James Kasakyan
Tuesday, May 17, 2016
CSC 59927 Big Data Management and Analysis

Final Project Report

All source code and instructions for running available at
https://github.com/JKasakyan/BDM-Final-Project

Table of Contents

Revisions to Data sources

Name	Years	Provider	Link
NYPD Motor Vehicle Collisions	2012-2013	NYC OpenData	https://data.cityofnewyork.us/Public-Safety/NYPD-Motor-Vehicle-Collisions/h9gi-nx95
Traffic Volume Counts	2012-2013	NYC OpenData	https://data.cityofnewyork.us/Social-Services/311-Service-Requests/fvrb-kbbt
~~**Vehicle Classification Counts**~~	~~2012-2013~~	~~NYC OpenData~~	~~https://datahub.cusp.nyu.edu/dataset/ae5u-upr6~~
311 Service Requests	2010-Present	NYC OpenData	https://data.cityofnewyork.us/NYC-BigApps/Traffic-Volume-Counts-2012-2013-/p424-amsu
Zip Volume Counts	2012-2013	Self generated	https://github.com/JKasakyan/BDM-Final-Project/blob/master/Datasets/zip_veh_count.csv

Explanation:

In the project proposal, the plan was to include the Vehicle Classification Counts dataset. The idea was to normalize not only by traffic volume, but also by the type of traffic (taxi, bus, etc.). This way if a zip code had a large number of taxis involved in accidents, for example, it could be determined if this was simply because a majority of the traffic in that area was taxi traffic, or if it signified that taxis were more prone to accidents in this area than other types of vehicles. However, because of time constraints, this dataset was not used in the final project.

Additionally, another dataset was needed that contained, for each zip code, the traffic volume and number of samples that produced that traffic volume. This dataset was generated from the Traffic Volume Counts dataset, and can be reproduced using the generate_vehicle_count_csv.py script (for convenience, it is provided in the repo at Datasets/zip_veh_count.csv)

Revisions to methodologies

Explanation:

In the project proposal, the hypothesized method for classification of zip codes was to use Spark's Machine Learning library (MLlib). More specifically, the idea was to use the MLlib decision tree classifier to classify areas as "safe", "ok", or "hazardous". However due to the small sample size of the data (144 zip codes), and due to the fact that a classifying model built using this library would likely not see much re-use, a different method for classification was used.

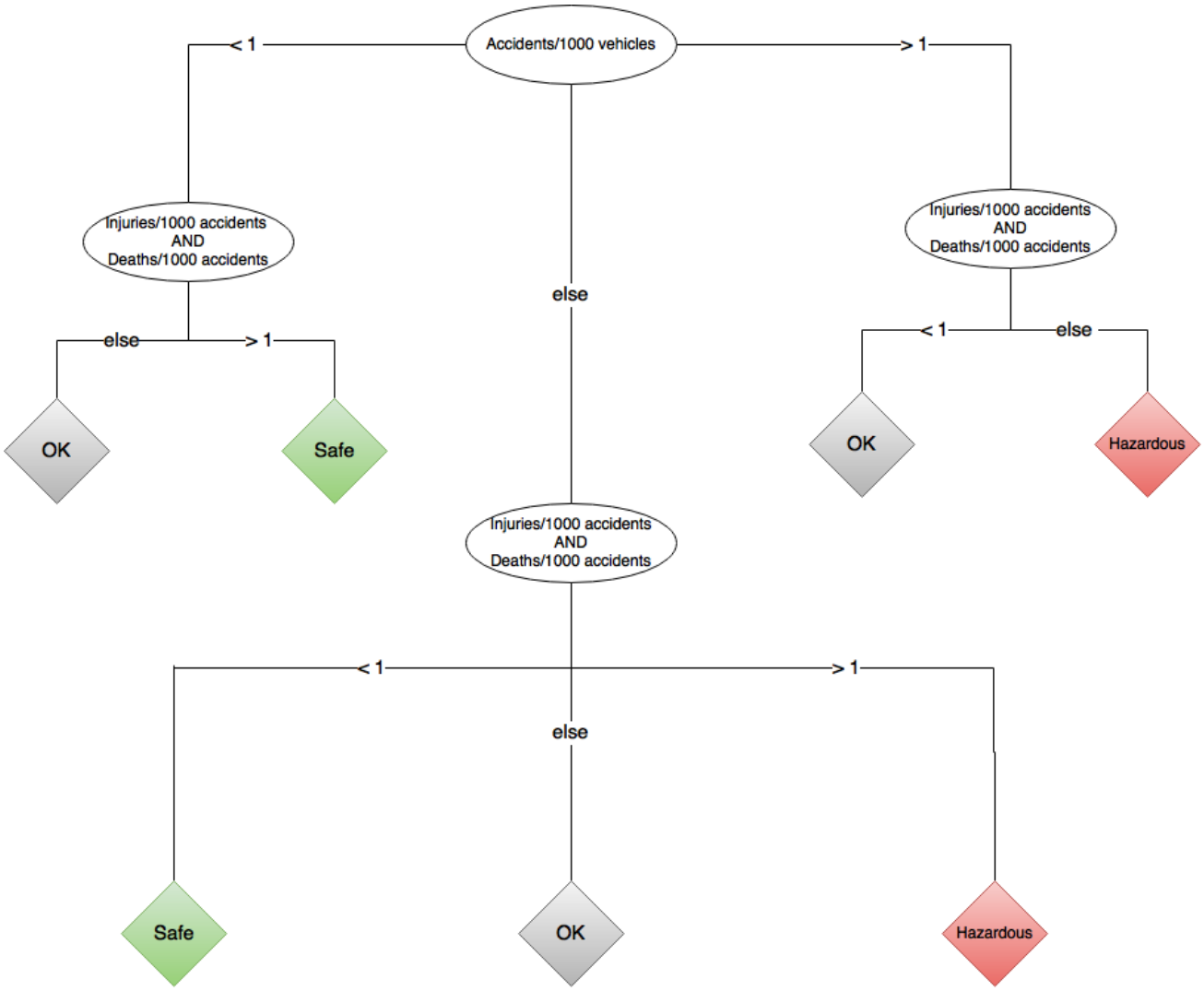
This involved computing the average and standard deviation in three "feature" categories for each zip code:

1. Accidents/1000 vehicles
2. Injuries/1000 accidents
3. Deaths/1000 accidents

Assuming a normal distribution in each of these feature categories, zip codes were classified according to their distance from the mean. The general idea was that those areas where multiple features had values greater than one standard deviation from the mean were the most hazardous.

The most heavily weighted category was accidents/1000 vehicles. This decision was made because it seemed reasonable that the most obvious feature of an area that is hazardous for motor vehicle traffic is that it is prone to motor vehicle accidents. Features like injuries/1000 accidents and deaths/1000 accidents, while still noteworthy, are not the determining factors for how dangerous an area is for vehicles. However they can provide insight into the safety level of areas that do not fall on either side of the extreme in terms of accidents/1000 vehicles.

Decision tree:



Overview of Source Code

Runnable scripts (/Code/Scripts) :

Name	Arg(s)	Output(s)
main.py	1. path/to/NYPD_Motor_Vehicle_Collsions.csv 2. path/to/311_Service_Requests_from_2010_to_Present.csv 3. path/to/output_from_script_generate_vehicle_count_csv 4. download*	Results and totals in directory FinalProjectOutputs
generate_vehicle_count_csv.py	1. path/to/Traffic_Volume_Counts__2012-2013.csv 2. output/path/	CSV with zip code traffic count, and number of samples saved in output/path

Auxiliary scripts (/Code/Scripts):

Name	Description
police_reports.py	Uses the NYPD collisions dataset to filter records for 2012-2013 and extrac data for each zip code like the total number of accidents, total persons injured, and top accident factors. police_reports.get_rdd returns an RDD with this data.
three_one_one.py	Uses the 311 dataset to filter records for 2012-2013 and extract data for each zip code like total number of complaints, top street related complaints, and top street related complaint descriptors. three_one_one.get_rdd returns an RDD with this data.
vehicle_volume_count.py	Uses the dataset produced by generate_vehicle_count._csv.py (/Datasets/zip_veh_count.csv) to normalize values for 2012-2013 based on traffic volume and number of samples. vehicle_volume_count.get_rdd returns an RDD with this data.

Walkthrough of Source Code

See the readme on https://github.com/JKasakyan/BDM-Final-Project for full instructions on running the code.

Police Reports

```
def get_rdd(sc, accident_csv, download=False, output_path=None)

Inputs:
sc – SparkContext
accident_csv – path to NYPD_Motor_Vehicle_Collisions.csv
download – Boolean indicating whether the output should be saved
in output_path directory. Default is False.
```

```
output_path – path to directory where outputs will be saved

Returns:
RDD with zip code as key, and relevant accident data for that zip
code in 2012-2013 (total accidents, total vehicles, top factors,
etc.)
```

This method begins by reading the NYPD Motor Vehicle Collisions dataset into a RDD:

```
accident_rdd = sc.textFile(accident_csv, use_unicode=False)
```

The first transformation is:

```
accident_zip_rdd = accident_rdd.mapPartitions(my_format)
      def my_format(records)
```

This transformation parses each accident record and extracts relevant fields like zip code and number of persons injured for records between 2012-2013, with zip code as the key.

The next transformation is:

```
result_rdd = accident_zip_rdd.aggregateByKey({}, seqFunc,
combFunc)
      def seqFunc(agg_dict, record)
      def combFunc(dict1, dict2)
```

This transformation takes data for each zip code, and uses a dictionary to aggregate various statistics for that zip code. These include the total number of accidents, total number of vehicles involved in accidents, and total number of persons injured. At the end of this transformation, each zip code has a dictionary containing those relevant statistics.

The next transformation is:

```
aggregated_dict = result_rdd.map(lambda x: x[1].fold({}, combine)
      def combine(agg_dict, record_dict)
```

This transformation merges all the dictionaries for each zip code into one dictionary containing the metadata for the entire set. It is later saved at FinalProjectOutputs/totals/NYPD_accident_2012_2013.

The next transformation is:

```
zip_statistics_rdd = result_rdd.mapPartitions(topn)
      def topn(zip_tuples)
```

This transformation takes the dictionary associated with each zip code, and runs two heapq.nlargest() operations to determine the top five accident factors and top five vehicle types involved in accidents for each zip code. In the case that a particular zip code had fewer than five unique accident factors or vehicle types involved in accidents, the top three are computed. If there are fewer than three types or factors, the top one is computed.

If the download flag is True, results are saved at FinalProjectOutputs/results/zip_NYPD_accident_report_2012_2013 and totals/NYPD_accident_2012_2013

Finally, the last RDD is returned to end the get_rdd() method of police_reports.py:

```
return zip_statistics_rdd
```

311

```
def get_rdd(sc, three_one_one_csv, download=False, output_path=No
ne)

Inputs:
sc – SparkContext
three_one_one_csv – path to
311_Service_Requests_from_2010_to_Present.csv
download – Boolean indicating whether the output should be saved
in output_path directory. Default is False.
output_path – path to directory where outputs will be saved

Returns:
RDD with zip code as key, and relevant street complaint data for
that zip code in 2012-2013 (total complaints, top complaint type,
top complaint descriptor, etc.)
```

This method begins by reading the data from the 311 dataset into an RDD:

```
rdd = sc.textFile(three_one_one_csv)
```

The first transformation is:

```
filtered_rdd = rdd.mapPartitions(select_fields)
```

This transformation filters the dataset for street related complaints in 2012-2013, and disregards any records that do not have a zip code or city associated with them. It yields a record with the zip code as the key, and relevant complaint data as the values.

The next transformation is:

```
zip_complaints_rdd = filtered_rdd.aggregateByKey({}, seqOp, combO
p)
      def seqOp(agg_dict, record)
      def combOp (dict1, dict2)
```

This transformation aggregates the street complaint related data for each zip code in a dictionary. Keys in this dictionary include the total number of complaints, and a count for each complaint type and description.

The next transformation is:

```
aggregate_dict = zip_complaints_rdd.map(lambda x: x[1]).fold({},
combine)
```

```
    def combine(agg_dict, record_dict)
```

This transformation combines all the dictionaries into one dictionary containing metadata for the dataset.

The next transformation is:

```
three_one_one_statistics_rdd = zip_complaints_rdd.mapPartitions(t
opn)
    def topn(zip_tuples)
```

This transformation takes the dictionary associated with each zip code, and computes the top 3 complaint types and top 5 complaint descriptions for that zip code. If there are fewer than three complaint types, it computes the top one. If there are fewer than five complaint descriptions, it computes the top three. If there are fewer than three, it computes the top one.

If the download flag is True, results are saved at FinalProjectOutputs/results/zip_three_one_one_2012_2013 and totals/three_one_one_2012_2013

Finally, the last RDD is returned to end the get_rdd() method of three_one_one.py:

```
return three_one_one_statistics_rdd
```

Vehicle Volume Count

```
def get_rdd(sc, count_csv_path, download=False, output_path=None)

Inputs:
sc – SparkContext
count_csv_path – path to zip_veh_count.csv (or location of output
of generate_vehicle_count_csv.py script)
download – Boolean indicating whether the output should be saved
in output_path directory. Default is False.
output_path – path to directory where outputs will be saved

Returns:
RDD with zip code as key, and normalized traffic volume for 2012-
2013 as value
```

The method first reads from the zip_veh_count into a RDD:

```
zip_num_vehicles_rdd = sc.textFile(count_csv_path)
```

The first transformation is:

```
zip_key_rdd = zip_num_vehicles_rdd.mapPartitions(map_multiple_zip)
    def map_multiple_zip(records)
```

This transformation deals with edge cases in the input file, where some records have two zip codes associated with them. When it finds these records it splits the vehicle count and

samples between the two zip codes in the record. It yields a record with a single zip code as the key, and a vehicle count and number of samples that produced that count as values.

The next transformation is:

```
result_rdd = zip_key_rdd.aggregateByKey({}, seqOp, combOp).mapPar
titions(dict_mapper)
      def seqOp(dict1, tup)
      def combOp(dict1, dict2)
      def dict_mapper(zip_tuples)
```

This transformation produces a dictionary for each zip code, which has aggregations for the vehicle count and number of samples. Then based on that information, the final normalized result is computed with a key equal to the zip code, and a lone value equal to the normalized traffic volume for 2012-2013.

If the download flag is True, results are saved at FinalProjectOutputs/results/zip_traffic_volume_2012_2013

Finally, the last RDD is returned to end the get_rdd() method of vehicle_volume_count.py:

```
return result_rdd
```

Main

```
def main(sc, sqlContext, path_to_accident_csv, path_to_three_one_
one_csv, path_to_count_csv, download=False)

Inputs:
sc – SparkContext
sqlContext- SQLContext
path_to_accident_csv – path to NYPD_Motor_Vehicle_Collisions.csv
path_to_three_one_one_csv – path to 311_Service_Requests_from
2010_to_Present.csv
count_csv_path – path to zip_veh_count.csv (or location of output
of generate_vehicle_count_csv.py script)
download – Boolean indicating whether the output should be saved.
Default is False.

Returns:
None. If download is True, will call get_rdd methods of previous
three scripts with their respective download flags set to True.
Will additionally save its outputs and aggregations.
```

The method begins by calling the get_rdd methods of each of the previous three scripts:

```
accident_rdd = get_accident_rdd(sc, path_to_accident_csv, downloa
d=download, output_path="FinalProjectOutputs")
```

```
three_one_one_rdd = get_three_one_one_rdd(sc, path_to_three_one_o
ne_csv, download=download, output_path="FinalProjectOutputs")
count_rdd = get_vehicle_rdd(sc, path_to_count_csv, download=downl
oad, output_path="FinalProjectOutputs")
```

And creates a single RDD by joining them:

```
joined_rdd= count_rdd.join(accident_rdd.join(three_one_one_rdd))
```

The first transformation is:

```
full_rdd = joined_rdd.mapPartitions(mapper)
     def mapper(tuples)
```

This transformation uses the normalized traffic volume for each zip code to compute classification features like accidents/1000 vehicles.

The next transformation is:

```
full_df = sqlContext.createDataFrame(full_rdd, schema)
```

This transformation creates a DataFrame for the RDD that has the classification features.

This is to make it easier to compute the mean and standard deviation for each feature by performing a single call on the DataFrame:

```
full_df.describe(['Accidents_Per_1000_Vehicles', 'Injuries_Per_10
00_Accidents', 'Deaths_Per_1000_Accidents', 'Vehicles_Involved_Pe
r_Accident'])
```

The top 10 and bottom 10 zip codes for each feature can also be computed with relative ease using the DataFrame:

```
top_10_accidents_per_1000_vehicles =
map(lambda x: x.asDict(), full_df.sort('Accidents_Per_1000_Vehicl
es', ascending=False).take(10))
top_10_injuries_per_1000_accidents =
 map(lambda x: x.asDict(), full_df.sort('Injuries_Per_1000_Accide
nts', ascending=False).take(10))
```

Finally, classification of zip codes can occur:

```
final_rdd = full_rdd.mapPartitions(classify_mapper)
      def classify_mapper(records)
```

A breakdown of the classification algorithm used can be found in section 2: Revisions to Methodologies.

The method ends by saving various results to the FinalProjectOutputs directory.

Project Deliverables

See Zip Code Motor Vehicle Safety Map