Advanced Feature Engineering and Regression Techniques

April 24, 2024

First, we import the relevant libraries for the preprocessing section.

[27]:
```
import os
import pandas as pd
import numpy as np
import scipy.stats as scps
```

Reading in the data.

[28]:
```
os.chdir('C:\\Users\ordav\Desktop\kaggle\HousePrices')
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
```

Getting some first impressions on the data

[29]:
```
train.head()
```

[29]:
```
   Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
0   1          60       RL         65.0     8450   Pave   NaN      Reg
1   2          20       RL         80.0     9600   Pave   NaN      Reg
2   3          60       RL         68.0    11250   Pave   NaN      IR1
3   4          70       RL         60.0     9550   Pave   NaN      IR1
4   5          60       RL         84.0    14260   Pave   NaN      IR1

  LandContour Utilities  ... PoolArea PoolQC Fence MiscFeature MiscVal MoSold  \
0         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      2
1         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      5
2         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      9
3         Lvl    AllPub  ...        0    NaN   NaN         NaN       0      2
4         Lvl    AllPub  ...        0    NaN   NaN         NaN       0     12

  YrSold  SaleType  SaleCondition  SalePrice
0   2008        WD         Normal     208500
1   2007        WD         Normal     181500
2   2008        WD         Normal     223500
3   2006        WD        Abnorml     140000
4   2008        WD         Normal     250000

[5 rows x 81 columns]
```

Our target variable is 'SalePrice'. We will seperate it from the explanatory variables. The Id is irrelevant as a feature, but we will need the test points' id's for the submissions.

[30]:
```
y_train = train.SalePrice
testId = test.Id
train.drop('Id', axis = 1, inplace=True)
test.drop('Id', axis = 1, inplace=True)
```

Looking at how nuch missing data we have. We will drop features with more than 20% missing entries.

[31]:
```
n = len(train.index)
n
```

[31]:
```
1460
```

[32]:
```
m = len(test.index)
m
```

[32]:
```
1459
```

[33]:
```
train.isna().sum()[train.isna().sum() > n * 0.2].sort_values()
```

[33]:
```
FireplaceQu     690
Fence          1179
Alley          1369
MiscFeature    1406
PoolQC         1453
dtype: int64
```

[34]:
```
test.isna().sum()[test.isna().sum() > m * 0.2].sort_values()
```

[34]:
```
FireplaceQu     730
Fence          1169
Alley          1352
MiscFeature    1408
PoolQC         1456
dtype: int64
```

[35]:
```
colsToDrop = ['FireplaceQu', 'Fence', 'Alley', 'MiscFeature', 'PoolQC']
train.drop(colsToDrop, axis = 1, inplace=True)
test.drop(colsToDrop, axis = 1, inplace=True)
```

[36]:
```
colsToDrop = ['TotalBsmtSF', 'TotRmsAbvGrd', 'GarageYrBlt', 'GarageCars']
train.drop(colsToDrop, axis = 1, inplace=True)
test.drop(colsToDrop, axis = 1, inplace=True)
```

We have a large amount of categorical variables, and a lot of them has a high number of categories. We want to create a function that will transform them to ordinal variables. Lets describe the algorithm:

1. Look at a categorical feature x.
2. Look at each category in it {z1, z2, . . . , zk}
3. Seperate the 'SalePrice' by the category the observation belongs to. We have k numerical vectors.
4. We compare each to vectors using Wilcoxon's rank sum test, to check for a significant difference. Since we have up to (k Choose 2) seperate null hypothesis, we need to make a Bonfferoni correction to our significance level to keep our FWER as 0.05. We do not want to be too conservative, so we will look at it like making k tests.
5. We now have s batches of categories. We consider each batch as equivalent (with respect to its effect on the target variable).
6. We choose a representative category from each batch and compute the mean of 'SalePrice' in the category.
7. We rank the batches by the mean we computed.
8. An observation's new ordinal feature is the rank of the batch where its category is located.

[37]:
```
from operator import itemgetter

def batches(x, alpha):
    vals = pd.unique(train[x])
    nUnique = len(vals)
    to_compare = {}
    for i in range(nUnique):
        to_compare[vals[i]] = train.loc[train[x] == vals[i], 'SalePrice']
    batches = {}
    appended = []
    for j in to_compare.keys():
        if j not in appended:
            batches[j] = [j]
            appended.append(j)
            for k in to_compare.keys():
                if (k not in appended):
                    t = scps.ranksums(x = to_compare[j], y = to_compare[k])
                    p = t[1]
                    if p >= (alpha / nUnique):
                        batches[j].append(k)
                        appended.append(k)
    return batches

def catToOrd(x, alpha):
    d = batches(x, alpha)
    e = []
    train_new = pd.Series(0, index=np.arange(len(train.index)))
    test_new = pd.Series(0, index=np.arange(len(test.index)))
    for cat in d.keys():
        mean_y = np.mean(train.loc[train[x] == cat, 'SalePrice'])
        e.append((cat, mean_y))
```

```
    e = sorted(e, key = itemgetter(1))
    for i in range(len(train.index)):
        filled = False
        for j in range(len(e)):
            if train[x][i] in d[e[j][0]]:
                train_new[0 ,i] = j
                filled = True
        if not filled:
            train_new[0, i] = np.nan
    for i in range(len(test.index)):
        filled = False
        for j in range(len(e)):
            if test[x][i] in d[e[j][0]]:
                test_new[0 ,i] = j
                filled = True
        if not filled:
            test_new[0, i] = np.nan

    return [train_new, test_new]

def catToOrdDF(alpha):
    colsToDrop = []
    types = train.dtypes
    for j in range(len(types)):
        if types[j] == object:
            x = train.columns[j]
            ordx = catToOrd(x, alpha)
            train['New'+x] = ordx[0]
            test['New'+x] = ordx[1]
            colsToDrop.append(x)
    train.drop(colsToDrop, axis = 1, inplace = True)
    test.drop(colsToDrop, axis = 1, inplace = True)
```

[38]:
```
catToOrdDF(0.05)
```

```
C:\Users\ordav\anaconda3\lib\site-packages\scipy\stats\stats.py:7784:
RuntimeWarning: invalid value encountered in double_scalars
  z = (s - expected) / np.sqrt(n1*n2*(n1+n2+1)/12.0)
```

We notice that the 'MSSubClass' feature is also categorical, although it encoded with numerical values.

[39]:
```
newCols = catToOrd('MSSubClass', 0.05)
```

[40]:
```
train.MSSubClass = newCols[0]
test.MSSubClass = newCols[1]
```

We can now drop the 'SalePrice' column from the training set.

[41]:
```
train.drop('SalePrice', axis = 1, inplace = True)
```

We need to remember that our process might produce features that are constant .We check for features like that.

[42]:
```
sum(train.std() == 0)
```

[42]: 4

We drop these columns.

[43]:
```
boolConst = (train.std() == 0)
colsToDrop = boolConst.index[boolConst]
train.drop(colsToDrop, axis = 1, inplace=True)
test.drop(colsToDrop, axis = 1, inplace=True)
```

We want to avoid high correlation between features. We check which of them has a high linear correlation using Pearson's Correlation Coefficient, and drop ones who are closely realated to others. We drop the ones with a smaller Pearson's Correlation Coefficient to the target variable.

[44]:
```
def dropCorrs(alpha):
    corrs = train.corr()
    corrCols = []
    appearences = []
    for i in range(len(corrs.index) - 1):
        for j in range(i):
            if abs(corrs).iloc[i,j] > alpha:
                corrCols.append([corrs.columns[i], corrs.columns[j]])
    colsToDrop = []
    for k in corrCols:
        if (k[0] not in colsToDrop) and (k[1] not in colsToDrop):
            p0 = y_train.corr(train[k[0]])
            p1 = y_train.corr(train[k[1]])
            if p0 > p1:
                colsToDrop.append(k[1])
            else:
                colsToDrop.append(k[0])
    return(colsToDrop)
```

[45]:
```
colsDrop = dropCorrs(0.75)
train.drop(colsDrop, axis = 1, inplace = True)
test.drop(colsDrop, axis = 1, inplace = True)
```

We are measured using the RMSE between the log of the predictions and the log of the real values, and therefore we will use the log of 'SalePrice' for training.

[46]:
```
log_y_train = np.log(1 + y_train)
```

We fill the missing data using Iterative Imputer, an imputation method that uses the other features and a regression model to fill in the missing entries.

[48]:
```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

imputer = IterativeImputer(max_iter=100)
train = pd.DataFrame(imputer.fit_transform(train), columns=train.columns)
test = pd.DataFrame(imputer.transform(test), columns=test.columns)
```

Scaling the data.

[49]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
train = pd.DataFrame(scaler.fit_transform(train), columns=train.columns)
test = pd.DataFrame(scaler.transform(test), columns=test.columns)
```

We can now start and train some models. Tuning their parameters will be done using 5-fold CV.

We first try to fit lasso to the data. We will also use this as feature selection method for different methods.

[50]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Lasso

paramGrid = {'alpha' : [i * 0.001 for i in range(1,100)]}
lasso = Lasso(max_iter = 1000)
gridSearch = GridSearchCV(estimator=lasso, param_grid=paramGrid)
g = gridSearch.fit(train, log_y_train)
print('Optimal alpha value: ' + str(g.best_params_['alpha']))
```

```
Optimal alpha value: 0.003
```

[52]:
```
lasso = Lasso(alpha = g.best_params_['alpha'], max_iter = 1000)
lasso.fit(train, log_y_train)
y_pred = lasso.predict(test)
y_pred = np.exp(y_pred) - 1
y_pred = pd.DataFrame({'Id' : testId, 'SalePrice' : y_pred})
y_pred.to_csv('HousePricePredictionsLasso.csv', index = False)
```

We got a score of 0.13428. Lets see if we can improve. We will try and use a random forest, with only the features that had a non-zero coefficient with lasso. We need to tune the number of trees, the level of pruning and the number of features we choose from at each split.

[53]:
```
train_reduced = train.loc[: , lasso.coef_ >  0]
test_reduced = test.loc[: , lasso.coef_ >  0]
```

[55]:
```
from sklearn.ensemble import RandomForestRegressor

paramGrid = {'n_estimators' : [500, 1000], 'ccp_alpha' : [0.001 * i for i in␣
 ↪range(4)],
             'max_features' : [0.2, 0.3]}
```

```
rFor = RandomForestRegressor()
gridSearch = GridSearchCV(estimator=rFor, param_grid=paramGrid)
g = gridSearch.fit(train_reduced, log_y_train)
print('Optimal ccp_alpha value: ' + str(g.best_params_['ccp_alpha']) + '\n' +
     'Optimal number of trees: ' + str(g.best_params_['n_estimators']) + '\n' +
     'Optimal number of features in each split: ' + str(g.
 ↪best_params_['max_features']))
```

```
Optimal ccp_alpha value: 0.0
Optimal number of trees: 500
Optimal number of features in each split: 0.3
```

[57]:
```
rFor = RandomForestRegressor(n_estimators = g.best_params_['n_estimators'],
                             ccp_alpha = g.best_params_['ccp_alpha'],
                            max_features = g.best_params_['max_features'])
rFor.fit(train_reduced, log_y_train)
y_pred = rFor.predict(test_reduced)
y_pred = np.exp(y_pred) - 1
y_pred = pd.DataFrame({'Id' : testId, 'SalePrice' : y_pred})
y_pred.to_csv('HousePricePredictionsRandomForest.csv', index = False)
```

We got a score of 0.13572. This is small decrease in preformence than we got using Lasso. It implies that the linear model is not a bad idea. Therefore, next we try and use a Support Vector Regressor with a linear kernel. We need to try and find the optimal epsilon (allowed error) and C (inverse regularization coefficient).

[64]:
```
from sklearn.svm import SVR
linearSvr = SVR(kernel = 'linear')
paramGrid = {'C' : [0.001 * i for i in range(1, 10)], 'epsilon' : [0.01 * i for␣
 ↪i in range(6, 10)]}
gridSearch = GridSearchCV(estimator=linearSvr, param_grid=paramGrid)
g = gridSearch.fit(train_reduced, log_y_train)
print('Optimal C value: ' + str(g.best_params_['C']) + '\n' +
     'Optimal epsilon value: ' + str(g.best_params_['epsilon']))
```

```
Optimal C value: 0.001
Optimal epsilon value: 0.07
```

[66]:
```
linearSvr = SVR(kernel = 'linear', C=g.best_params_['C'], epsilon = g.
 ↪best_params_['epsilon'])
linearSvr.fit(train_reduced, log_y_train)
y_pred = linearSvr.predict(test_reduced)
y_pred = np.exp(y_pred) - 1
y_pred = pd.DataFrame({'Id' : testId, 'SalePrice' : y_pred})
y_pred.to_csv('HousePricePredictionsLinearSVR.csv', index = False)
```

We got a score of 0.13680, which is worse than both of the previous models. We will therefore go back to tree based methods, specifically XGBoost. We will adjust the learning rate and the lambda & alpha regularization parameters (l2 & l1).

```
import xgboost as xgb
from sklearn.model_selection import GridSearchCV

xgb1 = xgb.XGBRegressor(n_estimators = 500, max_depth = 4)
paramGrid = {'learning_rate' : [0.01 * i for i in range(1, 5)],
             'reg_alpha' : [0.001 * i for i in range(1, 5)], 'reg_lambda' : [0.
 ↪00001 * i for i in range(1, 5)]}
gridSearch = GridSearchCV(estimator=xgb1, param_grid=paramGrid)
g = gridSearch.fit(train_reduced, log_y_train)
print('Optimal learning rate: ' + str(g.best_params_['learning_rate']) + '\n' +
     'Optimal lambda: ' + str(g.best_params_['reg_lambda']) + '\n' +
     'Optimal alpha: ' + str(g.best_params_['reg_alpha']))
```

```
Optimal learning rate: 0.04
Optimal lambda: 4e-05
Optimal alpha: 0.001
```

In order to avoid overfitting we will add an early stopping criteria. If we had not made an improvement in the last 10 steps, we will stop the training and return only the ensemble we have built upn to that point.

```
from sklearn.model_selection import train_test_split

xgb1 = xgb.XGBRegressor(n_estimators = 500, max_depth = 4,
                          learning_rate = g.best_params_['learning_rate'],
                         reg_lambda = g.best_params_['reg_lambda'],
                          reg_alpha = g.best_params_['reg_alpha'],␣
 ↪early_stopping_rounds = 10)

X_train_1, X_val, y_train_1, y_val = train_test_split(train_reduced,␣
 ↪log_y_train, test_size = 0.1)
xgb1.fit(X_train_1, y_train_1, eval_set = [(X_val, y_val)], verbose = 0)
y_pred = xgb1.predict(test_reduced)
y_pred = np.exp(y_pred) - 1
y_pred = pd.DataFrame({'Id' : testId, 'SalePrice' : y_pred})
y_pred.to_csv('HousePricePredictionsXGB.csv', index = False)
```

We got a small improvement to our best score with 0.13214.

Now we will try a new approach. We will combine all of the prior models into one using exponential weighting. This process is supposed to reduce prediction variance (and therefore avoid overfitting) with a slight increase in bias (compared to the best model).

First, we need to train the model on a subset of the training data. We will use a random subset of 90%. The rest of the data will be used in order to determine the weights.

```
from sklearn.model_selection import train_test_split
X_train_1, X_weights_1, y_train_1, y_weights_1 = train_test_split(train_reduced,␣
 ↪log_y_train, test_size = 0.1)
```

```
X_train_2, X_weights_2, y_train_2, y_weights_2 = train_test_split(train,␣
 ↪log_y_train, test_size = 0.1)
```

[70]:
```
lasso = Lasso(alpha = 0.003, max_iter = 1000)
lasso.fit(X_train_2, y_train_2)
y_pred = lasso.predict(X_weights_2)
e1 = np.exp(-np.linalg.norm(y_pred - y_weights_2))
```

[71]:
```
rFor = RandomForestRegressor(n_estimators = 500, max_features = 0.3, ccp_alpha=0)
rFor.fit(X_train_1, y_train_1)
y_pred = rFor.predict(X_weights_1)
e2 = np.exp(-np.linalg.norm(y_pred - y_weights_1))
```

[72]:
```
linearSvr = SVR(kernel = 'linear', C = 0.001, epsilon = 0.07)
linearSvr.fit(X_train_1, y_train_1)
y_pred = linearSvr.predict(X_weights_1)
e3 = np.exp(-np.linalg.norm(y_pred - y_weights_1))
```

[73]:
```
xgb1 = xgb.XGBRegressor(n_estimators = 500, max_depth = 4,
                          learning_rate = 0.04,
                         reg_lambda = 4 * (10 ** -5),
                          reg_alpha = 0.001, early_stopping_rounds = 10)

X_train_2, X_val_1, y_train_2, y_val_1 = train_test_split(X_train_1, y_train_1,␣
 ↪test_size = 0.1)
xgb1.fit(X_train_2, y_train_2, eval_set = [(X_val_1, y_val_1)], verbose = 0)
y_pred = xgb1.predict(X_weights_1)
e4 = np.exp(-np.linalg.norm(y_pred - y_weights_1))
```

[74]:
```
den = e1 + e2 + e3 +  e4
w1 = e1/den
w2 = e2/den
w3 = e3/den
w4 = e4/den
```

[78]:
```
y_pred_lasso = lasso.predict(test)
y_pred_rFor = rFor.predict(test_reduced)
y_pred_svr = linearSvr.predict(test_reduced)
y_pred_xgb = xgb1.predict(test_reduced)

y_pred = w1 * y_pred_lasso + w2 * y_pred_rFor +  w3 * y_pred_svr + w4 *␣
 ↪y_pred_xgb
y_pred = np.exp(y_pred) - 1
y_pred = pd.DataFrame({'Id' : testId, 'SalePrice' : y_pred})
y_pred.to_csv('HousePricePredictionsAggregation.csv', index = False)
```

We indeed manage to get a better result than all the other models: 0.12725!