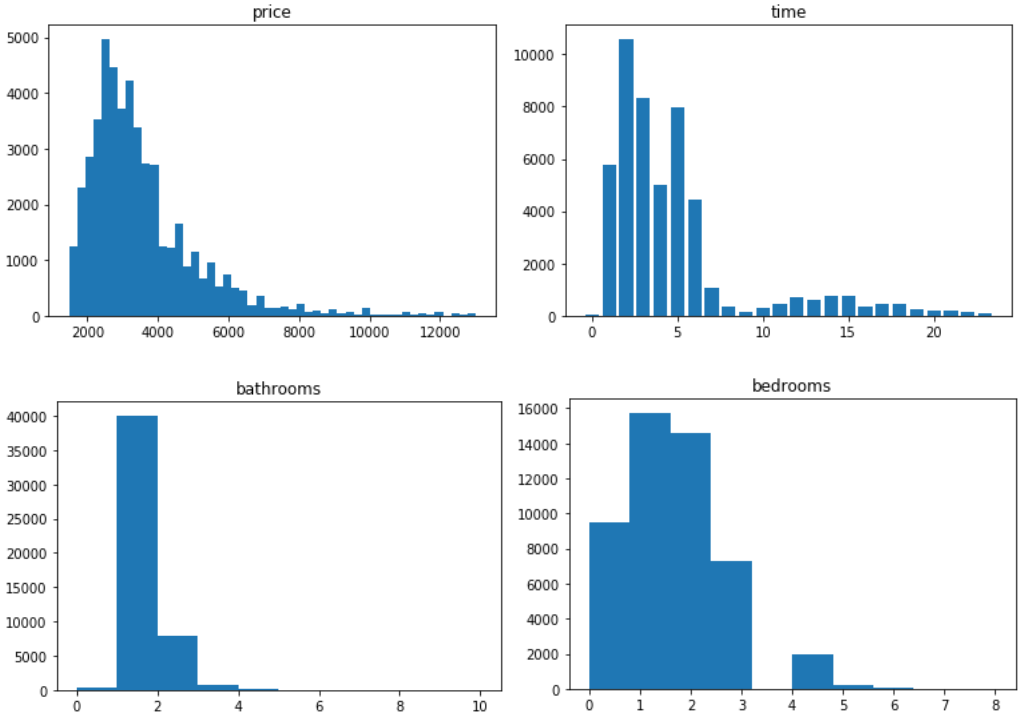
Final Report

Dekai Lin, Zherui Shao, Luowen Zhu

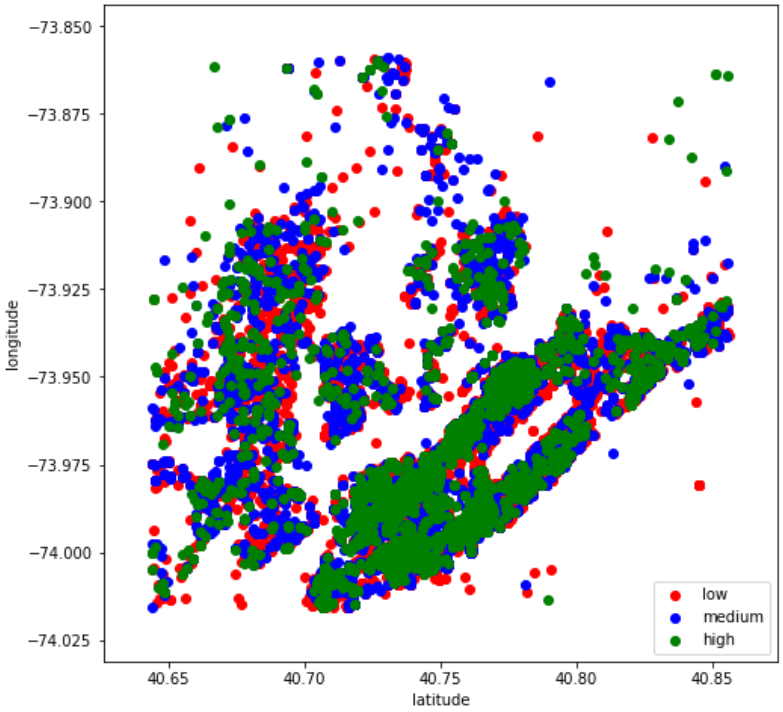
Code repo: https://github.com/DekaiLin/cmpt459project/blob/master/Final%20Report/final.ipynb

1. Exploratory data analysis:

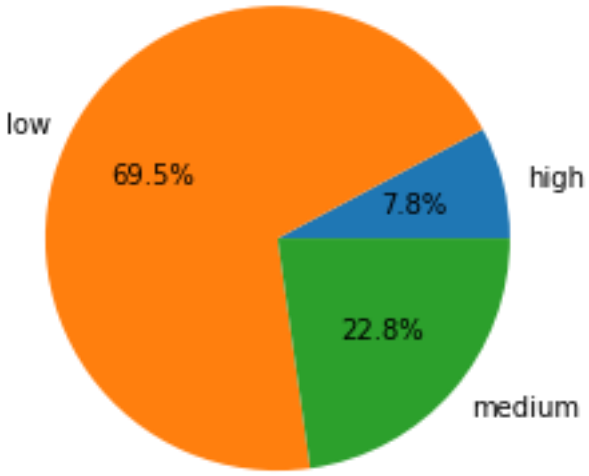
Attributes ['price', 'time', bathrooms', 'bedrooms'] are right-skewed distribution.



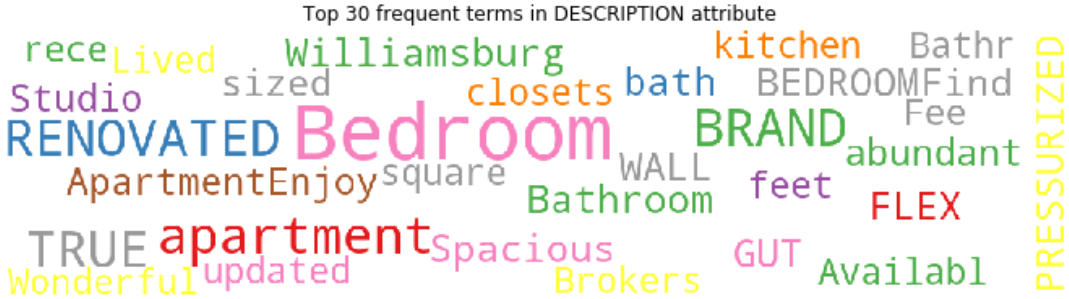
The plot of attributes ['latitude', 'longitude'] can roughly outline the New York City.

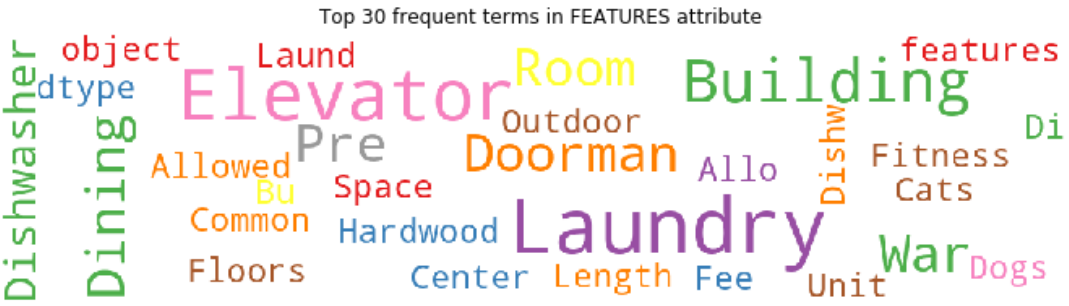


For the attribute ['interest_level'], 'low' label are the most, followed by 'midium' label, and finally 'high' label.

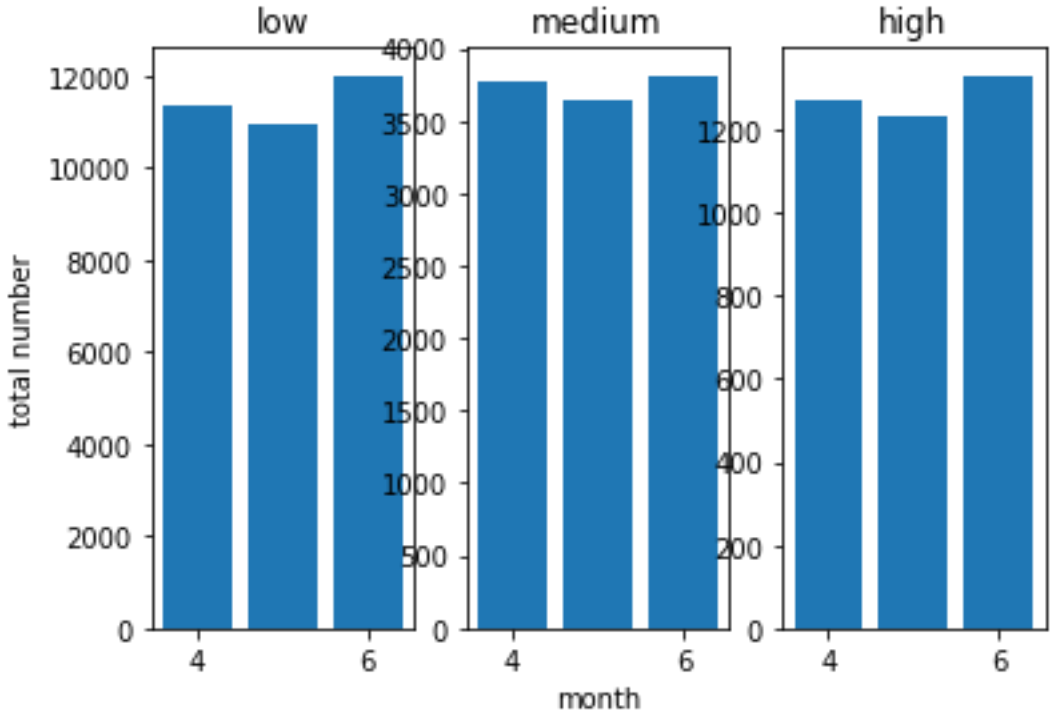


The word clouds below show the most frequents words appear in attribute ['description', 'features'].





There was no significant change in the interest_level over time. They are nearly uniform distribution.



2. Data pre-processing:

Missing value Attribute Name	Number of Missing Values	Description
bedrooms	0	
bathrooms	0	
building_id	8286	Building id is 0
created	0	
description	1685	No description
display_adderss	137	No address
features	3218	No features
latitude	12	Latitude is 0
listing_id	0	
longitude	12	Longitude is 0
manager_id	0	
photos	3615	No photos
price	0	
street_address	10	No address
Interest_level	0	

Outlier Attribute Name	Number of Outliers
bedrooms	0
bathrooms	0 (313 training data are 0 bathrooms)
latitude	38
longitude	16
price	4

Missing value:

Building id can use default value (such as "unknown") to fill in the missing value because this attribute has little effect to classification.

The missing value of description and features attribute can be supplemented by cross-references. If both of them are missing, the features can be extracted from photos.

For the latitude and longitude, we can replace the missing value by the New York's latitude and longitude.

For the display address and street address, we can get the address by latitude and longitude.

For the photos, we can only drop the missing value because we have no way to impute the missing photos from other attributes.

Outliers:

The latitude and longitude outliers can be removed because the data is got from New York. So the actual latitude and longitude can be manually obtained by the address.

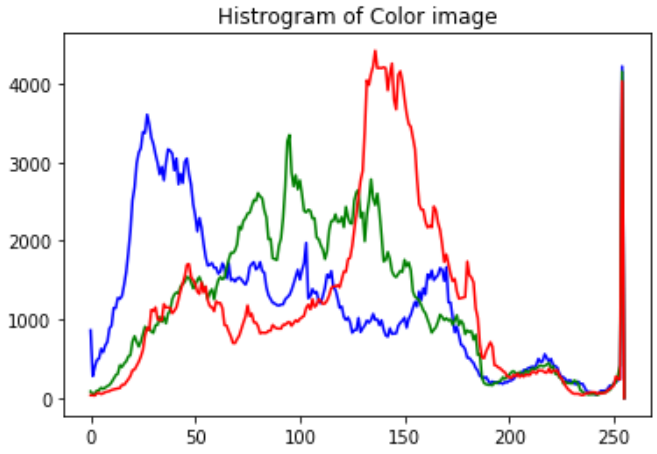
For the bathrooms, 0 bathrooms seem to be a outlier. This need to be checked by human. So these values cannot be removed.

For the price, the outlier values are unrealistic, so remove them.

For other not numerical attributes, they are not comparable, so outlier detection is not meaningful for them.

Features extraction from image:

we used histograms of colors to extract features.



Feature extraction from text:

For the text data we used term frequencies as features. For example, we used 250-dimension vector to map attribute ['features'] in term space and we drop the English stop words.

Additional features from external datasets: None.

Feature selection:

We first use ['bathrooms','bedrooms','latitude','longitude','price'] as our features because they are already there and we do not need to do calculation or transformation to get them.

In Question 5, we add ['year','month','day','hour','minute'] and TF vector of the ['features'] as features for improvement. The ['year','month','day','hour','minute'] is derived from the ['created'] attribute.

3. Classifiers:

In Milestone2, we used Decision tree, Logistic regression and SVM.

In Milestone3, we used Random Forest, Gradient Boost and Naïve Bayes classifier.

In the Final Report, we used XGBoost classifier.

We use the Python library Scikit-learn for the first six classifiers and xgboost for XGBoost classifier.

We had three improvement for the classifiers: Tune Parameters, Use more features and Add the unstructured attribute.

1. **Decision tree:**

   Tune Parameters: For the Decision tree classifier, we try with different depth of the tree and we find currently depth = 5 achieves the highest performance.

   Use more features: We add ['year','month','day','hour','minute'] into our training dataset. We derived these attributes from ['created'] attribute.

   Add the unstructured attribute: At this part, we try to use ['features'] as our new features. Firstly, we join the features in the ['features'] into a single string. So we can treat them as a short document, then we use CountVectorizer to create Term Frequency vector for each training object. Lastly, we hstack the new TF vector into the old attributes to create new training objects.

   Accuracy in cross-validation: 0.69488 → 0.69488 (a. tune parameter) → 0.68489 (b. use more features) → 0.65620 (c. add the unstructured data)

   Accuracy on Kaggle: 0.71082 → 0.71082 → 0.70834 → 0.68476

2. **Logistic regression:**

   Before training the classifier, we did the normalization for the data. This would allow all attributes to have similar weights. We used sklearn function 'make_pipline()' to combine the data transformation phase and the model phase. At first we used simple StandardScaler to do the normalization. However when we did feature engineering for improving performances of classifiers, we added information from the 'feature' attribute, and then the data became the form of sparse matrix on which StandardScaler does not work. So we used MaxAbsScaler to replace StandardScaler. MaxAbsScaler is designed for sparse matrix.

   In the first version of logistic regression classifier, we set random_state = 0, multi_class = 'auto', solver = 'sag' (We just randomly chose a solver at this step), and all other parameters were default. We applied 10-fold cross-validation on the classifier and the training dataset, and we used the logloss as the scoring metric. Then we averaged the ten scores got from cross-validation, and we used it to represent the performance of the classifier on the validation dataset. The score of cross-validation at this step is 0.7106723581500316. We trained this model on the training dataset and used it to work on test dataset. The score we got from kaggle website is 0.73383.

We used three different ways to improve the logistic regression classifier. The first one was tuning parameters. We tuned the 'solver' parameter in the function. There are five optional solvers: 'newton-cg', 'lbfgs', 'liblinear', 'sag' and 'saga'. The default solver is 'lbfgs'. However, the 'liblinear' solver is only for two-class problems, our problem is multiclass. Then we only tried the remaining four solvers. We used the same ten-fold cross-validation as before. We found that the four of them had similar performance. The mean logloss of them are: 'newton-cg': 0.710672588, 'sag': 0.710672358, 'saga': 0.710672573, 'lbfgs': 0.710672636. At this stage we still chose solver = 'sag' because it had the lowest negative logloss, so actually it is not different with the first version. The score of test dataset is also 0.73383.

The second improvement strategy is adding more features. At first we just used five features, there should be more information we can get from other features. Form the original attribute 'created', we got 'year', 'month', 'day', 'hour' and 'minute' five new attributes. In addition of adding new attributes, we also tuned parameters at this step, because for the new training data the old set of parameters may be not the best one. We still tried the four different solvers as before, and did the same cross-validation. This time the four of the also had the similar performance, but the best one was changed to 'lbfgs'. We trained this classifier on test dataset, and the score of the test dataset at this step we got from kaggle website is 0.72474. It shows that after this improvement the classifier had better performance.

The third improvement strategy is also adding more features, but this time we added the TF vector of the 'features' attribute. At this step, we also tried different parameters. As the same before, we tried four different solvers, and we used the same cross-validation as before. On the validation dataset, the 'newton-cg', 'sag' and 'saga' solver had the similar performance, while the 'lbfgs' solver had a significant better performance than them. The score of the classifier with solver = 'lbfgs' on the validation dataset is 0.65759. Then we trained it on the training dataset. The score of test dataset we got from kaggle website is 0.68127.

Accuracy in cross-validation: 0.71036 → 0.71036 → 0.71106 → 0.68127

Accuracy on Kaggle: 0.73383 → 0.73383 → 0.72474 → 0.65759

3. **SVM:**

At the beginning, the SVM is so slow that we only set the max_iter = 100. Then the number of iterations which is 100 is too small for the solver to get the converge classification. As a result, the SVM performance is even worse than Logistic regression.

In order to optimization, firstly we try to use different kernel and gamma to run.

Then we find currently 'sigmoid' kernel and when gamma = 1 achieves the highest performance. Secondly, we add [year, month, day, hour, minute] into our dataset. And we derived these attributes from [created] attribute.

At last, we try to use ['features'] as our new features. Firstly, we join the features in the ['features'] into a single string. So, we can treat them as a short document, then we use Count Vectorizer to create Term Frequency vector for each training object. Lastly, we hstack the new TF vector into the old attributes to create new training objects.

Accuracy in cross-validation: 0.77193 → 0.77334 → 0.77129 → 0.73357

Accuracy on Kaggle: 0.79225 → 0.79256 → 0.79062 → 0.77147

4. **Random Forest:**

   Tune Parameters: For the Random Forest, we try with different number of the n_estimators, max_depth and min_samples_leaf.

   Use more features: We add ['year','month','day','hour','minute'] into our training dataset. We derived these attributes from ['created'] attribute.

   Add the unstructured attribute: At this part, we try to use ['features'] as our new features. Firstly, we join the features in the ['features'] into a single string. So we can treat them as a short document, then we use CountVectorizer to create Term Frequency vector for each training object. Lastly, we hstack the new TF vector into the old attributes to create new training objects.

   Accuracy in cross-validation: 0.69819 → 0.69638 (a. tune parameter) → 0.67712 (b. use more features) → 0.58395 (c. add the unstructured data)

   Accuracy on Kaggle: 0.70964 → 0.70499 → 0.69065→ 0.67160

5. **Gradient Boosting:**

   In project milestone3 we used gradient boost classifier. It is an ensemble classifier and a boost classifier. Before training the classifier, we also did the normalization for the data as before. We used StandardScaler at first, and at the last stage we changed to use MaxAbsScaler.

   In the first version of gradient boost classifier, we used all the default settings. We applied 10-fold cross-validation on the classifier and the training dataset, and we used the logloss as the scoring metric. Then we averaged the ten scores got from cross-validation, and we used it to represent the performance of the classifier on the validation dataset. The score of cross-validation at this step is 0.6669975238805609. We trained this model on the training dataset and used it to work on test dataset. The score we got from kaggle website is 0.67689.

   We used three different ways to improve the gradient boost classifier. The first one was tuning parameters.

   There are many parameters that can be tuned in gradient boost classifier. We chose to tune 'n_estimators' and 'learning_rate'. We should try different set of these two parameters, while actually we finally tune each of these two parameters separately. We kept one of them constant and just tuned the other each time. The reason was that the running time of testing many set of different 'n_estimators' and 'learning_rate' was too long. At first we kept learning_rate = 0.1 which is the default learning rate, and we tried four values of n_esimators. According the scikit website, usually large number of estimators would result better performance, because the gradient boost classifier is robust to over-fitting (scikit-learn, 3.2.4.3.5, 2019). We started at the default value of n_esimators (100) and tried 300, 500, 700. We applied the same ten-fold cross-validation on the four classifiers. The result showed that among the four classifiers, higher value of n_esimators had better performance (lower mean logloss). We found that with the increase of n_estimators, the growth of training time of the classifier is around exponential. However with the increase of n_estimators, the decrease rate of mean logloss is decrease. Consider both of the running time cost and the performance, we decided to stop at n_estimators = 700. The score of the classifier with n_estimators = 700 on validation dataset is 0.65061125. Then we trained it on the training dataset and the score of test dataset we got from kaggle website is 0.667742. At next we fixed n_estimators as the default value 100, because the high n_estimators value would cost huge

training time, and we tried four different values of learning_rate: 0.05, 0.1, 0.3, 0.5, 1. We applied the same cross-validation on the five classifiers. The result showed that the classifier with learning_rate = 0.3 had the best performance. The score of it on the valid dataset is 0.6555805. The test score we got from the kaggle website is 0.67350. At this step we found that the classifier with n_estimaters = 100 and learning_rate = 0.3 had better performance than the classifier with n_estimaters = 700 and learning_rate = 0.1. In addition the running time of the former is significant lower than the latter.

The second improvement strategy is adding more features. Form the original attribute 'created', we got 'year', 'month', 'day', 'hour' and 'minute' five new attributes. In addition of adding new attributes, we also tuned parameters at this step. We still used the same strategy to tune the parameter n_estimators and learning_rate, and we did the same cross-validation. As the same before, in terms of the n_estimators, higher number would result better performance. The classifier with n_estimators = 700 had the best performance, and the score of it on the validation dataset is 0.622794. The test score we got from kaggle website of it is 0.63959. In terms of the parameter learning_rate, the classifier with learning_rate = 0.5 had the best performance, and the score of it on the validation dataset is 0.628609. test score we got from kaggle website of it is 0.65110. This time the classifier with n_estimaters = 700 and learning_rate = 0.1 had better performance than the classifier with n_estimaters = 100 and learning_rate = 0.3.

The third improvement strategy is also adding more features, but this time we added the TF vector of the 'features' attribute. At this step, we also tried different parameters. As the same before, we tried different values of n_estimators and learning_rate separately. In terms of the n_estimators, higher number would still result better performance. The classifier with n_estimators = 700 had the best performance, and the score of it on the validation dataset is 0.580790. The test score we got from kaggle website of it is 0.59758. In terms of the parameter learning_rate, the classifier with learning_rate = 0.5 had the best performance, and the score of it on the validation dataset is 0.590546. test score we got from kaggle website of it is 0.61673. This time the classifier with n_estimaters = 700 and learning_rate = 0.1 had better performance than the classifier with n_estimaters = 100 and learning_rate = 0.3.

Accuracy in cross-validation: 0.65717 → 0.60055 → 0.56266 → 0.52216

Accuracy on Kaggle: 0.67689 → 0.67742 → 0.63959 → 0.59758

6. **Naïve Bayes classifier:**

   To optimize this classification, we only use add [feature] as our new feature. Because it does not have parameter need to tune, the priors can calculate from training data distribution. To add the feature, we join the features in the [feature] into a single string at first. So, we can treat them as a short document, then we use Count Vectorizer to create Term Frequency vector for each training object. Lastly, we hstack the new TF vector into the old attributes to create new training objects.

   Accuracy in cross-validation: 0.81315 → None parameter tuning → 0.83428 → 0.84212

   Accuracy on Kaggle: 0.88510 → None parameter tuning → 0.90537 → 0.86621

7. **XGBoost classifier:**

   After the project milstone2 and milstone3 we read some discussions on the kaggle website. We found that many people mentioned the XGBoost classifier, so we decided to try it. The full name of XGBoost is extreme gradient boost. It is an advanced implementation of gradient boost.

Compared with gradient boost classifier, the XGboost classifier has regularization which helps it to deal with overfitting (Aarshay Jain, 2016). We used the library xgboost for XGboost classifier.

In the first version of XGboost classifier, we used all the default settings. We found that in our project milestone 2 and milestone 3 nearly all the classifiers, especially the gradient boost classifier which is most similar to XGboost classifier, had better performance after we add more attribute form the attribute 'created' and 'features', so we decided to start at this stage for XGboost classifier. Before training the classifier, we also did the normalization for the data as before. We used MaxAbsScaler. his time we applied five-fold cross-validation on the classifier and the training dataset, and we use the same way to represent the performance of the classifier on the validation dataset. The score of cross-validation at this step is 0.58393. The test score we got from kaggle website is 0.59619. This shows that the default XGboost classifier has the best performance compared with the six classifiers in milestone 2 and milestone 3.

We then started to tune the parameters of XGboost classifier. As same as gradient boost classifier, the XGboost classifier also has a lot of parameters. We had the experience of tuning n_estimators and learning_rate of gradient boost classifier, so we decided to start at this. However we knew that tuning this two parameters especially n_estimators needed a lot of time, so we tried to directly train it on training set and check the test score on kaggle website. We tried some combo of n_estimators and learning_rate and could not find a better one than the default one. At next, we keep all other parameters as default and tuned the max_depth and min_child_weight, because they have the biggest influence on the performance (Aarshay Jain, 2016). We tried all combo of max_depth = 4, 6, 8 and min_child_weight = 1, 3, 5. We used the same ten-fold cross-validation on the training dataset, and the result showed that the classifier with max_depth = 6 and min_child_weight = 3 had the best performance. It's mean logloss is 0.57921. The test score we got from kaggle website is 0.59380.

Accuracy in cross-validation: 0.58393 → 0.57921 (final improvement)

Accuracy on Kaggle: 0.59619 → 0.59380

Comparison of the results of the different methods:

In milestone 3, we try the NB classifier which is easy to use but have the bad performance and ensemble classifiers (Random Forest and Gradient Boosting) which needs more time to train but have a better performance. Compare with the milestone 2 and XGBoost classifiers performance, we get this rank: XGBoost > Gradient Boosting > Logistic Regression > Random Forest > Decision tree > SVM > NB.

Gradient Boosting and Logistic Regression achieve the top log loss because both of them have the ability that use mistake of current model to improve the next model.

Random Forest performance better than Decision tree is because it can reduce the effect of variance on a single decision tree.

The gradient boost classifier finally had the best performance (0.59758 of test score) among the six classifiers in the project milstone2 and milstone3 after all the improvements. The reason would be discussed specifically in the question four.

Comparing with other classifier, the SVM has a very high accuracy if it has large enough iteration during the training. However, the run time is also very long. During the test, if our iteration set to 1000, we may run a few days. So we set the iteration only 100 in milestone 2 which is so early for

a SVM classifier converge.

Nevertheless, the SVM classifier performance is still better than NB because NB needs an assumption that attributes are conditional independent given class label. But, in our selected features, 'bathrooms', 'bedrooms' and 'price' have some dependence on each other. It is common that more 'bedrooms will have more 'bathrooms' also 'price' will be higher because more people can stay at the same time. So, the conditionally independent assumption doesn't hold, and the performance is bad.

But on the other hand, NB training time is much less than SVM because the algorithm itself is very simple to run.

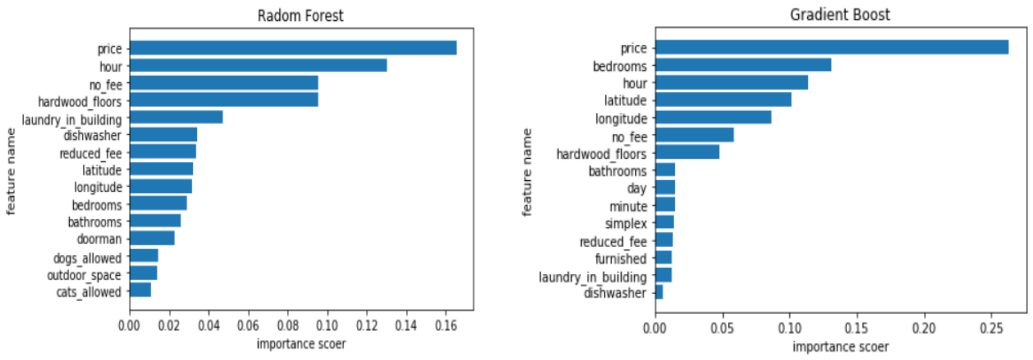
Compared with gradient boost classifier, XGboost classifier has better performance. The best test score of gradient boost classifier is 0.59758 and the best test score of XGboost is 0.59380. In addition, the training time of the XGboost classifier with best performance is significant shorter than the training time of the gradient boost classifier with best performance.

4. Lessons learnt:

1. Which features were most relevant, and why?

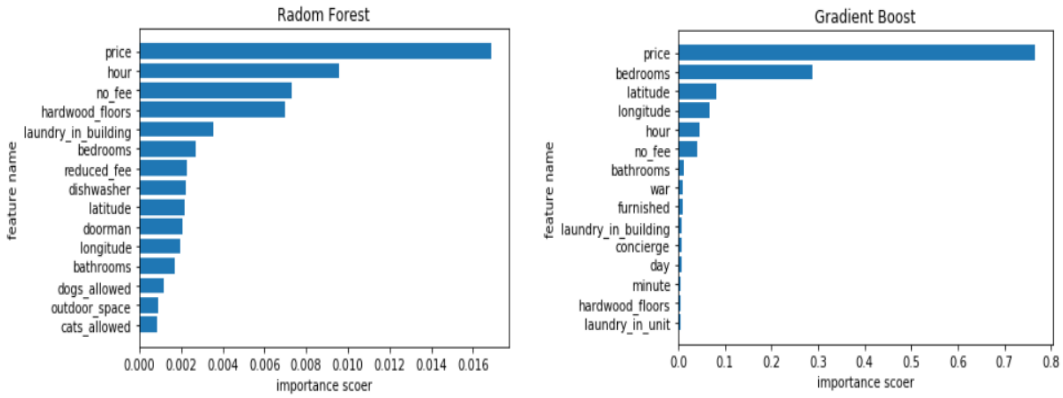
The most relevant feature is price. We used two methods to find which feature is the most relevant. We tried these two methods on the training dataset which was obtained from the last step of our feature selection. The training dataset contained 260 features in total. The first five features were 'bathrooms', 'bedrooms', 'latitude', 'longitude' and 'price'. The next five attributes were got from the original feature 'created'. The remaining features were got from the original feature 'features'.

The first method we used to find the feature importance was the sklearn default function 'feature_importances'. This method was simple and fast. It allowed us to get feature importance score for each feature from one trained classifier. We applied this method on both the best random forest classifier and the best gradient boost classifier. Both of the two result showed that the price had the highest feature importance score. The two graphs below shows the other most relevant features with their importance score.



The second method we used was permutation feature importance. We used python library sklearn for permutation feature importance. The basic idea of this method is to use one trained classifier on the dataset in which values of one chosen feature have been reshuffled to get the prediction score. The difference of the new prediction score and the old prediction score can

represent the importance of the chosen feature (scikit-learn developers, 2007-2019). We applied this method on both the best random forest classifier and the best gradient boost classifier. As the same before both of the two result showed that the price had the highest feature importance score. The two graphs below shows the other most relevant features with their importance score.



In conclusion, for the most relevant fifteen features, the two methods for the same trained classifier had similar results, while the same method for different trained classifier had different results. Combine the result from all the results, we found that the most relevant features are price, hour, bedrooms, latitude and longitude.

2. **If you created additional features from external datasets, how much did they help?**

   None.

3. **Which classifiers worked best, and why?**

   The classifier has the best performance is xgboost classifier. It is an advanced implementation of the gradient boost classifier, and the mechanism of this classifier is the same as gradient boost classifier. In addition, the gradient boost classifier had the best performance among the six classifiers in our project milstone2 and milstone3, and the performance of gradient boost classifier was just a little lower than xgboost classifier. Gradient boost classifier is an ensemble classifier, and it uses the boosting skill. We think it is the main reason why the xgboost and gradient boost classifier had the best performance. Boosting is a skill to reduce bias. Classifiers would make assumptions about the dataset, and if the dataset cannot match theses assumptions, classifiers would not make precise predictions. The gradient boost classifier would train and validate the same model on a training dataset many times, and in each round it would increase the weight of the misclassified examples. It assumed that the bias is form the model and it reduces the bias by this method. The result that xgboost classifier had the best performance showed that other classifiers we used had high bias in terms of the training data.

4. **Which classifier were more efficient to train?**

   Naïve Bayes classifier is the most efficient to train. Training is fast because only the probability of each class and the probability of each class given different input (x) values need to be calculated. No coefficients need to be fitted by optimization procedures.

   The class probabilities are simply the frequency of instances that belong to each class divided by the total number of instances.

   The conditional probabilities are the frequency of each attribute value for a given class value divided by the frequency of instances with that class value.

5. **Was overfitting a problem? If so, how did you address it?**

   We use cross-validation and try different parameters to avoid overfitting. We compare the logloss on the training dataset and the test dataset, if the test dataset logloss is much bigger, it has overfitting. (For example, we have tried decision tree with depth = 5 and depth = 10 at the second improvement. The logloss of depth = 5 is (0.68489 vs 0.70843) and for depth = 10 is (0.59077 vs 1.33995). So the difference of depth = 10 is too big, it is overfitting.). So, before we actually train the model, we try the cross validation on different parameters first to make sure we do not overfitting the model.

5. Recommendations for rental property owners:

1. **When planning a new rental property, what properties should it have to maximize the interest of potential renters?**

   As we discussed before, the most relevant features are price, hour and bedrooms, so we tried to analyze these three features at first. We printed the percentage of the number of examples with low, medium and high interest level in terms of each number of hours (0 to 23), each number of bedrooms (0 to 10) and each thousand of price. In terms of price, we found that with the increase of price, the percentage of low interest level would increase and the percentage of medium and high interest level would decrease. For bedrooms, two to four bedrooms had relatively lower percentage of low interest level and higher percentage of medium and high interest level. For hours, information created between 9:00 pm to 0:00 am had relatively lower percentage of low interest level and higher percentage of medium and high interest level. In additional, by analyzing the 'feature' attribute, we found the feature 'no_fee', 'hardwood_floors', 'laundry_in_building', 'dishwasher', 'doorman', 'outdoor_space', 'dogs_allowed' and 'cats_allowed' had higher importance score among total 250 features. So we thought in order to maximize the interest of potential renters, the rental property should have low price, two to four bedrooms, information created at night and the features we mentioned above.

2. **When promoting an existing rental property, which properties should one highlight?**

   For an existing rental property, many features that have the most significant ability to maximize the interest of potential renters cannot be changed. With this in mind, the first strategy to promote an existing rental property is decrease the price. However this strategy would decrease the interest of property holders. The second strategy is to avoid publishing the rental information at 1:00 am to 6:00 am. We found that most example created at this period had low interest_level. The third strategy is to equip the property with hardwood floor and dishwasher, and the property owner can also attract potential renters by allow cats and dogs. These features are more cared by general renters compared with other features.

6. References:

[1] Support Vector Machines¶. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/svm.html

[2] Naive Bayes¶. (n.d.). Retrieved from https://scikit-learn.org/stable/modules/naive_bayes.html

[3] Logistic regression: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[4] GradientBoost classifier: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

[5] XGBoost classifier: https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/
XGBoost classifier: https://shirinsplayground.netlify.com/2018/11/ml_basics_gbm/
[6] Feature importance: https://towardsdatascience.com/explaining-feature-importance-by-example-of-a-random-forest-d9166011959e
[7] Feature importance: https://scikit-learn.org/stable/modules/permutation_importance.html
[8] Naïve Bayes: https://machinelearningmastery.com/naive-bayes-for-machine-learning/

7. Section author: Dekai Lin (1 2 3 4 6) Zherui Shao (3 4 5 6) Luowen Zhu (3 4 6)