

COMP4433 Competition

House Price Prediction

AllSidesChecker

LI Tong 21101988D
LIU Yuzhou 21100602D
LU Zhoudao 21099695D
YANG Yifan 20075243D

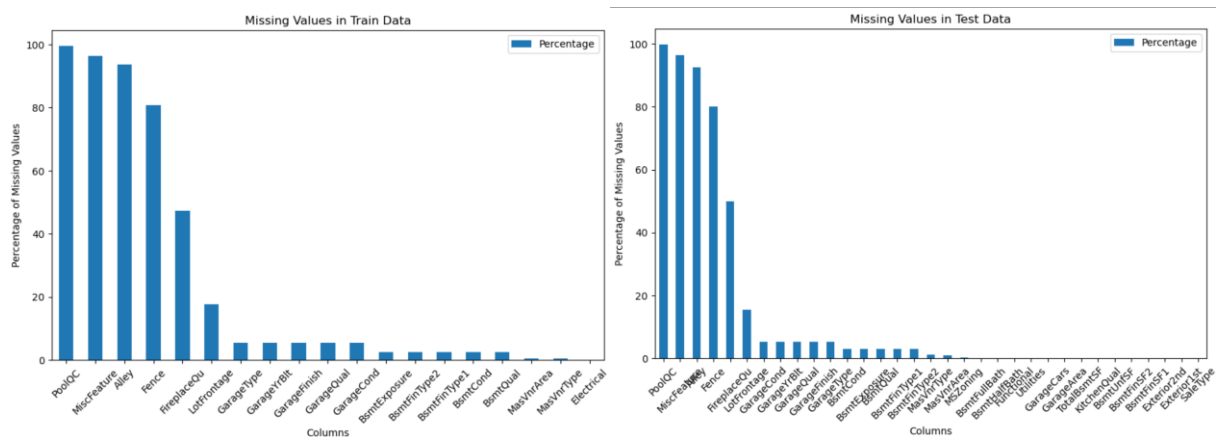
Final Score: 0.12728

Data analysis visualization

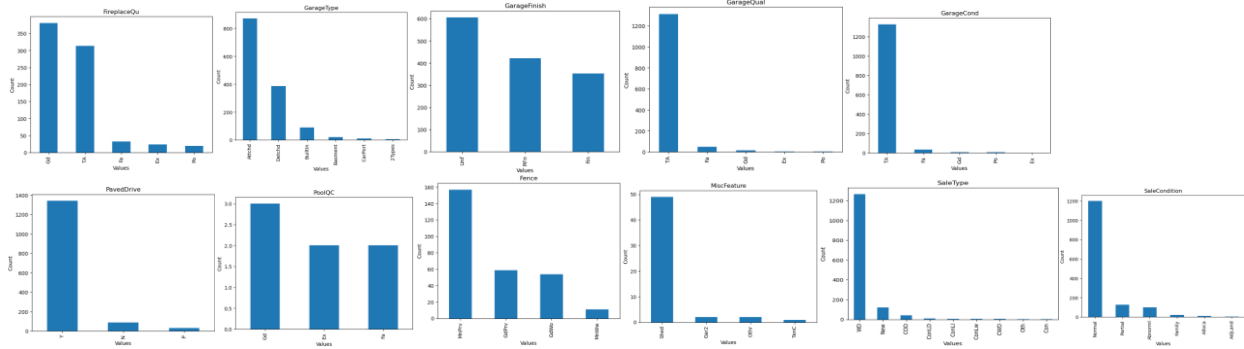
To analysis the data, we can first make the numerous visualizations which make the data more concise and clear.

Missing Value

Missing Values may lead to confusion to the analysis and processing in the following prediction. We first analysis the missing values in both training and testing data sets, by using the bar plots built in the library matplotlib.



[illegible]

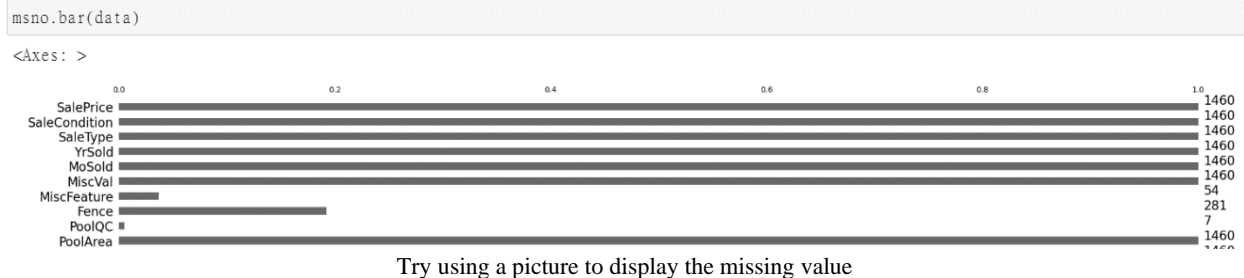


Null values Handling

For Null handling, we have two parts to deal with attributes which have null value.

Divide null attributes into two parts

To deal with these kinds of attributes, we mainly divide them into two parts using the percentage of null values in specific column.



At the beginning, we try to using pictures to display the missing value and null values. However, we gradually discovered that images can only show the general situation (or the trend of the data) and are not conducive to us setting the value of the defined percentage, because we use tables and specific percentage data to display so that we can conduct more precise analysis.

As shown, the list of data shows the specific number of missing values of every attribute and the percentage of it. Which makes it clear for us to observe and analyze.

```
In [7]: train_missing = missing_values_table(train)
train_missing
```

```
Out[7]:
```

	Missing Values	Percentage
PoolQC	1453	99.520548
MiscFeature	1406	96.301370
Alley	1369	93.767123
Fence	1179	80.753425
FireplaceQu	690	47.260274
LotFrontage	259	17.739726
GarageType	81	5.547945
GarageYrBlt	81	5.547945
GarageFinish	81	5.547945
GarageQual	81	5.547945
GarageCond	81	5.547945
BsmtExposure	38	2.602740
BsmtFinType2	38	2.602740
BsmtFinType1	37	2.534247
BsmtCond	37	2.534247
BsmtQual	37	2.534247
MasVnrArea	8	0.547945
MasVnrType	8	0.547945
Electrical	1	0.068493

According to our research, elements with excessive missing rates are not conducive to our predictions. At the same time, we also observed that in the information base of house price prediction, it seems to be very common to have a certain proportion of missing values (for example, the list shows that the percentage of missing values for some elements is about 5%).

Therefore, after referring to online methods and combining our own judgment, we decided:

1. Directly delete attributes with missing values greater than 15%.

```
i=0
m_list=[]
test_m_list = test_missing.index.tolist()
for col in train_missing["Percentage"]:
    if(col > 15):
        train = train.drop(train_missing["Percentage"].index[i], axis=1)
        print(train_missing["Percentage"].index[i])
        if train_missing["Percentage"].index[i] in test_missing.index:
            test = test.drop(test_missing["Percentage"].index[i], axis=1)
            test_m_list.remove(test_missing["Percentage"].index[i])
    else:
        m_list.append(train_missing["Percentage"].index[i])
    i+=1
```

Code to deal attribute which missing percentage > 15%

2. For attributes with missing values less than or equal to 15%, we will try to fill the missing values with appropriate values and then retain the attribute. Because these attributes may still have a certain value in prediction, such as BsmtFinType and so on.

Fit in missing values

In this section, we try three ways to fill missing values.

1. KNN (K-nearest neighbors)
2. Mice (Multiple Imputation by Chained Equations)
3. The combination of method 1 and 2.

```
def missing_value_all(NA_list, df):  
  
    #copy df which used in those two model  
    train_knn = df.copy(deep=True)  
    knn_imputer = KNNImputer(n_neighbors=num_of_neighbor, weights=weight_type)  
    train_mice = df.copy(deep=True)  
    mice_imputer = IterativeImputer()  
  
    for col_name in NA_list:  
        #KNN model  
        train_knn[col_name] = knn_imputer.fit_transform(train_knn[[col_name]])  
        #Mice  
        train_mice[col_name] = mice_imputer.fit_transform(train_mice[[col_name]])  
  
    modle_list = []  
    modle_list.append(train_knn)  
    modle_list.append(train_mice)  
    return modle_list
```

Code related to above 3 methods

The function above can invoke KNN, Mice or both to fit in the missing values which will be store in modle_list and return out (about method 3, it will take the average value of KNN and Mice as its result). Later, a masking function will used (not display here) to locate each missing values and match them with the value in modle_list and then replace them in the train and test data. By the way, whatever KNN or Mice, those functions can only handle numeric data, thus before these steps, we will first translate the objective values into numeric type.

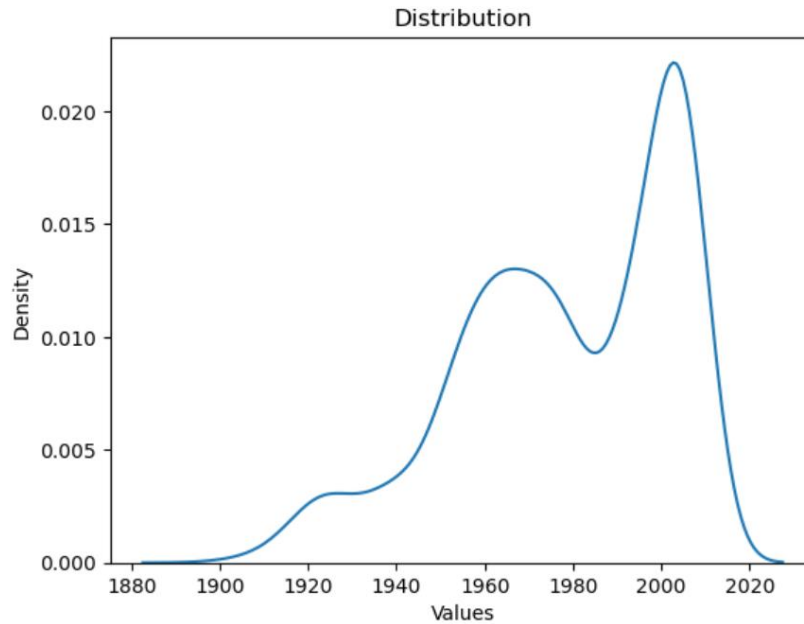
The thought process of missing value fit

As a step in data processing, this part was developed in early stage of the whole project. It is precisely because of this that as the project progressed, there were so many changes and the three filling methods that were finally retained.

In the initial development stage, we tried to directly use mean filling to handle missing values. For text information that is more difficult to process, we also try to ignore it directly.

However, after we understand the actual meaning of the data, we find both methods have considerable disadvantages:

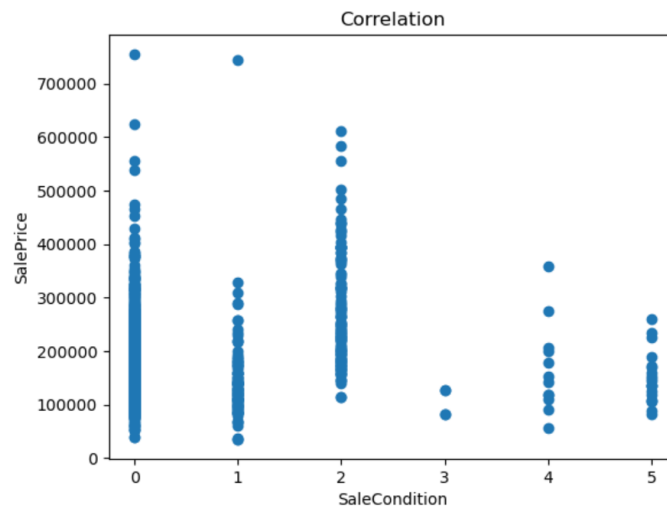
1. For using mean filling to handle values. Such filling may destroy the original characteristics of a column of data. For example, when deal with attribute “GarageYrBlt”, The main reason for our lack of data is that some houses do not have parking spaces, not because this data was not collected. Therefore, using average padding will destroy the overall trend of this data and cause it to lose its predictive value.



The mean for "GarageYrBlt" is around 1978

The picture above clearly shows the numerical distribution of the attribute "GarageYrBlt". It can be seen that when the overall value is tilted to the right, using the average value to fill missing values is obviously an undesirable method. This kind of filling The method will change the overall tilt and have a negative impact on house price predictions.

2. For directly ignore the text information. This method was disused after we developed the encoding function, Because the impact of text type attributes on result prediction is still immeasurable, these attributes such as "SaleCondition", Although this type of element generally describes the condition of a house, image data shows that it has a strong correlation with housing prices.



The screenshot above shows the correlation between “SalePrice” and “SaleCondition”, which is a text attribute just has a very simple encoding. The information in the figure clearly shows that this list of text attributes still has considerable predictive value. Therefore, for house price prediction, this type of information cannot be ignored.

Based on the above improvements, the prediction results of the existing three methods are as follows: (these result are under sigmoid)

KNN (method 1)	Mice (method 2)	Both (method 3)
0.13616	0.13977	0.13673

Based on the prediction results of the three methods, method one often has the most ideal results. This may be because the KNN algorithm is based on nearby data for reference and filling. In data used for housing price prediction, there is often a certain correlation between adjacent columns (such as “GarageArea” and “GarageCars”). This makes the KNN algorithm more effective.

Encoding (create a new way)

Categorical features cannot be analyzed by most models, we should turn them into numerical features. The most popular ways are labelled encoding, one-hot encoding, and frequency encoding. Whether the problem is regression or classification, the encoding ways are universal. However, for our project about house price prediction, some specific categorical features may lead to higher prices overall. For example, no matter if the pool quality is excellent or not, houses with pools are more expensive on average).

							SalePrice	
	count	mean	std	min	25%	50%	75%	max
PoolQC								
Ex	2.0	490000.000000	360624.458405	235000.0	362500.0	490000.0	617500.0	745000.0
Fa	2.0	215500.000000	48790.367902	181000.0	198250.0	215500.0	232750.0	250000.0
Gd	3.0	201990.000000	63441.392639	160000.0	165500.0	171000.0	222985.0	274970.0
NULL	1453.0	180404.663455	78168.872519	34900.0	129900.0	162900.0	213500.0	755000.0

So we want to assign different weights to different features based on the ‘SalePrice’. We create one encoding method as follows:

$$Num(cat = x) = \frac{xSalePrice.mean - allSalePrice.mean}{x.std}$$

Like the example, we can encode 'PoolQC=EX' to $(490000-180921.1959)/360624.4504$

For the None values in categorical columns, we don't want to consider them as missing values. For example, the house without any pools don't means the value is missed. We think we should consider this kind of NA as a specific feature. In that case, we didn't use the ways to handling missing values to handle them if the percentage of NA is not too much.

We compared the results of our method and labeled encoding. Our method is about 0.02 better:

0.1515					0.13673				
LotShape	LandContour	Utilities	LotConfig	LandSlope	LotShape	LandContour	Utilities	LotConfig	LandSlope
3	3	0	4	0	-0.232031	-0.009399	0.000375	-0.052117	-0.012259
3	3	0	2	0	-0.232031	-0.009399	0.000375	-0.047566	-0.012259
0	3	0	4	0	0.293279	-0.009399	0.000375	-0.052117	-0.012259
0	3	0	0	0	0.293279	-0.009399	0.000375	0.008314	-0.012259
0	3	0	2	0	0.293279	-0.009399	0.000375	-0.047566	-0.012259

Labelled encoding

Our encoding

We control for a single variable for comparison. I still use sigmoid which is mentioned in data transformation at this part. However, we eliminate this method based on the final score.

Feature selection

Delete columns with large percentage of same attribute

From the previous analysis, we can intuitively find that there has some features which have a large percentage of single attribute and just a few pieces of other attributes. For those features such as "Street" would have no help in the following prediction process and might provide the noise and affect the accuracy of results. Therefore, we choose to delete the those columns which have max percentage of same attribute over 85%.

```
for col in train.columns:
    value_counts = train[col].value_counts()

    percentage = value_counts / len(train) * 100

    max_percentage = percentage.max()

    if max_percentage > 85:
        train = train.drop(col, axis=1)
        test = test.drop(col,axis=1)
        print(col)
```


Combine the columns with higher correlation

Here we use the correlation matrix to set the condition. To see whether we need to combine different columns, since the feature selection may have more correlation with combined features.

Here we our thought is that, if two different columns have strong correlation ship. That means there is a high probability that there could be redundancy. So, reduce some columns which could make our data much clearer.

At the first stage, our thought is to drop some of those columns with high correlation ship, but finally we realized that if we just simply remove some of the columns, there might be loss of information in the graph, then we choose a way which compromised a little bit. Instead of dropping some of the redundant columns, we choose to combine them to get a completely new column, and drop the old columns. By doing so, we can reduce the number of columns, at the same time, also partially preserve the information in all of these columns.

```
In [36]: def correlation(dataset, threshold):
col_corr=dict() # set will contains unique values.
drop=set()
corr_matrix=dataset.corr() #finding the correlation between columns.
flag=True
for i in range(len(corr_matrix.columns)): #number of columns
    for j in range(i):
        if abs(corr_matrix.iloc[i,j])>threshold: #checking the correlation between columns.
            flag=False
            if(corr_matrix.columns[j] not in col_corr.keys()):
                col_corr[corr_matrix.columns[i]]=corr_matrix.columns[j]
                drop.add(corr_matrix.columns[j])
                drop.add(corr_matrix.columns[i])
    return col_corr, drop, flag #returning set of column names
```

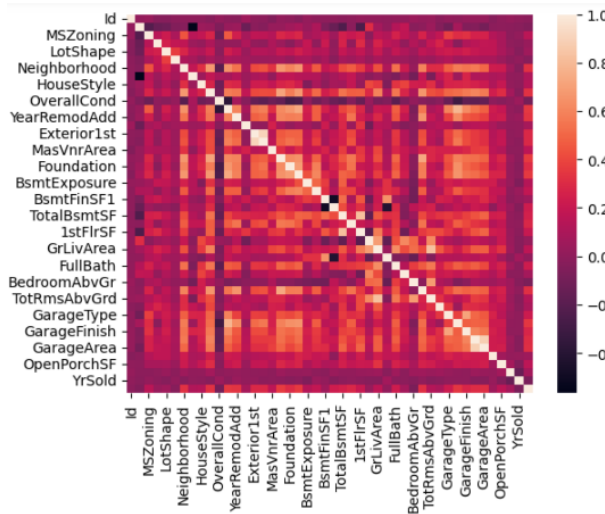
Construct the updated correlation matrix

```
comb = iterative(train, 0.65, combination=comb)
```

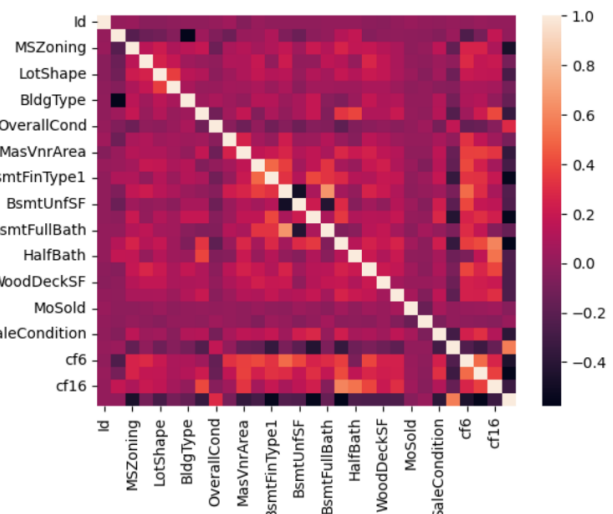
Combine columns with high correlation ship

Through iterations, it will check if any of these spaces still have a value less than the threshold we set. If so, we will combine the columns based on the updated correlation matrix.

So here we used the PCA (Principal Components Analysis) method to combine similar columns. This method is based on vector operations. It takes the angle between different vectors into consideration.



The matrix before combination



The matrix after combination

As we can see from the graph that the number of columns has been reduced a lot, and the color has become darker, which means that there are less correlations in the updated data set. Which could not only make training much more efficient, but also eliminate some redundant information which is harmful to the result.

With sigmoid, changing value of threshold gives different performance:

Threshold	Performance
0.75	0.14231
0.65	0.13616
0.60	0.13977
0.55	0.14032

We tried 4 different thresholds, and we found that the best value is attained at 0.65.

Explanation

When we combine different columns, information from different columns will be fused. This is a double-edged sword. As there could be redundant information be eliminated (The good thing). There could also be some useful information that could be ignored.

1. As threshold increases, the performance decreases (More columns could be remained). That might be because some redundant information is included which could affect the result.
2. As threshold decreases, the performance decreases (Less columns could be remained). That might be because some useful information has been eliminated, which is useful to improve the final result.

The performance seems to be represented by an open univariate quadratic function greater than 0 (idealized).

Data transformation

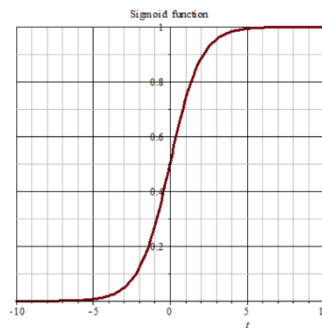
Data transformation is used to remove noise from data, scaled to fall within a small, specified range. We choose z-score normalization as our function:

```
train[train_col] = train[train_col].apply(lambda x: (x - x.mean()) / (x.std()))
```

$$v' = \frac{v - \text{mean}_A}{\text{std_dev}_A}$$

We use sigmoid function to make every data is between 0~100 at first. We think this can transform every number into 0~100, which may be better for our analysis. We don't need to handle outliers if we use this function.

$$S(t) = \frac{1}{1 + e^{-t}}$$



However, we also try not to use this function. Shockingly, deleting this function resulted in a significant improvement in performance!

0.13673

MSSubClass	MSZoning	LotArea	LotShape
29.478345	24.501843	38.875926	44.225117
29.478345	53.117239	55.935977	57.279863
45.932667	53.117239	44.124146	44.225117
34.618289	53.117239	57.073507	57.279863
29.478345	53.117239	50.333550	44.225117



0.12986

MSSubClass	MSZoning	LotArea	LotShape
-0.872264	-1.125360	-0.452531	-0.232031
-0.872264	0.124852	0.238564	0.293279
-0.163054	0.124852	-0.236125	-0.232031
-0.635860	0.124852	0.284851	0.293279
-0.872264	0.124852	0.013342	-0.232031

With sigmoid

Without sigmoid

The reason may be that sigmoid brings the outliers too close and eliminates the influence of negative numbers, which makes the model less sensitive to the data. Innovation is always good, but it doesn't always succeed, but we still gained a lot from this attempt.

By the way, I also try to just apply sigmoid into categorical columns, but the results are worse than using for all columns (0.12986 -> 0.13768).

We also compare the effects of the standard data transformation. Data transformation also leads to better result as follows:

0.13187

MSSubClass	MSZoning	LotArea	LotShape
20	-1.125360	6000	-0.232031
20	0.124852	12898	0.293279
50	0.124852	8160	-0.232031
30	0.124852	13360	0.293279
20	0.124852	10650	-0.232031

Without transformation

0.12986

MSSubClass	MSZoning	LotArea	LotShape
-0.872264	-1.125360	-0.452531	-0.232031
-0.872264	0.124852	0.238564	0.293279
-0.163054	0.124852	-0.236125	-0.232031
-0.635860	0.124852	0.284851	0.293279
-0.872264	0.124852	0.013342	-0.232031

with transformation

After learning data preprocessing, we found that using the attributes for all samples belonging to the same class to fill the missing value is smarter.

```
combined_df = pd.concat([train, test], axis=0)
combined_df[train_col] = combined_df[train_col].apply(lambda x: (x - x.mean()) / (x.std()))
train = combined_df[:len(train)]
test = combined_df[len(train):]
#test = test.drop(columns=["SalePrice"],axis=1)
#train[train_col] = train[train_col].apply(lambda x: (x - x.mean()) / (x.std()))
```

So we want to merge the training and test sets and use the overall mean and standard deviation for training, we found the score increased 0.000005, which proved this is useful.

Your Best Entry!

Your most recent submission scored 0.12741, which is an improvement of your previous score of 0.12746. Great job!

Split data

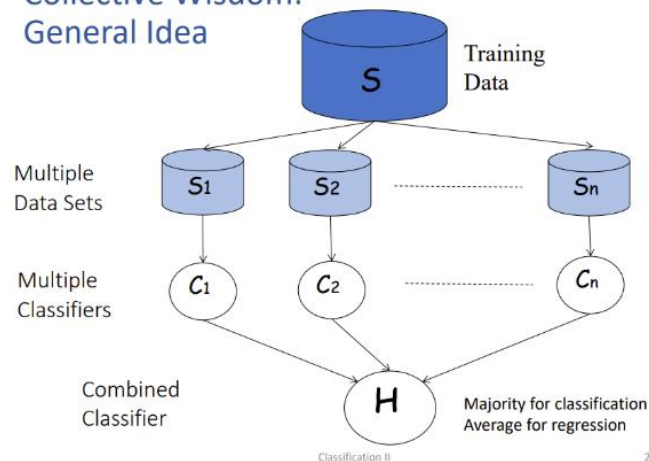
We split the data before modelling because we can test the performances based on the test data. This helps us select models and validation. We tested different 'test_size' like 0.1, 0.2, 0.15 and found 0.15 is the best choice among these.

```
X_train, X_eval, y_train, y_eval = train_test_split(train, train_y, test_size=0.15, random_state=1999)
print("Shape of X_train: ", X_train.shape)
print("Shape of X_eval: ", X_eval.shape)
print("Shape of y_train: ", y_train.shape)
print("Shape of y_eval", y_eval.shape)
```

Model

For this part, we have many ideas like using XGBoost to train our model, using k-fold cross-validation to estimate a lower bias than other methods, and combining different models to get a better result.

Collective Wisdom: General Idea



Screenshot of the slides in Lecture Note 9

Stack Regressor

To make a better prediction result, we try to build a new class which called *StackingAveragedModels* which is based on the basic models such as RandomForest and XGBoosting. The basic idea of creating this model is to use stack which could combine different basic individual models to make a new meta model as input so that the meta model could learn from the collective knowledge of the base models and might make more accurate predictions.

```

class StackingAveragedModels(BaseEstimator, RegressorMixin, TransformerMixin):
    def __init__(self, base_models, meta_model, n_folds=5):
        self.base_models = base_models
        self.meta_model = meta_model
        self.n_folds = n_folds

    def fit(self, X, y):
        self.base_models_ = [list() for x in self.base_models]
        self.meta_model_ = clone(self.meta_model)
        kfold = KFold(n_splits=self.n_folds, shuffle=True, random_state=156)

        out_of_fold_predictions = np.zeros((X.shape[0], len(self.base_models)))
        for i, model in enumerate(self.base_models):
            for train_index, holdout_index in kfold.split(X, y):
                instance = clone(model)
                self.base_models_[i].append(instance)
                instance.fit(X[train_index], y[train_index])
                y_pred = instance.predict(X[holdout_index])
                out_of_fold_predictions[holdout_index, i] = y_pred

        self.meta_model_.fit(out_of_fold_predictions, y)
        return self

    def predict(self, X):
        meta_features = np.column_stack([
            np.column_stack([model.predict(X) for model in base_models]).mean(axis=1)
            for base_models in self.base_models_ ])
        return self.meta_model_.predict(meta_features)
  
```

We first use some basic models ExtraTrees which based on random forest concept by constructing multiple decision trees using random selected feature subsets and split nodes, RandomForest which is an ensemble learning method by average or voted to make regression

prediction, XGBoosting which trains multiple decision trees and updates model parameters based on the gradients of the loss function, Bagging which use bootstrapping the training set, LGBM which builds decision trees using histogram-based algorithms and leaf-wise algorithms and KNeighbors which based on the k nearest neighborhoods to analysis the performance of these models and choose a best model to do prediction.

K-fold cross-validation has also been used in every single model we try to train. Since it is folded as equally sized subsets and could train the model in many times which could reduce dependence on a single partition. It also helps keep the stability and generalization capabilities of each model. We try to use this method to make the predictions more accurate in each training model.

To find the base models which could be used in this stacking model, we check the scores they get based on cross_val_score which cross validation is used in models that help mitigate the potential issues of overfitting or underfitting, R-squared score (r2_score) and root mean squared error (RMSE). Since we have split the training data as train one and evaluation one, we compare the prediction saleprice and the evaluation data actual price could get the scores like the result shown below.

```
CV: 0.87472135780052
R2_score (eval): 0.8882418280465572
RMSE: 29877.25231895645
CV: 0.8550704857837653
R2_score (eval): 0.8592404491197196
RMSE: 33530.493574846434
CV: 0.8747907917818166
R2_score (eval): 0.8829675967918176
RMSE: 30574.127456426784
CV: 0.8257585966765186
R2_score (eval): 0.8606139697801328
RMSE: 33366.498567931725
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000795 seconds.
You can set 'force_col_wise=true' to remove the overhead.
[LightGBM] [Info] Total Bins 2355
[LightGBM] [Info] Number of data points in the train set: 992, number of used features: 29
[LightGBM] [Info] Start training from score 179782.958669
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000618 seconds.
You can set 'force_col_wise=true' to remove the overhead.
[LightGBM] [Info] Total Bins 2361
[LightGBM] [Info] Number of data points in the train set: 993, number of used features: 29
[LightGBM] [Info] Start training from score 179628.659617
[LightGBM] [Info] Auto-choosing col-wise multi-threading, the overhead of testing was 0.000849 seconds.
You can set 'force_col_wise=true' to remove the overhead.
[LightGBM] [Info] Total Bins 2362
...
RMSE: 32081.107594900208
CV: 0.7569509224169311
R2_score (eval): 0.7608230930672869
RMSE: 43707.9696411782
```

As a conclusion from the observation of the result, we choose RandomForest, ExtraTrees, XGBoosting and Bagging as the basic model for that its CV and R2_score are higher while RMSE smaller and make the meta model be LGBM to train the stack model again. The following is the scores by the models.

```
models = [
    ('RandomForest', rmse_RandomForest, r2_score_RandomForest_eval, cv_RandomForest.mean()),
    ('XGB', rmse_XGBoost, r2_score_XGBoost_eval, cv_XGBoost.mean()),
    ('KNeighbors', rmse_KNeighbors, r2_score_KNeighbors_eval, cv_KNeighbors.mean()),
    ('Bagging', rmse_Bagging, r2_score_Bagging_eval, cv_Bagging.mean()),
    ('LGBM', rmse_LGBM, r2_score_LGBM_eval, cv_LGBM.mean()),
    ('ExtraTrees', rmse_ExtraTrees, r2_score_ExtraTrees_eval, cv_ExtraTrees.mean()),
    ('Stacking', rmse_stacking, r2_score_stacking_eval, cv_stacking.mean())
]

predict = pd.DataFrame(data = models, columns=['Model', 'RMSE', 'R2_Score(eval)', 'Cross-Validation'])
predict
```

	Model	RMSE	R2_Score(eval)	Cross-Validation
0	RandomForest	32832.221375	0.865042	0.855219
1	XGB	30574.127456	0.882968	0.874791
2	KNeighbors	43707.969641	0.760823	0.756951
3	Bagging	33322.457572	0.860982	0.837000
4	LGBM	32081.107595	0.871146	0.863011
5	ExtraTrees	31647.815172	0.874603	0.877905
6	Stacking	35255.786194	0.844382	0.862826

XGBoost's RMSE and r2 score is the best, cross-validation is a little less than ExtraTrees but not too much. We can conclude that XGBoost is the best model. It seems that our integrated model is not good enough, but we believe this is a good try.

Therefore, we use XGBoost model for the prediction for the former trails. We get 0.13377 the best based on this train.

submission_stack.csv

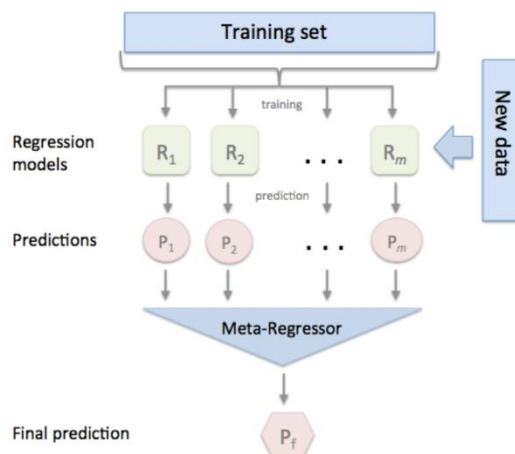
0.13377

Complete · Zhoudao LU · now

We also try to find better models based on k-fold cross-validation, model fusion and adjusting parameters. We haven't spent much time optimizing the hyperparameters. And maybe delete some awful models may benefit our predictions. So we still want to improve our modelling.

In the previous attempt, we referenced the existing codes on Kaggle and used our own *StackingAveragedModels* to fuse the models. However, an existing class, *StackingRegressor* may be able to directly help us complete this work.

Stacking regression is an ensemble learning technique to combine multiple regression models via a meta-regressor. The individual regression models are trained based on the complete training set; then, the meta-regressor is fitted based on the outputs -- meta-features -- of the individual regression models in the ensemble.



https://rasbt.github.io/mlxtend/user_guide/regressor/StackingRegressor/

We also want to optimize the hyperparameters to get a better model. We iteratively trained using cross validation for optimizing the hyperparameters with a Bayesian Optimization. Then we combined them into *stackingregressor*. We wish the fusion would make our modelling better.

```

from sklearn.ensemble import StackingRegressor

estimators = [('lasso', lasso_model), ('enet', ENet_model), ('gboost', gb_model), ('xgb', xgb_model), ('lgb', lgb_model)]
final_estimator = lasso_model
stacked_models = StackingRegressor(estimators=estimators, final_estimator=final_estimator)
  
```

We continue to use k-folds cross validation to train our data. We get a higher score than the previous XGBoost for the *stackingRegressor* (0.13377 -> 0.12803)! The improvement is very large.

submission_new.csv

0.12803

Complete · Zhoudao LU · now · ensemble_preds = stacked_pred

Because of the outstanding performance of *XGboost*, we want to combine *StackingRegressor* and *XGBoost* for prediction. For example, we assign 0.6 weights to *StackingRegressor* and 0.4 to *XGBoost*.

```
ensemble_preds = stacked_pred*0.60 + xgb_pred*0.40
```

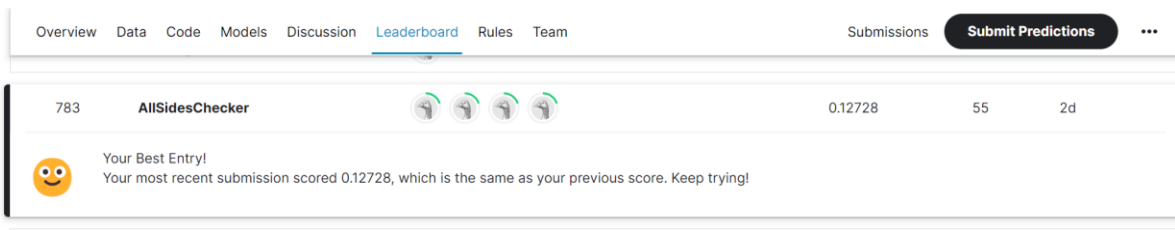
We try different weights between *StackingRegressor* and *XGBoost*






StackingRegressor	XGBoost	score
0	1	0.12785
0.6	0.4	0.12732
0.5	0.5	0.12728
0.8	0.2	0.12741
1	0	0.12803

submission_new.csv Complete · Zhoudao LU · 3m ago · $\text{ensemble_preds} = \text{stacked_pred} * 0.50 + \text{xgb_pred} * 0.50$	0.12728
submission_new.csv Complete · Zhoudao LU · 8h ago · $\text{ensemble_preds} = \text{xgb_pred}$	0.12785
submission_new.csv Complete · Zhoudao LU · 9h ago · $\text{ensemble_preds} = \text{stacked_pred}$	0.12803
submission_new.csv Complete · Zhoudao LU · 9h ago · $\text{ensemble_preds} = \text{stacked_pred} * 0.60 + \text{xgb_pred} * 0.40$	0.12732
submission_new.csv Complete · Zhoudao LU · 9h ago · $\text{ensemble_preds} = \text{stacked_pred} * 0.80 + \text{xgb_pred} * 0.20$	0.12741

Woo, we found the best model is based on 0.5 of the StackingRegressor prediction and 0.5 of XGBoost! Single StackingRegressor and XGBoost are the worst models in this case. This proves that this attempted model fusion works!

Screen capture of our team name and final score in Kaggle



Overview	Data	Code	Models	Discussion	Leaderboard	Rules	Team	Submissions	Submit Predictions	...
783	AllSidesChecker					0.12728	55	2d		
	Your Best Entry! Your most recent submission scored 0.12728, which is the same as your previous score. Keep trying!									

Which feature engineering effort is most useful in your competition work? Any reason behind?

Our group controlled of single variables, and compared the various ways by annotating the operations, changing the coefficients, such as trying to fill in the NULL value of the various methods, trying different encoding methods. Some of our attempts improved performance, while some degraded performance. However, all of them were good experiences!

We believe all feature engineering are useful. Every operation is meaningful to improve the performances. However, handling NULL is most useful for that with null data the dataset may have much noise that could not to do process later. If we don't have this step, we cannot continue to process the data; but without other steps, the modelling still can be done (it seems that some models also can handle categorical features directly).

If you have submitted more than 1 entry, describe the case your team has obtained the best improvement (not absolute performance).

We submitted more than 50 entries to compare the performance and listed them within our report in their respective parts.

It seems that our new encoding method obtained the best improvement (0.1515 -> 0.13673)! The reason is that we give higher weight to higher priced features based on price. This shows that just encode categorical features to 1, 2, 3 is not efficient than our new method based on different weights in this regression situation.

Reference

<https://www.kaggle.com/code/guanlintao/house-prices-predict-stack-regressor>

<https://www.kaggle.com/code/giovannabrodzamojska/house-prices-prediction-stackingregressor#Train>

https://rasbt.github.io/mlxtend/user_guide/regressor/StackingRegressor/

Course lecture notes