PROJECT REPORT

Under The School of AI's, Summer Fellowship Program, I got an amazing opportunity to work on an Independent AI/ML Project in the Summer'19. We were given GCP credits and a mentor to carry out a project from scratch. In this article, I have written a detailed account of the project I carried out, and the tools and services I used alongside.

Project Summary

I wanted to make it a complete Data-Centric Project, hence the datasets I choose had millions of data points. Any kind of ML or DL Algorithm must have good amount of data to train on but applying the same Algorithms to Big Data isn't as simple as running the same models locally. We have to deal with a lot more complexities and preprocessing the data in itself is a big task. Leveraging the Big Data Tools on GCP like BigQuery, Dataproc Clusters, AI Notebooks, and Dataflow Runners, made my work a lot easier. I was able to run various kinds of data-intensive Algorithms by carefully taking advantage of the GCP services as and when needed.

Project Idea and Datasets

Exploring Algorithms for Traveling Salesman Problem

It is one of the most famous Algorithms having several applications such as planning, scheduling, and other complex optimization problems. In general, the problem can be stated as given n cities, we need to find the shortest route that passes through each city. As an NP-complete Problem, it is possible that the worst case running time for any algorithm for the TSP increases super-polynomially or exponentially with the number of cities.[1]

The idea was to use AI Algorithms to solve TSP. The Algorithms are discussed in the next section. The data used was NYC's city bike Trips dataset available on BigQuery. The Location [(latitude, longitude) pairs] is considered as nodes and the distance is calculated using special metrics which makes up the Edge of the graph. Analytical Results show that Pershing Square, Broadway Station and Chambers stations seem to be the busiest and the most popular Stations for the Bike Trips.

Used Algorithms

1. **Nearest Neighbors**

- One of the most basic Algorithms for the Traveling Salesman Problem is the Nearest Neighbor Algorithm. It's a Brute Force Technique.
- *Basic Algorithm:*
    1. Make two sets of nodes, A and B, and put all nodes into set B.
    2. Put your starting node into set A.
    3. Pick the node which is closest to the last node which was placed in set A and is not in set A; put this closest neighboring node into set A.
    4. Repeat step 3 until all nodes are in set A and B is empty.

2. **Simulated Annealing**

- In the Traveling Salesman Problem, each state is typically defined as a permutation of the cities to be visited, and its neighbors are the set of permutations produced by reversing the order of any two successive cities.[3]

- The well-defined way in which the states are altered to produce neighboring states is called a "move", and different moves give a different set of neighboring states. These moves usually result in minimal alterations of the last state, in an attempt to progressively improve the solution through iteratively improving its parts.[3]
- *Basic Algorithm:*[4]
    1. First, an initial Temperature needs to be set, and create a random initial solution.
    2. Then, begin looping until our solution is met. Usually either the system has sufficiently cooled down, or a good-enough solution has been found.
    3. From here we select a neighbor by making a small change to our current state.
    4. We then decide whether to move to that neighbor solution.
    5. Finally, we decrease the temperature and continue looping.
- SA gives an approximate global optimum solution. The main function of SA is to go from one State (One possible shortest path) to another State until a global optimal solution is obtained. Refer to Wikipedia for a detailed Algorithm description.

3. Self-Organizing Maps

- SOM is a grid of nodes. Closely related to the idea of a model, that is, the real-world observation the map is trying to represent. They use Neighborhood-based techniques to preserve the topological properties of the input space.
- The purpose is to represent the model with a lower number of dimensions while maintaining the relations of similarity of the nodes contained in it. More similar to the nodes, more spatially closer they are organized. Hence, it makes SOM good for pattern visualization and organization of data.
- *Basic Algorithm:*[5]
    1. Each node's weights are initialized.
    2. A vector is chosen at random from the set of training data.
    3. Every node is examined to calculate which one's weight is most like the input vector. The winning node is commonly known as the BEST Matching Unit (BMU)
    4. Then the neighborhood of the BMU is calculated. The number of neighbors decreases over time.
    5. The winning Node is rewarded with becoming more like the sample vector. The neighbors also become more like the sample vector. The closer a node is to the BMU, the more its weight gets altered and the farther away the neighbor is from the BMU the less it learns.
    6. Repeat step 2 for N Iterations.
- Best Matching Unit is a technique which calculates the distance from each weight to the sample vector, by running through all weight vectors. The weight with the shortest distance is the winner.[5]

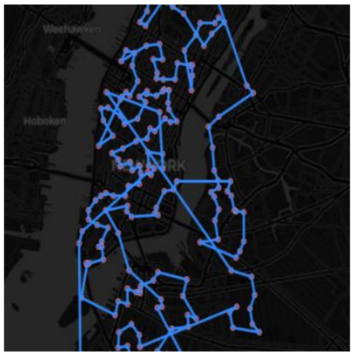
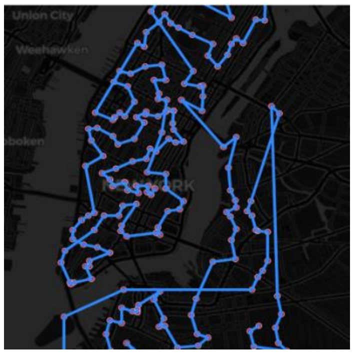
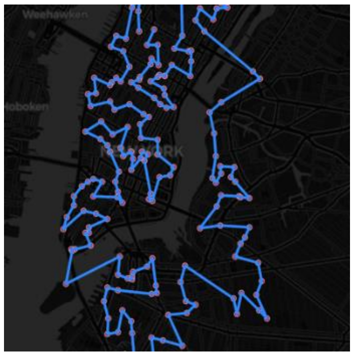
4. Ant Colony Optimization

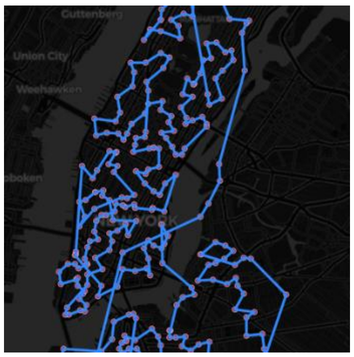
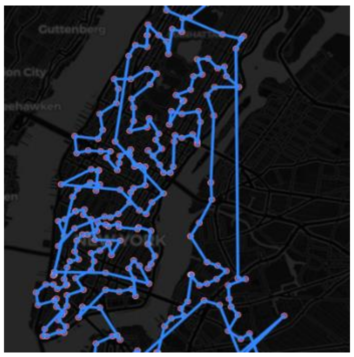
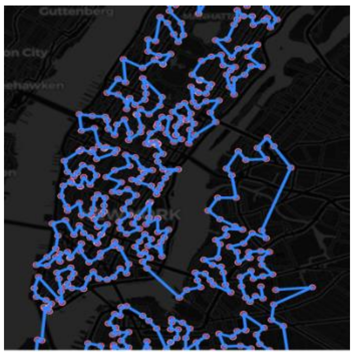
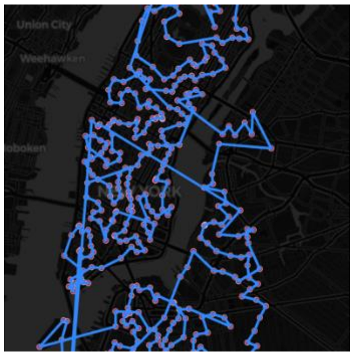
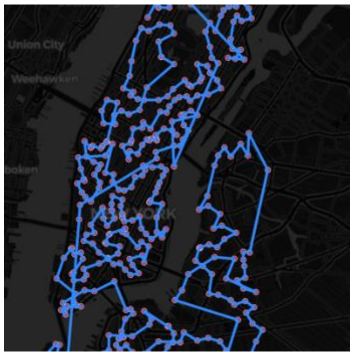
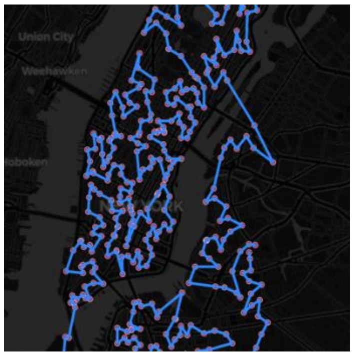
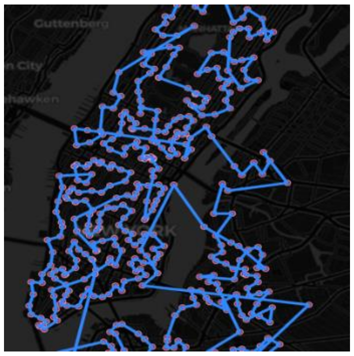
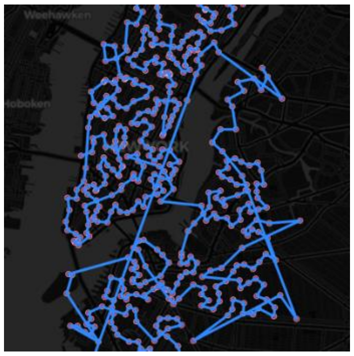
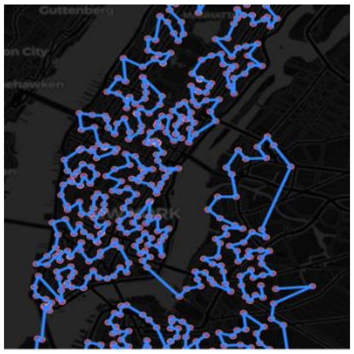
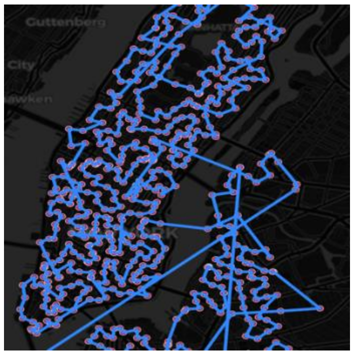
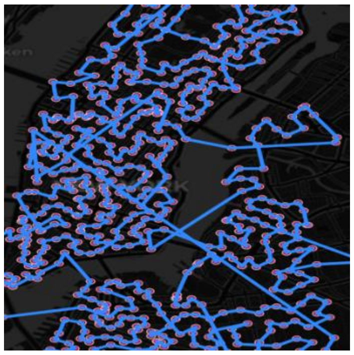
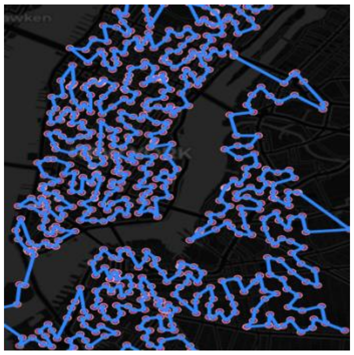
- In ACS, a set of cooperating agents cooperate to find good solutions to TSPs.
- According to the ACS paper, [6] ACS outperforms other nature-inspired Algorithms such as simulated annealing and evolutionary computation.
- *Basic Algorithm:*
    1. m ants are initially positioned on n cities chosen according to some initialization rule (random).
    2. Each ant builds a tour by repeatedly applying a stochastic greedy rule. While constructing its tour, an ant also modifies the amount of pheromone on the visited edges by applying the local updating rule.
    3. Once all ants have terminated the tour, the amount of pheromone on edges is modified again.
    4. An edge with a higher amount of pheromone is the desired choice.
- Though Ant System was useful for discovering good or optimal solutions for small TSPs, the time required to find such results made it infeasible for larger problems.

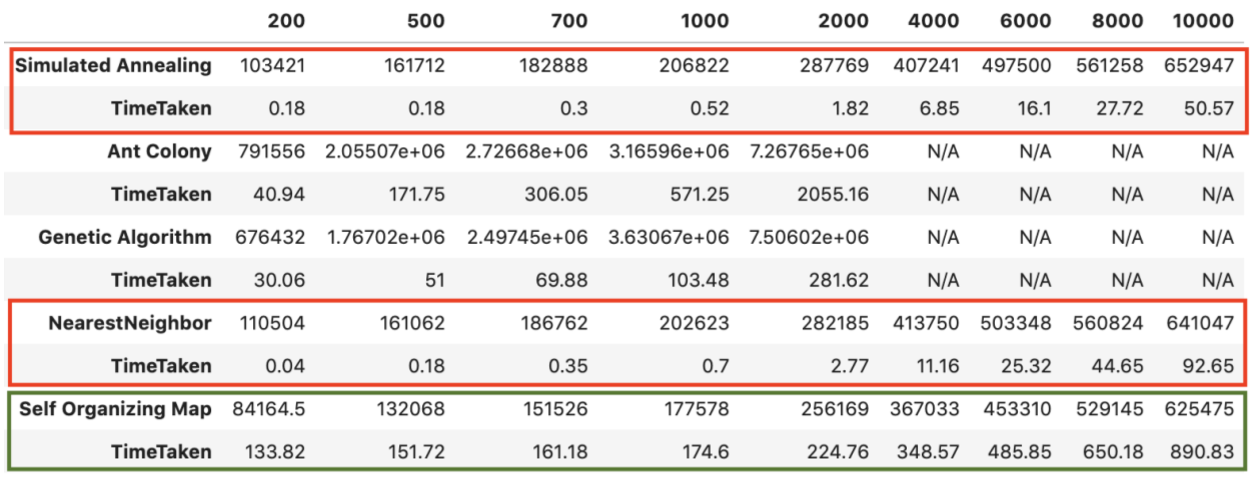
5. Genetic Algorithm

- A GA is a search heuristic that is inspired by the theory of Natural Evolution. This Algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation [7]
- The process of Natural Selection starts with the selection of the fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance of surviving. This process keeps on iterating and in the end, a generation with the fittest individuals will be found.[7]
- *Basic Algorithm:*[7]
  1. Generate Initial Population - Each individual element in the population is a solution to the problem you want to solve i.e. shortest path.
  2. Compute Fitness - How efficient the individual solution is in comparison to other paths
  3. Selection - Select fittest individuals (paths) and let them pass their genes to the next generation.
  4. Crossover - Offspring are created by exchanging the genes of the fittest individuals and exchanging the genes of parents among themselves until the crossover point is reached. The new offspring are added to the population.
  5. Mutation - Radom flipping of bits in the bit string with a low random probability.
  6. Compute Fitness - Compute fitness for the newly generated offspring after mutation.
  7. Repeat (3) - (6) Steps until Population has converged.

Results of TSP Algorithms

#	Nearest Neighbor	Simulated Annealing	Self-Organizing Maps
200			

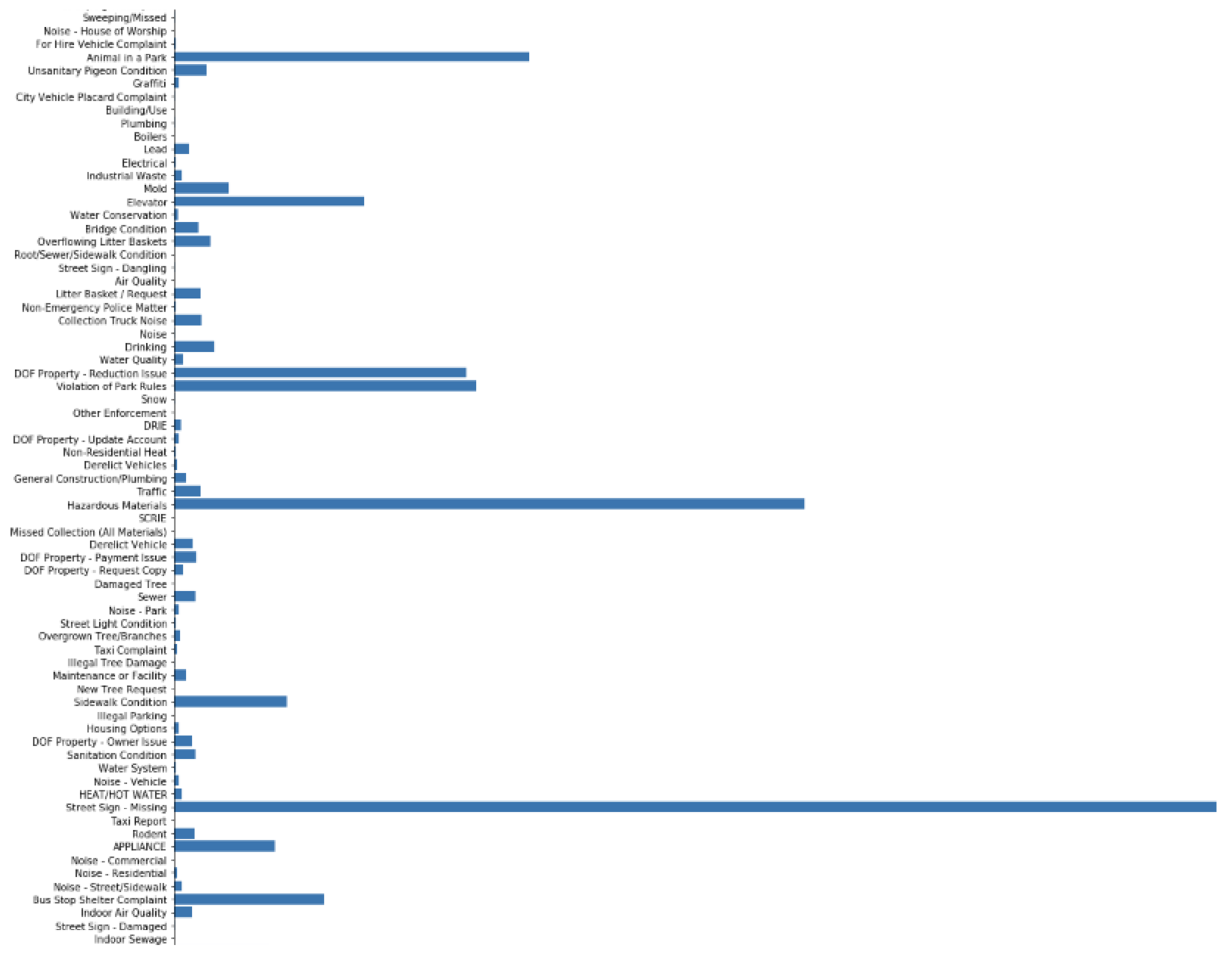
300			
400			
500			
698			



	200	500	700	1000	2000	4000	6000	8000	10000
Simulated Annealing	103421	161712	182888	206822	287769	407241	497500	561258	652947
TimeTaken	0.18	0.18	0.3	0.52	1.82	6.85	16.1	27.72	50.57
Ant Colony	791556	2.05507e+06	2.72668e+06	3.16596e+06	7.26765e+06	N/A	N/A	N/A	N/A
TimeTaken	40.94	171.75	306.05	571.25	2055.16	N/A	N/A	N/A	N/A
Genetic Algorithm	676432	1.76702e+06	2.49745e+06	3.63067e+06	7.50602e+06	N/A	N/A	N/A	N/A
TimeTaken	30.06	51	69.88	103.48	281.62	N/A	N/A	N/A	N/A
NearestNeighbor	110504	161062	186762	202623	282185	413750	503348	560824	641047
TimeTaken	0.04	0.18	0.35	0.7	2.77	11.16	25.32	44.65	92.65
Self Organizing Map	84164.5	132068	151526	177578	256169	367033	453310	529145	625475
TimeTaken	133.82	151.72	161.18	174.6	224.76	348.57	485.85	650.18	890.83

[Figure 1] The columns present the number of city stops. As seen the paths created by Self-Organizing Maps are shorter than the other two Algorithms, which is also evident in the figures. But the Time Taken by SOMs is much more than the other two making it infeasible for large datasets.

<u>Predicting the completion time of Service Requests</u>



[Figure 2] The above plot shows number a bar plot for different service Requests. As seen in the bar plot, only some of the requests are frequently made. This frequency distribution was considered by binning categorical variables.

New York City has a special 311 Service request number, where people can call up to put up a complaint about some particular issue. With 50+ Features, Feature selection process was in itself a major task. Analyzed the correlations and dependencies of various available features and finally selected suitable features for the Model building process.

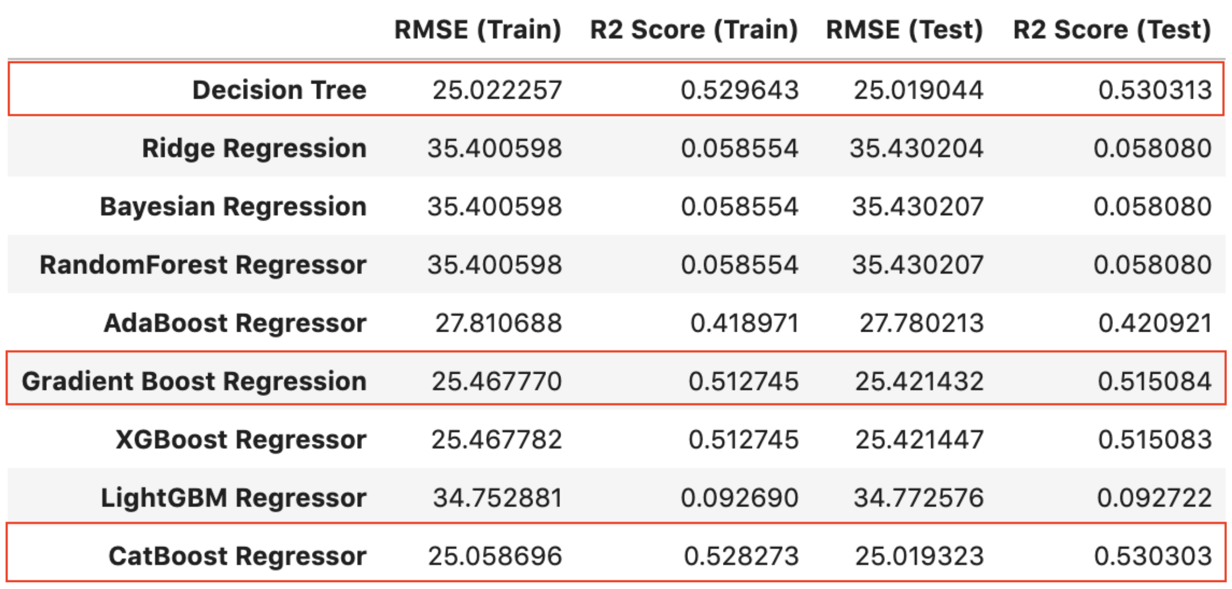
The NYC city bike trips dataset is available on BigQuery. I used Apache Beam scripts for preprocessing the 33M+ rows and ran the Models on cloud ML. Since, the data is huge, it is divided into 10 files and stored in GCS Bucket.

Exploratory Data Analysis and Data Engineering

- Examining Requests per region.
- Understanding the type of incidents occurring per Region.
- After exploring the number of complaints per complaint type, only ones with at least 400 requests was considered for further analysis.
- Understanding which agency is responsible for which type of Issue.
- Removed the extra/un-correlated features from the data.
- Created new Date and Time Features (day_of_week & time_of_day) having high influence on the requests.
- Understanding the feature category across all 10 data files, and deciding on the bins for the categorical variables. (incident_zip, location_type, agency, complaint_type, community_board)
- Once decided, the categorical data across all the sharded files was converted to have binned features for the data. Apache Beam script was written for preprocessing, which ran on the Dataflow runner.
- Target -> Completion_time = Closed_timestamp - Created_timestamp

Model Building

- Built a local Tensorflow Regression model to test the performance of the model. Even though I passed on just one file for training the model (~200MB), it took forever to run locally. Hence, running it on the cloud was the only solution.
- Made a Model pipeline for data preprocessing. Developed, trained and deployed a Tensorflow model on GCP. The Regression model didn’t perform so well.
- Created another Tree-based Models for the same problem. Tested various tree model performance with and without dimensionality Reduction. Selected the Decision Tree model based on RMSE and R2 scores for Train/Test splits.
- Used the same pipelines as before, but now converted the Tensorflow pipeline to scikit-learn.



	RMSE (Train)	R2 Score (Train)	RMSE (Test)	R2 Score (Test)
Decision Tree	25.022257	0.529643	25.019044	0.530313
Ridge Regression	35.400598	0.058554	35.430204	0.058080
Bayesian Regression	35.400598	0.058554	35.430207	0.058080
RandomForest Regressor	35.400598	0.058554	35.430207	0.058080
AdaBoost Regressor	27.810688	0.418971	27.780213	0.420921
Gradient Boost Regression	25.467770	0.512745	25.421432	0.515084
XGBoost Regressor	25.467782	0.512745	25.421447	0.515083
LightGBM Regressor	34.752881	0.092690	34.772576	0.092722
CatBoost Regressor	25.058696	0.528273	25.019323	0.530303

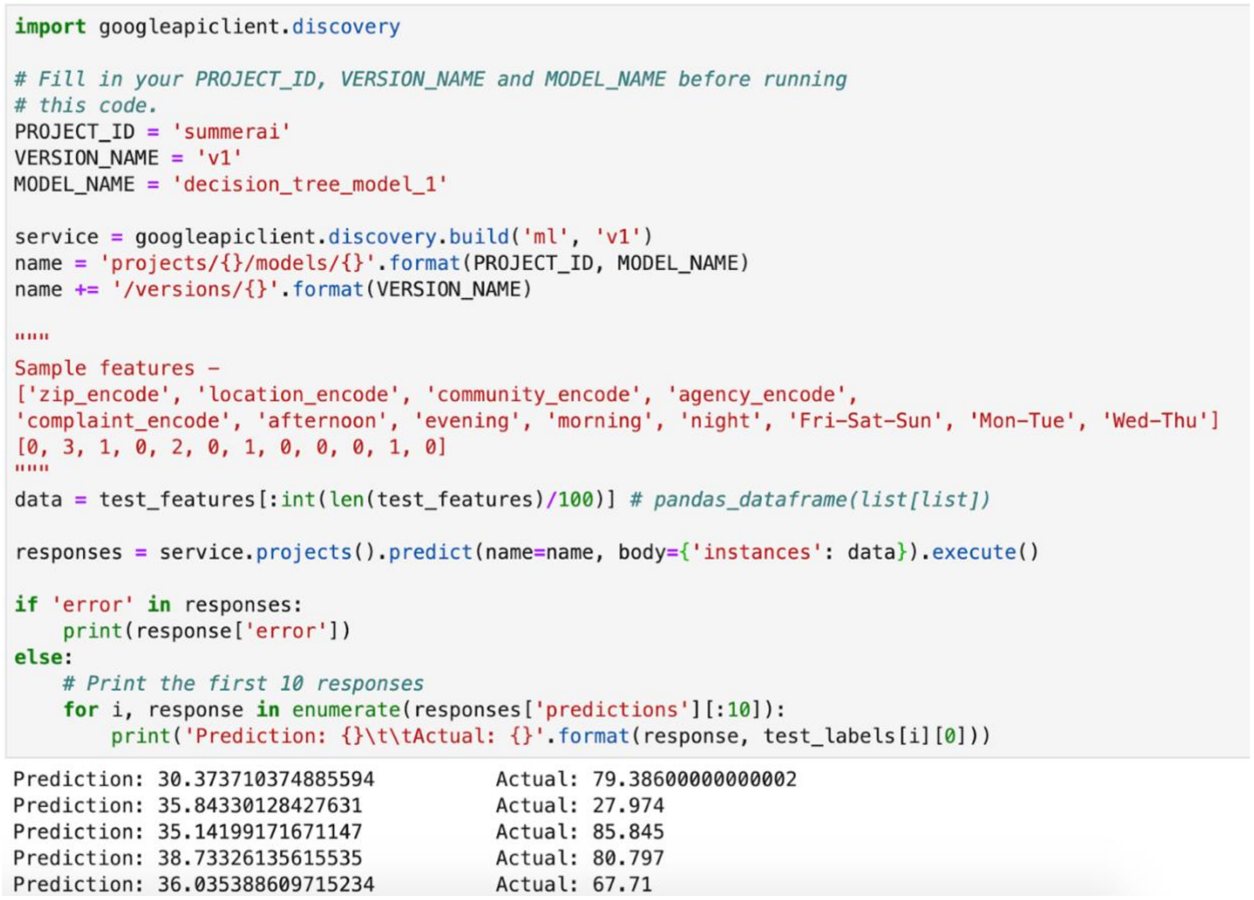
[Figure 3] Experimental Results of various Tree-Based and Regression models on the data.

Model Performance

- The Model is deployed online. Refer the below code snippet to query the model.
- Refer the below Categorical table for querying the model. Here, Requests refers to the number of complaint requests received the feature value. Eg. For all the values of incident_zip which have 5K+ Requests, they will fall in the same bin and will be encoded at 0. I have attached the json files to find Feature value belongs to which bin [‘dict_clusters’ Folder].

Actual Feature	Derived Feature	Description
incident_zip	zip_encode	0 Requests > 5K; 1 5K > Requests >= 2.5K; 2 2.5K > Requests >= 1K; 3 1K > Requests
location_type	location_encode	0 Requests >= 50K; 1 50K > Requests >= 20K; 2 20K > Requests >= 5K

		3 5K > Requests
community_board	community_encode	0 Requests >= 100K 1 100K > Requests >= 10K 3 10K > Requests
agency	agency_encode	0 Requests >= 500K 1 500K > Requests >= 400K 2 400K > Requests >= 200K 3 200K > Requests >= 150K 4 150K > Requests >= 30K 5 30K > Requests
complaint_type	complaint_encode	0 Requests >= 100K 1 100K > Requests >= 50K 2 50K > Requests
created_date	afternoon	12 noon <= created_time < 5 pm
	evening	5 pm <= created_time < 8 pm
	morning	6 am <= created_time < 12 noon
	night	8 pm <= created_time
created_date	Fri-Sat-Sun	Boolean feature
	Mon-Tue	Boolean feature
	Wed-Thur	Boolean feature



```
import googleapiclient.discovery

# Fill in your PROJECT_ID, VERSION_NAME and MODEL_NAME before running
# this code.
PROJECT_ID = 'summerai'
VERSION_NAME = 'v1'
MODEL_NAME = 'decision_tree_model_1'

service = googleapiclient.discovery.build('ml', 'v1')
name = 'projects/{}/models/{}'.format(PROJECT_ID, MODEL_NAME)
name += '/versions/{}'.format(VERSION_NAME)

"""
Sample features -
['zip_encode', 'location_encode', 'community_encode', 'agency_encode',
'complaint_encode', 'afternoon', 'evening', 'morning', 'night', 'Fri-Sat-Sun', 'Mon-Tue', 'Wed-Thu']
[0, 3, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0]
"""
data = test_features[:int(len(test_features)/100)] # pandas_dataframe(list[list])

responses = service.projects().predict(name=name, body={'instances': data}).execute()

if 'error' in responses:
    print(response['error'])
else:
    # Print the first 10 responses
    for i, response in enumerate(responses['predictions'][:10]):
        print('Prediction: {}\t\tActual: {}'.format(response, test_labels[i][0]))
```

```
Prediction: 30.373710374885594          Actual: 79.38600000000002
Prediction: 35.84330128427631           Actual: 27.974
Prediction: 35.14199171671147           Actual: 85.845
Prediction: 38.73326135615535           Actual: 80.797
Prediction: 36.035388609715234          Actual: 67.71
```

[Figure 4] Demo Code for Querying the Prediction Model. Input Format is clearly mentioned.

Motor Vehicle Collision Prediction

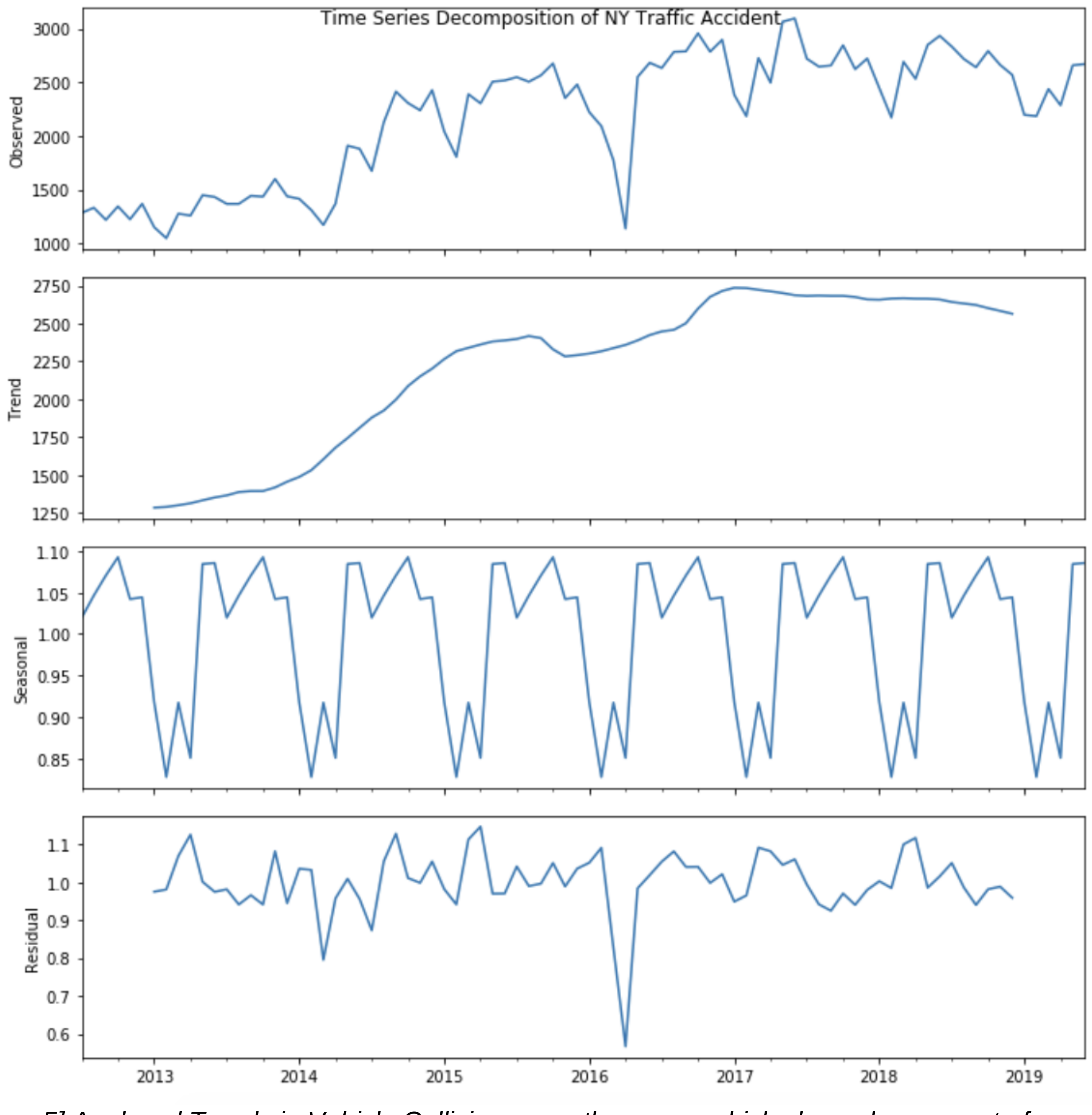
According to Wikipedia, 36,750 people died in Vehicle Collisions in 2018, and the number was 37K+ for the year 2017. [2] With Machine Learning Algorithms, given the features of the location and the past historical Collision rates, one could predict the number of collisions in the next 30 days. Having such probable statistical figures in hand, the department could take further steps to improve vehicle safety in a particular region and impose strict rules to avoid possible collisions.

The NYPD Motor Vehicle Collision dataset is available on BigQuery. This dataset is provided by the New York Police Department (NYPD) from 2012 to the present. With ~2M rows and 25+ Features, I used different Clustering and Time-Series Algorithms for the Model prediction.

Exploratory Data Analysis

- Clustering the NYC regions in 10 Clusters based on Geospatial information. The data has an inherit clustering due to the presence of 'borough' feature, but I wanted to perform clustering which had more precise subdivisions. Analyzed the Clusters.
- Checked which regions / Clusters have most people injured and killed. Focus on ones with higher values.

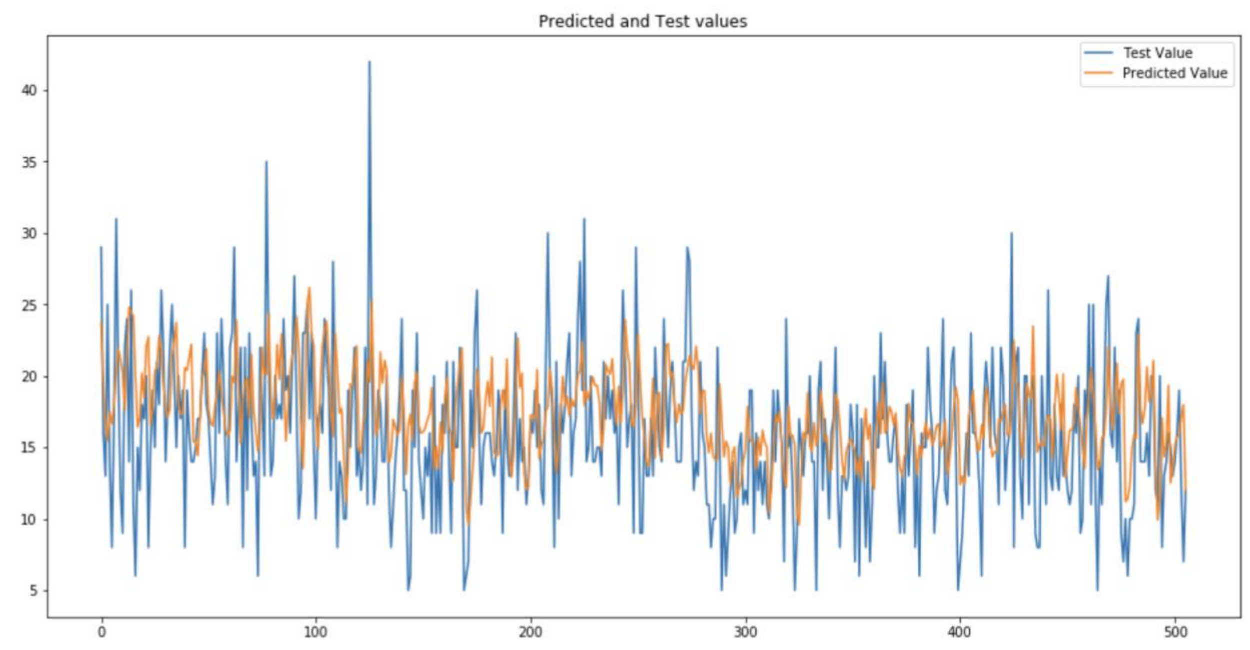
- Analyzed the Streets where there are most Collisions. Once we know which streets have higher risk we can improve the Quality by putting more control around that region. Having more Traffic controllers and police patrolling.
- Found that Sport Utility with MS / Sedan / Minibike and Sedan with Red T / Moped have many collisions.



[Figure 5] Analyzed Trends in Vehicle Collisions over the years, which showed some sort of seasonality as seen in the figure. Also, analyzed Hourly and monthly data.

Model Building, Training, and Deployment

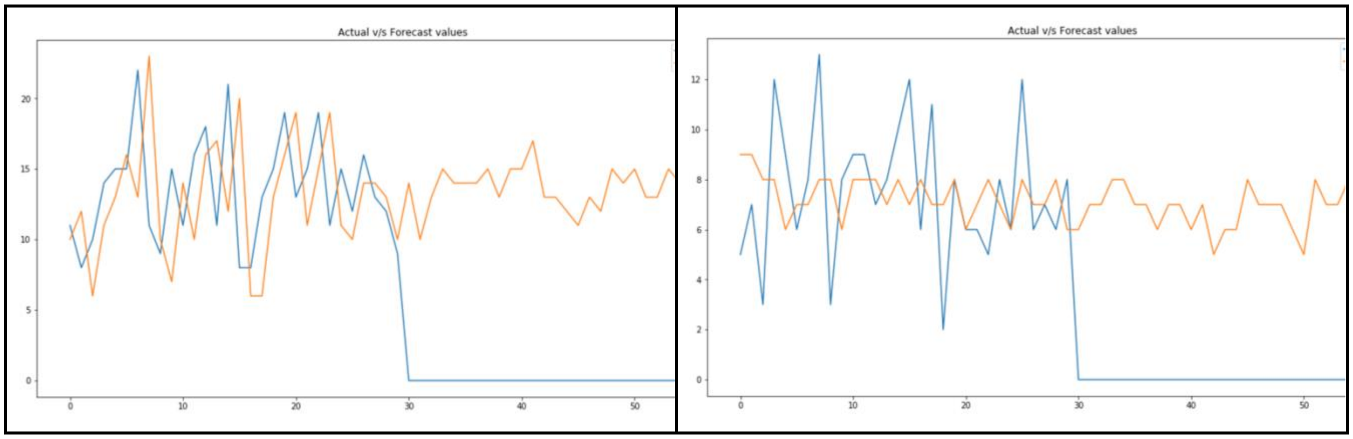
- Tried various Univariate and Multivariate Models to create the prediction model.
- Baseline Model: Last 2 weeks of average collisions. (RMSE: 5.95)
- Other Model Performances:



[Figure 6] 1D-CNN Performance (RMSE: 5.34)

Model	RMSE
ARIMA (Univariate)	5.3
AR (Univariate)	5.2
FB-Prophet (Univariate)	26.90
1D-CNN (Univariate)	5.34
CNN LSTM (Multivariate)	6.04
GRU (Univariate)	6.25

- 1D CNN Outperformed all other and was selected was testing on whole data and deployment.
- Checked if 1D CNN works on each borough. Created 5 different Models for the 5 boroughs and deployed it. Created Models for predicting number of Vehicle Collisions in Manhattan, Brooklyn, Queens, Bronx, and Staten Islands.
- We pass last 60 days of data for Testing, and it predicts the next 30 days of data. In the below figures, we check the Model's performance on the last 30 days data (For which we have the actual values) and the next 30 days (Predictions unknown)



[Figure 7] The Model for Manhattan and Bronx doesn't quite adapt to the extreme skews but it does get the hang of the general seasonality.

Conclusion

After working on three different problem statements, I have learned various Analytical and Algorithmic techniques like Regression models, Tree-based Models, Clustering-based Techniques, and Graph-based Models. Learned the Big Data and Cloud Tools and Services like Apache Beam, Spark Graphx, SparkML, Cloud-based Model Training and Hyperparameter Tuning, Model Deployment using CloudML. The deployed Models need to further be tweaked and improved.

References

[1] https://www.quora.com/What-are-practical-applications-of-the-travelling-salesman-problem
[2] https://en.wikipedia.org/wiki/Motor_vehicle_fatality_rate_in_U.S._by_year
[3] https://en.wikipedia.org/wiki/Simulated_annealing
[4] http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6
[5] https://towardsdatascience.com/self-organizing-maps-ff5853a118d4
[6] http://people.idsia.ch/~luca/acs-ec97.pdf
[7] https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3