MLND Capstone Report

Identifying Property Bargains

Yavor Paunov

January 16, 2018

1 Definition

1.1 Project Overview

Machine learning has already been used in the real-estate business, on platforms where buyers and sellers meet. Services such as Zillow in the United States and Booli in Sweden currently show the estimated value of the houses listed on their platforms. Having this point of reference is valuable for both buyer and seller. As a seller it's important to set a price that is not too low, while as a buyer the goal is to avoid overpaying. This in turn, leads to a more efficient housing market. This problem is particularly interesting to me as someone planning to shop for a home in the not too distant future.

This project examines the potential of ensemble learning methods in helping a potential house buyer find a bargain. Ensemble learning is a technique that combines the output of multiple models to obtain a more accurate prediction than any of the individual learners.

The "House Sales in King County" dataset was used to train the model. The data was downloaded from Kaggle: `https://www.kaggle.com/harlfoxem/housesalesprediction`.

1.2 Project Statement

The focus is on helping a potential buyer pay as little as possible for their new house. To that end, I develop a command-line application which gives a prediction for the final sale price of a given house. This can help a potential user of the service identify houses where the asking price is low compared to the predicted price, and eventually buy a cheaper home. Due to the available data, reliable predictions are only be available for houses in King County, Washington. Another limitation is time – the data limits us to a specific time frame, April 2014 to April 2015. However, given another similarly structured dataset, it would be fairly straightforward to obtain predictions for a different location and time.

1.3 Metrics

The metric used for evaluating the resulting model is R^2 – the coefficient of determination between the actual prices and those predicted by the model. The formula for the metric as implemented in *scikit-learn* is

$$R^2(y, \widehat{y}) = 1 - \frac{\Sigma_{i=0}^{n_{samples}-1}(y_i - \widehat{y}_i)^2}{\Sigma_{i=0}^{n_{samples}-1}(y_i - \overline{y})^2} \tag{1}$$

where $\overline{y} = \frac{1}{n_{samples}}\Sigma_{i=0}^{n_{samples}-i} y_i$.

The highest possible value and best score is 1.0, indicating predictions identical to the actual values. The score can be negatives as well, in case the model is sufficiently bad.

The R^2 score is the one used by default for scoring regression algorithms in *sklearn* which makes it very easy to compute. Another reason for sticking with R^2 is the fact that I find it easier to interpret given its limited range of possible values, in contrast to MSE and $RMSE$. This also leads to it being applicable even when comparing model performance with different datasets.

2 Anaylsis

2.1 Data exploration

The "House sales in King County" dataset was used for the project. The dataset consists of data for house sales in the King County area in Washington for a period of about one year starting in April 2014. The following features are present in the dataset:

id	the unique id of the sale
date	the date when the house was sold
price	the final selling price in US dollars
bedrooms	the number of bedrooms
bathrooms	the number of bathrooms
sqft_living	the living area in the house in square feet
sqft_lot	the area of the whole lot on which the house is built in square feet
floors	the number of floors in the house
waterfront	whether the house is built on a waterfront (1 or 0)
view	the quality of the view from the house, integer values ranging from 0 to 4
condition	the physical condition of the house, integer value ranging from 1 to 5
grade	the overall grade of the house in terms of construction and design, integer values ranging fro
sqft_above	the size of the living area above ground (living area minus basement), in square feet
sqft_basement	the size of the basement, in square feet
yr_built	the year the house was built
yr_renovated	the year the house was renovated
zipcode	the zipcode of the area where the house is located
lat	the geographical latitude of the house
long	the longitude longitude of the house
sqft_living15	the average house square footage of the 15 closest houses
sqft_lot15	the average lot square footage of the 15 closest houses

There are 21613 samples in the dataset. The data is of very high quality – there are no missing values. Outliers were not removed since no evidence was found that any of the values are erroneous. Also, the tested models are tree based and therefore robust to outliers.

2.2 Explortory Visualization

As can be seen in figure 1 **price** was highly skewed. The figures below show exactly by how much. Calculating the price per square foot lead to lower skewness, and log transforming the data lowered it even more. This is discussed further in 3.1.

2.3 Algorithms

The project was focused on using ensemble algorithms as the solution.

The final model was chosen after experimentation, and taking into consideration the data, as well as where we stand in regards to the bias-variance trade off when using a weak learner. For example, if we consistently get a model with high bias and low variance with a weak learner, perhaps switching to boosting would be appropriate.

The strengths and weaknesses of the explored algorithms are discussed below.

1. Random Forests

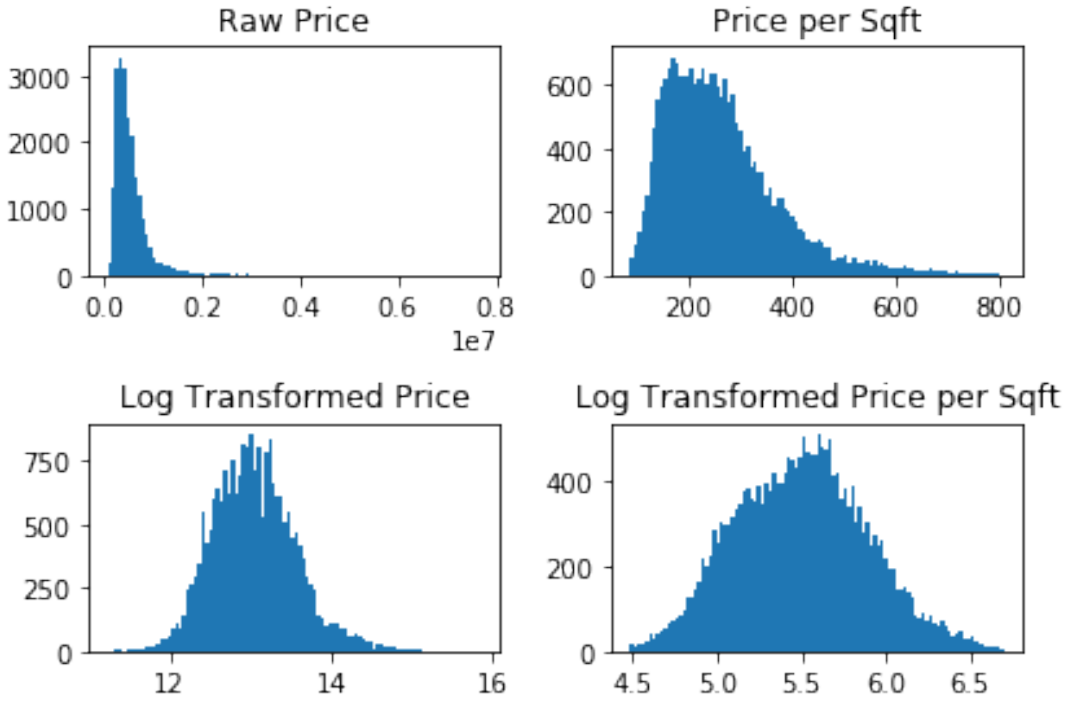

Figure 1: Distribution of house prices

Perform better than single decision trees due to reduced variance. Although bias is slightly increased, the decreased vairance more than makes up for that. Random forests are well suited for distributed computing.

With a large number of trees, it can take time to obtain predictions.

2. AdaBoost

   AdaBoost generally works well out of the box, and good results can be obtained with no parameter tuning. Overfitting tends not to be a problem.

   As a negative, noisy data can lead to overfitting since large errors are penalized harshly. In addition to that, the CPU and memory footprint can be large.

3. Gradient Boosting

   Correctly tuned parameters can greatly reduce overfitting, and generally help outperform other ensemble methods.

   Gradient boosting generally takes longer to train because trees are added sequentially. Parameter tuning is required in order to obtain the best results.

2.4 Benchmark

A linear regression model trained on the same data as the actual predictor was used as benchmark. A well tuned non-linear predictor should be able to outperform this basic model. The metric used to measure the performance of the benchmark model is the same as the one for the solution, namely the coefficient of determination.

3 Methodology

3.1 Data Preparation

Most of the features were useful without much modification. The one exception to that was feature scaling and transformation; all numerical data was passed through a min-max scaler and log transformer. Although this might have been unneccessary since we mainly focused on tree based models, feature scaling is a good practice in any case.

Few features were modified, and one outright removed. The **id** of a house has no bearing on the model's predictive powers, so it did not make the cut.

Following this paragraph is a discussion of the features which were used but required some amount of preprocessing.

3.1.1 Date

The raw value of the date column includes time of day but it is always set to 00:00:00, therefore it could be safely disregarded. The raw value we have is a string so we need to convert it to a *datetime* object, which in turn was converted to an integer equal to the number of days that have passed since the start of the year.

3.1.2 Zipcode

The **zipcode** is a categorical variable with a large number of possible values, 70 to be exact. That means that one-hot encoding it would introduce an explosion in dimensionality, so an alternative approach is preferable. As a solution to this, the mean value of the selling price per square foot of living area for each zipcode is introduced to the dataset.

3.1.3 Price

The predicted variable was changed from total sale price to sale price per square foot of living area. The value was obtained by dividing price by the area in square feet.

Initially, this was done in order to calculate the average price per square foot for each zipcode. As noted in 2.1 **price** was strongly skewed to the left. After applying log transform to the **price, skewness was reduced to a degree. However, inspecting the added *price_per_sqft** feature revealed that it was skewed much less, and applying log transformation brought the skewness drastically. The skewness of the raw **price** was measured at 4.024 while that of **price_per_sqft** was 1.248. After log transformation the values went down to 0.428 and 0.145 respectively.

Another important aspect to consider was predictions of values that fall outside the range of the

training set. Prediction on values outside the training range can be a problem for tree based methods in particular. Since the focus of this project is using ensemble methods, mainly based on decision trees, it was important to minimize that. In case of using **price** as dependent variable, there would be a chance that some values will not be seen by the model. It is not at all impossible that we might have to predict the price of a house that is much more expensive than any of the samples in the training dataset. With price per square foot this is less likely.

3.2 Implementation

3.2.1 Train test split

First of all, as mentioned earlier, the variable to predict was changed from total price to price per square foot.

```
prepared_data = prepare_data(data)

X = prepared_data.drop('price_per_sqft', axis=1)
y = prepared_data['price_per_sqft']
```

That was followed by dividing the dataset, with 25% of it being used for testing while the rest was used for training.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

3.2.2 Data Pipeline

Before doing this project I did not think about how testing data might affect the training. I became aware of this when I tried to convert the mean house price per zipcode, and realized that its value also depends on testing data but would be used for training. That led me to discover how to use pipelines, the *Pipeline* class built into *sklearn* which was used for data transformation and scaling. This ensured that when running a grid search, test data did not leak into the training data. The zipcode average was calculated based only on houses in the training dataset. The final step of the pipeline was left for the estimator.

```
from sklearn.preprocessing import MinMaxScaler, FunctionTransformer
from sklearn.pipeline import Pipeline

class ZipcodeTransformer(BaseEstimator, TransformerMixin):

    def fit(self, X, y):
        self.averages = y.groupby(X.zipcode).agg({'price_per_sqft': 'mean'})
        return self
```

```
    def transform(self, X):
        X = X.assign(zipcode_average=X['zipcode']
                     .map(self.averages.to_dict()['price_per_sqft']))
        # In case there is a zipcode that was not seen during fitting we
        # assign it the mean of all means
        X[X.zipcode_average==np.NaN]['zipcode_average'] = self.averages.mean()
        return X

def get_pipeline(estimator):
    steps = [
        ('zipcode_average', ZipcodeTransformer()),
        ('log', FunctionTransformer(np.log1p)),
        ('scaler', MinMaxScaler()),
        ('estimator', estimator)
    ]
    return Pipeline(steps)
```

Calculating the average price for a zipcode was slightly more tricky than initially anticipated. Initially, I attempted to do it with simple python lists and dicts, however that seemed unsatisfactory. Delving deep into the *numpy* and *pandas* documentation helped immensiely. Particularly helpful were the *pandas.Series* class and its methods *groupy*, *agg*, and *map*. Although not entirely difficult to get something working, getting an efficient solution using *numpy* and *pandas* was a small challenge.

3.2.3 Model Comparison

The following code was used to obtain the R^2 score for several ensemble models after fitting them on the training data, and predictin the test data:

```
estimators = [get_pipeline(GradientBoostingRegressor(random_state=42)),
              get_pipeline(AdaBoostRegressor(random_state=42)),
              get_pipeline(RandomForestRegressor(random_state=42)),
scores = map(lambda clf: clf.fit(X_train, y_train).score(X_test, y_test), estimators)
```

The table below shows how well the models performed in terms of R^2 score without any hyperparameter tuning:

Algorithm	R^2 **score**
Gradient Boosting	0.77797034981188107
Ada Boost	0.47654512137921423
Random Forest	0.76922999519128699

This was done in order to pick a model for further tuning. As can be seen in the table above, Gradient Boosting performed best.

3.3 Refinement

On the basis of the comparision between the various ensemble methods, Gradient Boosting was chosen as the algorithm to focus on.

As a final step of creating the prediction model, 5-fold cross-validated randomized search was employed in order to imporve performance on the test data. The parameters that were to be optimized were learning rate, the number of weak learners (decision stumps in this case), and the fraction of samples to use for each individual decision tree.

```
grid_search = RandomizedSearchCV(
    get_pipeline(GradientBoostingRegressor(random_state=42)),
    param_distributions={"estimator__learning_rate": expon(scale=.03),
                         "estimator__n_estimators":[100, 500, 700, 800],
                         "estimator__subsample": uniform(0., 1.) },
    random_state=42,
    cv=5)
grid_search.fit(X_train, y_train)
grid_search.score(X_test, y_test)
```

An improved r^2 score of 0.81028079137574782 was built thanks to the randomized search.

3.4 Application

After ensuring the model works, it had to be put to use as a command-line script.

The final implementation consists of two scripts: one used for training which can be found at in *src/train.py*, and another one for predictions found at *src/predict.py*.

The training script's input takes a **csv** file containing data about already completed house sales. The data is then be prepared and fed to the pipeline's **fit** method, with the resulting model being persisted on disk, using Python's *pickle* module. The path at which to save the model is also given as an argument. Optionally, a randomized search can be performed to find the most suitable hyperparametrs for the given data.

The prediction script also accepts a **csv** file, as well as the path to the *"pickled"* model. In addition to that, an argument specifying which of the rows in the **csv** file needs to be predicted. If that's not set, all rows will be predicted.

The data was divided into train and test datasets, saved at *data/train_ data.csv* and *data/test_ data.csv* respectively. This was done in order to be able to test the script with real-world data.

1. How to Use

   In order to train a model:

   ```
   src/train.py --csv_file=data/train_data.csv --model_path=model.pkl --param-search=True
   ```

In order to obtain a prediction for the fifth row in the **csv** file:

```
src/predict.py --csv_file=data/test_data.csv --model_path=model.pkl --index=3
```

4 Results

4.1 Model Evaluation and Validation

After tuning its parameters the model's accuracy rose by a few points. It is worth discussiing the parameters that resulted from that process, and how they could affect the final score. The learning rate was optimized to 0.109208986609. The learning rate, also called shrinkage, represents the contribution of each new tree added to the model. Essentially, reducing it increases training time while on the other hand helping reduce overfitting.

The number of weak learners defines how many base estimators the model consists of. The value we got from the randomized search was 700. Conversely to learning rate, a increasing the number of base estimators has the potential of increasing accuracy but at the expense of additional training time. Thus, there is a direct trade-off between number of estimators and learning rate, and a primary goal of tuning a gradient boosting model is finding a balance between the two.

The last parameter to tune was subsample – the fraction of the training data on which to train each base learner. The value obtained by the randomized search was 0.456069984217. With a value lower than 1.0 for this parameter, the algorithm is called Stochastic Gradient Boosting This is a technique which essentially combines bagging and gradient boosting. The benefit of tuning this parameter is that it can help decrease variance, and therefore help us achieve a higher score on the testing data.

In order to further ensure the model's robustness and its ability to generalize to new data, a 10-fold cross-validation score was calculated. The result of that was a mean score of 0.792154145565, a score lower than the one obtained earlier by the randomized search but still quite a bit higher than the benchmark.

4.2 Justification

The score of the final model is 0.81028079137574782, compared to the benchmark model's score of 0.71022557772698591. This is almost exactly 10% higher. On that basis alone the final stochastic gradient boosting model can be considered successful.

5 Conclusion

5.1 Free-Form Visualization

A particularly interesting plot to me is the learning curve . It shows how the model's accuracy improves as more samples are added to the training set. Figure 2 below shows the learning curve for the model configured as recommended by the randomized search.

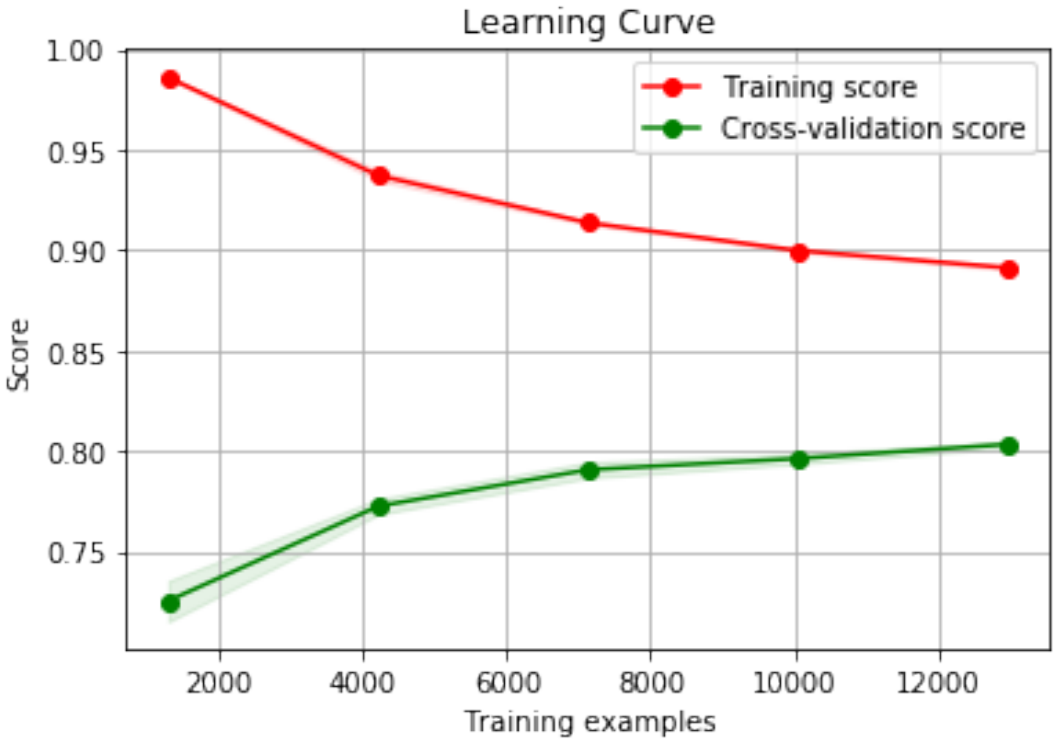

Figure 2: Plotted with code from `http://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html`

We can see that the score improves with the number of samples. Although that improvement gradually tapers off, it doesn't really stop. Based on that, my intuition tells me that working with an even bigger dataset might give as more reliable predictions.

Learning curves are useful in another way too. They help check our model for overfitting. We can identify overfitting if the training score is extremely good while the the testing one is low. In our case, the training score is indeed higher but I would argue that we still don't have overffiting. First of all, the training score is almost always higher than the test one, and in our case the difference between the two simply is not that great. Moreover, the test score is quite high in absolute terms – approximately 0.81 out of a maximum 1.00.

5.2 Reflection

The process was built around a few basic steps I learned from the previous nanodegree projects. The first step was to look at the available data, including some exploratory visualizations of it. That revealed that most of the data was numerical, except for the **zipcode** feature. That led to looking for ways to preprocess the data. Since there were many possible values for **zipcode**, one-hot encoding it would increase dimensionality too much. Therefore, the average sale price for the house's zipcode was introduced as an additional feature, while **zipcode** was removed. During this process, it was discovered that price per square foot might be a better variable to predict than total price, due to it being less skewed. The **date** feature was changed to day of year, essentially ignoring the year part of the data and only taking month and day into account, in order to pick up on any contribution seasonality would make.

Next, I divided the data into train and test datasets, and used them to compare the performance in terms of r^2 score of a few ensemble algorithms. Gradient boosting performed best with a score of 0.77797034981188107. After some hyperparameter tuning, the final model achieved a score of 0.81028079137574782.

The final step was to write the command-line application which uses the model. The application allows a model to be trained on a dataset, and optionally to performa a randomized parameter search. Predictions can be obtained by the other script composing the application.

An interesting aspect of the whole process was setting up the data preprocessing pipeline. This was not something I had to do for the previous nanodegree projects and taught me to learn about thing I did not consider before, like ensuring test data does not bleed into the training data. This is certainly going to stick with me for any future projects I might undertake.

Particularly diffifuclt for me was the initial step of the project: finding a topic to work on, and finding a good dataset. The decision to settle on this particular project was based on the fact that I might actually end up using it, or some version of it to help me buy a house in the near future. Although, to do that I will need a different dataset.

5.3 Improvement

Although this project was quite interesting to complete, it is important to note that its practical usefulness is rather limited due to the nature of the data. Mainly, the data is too far in the past to be reliable for predicting current prices. In additiona to that, the area for which we have data is limited to one county. A step to correct that would be to expand the data geocraphically, and make sure it is up-to-date.

Another possible improvement is to build a graphical user interface for the predictions. This could take the form of a web application where the user can enter data about the house, and get a predicted price. A more complex but possibly more user friendly solution would be to create a browser plugin that automatically scrapes data when a page of a house listing is visited by the user.