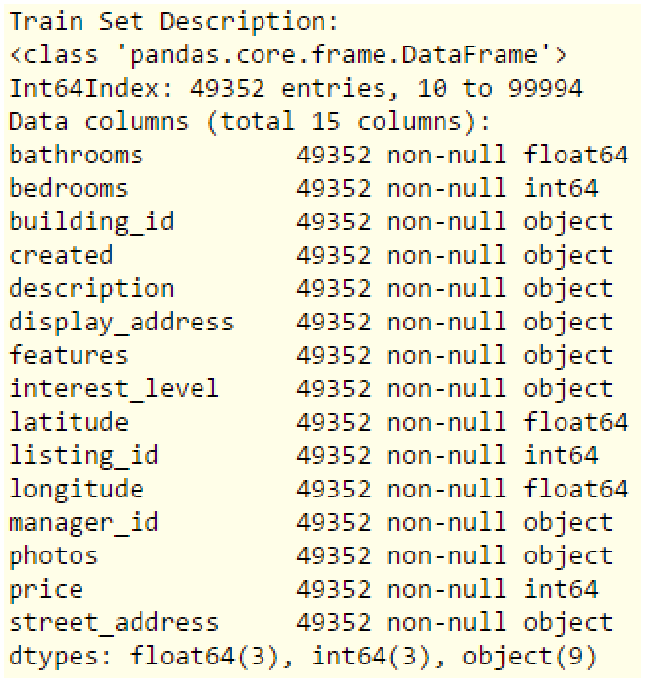
Rental Listing Inquiries Prediction

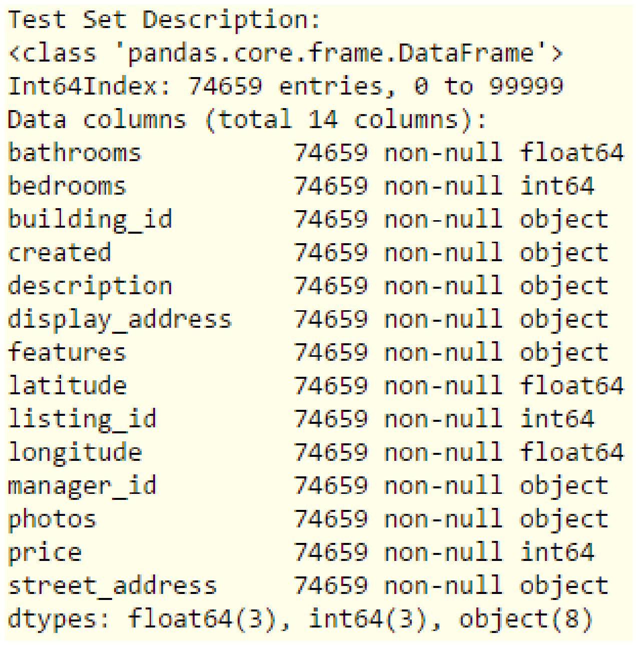
1) General Problem Description:

For this project, I worked on a Kaggle competition for Two Sigma's portfolio company RentHop. Renthop is a web search engine that let users search apartments for rent in major cities. They would like to improve their service by anticipating the number of inquiries a listing may have (inquiries are equivalent to the number of people that solicit more information after reviewing the listing). As a result, they expect to better understand customer preferences and identify listing quality issues.

Data provided is divided in train and test sets, with 49,352 entries and 74,659 entries respectively. There is a total of 14 features (columns) and the target variable: 'high', 'medium', or 'low' inquiries. My goal is to build a model to predict number of inquiries in test set. See below for data description:



```
Train Set Description:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 49352 entries, 10 to 99994
Data columns (total 15 columns):
bathrooms          49352 non-null float64
bedrooms           49352 non-null int64
building_id        49352 non-null object
created            49352 non-null object
description        49352 non-null object
display_address    49352 non-null object
features           49352 non-null object
interest_level     49352 non-null object
latitude           49352 non-null float64
listing_id         49352 non-null int64
longitude          49352 non-null float64
manager_id         49352 non-null object
photos             49352 non-null object
price              49352 non-null int64
street_address     49352 non-null object
dtypes: float64(3), int64(3), object(9)
```



```
Test Set Description:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 74659 entries, 0 to 99999
Data columns (total 14 columns):
bathrooms          74659 non-null float64
bedrooms           74659 non-null int64
building_id        74659 non-null object
created            74659 non-null object
description        74659 non-null object
display_address    74659 non-null object
features           74659 non-null object
latitude           74659 non-null float64
listing_id         74659 non-null int64
longitude          74659 non-null float64
manager_id         74659 non-null object
photos             74659 non-null object
price              74659 non-null int64
street_address     74659 non-null object
dtypes: float64(3), int64(3), object(8)
```

Now, an importance thing to notice is that I am not given the target variable in the test set. This means that the only way to test my model is using my train set through a validation set or some sort of Cross-Validation (CV) technique. Another relevant point is the metric used for evaluation; rather than using the more conventional accuracy (number of true labels predicted) metric, submission will be evaluated using the muti-class logarithmic loss formula:

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log p_{ij}$$

Where N = number of samples; M = number of classes; y = binary digit (1 for correct class and 0 otherwise); $log(p)$ = natural logarithm of the probability assign to correct class

Basically, this metric will greatly penalize you if your prediction for any entry is totally incorrect vs somehow incorrect (in terms of probabilities assigned). That is why, even though you can submit your answers in terms of binary digits, this practice is highly discouraged.

2) Data Pre-Processing:

It is important to note that most machine learning algorithms can only work with numerical data, so all features must be transformed to numbers (integers and/or floats) representing increment/decrement (nominal feature) or levels (ordinal features). Also, ordinal features can't have more than 2 levels: 0 and 1; otherwise, the algorithm will interpret any number higher than one as an increment from 1, and 1 as an increment from 0; in other words, it will treat it as a nominal feature. Since what we want is to indicate a 'yes or no' assignment, we must create new features per every single class we want to indicate. These new features are better known as 'dummy variables' since they are sub-models of the original feature.

These are the steps I took to prepare the data:

- The first thing I did was to encode (assign numbers) the 'building_id' and 'manager_id' features since they contained numbers mixed with letters.

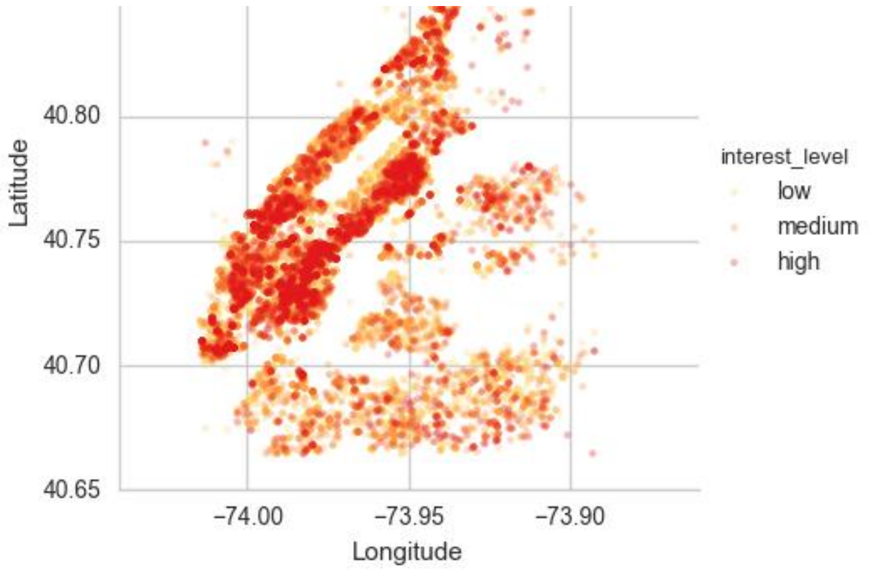
- I parsed the 'created' (for date) feature into month, day, and hour to find out if there is any variance explained by time difference. Now, this dataset contains only entries from April to June 2016. Since there is only 3 months, any cyclical effect for renting apartments can be ruled out.

- I decided to extract the length for text features. For 'description', I created a new feature called 'description_lenght', and for 'features' (things like doorman, elevator, fitness center, etc.) I created 'features_num', both indicating the number of words in the text. It should be obvious that there is a lot more value to be extracted from these features. Sentiment or text analysis must also be applied, but due to time constraint, I decided to stop here and maybe come back later to further squeeze these features.

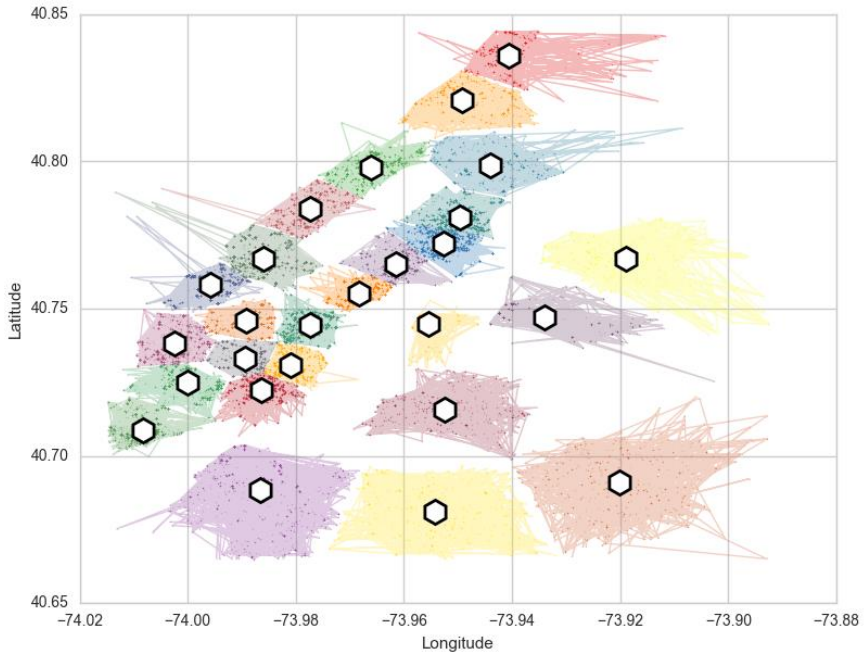
- As for text, I decided to do the same with the number of photos. I created the feature 'photos_num'. There could be some potential for the implementation of neural networks for image analysis and pattern recognition here, but this is beyond my scope for now.

- Next, I had to do something with 'building_id' and 'manager_id' beyond encoding. I ended up with more than 1000 IDs per feature so I had the necessity to compress these categories into a more manageable number (due to the number of dimensions). I used the frequency number of these IDs to compress them into 116 and 150 categories respectively. Basically, IDs with the same frequency were combined into single categories.

- For addresses and location, I decided to work only with 'latitude' and 'longitude'. It is sensible to say that the other features indicating addresses are quite redundant if I already have the location coordinates. My idea was to separate the coordinates into different regions in the map using a clustering algorithm. First, let's look at how the data was distributed before clustering:



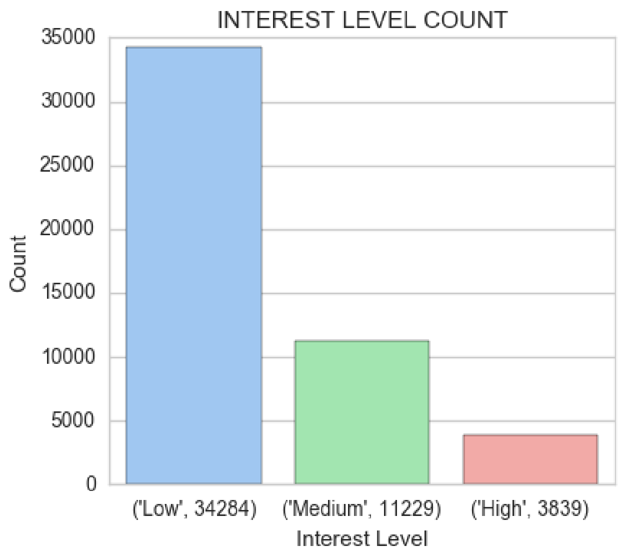
It is hard to notice any patter in the data since all points seem to be equally distributed. It is tempting to say that listings of Manhattan have higher interest, but Manhattan has higher number of listings in total. To compress all these coordinates in a few areas I used the KMeans algorithm. 26 clusters were used and I ended up with the following sections:



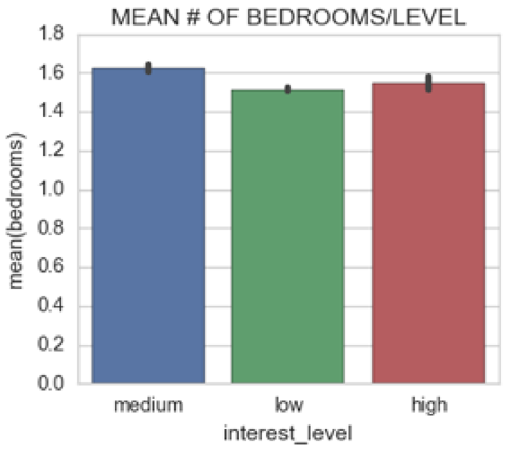
I created a feature with 26 categories using these clusters and dropped all the other location features for location. It should be noted that I am only working with 96% of the data for location. The lower and upper 2% of the data contained outliers that made impossible to plot these maps. Later I put all these outliers into a 1 category.

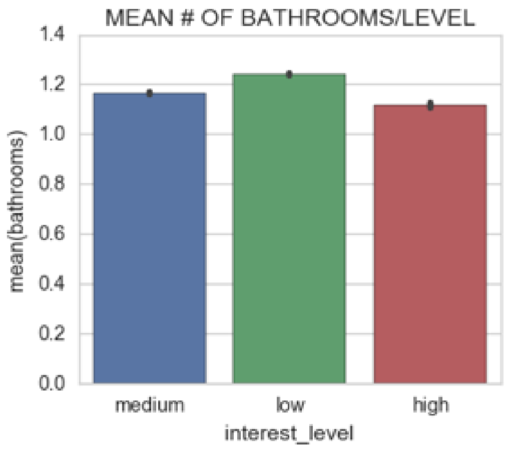
3) Data Visualization:

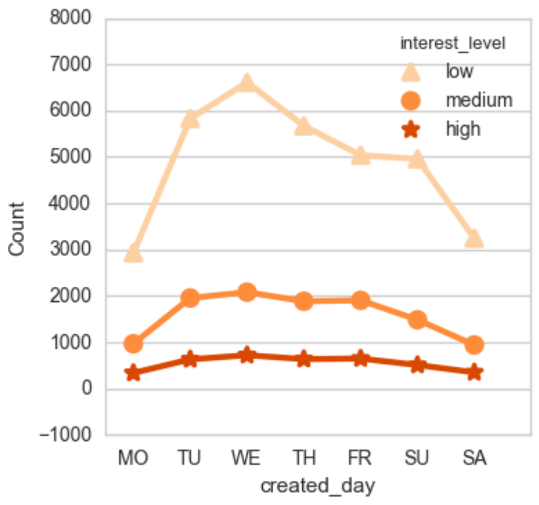
Let's review some plots I made to explore the data. Let's start with the target distribution:

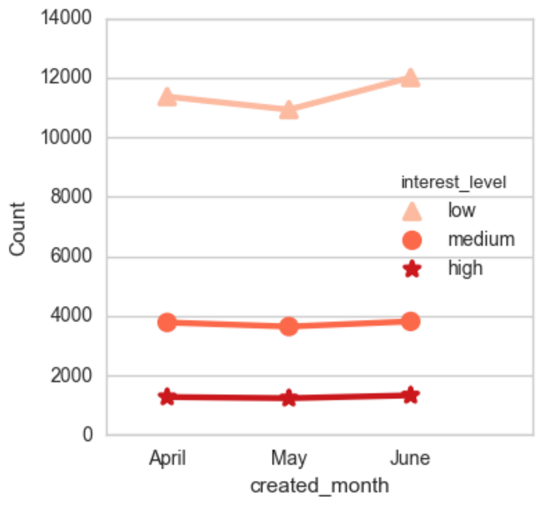


Now, let's see the distribution of the interest levels in the number of bedrooms, bathrooms, days and months:

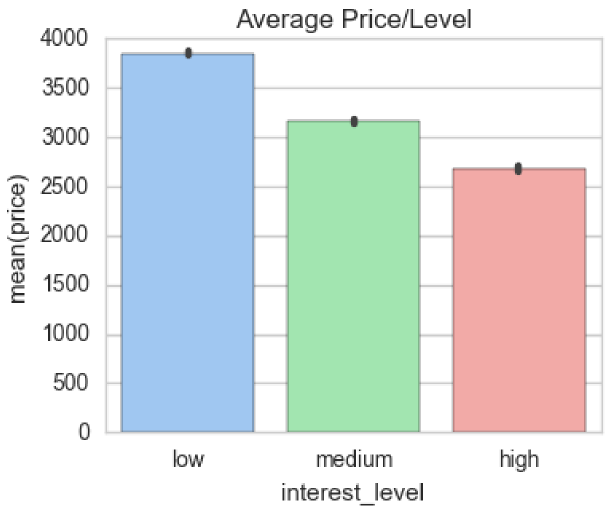


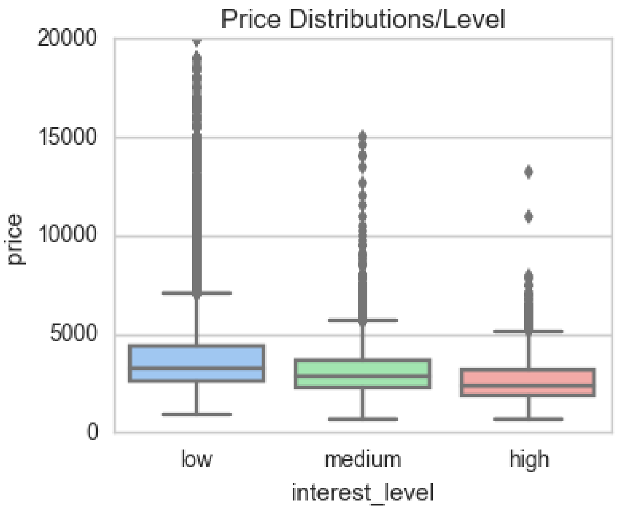






In the first plot, we can see that about 70% of the entries belong to the 'low interest' category. The next 2 plots show the interest level is equally distributed between number of bathrooms (about 1) and bedrooms (about 1.5). The last 2 plots show the same distribution for interest over time; month and day of the week seems to have no influence. The conclusion is that these features have little predictive power; they help little to explain the variance on the target variable. Let's explore price now:



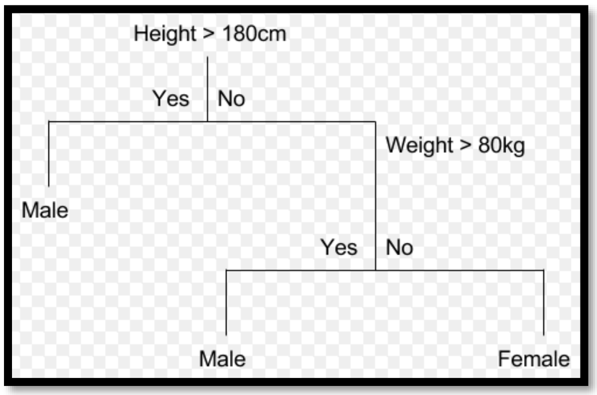


Now we can see a clear trend. The lowest the price (average and median) the higher the interest for the department. Price has a high predicting power. It is something that goes according to our intuition; the first thing that people consider when renting an apartment is probably price.

4) Model Selection & Implementation:

Some final steps I took before model implementation include preparing the test set just as I did with the train set, drop the features that I didn't need anymore, replace outliers in the 'price' feature with the mean of the column, and create dummy variables. The result was a training set with 329 columns.

The first model I attempted to run was a random forest. To understand this model, we must first understand how a decision tree algorithm works. This graph illustrates the logic behind a simple decision tree:



It selects a feature (root node), asks a question, and then creates branches according to the answer. In our example, if the height of a person is greater that 180cm. then we know he is a male, if not, we create another branch to use weight (a child node) as a predictor for Male or Female class.

But, how are the features selected? Behind the scenes there is an optimization algorithm trying to minimize the result of an equation called "Gini Impurity". This function measures how *pure* a node is, meaning: how many samples in the node the classification applies to. Let's look at this equation:

$$Gini = 1 - \sum_{i=1}^{C} (p_i)^2$$

*where **i =** number of classes, **Pi =** probability of class i. (Function runs independently at each node.)*

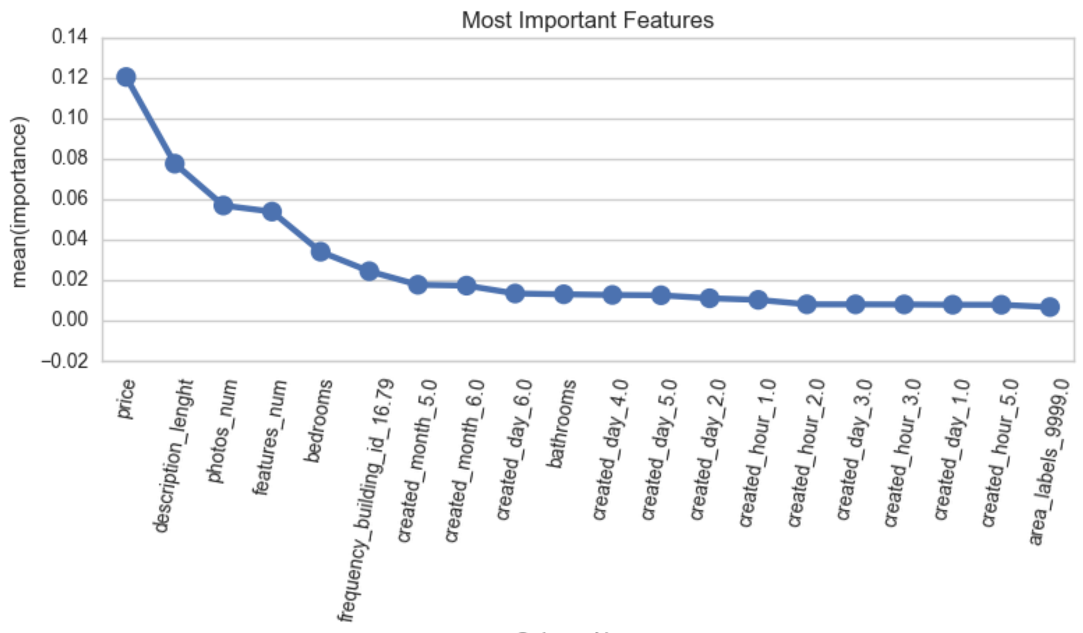
For example, let's say that all entries with more than 180cm are males and there are 90 samples at that node. Then, ***impurity =*** 1- (90/90)^2 = 0. We can say that the node is completely *pure*. Let's say there are other 90 samples, and 45 out of 90 weight more than 80kg. The best classification therefore is Male, but assuming there are 3 females in that node***, impurity =*** 1-(42/45)^2-(3/45)^2 = 0.124.

At the beginning, the algorithm goes over all features, looking for the best threshold to split the data. The best selection, the one that better minimizes our cost function, gets selected as our root node. Then the process repeats for the first level of child nodes, and so on. It is one of the easiest algorithms to understand and the results are easily interpretable (a white box). However, it suffers from a big limitation due to its orthogonal tendency for classification: it is highly sensitive to small modifications in our data; hence, it tends to overfit. Here is where Random Forest comes to the rescue.

A Random Forest is an ensemble method – essentially, a combination of 2 or more predictors. It is a powerful and highly customizable algorithm that has gained popularity in the machine learning community in the last few years. Basically, it produces many (parameter specified by user) trees, from randomly selected subsets of the data with replacement (known as *bootstrapping*), and with randomly selected features; then, it assigns a category or makes a prediction based on majority vote (hard voting classifier). There is even an extreme form called *Extremely Randomized Trees* which leaves behind the notion of *the best threshold* (to reduce impurity) and randomizes this parameter at each trees' nodes, ending up with large numbers of highly *impure* trees.

All this sounds counterintuitive, how is it possible to use less (and highly randomized) data and features and end up with higher predictive accuracy? Thanks to the *law of large numbers*. Let's use a simple example: the probability of getting a 6 after throwing a dice is 1/6; it means that *'on average'* the number 6 will appear every 6 throws. However, this doesn't mean that you can expect to get a 6 after throwing the dice 6 times, it means that only *'on average'* the number 6 will appear 1/6 of the time. Given enough throws, let's say 1 million, the number 6 will most likely *converge* to its true probability of 1/6. Now, notice the assumption of this experiment; every trial is independent; hence, the probability of each outcome doesn't alter the probability of the other, letting 6 converge to its *true probability* (or expected value). The same is true with Random Forest or with any other ensemble method. The more classifiers you have, and the more independent (randomized) they are from each other, the best they will converge towards the true probability of each class.

After running my first Random Forest I came across a big surprise: It was computationally expensive. It took about 10 min. to run a model with 4,000 trees. So, I decided to reduce my number of columns. Thankfully, Random Forest offers a handy way to understand feature importance. If we look at a tree, it is fair to say that the most relevant features are going to be at higher nodes, while less important features at the bottom. Higher nodes positions imply higher purity, or better ability to split the data with better predictive accuracy. After fitting thousands of trees (4,000 in my case), we can have a good estimate of what features are at higher and lower nodes. The algorithm stores this information automatically. The following plot was created using these percentages:



Price becomes the most relevant feature, then the length of description, then the number of pictures, and so on (distribution of the features agrees with intuition).

I decided to get the first 10 features (which explain about half of the data) and compress the other ones in 5 columns using Principal Component Analysis or PCA (a feature extraction technic to reduce the number of dimensions selecting the *planes* -from n-dimensional space- that retain the most variance and projecting all data point onto them). I ended up with 15 columns

Then I run the Random Forest again using cross-validation with 10 folds and 1,000 trees (Generally, the less columns you have the less trees you need). Speed was greatly increased but not as expected; It took about 5 min. to fit my data with Random Forest (understandable speed due to cross-validation). I obtained a 71.31% accuracy and a log loss of 0.6923.

I also run a simple Logistic Regression model to compare my previous scores. I obtained 69.67% accuracy and a log loss of 0.6914. At this point, I decided to halt the project due to time constraints. Algorithms were taking too long to run; I really wanted to try many other ensembles, parameters, and work more on my features. But I realized it was going to take too many additional days at the speed the algorithms were running with cross-validation (no wonder people team up to work on these competitions).

5) Conclusion & Recommendations:

There are many things that I could have done better, especially If I had some more time. In general, these competitions are a matter of trying multiple things with cross-validation and sticking with your best result (chosen metric score); possibilities are virtually endless. Options include different features (this is huge!), different algorithms, different parameters (this is computational expensive), and different ensembles. Steps I would take If I try something else include:

- First, text analysis is a must. There should be a lot of value inside ‘description’ and ‘features’ that I am totally missing to capture. It is reasonable to say that certain descriptions are better than others at capturing the interest of people. Also, people should prefer certain features more than others. Creating a *bag of words* should be necessary to improve my score.

- I would like to try some other options to compress ‘manager_id’ and ‘building_id’. Compressing by frequencies was a good option since I captured a lot of information, but maybe trying some other techniques would yield better results.

- Lastly for features, there could be some potential on avenue and/or street name.

- I should have spent more time on data exploration trying to find outliers.

- I could have done a better job at extracting and compressing features. I was afraid I was not going to be able to try all the models I wanted (due to speed) so I rushed to select only 10 features (with Random Forest) and compress everything else. I would have done a better job just using PCA to compress categories within themselves, not mixing.

- Due to the scoring metric (log loss), it could have been beneficial to apply some smoothing technique like *Laplace smoothing* to the final class probabilities.

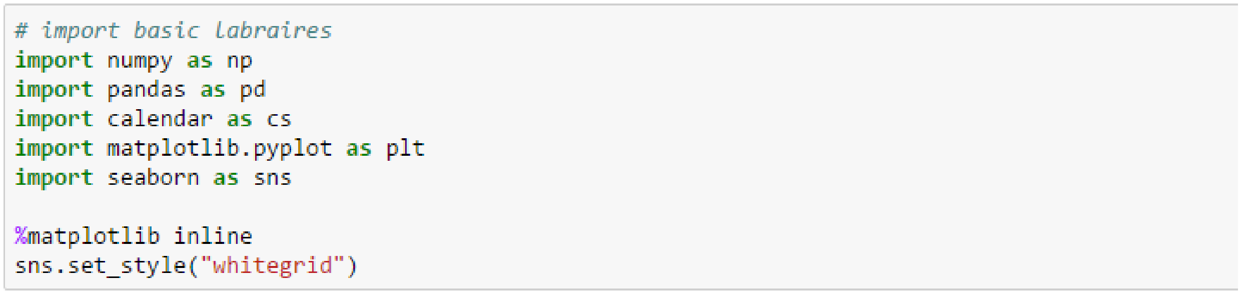
- Something I really wanted to try is the XGBoost algorithm. It works like Random Forest, but instead of creating a random tree at each iteration, it tries to correct each tree by improving the error rate of the other. This algorithm is rapidly becoming the standard on these competitions since it has shown to outperform even the powerful Random Forest. It has its complications and disadvantages though. For example, it is very sensitive (tends to overfit) to hyperparameter tuning, an area where Random Forest is very resilient.

Overall, it has been an extremely rewarding experience to spend time on this machine learning project. I am satisfied because I know exactly what steps to take If I wanted to improve my score. Also, having an accuracy score of 71% (not being so impressive) indicates that I haven’t made a gross mistake in the process. With a few more days, I surely would have been able to greatly improve my score.

6) Appendix:

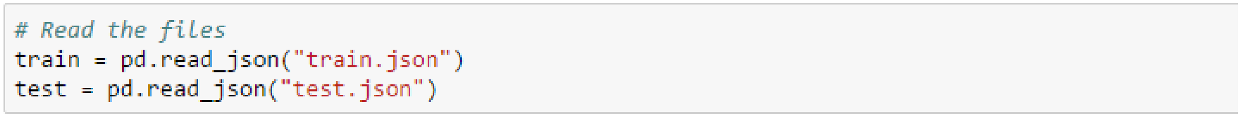
In this section I will append all the code I used to work on this project. I worked everything with Python using standard libraries for data manipulation, visualization and machine learning.

~Exploratory Data Analysis & Pre-processing~

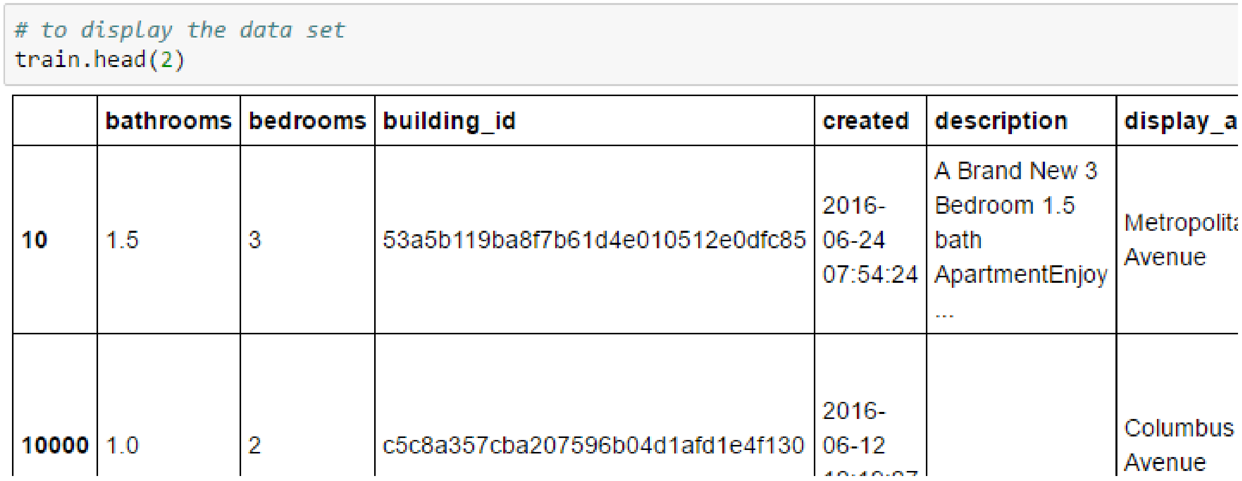


```
# import basic labraires
import numpy as np
import pandas as pd
import calendar as cs
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
sns.set_style("whitegrid")
```

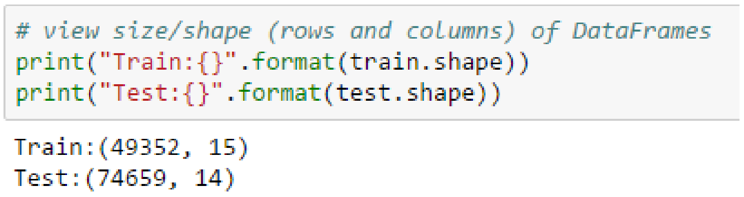


```
# Read the files
train = pd.read_json("train.json")
test = pd.read_json("test.json")
```



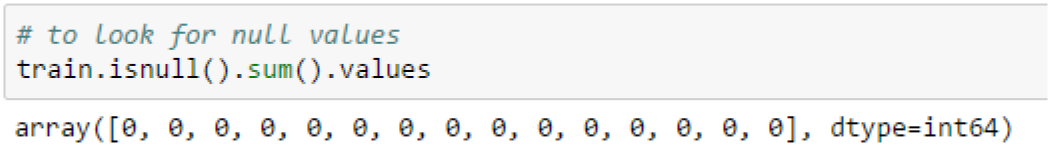
```
# to display the data set
train.head(2)
```

	bathrooms	bedrooms	building_id	created	description	display_a
10	1.5	3	53a5b119ba8f7b61d4e010512e0dfc85	2016-06-24 07:54:24	A Brand New 3 Bedroom 1.5 bath ApartmentEnjoy ...	Metropolit Avenue
10000	1.0	2	c5c8a357cba207596b04d1afd1e4f130	2016-06-12		Columbus Avenue



```
# view size/shape (rows and columns) of DataFrames
print("Train:{}".format(train.shape))
print("Test:{}".format(test.shape))
```

```
Train:(49352, 15)
Test:(74659, 14)
```



```
# to look for null values
train.isnull().sum().values
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int64)
```

Feature engineering

I will produce some basic features to easily work with the data at this stage.
I might explore creating more complex features after running my first models.

```
# to combine both datasets for pre-processing
train_test = train.append(test)
```

```
# to look for unique values
print(train_test['building_id'].nunique())
print(train_test['manager_id'].nunique())
print(train_test['interest_level'][~train_test['interest_level'].isnull()].count())
print(sum(train_test['interest_level'].isnull()))
```

```
11635
4399
49352
74659
```

-- Encoding Ordinal Features

```
# to replace 'id's (manager and building) & 'interest_level' with single numbers. Important to run
# some algorithms that can only work with numerical data (and to clean up some space in our DataFrame)
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()

train_test['building_id'] = encoder.fit_transform(train_test['building_id'])
train_test['manager_id'] = encoder.fit_transform(train_test['manager_id'])

# The same for 'interest_level" at train set. LabelEncoder() can't encode
# in 'just one section' of a dataset so I will separe
# train and test again

enc = LabelEncoder()
tr = train_test['interest_level'][~train_test['interest_level'].isnull()].copy()
trt = train_test['interest_level'][train_test['interest_level'].isnull()].copy()
tr = enc.fit_transform(tr)
train_test['interest_level'] = np.concatenate((tr,trt)).copy()

print(train_test['building_id'].nunique())
print(train_test['manager_id'].nunique())
print(train_test['interest_level'][~train_test['interest_level'].isnull()].count())
print(sum(train_test['interest_level'].isnull()))
# CHECK: the number of unique values should be the same after encoding
```

```
11635
4399
49352
74659
```

-- Date/time Features

```
# to easily parse date/time objects
train_test['created'] = pd.to_datetime(train_test['created'])

# to create 4 new variables: month, day, and hour
# 'Year' is not needed since all entries are from 2016
train_test['created_month'] = train_test['created'].dt.month
train_test['created_day'] = train_test['created'].dt.weekday_name
train_test['created_hour'] = train_test['created'].dt.hour
```

-- Description lenght & # of features and photos Feature

```
# to create a variable for lenght of 'description', then drop the column.
train_test['description_lenght'] = train_test['description'].apply(lambda string: len(string.split()))
train_test.drop('description',axis=1,inplace=True)
```

```
# to create a variable for lenght of 'features', then drop the column.
train_test['features_num'] = train_test['features'].apply(lambda ls: len(ls))
train_test.drop('features',axis=1,inplace=True)
```

Note: there could be a lot of potential inside these 2 features. More text analysis could yield better results

```
#to create a variable for number of 'photos', then drop the column.
train_test['photos_num']=train_test['photos'].apply(lambda ls: len(ls))
train_test.drop('photos',axis=1,inplace=True)
```

-- 'building_id' & 'manager_id' Features

There are 2 ordinal features with multiple levels that have the potential to greatly increase the number of dimensions after creating dummy variables: 'building_id' and 'manager_id'. I will deal with them now by reducing/combining the number of levels. There are multiple approches that could be used. I will compress categories by frequencies.

```
# to separate train and test again. Since Frequency will be used, the total must be only taken from train set
# REMEBER: train set must never 'learn' from test set
f_train = train_test[train_test['interest_level'].isnull()==False].copy()
f_test = train_test[train_test['interest_level'].isnull()].copy()
```

```
# to create a dataframe with frequencies
pan = pd.DataFrame(f_train['building_id'].value_counts())
pan.reset_index(inplace=True)
pan.columns = ('building_id','count')
```

```
pan['frequency'] = round((pan['count']/(sum(pan['count'])))*100,3)
```

```
pan.head(4)
# We can see that 16% of listing are for apt.id 0, 5.5% for apt.ID 6806, and so on
```

	building_id	count	frequency
0	0	8286	16.790
1	6806	275	0.557
2	810	215	0.436
3	5859	213	0.432

```
# CHECK: verify that there is a unique frequency per count
print(pan['count'].nunique(),pan['frequency'].nunique())
```

```
116 116
```

```
# to create a dictionary object to map these fequencies values into my main trainnig set
# to convert values to strings. REMEMBER: distionaries take only strings types as 'keys'
pan['building_id'] = pan['building_id'].apply(lambda x: str(x))
keys = list(pan['building_id'].values)
values = list(pan['frequency'].values)
```

```
DICT1=dict(zip(keys,values))
```

```
# to check for errors. Frequencies should match
pan['fre_check']=pan['building_id'].map(DICT1)
```

```
pan.head(4)
```

	building_id	count	frequency	fre_check
0	0	8286	16.790	16.790
1	6806	275	0.557	0.557
2	810	215	0.436	0.436
3	5859	213	0.432	0.432

```
# to convert f_train.building_id to str. Necessary to map DICT1
f_train['building_id'] = f_train['building_id'].apply(lambda x: str(x))
```

```
# to map DICT1 and create the new categorical column
f_train['frequency_building_id']=f_train['building_id'].map(DICT1)
f_train['frequency_building_id'].nunique()
```

```
116
```

```
f_train['building_id'].nunique()
```

```
7585
```

I have transformed an ordinal feature with 7585 values to only 116. Some information might be lost but computational speed will greatly increase. I will to the same procedure with 'manager id'.

```
# to create a dataframe with frequencies
pan2=pd.DataFrame(f_train['manager_id'].value_counts())
pan2.reset_index(inplace=True)
pan2.columns = ('manager_id','count')
pan2['frequency'] = round((pan2['count']/(sum(pan2['count'])))*100,5)
```

```
# to verify that there is a unique frequency per count
print(pan2['count'].nunique(),pan2['frequency'].nunique())
```

```
150 150
```

```
# to create a dictionary to map these fequencies values into my main trainnig set
pan2['manager_id'] = pan2['manager_id'].apply(lambda x: str(x))
keys = list(pan2['manager_id'].values)
values = list(pan2['frequency'].values)
DICT2=dict(zip(keys,values))
```

```
# to convert f_train.manager_id to str. Necessary to map DICT2
f_train['manager_id'] = f_train['manager_id'].apply(lambda x: str(x))
```

```
# to map DICT2 and create the new categorical column
f_train['frequency_manager_id']=f_train['manager_id'].map(DICT2)
f_train['frequency_manager_id'].nunique()
```

```
150
```

```
# to drop 'building_id' & 'manager_id'. Don't nee them anymore
f_train.drop(['building_id','manager_id'],axis=1,inplace=True)
```

-- Location Feature

For location feature I have 4 variables: "display_addres", "street addres", and "longitude"/"latitude".
I will delete the first 2 and work only with 'latitude' and 'longitude' since having all together is kind of redundant.
I will use clustering (KMeans) to divide the coordinates by regions/segments and create a column to convert them into categories

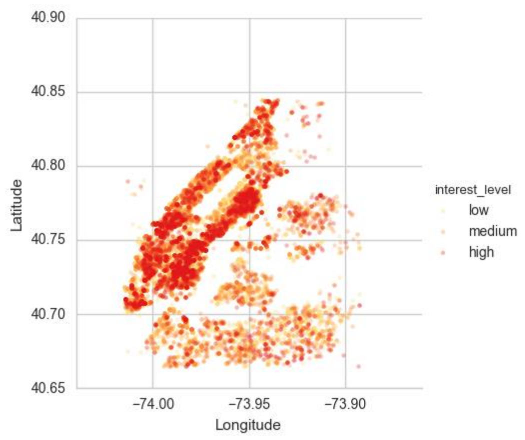
```
# to reverse encoding on labels for visualization purposes
f_train['interest_level'] = enc.inverse_transform(tr)
```

```
# to create a set with about 96% of points since most of the listings should be from Manhattan, Brooklyn, Queens
# I will treat the remaining points as 'outliers' and will assign them to a single category later
ny_map = f_train[(f_train['longitude']>f_train['longitude'].quantile(0.02))
                            &(f_train['longitude']<f_train['longitude'].quantile(0.98))
                            &(f_train['latitude']>f_train['latitude'].quantile(0.02))
                            &(f_train['latitude']<f_train['latitude'].quantile(0.98))].copy()
```

```
# to visualize listings in the city
sns.lmplot(x="longitude", y="latitude", fit_reg=False, palette="YlOrRd", hue='interest_level',
           hue_order=['low', 'medium', 'high'], size=4.5, scatter_kws={'alpha':0.3,'s':10},
           data=ny_map)

plt.xlabel('Longitude')
plt.ylabel('Latitude')
```

```
<matplotlib.text.Text at 0x2d97c93b390>
```



```
other_points_index=[]
for i in f_train.index:
    if i not in ny_map.index:
        other_points_index.append(i)
print('Number of lisings not in the city:',len(other_points_index))
print('Percentage of listings not in city:{:.2%}'.format(len(other_points_index)/len(f_train.index)))
```

```
Number of lisings not in the city: 3761
Percentage of listings not in city:7.62%
```

Now, I will use Kmeans to find a fair number of areas

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters=26, init='random', n_init=12,
            max_iter=300, random_state=0)
y_km = km.fit_predict(ny_map[['latitude','longitude']])
```

```
# clusters assign
ny_map['area_labels']=y_km
np.unique(y_km)
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25])
```

```
# centers of clusters
km.cluster_centers_[:,0]
```

```
array([ 40.83604361,  40.72239797,  40.71565307,  40.74724124,
        40.75812641,  40.77204182,  40.79858794,  40.78115403,
        40.74445273,  40.72488668,  40.79787123,  40.70858062,
        40.76707428,  40.73304799,  40.76527601,  40.68850881,
        40.73809486,  40.7838935 ,  40.69090386,  40.74582066,
        40.7552913 ,  40.82088464,  40.73075945,  40.74482011,
        40.68092138,  40.76705937])
```

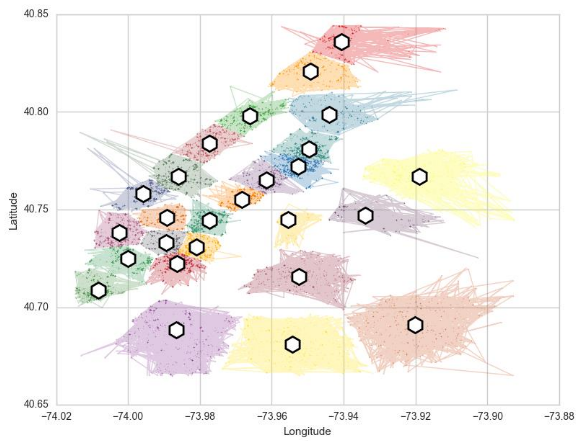
```
# Let's visualize the clusters
from matplotlib import cm

plt.figure(figsize=(9,7))

for i in np.unique(y_km):
    plt.plot(ny_map[ny_map['area_labels']==i]['longitude'],
             ny_map[ny_map['area_labels']==i]['latitude'],
            color=cm.Set1(X=i/40), alpha=.3, lw=1, marker='o', ms=1 )

plt.plot(km.cluster_centers_[:,1],km.cluster_centers_[:,0], lw=0, marker="h", ms=16, mfc='white',
        markeredgewidth=2, markeredgecolor="black")
plt.xlabel('Longitude')
plt.ylabel('Latitude')
```

```
<matplotlib.text.Text at 0x2d97d021908>
```



Now, Let's assign the area labels to our train set f_train

```
# to select rows where coordinates don't belong to our city map and create a new set
t_other = f_train.loc[other_points_index,:].copy()
# to create a new label for these other points
t_other['area_labels'] = 9999
# to concatente 'ny_map' and 't_other'
fin_train = ny_map.append(t_other)
```

```
# Now we can drop the location features that we won't use anymore
fin_train.drop(['latitude','longitude','street_address','display_address'],axis=1,inplace=True)
```

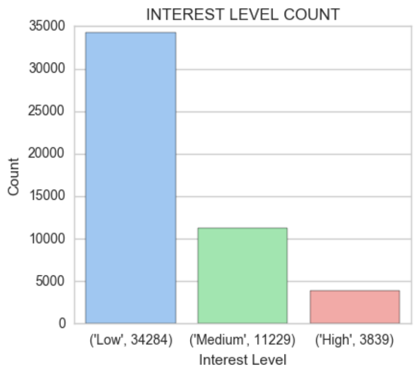
```
# CHECK: number of rows should be the same as the beginning: 49352
print("Train:{}".format(fin_train.shape))
```

```
Train:(49352, 15)
```

Data Visualization

- Target Distribution

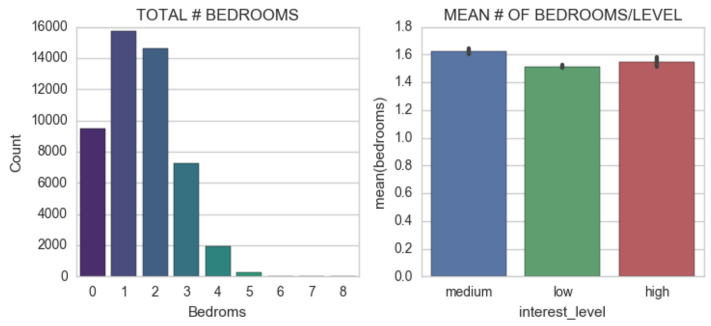
```
plt.figure(figsize=(4.5,4))
sns.countplot(fin_train['interest_level'], order=['low','medium','high'], palette="pastel")
plt.xlabel('Interest Level')
plt.ylabel('Count')
plt.title('INTEREST LEVEL COUNT')
plt.xticks(range(3),[('Low',train.interest_level.value_counts()[0]),
                     ('Medium',train.interest_level.value_counts()[1]),
                     ('High',train.interest_level.value_counts()[2])])
plt.tight_layout()
```



- Bathrooms & Bedrooms

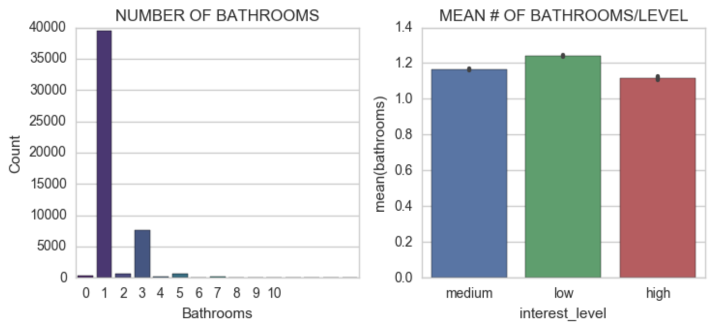
```
plt.subplots(figsize=(7.5,3.5))
plt.subplot(121)
sns.countplot(fin_train['bedrooms'], palette ='viridis')
plt.xlabel('Bedroms')
plt.ylabel('Count')
plt.title('TOTAL # BEDROOMS')

plt.subplot(122)
sns.barplot(y=fin_train['bedrooms'],x=fin_train['interest_level'])
plt.title('MEAN # OF BEDROOMS/LEVEL')
plt.tight_layout()
```



```
plt.subplots(figsize=(7.5,3.5))
plt.subplot(121)
sns.countplot(fin_train['bathrooms'], palette ='viridis',)
plt.xlabel('Bathrooms')
plt.ylabel('Count')
plt.xticks(range(0,11),range(0,11))
plt.title('NUMBER OF BATHROOMS')

plt.subplot(122)
sns.barplot(y=fin_train['bathrooms'],x=fin_train['interest_level'])
plt.title('MEAN # OF BATHROOMS/LEVEL')
plt.tight_layout()
```

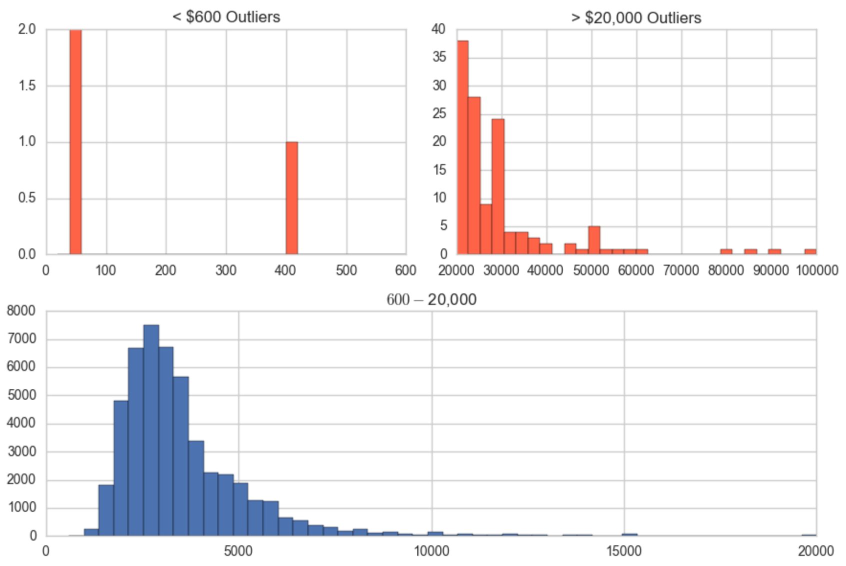


-- Price Outliers

```
plt.subplots(figsize=(9,6))
plt.subplot(221)
plt.hist(fin_train['price'],bins=30,range=(0,600),color='tomato')
plt.title('< $600 Outliers')

plt.subplot(222)
plt.hist(fin_train['price'],bins=30,range=(20000,100000),color='tomato')
plt.title('> $20,000 Outliers')

plt.subplot(212)
plt.hist(fin_train['price'],bins=50,range=(600,20000),)
plt.title('$600 - $20,000')
plt.tight_layout()
```



```
# to create a mean without counting outliers
mean = fin_train[(fin_train['price']>600)&(fin_train['price']<20000)]['price'].mean()
print(mean)
```

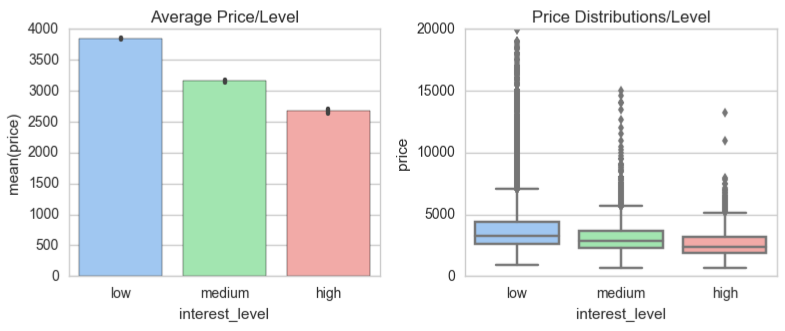
```
3599.682007518033
```

```
# to replace outliers with the mean
fin_train.loc[fin_train[fin_train['price']<=600]['price'].index,'price'] = mean
fin_train.loc[fin_train[fin_train['price']>=20000]['price'].index,'price'] = mean
```

- Average & Median Price

```
plt.subplots(figsize=(8,6))
plt.subplot(221)
sns.barplot(x= fin_train['interest_level'], y= fin_train['price'],
            palette ='pastel', order=['low','medium','high'])
plt.title('Average Price/Level')

plt.subplot(222)
sns.boxplot(x= fin_train['interest_level'], y= fin_train['price'],
            palette ='pastel', order=['low','medium','high'])
plt.title('Price Distributions/Level')
plt.tight_layout()
```



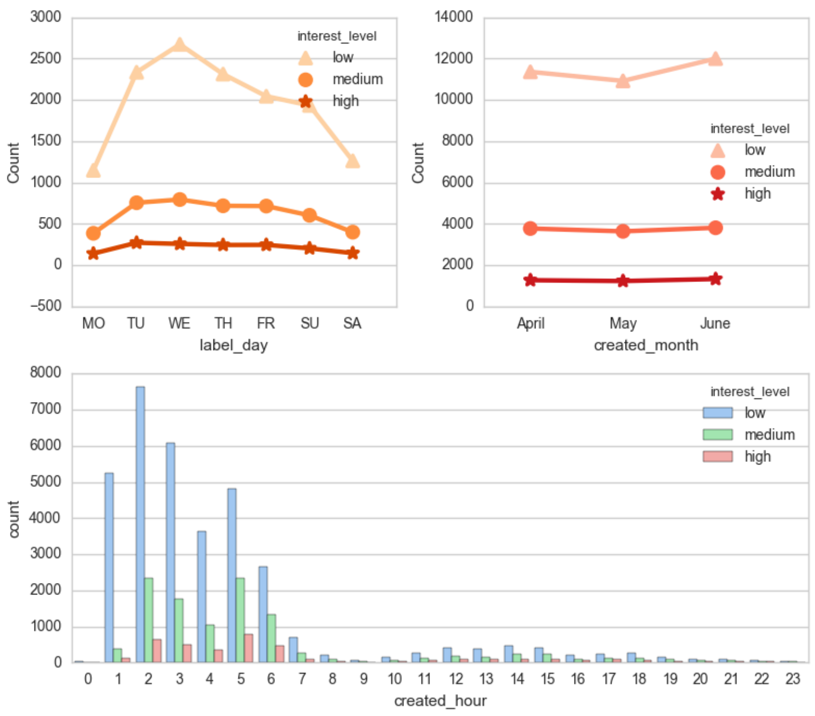
- Level Distributions per Month, Day, and Hour

```
# to encode days to plot them by distribution
e_day = LabelEncoder()
tp = pd.DataFrame()
tp['label_day'] = e.fit_transform(fin_train['created_day'])
new_tp = fin_train.copy()
new_tp['label_day'] = tp['label_day'].copy()
```

```
plt.subplots(figsize=(8,7))
sns.pointplot(x="label_day",y="label_day", hue="interest_level", order =[1,5,6,4,0,2,3],
              hue_order=['low', 'medium', 'high'], ax=plt.subplot(221),
              data=new_tp, palette='Oranges', estimator=lambda x: len(x), markers=["^", "o","*"])
plt.xticks(range(0,8),'MO,TU,WE,TH,FR,SU,SA'.split(','))
plt.ylabel('Count')
sns.pointplot(x="created_month", y="created_month", hue="interest_level",
              hue_order=['low', 'medium', 'high'], ax=plt.subplot(222), estimator=lambda x: len(x),
              data=fin_train, palette='Reds',markers=["^", "o","*"])
plt.xticks(range(0,4),'April,May,June'.split(','))
plt.ylabel('Count')

sns.countplot(x="created_hour", hue="interest_level",
              hue_order=['low', 'medium', 'high'], ax=plt.subplot(212),
              data=fin_train, palette='pastel')

plt.tight_layout()
```



Data Preparation

At this step I will prepare both datasets fin_train and f_test for model implementation

```
fin_train.head(1)
```

	bathrooms	bedrooms	interest_level	listing_id	price	created_month	created_day	created_hour	de
10	1.5	3	2	7211212	3000.0	6	Friday	7	93

```
# I don't need 'created' anymore since I have extracted new features from it
fin_train.drop('created',axis=1,inplace=True)
```

```
# Let's create a function to transform my f_test set exactly as fin_train set
def transform_test(mtest,mtrain):

    # for 'manager_id' and 'building_id' frequencies.
    mtest['building_id'] = mtest['building_id'].apply(lambda x: str(x))
    mtest['manager_id'] = mtest['manager_id'].apply(lambda x: str(x))
    mtest['frequency_building_id']=mtest['building_id'].map(DICT1)
    mtest['frequency_manager_id']=mtest['manager_id'].map(DICT2)

    # for latitude and longitude
    ny_map = mtest[(mtest['longitude']>mtest['longitude'].quantile(0.02))
                    &(mtest['longitude']<mtest['longitude'].quantile(0.98))
                    &(mtest['latitude']>mtest['latitude'].quantile(0.02))
                    &(mtest['latitude']<mtest['latitude'].quantile(0.98))].copy()

    km = KMeans(n_clusters=26, init='random', n_init=12,
           max_iter=300, random_state=0)
    y_km = km.fit_predict(ny_map[['latitude','longitude']])
    ny_map['area_labels']=y_km

    other_points_index=[]
    for i in mtest.index:
        if i not in ny_map.index:
            other_points_index.append(i)

    oth = mtest.loc[other_points_index,:].copy()
    oth['area_labels'] = 9999
    fin_y = ny_map.append(oth)

    # to drop all columns not in train
    fin_y.drop(['building_id','manager_id','latitude',
              'longitude','street_address','display_address','created'],axis=1,inplace=True)

    # to encode 'interest_level' in mtrain, save the index for train and the target
    enc = LabelEncoder()
    mtrain['interest_level'] = enc.fit_transform(mtrain['interest_level'])
    train_index = list(mtrain.index)
    target = mtrain['interest_level']

    # to concatenate train and test
    conc = mtrain.append(fin_y)

    # to encode 'created_day'
    enc2 = LabelEncoder()
    conc['created_day'] = enc2.fit_transform(conc['created_day'])
    # to drop columns that won't be part of our model, fill null values, and convert values in integers
    conc = conc.drop(['interest_level','listing_id'],axis=1)
    conc = conc.fillna('9999999999')
    conc = conc.applymap(lambda x: float(x))

    return conc, train_index, target
```

```
# Let's transform our test set!
conc, train_index, target = transform_test(f_test,fin_train)
```

```
# to know the percentage of null values, or the levels/categories in test set not found in train set
print('{:.2%}'.format((conc[conc['frequency_building_id']==9999999999].count()['price'])/len(test.index)))
print('{:.2%}'.format((conc[conc['frequency_manager_id']==9999999999].count()['price'])/len(test.index)))
print('{:.2%}'.format((conc[conc['area_labels']==9999999999].count()['price'])/len(test.index)))
```

```
7.52%
1.97%
0.00%
```

```
# to convert values of categorical features into strings. Necessary to call the get_dummies function
my_list = ['frequency_building_id','frequency_manager_id','area_labels',
           'created_day','created_hour','created_month']
for i in my_list:
    conc[i] = conc[i].apply(lambda x: str(x))

#to create the dummy variables
t_dummies = pd.get_dummies(conc, columns= my_list, drop_first=True)
```

```
t_dummies.shape
```

```
(124011, 329)
```

```
# to extract x_train, y_train, and x_test
X = t_dummies.loc[list(target.index),:].copy()
y = list(target.values).copy()
X_test = t_dummies.loc[list(f_test.index),:].copy()
```

~ MODEL Selection & Implementation ~

I will start using RandomForest since it is one of the most powerful and easy to implement algorithms by default

```
from sklearn.ensemble import RandomForestClassifier
labels = X.columns
my_forest = RandomForestClassifier(n_estimators=4000, random_state=0, n_jobs=-1)
my_forest.fit(X,y)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=4000, n_jobs=-1, oob_score=False, random_state=0,
            verbose=0, warm_start=False)
```

It took about 10 minutes to fit my data. If I want to try many models I definitely need to remove irrelevant features. Thankfully Random Forest has it's own method to detect irrelevant features. And after 4,000 trees in the forest, estimation should be really good. There are many other techniques to assess feature importance and dimension reduction. But, due to simplicity and time constraint, I will only use the Random Forest estimator.

- Feature Importance

```python
# Assign the importance percentage of each feature
importance = my_forest.feature_importances_
```

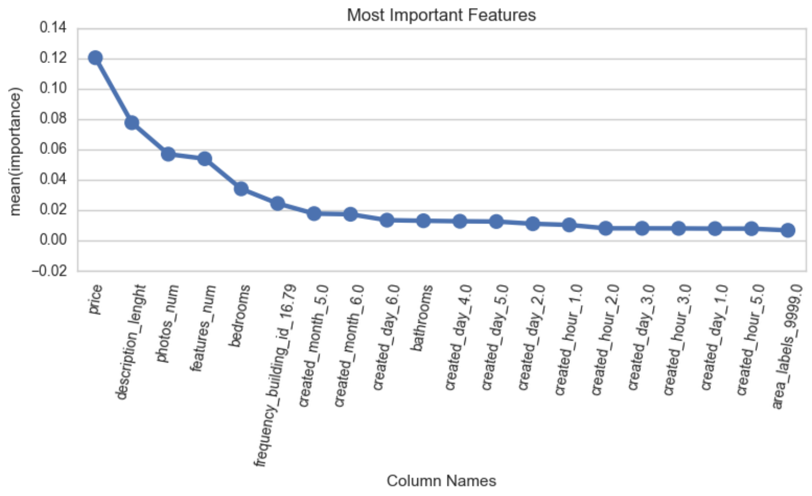
```python
# CHECK: number of assignments
print('Number of columns: {}'.format(X.shape[1]))
print('Importance %s assigned: {}'.format(len(importance)))
```

```
Number of columns: 329
Importance %s assigned: 329
```

```python
# to create dataframe with assigments
f_importance = pd.DataFrame({'cols':labels,'importance':importance})
```

```python
# sort in descending order and assign top 20
sort = f_importance.sort_values('importance',ascending=False).head(20)
```

```python
# to visualize relevance
plt.subplots(figsize=(9,3))
sns.pointplot('cols','importance',data=sort)
plt.xticks(range(20),sort.cols,rotation=80)
plt.tight_layout
plt.xlabel('Column Names')
plt.title('Most Important Features')
```



- PCA for Dimensionality Reduction

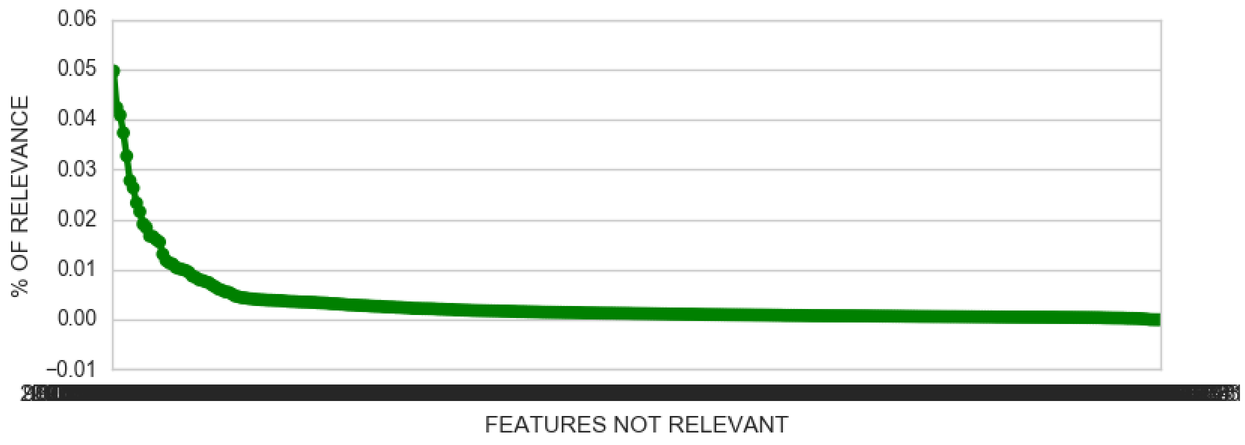
```python
# To create a set with only the 10 most relevant features
X10 = X.loc[:,f_importance.sort_values('importance',ascending=False).head(10)['cols'].values]
```

```
# to create list with column names of irrelevant features
not_in=[]
for i in X.columns:
    if i not in X10.columns:
        not_in.append(i)

X_other = X.loc[:,not_in]
```

```
#CHECK: feature importance with PCA
from sklearn.decomposition import PCA
pca = PCA(n_components=None)
X_none = pca.fit_transform(X_other)
```

```
# to visualize importance of other features
plt.subplots(figsize=(9,3))
sns.pointplot(pca.explained_variance_ratio_[::-1],pca.explained_variance_ratio_,
              color='green',markers='.')
plt.ylabel('% OF RELEVANCE')
plt.xlabel('FEATURES NOT RELEVANT')
```



```
# to compress the features
pca2 = PCA(n_components=5)
comp = pca2.fit_transform(X_other)
comp.shape
```

```
(49352, 5)
```

```
# to create final X set with 15 features
coln=['cm1','cm2','cm3','cm4','cm5']
for i,x in zip(coln,range(5)):
    X10[i] = comp[:,x]

X15 = X10.copy()
X15.shape
```

```
(49352, 15)
```

```
# to compress columns in test set
test_irrelevants = X_test.loc[:,X_other.columns].copy()
test_comp = pca2.fit_transform(test_irrelevants)
X_10test = X_test.drop(X_other.columns,axis=1)
for i,x in zip(coln,range(5)):
    X_10test[i] = test_comp[:,x]

X15test = X_10test.copy()
X15test.shape
```

```
(74659, 15)
```

FINAL SETS: X15 & X15test

- Implementing Random Forest with Cross-Validation

```
# to get scores for accuracy and log loss for Random Forest with cross-validation
from sklearn.model_selection import cross_val_score
from sklearn.metrics import log_loss
```

```
RF_scores_none = cross_val_score(
    estimator=RandomForestClassifier(n_estimators= 1000, random_state=0, n_jobs=-1),
                 X=X15, y=y, cv=10, n_jobs= -1, scoring = None,)
```

```
RF_scores_negloglss = cross_val_score(
    estimator=RandomForestClassifier(n_estimators= 1000, random_state=0, n_jobs=-1),
                 X=X15, y=y, cv=10, n_jobs= -1, scoring= 'neg_log_loss')
```

```
print('Prediction Accuracy: {:.2%}'.format(RF_scores_none.mean()))
print('Log Loss: {:.2%}'.format(RF_scores_negloglss.mean()))
```

```
Prediction Accuracy: 71.31%
Log Loss: -64.40%
```

- Implementing Logistic Regression with Pipeline

```
# to implement a Logistic Regression for comparison purposes
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
# Pipeline: a handy tool to automate pre-processing tasks.
# I am using it to scale the data (necessary for Logistic Regression)
pipe_lr = Pipeline([('scl',StandardScaler()),
                    ('clf',LogisticRegression(C=1000.0,random_state=1))])
```

```
# initiate the estimator using the pipeline instead
LR_scores_none = cross_val_score(estimator= pipe_lr,
               X=X15, y=y, cv=10, n_jobs= -1, scoring = 'accuracy')
```

```
LR_scores_neglogloss = cross_val_score(estimator=pipe_lr,
                X=X15, y=y, cv=10, n_jobs= -1, scoring = 'neg_log_loss')
```

```
print('Prediction Accuracy: {:.2%}'.format(LR_scores_none.mean()))
print('Log Loss: {:.2%}'.format(LR_scores_neglogloss.mean()))
```

```
Prediction Accuracy: 69.72%
Log Loss: -69.23%
```

```
# to look for best 'C' parameter
from sklearn.model_selection import GridSearchCV

param_grid = {'C': [0.001,0.01,0.1,1,10,100,1000,2000]}
grid1 = GridSearchCV(LogisticRegression(random_state=1), param_grid, cv= 10, scoring= 'accuracy',
                    n_jobs = -1)
grid2 = GridSearchCV(LogisticRegression(random_state=1), param_grid, cv= 10, scoring= 'neg_log_loss',
                    n_jobs = -1)

grid1.fit(X15,y)
grid2.fit(X15,y)

print('Accuracy %: {:.2%}'.format(grid1.best_score_))
print('with parameter: ',grid1.best_params_)
print('Log Loss: {:.2%}'.format(grid2.best_score_))
print('with parameter: ',grid2.best_params_)
```

```
Accuracy %: 69.67%
with parameter:  {'C': 1000}
Log Loss: -69.14%
with parameter:  {'C': 2000}
```

- Final Submission: Random Forest

Due to time contraint I will submit my results using Random Forest

```
# to use CV to estimate my accuracy using all the data before PCA
RF_accuaracy = cross_val_score(estimator=RandomForestClassifier(n_estimators= 1000, random_state=0, n_jobs=-1),
               X=X, y=y, cv=10, n_jobs= -1, scoring = None,)
```

```
print('Prediction Accuracy: {:.2%}'.format(RF_accuaracy.mean()))
```

```
Prediction Accuracy: 69.03%
```

I believe this result is due to the small estimator number (1000 trees). The more data used, the more trees we need

```
# to initiate final estimator wit 2000 trees
forest_final = RandomForestClassifier(n_estimators=2000, random_state=0, n_jobs=-1)
forest_final.fit(X15,y)
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_split=1e-07, min_samples_leaf=1,
            min_samples_split=2, min_weight_fraction_leaf=0.0,
            n_estimators=2000, n_jobs=-1, oob_score=False, random_state=0,
            verbose=0, warm_start=False)
```

```
predictions = forest_final.predict(X15test)
predictions_prob = forest_final.predict_proba(X15test)
```

```python
# array of probabilities assigned to each class
predictions_prob
```

```
array([[ 0.261875  ,  0.48975   ,  0.248375  ],
       [ 0.29145   ,  0.3325    ,  0.37605   ],
       [ 0.25371667,  0.4565    ,  0.28978333],
       ...,
       [ 0.193     ,  0.579     ,  0.228     ],
       [ 0.24908333,  0.457     ,  0.29391667],
       [ 0.28278333,  0.387     ,  0.33021667]])
```

```python
# to create a DataFrame with previous array
submission = pd.DataFrame()
```

```python
submission['listing_id'] = test['listing_id'].copy()
submission['high'] = predictions_prob[:,0]
submission['medium'] = predictions_prob[:,2]
submission['low'] = predictions_prob[:,1]
```

```python
submission.head(4)
```

	listing_id	high	medium	low
0	7142618	0.261875	0.248375	0.48975
1	7210040	0.291450	0.376050	0.33250
100	7103890	0.253717	0.289783	0.45650
1000	7143442	0.279750	0.343750	0.37650

```python
# to create a CSV file
submission.to_csv('submission.csv',index=False)
```