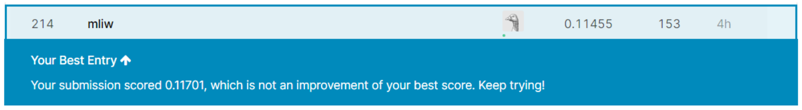
Top 10.6% on Kaggle House Price Competition with only **ONE** entry

Mingwei Li[*]

Abstract

I used to rank top 5% in the kaggle House Price Competition. My score on the public leader board is listed in the figure 1.



```
214    mliw                         0.11455    153    4h
Your Best Entry
Your submission scored 0.11701, which is not an improvement of your best score. Keep trying!
```

Figure 1: My Public Score

However, I've made more than 100 entries to achieve this score, which means I'm actually tuning my model on the test data set. In fact, the problem of data leakage arises when you make submissions more than once. In this essay, I want to take a challenge by making only one submission and the score, which ranks top 10.6%, is listed in figure 2.

```
final_5_0.10117320406891336.csv          0.11701
4 hours ago by mliw
add submission details
```

Figure 2: My Public Score with only **one** entry

This essay aims to tell you how to achieve the score of 0.11701 with only one entry from the very beginning. With the help of Genetic Algorithm and Bayesian Optimization(hyperopt), this **IS NOT** a lucky score achieved randomly, but a certain result we can **ALMOST SURELY** get.

The whole passage has 3 chapters. In the first chapter, we mainly talk about how to conduct feature selection with GA and parameter selection with hyperopt(A powerful python package). In the second chapter, feature engineering, which is more important than model selection, is explicitly described. The principle of Bayesian Optimization is introduced in the chapter 3.

The source code of this essay is available at here. Please star my project if you feel it's helpful, thanks!!

[*]Mingwei Li is a student of the School of Mathematical Science, Peking University

Contents

1 Feature Selection and Model Tuning **3**
1.1 Model Evaluation 4
1.2 Feature Selection 4
1.3 Model Tuning 6
1.4 Model Stacking 7

2 Feature Engineering and Conclusion **9**
2.1 Summary of Feature Engineering 9
2.1.1 Imputing Missing Values 9
2.1.2 Designing New Features 9
2.1.3 Deleting Outliers 10
2.2 Conclusion 11

3 The Principle of hyperopt **12**
3.1 Algorithm Principle of hyperopt[1] 12
3.2 Analysis of the Source Code of hyperopt 13
3.3 Conclusion of hyperopt 14

1 Feature Selection and Model Tuning

This chapter assumes we've already finished the feature engineering. *train_x* is a matrix of shape $(1454, 305)$, *train_y* is a vector of $(1454, 1)$. Our mission is to conduct validation and fitting on *train_x* and *train_y*. Then the fitted model is used to make prediction on *test_x*, a matrix of $(1459, 305)$. The final prediction would be uploaded to the website of Kaggle.

The House Price Competition is a regression task, and multiple models can be used to solve this problem. In this chapter, we take SVR as an example to demonstrate the whole process of **feature selection**, **model tuning** and **model stacking**. As we've mentioned before, *train_x* has 305 columns, which means we have 305 features. The model performance would be harmed if we input too many features. Therefore, it's very important to conduct feature selection. In addition, SVR is a model which comprises many parameters. Some of the parameters can effect the final performance significantly. In all, we need to find **(1)** the best feature combination and **(2)** model parameters to minimize cross-validation error.

The following codes demonstrate our *base_models*.

```
kernel_ridge = make_pipeline(RobustScaler(), KernelRidge(kernel='polynomial'))
kernel_ridge = make_pipeline(RobustScaler(), KernelRidge(kernel='polynomial',degree=2))
robust_lightgbm = make_pipeline(RobustScaler(),LGBMRegressor(random_state=42,objective="huber",n_jobs=2)
                                                    )
normal_lasso = make_pipeline(RobustScaler(), LassoCV(n_jobs=2,random_state=42))
normal_ridge = make_pipeline(RobustScaler(), Ridge())
normal_ridge = make_pipeline(RobustScaler(), RidgeCV())
normal_svr = make_pipeline(RobustScaler(), SVR())
robust_xgb = make_pipeline(RobustScaler(), XGBRegressor(random_state=1,objective="reg:squarederror",
                                                    n_jobs=2))
```

When we set **ONE ENTRY** restriction, the technological route is as followed.

Algorithm 1 Technological Route **with ONE ENTRY** restriction

```
1: Input base_models
2: for each Single_Model in base_models do
3:     Find the best Feature_Subset through Genetic Algorithm(feature selection)
4:     Find the best Model_Para with the help of hyperopt(model tuning)
5:     Make Prediction and Holdout_Prediction based on Model_Para and Feature_Subset
6: end for
7: Determine the Weights of each Single_Model based on their Holdout_Prediction
8: Make Final_Prediction based on Weights, Model_Para and Feature_Subset
```

When we remove **ONE ENTRY** restriction, the technological route is as followed.

Algorithm 2 Technological Route **without ONE ENTRY** restriction

```
1: Input base_models
2: for each Single_Model in base_models do
3:     Find the best Feature_Subset through Genetic Algorithm(feature selection)
4:     Find the best Model_Para with the help of hyperopt(model tuning) of Each_Generation.
       Step 4 tunes model parameters on each generation of Genetic Algorithm
5:     Make and test Prediction of Each_Generation
       Step 5 requires to submit multiple entries to kaggle
6:     Find the best (Model_Para, Feature_Subset) based on public score
7:     Make Prediction and Holdout_Prediction based on Model_Para and Feature_Subset
8: end for
9: Determine the Weights of each Single_Model based on their Holdout_Prediction
10: Make Final_Prediction based on Weights, Model_Para and Feature_Subset
```

1.1 Model Evaluation

We usually adopt 5-Fold cross validation to evaluate the performance of a certain model. For instance, all samples are divided into 5 parts equally. The model is fitted on 4 of the 5 parts, and the remaining part is used to calculate prediction error. Therefore, we are able to get 5 cross-validation error since each one of the five parts can be used as *test data set*. The average of the 5 errors is then calculated as an indicator of model performance.

The following code is the evaluation function of the House Price Competition. The standard deviation of the 5 holdout errors is added to the average value as a kind of penalty. We expect models whose cross-validation errors are both small and stable. Moreover, the variable *individual* in the following code stands for feature selection. *individual* is a vector consists of zeros and ones. This vector of length 305(the number of total features) represents our feature selection. The variable *total_data_cache* is our data. **Our mission** is to find the model and *individual*(feature selection) to minimize the value of *nm_penalty* function.

```
def nm_penalty(model,total_data_cache,individual):

    train_x = total_data_cache[0].copy()
    train_y = total_data_cache[1].copy()

    train_x = pd.DataFrame(train_x)
    train_y = pd.DataFrame(train_y)
    individual = np.array(individual).astype(bool)
    train_x = train_x.loc[:,individual]

    holdout = produce_holdout(model,total_data_cache,individual)
    result = np.mean(holdout)+np.std(holdout)
    print(result)

    return -result
```

WARNING:A small CV-error doesn't necessarily lead to a high public score(small test error). In fact, Genetic Algorithm and hyperopt is able to achieve a very low cross-validation error like 0.10. However, the test error of such model is usually 0.13 or even higher. We can regard the phenomenon of low CV-error accompany with high test error as **over-fitting** on the meta level. It's very difficult to avoid such phenomenon.

1.2 Feature Selection

This problem has 305 features in all. The ultimate goal of feature selection is to find a subset of these 305 features which could minimize cross-validation error based on a given model. Genetic Algorithm is a suitable method to solve this problem. The following is the pseudo code of GA. Just as we've mentioned before, the *individual* of GA is a 0-1 vector which represents our feature selection. The negative value of cross validation error is fitness value. The goal of GA is to find the individual with the highest fitness value. In order to achieve this goal, we generate a population of 0-1 vectors randomly and calculate their cv-error. Then, the *individual* with high fitness is selected through tournament. Mutation(change 0,1 randomly) and Mating(Intersection between excellent *individuals*) are executed on those vectors. As a result, a new population with higher fitness value is generated. After many rounds of iteration, we can get a population consisted of excellent *individuals*.

Algorithm 3 Genetic Algorithm for Feature Selection

```
1: Input generation, population_size, mate_probability, mutation_probability
2: Generate a Population of population_size randomly
3: for $i = 1$ to generation do
4:     Select the best population_size Individuals from current Population through tournament as the updated
       Population
5:     Conduct mutation and mating on the current Population
6: end for
7: Select the best Individual from the currtent Population
```

The Genetic Algorithm of this essay is mainly based on this project. I want to show my great appreciation for the author of this project. However, some codes are modified to increase the flexibility of GA. More importantly, some crucial parameters are also changed. Therefore, I strongly suggest that you should choose parameters based on my project or your own intuition.

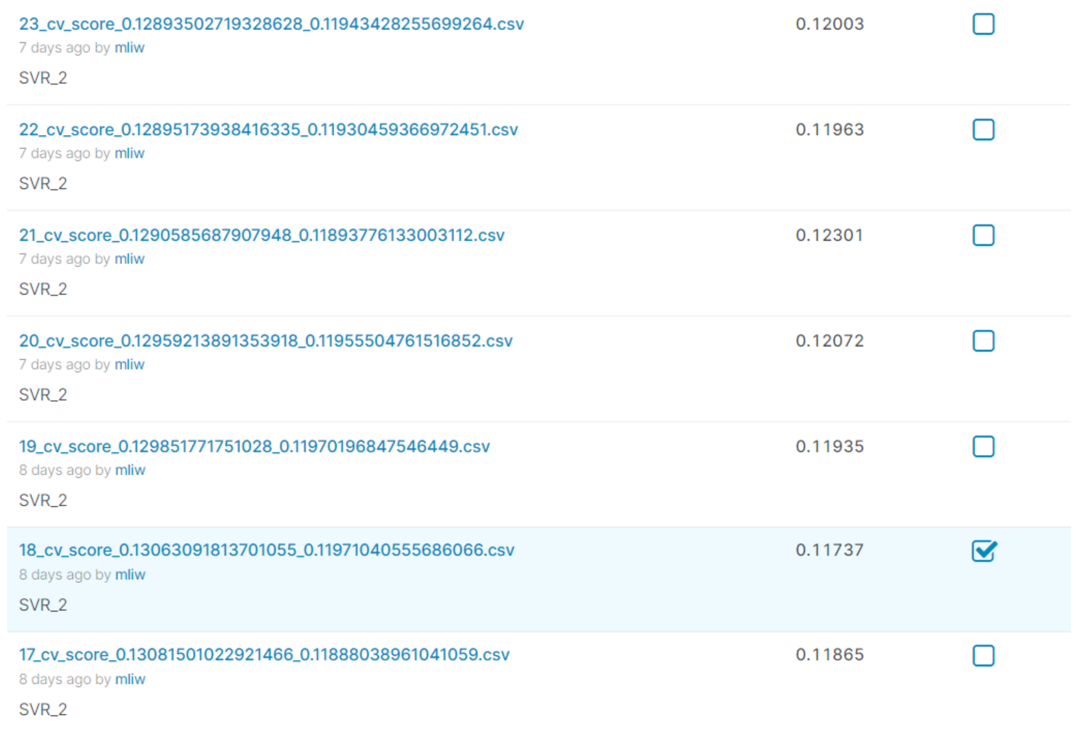
The following code is a part of my SVR. A larger *populations* or *generations* would provide better result, but also spend more time. In order to avoid over-fitting on the meta-level, the value of these variables shouldn't be too large.

```
# 2 Define model and parameters
normal_svr = make_pipeline(RobustScaler(), SVR())
svr_dic = {
  'gamma':hyperopt.hp.uniform('gamma',1e-5,1e-2),
  'C':  hyperopt.hp.uniform('C',1e-2,40),
  'epsilon':  hyperopt.hp.uniform('epsilon',1e-4,1e-2)
}


# 3 Genetic Algorithm for feature selection
key = "svr_2"
probability = 0.4
bench_model = normal_svr
length_of_features = train_x.shape[1]


# 4 Start evolving and tunning
populations = 200
generations = 35

selector = FeatureSelectionGA(bench_model,total_data_cache,length_of_features,nm_penalty,probability)
```

In previous trials which I don't set **ONE ENTRY** restriction, the best individual of each generation is saved and tested as figure 3. The name rule is as followed.

For example, the file name 18_cv_score_0.13063091813701055_0.11971040555686066.csv represents the final prediction based on the features selected in the 18th generation. 0.13063091813701055 is the cross validation error defined in the **Model Evaluation** section. The model of SVR has multiple parameters like 'gamma','C' and 'epsilon'. Therefore, we are able to reduce cross validation error by simply tuning parameters. 0.11971040555686066 is the cv-score after parameter tuning. We can find the public score of the 18th generation is 0.11737, which ranks top 10.94%. The public score of the prediction from the 23th generation is 0.12003, worse than that of the 18th generation. It's apparent that a low cv-error doesn't necessarily cause low public error.

In fact, the House Price Competition becomes quite easy when we remove **ONE ENTRY** restriction. The GA algorithm would make predictions of different generations with a decreasing trend of cv-error. We only need to submit those predictions one by one. This method is very efficient. It takes less than 20 entries to produce a answer of 0.11737(top 10.94%) in figure 3. Since several models can be used to conduct regression, we can adopt Genetic Algorithm to different base models. As there is no **ONE ENTRY** restriction, we are able to select the base prediction with the lowest public error from a certain number of generations. After that, stacking could be used to blend those excellent base predictions and to reach a better place in the public leader board. I achieve 0.11455(top 5%) through this method. **In fact, there is no difference between such method and taking the eggs from a chicken coop.**



23_cv_score_0.12893502719328628_0.11943428255699264.csv 0.12003 ☐
7 days ago by mliw
SVR_2

22_cv_score_0.12895173938416335_0.11930459366972451.csv 0.11963 ☐
7 days ago by mliw
SVR_2

21_cv_score_0.1290585687907948_0.11893776133003112.csv 0.12301 ☐
7 days ago by mliw
SVR_2

20_cv_score_0.12959213891353918_0.11955504761516852.csv 0.12072 ☐
7 days ago by mliw
SVR_2

19_cv_score_0.129851771751028_0.11970196847546449.csv 0.11935 ☐
8 days ago by mliw
SVR_2

18_cv_score_0.13063091813701055_0.11971040555686066.csv 0.11737 ☑
8 days ago by mliw
SVR_2

17_cv_score_0.13081501022921466_0.11888038961041059.csv 0.11865 ☐
8 days ago by mliw
SVR_2

Figure 3: The Best Individual of each Generation(SVR)

The Genetic Algorithm is based on Deap, a simple but powerful python package. The documentation of this package is not difficult.

1.3 Model Tuning

After feature selection is finished, we can begin to tune our model. Model tuning is a black box optimization problem. Let's assume $x \in \mathbb{R}^n$ represents model parameters, and $f(x)$ is the corresponding cross validation error. We don't know the explicit form of $f(x)$, and our goal is to find x^* to minimize $f(x^*)$. hyperopt, a python Bayesian optimization package, can be used to solve our problem. This package could be regarded as a senior version of random research. The next chapter of this essay would discuss the basic principle of this package.

The following code demonstrates the optimization on svr_dic with 2000 trials.

```
svr_dic = {
'gamma':hyperopt.hp.uniform('gamma',1e-5,1e-2),
'C':  hyperopt.hp.uniform('C',1e-2,40),
'epsilon':  hyperopt.hp.uniform('epsilon',1e-4,1e-2)
}
tunning_train_x = train_x.loc[:,items["name"]]

def objective(param):
    tuning_pipeline = make_pipeline(RobustScaler(),SVR(**param))
    loss = -nm_penalty(tuning_pipeline,[tunning_train_x,train_y],np.ones(tunning_train_x.shape[1]))
    return loss
trials = hyperopt.Trials()

best = hyperopt.fmin(objective,
space=svr_dic,
algo=hyperopt.tpe.suggest,
max_evals=2000,
trials=trials)
```

1.4 Model Stacking

8 base models are adopted in this competition. After model tuning and feature selection of every single model, we can begin to blend these models. The details of blending are listed here.

At first, 10-Fold holdout predictions are produced as the following codes. The whole train data set is divided into 10 parts equally. Then the base models fit on 9 of the 10 parts and predict on the remaining part. The predictions on the remaining parts are collected as holdout prediction.

```
def produce_holdout(model,total_data_cache):

    train_x = total_data_cache[0].copy()
    train_y = total_data_cache[1].copy()
    train_x = pd.DataFrame(train_x)
    train_y = pd.DataFrame(train_y)
    train_x = train_x.loc[:,model["name"]]
    estimator = model["model"]

    kf = KFold(N_FOLDS, shuffle=True, random_state=42)
    kf.get_n_splits(train_x,train_y)
    result = pd.DataFrame(np.zeros(train_x.shape[0]), index = train_x.index)

    for train_index, test_index in kf.split(train_x, train_y):
        tem_train_x, tem_train_y = train_x.iloc[train_index,:], train_y.iloc[train_index,:]
        tem_test_x, tem_test_y = train_x.iloc[test_index,:], train_y.iloc[test_index,:]
        tem_train_y = tem_train_y.values.reshape(-1)
        estimator.fit(tem_train_x,tem_train_y)
        prediction = estimator.predict(tem_test_x)
        result.iloc[test_index,:] = prediction.reshape(-1,1)

    return result
```

Secondly, the weights of base models are determined by quadratic optimization on their holdout predictions. The ground truth of holdout prediction is *train_y*, and this quadratic optimization can be described as followed.

$$Minimize \left\| \sum_{i=1}^{n} \omega_i \times holdout\_prediction_i - train\_y \right\|_2^2$$

$$s.t \sum_{i=1}^{n} \omega_i = 1, \omega_i \geq 0$$

The corresponding python code is listed below. cvxopt is used to conduct this optimization.

```
# 3.1 Start convex optimization
from cvxopt import solvers
from cvxopt import matrix

num = holdout_collect.shape[1]
P_mat = matrix(2*np.dot(holdout_collect.T,holdout_collect))
q_mat = matrix(-2*np.dot(target.T,holdout_collect))
G_mat = matrix(np.concatenate((np.eye(num),-np.eye(num)),axis = 0))
h_mat = matrix(np.concatenate((np.ones(num),np.zeros(num)),axis = 0))
b_mat = matrix(1.)
A_mat = matrix(np.ones(num)).T
solution = solvers.qp(P_mat,q_mat,G_mat,h_mat,A_mat,b_mat)
final_weight = np.array(solution["x"]).reshape(-1)

# 3.2 Test holdout prediction
final_holdout = np.sum(holdout_collect*final_weight,axis = 1)
holdout_err = np.sqrt(mean_squared_error(train_y,final_holdout))

return final_weight,holdout_err
```

Thirdly, it's very crucial to determine which base models are involved in the final stacking. We can achieve a low cv-error when all base models are incorporated in the final stacking. However, this would cause the problem of over-fitting on the meta-level. Therefore, a greedy algorithm is adopted to solve this problem. The results of this greedy algorithm are listed here and presented in figure 4.

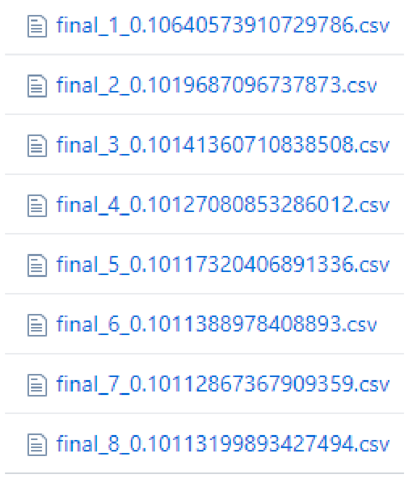

final_1_0.10640573910729786.csv
final_2_0.1019687096737873.csv
final_3_0.10141360710838508.csv
final_4_0.10127080853286012.csv
final_5_0.10117320406891336.csv
final_6_0.1011388978408893.csv
final_7_0.10112867367909359.csv
final_8_0.10113199893427494.csv

Figure 4: Stacking Results

The name rule of figure 4 is as followed. For instance, final_5_0.10117320406891336.csv represents the final prediction made by stacking 5 base models. 0.10117320406891336 is the corresponding cv error on the training data. It's apparent that more base models lead to a lower cv error in figure 4. However, there is no significant decrease in cv error after 5 base models are already involved in the stacking model. Therefore, we submit final_5_0.10117320406891336.csv as the ultimate submission to this competition, and the public score(0.11701) is listed in figure 2.

The code of greedy algorithm is listed below. The base models are added into the stacking model one by one to minimize cv error.

```
model_list = [kernel_ridge,kernel_ridge_1,normal_lassocv,normal_ridge_1,normal_ridgecv,svr_2,lightgbm,
                                                     xgb]
key_list = ["kernel_ridge","kernel_ridge_1","normal_lassocv","normal_ridge_1","normal_ridgecv","svr_2","
                                                     lightgbm","xgb"]
model_dic = dict(zip(key_list,model_list))

result = {}
record_list = []
while True:
    record_err = float("inf")
    for md_key in key_list:
        tem_record_list = record_list.copy()
        tem_record_list.append(model_dic[md_key])
        weight,err = test_combination(tem_record_list)
        if err<record_err:
            record_err = err
            md_to_add = model_dic[md_key]
            key_to_add = md_key
            print(record_err)
        record_list.append(md_to_add)
        print(len(record_list))
        print("="*30)
        result.update({len(record_list):[record_err,record_list.copy()]})
        key_list = list(set(key_list)-set([key_to_add]))
        if len(key_list)==0:
            break
```

2 Feature Engineering and Conclusion

The whole process of data analysis is listed in figure 5, and the last 3 parts in the dashed box have been introduced in chapter 1. **Feature Selection**, **Model Tuning** and **Model Stacking** can be automated by Genetic Algorithm and Bayesian Optimization effectively. However, **Feature Engineering** can only be done by human and requires a lot of tedious work. The code of the whole feature engineering process is at here.

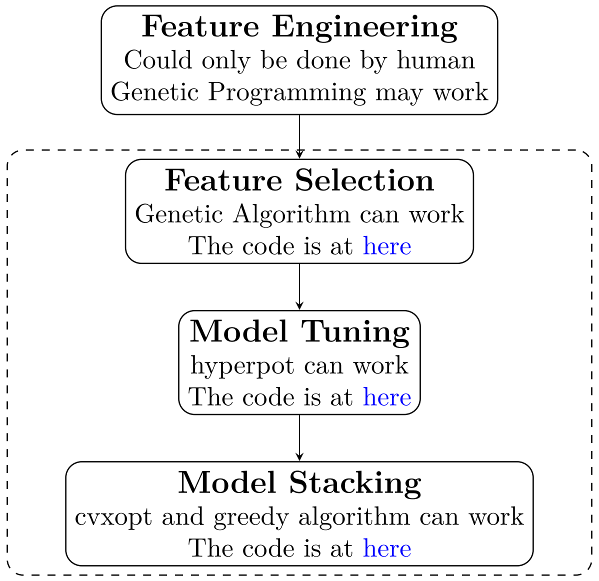


Figure 5: The Whole Process of Data Analysis

2.1 Summary of Feature Engineering

2.1.1 Imputing Missing Values

We should follow the data description and our own intuition to conduct imputation.

2.1.2 Designing New Features

There are 79 features before transforming categorical variable to numerical variable. XGBRegressor is used to calculate the cv-error of every single feature. The top 20 features regarding cv-error are listed below.

```
['OverallQual', 'Neighborhood', 'GarageCars', 'ExterQual', 'BsmtQual', 'KitchenQual','GrLivArea',
'GarageArea', 'YearBuilt', 'GarageFinish', 'GarageYrBlt', 'FullBath', 'TotalBsmtSF', 'GarageType',
'MSSubClass', 'YearRemodAdd', 'Foundation', 'TotRmsAbvGrd', 'Fireplaces', '1stFlrSF']
```

These excellent features represent the **Area**, **Garage**, **Year** and **Quality** of houses. Therefore, we should design new features of these aspects.

The new features are as followed:

```
def area_per_car(clause):
    result = clause['GarageArea'] / clause['GarageCars'] if clause['GarageCars']!=0 else 0
    return result

train_test["area_per_car"] = train_test.apply(area_per_car,axis = 1)
train_test["above_and_ground_area"] = train_test["TotalBsmtSF"]+train_test["GrLivArea"]
train_test['Total_Bathrooms'] = (train_test['FullBath'] + (0.5 * train_test['HalfBath']) +
train_test['BsmtFullBath'] + (0.5 * train_test['BsmtHalfBath']))
train_test["one_and_two"] = train_test["1stFlrSF"]+train_test["2ndFlrSF"]
train_test['Total_Porch_Area'] = (train_test['OpenPorchSF'] + train_test['3SsnPorch'] + train_test['
                                                     EnclosedPorch'] + train_test['ScreenPorch'] +
                                                     train_test['WoodDeckSF'])
```

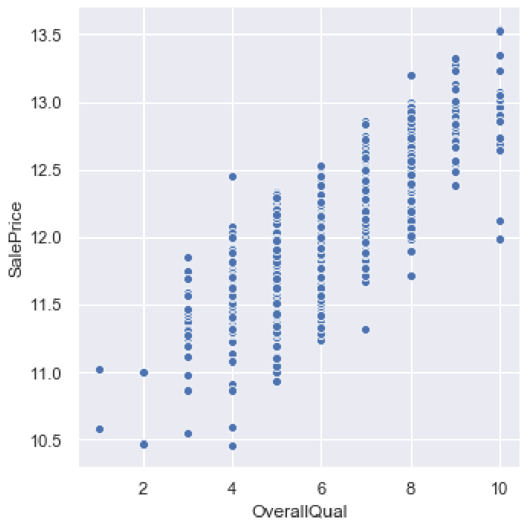
The detailed descriptions of different aspects of houses are as followed.

```
"""
{'OverallQual':Rates the overall material and finish of the house}
{'Neighborhood':Physical locations within Ames city limits}
{'GarageCars':"Size of garage in car capacity",'GarageArea': "Size of garage in square feet",'
                                                    GarageFinish': "Interior finish of the garage" \
  'GarageYrBlt': "Year garage was built", 'GarageType': "Garage location"}
{'ExterQual':Evaluates the quality of the material on the exterior,'BsmtQual': Evaluates the height of
                                                    the basement \
  'Foundation': "Type of foundation"}
{'KitchenQual': Kitchen quality}
{'GrLivArea': Above grade (ground) living area square feet,'TotRmsAbvGrd':"Total rooms above grade (does
                                                     not include bathrooms)", \
  'FullBath': "Full bathrooms above grade",'1stFlrSF': "First Floor square feet",'TotalBsmtSF: "Total
                                                    square feet of basement area"}
{'YearBuilt': "Original construction date",'YearRemodAdd': "Remodel date (same as construction date if
                                                    no remodeling or additions)"}
{'MSSubClass': "Identifies the type of dwelling involved in the sale."}
{'Fireplaces': "Number of fireplaces"}
"""
```

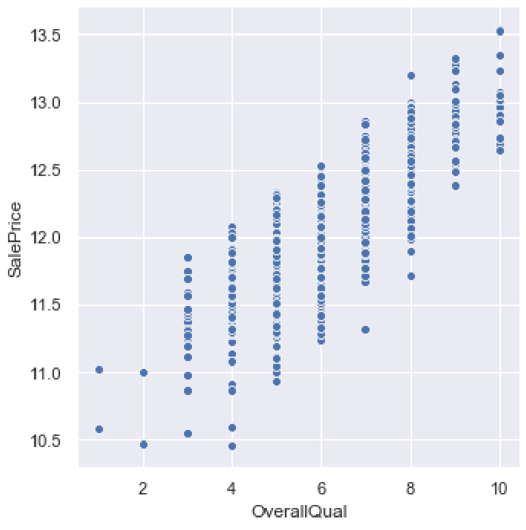
2.1.3 Deleting Outliers

Whatever the problem we are trying to solve, there is a perfect function $f(x)$ which can best describe the relationship between x and y. Our goal is to approximate this perfect function as closely as possible, and machine learning is one of the tools which can help us achieve this goal. However, outliers would bring bias to our final model.

The feature *OverallQual* is the feature with the lowest cv-error, and figure 6 demonstrates the influence of outliers to cv-error. An improvement of 0.00345 is made by deleting 3 outliers.



(a) Scatter Plot **with** Outliers(cv-error 0.2303)



(b) Scatter Plot **without** Outliers(cv-error 0.2269)

Figure 6: Influence of Outliers to Cross Validation Error

The outliers of important features are detected by human eyes, and 7 data points are removed from the train data.

```
logi_0 = np.logical_and(tem_train.SalePrice>=12.25, tem_train.OverallQual==4)
logi_1 = np.logical_and(tem_train.SalePrice<=11.5, tem_train.OverallQual==7)
logi_2 = np.logical_and(tem_train.SalePrice<=12.5, tem_train.OverallQual==10)
logi_3 = np.logical_and(tem_train.SalePrice<=12.5, tem_train.above_and_ground_area>=6000)
logi_4 = np.logical_and(tem_train.SalePrice<=11, tem_train.ExterQual=="Gd")
logi_5 = np.logical_and(tem_train.SalePrice<=12.5, tem_train.one_and_two>=4000)
logi_6 = np.logical_and(tem_train.SalePrice<=11.5, tem_train.GarageArea>=1200)
logi_7 = np.logical_and(tem_train.SalePrice<=12.5, tem_train.TotalBsmtSF>=5000)
```

2.2 Conclusion

Feature Selection, **Model Tuning** and **Model Stacking** can be automated by Genetic Algorithm and Bayesian Optimization effectively. However, **Feature Engineering** relies heavily on domain knowledge and human intuition.

As for this problem, there is a large room for improvement in **Feature Engineering**. For example, we could change the skewness of some features to make it more like a normal distribution. In addition, some numerical features like *MSSubClass* can be treated as categorical features. A better public score can be achieved if more efforts are involved in feature engineering.

3 The Principle of hyperopt

This chapter will focus on the analysis of the principle and source code of hyperopt. In the analysis of hyperopt, we will refer to statistical terms such as posterior probability distribution. The essence of hyperopt is to fit unknown functions with statistical models. It needs to be emphasized again that statistical models are only means to solve problems, and some statistical concepts may not have practical significance.

3.1 Algorithm Principle of hyperopt[1]

We first consider a simple case in which the argument x of the unknown function $f(x)$ is a real number rather than a vector. After random sampling of x, we can use the function f to calculate the function value of different x. Then, all x can be divided into two categories based on the corresponding $f(x)$. $\{x : f(x) < y^*\}$and $\{x : f(x) \geq y^*\}$, respectively. (y^* is a given parameter, and the default value of hyperopt is 25% quantile of all function values of the samples). We regard $\{x : f(x) < y^*\}$ and $\{x : f(x) \geq y^*\}$ as two random variables, and all samples are divided into two categories as sampling of these two random variables. When we have samples of these two random variables, these samples can be used to estimate the probability density function (PDF). hyperopt uses the Gaussian mixture model (GMM) to estimate these two PDFs. The Gaussian mixture model uses several probability density functions of Gaussian distribution to approximate a certain probability density function. Equation 1 is an explicit form of GMM model.

$$P(x) = \sum_{i=1}^{n} a_i \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad \sum_{i=1}^{n} a_i = 1 \tag{1}$$

The following code comes from the source code of hyperopt, *scope.adaptive_parzen_normal* combines new observation *obs*, *prior_mu* and *prior_weight* to get posterior *mus*, *sigmas*.

```
def ap_uniform_sampler(obs, prior_weight, low, high, size=(), rng=None):
    prior_mu = 0.5 * (high + low)
    prior_sigma = 1.0 * (high - low)
    weights, mus, sigmas = scope.adaptive_parzen_normal(
    obs, prior_weight, prior_mu, prior_sigma
    )

    return scope.GMM1(
    weights, mus, sigmas, low=low, high=high, q=None, size=size, rng=rng
    )
```

Then GMM1 simulates the distributions of these 2 random variables($\{x : f(x) < y^*\}$and $\{x : f(x) \geq y^*\}$). The following code is a part of the source code of GMM1 function. It's apparent that GMM1 samples from a multinomial distribution based on the variable *weights*. After the variable *active* is sampled, GMM1 get final samples based on *mus* and *sigmas*.

```
active = np.argmax(rng.multinomial(1, weights, (n_samples,)), axis=1)
samples = rng.normal(loc=mus[active], scale=sigmas[active])
```

At this stage, There are 2 GMM which could provide samples of $\{x : f(x) < y^*\}$and $\{x : f(x) \geq y^*\}$.

The whole process can be summarized as followed: At first, we sample x randomly. Then all samples of x are divided into 2 categories based on cut point y^*. GMM is used to simulate the PDF of these 2 categories(The $l(x)$ and $g(x)$ in equation 2). We set $\gamma = p(y < y^*)$ for the convenience of reasoning.

$$p(x|y) = \begin{cases} l(x) & if \quad y < y^* \\ g(x) & if \quad y \geq y^* \end{cases} \tag{2}$$

The essay[1] sets an optimization goal:

$$EI_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y)p(y|x)dy = \int_{-\infty}^{y^*} (y^* - y)\frac{p(x|y)p(y)}{p(x)}dy \tag{3}$$

According to equation 2:

$$p(x) = \int_{\mathbb{R}} p(x|y)p(y)dy = \gamma l(x) + (1-\gamma)g(x) \tag{4}$$

Consider the numerator of equation 3:

$$\int_{-\infty}^{y^*} (y^* - y)p(x|y)p(y)dy = l(x)\int_{-\infty}^{y^*} (y^* - y)p(y)dy = l(x)y^*\gamma - l(x)\int_{-\infty}^{y^*} yp(y)dy \tag{5}$$

Use equation 5 to divide equation 4, we have:

$$EI_{y^*}(x) = \frac{l(x)y^*\gamma - l(x)\int_{-\infty}^{y^*} yp(y)dy}{\gamma l(x) + (1-\gamma)g(x)} = \frac{y^*\gamma - \int_{-\infty}^{y^*} yp(y)dy}{\gamma + (1-\gamma)\frac{g(x)}{l(x)}} \tag{6}$$

In order to maximize the optimization goal, we need to minimize $\frac{g(x)}{l(x)}$. The dual problem is to maximize $log(l(x)) - log(g(x))$. hyperopt would get $\_default\_n\_EI\_candidates$ samples based on $l(x)$ and calculate the corresponding $log(l(x)) - log(g(x))$ values of these samples. The x with the maximum $log(l(x)) - log(g(x))$ is selected as the best parameter of this round. After that, we would input x to the unknown function to calculate the corresponding $f(x)$. Calculating $f(x)$ usually costs a lot of time. As for machine learning, calculating $f(x)$ is the same as fitting on the whole data set multiple times to calculate cross validation error. Finally, hyperopt would start a new round of circulation, explore new x until the number of circulation reaches the maximum value set before.

3.2 Analysis of the Source Code of hyperopt

The pseudocode of hyperopt is listed below.

Algorithm 4 Pseudocode of hyperopt

```
1: Input $max\_evals$, $feature\_space$, $objective\_fn$
2: Generate 20 samples from $feature\_space$ and calculate their $objective\_fn$ value
3: Use variable trials to record current samples and their $objective\_fn$ value
4: for i=1 to $max\_evals$ do
5:     Set cut point $y^*$ based on the $objective\_fn$ value in the variable trials
       # The default value of $y^*$ is the 25th-percentile of $objective\_fn$ value
6:     Set the variable tem_feature to empty
7:     for j=1 to dimension of $feature\_space$ do
8:         Divide all samples of $feature\_space$[j] in variable trials into 2 groups based on $y^*$
9:         Use GMM model to simulate the pdf of these 2 groups ($l(x)$ and $g(x)$)
10:        Use $l(x)$ to get 24 samples
11:        Calculate the value of $log(l(x)) - log(g(x))$ of each $x$ in the previous 24 samples
12:        Incorporate the $x$ with highest log-likelihood difference into the variable tem_feature
13:    end for
14:    Calculate $objective\_fn$ value of the variable tem_feature in the previous loop
       # Calculation of $objective\_fn$ value is expensive
15:    Incorporate the variable tem_feature and its $objective\_fn$ value into the variable trials
16: end for
17: Select the best individual from the variable trials
```

We assume the input of $objective\_fn$ is a parameter vector of multiple dimensions, the output is the cross validation error of bench mark model. At first, hyperopt selects 20 parameter vectors randomly, and calculate their cross validation errors. *rand.suggest* in the following codes represents random sampling on the parameter vector.

```
if len(docs) < n_startup_jobs:
    # N.B. THIS SEEDS THE RNG BASED ON THE new_id
    return rand.suggest(new_ids, domain, trials, seed)
```

Then, hyperopt divides samples from each dimension of parameter space into 2 categories based on $oloss\_gamma$. The obs_below represents model parameters whose cross validation error is lower than $oloss\_gamma$, and obs_above represents model parameters whose cross validation error is higher than $oloss\_gamma$.

```
for nid in prior_vals:
    # construct the leading args for each call to adaptive_parzen_sampler
    # which will permit the "adaptive parzen samplers" to adapt to the
    # correct samples.
    obs_below, obs_above = scope.ap_split_trials(
    obs_idxs[nid], obs_vals[nid], obs_loss_idxs, obs_loss_vals, oloss_gamma
    )
    obs_memo[prior_vals[nid]] = [obs_below, obs_above]
```

After that, hyperopt selects $n\_EI\_candidates$ parameter candidates randomly based on $l(x)$(The probability density function in equation 2), and calculates their corresponding $log(l(x)) - log(g(x))$ values. The parameter with the largest difference in likelihood is selected.

```
# calculate the log likelihood of b_post under both distributions
below_llik = fn_lpdf(*([b_post] + b_post.pos_args), **b_kwargs)
above_llik = fn_lpdf(*([b_post] + a_post.pos_args), **a_kwargs)
# compute new_node based on below & above log likelihood
new_node = scope.broadcast_best(b_post, below_llik, above_llik)
```

As the model parameter combination is a vector of multiple dimensions, hyperopt conducts samples dividing, GMM simulation and calculating $log(l(x)) - log(g(x))$ for every single element of this vector. The final selected parameter combination and its cv-error are record in $self.trial$.

```
if len(new_trials):
    self.trials.insert_trial_docs(new_trials)
    self.trials.refresh()
```

3.3 Conclusion of hyperopt

Advantages: When the dimension of parameter vectors is low or the relationship between different parameters is clear, hyperopt would have a very excellent performance.

Disadvantages: hyperopt naturally models on every single parameter individually. This package would build m independent GMM models if the dimension of model parameters is m. hyperopt may have poor performance if the interaction between different parameters is complicated. Some essays indicate that the difference between random search and Bayesian Optimization is subtle when the dimension of model parameters is high. Therefore, we suggest that some parameters can be optimized by batches, just like the SMO algorithm of SVM.

Reference

[1] J. Bergstra, R. Bardenet, Y. Bengio *et al.* "*Algorithms for hyper-parameter optimization*". *Advances in Neural Information Processing Systems*, **2011**: 2546–2554.