Greater Seattle Area Housing–Sales Price Prediction

Francis Tan

2017-02-27

Abstract

The goal of this project is to predict the sale price of a property by employing various predictive machine learning models in an ensemble given housing data such as the number of bedrooms/bathrooms, square footage, year built as well as other less intuitive variables as provided by the Zillow API.

Introduction

The Greater Seattle Area housing market has gone through a dramatic price increase in recent years with the median valuation at $609,100, an 11.3% increase from last year and a 4.7% increase forecasted for the next year[1]. Because of the dramatic change that has occurred in a short period of time, it has become increasingly difficult to predict property values. We believe that a machine learning model can predict a property value with reasonable accuracy given the property features and the time aspect (*lastSoldDate*).

Data Collection and Preprocessing

Data Collection Process

The most important element of any data science project is the data itself. This project heavily utilizes data from Zillow, a digital real estate destination for buyers, sellers, and agents. Fortunately, Zillow provides a public API which provides a convenience to an otherwise tedious task. Although the availability of a public API has made the data collection process simple, there are some limitations that we had to be cognizant of. Our vision was to start with a "seed" property which in turn would collect "comps" or comparables. Comps are simply other properties that have similar features to our seed property. This will provide a buyer an idea of what the value of the property should be.

The first limitation is that the full set of information that we were looking for cannot be extracted from one API endpoint. Zillow does not provide an endpoint which returns property information of comps given a seed property. What it provides instead is one endpoint that returns a list of comp property IDs (Zillow Property ID or ZPID) given a seed property address and a separate endpoint that returns property information given a ZPID. Furthermore, the comp endpoint returns a maximum of 25 comps per seed property. Thus the collection process is divided into three steps:

1. Collect comp IDs given a property address using *GetDeepSearchResults*.
2. Loop through each ZPID, collect 25 more comps for each, and append results to list of the other ZPIDs.
3. Collect property information for each ZPID collected using *GetDeepComps*.

The second limitation is that Zillow has limited the number of calls allowed per day to 1000. This poses a problem if one's intent was to collect a significant amount of data. This limits our collection process further since we had to resort to making two calls. A simple solution was to include a sleep timer of 24 hours when a

[1] Source: Zillow Data as of February 2017.

call encounters a rate limit warning. Although somewhat inconvenient, the solution achieved what we needed to accomplish.

Training Data

The table below is just a sample of what the training data looks like. We've removed many of the columns to make sure the table fits in the page. This is only to provide an idea of the formatting.

street	city	zip	finishedSqFt	lastSoldDate	lastSoldPrice
18314 48th Ave W	Lynnwood	98037	1223	11/07/2016	315000
19011 Grannis Rd	Bothell	98012	1296	10/06/2016	353000
2625 189th St SE	Bothell	98012	3226	02/01/2016	405000
719 John Bailey Rd	Bothell	98012	1200	07/22/2016	360000
5113 212th St SW	Lynnwood	98036	2300	05/12/2016	430000
21132 49th Ave W	Lynnwood	98036	1654	08/01/2016	305000
4715 212th St SW	Lynnwood	98036	1314	08/17/2016	315000
4717 212th St SW	Lynnwood	98036	2112	08/12/2016	445500
7525 Maltby Rd	Snohomish	98296	1936	09/30/2016	491000
22513 44th Ave W	Mountlake Terrace	98043	1344	05/12/2016	443190

Printing the *shape* attribute shows that we have 2826 observations and 23 columns.

```
>>> training_raw.shape
(2826, 23)
```

Finally, we have the following columns

```
>>> training_raw.dtypes
zpid                      int64
street                   object
city                     object
state                    object
zip                      object
FIPScounty               object
useCode                  object
taxAssessmentYear        object
taxAssessment            object
yearBuilt                object
lotSizeSqFt              object
finishedSqFt              int64
bathrooms                object
bedrooms                 object
lastSoldDate             object
lastSoldPrice             int64
zestimate                object
zestimateLastUpdated     object
zestimateValueChange     object
zestimateValueLow        object
zestimateValueHigh       object
zestimatePercentile       int64
region                   object
dtype: object
```

Since the goal of this project is to predict the sale price, it is obvious that the *lastSoldPrice* should be the response variable while the other columns can act as feature variables. Of course, some processing such as dummy variable conversion is required before training begins.

Data Processing

The next step is to process and clean the data. First let's take a look at each variable and decide which ones need to be excluded. ZPID and street address logically do not affect sales price and thus can be excluded. Street address may explain some sale price variabilty, however it requires further processing for proper formatting, that is, we must eliminate unit numbers, suffix standardization (Dr. vs Drive), etc. This proves to be a difficult task that is beyond the scope of this project. Further, the effects of this variable is closely related to region. We have chosen to exclude it here but may be worth exploring further in the future. Finally, the state variable can also be excluded here as we are keeping the scope of this project to WA only.

```
>>> training = training_raw  # Save original data intact
>>> training.drop(['zpid', 'street', 'state'], axis=1, inplace=True)
>>> training.dtypes
city                    object
zip                     object
FIPScounty              object
useCode                 object
taxAssessmentYear       object
taxAssessment           object
yearBuilt               object
lotSizeSqFt             object
finishedSqFt             int64
bathrooms               object
bedrooms                object
lastSoldDate            object
lastSoldPrice            int64
zestimate               object
zestimateLastUpdated    object
zestimateValueChange    object
zestimateValueLow       object
zestimateValueHigh      object
zestimatePercentile      int64
region                  object
dtype: object
```

We can see that many of these variables are of type *object*. We'll need to convert these to the appropriate types. Most of these columns, excluding date columns, can be converted to numeric.

```
cols = training.columns[training.columns.isin([
    'taxAssessmentYear',
    'taxAssessment',
    'yearBuilt',
    'lotSizeSqFt',
    'finishedSqFt',
    'bathrooms',
    'bedrooms',
    'lastSoldPrice',
    'zestimate',
    'zestimateValueChange',
    'zestimateValueLow',
    'zestimateValueHigh',
    'zestimatePercentile'
```

```
])]
for col in cols:
    training[col] = pd.to_numeric(training[col])
```

Now let's convert *lastSoldDate* and *zestimateLastUpdated* to dates.

```
cols = training.columns[training.columns.isin([
    'lastSoldDate',
    'zestimateLastUpdated'
])]
for col in cols:
    training[col] = pd.to_datetime(training[col],
infer_datetime_format=True)
```

One problem with the *datetime* data type is that it will not work with scikit-learn. So we need to convert this to a numerical type. One way to resolve this is to separate the date columns into year, month, and day.

```
training = training.assign(
    lastSoldDateYear=pd.DatetimeIndex(training['lastSoldDate']).year,
lastSoldDateMonth=pd.DatetimeIndex(training['lastSoldDate']).month,
    lastSoldDateDay=pd.DatetimeIndex(training['lastSoldDate']).day,
    zestimateLastUpdatedYear=pd.DatetimeIndex(
        training['zestimateLastUpdated']
    ).year,
    zestimateLastUpdatedMonth=pd.DatetimeIndex(
        training['zestimateLastUpdated']
    ).month,
    zestimateLastUpdatedDay=pd.DatetimeIndex(
        training['zestimateLastUpdated']
    ).day
)
training.drop(['lastSoldDate', 'zestimateLastUpdated'], axis=1,
inplace=True)
training.dtypes
```

Next we need to see which of these variables need to be converted to factor variables. City, state, zip, FIPScounty, useCode, and region all qualify. One thing to caution when creating dummy variables is the number of unique categories each column has. Large number of categories may be impractical for this project as it requires a significant amount of computing resources.

```
>>> training['city'] = training['city'].astype('category')
>>> training['city'].describe()
count        2826
unique         37
top       Seattle
freq          506
Name: city, dtype: object
>>> training['zip'] = training['zip'].astype('category')
>>> training['zip'].describe()
count      2826
unique       52
top       98033
freq        206
Name: zip, dtype: object
>>> training['FIPScounty'] = training['FIPScounty'].astype('category')
>>> training['FIPScounty'].describe()
```

```
count      2826
unique        3
top       53033
freq       2127
Name: FIPScounty, dtype: object
>>> training['useCode'] = training['useCode'].astype('category')
>>> training['useCode'].describe()
count             2826
unique               2
top        SingleFamily
freq              2785
Name: useCode, dtype: object
>>> training['region'] = training['region'].astype('category')
>>> training['region'].describe()
count        2826
unique        147
top       Bothell
freq          257
Name: region, dtype: object
```

We can see that none of these variables have an unreasonably high number of unique categories with the exception of region. It contains 147 categories which may be too high, however, we will assume that our machine can handle this for now.

Let's take a look at our columns now.

```
>>> training.dtypes
city                            category
zip                             category
FIPScounty                      category
useCode                         category
taxAssessmentYear                float64
taxAssessment                    float64
yearBuilt                        float64
lotSizeSqFt                      float64
finishedSqFt                       int64
bathrooms                        float64
bedrooms                         float64
lastSoldPrice                      int64
zestimate                        float64
zestimateValueChange             float64
zestimateValueLow                float64
zestimateValueHigh               float64
zestimatePercentile                int64
region                          category
lastSoldDateDay                    int32
lastSoldDateMonth                  int32
lastSoldDateYear                   int32
zestimateLastUpdatedDay            int32
zestimateLastUpdatedMonth          int32
zestimateLastUpdatedYear           int32
dtype: object
```

Before we convert the categorical columns to dummy variables, let's look at the correlations compared to the sales price.

```
>>> training.corr()['lastSoldPrice'].sort_values(ascending=False,
inplace=False)
lastSoldPrice              1.000000
zestimateValueLow          0.980385
zestimate                  0.975082
zestimateValueHigh         0.970668
taxAssessment              0.882125
finishedSqFt               0.705464
bathrooms                  0.529689
bedrooms                   0.358294
zestimateLastUpdatedDay    0.236169
zestimateValueChange       0.231479
yearBuilt                  0.168197
lotSizeSqFt                0.060767
lastSoldDateMonth          0.031076
lastSoldDateYear           0.023216
zestimateLastUpdatedYear   0.011326
taxAssessmentYear          0.002857
lastSoldDateDay           -0.037264
zestimatePercentile             NaN
zestimateLastUpdatedMonth       NaN
Name: lastSoldPrice, dtype: float64
```

As suspected, *zestimate* columns are highly correlated to the sales price. A *zestimate* is essentially Zillow's predicted value. Since we are trying to achieve the same thing in this project, let's not include Zillow's efforts here.

```
training.drop(['zestimate', 'zestimateLastUpdatedYear',
               'zestimateLastUpdatedMonth', 'zestimateLastUpdatedDay',
               'zestimateValueChange', 'zestimateValueLow',
               'zestimateValueHigh', 'zestimatePercentile'],
              axis=1, inplace=True)
```

Here are the fininshed columns.

```
>>> training.dtypes
city                 category
zip                  category
FIPScounty           category
useCode              category
taxAssessmentYear     float64
taxAssessment         float64
yearBuilt             float64
lotSizeSqFt           float64
finishedSqFt            int64
bathrooms             float64
bedrooms              float64
lastSoldPrice           int64
region               category
lastSoldDateDay         int32
lastSoldDateMonth       int32
lastSoldDateYear        int32
dtype: object
>>> training.describe()
       taxAssessmentYear  taxAssessment    yearBuilt    lotSizeSqFt  \
```

```
count        2823.000000   2.823000e+03  2824.000000    2810.000000
mean         2014.983351   5.323108e+05  1969.516289   11807.443772
std             0.317356   3.862534e+05    24.165808   17980.249333
min          2008.000000   9.700000e+04  1900.000000     814.000000
25%          2015.000000   3.186000e+05  1955.000000    7200.000000
50%          2015.000000   4.277000e+05  1971.000000    8738.000000
75%          2015.000000   6.080000e+05  1987.000000   11884.000000
max          2015.000000   5.776000e+06  2016.000000  765236.000000

       finishedSqFt    bathrooms     bedrooms  lastSoldPrice
lastSoldDateDay  \
count   2826.000000  2810.000000  2824.000000   2.826000e+03
2826.000000
mean    2168.015216     2.355338     3.495397   7.165524e+05
17.213376
std      914.006470     0.887511     0.833232   4.927167e+05
9.022143
min      520.000000     0.100000     0.000000   1.500000e+05
1.000000
25%     1520.000000     2.000000     3.000000   4.270000e+05
10.000000
50%     2000.000000     2.100000     3.000000   5.772500e+05
17.000000
75%     2600.000000     3.000000     4.000000   8.250000e+05
25.000000
max     6753.000000     7.000000     9.000000   5.800000e+06
31.000000

       lastSoldDateMonth  lastSoldDateYear
count        2826.000000       2826.000000
mean            6.523355       2015.995046
std             3.016119          0.070223
min             1.000000       2015.000000
25%             4.000000       2016.000000
50%             6.500000       2016.000000
75%             9.000000       2016.000000
max            12.000000       2016.000000
```

	taxAssessmentYear	taxAssessment	yearBuilt	lotSizeSqFt	finishedSqFt	bathrooms
count	2823.000000	2.823000e+03	2824.000000	2810.000000	2826.000000	2810.000000
mean	2014.983351	5.323108e+05	1969.516289	11807.443772	2168.015216	2.355338
std	0.317356	3.862534e+05	24.165808	17980.249333	914.006470	0.887511
min	2008.000000	9.700000e+04	1900.000000	814.000000	520.000000	0.100000
25%	2015.000000	3.186000e+05	1955.000000	7200.000000	1520.000000	2.000000
50%	2015.000000	4.277000e+05	1971.000000	8738.000000	2000.000000	2.100000
75%	2015.000000	6.080000e+05	1987.000000	11884.000000	2600.000000	3.000000
max	2015.000000	5.776000e+06	2016.000000	765236.000000	6753.000000	7.000000

	bedrooms	lastSoldPrice
count	2824.000000	2.826000e+03
mean	3.495397	7.165524e+05
std	0.833232	4.927167e+05
min	0.000000	1.500000e+05
25%	3.000000	4.270000e+05
50%	3.000000	5.772500e+05
75%	4.000000	8.250000e+05
max	9.000000	5.800000e+06

We can see that the median price in our data set is $577,000 which is quite high!

As it turns out, we have NaNs in our data as seen below:

```
print(training.isnull().any())
```

```
city                  False
zip                   False
FIPScounty            False
useCode               False
taxAssessmentYear      True
taxAssessment          True
yearBuilt              True
lotSizeSqFt            True
finishedSqFt          False
bathrooms              True
bedrooms               True
lastSoldPrice         False
region                False
lastSoldDateDay       False
lastSoldDateMonth     False
lastSoldDateYear      False
dtype: bool
```

We need to remove these as NaNs will not work in the training process.

```
training.dropna(inplace=True)
print(training.isnull().any())
```

```
city                  False
zip                   False
FIPScounty            False
useCode               False
taxAssessmentYear     False
taxAssessment         False
yearBuilt             False
lotSizeSqFt           False
finishedSqFt          False
bathrooms             False
bedrooms              False
lastSoldPrice         False
region                False
lastSoldDateDay       False
lastSoldDateMonth     False
lastSoldDateYear      False
dtype: bool
```

Dummy Variables

Finally, let's make the dummy variable conversion. This can easily be achieved using the *get_dummies* function.

```
training = pd.get_dummies(training, columns=['city', 'zip',
'FIPScounty',
                                            'useCode', 'region'],
                          drop_first=True)
print(training.shape)
```

```
(2794, 247)
```

We have 245 columns as shown above, which we can verify by adding the number of unique categories - 1 with the number of non-categorical columns.

Training

Now that we have prepared the data, we can begin the training process. Since we do not have new test data on hand, we will need to split a portion for final evaluation. Let's set aside 20% of the data for just that. We achieve this using scikit-learn's *train_test_split* function.

```
>>> x_train, x_test, y_train, y_test = train_test_split(
...     training.drop('lastSoldPrice', axis=1),
...     training['lastSoldPrice'],
...     test_size=0.2,
...     random_state=1201980
... )
...
```

The general plan here is to train several different models, make predictions for each of those models, and use those predictions to train and predict a separate ensemble model. In essence, we will have two layers of training/prediction processes. Each model will be cross-validated with 10-folds using the K-fold method and will utilize a grid-search to find the best combination of parameters.

Ridge Regression

Ridge regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of coefficients[2]. We are also building a grid of alphas that range from 0.1 to 10 in increments of 0.2. Since we have a relatively small dataset, we can afford to build a large grid.

```
# ridge = linear_model.Ridge()
# param_grid = {
#     'alpha': np.arange(0.1, 10, 0.1)
# }
# grid = GridSearchCV(ridge, param_grid, cv=10, n_jobs=-1,
#                     scoring='neg_mean_squared_error')
# grid.fit(x_train, y_train)
# print(grid.best_params_)
# print(math.sqrt(abs(grid.best_score_)))
```

As we can see, the grid search has found that an alpha of about 7.4 produced the best RMSE at 198433. This is not nearly as accurate as I would like it to be but it is a good start. We will need to build models and

[2] Scikit-learn Documentation-Ridge Regressino (http://scikit-learn.org/stable/modules/linear_model.html#ridge-regression)

combine R them in an ensemble. We hope to produce better results from the combined predictions. Let's store the predictions of this model for now.

```
# ridge_prediction = pd.DataFrame(grid.predict(x_train))
```

Support Vector Machine

```
# svm = svm.SVR()
# param_grid = {
#     'C': np.arange(1.0, 100, 1)
# }
# grid = GridSearchCV(svm, param_grid, cv=10, n_jobs=-1,
#                      scoring='neg_mean_squared_error')
# grid.fit(x_train, y_train)
# print(grid.best_params_)
# print(math.sqrt(abs(grid.best_score_)))
#
# svm_prediction = pd.DataFrame(grid.predict(x_train))
#
# #' ## Lasso
#
# #' Lasso is a generalized linear model that has a tendency to prefer
solutions
# #' with fewer parameter values[^lasso]. Our particular project isn't
considered
# #' a high dimensional problem and thus this model should be
appropriate.
#
# #' [^lasso]: Scikit-learn Documentation-Lasso (http://scikit-
learn.org/stable/modules/linear_model.html#lasso)
#
# lasso = linear_model.Lasso()
# param_grid = {
#     'alpha': np.arange(0.1, 10, 0.1),
#     # 'max_iter': np.arange(2500, 2500, 1)
#     'max_iter': np.array([100])
# }
# grid = GridSearchCV(lasso, param_grid, cv=10, n_jobs=-1,
#                      scoring='neg_mean_squared_error')
# grid.fit(x_train, y_train)
# print(grid.best_params_)
# print(math.sqrt(abs(grid.best_score_)))
#
# lasso_prediction = pd.DataFrame(grid.predict(x_train))
#
# #' ## Decision Tree
#
# #' The Decision Tree model is one of the more simple and
interpretable models.
# #' We have chosen to include it here for its simplicity.
#
# tree_reg = tree.DecisionTreeRegressor()
# param_grid = {}
```

```
# grid = GridSearchCV(tree_reg, param_grid, cv=10, n_jobs=-1,
#                     scoring='neg_mean_squared_error')
# grid.fit(x_train, y_train)
# print(math.sqrt(abs(grid.best_score_)))
# tree_prediction = fit.predict(x_train)
```

Ensemble

Now that we have trained and predicted using individual models separately, we need to use those predictions as input for a separate training model. The below table shows the CV validated RMSEs for our individual models.

Model	RMSE
Ridge Regression	198433
Support Vector Regression	0
Lasso	0
Decision Tree	0