DS 4400: Machine Learning and Data Mining I

Spring 2021

Final Report

Project Title: An Automated Ticket-Writing Machine!

TA: Saurabh Parkar

Team Members: Benjamin Kosiborod and Victoria Staada

Code:

https://github.com/bendavp/nyc-parking-ticket

Slides:

https://docs.google.com/presentation/d/1mycXWitilRD46ZzpLQsll7HBmsX6KYecnDFUWiZd9gs/

Problem Description

For our project, we are trying to train a machine learning model to predict the violation code of a ticket written in New York City. The violation code is a standardized number used throughout the five boroughs of New York City that denotes the specific violation that occurred during parking. We do this by using a 2019 dataset provided by Open Data NYC that contains approximately 11.5 million rows of parking ticket examples with around 43 columns. We decided to use the 2019 fiscal year data as we did not want the unpredictable effects of COVID to influence our results, which is why we did not want to use the 2020 dataset or the available partial 2021 dataset.

This is a classification problem because violation codes are ordinal. Therefore, they can be categorized, where each violation code is its own category. This is not a regression problem because we are not predicting a value, but rather we are predicting the class of the violation of a given ticket.

We find this quite an interesting and important project because it can help people understand the type of violation that they might incur if they park illegally in NYC. We hope that it can even help people make a decision as to where to park their car in NYC, as well as what time is the most optimal for them to park. For instance, some areas might be more heavily or more strictly patrolled, with higher fines associated with a particular violation. Even though different violation codes may be associated with the same type of violations, they can come with different fines. It would also certainly be helpful to know whether particular attributes of a vehicle or particular attributes of a specific area of NYC are associated with certain violation codes. Perhaps trucks parked on Broadway are less likely to receive a street cleaning violation, but are more likely to receive a double parking violation, while trucks parked in a more suburban area of Queens are less likely to receive a double parking violation, but more likely to receive a street cleaning violation. Such a model can help not only drivers of passenger cars, but also taxi operators, truck drivers, and the management of these respective companies. It can allow them to plan their budget on parking violations more accordingly.

Additionally, this can be an incredibly useful tool for police, as this model can double-check their work. It can possibly anticipate whether a ticket should be written with a higher or lower fine, especially if a police officer is new or patrolling outside of their usual precinct and may not be familiar with the typical procedures and protocols of a precinct in which they do not usually work. Moreover, it can help police chiefs and administrators evaluate whether they are writing tickets throughout the five boroughs in a fair way, or determine whether they are biassed towards a particular vehicle make/model, a particular day, time, or so forth.

As we are not the first to think about this problem, we found a few references that helped us learn more about related work in the space. A really fascinating resource we found was a blog post (1), where the author had visualized NYC parking ticket violation data that she had obtained from NYC Open Data. She had visualized the data into an interactive Tableau visualization. It had a heat map of the days and times of violations, as well as a heatmap of the areas of Manhattan. If you selected a particular area, a street view map would appear and show dots in all of the locations that violations were recorded in that area. There was also a section for main

offenders and another section for violation descriptions that when selected, would filter the dataset and update the other visuals accordingly. For instance, if filtered on taxis, you would see the heatmap of the days and times of violations from only taxis. This resource was one of the main inspirations for our project idea.

In our research, we had also found a couple of academic papers that are related to our project. One paper (2) had focused its research on predicting the spatiotemporal legality of on-street parking using data obtained from NYC Open Data and machine learning models such as kNN, logistic regression, Naive Bayes, SVM and random forest. In addition to the NYC parking ticket violation data, they had used a collection of points of interest to better understand the surrounding environments that parking violations were found and had used a collection of human mobility patterns to better illustrate the time spent at a certain point of interest. Another paper (3) that was quite interesting had conducted its research on predicting double parking events in NYC using machine learning models such as decision trees and random forest. They had used data from DOT camera feeds, parking violation tickets from NYC Open Data, 311 service requests, and geo-located twitter data in order to create their framework. The top few features that affected their prediction the most were the volume of traffic, number of hotel rooms near the violation, and the length of the block.

References

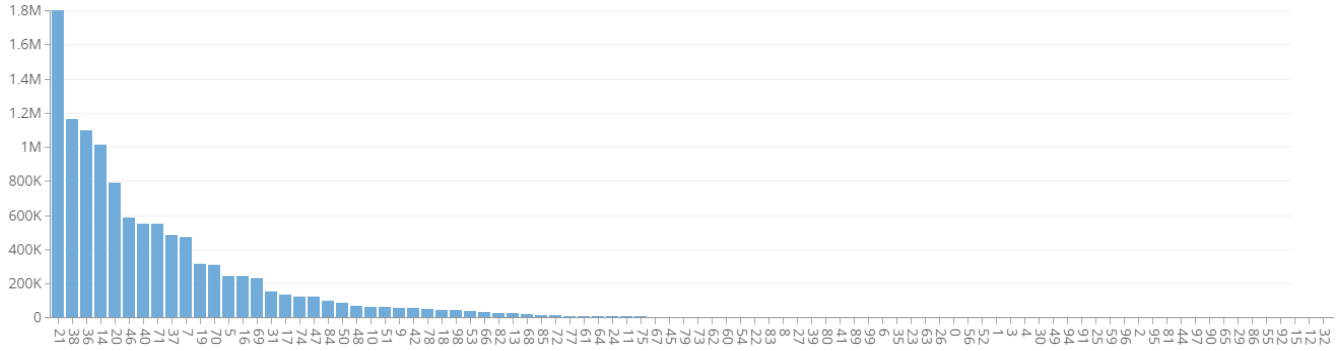
(1) https://interworks.com/blog/modonnell/2016/03/18/dont-get-ticketed-nyc-parking-violations-data-viz/
(2) https://www.tandfonline.com/doi/full/10.1080/19475683.2019.1679882
(3) https://www.researchgate.net/publication/318990056_A_Data-driven_Approach_to_Predict_Double_Parking_Events_Using_Machine_Learning_Techniques

Dataset and Exploratory Data Analysis

The dataset we are using can be found here:
https://data.cityofnewyork.us/City-Government/Parking-Violations-Issued-Fiscal-Year-2019/faiq-9dfq

The data we are working with is part of a collection of 750 free public datasets published by New York City agencies and other partners under NYC Open Data. For this project, we decided to use a dataset that focuses on parking violations issued. The NYC Department of Finance collects data on all parking violations issued in NYC for each fiscal year. The dataset we are using is only of the parking violations issued during the 2019 fiscal year, but there are other datasets ranging as far back as 2014. We figured it would be best to explore the most recent dataset that was not affected by COVID, as well as only focus on one fiscal year due to the large number of violations issued in a year. In particular, our dataset contains approximately 11.5 million records, which amounts to over two gigabytes of data. It also contains 43 columns, of which we initially believed we would use about 20. There are 99 classes, or violation codes, in which we are trying to classify our data into.

Violation Codes



Top 5 and Bottom 5 Violation Codes

Violation Code	Violation Description	Amount of Violations Issued	Percentage of Dataset
21	No Parking Street Cleaning	1,803,467	~16%
38	Failure to Display Muni-Meter Receipt	1,165,883	~10%
36	Photo School Zone Speed Violation	1,098,298	~10%
14	No Standing Day Time Limits	1,013,584	~9%
20	No Parking Day Time Limits	795,751	~7%
58	No Parking Marginal Street/Waterfront	14	~0%
88	Unaltered Commercial Vehicle/Address	6	~0%
87	Unaltered Commercial Vehicle	6	~0%
93	Remove/Replace Flat Tire	5	~0%
43	Expired Meter Commercial Meter Zone	3	~0%

With a dataset that is made up of violations issued by members of the Police Department, Fire Department, Department of Transportation, and so on, there is no guarantee that every violation will be written in the same style. While going through the dataset, we found that several columns

were mostly left empty, and the ones that were full of data had multiple ways of writing the same thing. For instance, we found 40+ ways to refer to a white, mostly white car. With issues rampant in many of the columns, it was vital that we only focused our efforts on the columns that we figured would be useful for our project. The columns that we have decided to focus on include: registration_state, plate_type, issue_date, violation_code, vehicle_body_type, vehicle_make, issuing_agency, street_code1, street_code2, street_code3, vehicle_expiration_date, violation_precinct, issuer_precinct, violation_time, violation_county, vehicle_color, vehicle_year, and meter?. Part of the reason we cut down the columns from 43 to 18, was due to the lack of documentation describing each of the columns and the data that it holds. If we couldn't understand the codes that they had used and couldn't find any information about them in our research, we did not see any benefit that we would gain from using them.

Approach and Methodology

Before we can tackle our problem, the first step that we took was to clean this dataset. Since we had more than enough records to play with, namely around 11.5 million total, we figured that we could thoroughly clean the dataset. This means that we had the freedom to drop many of our rows, as there was no way that we could train a model on such a high number of records anyway. In fact, the bulk of our work up until the milestone had been data cleaning, feature selection, encoding, and normalization.

We started this process by reading in our data. This by itself was a bit of a challenge because, with 11.5 million records, we knew it would take a very long time to read all of our data into memory. First, we experimented with starting the project on our local machines, and seeing if we could download all of the data from the Open Data NYC API directly. This took a substantial amount of time on our personal machines, which led us to try searching for another approach. We then attempted to use Google Colab in the hopes that it would be a more stable, dedicated platform, and therefore have slightly faster download times. Unfortunately, it did not work out. In the end, we settled on creating a NEU Discovery account, which allowed us to connect to the Discovery instances and run Jupyter Notebook servers on those instances. While the download time for the data still ended up being around 10 minutes, this was the fastest method we had tried yet. Plus, we were able to allocate more memory to Jupyter than we could on our machines, which sped up our process.

Next, we investigated each column in our data to establish what our target and features should be so that we can start filtering out columns. We started with 43 columns, but after settling on violation code as our target, we had 42 features to work with. This was quite a challenge, as we had to determine which columns were worth keeping, had too little data, were too difficult to clean, or if any columns would contribute to the overfitting of our data.

We mainly used the Open Data NYC visualization website to visualize and evaluate the data that we were working with. This was an appropriate tool for us to use, as opposed to doing it manually in Jupyter Notebooks, because this provided a quick way for us to create visualizations since we had so much data. The Open Data NYC visualization tool allowed us to both visualize each feature as we were working with it in parallel in Jupyter Notebook, as well as view some example data and a summary table to give us an idea as to the kind of values and data types to

expect from a given visualization.

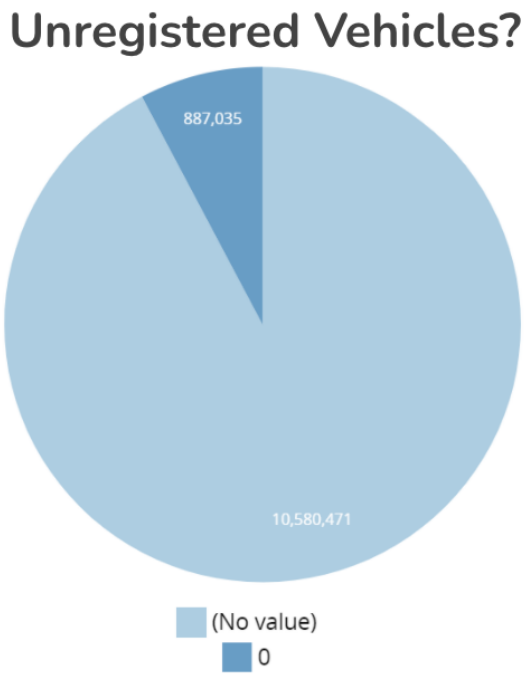


Figure 1: A screenshot of the Open Data NYC Visualization tool, displaying a pie chart of the values contained in the "Unregistered Vehicle?" column. With 85% of the values being NaN and 15% being 0, we decided to drop this column entirely as it was mostly empty, and the data that it did have was completely useless.

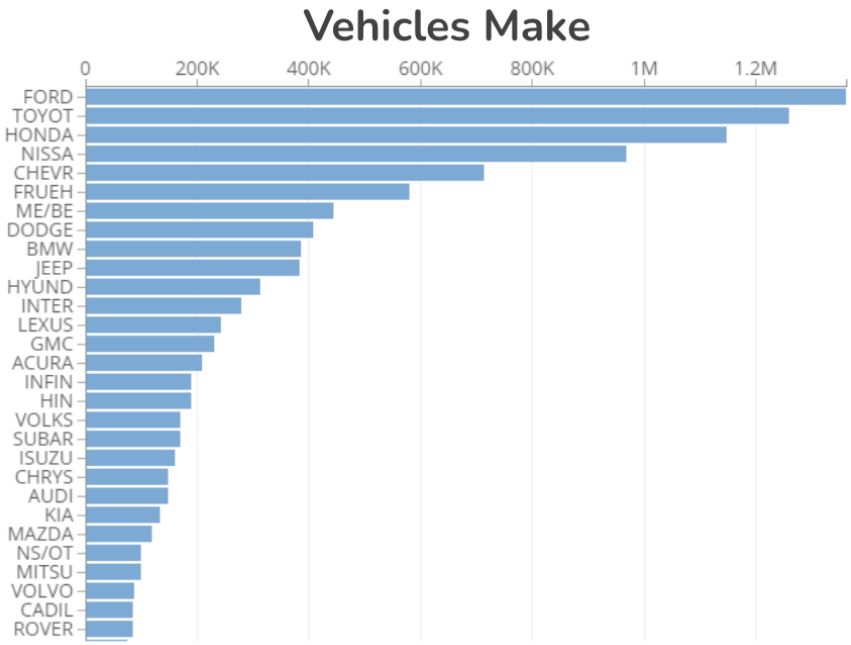


Figure 2: This screenshot displays a bar chart of vehicle makes in our dataset. This bar chart helped us identify misspelled/incorrectly entered vehicle makes and allowed us to correct the data entry during cleanup. We also identified makes that we could not recognize or that had very few records associated with them, which greatly aided us in the cleanup process as it made it easy to drop those values.

We first decided to drop identifying information, such as plate ID and summons number, as these identifying features were unique for every row and would not help us with our analysis. Next, we

decided to drop columns with too many null/NaN values (80%+). Then, we dropped columns that we could not parse or understand because the data documentation was quite sparse. We often had to rely on logic, good guesswork, or documentation that had been left by others that had analyzed this dataset. We attempted to reach out to Open Data NYC to get an explanation for a certain column, but they did not get back to us. Finally, we dropped columns that we felt would skew the prediction too much to a certain violation code, because we did not want to overfit our data.

Once we dropped the columns that we did not want to work with, we ended up with 17 features that we now had to clean. Some features were quite simple to clean either because they were numerical, and the potential values for those features were self-explanatory, or because we had good documentation of the potential values for that data. We struggled with cleaning some of the other features either because there was a lot of erroneous and repetitive incorrect data entry, or because we simply did not understand how those columns worked and had to do quite a bit of research for ourselves. For example, we did not know what to make of street codes for a long time, as these encodings are specific to New York City, and are quite poorly documented.

We also struggled with cleaning categorical columns that were strings, such as vehicle make, vehicle color, and vehicle type/class. Essentially, we manually went through the unique values of these columns and mapped each unique value to our own value, interpreting some of the values along the way, which was quite difficult. If there were too few records for a particular value, we simply dropped those rows, as it was too much effort to go through each incorrect data entry to fix it. Since we had a lot of data to work with, we did not mind dropping some of the records with incorrect data for a few of the features. This is where we had to rely on the work of previous analyses, as well as our own adequate guesswork. For one column, in particular, meter number, we decided that it would be easier to encode the columns as a boolean, meter?, which told us whether the car was parked at a meter or not, rather than drop the usage of meter numbers entirely.

Finally, once we went through each of the 18 columns and manually cleaned all of them, we dropped any rows that contained null/NaN entries, reset the index, and had a clean DataFrame to work with. We started with approximately 11.5 million rows and ended up with approximately 4.5 million rows, which is still plenty of data to work with. So much so that we decided for the sake of time to sample 100K records from our 4.5 million clean records to use for training and testing the algorithm. We split these 100K records into training and testing sets using scikit-learn's standard test_train_split method, which randomly splits the data we sampled into a training set (75% of the data) and a testing set (25% of the data).

Once we had our training and testing sets, we were ready to train and test some machine learning algorithms, but first, we had to properly encode and scale our features. Since most of our features are categorical, we used sklearn's OrdinalEncoder to encode our categorical columns. Afterward, we coerced our two DateTime columns into int columns using pandas's astype() method. Before testing and training our supervised models, we made an attempt to look for outliers and remove them by using Isolation Forest. Finally, we trained and tested 4 different supervised models: k-Nearest Neighbors, Decision Trees, Random Forest, and AdaBoost, as well as one unsupervised, Feed-forward Neural Network model. This was relatively straightforward, with a

lot of the legwork relegated to scikit-learn and keras/tensorflow.

For each model that we trained and tested, we calculated a number of metrics on the performance of the model. Firstly, we calculated accuracy and error for all of the models, as well as printed a classification report that broke down the precision, recall, and f1 scores for each class that our model encountered, in both training and testing, as well as giving is the macro and weighted averages for precision, recall, and f1 scores. We also calculated the false positive and true positive rates for each class that our model encountered in training, which we then micro-averaged to generate the ROC curve, as well as the AUC value for each supervised model, to generate our final supervised model ROC curve comparison graph. Finally, we used in-class code to calculate the loss and accuracy values for our FFNN model in both training and testing at each epoch, then plotted the training and testing loss value at each epoch over time to create a loss graph.

Discussion and Result Interpretation

k-nearest neighbors

Training

	Precision	Recall	F1 Score	Support
accuracy			0.40	65661
macro avg	0.21	0.16	0.12	65661
weighted avg	0.41	0.40	0.38	65661

Testing

	Precision	Recall	F1 Score	Support
accuracy			0.17	21827
macro avg	0.04	0.04	0.04	21827
weighted avg	0.16	0.17	0.16	21827

We started with a basic kNN classifier instance from scikit-learn, and trained and tested it. We found poor initial results, and so we experimented with changing the k value to several different numbers of neighbors, from the low range at 1 and 5 nearest neighbors, all the way up to 100 neighbors, thinking that perhaps with so much data, kNN may produce more reliable results with a high number of neighbors.

Surprisingly to us, we found that sticking with scikit-learn's default, at k=5, worked the best for our data. Unfortunately, but expectedly, our results were still quite poor. We suspect this is because kNN is not meant to be used on a high-dimensionality dataset with lots of features and data, especially since we were not sure how separable our data was.

Decision Trees

Training

	Precision	Recall	F1 Score	Support
accuracy			0.51	65661
macro avg	0.59	0.25	0.29	65661
weighted avg	0.53	0.51	0.47	65661

Testing

	Precision	Recall	F1 Score	Support
accuracy			0.41	21827
macro avg	0.21	0.13	0.15	21827
weighted avg	0.37	0.41	0.37	21827

Once again, we started with a basic decision tree classifier instance from sklearn, and trained and tested it. Our initial results were similar to the above summary, in that we had low accuracy for both training and testing. We found that these results were improved when we tried using different max_depth values for the tree we were training. After trial and error with a few different max_depth values, we found that setting the maximum tree depth to 13 produced the best results for our data, which are shown above.

While these results are not very promising, once interesting detail that we noticed is that it seems like the decision tree model did not overfit to our training data very much compared to our other models. Although decision trees ended up being only our second-best model, we feel that there is a lot of potential for a tree-based model with this data. This led us to select tree-based models for our next two supervised models.

Random Forest

Training

	Precision	Recall	F1 Score	Support
accuracy			1.00	65661
macro avg	1.00	1.00	1.00	65661
weighted avg	1.00	1.00	1.00	65661

Testing

	Precision	Recall	F1 Score	Support
accuracy			0.47	21827
macro avg	0.31	0.21	0.22	21827
weighted avg	0.43	0.47	0.42	21827

We started with a base random forest model from scikit-learn, just like the other models, except that in this case, we thought that it might be good to set a max_depth of this model to the same as our decision tree model, at 13, since the decision tree performed best at that max_depth. However, we ended up finding that not setting any max_depth at all resulted in our best random forest model. Not setting a max_depth allowed the tree to be fully expanded into pure nodes at the end.

This, in combination with the bagging methodology of random forest models likely allowed our model to achieve perfect accuracy at training. However, we noted that the testing accuracy dropoff is rather significant, dropping from perfect accuracy to only 47% accuracy. For this reason, we think that the random forest model that we built likely overfits to our training data to a significant degree.

We also experimented with setting num_estimators beyond the default of 100. We tried to set it to both 250 and 500 estimators. While the Discovery cluster we were using handled this smoothly, with the whole process taking less than 10 minutes, we did not find any improvement in testing with more estimators, neither in training (we cannot do better than perfect accuracy) nor in testing.

AdaBoost

Training

	Precision	Recall	F1 Score	Support
accuracy			0.60	65661
macro avg	0.89	0.74	0.80	65661
weighted avg	0.62	0.60	0.60	65661

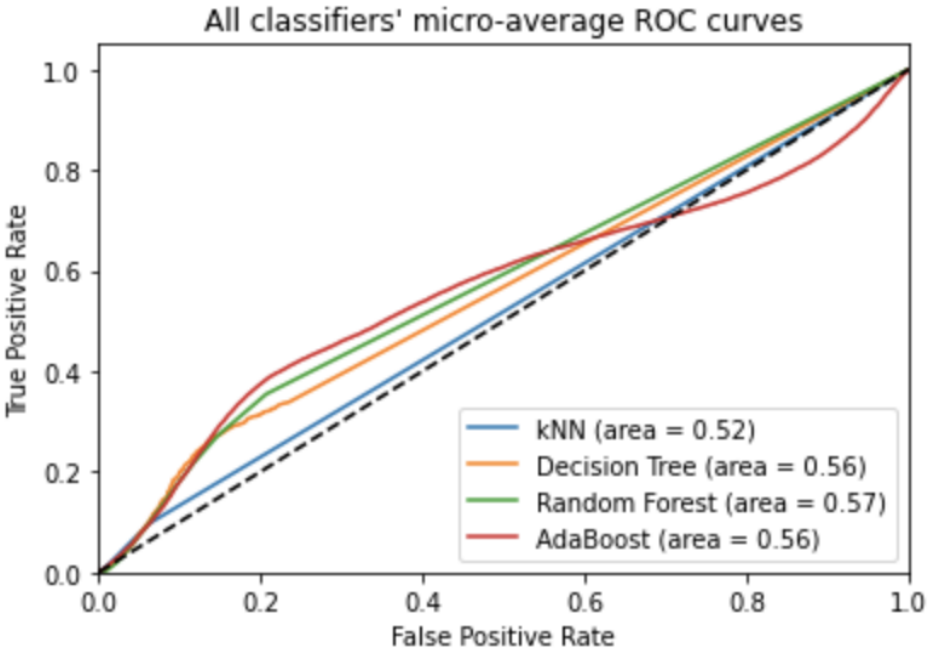
Testing

	Precision	Recall	F1 Score	Support
accuracy			0.29	21827
macro avg	0.20	0.11	0.13	21827
weighted avg	0.29	0.29	0.28	21827

We started with an AdaBoost model that we fed our decision tree model as our base estimator. This is because thus far, our decision tree model did a relatively good job, and overfit to our training data the least. kNN is not a suitable model to use as a base estimator for AdaBoost, and it did not perform well on our data anyways.

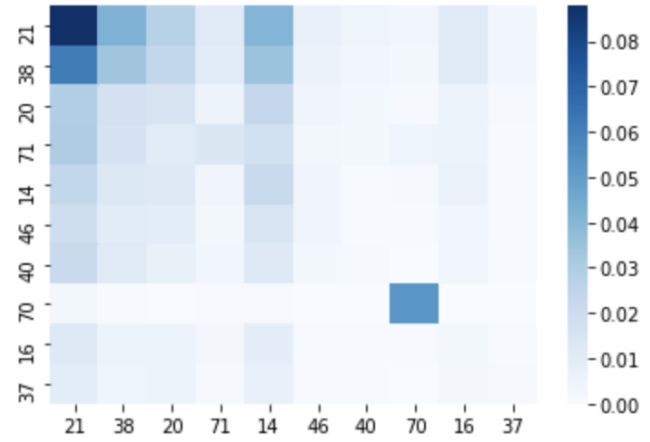
We attempted to use our random forest model as the base estimator for AdaBoost, not knowing whether the Discovery cluster can actually handle that many computations (the default

num_estimators for Random Forest is 100, and 50 for AdaBoost). We discovered that unfortunately this was too many computations for Discovery, and it was not able to train an AdaBoost model with random forest models as the base estimators.

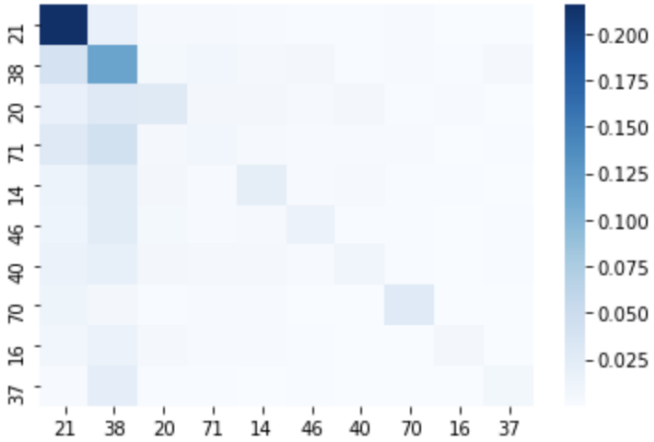


Above is an ROC curve that shows the micro-average testing TPRs/FPRs for each supervised model that we trained and tested. As expected, kNN has the lowest AUC among all the models, and performs most closely to random chance. The decision tree and random forest models both have quite similar ROC curves, with random forest having a very slightly large AUC and being slightly more balanced than the decision tree. This is especially evident at FPR=0.1, where you can see the orange line of the decision tree has slightly higher corresponding TPR values compared to the random forest model, which quickly overtakes the decision tree model at FPR=0.2.

The AdaBoost curve is the most interesting of the four, because its area is the same as the area of the decision tree model, however it does better than the decision tree for FPR rates between approximately 0.1-0.6, and consequently, does worse than the decision tree model for FPR > 0.6.

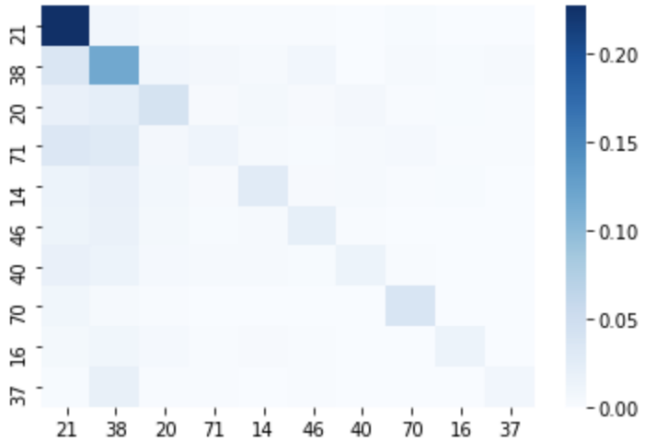


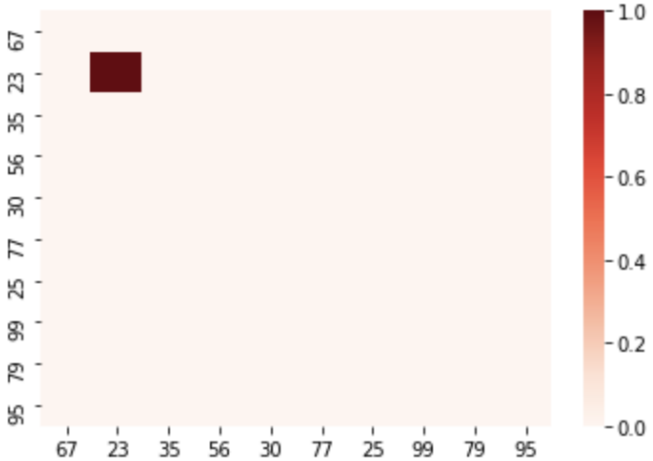
k-nearest neighbors

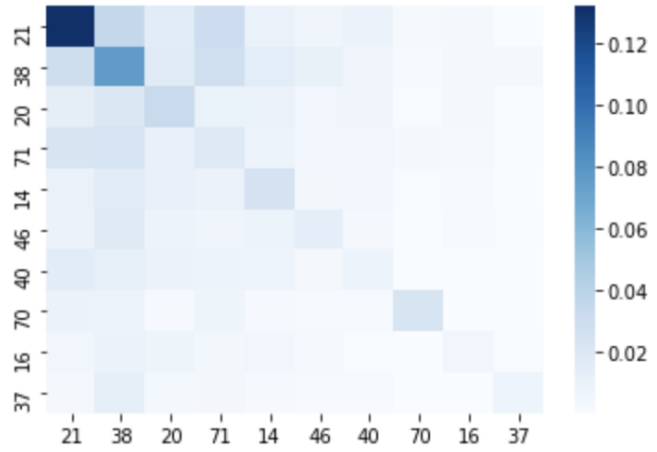


Decision Trees

Random Forest







AdaBoost

These are the confusion matrices that were produced for each model that we trained & tested. They are in the order that they appear in the model breakdowns above, namely, the kNN matrix is shown first, followed by the decision tree matrix, the random forest matrices, and the AdaBoost matrix. For each model, we attempted to create two confusion matrices: one for the top 10, and one for the bottom 10 most common features in the data sample. The top 10 matrix is shown for each model in blue in the left side column, while the bottom 10 matrix for the random forest model is shown in the right side column.

The reason that only the random forest model has the bottom 10 matrix displayed is because when we attempted to create a bottom 10 matrix for our other models, it turned out that each sample in the bottom 10 classes were misclassified outside of the bottom 10 classes, therefore the confusion matrix was simply a matrix of zeros, which is not useful for analysis. The confusion matrices were scaled, and therefore the legend displays percentage values rather than number of support records.

Looking at the kNN top 10 matrix, it seems like the top 10 classes were somewhat well-supported by the model, however there is a lot of misclassification among the 10 classes. Notably, class 21 (our most popular violation code -- “No Parking Street Cleaning”), as well as class 70 (“Registration Sticker Expired/Missing”) were quite well-supported by the kNN model, but there is still quite a bit of misclassification among the other top 10 most common violation codes.

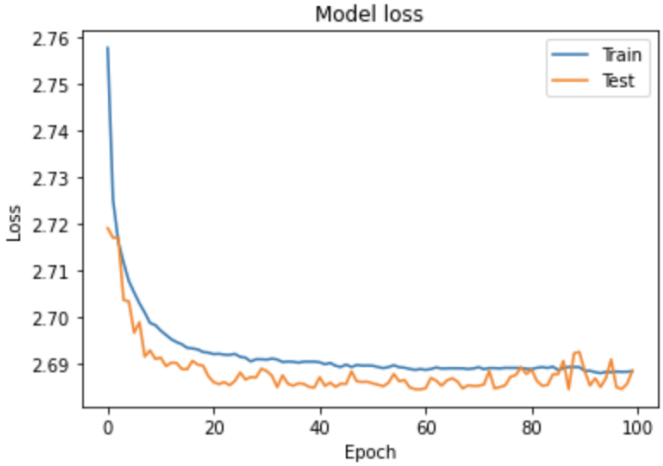
For the other models’ top 10 matrices, they all look quite similar, with the most correctly classified class being class 21, with some misclassification among the other top 10 most popular classes. Classes 38 (“Failure to Display Muni-Meter Receipt”) and 70 also tend to have more true positive classifications compared to the other most popular classes in our dataset. Interestingly, the decision tree and random forest models tend to misclassify classes 21 and 38 more than the other 8 classes among the top 10 most popular. This may simply be a result of 21 and 38 having many more instances in our dataset compared to the other popular classes.

Finally, looking at the random forest model's bottom 10 most popular classes confusion matrix, we see that pretty much all of these classes are completely misclassified outside of the bottom 10 most popular classes, except for class 23 which had all of its instances correctly classified by the model. This is likely because random forest ended up being our most accurate model, and so there is a higher chance that it will predict at least one of these classes correctly, despite the low number of samples of each of these classes (<20 samples per class in a dataset of ~100K instances).

Neural Network (FFNN)

```
Training duration: 57.86256742477417
1379/1379 [==============================] - 2s 1ms/step - loss: 2.6884 - accuracy: 0.1932

Network's test loss and accuracy duration: 59.784929037094116
```




```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_24 (Dense)             (None, 500)               9000
_________________________________________________________________
activation_24 (Activation)   (None, 500)               0
_________________________________________________________________
dense_25 (Dense)             (None, 250)               125250
_________________________________________________________________
activation_25 (Activation)   (None, 250)               0
_________________________________________________________________
dense_26 (Dense)             (None, 99)                24849
_________________________________________________________________
activation_26 (Activation)   (None, 99)                0
=================================================================
Total params: 159,099
Trainable params: 159,099
Non-trainable params: 0
```

We started with code provided in class to us by Professor Oprea. Using the keras package, we were able to modify her example a little bit to fit out particular data. We started with a fully

connected input layer with 500 nodes. Next, we used a fully connected hidden layer with 250 nodes, followed by a fully connected output layer with 99 nodes. We used the sigmoid activation function for the input and hidden layer, as we found this to increase accuracy in the neural network model, and a softmax activation function for the output layer (this is required for classification problems).

The result of the neural network model was quite surprising to us, as we expected this more complex model to work better with our complex dataset, and give us more accurate results. On the contrary, the model provided an accuracy that was slightly higher than our kNN model (our worst model). We tried including dropout layers with dropout values of 0.4 after the input and hidden layers, however this did not significantly improve the accuracy of our neural network, so we do not think that overfitting is the issue here. It may simply be that our features are not strongly predictive of the violation code of the ticket (our target).

Conclusion

If this project were to be continued in the future, we would have liked to have further investigated a few factors. First of all, we would have liked to implement cross-validation for our dataset if we had more time, as it typically helps with the challenge of overfitting and could have potentially helped boost the accuracy of all of our models. However, with the size of our dataset and the amount of time spent on cleaning the data, cross-validation would have probably helped minimally, which is why our time was spent more on other aspects.

We also would have liked to spend more time experimenting with automatic feature selection to select features that might have had more impact on improving our model and removing those that might have less of an impact to see if it could have improved our results (similar to our automatic outlier detection process). Another aspect we could have adjusted in our model is seeing if imputing data that is missing rather than dropping it from the dataset would have made a difference on the outcome of our predictions.

On a similar note, we would have liked to spend more time tuning our hyperparameters for our models to see if we could have improved our models some more, perhaps using automatic hyperparameter tuning (with scikit-learn's GridSearchCV, for example). But, we noted that the few hyperparameters we did adjust resulted in minimal improvements.

Overall, there are a lot of options for us to continue this project in the future, but it is unlikely that most of these changes would significantly change the predictive power of our models. The ultimate conclusion that we made from this project is that our chosen dataset may simply not be a great candidate for a machine learning model (at least, to predict the target that we have chosen), and in general, *that machine learning is **not** always the answer*!

Team member contribution

Victoria:

- background research
- feature selection
- feature visualization

Benjamin:

- coding
- data cleanup
- machine learning algorithm research