The Great Real Estate Data Challenge(XGBoosted Model)

Extreme Gradient Boosting is a very powerful machine learning technique that corrects its own mistakes iteratively while boosting the performance of each subsequent decision tree. In the original dataset we were given, the expectation was to find & predict house prices based on a number of variables relating to the expected outcome. From the original prompt, we were to build an optimized model that was able to identify the potential gain from housing investments and categorize them based on that gain.

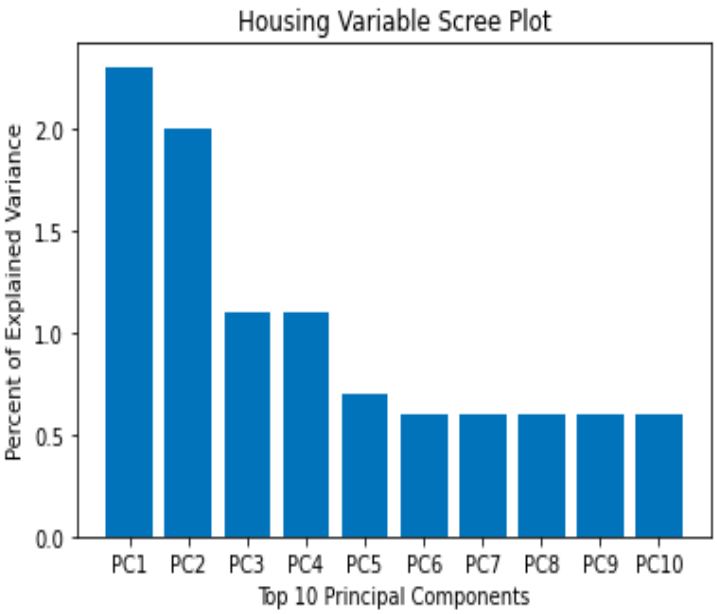
Like all data science projects, our first step was to take an initial look at our data, and clean it up as best as possible. Our given data was already split into two files, one for training and one for testing. Since we were still building our model, obviously we began with our training file, 11 columns in all:

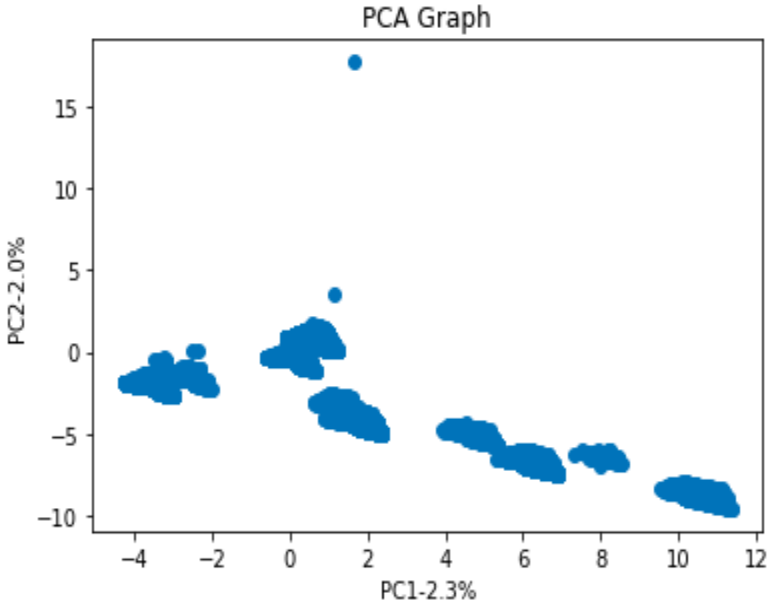
- Year
- Date
- Locality
- Address
- Estimated Value
- Sale Price
- Property[Housing]
- Residential[Type]
- Num_rooms
- Carpet_area
- Property_Tax_Rate

Eleven dimensions of variability isn't very taxing to a single machine, however some of these columns contain over 500,000 unique values(Address, Date, SalePrice(Not Useful in modeling)), and unless we want our machines to finish modeling sometime next week, we're going to need to make choice. If we look into our other variables, we've already got the years, and we could extract just the month from each date to simplify one column. The real thorn in the side of our model lies in our Address column. It's too much information in one column with numbers, locations, streets, and every single one being its own unique value. In light of that I made the decision to just remove the column entirely. In the end, the variability of the column does not do much for models & slows down our learning process too much.

For the rest of the columns, I still wanted to understand how the variables would affect our models. By compressing our variables to their two top dimensions, we could easily identify the variability of each sample and any variable added to future models. PCA(Principal Component Analysis) is very handy to display exactly that. Before compressing our data we had to make a few modifications. PCA doesn't handle categorical data well, like at all. Prior to running our Analysis we'd have to encode those variables with dummies. This does affect the performance of our PCA, dropping the power of our first Principal Components significantly (pc1=40% --> pc1=2.3%). We're going to have to take this hit since we have so much variability in our dataset. The outputs of compression components and loading scores are pictured below:

PCA Discovery






```
Variable importance ranked 1-10: num_rooms                0.444885
carpet_area                    0.428199
Residential_Condominium        0.409572
Property_Condo                 0.405183
Residential_Detached House     0.224717
Residential_Triplex            0.220700
Property_Three Family          0.218968
Property_Single Family         0.218532
Residential_Fourplex           0.151676
Property_Four Family           0.150538
dtype: float64
```

.

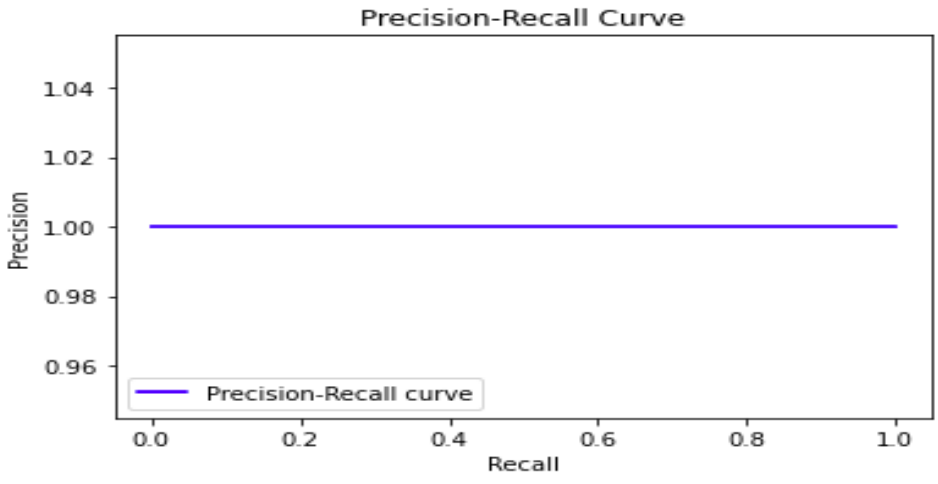
Not so great, our first two PCs only account for ~5% of the total variation among our data. That doesn't mean we glean no information from our compression however. The loading scores among a majority of the Principal components Align almost identically & even before we added our dummies, our top loading scores remained consistent. So, with a grain of salt we can take these calculations at face value. What do these loading scores mean? Well, essentially they are giving us the effect each variable has on the outcome. We have the num_rooms & carpet_area columns defining most of the variability here with Residential_Condoniums & Property_Condo right behind them (Loading Scores defined in power 0-1).

What conclusions or real-world examples could we link to the findings? Well, to sum up the largest affectation of our variables, we might say "Condominiums with large floor plans tend to dominate the housing market's variability in terms of each property's gain." This hypothesis would strongly mimic the real world market where we see Condos as a prime target for realtors and multi-level marketing (pyramid schemes) to sell for the largest profit margins possible within the real estate market. This means the data we're working with has a great resemblance to what you might find in the real world market. Our component compression can confirm the validity of our data to train our XGBoost model.

Our model was fit by transforming the frame into a DataMatrix(categorically enabled to save from manually encoding the variables), and connecting our model to the transposed matrix. Dmatrices allow more flexible data fits to be placed in the XGBoost API to save from some major headaches when transforming your data.

```
In [17]: #initiate our Dmatrix
         dtrain = xgb.DMatrix(X, label=y, enable_categorical=True)

In [18]: #train our XGBoost as a linear regression model to estimate house prices
         params = {'objective': 'reg:squarederror', 'eval_metric' : 'aucpr' }
         lnr_xgb = xgb.train(params, dtrain)

In [19]: #with the same Dmatrix, we can predict the outcomes for our training set
         y_pred = lnr_xgb.predict(dtrain)
         y_pred

Out[19]: array([974451.56, 202426.11, 777268.  , ..., 302950.25, 302950.25,
                509045.94], dtype=float32)

In [20]: #We have to change our outputs to binary to assess the accuracy & precis
         #A threshold classfies outputs as 1 & 0 for confusion matrix to predict
         threshold = np.median(y_pred)
         # Bins classify outputs as correct or incorrect
         y_bin = np.where(y_pred >= threshold, 1, 0)
```

A model is only as good as its fit, and we can test this one's accuracy in a number of ways. For this current model, I used the precision-recall curve to measure how well our model could return an accurate, relevant answer.



Frankly the data we used was pretty clean, used for practice purposes only and our XGBoost comes quite optimized out of the box. For more complex data we might use a Grid Search with cross validation to find the best hyperparameter. But, it seems our data works great straight out of the box. With our newly created model, we could begin feeding in our testing data to project new sale prices onto our data. We established a new data matrix from our test set in the same fashion as the original, and ran it through the predicted function of our XGB model.

Our last steps were the simplest, creating new segments for our predicted house prices. By subtracting our estimated value from the new sales value, we could create a new column; "Gain." Gain would represent the total profit made from the investment in the property. With our new gain column we could begin segmentation by using the quantile function to create quartiles marking in the range of our new gain column. These quantiles were transformed into the four segments our original prompt had requested. Budget, Standard, Valuable, and Premium. Inserting all this new data into the testing dataframe, we could now export the predicted data for further analysis.