# Udacity MLND Program

October 7, 2019

## 1 Machine Learning Engineer Nanodegree

### 1.1 Capstone Project

Adriano Falsarella Monte

### 1.2 Course Mentor

Fernando Marcos Wittmann

### 1.3 Definition

#### 1.3.1 Project Overview

What is a fair offer to place when selling or buying a home? How to efficiently come to that fair price? The real estate area has been adopting machine learning to analyse historical data with several criterias to find a well balanced model to help predicting house prices.

After talking to Loft's CTO in a local event at Sao Paulo - Brazil, I've found out that Loft uses machine learning to predict the house prices in their business, and data science is one of their core values. In order to exercise house prices predictions, let's use the Kaggle's House Prices' getting started competition public dataset, which provides an appropriate input for this project.

To predict a house price, a model should be trained with historical containing the houses features and their corresponding house sold price. Despite that the houses in Ames - Iowa are different from the houses here in Brazil, the dataset provided in Kaggle's House Prices' competition still gets the essence of the work that would be done if the data was from a different location since most of the features would also be applicable in different contexts.

#### 1.3.2 Problem Statement

Given a dataset publicly available at Kaggle of homes in Ames - Iowa, having 79 house features, and containing 1460 entry points for training and 1459 entry points for testing, build the best model to predict house sale prices. A great model will predict the house prices as closest as possible to its actual labelled price.

In the conversation I had with Loft's CTO, he told me that with the house's dataset, they make some clusters, and for each cluster they apply a separate regression. I love how it embraces both unsupervised and supervised learning to get a refined result, so the idea is to try it out, exercising both concepts in this Capstone Project.

After clustering the data, find the best regression model for each cluster. With the trained models, to make a prediction, we need to first predict which cluster it is better represented, and then use that cluster's trained regression model to predict the house price.

### 1.3.3 Metrics

There are many evaluation methods available in scikit learn in the `sklearn.metrics` package. As per the proposed solution, we'll have to take one clustering metric and at one regression metric.

For the clustering part, the project uses the silhouette score as the metric to find the best number of clusters, since it's the one that we've used most in the course. This evaluation method is provided by scikit learn in `sklearn.metrics.silhouette_score` module.

For the regression part, the project uses the root mean squared log error (RMSLE) as the metric to find the best regression model, since it's the metric that is used as evaluation in the Kaggle competition in the Leaderboard ranking, but mainly because the squared error is aligned with the business objectives (we should predict prices with the least error possible between the prediction and the actual price). This evaluation method is also provided by scikit learn in `sklearn.metrics.mean_squared_log_error` module.

To make all these measurements and processes replicable, the project always set a constant `random_state` when applicable. Also, a fixed and public dataset was chosen, provided at Kaggle, and with a mirror in this Project's repository for both train and test data.

## 1.4 Analysis

### 1.4.1 Data Exploration

There's a brief description of each feature at Kaggle, as well as a `txt` file with detailed data description, mirrored in this Project's repository as well. Since there's lots of features, let me try to summarize them:

The dataset is very feature-detailed, providing several characteristics about the dataset's houses, such as: the sale price in dollars (the target variable), the date, type, and condition of the sale, the building class, the zoning class, the lot area in square feet, the street acccess type and road proximity, the property configuration, shape, and slope, neighborhood quality, dwelling type and style, the property's condition and material quality, the construction and remodeling date, infos about the property's roof, foundation, electrical system, exterior, masonry veneer, garage, pool, porch, deck, and basement, the property's heating, fireplace, and air conditioning quality, infos about the rooms like bathrooms, bedroom, and kitchen, and finally how functional the home is overall.

Lots of the features are numerical values, some of them are continuous like dollars, area, distance, and some of them are discrete like quantity, year, month.

One point of attention here is that `MSSubClass` number (for example) is just an identifier, as per the dataset's author description, and should be transformed into categorical data.

There are also lots of categorical features, some of them are nominal like types, styles, and shapes, and they should be one-hot-encoded, but there are also some categorical values that are ordinal, that may be ordered and transformed into numberical values, like class, grades, quality. Some of them may have their values scaled.

One point of attention here is that some ordinal features such as `LandSlope` and `GarageFinish` (for example) are ordinal values, as per the dataset's author description, and should be transformed into numerical data.

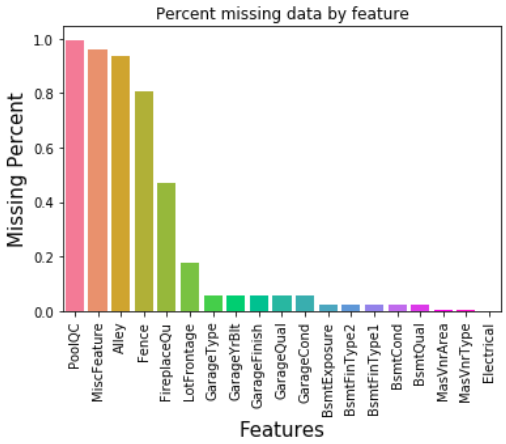
| | Id | MSSubClass | LotFrontage | LotArea | OverallQual | OverallCond | YearBuilt | YearRemodAdd | MasVnrArea | BsmtFinSF1 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 1460.000000 | 1460.000000 | 1201.000000 | 1460.000000 | 1460.000000 | 1460.000000 | 1460.000000 | 1460.000000 | 1452.000000 | 1460.000000 | ... |
| **mean** | 730.500000 | 56.897260 | 70.049958 | 10516.828082 | 6.099315 | 5.575342 | 1971.267808 | 1984.865753 | 103.685262 | 443.639726 | ... |
| **std** | 421.610009 | 42.300571 | 24.284752 | 9981.264932 | 1.382997 | 1.112799 | 30.202904 | 20.645407 | 181.066207 | 456.098091 | ... |
| **min** | 1.000000 | 20.000000 | 21.000000 | 1300.000000 | 1.000000 | 1.000000 | 1872.000000 | 1950.000000 | 0.000000 | 0.000000 | ... |
| **25%** | 365.750000 | 20.000000 | 59.000000 | 7553.500000 | 5.000000 | 5.000000 | 1954.000000 | 1967.000000 | 0.000000 | 0.000000 | ... |
| **50%** | 730.500000 | 50.000000 | 69.000000 | 9478.500000 | 6.000000 | 5.000000 | 1973.000000 | 1994.000000 | 0.000000 | 383.500000 | ... |
| **75%** | 1095.250000 | 70.000000 | 80.000000 | 11601.500000 | 7.000000 | 6.000000 | 2000.000000 | 2004.000000 | 166.000000 | 712.250000 | ... |
| **max** | 1460.000000 | 190.000000 | 313.000000 | 215245.000000 | 10.000000 | 9.000000 | 2010.000000 | 2010.000000 | 1600.000000 | 5644.000000 | ... |

8 rows × 38 columns

| | MSZoning | Street | Alley | LotShape | LandContour | Utilities | LotConfig | LandSlope | Neighborhood | Condition1 | ... | GarageType | GarageFinish |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 1460 | 1460 | 91 | 1460 | 1460 | 1460 | 1460 | 1460 | 1460 | 1460 | ... | 1379 | 1379 |
| **unique** | 5 | 2 | 2 | 4 | 4 | 2 | 5 | 3 | 25 | 9 | ... | 6 | 3 |
| **top** | RL | Pave | Grvl | Reg | Lvl | AllPub | Inside | Gtl | NAmes | Norm | ... | Attchd | Unf |
| **freq** | 1151 | 1454 | 50 | 925 | 1311 | 1459 | 1052 | 1382 | 225 | 1260 | ... | 870 | 605 |

4 rows × 43 columns

Numerical and categorical data sample




| # of outliers | feature |
|---|---|
| 228 | LotDepth |
| 208 | EnclosedPorch |
| 167 | BsmtFinSF2 |
| 143 | NeighborhoodMeanPrice |
| 121 | LotFrontage |
| 116 | ScreenPorch |
| 98 | MasVnrArea |
| 77 | OpenPorchSF |
| 69 | LotArea |
| 68 | KitchenAbvGr |
| 61 | SalePrice |
| 61 | TotalBsmtSF |
| 52 | MiscVal |

Missing values analysis, and outliers analysis

**Missing values and outliers** The Missing values chart shows us, in addition to the `count` value in the description tables above, that some features have missing values which should be handled considering each own particularities and percentage of missing values.

After handling missing values, it was noticed some feature outliers (some of these features are transformed data). These outliers datapoints (see Feature outliers table) will influence the regression model in a bad way, and they should be dropped to avoid that. It was considered an outlier if the feature datapoint was beyond the edge quartiles, the same technique applied in the Clustering Project.

### 1.4.2 Exploratory Visualization

One of the main selected features was the NeighborhoodMeanPrice, which was created by the mean SalePrice of each Neighborhood. Before starting the project, I asked my partner:

> What criterias do you think that best defines the price of a house?

And she said:

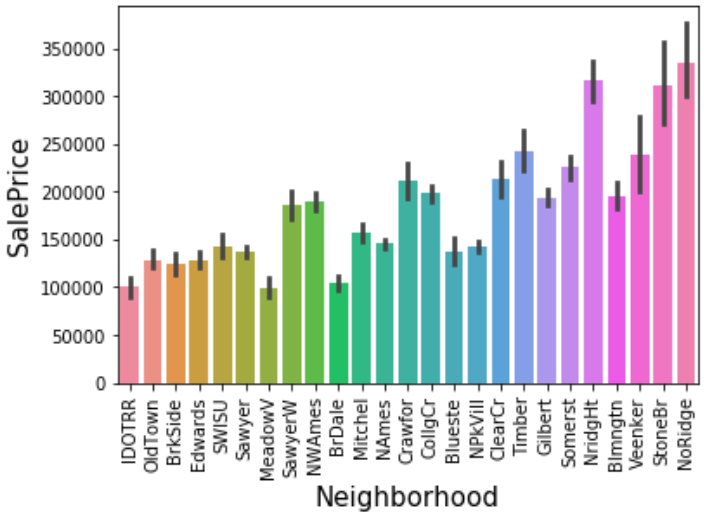
> Hmm... The house size, quality, and the neighborhood, I guess.

Well, that actually makes sense, and as expected, the Neighborhood indeed represents a lot of how much a house should be valued.

Note that before plotting, the dataset was sorted by its `SalePrice`, and the chart also followed with an ascending neighborhood sale price, with a few exceptions. Those exceptions are probably due to some outlier prices in some neighborhoods, but one thing is unquestionable: the neighborhood dictates a lot of the house prices.

```
In [5]: from IPython.display import Image
        Image("images/neighborhood_wide.png")
```

```
Out[5]:
```



### 1.4.3 Algorithms and Techniques

- Data Processing Techniques

I intended to properly process data, and guarantee that no garbage is coming as input, so considerable effort by using different methods and techniques should be put to handling missing values 1,2,3,4,5,6 (dropping features, mean imputation, mode imputation, new type imputation, fill with zero, imputation by deriving from other features), making type conversion 1,2,3 4,5,6,7,8 (type casting, mean encoding, one hot encoding, mapping values), handling outliers (dropping rows), aggregating features (sum, weighted sum, subtraction, age calculation, mean, division, weighted boosting), scaling features 1,2,3,4 (log scaling, mean normalization, min-max scaling), selecting features (problem relevance, feature variance, feature correlation, target correlation, feature importance).

- Clustering Algorithms and Techniques

Before clustering, I intended to perform a PCA transformation to avoid the curse of dimensionality (and review the Clustering with different number of components).

In this moment I intended to try different clustering models with different algorithms (such as K-Means, GMM, and Random Forest), evaluating them with `silhouette_score` to have a feeling on the performance of each algorithm (do that for each number of clusters in a reasonable range of values).

With the selected model and number of clusters through the `silhouette_score`, predict the cluster of each house in both train and test data.

(I actually ended up not using PCA, GMM and Random Forest, using KMeans only for this clustering part, as explained in detail in the Implementation section)

**How K-Means works under the hood?** For a given number of clusters `n_clusters`, K-Means will seed `n_clusters` with random centroids and assign each data point to its closest centroid, then for each centroid, it recalculates its new centroid point based on the corresponding assigned datapoints. It will then replay this process until convergence (all assigned data points are the same that was already assigned, so the centroid doesn't change), or it reaches the `max_iter` parameter, defaulted with 300 runs. Also, the result may change with different initial random centroids, so it will rerun KMeans algorithm `n_init` times (defaulted with 10 reruns), and take into consideration just the run that had the best `inertia` (an error score based on the sum of the squared distances of the samples to their closest cluster center). To predict a new datapoint it is just a matter of calculating which cluster is the closest one to this new datapoint.

- Regression Algorithms and Techniques

For each cluster found, I intended to train models with different algorithms (Logistic Regression, Stochastic Gradient Descent, XGBoost Regressor) and benchmark each of them with K-Fold cross-validation using `mean_squared_log_error` to measure and validate the performance of each algorithm, making sure that they generalize well.

Also perform hyperparameters grid search with both GridSearchCV and RandomizedSearchCV, and if applicable do some quick round of EDA and/or data processing, and rerun the benchmark with the changes in order to find the best model for each cluster. Reprocess everything after having inserted an additional 10% of noisy data to assert the robustness of the solution.

(I actually ended up not using Stochastic Gradient Descent, neither XGBoost Regressor, but ended up using a possible combination of Linear Regression, Random Forest Regressor, Gradient Boost Regressor, Lasso, Ridge, and Multi-Layer Perceptron Regressor for this regression part, as explained in detail in the Implementation section)

**How Linear Regression works under the hood?** The algorithm will initialize with an `a` and `b` for an initial line (`y = ax + b`, for simplification), calculate the error (weighted distance between the datapoints and that line), and adjust other values for `a` and `b`. It will replay this process until it reaches the least error possible (increasing or decreasing `a` or `b` would result in a worse error score), meaning that it has found the best line. To predict the result for a new datapoint, it will just fill the `x` provided in the new datapoint, and with the calculated `a` and `b`, it would then calculate the predicted `y` value.

**How Random Forest Regression works under the hood?** This algorithm will create `n_estimators` of Decision Trees, which means that it would create some random trees that for each branch it would have a rule that asks if the datapoint attends or not that rule, and depending on the result, it moves to a different branch that looks for another rule, until getting in a leaf in the tree. For each datapoint sample it would calculate which leafs it would be, and in the end, the "forest" of decision trees gives a compound result that's called "voting". It will then predict a new datapoint result with the majority value that the decision trees have "voted" (ended in their leafs), and in the case of the regression, a weight and balanced value based on the leafs "votings". The algorithm also provides us to set: the `max_depth` of the decision trees, a `min_samples_split` required to make another branch from an existing node, and other related parameters.

**How Gradient Boost Regressor works under the hood?** This algorithm works similarly to Random Forest, but instead of having several Decision Trees working in parallel, and using a majority voting system, it will make a chain of Decision Trees, one knowing the results of the previous one, calculating their errors (`loss` function), and optimizing (changing) its result by shrinking the former tree's nodes weighted by a `learning_rate` ratio by calculating a "gradient" that tells where the new tree should go, having the known former trees errors. It will then proceed with the chain until it reaches `n_estimators` or the calculated "gradient" is negative (which means they already got the best tree. Prediction and parameter tuning works similarly as the Random Forest.

**How Lasso works under the hood?** Lasso is a type of Linear Regression that shrinks the datapoints (that's why it provides a `copy_X` parameter, so it won't mutate the original `X`) in order to simplify the regression, be less prone to noisy data, handle multicollieanity better, and perform better predictions. This process of penalizing the regression coefficients is called Regularization, and in the case of Lasso, it uses the L1 Regularization, which would add to the error function the sum of the absolute coefficient weights (the `a` and `b` explained in the Linear Regression topic above) to encourage having the simplest formula possible, preventing an overfitting. The process and prediction is then very similar to Linear Regression.

**How Ridge works under the hood?** Ridge is also a type of Linear Regression, that also shrinks the datapoints as Lasso algorithm does, but instead of using L1 Regularization, it uses L2 Regularization as the additional error function. L2 Regularization works similarly as the L1 Regularization, but it actually returns the sum of the *squared* coefficient weights, penalyzing high coefficient

weights even more. The process and prediction is also very similar to Linear Regression and Lasso.

**How Multi-Layer Perceptron Regressor works under the hood?** This regressor uses the concept of neural network. A perceptron is a neuron, and the multi-layer is a the connected mesh of neurons, the neural network. They work better with features with values between 0 and 1, because a perceptron receives a value and pass it to a simple activation function: `identity` (a function that returns the same value that it received), `logistic` (sigmoid function), `tanh` (hyperbolic tan function), `relu` (return 0 if negative, of `identity` otherwise). Each perceptron will redirect to another one in the network, depending on its response value. The redirection decision may be changed by changing the nodes weights, resulting in a different path, and different result. After passing all datapoints, it will review the weights and retry until find the weights that results in the least squared loss possible. The process of teaching how to improve the weights by recalculating them can be tweaked with the `solver` parameter, together with the `learning_rate` and so on. With the trained network, it is just a matter of passing a new datapoint into the network and take the result as the predicted value.

### 1.4.4 Benchmark

After handling all missing values, let's set a naive baseline.

- Naive Prediction (Simple Math)

A simple math Naive Predictor was built to have a minimum base benchmark to beat with a more sophisticated method. This naive prediction can be calculated by the Lot Area of the predicting house multiplied by the dataset's mean Sale Price per Lot Area square feet, which can be calculated dividing the dataset's mean Sale Price, by the dataset's mean Lot Area.

The Naive Predictor should be naïve, yet reasonable, and as previously described, it can be achieved with simple math, as exemplified in the following code:

```
naive_train_X = aligned_encoded_train_data.drop(columns=['Id', 'SalePrice'])
naive_train_y = aligned_encoded_train_data['SalePrice']

mean_sale_price_per_lot_area = naive_train_y.mean() / naive_train_X['LotArea'].mean()

def naive_fit(X, y):
    mean_sale_price_per_lot_area = y.mean() / X['LotArea'].mean()
    return mean_sale_price_per_lot_area

def naive_predict(X, mean_sale_price_per_lot_area):
    return X['LotArea'] * mean_sale_price_per_lot_area
```

This predictor was not only trained and tested with KFold but was also submitted to the Kaggle's competition to understand how it performs by itself:

| Train score | Test score | Kaggle score |
|---|---|---|
| 0.5034265623764532 | 0.522732597478724 | |
| 0.5299006769793394 | 0.5109719880215422 | 0.55909 |

- Naive Prediction (Linear Regression)

In order to also use an algorithm in the naive prediction phase, a simple Linear Regression was made with all the features. It trained nicely, but was very overfitted: the test and Kaggle scores were very high:

| Train score | Test score | Kaggle score |
|---|---|---|
| 0.12314555019521473 | 1.8408330078556492 | |
| 0.13708035922180273 | 1.5926004067904753 | 1.38515 |

It probably overfitted due to the curse of dimensionality: there were too many features! So, it was reasonable to give a chance of making a Linear Regression with one feature, `LotArea` - the same used in the "simple math" Naive Prediction. Let's see how it performed:

| Train score | Test score | Kaggle score |
|---|---|---|
| 0.3633472894098743 | 0.3927671323761031 | |
| 0.40727444683178804 | 0.3795332401680835 | 0.42035 |

Nice, it didn't overfitted as the first try, and it scored better than the simple math predictor. Let's use `~0.4` as the Naive Prediction score.

- Kaggle Competition

Analysing the Kaggle's House Prices Leaderboard data, after removing some last mile submission outliers, I intended to have as the main benchmark to achieve a submitted score under `0.138`, and then under the first quartile of `0.120`.

## 1.5 Methodology

### 1.5.1 Data Preprocessing

**Handling missing values 1,2,3,4,5,6**

- Dropping columns

When there are too many missing data in a feature, it is reasonable to drop its corresponding column. It was done for `PoolQC`, `MiscFeature`, `Alley`, and `Fence` features.

- New type imputation

When there are some missing categorical data and they mean none or empty or other or no value, it is fine to create a new category to represent them, specially when they have their own importance when evaluating their correspondence with the target value. It was done for `FireplaceQu`, `GarageType`, `GarageFinish`, `GarageQual`, `GarageCond`, `BsmtExposure`, `BsmtFinType2`, `BsmtFinType1`, `BsmtCond`, `BsmtQual`, `MasVnrType`, `Exterior1st`, and `Exterior2nd`.

The following helper facilitates making that kind of imputation:

```
def fillna_with(feature, new_type = 'N/A'):
    train_data[feature].fillna(new_type, inplace=True)
    test_data[feature].fillna(new_type, inplace=True)
```

- Fill with zero

When there are some missing numerical data and they mean none or empty or no value, and zero could represent it nicely, fill them with zero, specially when they they are missing just in the test data and you don't want to look into them. It was done for `MasVnrArea`, `BsmtFullBath`, `BsmtHalfBath`, `BsmtFinSF1`, `BsmtFinSF2`, `BsmtUnfSF`, `TotalBsmtSF`, `GarageCars`, and `GarageArea`.

- Mode imputation

When there are some missing data and they can't mean none or zero, specially when they are missing just in the test data and you don't want to look into them, it may be reasonable to fill them with their most frequent value, since it's the most probable feature value. It was done for `Electrical`, `MSZoning`, `Utilities`, `Functional`, `KitchenQual`, and `SaleType`.

The following helper facilitates making that kind of imputation when there is missing values in the testing data:

```
def fillna_test_with(feature, value):
    test_data.loc[test_data[feature].isnull(), feature] = value

def fillna_test_with_train_mode(feature):
    feature_train_mode = train_data[feature].describe().top
    fillna_test_with(feature, feature_train_mode)
```

- Imputation by deriving from other features

When a feature can be derived from other one, it is makes sense to use some math to establish or simulate a resonable value that would still get the essence of that datapoint. It was done with `LotFrontage`, and `GarageYrBlt`.

For example, the following shows how `LotFrontage` was derived from `LotArea`, even when it is missing in the test data:

```
# LotFrontage can be derived from LotArea, impute based on a calculated mean Lot depth
lot_depth = train_data['LotArea'] / train_data['LotFrontage']
lot_depth_mean = lot_depth.mean()

# Populates a possible probable missing LotFrontage, by dividing its LotArea by the (generated)
train_lot_frontage_nan_indexes = train_data['LotFrontage'].isnull()
train_lot_frontage_missing_imputation = train_data[train_lot_frontage_nan_indexes]['LotArea'] /
train_data.loc[train_lot_frontage_nan_indexes, 'LotFrontage'] = train_lot_frontage_missing_imput

test_lot_frontage_nan_indexes = test_data['LotFrontage'].isnull()
test_lot_frontage_missing_imputation = test_data[test_lot_frontage_nan_indexes]['LotArea'] / lot
test_data.loc[test_lot_frontage_nan_indexes, 'LotFrontage'] = test_lot_frontage_missing_imputati
```

**Making type conversion 1,2,3,4,5,6,7,8**

- Type casting

When the data is not represented in its best type, converting it is the way to go. In our case, `MSSubClass` and `MoSold` were converted from `int64` to `object` (python dataframe's `string`). Also, `LotFrontage`, `LotDepth` (feature created), `MasVnrArea`, and `GarageYrBlt` were converted from `float64` to `int64` for convenience.

- One hot encoding (dummy values)

When a categorical feature is nominal, we create new binary columns, each column represents one category and their dummy values will tell if that datapoint is true (1) or false (0) for that column. This process is called "one hot encoding", and it adds lots of features, increasing the dataset's dimensionality. This process was done to `MSSubClass`, `MSZoning`, `LandContour`, `Street`, `LotConfig`, `BldgType`, `HouseStyle`, `RoofStyle`, `RoofMatl`, `MoSold`, `MasVnrType`, `Foundation`, `Heating`, `Electrical`, `PavedDrive`, `Condition1`, `Condition2`, `Exterior1st`, and `Exterior2nd`.

One intesting issue is that, when the set of a nominal feature differs in train and test data, it may end up with different shapes for the training data and the testing data. To make them have the same shape, there's an `align` method from panda's `DataFrame` which will create missing columns in both sides, with a given value, in this case, we want to fill with `0`, since the dummy value would indeed be that:

```
# ... one hot encoding

print('After one hot encoding shapes:')
display(encoded_train_data_with_dummies.shape)
display(encoded_test_data_with_dummies.shape)

# https://stackoverflow.com/a/47240395/1064325
aligned_encoded_train_data, aligned_encoded_test_data = encoded_train_data_with_dummies.align(
    encoded_test_data_with_dummies, join='outer', axis=1, fill_value=0
)

print('After one hot encoding alignment shapes:')
display(aligned_encoded_train_data.shape)
display(aligned_encoded_test_data.shape)
```

Output:

> Before one hot encoding shapes: (1460, 76) (1459, 75)
>
> After one hot encoding shapes: (1460, 199) (1459, 187)
>
> After one hot encoding alignment shapes: (1460, 201) (1459, 201)

Also note that the `SalePrice` have also been (unwantedly) aligned, so we may drop it.

- Mapping values

When a categorical feature is ordinal, it makes sense to map their categories to ordered integers, so the algorithms will work better. The following are the features that were mapped to ordered numbers: `LotShape`, `LandSlope`, `Utilities`, `ExterQual`, `ExterCond`, `BsmtQual`, `BsmtCond`, `BsmtExposure`, `HeatingQC`, `CentralAir`, `KitchenQual`, `Functional`, `FireplaceQu`, `GarageFinish`, `GarageQual`, `GarageCond`, `BsmtFinType1`, and `BsmtFinType2`.

- Target mean encoding

When the data is nominal, but it looks like that it could actually be ordered, but not logically as a grade value, demanding some extra effort, one good option is to use the target mean encoding. The `Neighborhood` and the `GarageType`, for example, may be ordered if we consider their average target value, and that's what the target mean encoding technique is for.

```
encoded_train_data = train_data.copy()
encoded_test_data = test_data.copy()

def mean_encode(data, t_data, feature, encoded_feature_name):
    categories = data[feature].unique()

    # target mean encode
    for category in categories:
        category_indexes = data[feature] == category
        t_category_indexes = t_data[feature] == category
        category_target_mean = data[category_indexes]['SalePrice'].mean()
        data.loc[category_indexes, feature] = category_target_mean
        t_data.loc[t_category_indexes, feature] = category_target_mean

    # rename
    data[encoded_feature_name] = data[feature].astype('float64').round(0).astype('int64')
    t_data[encoded_feature_name] = t_data[feature].astype('float64').round(0).astype('int64')

    # drop old column name
    data.drop(columns=[feature], inplace=True)
    t_data.drop(columns=[feature], inplace=True)

# target-mean-encode Neighborhood and GarageType into NeighborhoodMeanPrice and GarageTypeMeanPr
mean_encode(encoded_train_data, encoded_test_data, 'Neighborhood', 'NeighborhoodMeanPrice')
mean_encode(encoded_train_data, encoded_test_data, 'GarageType', 'GarageTypeMeanPrice')
```

**Handling outliers**

- Dropping rows

When we find outliers, it may impact our model negativelly. Hence, those datapoints can reasonably be dropped if the test and validation results shows an improvement without them, which in general does. In this case, after removing outliers from the training data, we've gone from 1460 datapoints to 1028, after analysing datapoints that are outliers in more than one feature for the non-binary features.

**Aggregating features** Examples:

- Sum

```
good_data['FloorSF'] = good_data['1stFlrSF'] + good_data['2ndFlrSF']
test_data['FloorSF'] = test_data['1stFlrSF'] + test_data['2ndFlrSF']
```

- Weighted sum

```
good_data['BsmtBathrooms'] = good_data['BsmtFullBath'] + (good_data['BsmtHalfBath'] * 0.5)
test_data['BsmtBathrooms'] = test_data['BsmtFullBath'] + (test_data['BsmtHalfBath'] * 0.5)
```

- Subtraction

```
good_data['NetBuildingMeanPrice'] = good_data['NeighborhoodMeanPrice']
                                    - good_data['GarageTypeMeanPrice']
                                    - good_data['MiscVal']
test_data['NetBuildingMeanPrice'] = test_data['NeighborhoodMeanPrice']
                                    - test_data['GarageTypeMeanPrice']
                                    - test_data['MiscVal']
```

- Age calculation

```
good_data['LotAge'] = 2019 - good_data['YearBuilt']
test_data['LotAge'] = 2019 - test_data['YearBuilt']
```

- Mean

```
good_data['OverallRating'] = (good_data['OverallQual'] + good_data['OverallCond']) / 2
test_data['OverallRating'] = (test_data['OverallQual'] + test_data['OverallCond']) / 2
```

- Division

```
train_data['LotDepth'] = train_data['LotArea'] / train_data['LotFrontage']
test_data['LotDepth'] = test_data['LotArea'] / test_data['LotFrontage']
```

- Weighted boosting

```
good_data['WeightedNeighborhoodMeanPrice'] = good_data['NeighborhoodMeanPrice']
                                             * (1. + (.1 / good_data['LotAge']))
test_data['WeightedNeighborhoodMeanPrice'] = test_data['NeighborhoodMeanPrice']
                                             * (1. + (.1 / test_data['LotAge']))
```

**Scaling features 1,2,3,4**

- Mean normalization

Initially I had applied the mean normalization for currency and area related features, but I discovered that it performed much better when not normalizing them. That finding was actually what have put me with `0.19446` in Kaggle, and beat the Naive Predictor.

- Min-max scaling

As a good practice, I also have applied scaling to some features related to time, grade, and quantity, using the min-max scaling technique. Some created features that felt into time, grade or quantity, also had their values applied with min-max scaling.

- Log scaling

I read about log scaling, and tried it, since the mean normalization failed in giving good results, but it also have failed, so I just dropped this attempt, but it's still worth mentioning and documenting it.

**Selecting features 1** In order to avoid the curse of dimensionality, feature selection is a very important step to have a good model. Let's see different approaches that were used in this Capstone Project:

- Problem relevance

Some features may be irrelevant, such as datapoint IDs or non-related features. In this case, I decided to drop `SaleType`, and `SaleCondition`, since I was interested in discovering the house prices based in the house data, and not in the sale data.

- Feature variance

Features that have very low variance shouldn't help much with supervised learning, because they have little or nothing to tell us. Since we have a lot of features, let's drop the features which variance are lower than `0.2`:

```
def run_var_analysis():
    features = reduced_var_train_data.drop(columns=['Id', 'SalePrice'])

    var_feats = features.var()
    var_feats_ordered = var_feats.sort_values(ascending=True)
    return var_feats_ordered[var_feats_ordered < 0.2]

low_variance_feats = run_var_analysis()

reduced_var_train_data.drop(columns=low_variance_feats.index, inplace=True)
reduced_var_test_data.drop(columns=low_variance_feats.index, inplace=True)
```

- Feature correlation

Features that are very correlated to some other feature also won't help much, since the other feature already gives us the same essence of that feature, which means that when we find a highly correlated (above `0.8`) tuple of features, we can drop one of that features without losing much.

```
# find highly correlated features
def run_corr_analysis():
    features = reduced_train_data.drop(columns=['Id', 'SalePrice'])

    corr = features.corr().abs()
    corr[corr == 1] = 0
    corr_cols = corr.max().sort_values(ascending=False)
    return corr_cols[corr_cols > 0.8]

corr_results = run_corr_analysis()

corr_result_index = 0
for corr_result in corr_results.index:
    # drop just one feature of each "tuple"
    if (corr_result_index % 2 == 1):
```

```
        reduced_train_data.drop(columns=[corr_result], inplace=True)
        reduced_test_data.drop(columns=[corr_result], inplace=True)

    corr_result_index = corr_result_index + 1
```

- Target correlation

Features that have little correlation (under `0.2`) with the target may also be dropped because they wouldn't help the supervised learning model to predict the target value.

```
def run_target_corr_analysis():
    features = reduced_train_data.drop(columns=['Id'])

    corr = features.corr().abs()
    target_corr = corr['SalePrice'].sort_values(ascending=True)
    return target_corr[target_corr < 0.2]

lowly_correlated_feats = run_target_corr_analysis()

reduced_train_data.drop(columns=lowly_correlated_feats.index, inplace=True)
reduced_test_data.drop(columns=lowly_correlated_feats.index, inplace=True)
```

- Feature importance

Finally, from the remaining features, we may drop features with low importance. Some regressors outputs a feature importance array telling which features were more significant to make the regression prediction itself. In this case I've used `RandomForestRegressor` to calculate the feature importance, and dropped features which importance were lower than `0.02`.

```
from sklearn.ensemble import RandomForestRegressor

rfr = RandomForestRegressor(n_jobs=-1, random_state=42, n_estimators=50)

feature_selected_train_data = reduced_train_data.copy()
feature_selected_test_data = reduced_test_data.copy()

def run_importance_analysis():
    features = feature_selected_train_data.drop(columns=['Id', 'SalePrice'])

    rfr_importance_analyser = rfr.fit(features, feature_selected_train_data['SalePrice'])
    return features.columns[rfr_importance_analyser.feature_importances_ < 0.02]

low_importance_feats = run_importance_analysis()

feature_selected_train_data.drop(columns=low_importance_feats, inplace=True)
feature_selected_test_data.drop(columns=low_importance_feats, inplace=True)
```

**Feature transformation**

- Dimensionality reduction

Before clustering, I intended to perform a PCA transformation, but even with different number of components, I couldn't get a better result than when I wasn't using PCA, so I ended up dropping it from the Project itself.

### 1.5.2 Implementation

- Metrics

During this Capstone Project's development, there were *three* main important things to note regarding metrics.

The first is that I had to handle negative input values (replacing with 0) before passing to `mean_squared_log_error` in a custom scorer, otherwise it would throw an error. The negative results was being probably intensified because of the second issue: I forgot to set `greater_is_better` to `False` when creating the scorer with `make_scorer` for the grid search scoring.

The last one is that at first I was using `mean_squared_log_error` alone, but then, at some late point I realized that I missed to also apply the squared root of the `mean_squared_log_error`. Hopefully, the trained model results didn't change after that fix, but it became much better to properly compare results with Kaggle submission.

Summarizing the issues:

1) When using `mean_squared_log_error` always remember of handling negative values beforehand.
2) When using `make_scorer` make sure you've set the proper `greater_is_better` parameter.
3) Use `np.sqrt` in addition to `mean_squared_log_error` if you want to calculate the "root mean squared log error" (RMSLE)

After struggling with these issues, I had everything fixed. The metric creation ended up with the following code:

```
def custom_scorer(y, y_pred):
    y_pred[y_pred < 0] = 0
    return np.sqrt(mean_squared_log_error(y, y_pred))

scorer = make_scorer(custom_scorer, greater_is_better=False)
```

- Clustering Algorithm

I was advised in the Capstone Proposal review that doing Clustering before Regression wouldn't be significant for this dataset, so I almost dropped all of this. In the end I was trying so hard lots of things, that I ended up trying KMeans. It really haven't made a significant improvement, but it improved my final result, so I kept it. Since it wasn't the main objective, I didn't went through different clustering techniques.

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
```

```
clusters_range = range(2, 11)
scores = []

for n_clusters in clusters_range:
    kmeans = KMeans(n_clusters=n_clusters, n_jobs=-1, random_state=42)
    result = kmeans.fit_predict(feature_selected_train_data.drop(columns=['Id', 'SalePrice']))

    curr_score = silhouette_score(feature_selected_train_data, result)
    scores.append(curr_score)

cluster_scores_df = pd.DataFrame({ 'scores': scores, 'n_clusters': clusters_range })

best_score = cluster_scores_df['scores'].max()
n_clusters = cluster_scores_df[cluster_scores_df['scores'] == best_score]['n_clusters'].values[0

kmeans = KMeans(n_clusters=n_clusters, n_jobs=-1, random_state=42)
result = kmeans.fit_predict(feature_selected_train_data.drop(columns=['Id', 'SalePrice']))
test_result = kmeans.predict(feature_selected_test_data.drop(columns=['Id']))

feature_selected_train_data['Cluster'] = result
feature_selected_test_data['Cluster'] = test_result
```

Actually, it was just in the clustering attempt (attempt #28) that I could beat myself in validation with a score of `0.18706` at Kaggle (I had firstly beaten the Naive Predictor in my attempt #6 by not normalizing the currency features, when reached `0.19446` in the competition). Despite it's not much, it helped.

One interesting finding that have helped to perform (very slightly) better was to use one extra cluster than the best scored one (from `0.18706` to `0.18616`, and later from `0.18163` to `0.18140`).

I also tried to "recluster" one cluster that was being badly scored, but it ended up having an even worse score, so I dropped that attempt as well. Lesson learned.

- Regression

I ended up not trying XGBoost Regressor, neither LightGBM Regressor because I needed to upgrade my python environment to install them, and at that point I didn't have enough time for that. In the other hand, I did tried lots of other regressors available in `sklearn`, such as: `LinearRegression`, `RandomForestRegressor`, `GradientBoostingRegressor`, `Lasso`, `Ridge`, and `MLPRegressor` (Multi-layer Perceptron Regressor), and they did pretty well.

I did have used `KFold` and `GridSearchCV`, but didn't used `RandomizedSearchCV` as I planned at first, but I think it could still do the job.

**Tip: prefer reusable functions instead of copy-pasting** Initially I was copy-pasting the model tuning code, and switching the algorithm and the cross validation hyper parameters. But when I had to tweak something, I had to change it in all copy-pasted code, and also, it simplifies a lot the reading, the searching, and the maintainability in the notebook. To overcome this issue, I've created two helper functions (`kfold` and `tune_model`) which made it much easier to implement the tuning for all those 6 regressors (and also some others that I tried but dismissed due to poor performance).

```
from sklearn.metrics import mean_squared_log_error, make_scorer
from sklearn.model_selection import KFold, GridSearchCV

def kfold(model, name, cluster, train_X, train_y):
    kf = KFold(n_splits=2, random_state=42, shuffle=False)

    train_scores = []
    test_scores = []

    for kf_chunks in kf.split(train_X):
        train_index = kf_chunks[0]
        test_index = kf_chunks[1]

        X_train = train_X.iloc[train_index]
        X_test = train_X.iloc[test_index]
        y_train = train_y.iloc[train_index]
        y_test = train_y.iloc[test_index]

        model_predictor = model.fit(X_train, y_train)
        y_train_predicted = model_predictor.predict(X_train)
        y_test_predicted = model_predictor.predict(X_test)

        y_train_predicted[y_train_predicted < 0] = 0
        y_test_predicted[y_test_predicted < 0] = 0

        train_score = np.sqrt(mean_squared_log_error(y_train, y_train_predicted))
        test_score = np.sqrt(mean_squared_log_error(y_test, y_test_predicted))

        train_scores.append(train_score)
        test_scores.append(test_score)

        print('[{}][{}] Current fold train performance'.format(name, cluster), train_score)
        print('[{}][{}] Current fold test performance'.format(name, cluster), test_score)

    return train_scores, test_scores

def tune_model(Model, name, default_params, cv_params):
    best_params = []
    merged_params_arr = []
    model_train_scores = []
    model_test_scores = []
    mean_train_scores = []
    mean_test_scores = []

    for cluster in range(n_clusters):
        current_cluster_filter = feature_selected_train_data['Cluster'] == cluster

        gs_train_X = feature_selected_train_data[current_cluster_filter].drop(columns=['Id', 'Sa
```

```
        gs_train_y = feature_selected_train_data[current_cluster_filter]['SalePrice']

        gs = GridSearchCV(Model(**default_params), cv_params, n_jobs=-1, cv=2, scoring=gs_scorer

        gs_predictor = gs.fit(gs_train_X, gs_train_y)

        best_params.append(gs_predictor.best_params_)

        merged_params = dict()
        merged_params.update(default_params)
        merged_params.update(gs_predictor.best_params_)
        merged_params_arr.append(merged_params)

        train_scores, test_scores = kfold(Model(**merged_params), name, cluster, gs_train_X, gs_

        model_train_scores.append(train_scores)
        model_test_scores.append(test_scores)

        mean_train_scores.append(np.mean(train_scores))
        mean_test_scores.append(np.mean(test_scores))

    model_arr = np.full((1, n_clusters), name, object)[0]
    return pd.DataFrame({
        'cluster': range(n_clusters),
        'params_cv': best_params,
        'params_full': merged_params_arr,
        'model': model_arr,
        'train_scores': model_train_scores,
        'test_scores': model_test_scores,
        'train_score': mean_train_scores,
        'test_score': mean_test_scores,
    })
```

After implementing all those regressors tuning, I've selected the best regressor for each cluster, and rebuilt them in order to submit the final mixed value. It was my Kaggle submission attempt #41, my last resort, and it indeed improved my mark from `0.18228` to `0.18163` by using multiple mixed custom models for each cluster (Lasso and Ridge have got the best results), instead of just Linear Regressions for each cluster.

```
# join the performance of all dataframes so we can select the best one easier
all_df = linear_regression_df.copy()
# all_df = all_df.append(random_forest_df.copy()) # overfit
# all_df = all_df.append(gradient_boost_df.copy()) # overfit
all_df = all_df.append(lasso_df.copy())
all_df = all_df.append(ridge_df.copy())
all_df = all_df.append(mlpr_df.copy())

# initialize the selected models for each cluster
```

```
selected_models = pd.DataFrame({
    'cluster': [],
    'model': [],
    'params_full': [],
    'test_score': []
})

# for each cluster find the model that performed better
for cluster in range(n_clusters):
    cluster_best_models = all_df[all_df['cluster'] == cluster]
                            .sort_values(by='test_score', ascending=True)
    cluster_best_model = cluster_best_models.head(1)[
                            ['cluster', 'model', 'params_full', 'test_score']
                        ]
    selected_models = selected_models.append(cluster_best_model)

selected_models['cluster'] = selected_models['cluster'].astype(int)
selected_models
```

**Avoiding overfitted models** One important tip is to avoid overfitted models (as we can see in the code below, the Random Forest Regressor and the Gradient Boosting Regressor were commented out since they had overfitted). It can be easily noticed when comparing the training prediction score with the testing prediction score of all the kfolds. The scores should be close to each other, and sometimes the training score may even be slightly worse than the testing score. When the training score is way too better than the testing score, take a look into this and avoid that, or else the validation score tend be even worse than the testing score. I thought of adding a `score_ratio` column to use as some kind of "regularization", but I ended up just commenting out the ones that were consistently having overfitted results with the automated GridSearchCV hyperparameters.

### 1.5.3 Refinement

In addition to the aforementioned refinements, let me summarize the refinement journey:

**What have helped:**

- Kaggle Attempt #06: scored 0.19446

Removed currency features normalization (used Random Forest Regressor with `max_depth=10, n_estimators=100, min_samples_leaf=3, min_samples_split=2`).

- Kaggle Attempt #28: scored 0.18706

Clustered Linear Regression (used Linear Regressions with `fit_intercept=True, normalize=True`).

- Kaggle Attempt #30: scored 0.18616

Clustered (with one extra cluster) Linear Regression (used Linear Regressions with `fit_intercept=True, normalize=True`).

- Kaggle Attempt #39: scored 0.18228

Created more Features (used Linear Regressions with `fit_intercept=True`, `normalize=True`).

- Kaggle Attempt #41: scored 0.18163

Mixed algorithms, no extra cluster (Lasso and Ridge with the params defined in their dynamic dataframes)

- Kaggle Attempt #42: scored 0.18140

Mixed algorithms, with one extra cluster (Lasso and Ridge with the params defined in their dynamic dataframes)

**What haven't helped:**

- Relaxing feature selection to have more features (curse of dimensionality)
- Making feature normalization or log scaling to currency features
- Some algorithms performed really bad: `SGDRegressor`, for example

## 1.6 Results

### 1.6.1 Model Evaluation and Validation

To get the dynamic selected model for each cluster, we need to rebuild the model with the stored best algorithm and hyperparameters. With the final models trained, let's validate them with some unseen data like: a perturbated trained X, the recovered outliers datapoints, and the Kaggle validation data (by submitting the final result):

```
# since the selected models are dynamic,
# we need to dynamically build the model in this phase
# the params will come from `params_full` column of the `selected_models` df
def build_model(model, params):
    if (model == 'Linear Regression'):
        return LinearRegression(**params)
    if (model == 'Lasso'):
        return Lasso(**params)
    if (model == 'Ridge'):
        return Ridge(**params)
    if (model == 'MLPR'):
        return MLPRegressor(**params)
    # other models may be added here if necessary

results = []
ids = []

sensitivity_results_df = pd.DataFrame({
    'cluster': [],
```

```
    'model': [],
    'old_test_score': [],
    'perturbated_score': [],
    'outliers_score': [],
})

def perturbate_features(df, perturbation_ratio):
    perturbated_df = df.copy()
    for feature in df.columns:
        perturbated_df[feature] = perturbated_df[feature] * (1 + perturbation_ratio)
    return perturbated_df

for cluster in range(n_clusters):
    current_cluster_filter = feature_selected_train_data['Cluster'] == cluster

    # create X and y for training
    train_X = feature_selected_train_data[current_cluster_filter].drop(columns=['Id', 'SalePrice
    train_y = feature_selected_train_data[current_cluster_filter]['SalePrice']

    # create X and y for outliers validation
    outliers_cluster_filter = outliers_sa_data['Cluster'] == cluster
    outliers_sa_data_X = outliers_sa_data[outliers_cluster_filter].drop(columns=['Id', 'SalePric
    outliers_sa_data_y = outliers_sa_data[outliers_cluster_filter]['SalePrice']

    # create X (and reuse train_y) for a sensitivity analysis validation
    perturbated_X = perturbate_features(train_X, 0.05)

    # create X and y for Kaggle validation
    current_cluster_filter = feature_selected_test_data['Cluster'] == cluster
    predicting_X = feature_selected_test_data[current_cluster_filter].drop(columns=['Id', 'Clust
    predicting_ids = feature_selected_test_data[current_cluster_filter]['Id']

    # create the model dynamically for this cluster
    selected_model_df = selected_models[selected_models['cluster'] == cluster]
    model = build_model(selected_model_df['model'].values[0], selected_model_df['params_full'].v

    # fit and predict
    model = model.fit(train_X, train_y)
    prediction = model.predict(predicting_X)
    perturbated_prediction = model.predict(perturbated_X)
    outliers_prediction = model.predict(outliers_sa_data_X)

    # save Kaggle results for further submission
    results = np.concatenate((results, prediction))
    ids = np.concatenate((ids, predicting_ids))

    # score results
    cluster_length = train_X.shape[0]
```

```
    sensitivity_results_df = sensitivity_results_df.append(pd.DataFrame({
        'cluster': [cluster],
        'old_test_score': [selected_model_df['test_score'].values[0]],
        'perturbated_score': [custom_scorer(train_y, perturbated_prediction)],
        'outliers_score': [custom_scorer(outliers_sa_data_y, outliers_prediction)],
        'model': [selected_model_df['model'].values[0]],
    }))

sensitivity_results_df
```

| cluster | model | test_score | outliers_score | perturbated_score |
|---|---|---|---|---|
| 0 | Lasso | 0.107076 | 0.230992 | 0.120247 |
| 1 | Lasso | 0.169924 | 0.232484 | 0.179695 |
| 2 | Ridge | 0.085253 | 0.158811 | 0.090441 |
| 3 | Lasso | 0.114486 | 0.190018 | 0.151876 |

**Robustness results and analysis** We can notice that with unseen data, it performs worser than its initial test score. The Kaggle validation score was `0.18140` while the best cluster test score was `0.085253`, and with a perturbated X, despite it performed better than it has performed in the Kaggle's score, it demonstrated that some clusters had a larger impact (0 and 3) than other ones (1 and 2). One important thing to note is that the outliers score have scored even worser than the Kaggle score, what probably means that the Kaggle validation data doesn't rely just on outliers, but it may mean that the outliers in the Kaggle data is impacting its score negatively. I've tried to drop the outliers removal, but it didn't performed nicely.

Despite that they have underperformance in this robustness analysis, they actually still performed better than the Naive Predictor that scored `0.42035` in Kaggle. That's good! But I would still say we can't fully rely in this model, specially when trying to predict some house price which datapoints end up being outliers.

### 1.6.2 Justification

Hopefully, as described in my journey, I could build indeed a better model than the Naive Predictor baseline score, and after that I could still beat myself 5 times more:

Kaggle Naive Predictor: scored 0.42035
Kaggle Attempt #06: scored 0.19446
Kaggle Attempt #28: scored 0.18706
Kaggle Attempt #30: scored 0.18616
Kaggle Attempt #39: scored 0.18228
Kaggle Attempt #41: scored 0.18163
Kaggle Attempt #42: scored 0.18140

I actually couldn't get even close to the Kaggle's mark (`~0.13`) I intended to reach at first, the competition was more challenging than I thought, which made me strieve to find the best I could, learning and trying lots of things! Hopefully, the Udacity reviewer that revised my Capstone Proposal commented that the benchmarking was more about comparing final results with the Naive Predictor than comparing with other real world solutions.

Anyway, I think this is still a reasonable solution, I did my best, and it did pretty well solving the house prices prediction problem when compared with the Naive Predictor.

## 1.7 Conclusion

### 1.7.1 Free-Form Visualization

I'd like to share the feature importance of the selected features. That should give us a bright understanding on what mainly makes the house prices to be what they are. That's what a business executive in real estate area would like to confirm, despite they might already have some good notion, this is a data driven output.

| Feature | Importance score |
|---|---|
| HouseSF | 0.48442083 |
| WeightedNeighborhoodMeanPrice | 0.34660865 |
| FloorSF | 0.03043113 |
| GarageArea | 0.02919309 |
| BsmtFinSF1 | 0.02213524 |

As we can see, there are two main features that mainly dictates the house price: how many square feets the house have, and the neiborhood mean house price (slightly weighted/boosted by their different lot ages). The other features are very less important than the first two (even less than 1/10), but they helped in fine tuning: `FlootSF`, `GarageArea`, and `BsmtFinSF1`.

### 1.7.2 Reflection

In this Capstone Project implementation, a lot of reading, learning, trying, and discovering was necessary to do in order to properly act in different parts of the Machine Learning process, and I'd like to highlight the most interesting findings and challenges in the implementation process.

- Handling different shapes on training and testing data after doing one hot encoding

As a good practice, we shouldn't look into the validation data, but that led to having different shapes on training and testing data after doing one hot encoding, since some categories of categorical data were present just on training or just on testing data. How to do that without looking into the validation data?

To make them have the same shape, there's an `align` method from panda's `DataFrame` that we can use to create the missing columns in both sides, with a given filling value (`0` in this case of one hot encoding):

```
# ... one hot encoding

print('After one hot encoding shapes:')
display(encoded_train_data_with_dummies.shape)
display(encoded_test_data_with_dummies.shape)

# https://stackoverflow.com/a/47240395/1064325
aligned_encoded_train_data, aligned_encoded_test_data = encoded_train_data_with_dummies.align(
```

```
    encoded_test_data_with_dummies, join='outer', axis=1, fill_value=0
)

print('After one hot encoding alignment shapes:')
display(aligned_encoded_train_data.shape)
display(aligned_encoded_test_data.shape)
```

Output:

> Before one hot encoding shapes: (1460, 76) (1459, 75)
>
> After one hot encoding shapes: (1460, 199) (1459, 187)
>
> After one hot encoding alignment shapes: (1460, 201) (1459, 201)

- Target mean encoding 1

I actually thought of doing this encoding alone, but then started to search in the literature if that approach existed. It is a rather rare type of encoding, and in this case it was this technique that originated the `NeighborhoodMeanPrice`, which was later boosted and turned `WeightedNeighborhoodMeanPrice`: the second most important feature!

- Feature selection

The curse of dimensionality is real. Using different approaches to adjust to a reduced dimensionality was essencial to get a good result. The research was thorough, and I was happy that I could start contributing with the community at StackOverflow (even that it was with a smal post):

> **Question**: *List Highest Correlation Pairs from a Large Correlation Matrix in Pandas?*
>
> **Answer**:
>
> [...] (In my case) I just wanted to drop some highly correlated features as part of a feature selection phase.
>
> So I ended up with the following simplified solution:
>
> ```
> # map features to their absolute correlation values
> corr = features.corr().abs()
>
> # set equality (self correlation) as zero
> corr[corr == 1] = 0
>
> # of each feature, find the max correlation
> # and sort the resulting array in ascending order
> corr_cols = corr.max().sort_values(ascending=False)
>
> # display the highly correlated features
> display(corr_cols[corr_cols > 0.8])
> ```
>
> In this case, if you want to drop correlated features, you may map through the filtered `corr_cols` array and remove the odd-indexed (or even-indexed) ones.

- Reading about how algorithm works under their hood

It was important to recap some aspects that I've learned through the course, and understand better some algorithms that I didn't use before such as `Lasso` and `Ridge`, for example. Reviewing some topics like L1 and L2 Regularization and being able to understand them better gives us more confidence on what we're doing, consolidating our knowledge and experience. It's not just a matter of putting things and try them, experience and knowledge will lead us to the right path to get better results faster and with confidence. Learning is a continuing process, thank you Udacity, mentors and reviewers for incentivating us to keep in the right path.

### 1.7.3 Improvement

There are a list of things that I think that could improve the results (in order of relevance):

- Stacking and Blending 1

I have looked at some Kaggle kernels using Stacking and Blending that have performed much better than my attempts. I haven't tried them, I think it will require some more study and care to use them in the right way, knowing what I'm doing, so I prefered to keep with what I've described in the Capstone Proposal.

- Train an outlier house prices predictor

Based on the "Robustness results and analysis" section above, one idea with good potential that I have thought was to train a separate predictor for outlier datapoints. So, in order to predict a value, it first checks if it is an outlier or not, if so, it will use the trained outlier predictor, otherwise it would use the corresponding cluster model. Since I guess that it is the outliers in the validation data that are pushing the score negatively, I think this idea has a nice potential!

- XGBoostRegressor and LightGBMRegressor

Some reviewers and mentors have advised me to use those algorithms since they perform nicely, but I ended up not willing to update my entire python environment from 2.x to 3.x just to use them, so I decided to explore the most I could from the available regressors in the scikit learn version I used in this Capstone Project.

- VotingRegressor 1

`VotingRegressor` is another package that was unavailable in the sklearn version I'm using, but combining results looked interesting, and possibly useful.