# Detroit Blight Ticket Compliance

Cedric Yu[♮]

**Abstract**

We summarise our findings on the (expired) Kaggle competition— which was also the final assignment of the "Applied Machine Learning in Python" course on Coursera offered by The University of Michigan— predicting whether a given blight ticket in Detroit will be paid on time. This report was written as I revisited this project after submitting the assignment. This time I did things differently from the last time.

## 1 Problem Statement

In this competition, we are tasked with predicting whether a given set of blight tickets will be paid on time. Blight violations are issued by the city of Detroit, Michigan, to individuals who allow their properties to remain in a deteriorated condition. Every year, the city issues millions of dollars in fines to residents and every year, but many of these fines remain unpaid. Enforcing unpaid blight fines is a costly and tedious process, so the city wants to know how to increase blight ticket compliance.

### 1.1 Datasets

We are given the training and test datasets. There are too many given features to be listed here; see the description in the accompanying `.py` files. The features include the violator's name and mailing address, address of the property in violation, violation code, ticket issue datetime, hearing datetime and details of the fine. There are also features that are only available in the training set but not in the test set; we discard those features.

The target is `compliance`: `1` means responsible and compliant, `0` means responsible but non-compliant, and `NaN` means not responsible.

### 1.2 Evaluation Metric

The results will be evaluated by the area-under-the-curve (auc) of the Receiver operating characteristic (ROC) curve; we want to maximise auc, so that we can maximise the true positive rate and minimise the false positive rate by adjusting the decision threshold by which we classify an instance into `0` or `1`.

#### Preliminary Observations and Processing

The training set (`train.csv`) contains 250, 306 instances, while the test set (`test.csv`) has 61, 001 instances.

We first parse the datetime columns, namely `ticket_issued_date` and `hearing_date`, of both the training and test sets, extract the respective `year`, `month`, `day`, `weekday` and `hour` features, and save the sets as new `csv` files. The datasets do not take up a large amount of memory, so we do not downcast the data types of the columns.

[♮]cedric.yu@nyu.edu. Last updated: September 22, 2021.

An immediate observation is that, in the training set there are 90,426 instances with target label `NaN` which means “not responsible". Obviously, the first we do is to drop these instances, after which we are left with 159,880 instances. Next, we drop the columns `payment_amount`, `payment_date`, `payment_status`, `balance_due`, `collection_status`, and `compliance_detail` which are only available in the training set. After that, we have several columns with some number of missing values:

| | |
|---|---|
| violator_name | 26 |
| violation_zip_code | 159880 |
| mailing_address_str_number | 2558 |
| mailing_address_str_name | 3 |
| state | 84 |
| zip_code | 1 |
| non_us_str_code | 159877 |
| grafitti_status | 159880 |
| hearing_date_year | 227 |
| hearing_date_month | 227 |
| hearing_date_weekday | 227 |
| hearing_date_day | 227 |
| hearing_date_hour | 227 |

Of these, `violation_zip_code`, `non_us_str_code` and `grafitti_status` have so many missing values that we expect to drop them.

# 2 Exploratory Data Analysis

We make plots in `dataset_study.py`.

## 2.1 Target: `compliance`

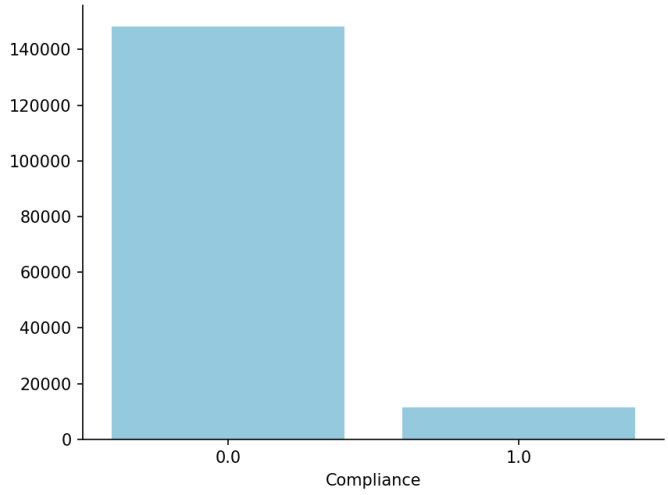



Figure 1: `compliance` in the training set.

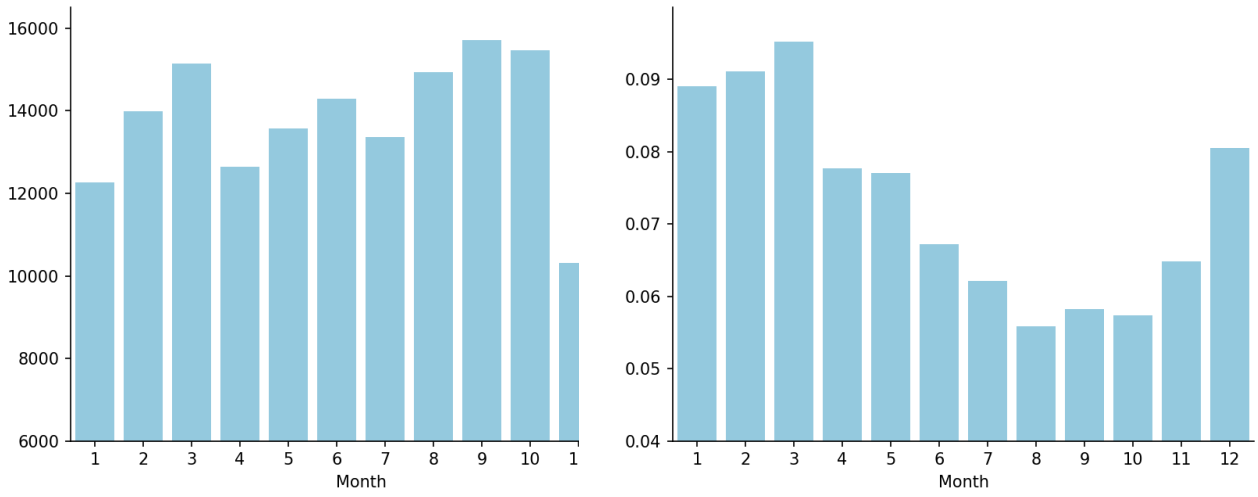
Having discarded the instances with `compliance` being `NaN` meaning not responsible, from the training set, we first get Figure 1 for the target label.

That is, the vast majority of the tickets were not complied; that's why we have a problem to solve to begin with. This means we have a **class imbalance**. This needs to be taken care of when we train machine learning models.

## 2.2 Datetime Features

Next, we look at the datetime features extracted from the features `ticket_issued_date` and `hearing_date`. We learn that the ticket issue dates in the training set were from the years 1988, 2004-2011, while those in the test set were from the years 2012-2016. There are respectively only 1 and 15 instances in years 1988 and 2004. This motivates us to drop instances in years 1988 and 2004. Including dropping the `NaN` in `compliance`, this makes **two truncations**. Similarly, from `hearing_date` we learn that the hearings in the test set happened after those in the training set. Anticipating the use of tree-based algorithms, the `year` features will be less than useful; we drop these two columns. The are missing values in `hearing_date`; we will fill those with the most frequent occurrence.

Next, for the `month`, `day`, `weekday` and `hour` features, we do not see anything particular that stands out, except that hearings only took place on weekdays (as expected). Another thing to look at is how we should encode these categorical features. We looked at the frequency plots and the target mean plots to draw inspirations, but did not always see any particular preference towards one or another. See e.g. Figure 2 for `ticket_issued_date_month`.




(a) Count plot

(b) Target mean

Figure 2: Histograms for `ticket_issued_date_month` from the twice-truncated training set.

### 2.3 Mailing Address Features

We now look at the features about the violators' mailing addresses. As a reminder, from the last section we saw too many NaN for violation_zip_code, so we dropped that column. Among the relevant columns, we only keep the columns country, state, city and zip_code, which should already contain enough information; we drop mailing_address_str_number, mailing_address_str_name and non_us_str_code.

country contains USA, Cana, Aust, Egyp and Germ. Only 11 in the twice-truncated training set are not USA. Looking at the state feature of those with country == USA, we find that it contains some non-US states(+DC), such as QC and BC in Canada. Our way of cleaning this up is as follows. We now use country to classify whether a mailing address is a US address, or not. To count as *really* being in the US, we demand that state is one of the 50 states(+DC).

Looking at state, we find the most are MI (obviously). As such, we will use frequency encoding on this feature.

Of those with mailing addresses inside the US, we study the zip_code. Most are 5-digit as expected, but there are also some with less than 5 digits, which we count as invalid and set as '0'. Most other ones are 9- or 10-digit, which are genuine zip codes with the extra 4-digit suffix, with or without a hyphen. We extract the 5-digit zip codes using regex. There are respectively 113 and 2 instances with 6- and 7-digit zip_code. A closer look at the mailing addresses suggests that those number are somewhat close but not exactly the correct zip codes. Nonetheless, they do not make up a large part of the dataset, so we simply just extract the first 5 digits as the zip codes. For the non-US instances, we set the zip codes to '0'.

### 2.4 violation_code and violation_description

The violation_description is just the explanation of the violation_code in words; drop the former.

### 2.5 Fine and Fees

We find that admin_fee and state_fee are the same for all instances. clean_up_cost is zero for all instances in the training set. Though the test set has $1,580$ instances out of $61,001$ with non-zero clean_up_cost, We cannot tell anything about it from the training data, so we drop this too. judgment_amount is a simple sum of fees.

All in all, we drop admin_fee, state_fee, clean_up_cost and judgment_amount.

## 3 Feature Pre-processing and Engineering

We now describe the feature pre-processing and engineering procedure, implemented in pre-processing.py.

### 3.1 Workflow

We first describe our workflow, where the starting point is the datetime-processed training set.

1. Load datetime-processed training set containing `year`, `month`, `weekday`, `day`, `hour` extracted from parsed datetime columns `ticket_issued_date` and `hearing_date`
2. Diacard columns that are not in the test set
3. Drop instances with `NaN` target label `compliance`
4. Restrict to instances with `ticket_issued_date_year` after year 2004
5. Fill (the one) `NaN` of `zip_code` with `'0'`
6. Separate features and labels
7. Train-validation split: we use a 80-20 split
8. Drop the following columns:

```
cols_to_drop1 = ['ticket_id', 'violation_zip_code',
'violation_description', 'admin_fee', 'state_fee',
'judgment_amount', 'inspector_name', 'violator_name',
'violation_street_number', 'violation_street_name',
'mailing_address_str_number', 'mailing_address_str_name',
'non_us_str_code', 'city', 'clean_up_cost', 'grafitti_status',
'ticket_issued_date_year', 'hearing_date_year']
```

9. Process mailing addresses
10. Fillna in `hearing_date` columns with most frequent occurrences
11. Encode categorical variables
12. Fill any remaining missing values (due to encoding) by the columns means
13. Apply `MinMaxScaler` (we will use KNN)
14. Repeat Steps 8-13 for the datetime-processed test set.

## 3.2 Processing Mailing Addresses

To process the mailing address features in the way outlined in Section 2.3, we use define the following function which is applied on the datasets:

```
import re

def country_zip_process(df) :

    US_states = ["AL", "AK", "AZ", "AR", "CA", "CO", "CT", "DC", "DE", "FL", "GA",
              "HI", "ID", "IL", "IN", "IA", "KS", "KY", "LA", "ME", "MD",
              "MA", "MI", "MN", "MS", "MO", "MT", "NE", "NV", "NH", "NJ",
              "NM", "NY", "NC", "ND", "OH", "OK", "OR", "PA", "RI", "SC",
              "SD", "TN", "TX", "UT", "VT", "VA", "WA", "WV", "WI", "WY"]

    def country_zip_func(row) :
```

```
        if row['state'] in US_states:
            row['country'] = 1
            if (len(re.findall("\d{5,5}",row['zip_code'])) > 0):
                row['zip_code']=int(re.findall("\d{5,5}",row['zip_code'])[0])
            else:
                row['zip_code'] = 0
        else :
            row['country'] = 0
            row['state'] = 'not_in_US'
            row['zip_code'] = 0
        return row

    df1 = df.apply(country_zip_func, axis = 1)

    return df1
```

## 3.3 Fillna in `hearing_date` columns

We fill the missing values in `hearing_date_month`, `hearing_date_day`, `hearing_date_weekday` and `hearing_date_hour` by their most frequency occurrences. They are respectively:

```
month_fill = 4
weekday_fill = 1
day_fill = 20
hour_fill = 9
```

## 3.4 Encoding Categorical Variables

The columns to encode are respectively:

- one-hot encoding:

  ```
  object_cols=['agency_name', 'disposition']
  ```

- frequency encoding:

  ```
  freq_cols2 = ['violation_code', 'hearing_date_day',
  'ticket_issued_date_weekday', 'hearing_date_weekday',
  'hearing_date_hour', 'state']
  ```

- mean encoding:

  ```
  target_mean_cols3 = ['ticket_issued_date_month',
  'hearing_date_month', 'ticket_issued_date_day',
  'ticket_issued_date_hour']
  ```

## 3.5 Feature Selection

We use the following features:

```
['zip_code', 'country', 'fine_amount', 'late_fee', 'discount_amount',
'agency_name_Buildings, Safety Engineering & Env Department',
'agency_name_Department of Public Works',
'agency_name_Detroit Police Department',
'agency_name_Health Department', 'agency_name_Neighborhood City Halls',
'disposition_Responsible (Fine Waived) by Deter',
'disposition_Responsible by Admission',
'disposition_Responsible by Default',
'disposition_Responsible by Determination',
'violation_code_freq_encoded', 'hearing_date_day_freq_encoded',
'ticket_issued_date_weekday_freq_encoded',
'hearing_date_weekday_freq_encoded', 'hearing_date_hour_freq_encoded',
'state_freq_encoded', 'ticket_issued_date_month_mean_encoded',
'hearing_date_month_mean_encoded',
'ticket_issued_date_day_mean_encoded',
'ticket_issued_date_hour_mean_encoded']
```

# 4 Models

We performed modeling training in `master.py` and .`master_keras_NN.py`. We also have a pipeline version of the code based on the previous attempt. We use `sklearn.neighbors.KNeighborsClassifier`, `xgboost.XGBClassifier`, `sklearn.ensemble.RandomForestClassifier`, `lightgbm.LGBMClassifier` and deep neural network from Tensorflow. We train the models with the objective of maximising the auc. For `KNeighborsClassifier`, we use `GridSearchCV` to find the optimal hyperparameter `k` (we keep `p` default). For the neural network, we use `keras_tuner` with `BayesianOptimization` to find the optimal architecture. For the rest, we use `RandomizedSearchCV` to find the optimal hyperparameters.

Our best auc's for these models with optimal hyperparameters (to the extent we know) are summarised in the following ROC plot on the validation set.

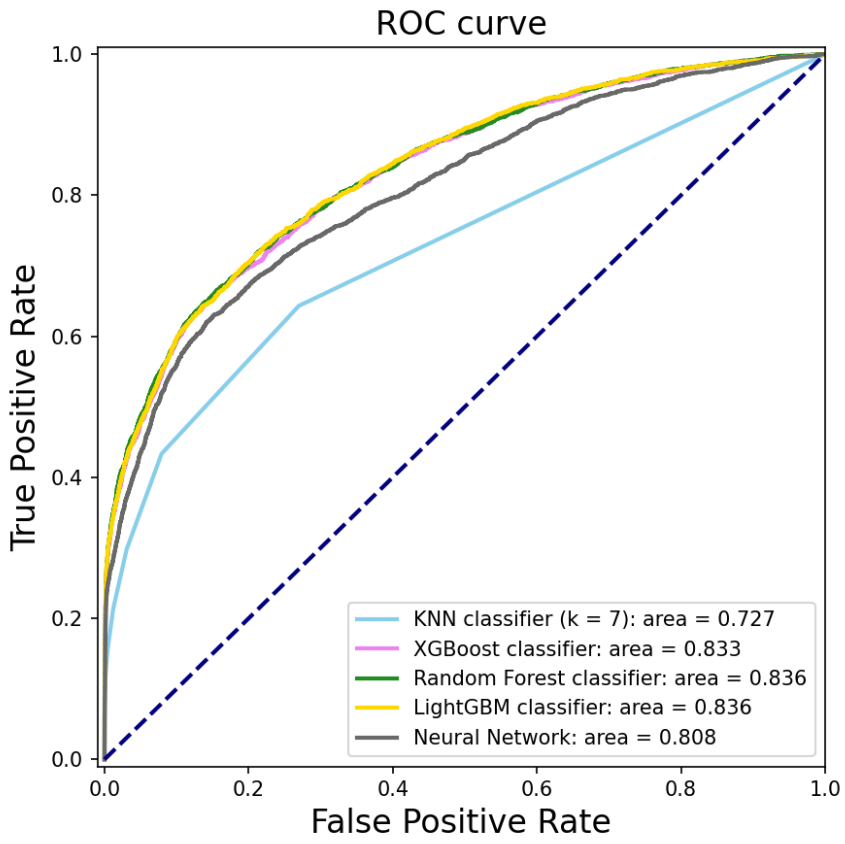



Figure 3: ROC curves of models with optimal hyperparameters on the validation set.

We see that, within the model hyperparameter spaces we probed, the tree-based classifiers namely `XGBClassifier`, `RandomForestClassifier` and `LGBMClassifier` give the highest auc of 0.833 – 0.836.

For `KNeighborsClassifier`, we got the optimal `k=7`.

For `xgboost.XGBClassifier`, we got the following set of optimal hyperparameters:

```
XGBC_model = XGBClassifier(scale_pos_weight = (127891.-9247.)/9247.,
                           eval_metric='auc',
                           n_estimators = 900,
                           max_depth = 7,
                           learning_rate = 0.01,
                           n_jobs = 8)
```

Note that we have addressed class imbalance in training the model using the `scale_pos_weight` of the training set that with which we fit the model.

For `RandomForestClassifier`, we used

```
RFC = RandomForestClassifier(class_weight = 'balanced',
                             n_estimators = 700,
                             max_depth = 20,
                             max_features= 'auto',
                             min_samples_leaf = 1,
```

```
                            min_samples_split = 16,
                            random_state = 0,
                            n_jobs = -1)
```

Here we used `class_weight = 'balanced'`.

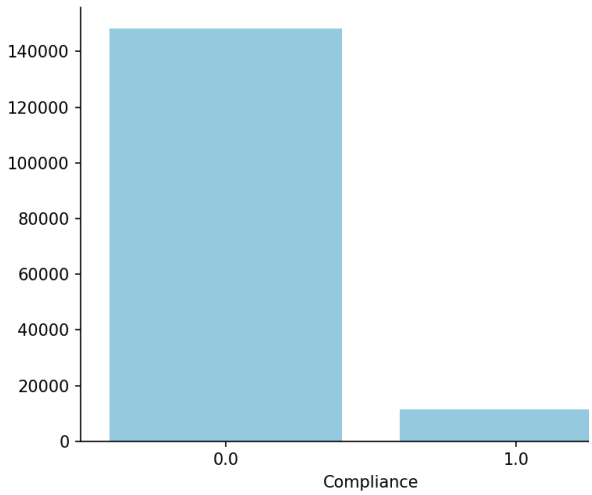
For `LGBMClassifier`,

```
LGBMclf = lgb.LGBMClassifier(class_weight = 'balanced',
                             learning_rate = 0.01,
                            num_leaves = 800,
                            n_estimators = 500,
                            num_iterations = 900,
                            max_bin = 500,
                            feature_fraction = 0.7,
                            bagging_fraction = 0.7,
                            lambda_l2 = 0.5,
                            max_depth = 7,
                            silent = False)
```

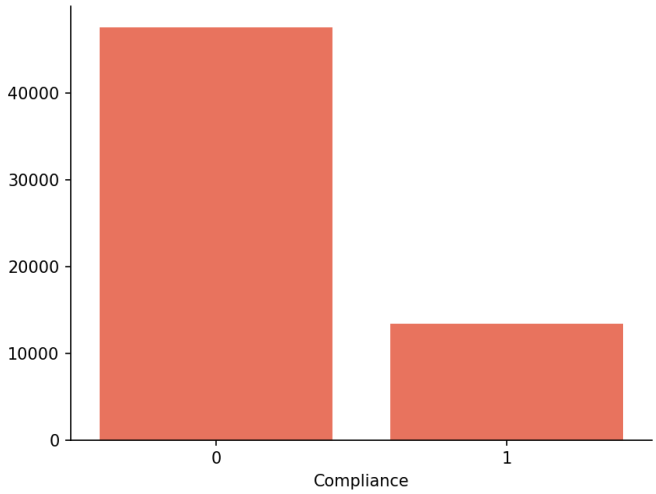
and we used `eval_metric = 'auc'` when we fit the model.

# 5 Predictions

Using `LGBMClassifier`, the test set predictions take the following distribution:




(a) Training set compliance




(b) Test set compliance (predicted)

Figure 4: Compliance: training set VS test set predictions using `LGBMClassifier`.

Here we have included the training set plot Figure 1 in Figure 4a for comparison. It appears that the predicted test set has a higher ticket compliant rate than the training set: 22.0% versus 7.25%.

Our original attempt, for the submission of the assignment, achieved slightly lower validation auc's in general. The test auc from the best performing model was 0.774, which was slight lower

than the validation auc's. Possible remedies include regularising the models to reduce overfitting. We expect the current approach to result in a better test auc.

Finally, we look at the feature importances of the best performing models, `XGBClassifier`, `RandomForestClassifier` and `LGBMClassifier`.

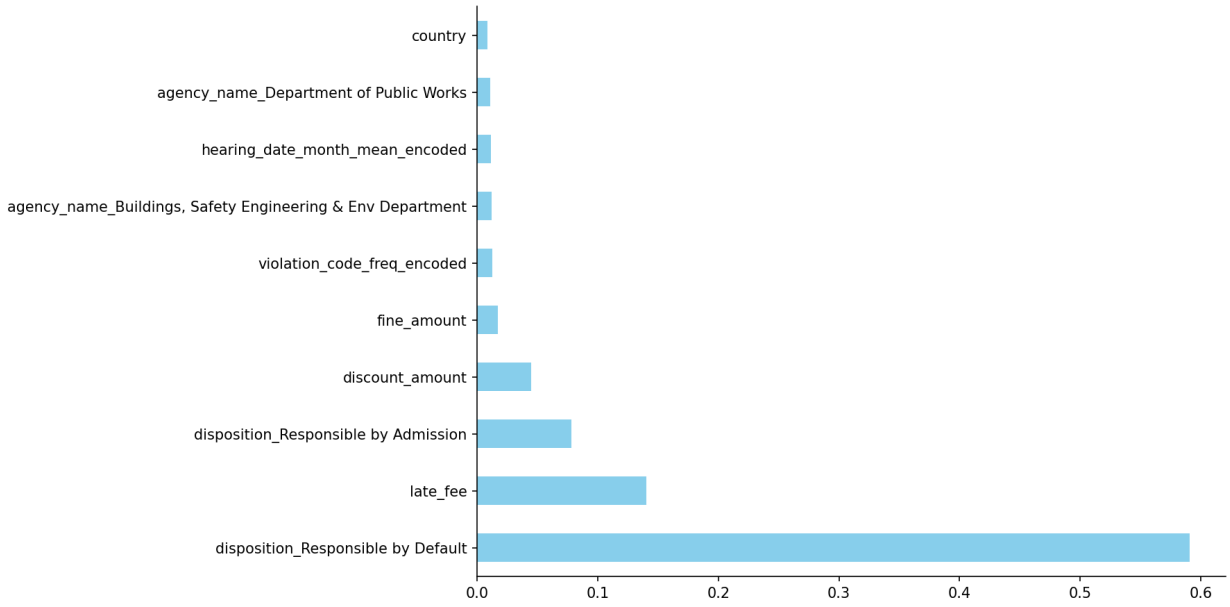



Figure 5: Feature importances in `XGBClassifier`.

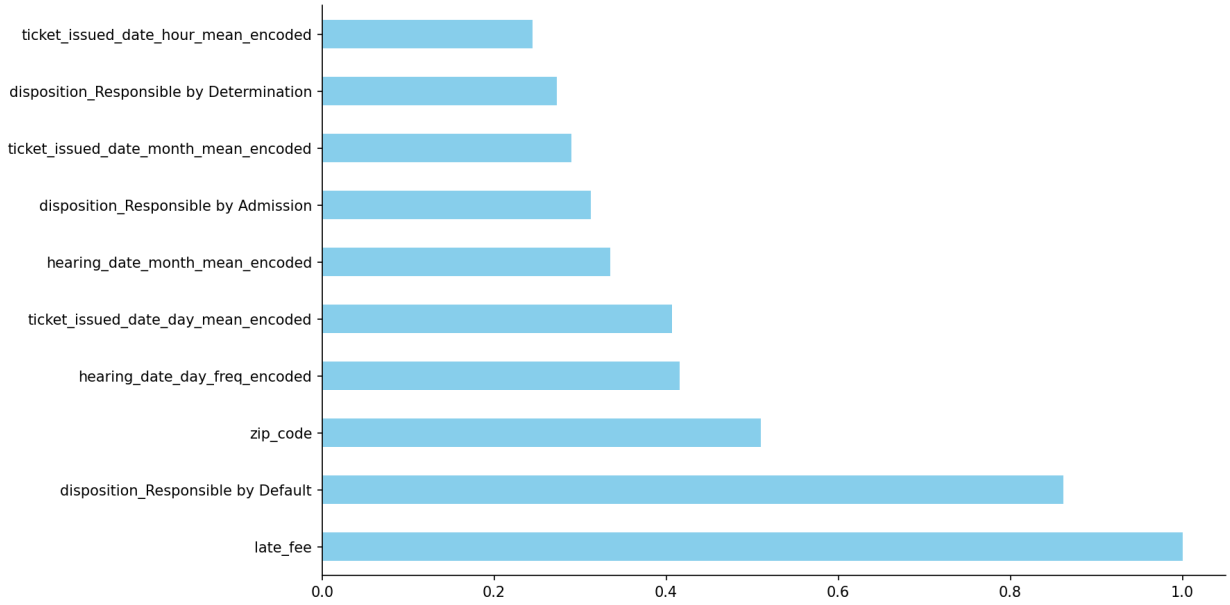



Figure 6: Feature importances in `RandomForestClassifier`.

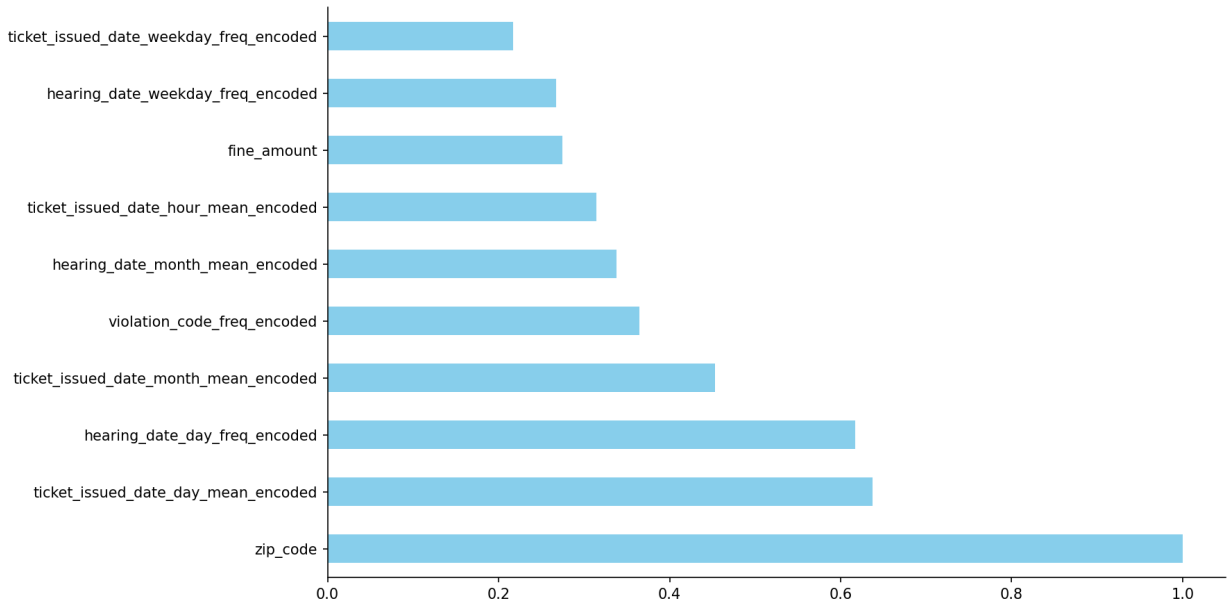
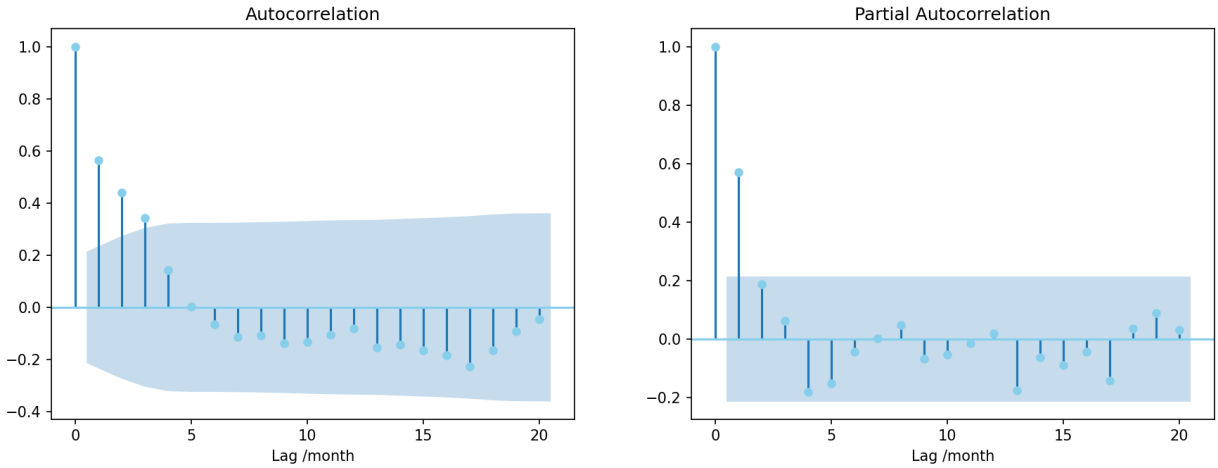



Figure 7: Feature importances in `LGBMClassifier`.

Though the features importances vary among different models, we can make sense of some of them. For instance, it makes sense that compliance depends on `deposition_Responsible by Default`: if the violator does not show up in the hearing, it is natural to expect him/her to not comply with the ticket payment either. It also makes sense that `zip_code` of the mailing address is important; zip codes with lower incomes and higher poverty rates may have lower compliance.

# 6 (Partial/) Auto-Correlation

Out of curiosity, we looked at the mean compliance rate of each month (of the ticket issued date) in the training set; this is a time series. We plotted the auto-correlation and partial auto-correlation:




(a) Auto-correlation

(b) Partial auto-correlation

Figure 8: Auto-correlation and partial auto-correlation of monthly mean compliance in the training set.

We see that the auto-correlation decays gradually from zero lag, while the partial auto-correlation is large at lag=one month only. We conclude that it appears to be described by an auto-regressive AR(1) model.

# 7 Dimensionality Reduction

We also explored the impact of dimensionality reduction on the model performance at the end of `master.py`. Specifically, we performed dimensionality reduction on the processed datasets, respectively by principal component analysis (PCA) and truncated SVD decomposition with `sklearn`. We use
`xgboost.XGBClassifier` with the optimal hyperparameters given above, and evaluated the validation set AUC for different choices of the number of components `n_components` to which the engineered features reduce. The results are given in Figure 9. It shows that performance worse with the application of dimensionality reduction, even when `n_components` equals the number of engineered features (24).

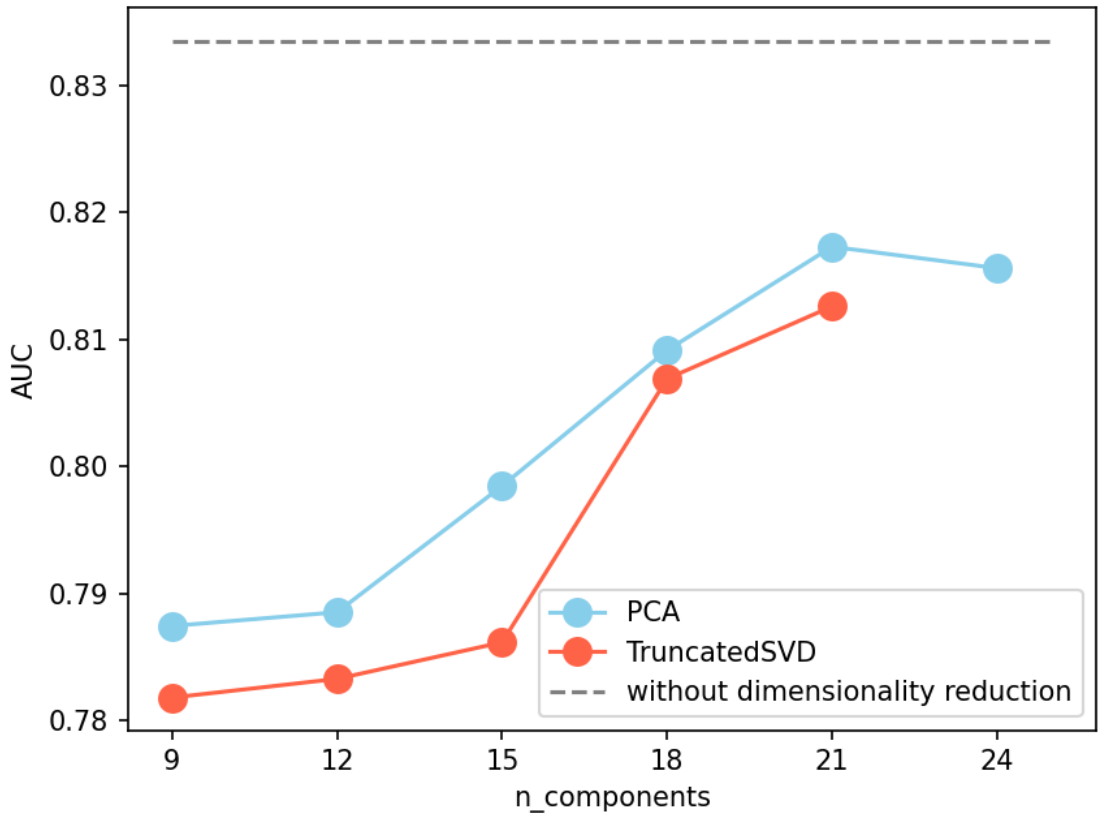



Figure 9: AUC from `XGBClassifier`, after dimensionality reduction using PCA and Truncated SVD respectively.

This may suggest that our problem has not suffered from the curse of dimensionality that would have hindered performance with too many features. Moreover, the reduction did not result in a significant decrease in model training time, given the not-too-large dataset and the high speed of `xgboost`.

## 8 Acknowledgment

We acknowledge the authors from various sources online, whose tools and techniques were borrowed and implemented in our codes. (I didn't keep track of the references.)