2020
Spring

CMPT 459 Final Report

FlyingPiggy

Jinze Wu
Yizhou Chen
Luoxi Meng

Content

CMPT459 Final Report

1. Exploratory data analysis

1.1 Distributions for the most important attributes

In this subsection, we provide distribution histograms of 6 variables (no outliers): *price, longitude, latitude, bedrooms, bathrooms, interest_level*. The plots in this part will help us to build a basic idea of these three variables.

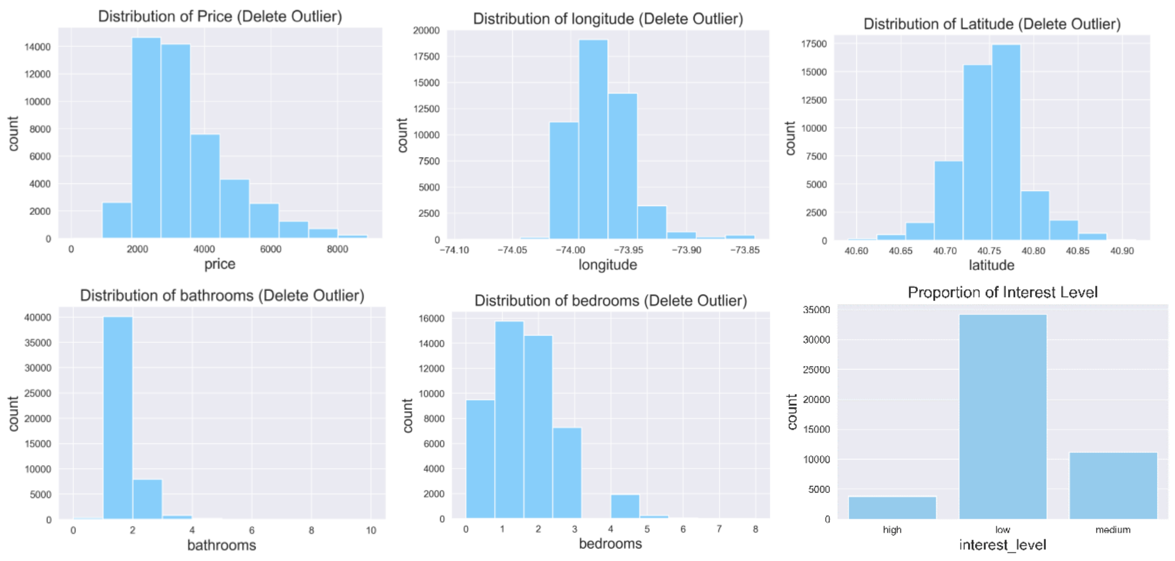


Figure 1 Distributions for the most important attributes

1.2 Listing trends over time

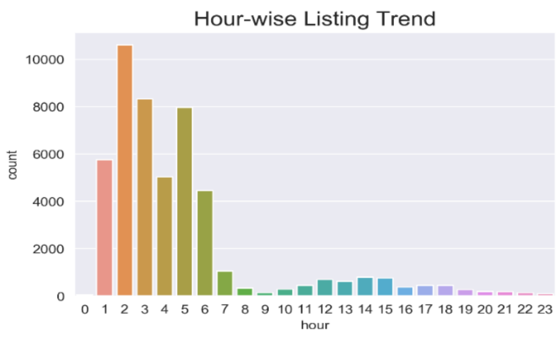


Figure 2 Listing trends over time

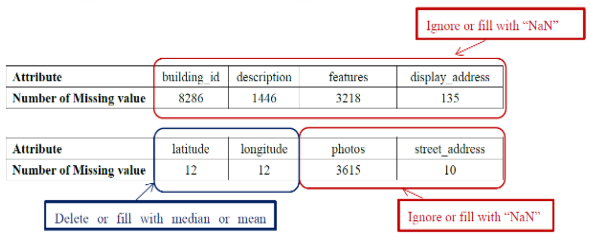
2. Data pre-processing

2.1 Dealing with missing values, outliers

2.1.1 Dealing with missing values

- If the missing data is a class label (like **interest_level** in this dataset), we can drop that data, because it will not help us in classification.
- If the data are attributes like **building_id**, we can fill it with "unknown" or "NaN", because this type of data, which is used for identifying the object we are analyzing (in this case, house) helps little in classification.
- If the data are categorical, boolean or numerical data that mean a lot to our classification, we will use the most probable value to fill the missing data (regression method). For example, we can try to figure out the missing price according to the number of **bedrooms** and **bathrooms** of this listing. If the most probable value is hard to obtain, we can use the attribute mean or median to fill the missing value, like the case of longitude and latitude.

- As for other types of missing values, like **photos** and **descriptions**, we can leave it to feature extraction first and deal with it after robust features are extracted. In most cases, it is reasonable that some houses have no photo or description, so we can ignore them as well.



Attribute	building_id	description	features	display_address
Number of Missing value	8286	1446	3218	135

Attribute	latitude	longitude	photos	street_address
Number of Missing value	12	12	3615	10

Ignore or fill with "NaN"
Delete or fill with median or mean
Ignore or fill with "NaN"

2.1.2 Dealing with outlier values

- **Tukey's test**
  Calculate the first quartile Q1,the median Q2, the third quartile Q3, and let interQuartile range IQR be |Q3-Q1|. Then define the values either Q1-3IQR or above Q3+3IQR as extreme outliers, the values either below Q1-1.5IQR or above Q3+1.5IQR as mild outliers.
  In this part, we tried Tukey's test to help to find a proper range to define outliers in different attributes. We only detect outliers for quantifiable attributes： Bathrooms, Bedrooms, Price, Latitude, Longitude. We remove all outliers.
- **Bathroom & Bedroom**
  We thought that when dealing with the outlier of the bathroom and bedroom, they should be considered at the same time. So we set a variable for the difference between bedroom and bathroom. Then, using the outlier function, we found the Outlier(bedrooms & bathrooms) = 616.
- **Price**
  By the outlier function, we found the number of price outliers was 1223, whose price was higher than 8900. However, by observing the price histogram, we thought that it was better to set the price greater than 20000 as the outlier. Therefore, the Outlier(Price) = 109.
- **Latitude & Longitude**
  By the outlier function, we found Outlier (Latitude) = 147 and Outlier (Longitude) = 411.

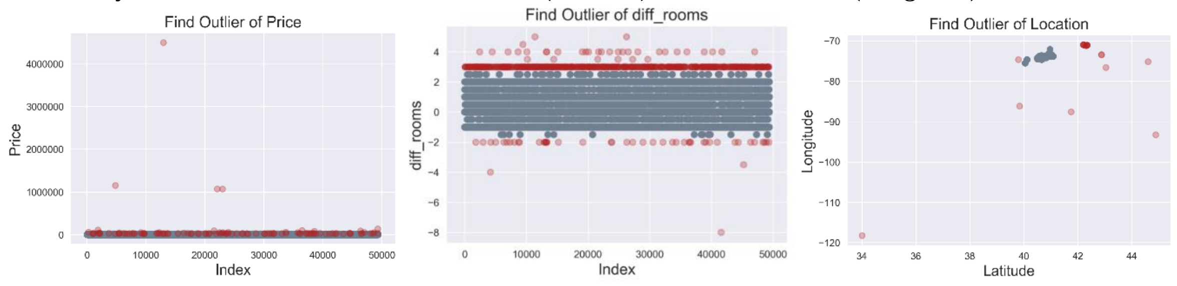


Figure 3 Outlier detection

2.2 Feature Extraction

2.2.1 Images Feature Extraction

- **Color histogram**

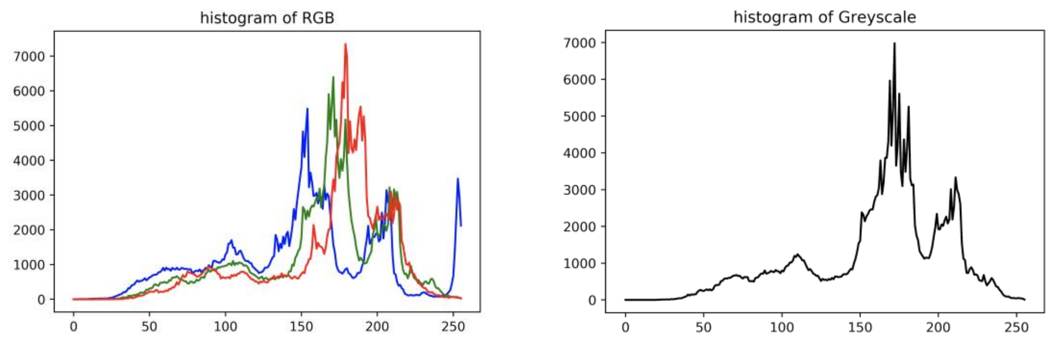


Figure 4 Image Histogram

We calculated the histogram of RGB and grayscale with cv2.calcHist() method. Then we extracted the vectors with some important statistical quant including mean and standard deviation as new features.

- **Edge Detection with Prewitt Kernel**

We could detect edges by locating value changes in the pixel table (for example, grayscale or RGB). Prewitt kernel uses values surrounding the selected pixel and multiply it with the kernel, then sum up to get a final value for this pixel. This could be achieved by using prewitt_h and prewitt_v, two filter functions in skimage packet. After the work, we can choose an image and find its edges by locating the data in the table.

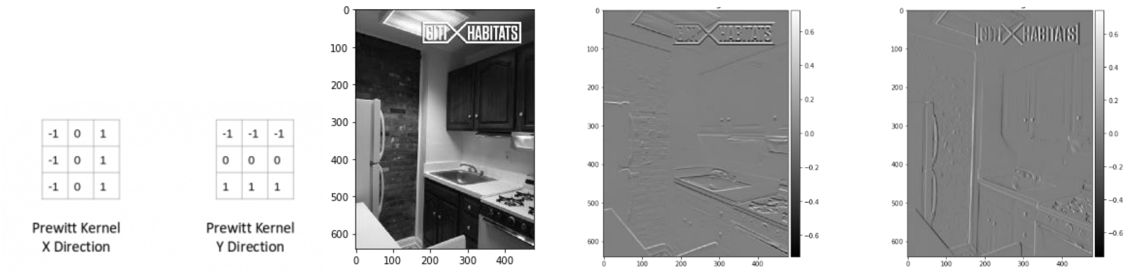
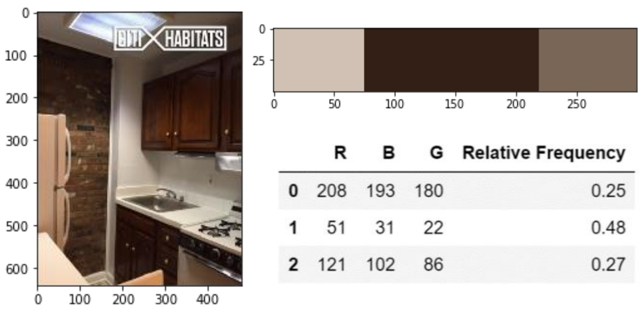


Figure 5 Prewitt Kernel and an example of edge detection

- **Find Primary Colors**

To find the primary colors, we apply **k-means** clustering. This algorithm separates the original n data into k clusters. Data in the same cluster are regarded as "more similar" than data outside this cluster. So, it basically takes 2 steps to find the primary colors: first, cluster the RBG data of an image; and second, calculate relative frequency for each cluster.
We present the result with a color bar and a table of relative frequency of each primary color.
After the work, we can locating values in the same place of the 2 tables, which is a pair of (color, relative frequency), and then use the function *plot_colors* to see the three primary colors and their relative frequency of this image.

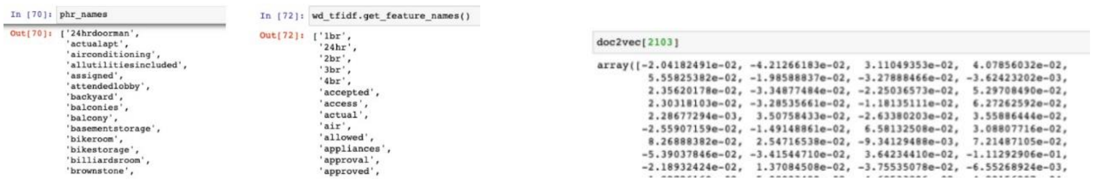


	R	B	G	Relative Frequency
0	208	193	180	0.25
1	51	31	22	0.48
2	121	102	86	0.27

Figure 6 An example of finding primary colors

2.2.2 Text Feature Extraction

- **Cleaning**
  For *description*: We used regular expressions to replace the acronyms and symbols in *description*.
  For *features*: Because most of the features exist in the form of phrases, we first process phrases to ensure their integrity, such as turning 'cat allowed' into 'catallowed'.
- **TFIDF vector**
  TF - Term Frequency, IDF - Inverse Document Frequency. We used the Tfidfvector() method in the scikit-learn library to calculate the TFIDF value of a word in the whole corpus, and we selected the most important 80 words to be the new features.
  phr_tfidf = TfidfVectorizer(min_df=2, max_features=200, strip_accents='unicode', lowercase=True, token_pattern=r'\w{3,}', stop_words='english')



```
In [70]: phr_names
Out[70]: ['24hrdoorman',
          'actualapt',
          'airconditioning',
          'allutilitiesincluded',
          'assigned',
          'attendedlobby',
          'backyard',
          'balconies',
          'balcony',
          'basementstorage',
          'bikeroom',
          'bikestorage',
          'billiardsroom',
          'brownstone',
```

```
In [72]: wd_tfidf.get_feature_names()
Out[72]: ['1br',
          '24hr',
          '2br',
          '3br',
          '4br',
          'accepted',
          'access',
          'actual',
          'air',
          'allowed',
          'appliances',
          'approval',
          'approved',
```

```
doc2vec[2103]

array([-2.04182491e-02, -4.21266183e-02,  3.11049353e-02,  4.07856032e-02,
        5.55825382e-02, -1.98588837e-02, -3.27888466e-02, -3.62423202e-03,
        2.35620178e-02, -3.34877484e-02, -2.25036573e-02,  5.29708490e-02,
        2.30318103e-02, -3.28535661e-02, -1.18135111e-02,  6.27262592e-02,
        2.28677294e-03,  3.50758433e-02, -2.63380203e-02,  3.55886444e-02,
       -2.55907159e-02, -1.49148861e-02,  6.58132508e-02,  3.08807716e-02,
        8.26888382e-02,  2.54716538e-02, -9.34129488e-03,  7.21487105e-02,
       -5.39037846e-02, -3.41544710e-02,  3.64234410e-02, -1.11292906e-01,
       -2.18932424e-02,  1.37084508e-02, -3.75535078e-02, -6.55268924e-03,
```

Figure 7 Result of TFIDF and doc2vec vector

- **doc2vec vector**
  Since there are sentences in *description*, so the context of a word is also important. We used the doc2vec() method in the gensim library to calculate a vector for each document. The doc2vec model can reflect the similarity between documents.
  doc2vec=Doc2Vec(taggedDoc, vector_size=120, min_count=20, window=4, negative=5, sample=1e-5, workers=4)

2.3 Additional Feature

Feature: Distance to the nearest subway station
Process:

1) Download the New York's Subway dataset from NYC Open Data
2) Using haversine formula, Custom function and pandas to compute the distance from the Geographical coordinates
3) Add the distance values to the original dataset as a new feature.

2.4 Feature Selection

We used three functions to evaluate the features, including ExtraTree, SelectKBest-F-score, SelectKBest-Mutual-Information. We found that SelectKBest-MI performed best. For different classifiers, we chose different number of features.

3. Classifiers

3.1 Logistic Regression - from sklearn.linear_model import LogisticRegression

3.1.1 Optimizations

- **Tune the parameters**
  I tried different ways to deal with multi-class and different solvers to deal with the gradient iteration. I also tuned the regularization term. Besides, I tried to remove the balanced class-weight and increase the maximal number of iterations:
  - Remove *class_weight='balanced'*
  - Change the method to deal with multi-class classification *multi_class='multinomial'*
  - Change the solver of the gradient ascent iteration *solver='sag'*
  - Tune the parameter *C=10.0*
  - Increase the maximum iterations *max_iter=500*

  LogisticRegression(penalty='l2', C=10.0, solver='sag', multi_class='multinomial', max_iter=500)
- **Modify the dataset:**
  For the Tfidf and Doc2Vec vectors:
  - change the length of the vectors
  - slightly adjust several other parameters
- **I tried to several methods to combine the three features or the results of the three classifiers:**
  - Add a few feature words extracted manually from features and descriptions to the original attributes. If the word exists in this record, then set the corresponding attribute value to be 1, otherwise 0.
  - Set the probabilities obtained from the three original classifiers as new attributes (9 attributes in total), and do logistic regression classification on the new 9 attributes.
  - Turn the Tfidf vectors of features and descriptions into attributes (number of attributes equals to the length of the vector) and let the TFIDF value be the attribute value, combine them with the original attributes.

3.1.2 Accuracy and score

	Score of Validation	Score of Kaggle
Improved Result	0.69710	0.69609

3.2 XGBoosting - from xgboost.sklearn import XGBClassifier

3.2.1 Optimizations

- **Use *GridSearchCV* to tune the parameters:**
  - Param_test1 = { '*max_depth*': [3,5,7,9], '*min_child_weight*':[1,3,5] } - got (9, 5)
  - Param_test2 = {'*max_depth*': [8,9,10], '*min_child_weight*':[4,5,6] } - got (9, 6)
  - Param_test3 = {'*gamma*': [i / 10.0 for i in range(0,5)] } - got gamma=0.1
  - Param_test4 = {'*subsample*':[i / 10.0 for i in range(6,10)], '*colsample_bytree*': [i / 10.0 for i in range(6,10)] } - got (0.9, 0.7)
- **Use *xgb.cv* to find the best *n_estimater*:**

    - Reduce *learning_rate* to 0.01
    - Increase *n_estimators* to 800
- **Modify Dataset:**
    - Use Extra-Trees to select the most important features
    - Add additional Features *distance_to_sub*
- **Model fusion**
    - Use the trained XGBoosting to extract features, and then do Logistic Regression
    - Let XGBoosting and RandomForeset be base classifiers and Logistic Regression be meta classifier and apply stacking

3.2.2 Accuracy and score

	Score of Validation	Score of Kaggle
Improved Result	0.74860	0.57766

3.3 Decision Tree - from sklearn.tree import DecisionTreeClassifier

3.3.1 Optimizations

- **Add new feature: distance to the nearest subway station, 20 most important TFIDF of features and descriptions(10 features, 10 descriptions)**
  Before I added new data, I tried to adjust parameters several times and used cross validation and kaggle to evaluate the performance of the classifier. I found that it has been upgraded to the best, but I think the result can be better, so I added some new data. I found the subway station data of New York City on the Internet and used python to calculate the distance from the nearest subway station to each house. Besides, I deleted the manually selected text data.
- **Use 5-Fold cross-validation to tune the parameter of decision tree**
  Because the cross validation of 10 fold took a lot of time, we used the cross validation of 5-fold to adjust the parameters and find the parameters that could make the cross validation result the best. We made min_impurity_decrease=0.0007 and max_leaf_nodes=80 and min_sample_leaf = 5.
- **Use PCA(Principal components analysis) to optimize features**
  I used principal component analysis to construct the data of six components, and I found that it greatly improved the operation speed and the results were better than before.

3.3.2 Accuracy and score

	Score of Validation	Score of Kaggle
Improved Result	0.7053	0.6910

3.4 Random Forest - from sklearn.ensemble import RandomForestClassifier

3.4.1 Optimizations

- **Process the dataset**
  Since we have added some features with high correlation before, we mainly did some data cleaning in this step. Due to the addition of some text vector data, we need to delete some data with low correlation degrees to prevent the poor fitting of the decision tree(estimator). Therefore, we used ExtraTree, F-score and SelectKBest to select new features.
- **Use PCA(Principal components analysis) to optimize features**
  I used principal component analysis to construct the data of 30 components, and I found that it greatly improved the results.
- **Tune the parameter of RandomForestClassifier**
  We used log_loss as the criteria to evaluate the result of prediction. Firstly, we identified an n_estimators that would stabilize the results, then we used a method similar to GridSearchCV to find the optimal parameters (two layers of ‘For Loops). We found that when the max_depth = 21 and min_samples_split = 40, the tree had the best performance.

3.4.2 Accuracy and score

	Score of Validation	Score of Kaggle
Improved Result	0.6247	0.6228

3.5 Support Vector Classifier (SVC) – from sklearn.svm import SVC

3.5.1 Optimizations

- **Decrease the number features** from tfidf from 300 to 40, and **drop those highly correlated features.**
  I evaluate the correlation of each 2 features with items in X.corr() and drop the features that are highly correlated to another one. Additionally, with the library statsmodels.api I evaluate how a feature helps to the regression of the result of y_train and the drop those insignificant features (though this idea was not implemented finally, resulting from the observation that it might lead to overfitting).
- **Apply PCA to reduce the data into main components.**
  PCA helps to reduce the noise of data, and make the implementation more robust. I use 6 components in this case. And this greatly improve the efficiency in the model training process.
- **Use GrideSearchCV to find the best optimal values for parameters.**
  GridSearchCV is a method from sklearn.model_selection that provides an approach to do the exhaustive search with all the combinations of the parameter in a "parameter grid". I use this method and set the "scoring" parameter of it to "accuracy". Then all the parameter combinations in the grid are evaluated by their accuracy with 5-fold cross validation. Then we can find the best parameters.

3.5.2 Accuracy and score

	Score of Validation	Score of Kaggle
Improved Result	0.695	0.7516

3.6 Gradient Boosted Decision Tree (GBDT)

– from sklearn.ensemble import GradientBoostingClassifier

3.6.1 Optimizations

- **Dataset modification - apply PCA to reduce the data into main components.**
  PCA helps to reduce the noise of data, and make the implementation more robust. To set the number of principle components, there is a trade-off between the robustness of a small number of components and the reservation of variance of original features. I plot a figure to find an optimal value.

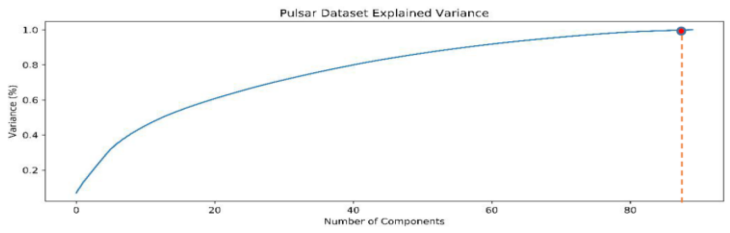


Figure 8 Finding the optimal value for PCA components

According to the figure, I choose n_components=90.
Compared with the classification without PCA, we can say for sure that PCA transformation helps to improve buth the efficiency and accuracy of classification

- **Parameter Tuning**
  Tool: the parameter tuning process is based on the functionality of GridSearchCV(), which applies an exhaustive search on the parameters provided in a parameter grid.
  Order of Parameter Tuning
  We tuning the parameter in the descending order of its impact on the model, which means:
  1) Tune n_estimators, learning_rate
  2) Tune max_depth, min_samples_split
  3) Tune min_samples_leaf
  4) Tune max_features

  We fix the value with a higher impact on outcome first, and then fix its value to tune other parameters. Finally, we increase the n_estimator and decrease learning_rate proportionally to improve robustness.

3.6.2 Accuracy and score

	Score of Validation	Score of Kaggle
Improved Result	0.7374	0.6651

3.7 Comparison of the results of the different methods

3.7.1 Three different ensemble classifiers (XGB\GBDT\RF)

Performance comparison:

	Score of Validation(Accuracy)	Score of Kaggle
XGB	0.7486	0.5776
GBDT	0.7374	0.6651
Random Forest	0.7242	0.6228

According to the table, the performance of the XGBoosting classifier is best on both cross validation and Kaggle. Besides, the random forest classifier has better score on Kaggle than GBDT, but worse score on the validation dataset than GBDT. We would like to compare these three classifiers and explain the performance of the classifiers.

Analysis:

- **RF** vs **Boosting Method (GBDT/XGBoosting)**
  Random Forest is based on the bagging method, whereas, XGBoosting and GBDT are based on the boosting method. Therefore, Random Forest implements random sampling with replacement, where each sample has the same weight. However, the boosting methods (both GBDT and XGBoosting) learns a sequence of classifiers on a weighted training dataset. Weights of round *t+1* depend on weights of round *t*. The Random Forest classifier uses a grown decision tree as base estimators which has high variance and low bias. On the other hand, XGBoosting and GBDT use shallow decision trees as base estimators which have high bias and low variance. Random Forest improves performance by reducing model variance, and boosting methods improve performance by reducing model bias. Random forest is not sensitive to outliers, GBDT is more sensitive to outliers. In conclusion, Random Forest can train the model faster, but the classification results may be worse than XGB.
- **XGB vs GBDT(Gradient Boosting Decision Tree)**
  Both XGBoosting and GBDT follow the principle of gradient boosting. However, the difference lies in details of modeling. Specifically, XGBoosting used a more regularized model formalization to control over-fitting for samples with missing values. Besides, XGBoost is quite memory-efficient and can be parallelized. In conclusion, XGBoosting has better performance than GBDT.

3.7.2 Random forest vs decision tree

Performance comparison:

	Score of Validation(log loss)	Score of Kaggle
Decision Tree	0.7053	0.6910
Random Forest	0.6247	0.6228

According to the table, the performance of the Random Forest classifier is better than Decision Tree. And performance of RF is quite stable.

Analysis:

- There is no overfitting problem in Random Forest classifiers.
- Random Forest Classifier provides a more reliable feature importance estimation.
- The grown decision trees have high variance, but low bias. Random Forest introduces randomization into the split selection. Thus, RF perfectly addresses the overfitting problem.
- Because of all the above reasons, the Random Forest classifier helps to get the accurate classification results effectively.

3.7.3 SVC vs LR

Performance comparison:

	Score of Validation(Accuracy)	Score of Kaggle
SVC	0.7053	0.7516
LR	0.6971	0.6960

According to the table, the performance of LR is better than the performance of SVC. Besides, we found that the LR's training time is greatly less than the SVC's.

Analysis:

Both Logistic Regression and SVM find a hyperplane for the training data that best classifies the data. Logistic Regression assumes a probabilistic model, and maximizes the likelihood of the training data. Support Vector Machine (SVM) maximizes the margin between classes. Here, we use 'rbf' kernel in SVC. Since the training of SVC is quite time-consuming, we do not have enough time for further tuning the 'kernel' parameters and figure out the best one, which may be the main cause for the performance loss. Meanwhile, Logistic Regression is quite robust to overfitting in practice.

4. Lessons learnt

4.1 relevant features

We measure the features importance in our best classifier, XGBoosting Classifier. And here is the results:

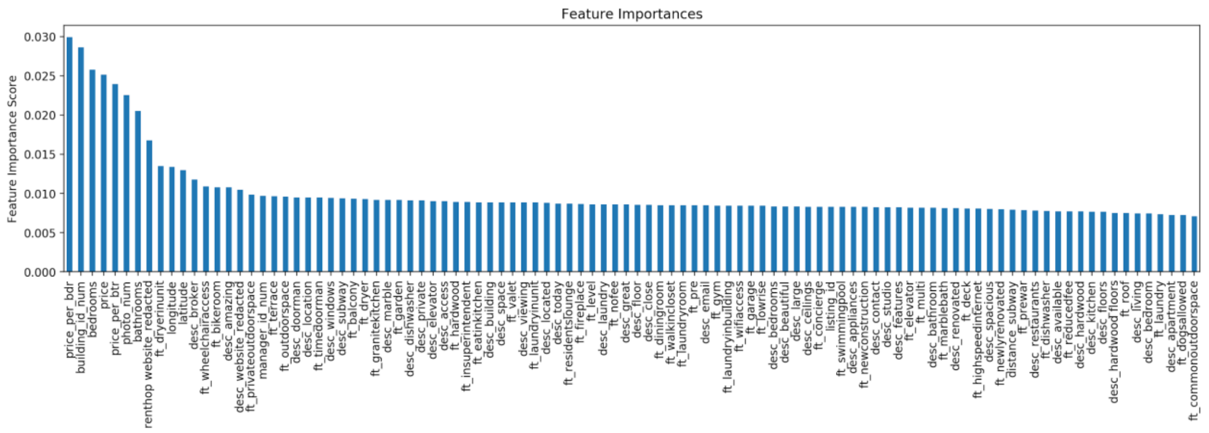


Figure 9 Feature Importance in XGBoosting Classifier

- Importance > 0.02: *price_per_bdr, building_id_num, bedrooms, price, price_per_btr, photo_num, bathrooms*
  The price per bedroom and number of bedrooms, price per bathroom and number of bathrooms, and total price have over 0.02 important score and are the most relevant ones. This is reasonable because demand and affordability are the most critical factors when choosing a house. *price_per_bdr* and *price_per_btr* reflect the price-performance ratio better. *photo_num* is also important, since the photos are the most intuitive method to display a house online.
- 0.02 > Importance > 0.01: *desc_kagglemanager renthop website_redacted, ft_dryerinunit, longitude, Latitude, desc_broker, ft_wheelchairaccess, ft_bikeroom, desc_amazing, desc_website_redacted*
  Longitude and latitude are also relevant since they show the location of the house. Some feature extracted from features and description are also relevant. Dryer in unit, wheel chair access and bike room are three important elements closely related to the life quality. Using 'amazing' in description also matters. The *website_redacted* may show the attitude of the owner and the quality of the advertisement. Besides, having a broker may also help win trust from customers.
- 0.01 > Importance > 0.95: *ft_privateoutdoorspace, manager_id_num, ft_terrace, ft_outdoorspace, desc_doorman*
  Private outdoor space, terrace, and doorman are also relevant factors. *manager_id_num* shows the identity of the manager, and people may prefer managers with good reputation.

4.2 Additional features from external datasets

We add the feature 'subway distance', which is the distance from the house to the nearest subway station. We found the New York's subway dataset from the Internet and used python to calculate the distance. It really helped improve the performance a lot. For example, when we did logistic regression, we didn't modify anything and added the distance feature, the score on the Kaggle was improved from 0.69647 to 0.69609.

4.3 Best Classifier - XGBoosting

The **XGBoosting** Classifier worked best. And here is the reason:

- **Regularization:** XGBoosting has in-built L1 (Lasso Regression) and L2 (Ridge Regression) regularization which prevents the model from overfitting.
- **Handling Missing Values:** XGBoosting has an in-built capability to handle missing values. When XGBoosting encounters a missing value at a node, it tries both the left and right hand split and learns the way leading to higher loss for each node.
- **Cross Validation:** XGBoosting allows users to run a cross-validation at each iteration of the boosting process and thus it is easy to get the exact optimum number of boosting iterations in a single run.
- **Column Sampling:** XGBoosting could do column subsampling to reduce overfitting problems and save computing time.

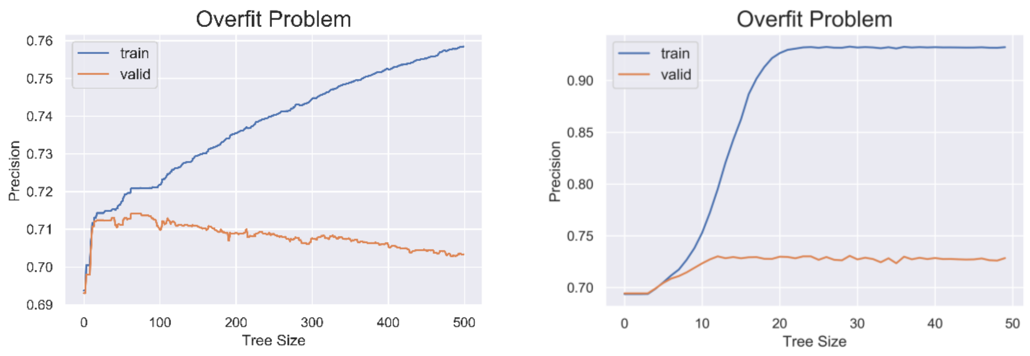
4.4 Most Efficient Classifier – Random Forest

The meaning of efficient is achieving maximum productivity with minimum efforts or expenses. We would like to calculate the ratio of the score on the Kaggle and the training time of the classifiers to find the most efficient classifier.
Here is the score and training time of classifiers:

	DT	SVM	LR	RF	XGB	GBDT
Score(Kaggle)	0.6910	0.7516	0.6961	0.6228	0.5777	0.6651
Time(sec)	0.9400	2218.0000	96.6000	67.3000	3453.0000	953.0000
1/Score	1.4472	1.3305	1.4366	1.6057	1.7311	1.5035
1/Time	1.0638	0.0005	0.0104	0.0149	0.0003	0.0010
log(1/score)	0.1605	0.1240	0.1573	0.2057	0.2383	0.1771
log(1/time)	0.0269	-3.3460	-1.9850	-1.8280	-3.5382	-2.9791

Because we couldn't find a standard for efficient classifiers, we tried to calculate the most efficient classifier by our own ideas. We found that the training time consumption of SVM, XGB and GBDT were much longer than the other three classifiers, therefore, we firstly excluded these three classifiers. Then, for Decision Tree, Logistic Regression and Random Forest, it was easy to find that Random Forest classifier used the less time and had better performance. Although the training time of decision tree is negligible, its performance was far inferior to Random Forest classifier.
In conclusion, we thought the most efficient classifier was **Random Forest** classifier.

4.5 Overfitting Problem and Its Solutions



a) Decision Tree **b) Random Forest**

Figure 10 Precision vs Tree Size (for Decision Tree)

Yes. We observe overfitting on all the classifiers except for Logistic Regression when comparing the scores on train data with the scores on test data. Take Decision Tree as an example. From the above figure a), we find that when the tree grows too deep, the score on train data keep increasing, but the score on validation data keep decreasing. For Logistic Regression, the score on train dataset and test dataset always increase or decrease simultaneously. For SVC, we also detect overfitting in the case similar to decision tree.

In Milestone 3, the classifiers we choose (Random Forest, GBDT, XGB) are all ensemble classifiers based on Decision Tree. Thus, the overfitting problem comes as well.
To address the overfitting problem:

- For Decision Tree, we tune the parameters "max_depth", "min_impurity_decrease", "max_leaf_nodes", "min_sample_leaf" to prevent the decision tree from going too deep.
- For Logistic Regression, we set the parameter 'penalty=l2' to apply a regularization, and tuned the parameter 'C' to adjust the regularization strength.
- For SVC, we set the hyperparameter 'C' to a classifier with the parameters 'kernel' and 'gamma' fixed.
- For Random Forest, as we can see in the figure b), as the tree grows deeper, it does not necessarily result in overfitting. RF addresses the problem of overfitting that occurs in its base estimators (Decision Tree).
- For GBDT, we apply the method of tuning the boosting parameters first ('n_estimators' and 'learning_rate'). And then tuning the tree-specific parameters ('max_depth', 'min_samples_split', etc.) to address the overfitting problem of base estimators.
- For XGB, we tune the parameters 'min_child_weight' and 'max_depth' to limit the minimum weighted sum of the children and the maximum depth of the trees, and also tuned the parameter 'gamma' and 'alpha' to adjust the strength of regularization.

5. Recommendations for rental property owners

5.1 Properties to maximize the interest of potential renters when planning a new rental

In the last section, we find that the most relevant features are *price_per_bdr, building_id_num, bedrooms, price, price_per_btr, photo_num, bathrooms,* as well as *longitude, latitude* and some text features. Most are relevant with **the number of bedrooms/ bathrooms**, **price** and **location**. So, we suggest a new rental to:

- **Have a proper number of bedrooms and bathrooms.**
  The number of bedrooms and bathrooms are important features, but it depends on the budget of owners and the actual need of customers and may be hard for us to predict an optimal value. But it is important to design the house with appropriate number of bedrooms and bedrooms that ensures that your house won't be look strange (with too few or too many bedrooms/bathrooms).
- **Have a proper price per bedroom.**
  The average *price_per_bdr* of high, medium, low interest level is 1187, 1357 and 1694, and the *price_per_bdr* of most rentals in high interest level are between 500 and 2000 (in *Figure 11*). So, a lower price may lead to high interest.

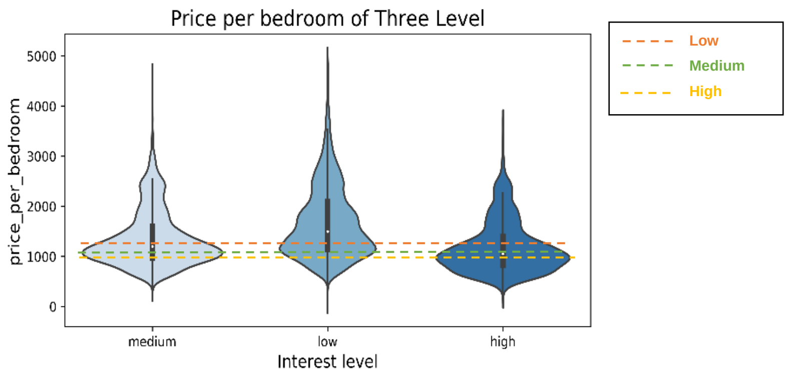


Figure 11 Price per bedroom of three interest levels

  When building a new house/apartment as a rental, it requires the owner to decide the budgets for every bedroom and bathroom as well as focus on the budget for the whole house/apartment.

- **Have a good location which provides a safe, convenient and comfortable living environment.**
  We use DBSCAN algorithm to find clusters of geographical locations of listings with "high" interest level, the result is presented in *Figure 12*. From this figure, we find that this clusters are mainly located along some main avenues, such as 1st Avenue, 2nd Avenue, 3rd Avenue. Furthermore, their locations are at the crossing of one of the main avenues and streets. This finding proves the correctness of our assumption that rentals located in places with an easy access to the transportation and public facilities tend to get a "high" interest level.

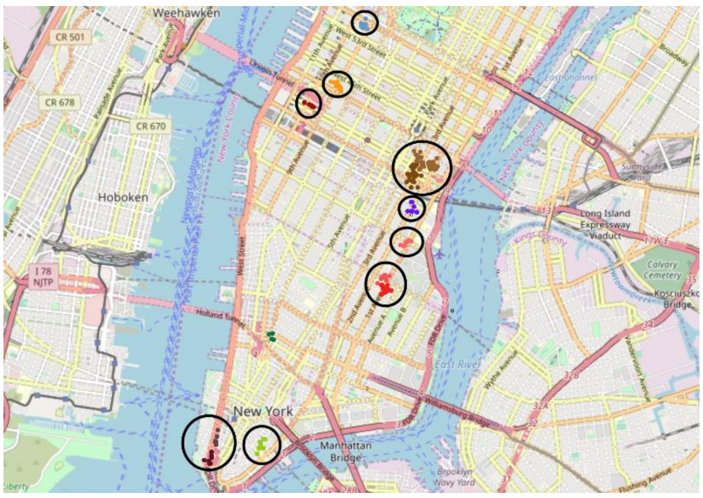


Figure 12 Clusters of geographic locations of "high" interest level listings

Based on these important findings, our suggestion about rental location is as follows:

- **At the crossing of a main avenue and a main street**.
- **Be close to some public facilities**, like subway, airport or supermarket as our additional feature "subway_distance" suggests.

Some text features are also very important, such as *ft_dryerinunit, ft_wheelchairaccess, ft_bikeroom.* So, we suggest a new building as rental to:

- **Have dry in a unit, wheelchair access, bike room, private or public outdoor space, terrace, and doorman.**
  These features are attractions for the customers who have their special needs. However, some common needs like the security should be focused when building a new house.

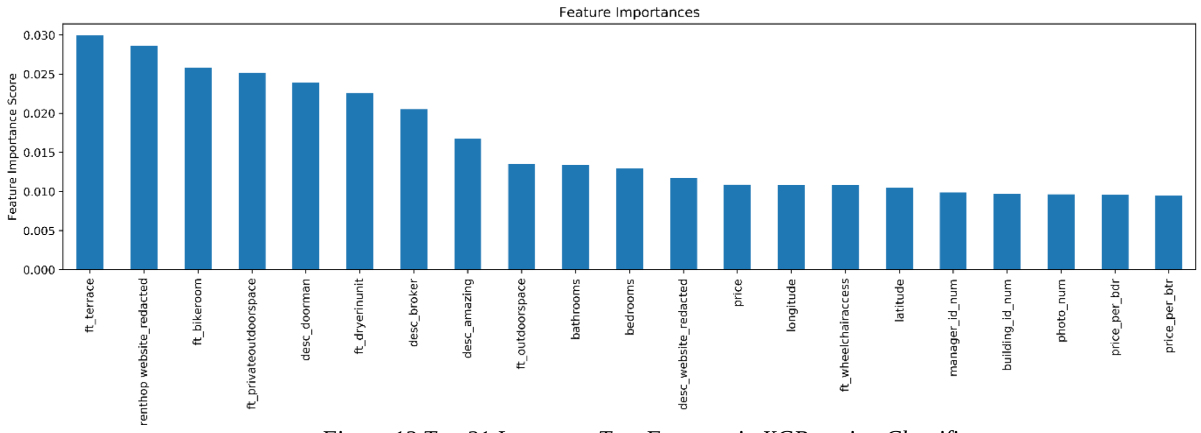


Figure 13 Top 21 Important Text Features in XGBoosting Classifier

5.2 Properties to highlight when promoting an existing rental

- **Use 'amazing' to describe the rental properties.**
- **Find a manager with a good reputation.**
  We find some listings have the common *manager_id* and they all belong to the subset of listings with "high" interest rate. It is reasonable to think that the common manager of these listings enjoys a high reputation, so he/she can be a bonus to these listings when the customers evaluate a listing.
- **Display a proper number (around 5) of photos**
  We evaluate the ratio of the number of listings which have no photo to the total number of listing in each interest level. And we find:
  In the subset of "high" interest level, $\frac{\text{number of listings with no photo}}{\text{total number of listings}} = 1.3\%$
  And in the subset of "low" interest level, $\frac{\text{number of listings with no photo}}{\text{total number of listings}} = 10.0\%$
  Thus, we come to the conclusion that it is important to provide photos.

- **Based on the fact, add as many positive features or descriptions as possible**
  If you have some additional properties that might be an attraction for some customers (like large outer space, additional parking lots), you have to list it out in your "descriptions" or "features" columns.

6. References

[1] Pedregosa *et al.*(2011). Scikit-learn: Machine Learning in Python, JMLR 12, pp. 2825-2830.
[2] XGBoost Documentaion. Retrieved from https://xgboost.readthedocs.io/en/latest/ [Accessed: 2020, April 02]
[3] Aarshay Jain.(2016, February 21) Complete Machine Learning Guide to Parameter Tuning in Gradient Boosting (GBM) in Python [Blog post]. Retrieved from https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/
[4] Adrian Rosebrock.(2014, May 26). OpenCV and Python K-Means Color Clustering [Blog post] Retrieved from https://www.pyimagesearch.com/2014/05/26/opencv-python-k-means-color-clustering/
[5] Aishwarya Singh.(2019, August 29). 3 Beginner-Friendly Techniques to Extract Features from Image Data using Python [Blog post] Retrieved from https://www.analyticsvidhya.com/blog/2019/08/3-techniques-extract-features-from-image-data-machine-learning-python/
[6] Marvin Zhang.(2017, August 24). XGBoosting Parameter Tuning [Blog post] Retrieved from https://zhuanlan.zhihu.com/p/28672955
[7] AARSHAY JAIN.(2016, March 1) Complete Guide to Parameter Tuning in XGBoost with codes in Python [Blog post] Retrieved from https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/
[8] Neil Liberman.(2017, Janurary 26) Decision Trees and Random Forests [Blog post] Retrieved from https://towardsdatascience.com/decision-trees-and-random-forests-df0c3123f991