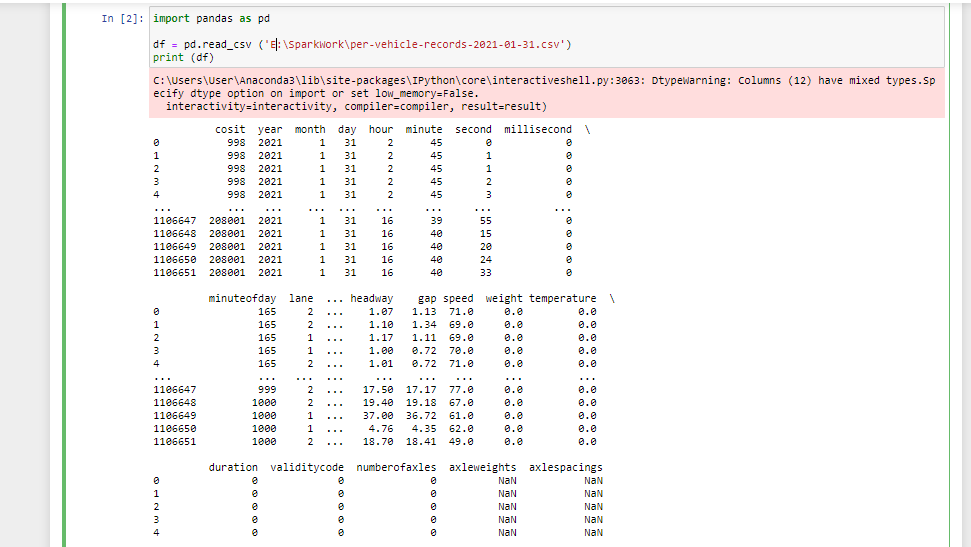
1106649 0 0 0 NaN NaN

1106650 0 0 0 NaN NaN

1106651 0 0 0 NaN NaN

[1106652 rows x 26 columns]

Screenshot



Summary Statistics of the Data

1106652 rows x 26 columns]

<bound method DataFrame.sum of cosit year month day hour minute second millisecond \

0 998 2021 1 31 2 45 0 0

1 998 2021 1 31 2 45 1 0

2 998 2021 1 31 2 45 1 0

3 998 2021 1 31 2 45 2 0

4 998 2021 1 31 2 45 3 0

... ... ... ... ... ... ... ... ...

1106647 208001 2021 1 31 16 39 55 0

1106648 208001 2021 1 31 16 40 15 0

1106649 208001 2021 1 31 16 40 20 0

1106650 208001 2021 1 31 16 40 24 0

1106651 208001 2021 1 31 16 40 33 0

minuteofday lane ... headway gap speed weight temperature \

0 165 2 ... 1.07 1.13 71.0 0.0 0.0

1 165 2 ... 1.10 1.34 69.0 0.0 0.0

2 165 1 ... 1.17 1.11 69.0 0.0 0.0

3 165 1 ... 1.00 0.72 70.0 0.0 0.0

4 165 2 ... 1.01 0.72 71.0 0.0 0.0

... ... ... ... ... ... ... ... ...

1106647 999 2 ... 17.50 17.17 77.0 0.0 0.0

1106648 1000 2 ... 19.40 19.18 67.0 0.0 0.0

1106649 1000 1 ... 37.00 36.72 61.0 0.0 0.0

1106650 1000 1 ... 4.76 4.35 62.0 0.0 0.0

1106651 1000 2 ... 18.70 18.41 49.0 0.0 0.0

duration validitycode numberofaxles axleweights axlespacings

0 0 0 0 NaN NaN

1 0 0 0 NaN NaN

2 0 0 0 NaN NaN

3 0 0 0 NaN NaN

4 0 0 0 NaN NaN

... ... ... ... ... ...

1106647 0 0 0 NaN NaN

1106648 0 0 0 NaN NaN

1106649 0 0 0 NaN NaN

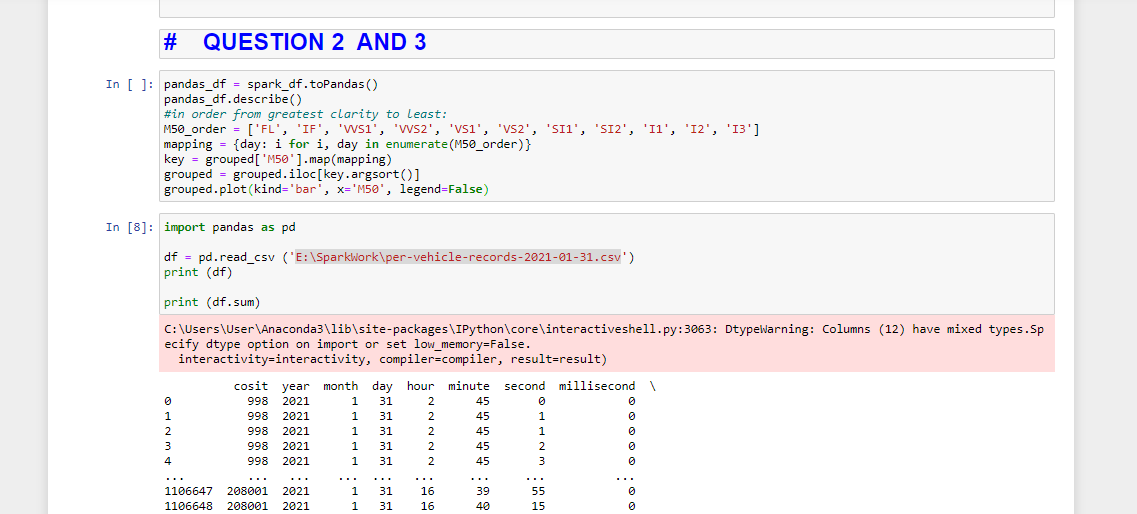
1106650 0 0 0 NaN NaN

1106651 0 0 0 NaN NaN

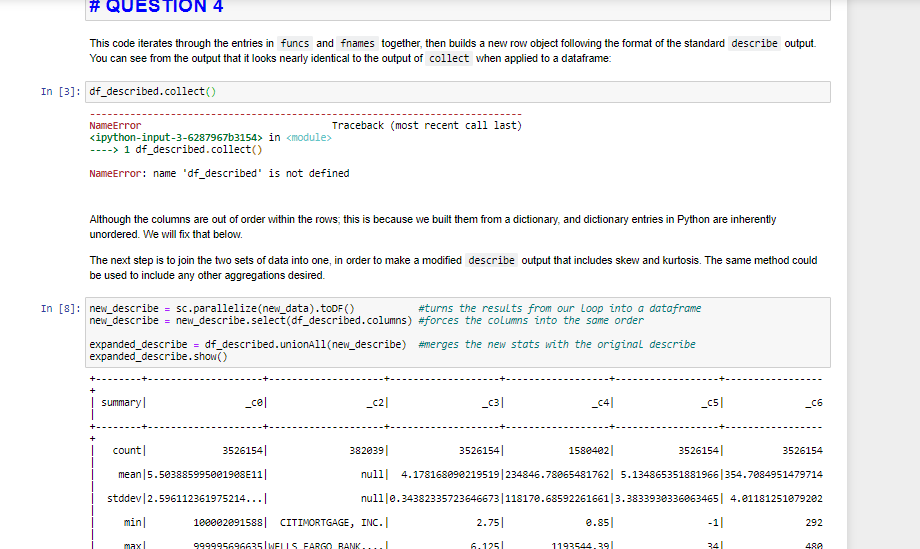
[1106652 rows x 26 columns]>

​

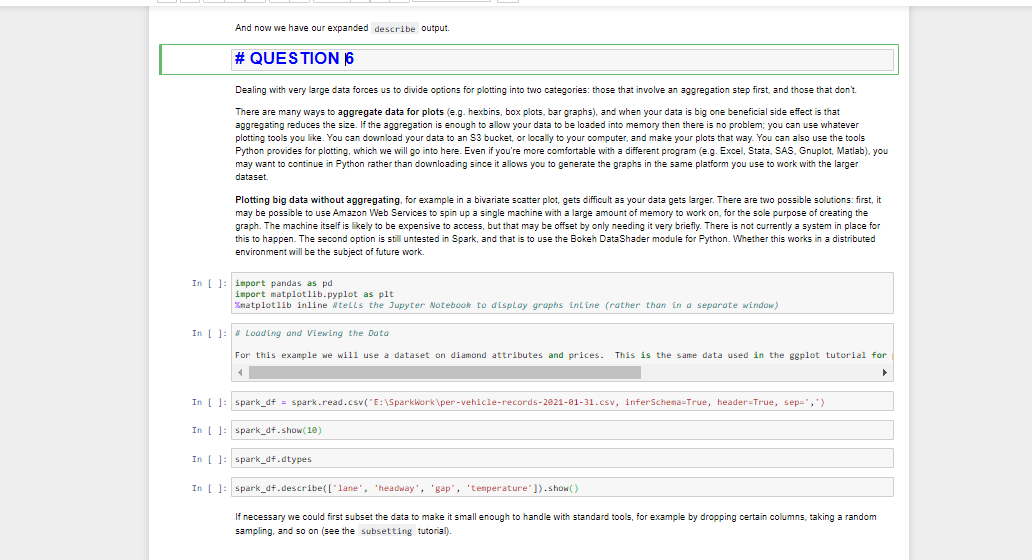
QUESTION 2 AND 3



QUESTION 4



QUESTION 6



CODE

1. Download the above mentioned dataset file. Once downloaded move the file into HDFS. Next, load the data from HDFS into a DataFrame using PySpark. Once a DataFrame is created answer the following questions.
2. Calculate the usage of Irish road network in terms of percentage grouped by vehicle category.
3. Calculate the highest and lowest hourly flows on M50 - show the hours and total number of vehicle counts.
4. Calculate the evening and morning rush hours on M50 - show the hours and the total counts.
5. Calculate average speed between each junction on M50 (e.g., junction 1 - junction2, junction 2 - junction 3, etc.).
6. Calculate the top 10 locations with highest number of counts of HGVs (class). Map the COSITs with their names given on the map.

**Summary Statistics\_**

*Abstract: Here we will cover several common ways to summarize data. Many of these methods have been dicussed in other tutorials in different contexts.*

*Main operations used: describe, skewness, kurtosis, collect, select*

**QUESTION 1**

First we will load the same csv data we've been using in many other tutorials, then pare it down to a manageable subset for ease of use:

In [1]:

vehicle\_counter\_DF = spark.read.csv('E:\SparkWork**\**

per-vehicle-records-2021-01-31.csv', inferSchema = **True**, header = **True**)

• Youcancheckthetypeof vehicle counter DF:

3

type(vehicle\_counter\_DF) *# output: pyspark.sql.dataframe.DataFrame*

*# Countthenumberoflines/recordsin vehicle counter DF:*

vehicle\_counter\_DF.count() *# output: xxxxxx (total number of records)*

*#Takealookatthefirst5linesin vehicle counter DF:*

vehicle\_counter\_DF.show(5) *# output: 5 lines from the vehicle\_counter\_DF*

*# Aseachlinehascomma-separatedvalues,youcansplititbycomma:*

vehicle\_counter\_DF.printSchema *# output: the schema of the vehicle\_counter\_DF dataframe*

**---------------------------------------------------------------------------**

**NameError** Traceback (most recent call last)

**<ipython-input-1-96410f618546>** in <module>

**----> 1** df **=** spark**.**read**.**csv**('s3://ui-spark-social-science-public/data/Performance\_2015Q1.txt',** header**=False,** inferSchema**=True,** sep**='|')**

**NameError**: name 'spark' is not defined

In [2]:

df = df[['\_c0', '\_c2', '\_c3', '\_c4', '\_c5', '\_c6']]

In [3]:

df.show(5)

+------------+-----+-----+----+---+---+

| \_c0| \_c2| \_c3| \_c4|\_c5|\_c6|

+------------+-----+-----+----+---+---+

|100002091588|OTHER|4.125|null| 0|360|

|100002091588| null|4.125|null| 1|359|

|100002091588| null|4.125|null| 2|358|

|100002091588| null|4.125|null| 3|357|

|100002091588| null|4.125|null| 4|356|

+------------+-----+-----+----+---+---+

only showing top 5 rows

Note that the format \_c0, \_c1, ..., \_cN is the default column names Spark uses if your data doesn't come with headers. For more on this, and renaming them, see the pySpark tutorial named *basics 1.ipynb*

**Describe**

The first thing we'll do is use the describe method to get some basics. Note that **describe will return a new dataframe** with the parameters, so we'll assign the results to a new variable and then call show on it:

In [4]:

df\_described = df.describe()

df\_described.show()

+-------+--------------------+--------------------+-------------------+------------------+------------------+-----------------+

|summary| \_c0| \_c2| \_c3| \_c4| \_c5| \_c6|

+-------+--------------------+--------------------+-------------------+------------------+------------------+-----------------+

| count| 3526154| 382039| 3526154| 1580402| 3526154| 3526154|

| mean|5.503885995001908E11| null| 4.178168090219519|234846.78065481762| 5.134865351881966|354.7084951479714|

| stddev|2.596112361975214...| null|0.34382335723646673|118170.68592261661|3.3833930336063465| 4.01181251079202|

| min| 100002091588| CITIMORTGAGE, INC.| 2.75| 0.85| -1| 292|

| max| 999995696635|WELLS FARGO BANK,...| 6.125| 1193544.39| 34| 480|

+-------+--------------------+--------------------+-------------------+------------------+------------------+-----------------+

Aside from the five included in describe, there are a handful of other built-in aggregators that can be applied to a column. Here we'll apply the skewness function to column \_c3:

In [5]:

**from** **pyspark.sql.functions** **import** skewness, kurtosis

**from** **pyspark.sql.functions** **import** var\_pop, var\_samp, stddev, stddev\_pop, sumDistinct, ntile

df.select(skewness('\_c3')).show()

+------------------+

| skewness(\_c3)|

+------------------+

|0.5197993394959904|

+------------------+

**Expanding the Describe Output**

One convenient thing we might want to do is put all our summary statistics together in one spot - in essence, expand the output from describe. Below I'll go into a short example:

In [6]:

**from** **pyspark.sql** **import** Row

columns = df\_described.columns *#list of column names: ['summary', '\_c0', '\_c3', '\_c4', '\_c5', '\_c6']*

funcs = [skewness, kurtosis] *#list of functions we want to include (imported earlier)*

fnames = ['skew', 'kurtosis'] *#a list of strings describing the functions in the same order*

**def** new\_item(func, column):

*"""*

*This function takes in an aggregation function and a column name, then applies the aggregation to the*

*column, collects it and returns a value. The value is in string format despite being a number,*

*because that matches the output of describe.*

*"""*

**return** str(df.select(func(column)).collect()[0][0])

new\_data = []

**for** func, fname **in** zip(funcs, fnames):

row\_dict = {'summary':fname} *#each row object begins with an entry for "summary"*

**for** column **in** columns[1:]:

row\_dict[column] = new\_item(func, column)

new\_data.append(Row(\*\*row\_dict)) *#using \*\* tells Python to unpack the entries of the dictionary*

print(new\_data)

[Row(\_c0='-0.00183847089866', \_c2='None', \_c3='0.519799339496', \_c4='0.758411576756', \_c5='0.286480156084', \_c6='-2.69765201567', summary='skew'), Row(\_c0='-1.19900726351', \_c2='None', \_c3='0.126057726847', \_c4='0.576085602656', \_c5='0.195187780089', \_c6='24.7237858944', summary='kurtosis')]

In [ ]:

998 2021 1 31 2 45 0 0 165 2 Ch 2 0 2 CAR 5.20 1.07 1.13 71 0 0 0 0 0 \* \*

**QUESTION 2 AND 3**

In [ ]:

pandas\_df = spark\_df.toPandas()

pandas\_df.describe()

*#in order from greatest clarity to least:*

M50\_order = ['FL', 'IF', 'VVS1', 'VVS2', 'VS1', 'VS2', 'SI1', 'SI2', 'I1', 'I2', 'I3']

mapping = {day: i **for** i, day **in** enumerate(M50\_order)}

key = grouped['M50'].map(mapping)

grouped = grouped.iloc[key.argsort()]

grouped.plot(kind='bar', x='M50', legend=**False**)

In [8]:

**import** **pandas** **as** **pd**

df = pd.read\_csv ('E:\SparkWork\per-vehicle-records-2021-01-31.csv')

print (df)

print (df.sum)

C:\Users\User\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py:3063: DtypeWarning: Columns (12) have mixed types.Specify dtype option on import or set low\_memory=False.

interactivity=interactivity, compiler=compiler, result=result)

cosit year month day hour minute second millisecond \

0 998 2021 1 31 2 45 0 0

1 998 2021 1 31 2 45 1 0

2 998 2021 1 31 2 45 1 0

3 998 2021 1 31 2 45 2 0

4 998 2021 1 31 2 45 3 0

... ... ... ... ... ... ... ... ...

1106647 208001 2021 1 31 16 39 55 0

1106648 208001 2021 1 31 16 40 15 0

1106649 208001 2021 1 31 16 40 20 0

1106650 208001 2021 1 31 16 40 24 0

1106651 208001 2021 1 31 16 40 33 0

minuteofday lane ... headway gap speed weight temperature \

0 165 2 ... 1.07 1.13 71.0 0.0 0.0

1 165 2 ... 1.10 1.34 69.0 0.0 0.0

2 165 1 ... 1.17 1.11 69.0 0.0 0.0

3 165 1 ... 1.00 0.72 70.0 0.0 0.0

4 165 2 ... 1.01 0.72 71.0 0.0 0.0

... ... ... ... ... ... ... ... ...

1106647 999 2 ... 17.50 17.17 77.0 0.0 0.0

1106648 1000 2 ... 19.40 19.18 67.0 0.0 0.0

1106649 1000 1 ... 37.00 36.72 61.0 0.0 0.0

1106650 1000 1 ... 4.76 4.35 62.0 0.0 0.0

1106651 1000 2 ... 18.70 18.41 49.0 0.0 0.0

duration validitycode numberofaxles axleweights axlespacings

0 0 0 0 NaN NaN

1 0 0 0 NaN NaN

2 0 0 0 NaN NaN

3 0 0 0 NaN NaN

4 0 0 0 NaN NaN

... ... ... ... ... ...

1106647 0 0 0 NaN NaN

1106648 0 0 0 NaN NaN

1106649 0 0 0 NaN NaN

1106650 0 0 0 NaN NaN

1106651 0 0 0 NaN NaN

[1106652 rows x 26 columns]

<bound method DataFrame.sum of cosit year month day hour minute second millisecond \

0 998 2021 1 31 2 45 0 0

1 998 2021 1 31 2 45 1 0

2 998 2021 1 31 2 45 1 0

3 998 2021 1 31 2 45 2 0

4 998 2021 1 31 2 45 3 0

... ... ... ... ... ... ... ... ...

1106647 208001 2021 1 31 16 39 55 0

1106648 208001 2021 1 31 16 40 15 0

1106649 208001 2021 1 31 16 40 20 0

1106650 208001 2021 1 31 16 40 24 0

1106651 208001 2021 1 31 16 40 33 0

minuteofday lane ... headway gap speed weight temperature \

0 165 2 ... 1.07 1.13 71.0 0.0 0.0

1 165 2 ... 1.10 1.34 69.0 0.0 0.0

2 165 1 ... 1.17 1.11 69.0 0.0 0.0

3 165 1 ... 1.00 0.72 70.0 0.0 0.0

4 165 2 ... 1.01 0.72 71.0 0.0 0.0

... ... ... ... ... ... ... ... ...

1106647 999 2 ... 17.50 17.17 77.0 0.0 0.0

1106648 1000 2 ... 19.40 19.18 67.0 0.0 0.0

1106649 1000 1 ... 37.00 36.72 61.0 0.0 0.0

1106650 1000 1 ... 4.76 4.35 62.0 0.0 0.0

1106651 1000 2 ... 18.70 18.41 49.0 0.0 0.0

duration validitycode numberofaxles axleweights axlespacings

0 0 0 0 NaN NaN

1 0 0 0 NaN NaN

2 0 0 0 NaN NaN

3 0 0 0 NaN NaN

4 0 0 0 NaN NaN

... ... ... ... ... ...

1106647 0 0 0 NaN NaN

1106648 0 0 0 NaN NaN

1106649 0 0 0 NaN NaN

1106650 0 0 0 NaN NaN

1106651 0 0 0 NaN NaN

[1106652 rows x 26 columns]>

**QUESTION 4**

This code iterates through the entries in funcs and fnames together, then builds a new row object following the format of the standard describe output. You can see from the output that it looks nearly identical to the output of collect when applied to a dataframe:

In [3]:

df\_described.collect()

**---------------------------------------------------------------------------**

**NameError** Traceback (most recent call last)

**<ipython-input-3-6287967b3154>** in <module>

**----> 1** df\_described**.**collect**()**

**NameError**: name 'df\_described' is not defined

Although the columns are out of order within the rows; this is because we built them from a dictionary, and dictionary entries in Python are inherently unordered. We will fix that below.

The next step is to join the two sets of data into one, in order to make a modified describe output that includes skew and kurtosis. The same method could be used to include any other aggregations desired.

In [8]:

new\_describe = sc.parallelize(new\_data).toDF() *#turns the results from our loop into a dataframe*

new\_describe = new\_describe.select(df\_described.columns) *#forces the columns into the same order*

expanded\_describe = df\_described.unionAll(new\_describe) *#merges the new stats with the original describe*

expanded\_describe.show()

+--------+--------------------+--------------------+-------------------+------------------+------------------+-----------------+

| summary| \_c0| \_c2| \_c3| \_c4| \_c5| \_c6|

+--------+--------------------+--------------------+-------------------+------------------+------------------+-----------------+

| count| 3526154| 382039| 3526154| 1580402| 3526154| 3526154|

| mean|5.503885995001908E11| null| 4.178168090219519|234846.78065481762| 5.134865351881966|354.7084951479714|

| stddev|2.596112361975214...| null|0.34382335723646673|118170.68592261661|3.3833930336063465| 4.01181251079202|

| min| 100002091588| CITIMORTGAGE, INC.| 2.75| 0.85| -1| 292|

| max| 999995696635|WELLS FARGO BANK,...| 6.125| 1193544.39| 34| 480|

| skew| -0.00183847089866| None| 0.519799339496| 0.758411576756| 0.286480156084| -2.69765201567|

|kurtosis| -1.19900726351| None| 0.126057726847| 0.576085602656| 0.195187780089| 24.7237858944|

+--------+--------------------+--------------------+-------------------+------------------+------------------+-----------------+

And now we have our expanded describe output.

**QUESTION 6**

Dealing with very large data forces us to divide options for plotting into two categories: those that involve an aggregation step first, and those that don't.

There are many ways to **aggregate data for plots** (e.g. hexbins, box plots, bar graphs), and when your data is big one beneficial side effect is that aggregating reduces the size. If the aggregation is enough to allow your data to be loaded into memory then there is no problem; you can use whatever plotting tools you like. You can download your data to an S3 bucket, or locally to your computer, and make your plots that way. You can also use the tools Python provides for plotting, which we will go into here. Even if you're more comfortable with a different program (e.g. Excel, Stata, SAS, Gnuplot, Matlab), you may want to continue in Python rather than downloading since it allows you to generate the graphs in the same platform you use to work with the larger dataset.

**Plotting big data without aggregating**, for example in a bivariate scatter plot, gets difficult as your data gets larger. There are two possible solutions: first, it may be possible to use Amazon Web Services to spin up a single machine with a large amount of memory to work on, for the sole purpose of creating the graph. The machine itself is likely to be expensive to access, but that may be offset by only needing it very briefly. There is not currently a system in place for this to happen. The second option is still untested in Spark, and that is to use the Bokeh DataShader module for Python. Whether this works in a distributed environment will be the subject of future work.

In [ ]:

**import** **pandas** **as** **pd**

**import** **matplotlib.pyplot** **as** **plt**

%**matplotlib** inline #tells the Jupyter Notebook to display graphs inline (rather than in a separate window)

In [ ]:

*# Loading and Viewing the Data*

For this example we will use a dataset on diamond attributes **and** prices. This **is** the same data used **in** the ggplot tutorial **for** plotting **in** R. After we load it we'll take a quick look at what it contains.

In [ ]:

spark\_df = spark.read.csv('E:\SparkWork\per-vehicle-records-2021-01-31.csv, inferSchema=True, header=True, sep=',')

In [ ]:

spark\_df.show(10)

In [ ]:

spark\_df.dtypes

In [ ]:

spark\_df.describe(['lane', 'headway', 'gap', 'temperature']).show()

If necessary we could first subset the data to make it small enough to handle with standard tools, for example by dropping certain columns, taking a random sampling, and so on (see the subsetting tutorial).

**Using Matplotlib**

There are many graphing solutions in Python, but the most common one by far is Matplotlib. It is very flexible; literally every aspect of a graph can be manipulated and fine-tuned if desired. **This can make it a great tool, but that same flexibility often leads to complex and verbose code.** The Python plotting environment is not as well-developed as in R, for example, with ggplot.

We will start with a very simple example that uses only a bare minimum of the things Matplotlib can do, and let it otherwise fill in default settings for us. Remember, in order to graph in Matplotlib our data cannot be distributed; it must be small enough to be collected. Therefore, we begin by extracting the values of the two columns we want to plot directly from the Spark dataframe:

In [ ]:

carat = spark\_df[['carat']].collect()

price = spark\_df[['price']].collect()

In [ ]:

Using `collect` has technically loaded into memory two lists of row objects that only have one entry each (`carat` **in** one **and** `price` **in** the other). We could convert these to plain Python lists **if** we wanted, but Matplotlib knows how to work **with** these, so there's no need. Take a look at the first five entries of each, to get an idea:

In [ ]:

print(carat[:5])

print(price[:5])

**QUESTION 5**

Matplotlib leverages the object-oriented nature of Python by creating objects that we then manipulate properties for. To start, we first define a **figure object**, named fig, then create one or more **subplots** inside of the figure by defining an **axis object**, named ax, that points to the single subplot in our figure.

Each figure can have one or more subplots within it, arranged according to the fig.add\_subplot(a,b,c) line below. Here, 'a' defines how many rows (of plots) your figure has, 'b' defines how many columns, and 'c' defines which plot within the figure the axis object points to. By passing '1,1,1' as we do below, we tell it the figure has 1 row, 1 column, and that we want to work with the 1st (and in this case only) subplot out of those:

In [ ]:

fig = plt.figure() *#an empty figure*

ax = fig.add\_subplot(1,1,1) *#a single subplot within the figure*

Which as you can see is currently an empty figure, because we haven't given it any data or settings. Note that each time a figure is displayed or saved, Matplotlib clears the queue and we need to define it again.

If we wanted, say, four different plots to show up, one in each corner of the figure, we would create four subplots, each with its own axis pointer, and tell it to have 2 rows, 2 columns, and then number the plots 1-4. We could then set attributes on the entire figure by modifying properties of fig, or on each individual plot by modifying attributes on ax1, ax2, ax3, or ax4:

In [ ]:

fig = plt.figure()

ax1 = fig.add\_subplot(2,2,1) *#a 2x2 grid of plots, plot 1*

ax2 = fig.add\_subplot(2,2,2) *#plot 2*

ax3 = fig.add\_subplot(2,2,3) *#plot 3*

ax4 = fig.add\_subplot(2,2,4) *#plot 4*

fig.suptitle('Title for the Figure')

ax1.set\_title('Title for the First Subplot')

In [ ]:

fig = plt.figure()

ax = fig.add\_subplot(1,1,1)

ax.plot(carat, price, 'go', alpha=0.1)

ax.set\_xlabel('Cosit')

ax.set\_ylabel('speed')

ax.set\_title('headway')

**Using Pandas**

Plotting in Python can be simplified if you first turn your Spark dataframe into a **Pandas** dataframe. In addition to Pandas convenient plotting tools, this also gives you access to many other dataframe manipulations similar to the R language. Note once again that you can only use Panads on datasets small enough to fit on one system; like native R dataframes, they do not support a distributed environment. Therefore you may need to subset a large distributed Spark dataframe before it can be loaded and worked with this way. As with Matplotlib, Pandas can easily handle our diamonds dataset however.

**R, Pandas, pySpark and SparkR all share a lineage**, and while there are differences that can trip you up when going between them there are also a lot of similarities that will ease transitions back and forth.

**Spark dataframes have a built-in method to turn themselves into Pandas dataframes:**

In [ ]:

pandas\_df = spark\_df.toPandas()

pandas\_df.describe()

In [ ]:

*#in order from greatest clarity to least:*

M50\_order = ['FL', 'IF', 'VVS1', 'VVS2', 'VS1', 'VS2', 'SI1', 'SI2', 'I1', 'I2', 'I3']

mapping = {day: i **for** i, day **in** enumerate(clarity\_order)}

key = grouped['M50'].map(mapping)

grouped = grouped.iloc[key.argsort()]

In [ ]:

grouped.plot(kind='bar', x='M50', legend=**False**)

**Pandas and Matplotlib Together**

If the default formatting of the Pandas graph looks a lot like the Matplotlib graph we created earlier, it should; Pandas plotting is built on Matplotlib. It just handles a lot of the more verbose parts of Matplotlib in the background for us.

In fact we can even use them together, if we want the convenience of Pandas automatic plotting with the ability to alter every last detail that Matplotlib provides. First we use the Pandas plot method to *assign an axis object* (ax), rather than calling it without assigning it to anything as we did above.

Pandas will then handle all the default formatting of the axis for us, allowing us to override whatever features we want afterword. In this case we set the labels and titles; you can see it's otherwise the same graph as above.

In [ ]:

*#same line as in the previous code block, only beginning with "ax ="*

ax = grouped.plot(kind='bar', x='clarity', legend=**False**)

*#then we set some options using Matplotlib commands*

ax.set\_xlabel('Clarity')

ax.set\_ylabel('speed')

ax.set\_title('lane')

**Misc**

Jupyter notebooks automatically display our figure for us when a codeblock that creates one is executed; this is a convenience that is due to the %matplotlib inline "magic" function that we called at the top of the notebook. Normally after our figure is ready we must call plt.show() to see the graph.

Along these lines, if we want to save a figure we can call plt.savefig(<path to an S3 bucket>) once it's created. It is also possible to right-click on an image once Jupyter displays it and save it locally to your computer, though if you're creating many graphs, for example inside of a loop, this may not be practical.

A few references for making your Matplotlib graphs even nicer:

**Text on Matplotlib graphs:** <http://matplotlib.org/users/text_intro.html>