IBM Machine Learning

Course 2: Supervised Learning: Regression

Topic: Concrete Strength regression

1. Introduction

Concrete is the most important material in civil engineering. The strength of concrete will affect the durability and safety of building. We would like to predict the compressive strength of concrete from the mixture ingredients. There are 9 columns in this dataset. The first 7 column is the amount of ingredient and the 8th column is how long has the ingredient been mixed. The last column is the target variable. The dataset can be found in kaggle: https://www.kaggle.com/maajdl/yeh-concret-data

2. Explorative Data Analysis

In [115]:
```
## import the library and dataset
import warnings
warnings.filterwarnings('ignore')

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

data=pd.read_csv("data/Concrete_Data.csv")
```

2.1 The datatype of each column

In [116]:
```
data.shape
## There are 1030 rows and 9 columns
```
Out[116]:
```
(1030, 9)
```

In [117]:
```
data.dtypes
## all the columns are numeric
```
Out[117]:
```
cement              float64
slag                float64
flyash              float64
water               float64
superplasticizer    float64
coarseaggregate     float64
fineaggregate       float64
age                   int64
csMPa               float64
dtype: object
```

In [118]:
```
## First few rows
data.head()
```
Out[118]:

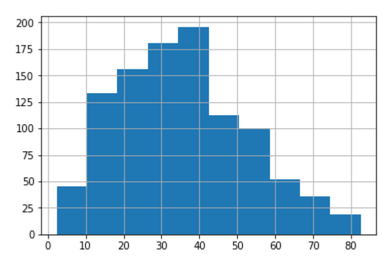
	cement	slag	flyash	water	superplasticizer	coarseaggregate	fineaggregate	age	csMPa
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30

In [119]:
```
data.info()
## There is not missing value
```
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1030 entries, 0 to 1029
Data columns (total 9 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   cement            1030 non-null   float64
 1   slag              1030 non-null   float64
 2   flyash            1030 non-null   float64
 3   water             1030 non-null   float64
 4   superplasticizer  1030 non-null   float64
 5   coarseaggregate   1030 non-null   float64
 6   fineaggregate     1030 non-null   float64
 7   age               1030 non-null   int64
 8   csMPa             1030 non-null   float64
dtypes: float64(8), int64(1)
memory usage: 72.5 KB
```

2.2 Normality of target variable

Linear Regression has a few assumptions. The normality of target variable usually will lead to better results

In [120]:
```
data.csMPa.hist()
```
Out[120]:
```
<matplotlib.axes._subplots.AxesSubplot at 0xbde38d0>
```



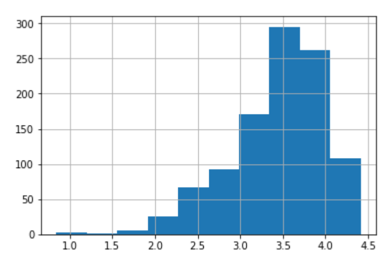
In [121]:
```
## It does look normal. Let's verify statistically
from scipy.stats import normaltest # D'Agostino K^2 Test

normaltest(data.csMPa)
```
Out[121]:
```
NormaltestResult(statistic=33.64775006912804, pvalue=4.937236241653123e-08)
```

The p-value is very low which means the distribution is not normal, We try use different data transformations.

2.3 Transformation of target

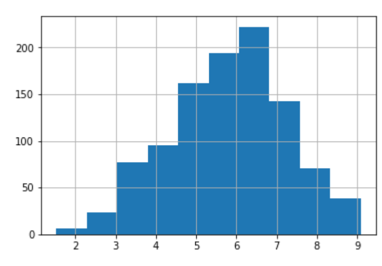
In [122]:
```
## log transformation
log_csMPa=np.log(data.csMPa)
log_csMPa.hist()
```
Out[122]:
```
<matplotlib.axes._subplots.AxesSubplot at 0xbe73470>
```



In [123]:
```
normaltest(log_csMPa)
```
Out[123]:
```
NormaltestResult(statistic=116.49011367635413, pvalue=5.0639944034032195e-26)
```

The normality after logarithm transformation becomes worse. This transformation won't be used.

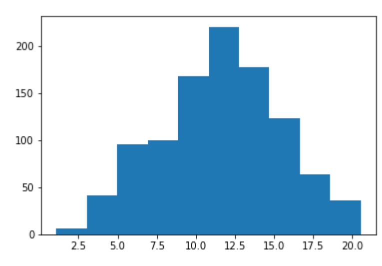
In [124]:
```
## square root transformation
sqrt_csMPa=np.sqrt(data.csMPa)
sqrt_csMPa.hist()
```
Out[124]:
```
<matplotlib.axes._subplots.AxesSubplot at 0xbef00b8>
```



In [125]:
```
normaltest(sqrt_csMPa)
```
Out[125]:
```
NormaltestResult(statistic=16.805694309514372, pvalue=0.00022422800266260203)
```

In [126]:
```
## boxcox transformation
from scipy.stats import boxcox
bc_csMPa=boxcox(data.csMPa)
lam=bc_csMPa[1]
plt.hist(bc_csMPa[0])

normaltest(bc_csMPa[0])
```
Out[126]:
```
NormaltestResult(statistic=16.991190824768964, pvalue=0.00020436653985295058)
```



Even though the data after square root and cobox are still not normal, the shape become much better. We will compare the model performance before and after transformation

2.4 Testing regression after transformation

In [151]:
```
## create train test split
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

X=data.iloc[:,:-1]
y=data.csMPa
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.3, random_state=345)

y_train_sqrt=np.sqrt(y_train)
y_train_bc, lam=boxcox(y_train)

## Build three model use original target variable, squared root target variable and
## box-cox transformed target variable
lr_origin=LinearRegression().fit(X_train, y_train)
lr_sqrt=LinearRegression().fit(X_train, y_train_sqrt)
lr_bc=LinearRegression().fit(X_train, y_train_bc)
```

In [154]:
```
## model performance
## lr_origin performance
y_pred_origin=(lr_origin.predict(X_test))
r2_score(y_pred_origin, y_test)
```
Out[154]:
```
0.3958357096077
```

In [155]:
```
## lr_sqrt performance
y_pred_sqrt=(lr_sqrt.predict(X_test))**2
r2_score(y_pred_sqrt, y_test)
```
Out[155]:
```
0.44840374471025446
```

In [156]:
```
## lr_bc performance
from scipy.special import inv_boxcox
y_pred_bc=inv_boxcox(lr_bc.predict(X_test), lam)
r2_score(y_pred_bc, y_test)
```
Out[156]:
```
0.43688157875317046
```

The result has shown that the model performace increases after target variable transformation. We will use square root transformation of target varible for future analysis

3. Transform the independent variable

We will continue to transform the independent variables

3.1 create polynomial features

In [131]:
```
from sklearn.preprocessing import (StandardScaler,
                                   PolynomialFeatures)
```

In [132]:
```
pf=PolynomialFeatures(degree=2, include_bias=False)
X_pf=pf.fit_transform(X)
```

3.2 create train test split

In [133]:
```
from sklearn.model_selection import train_test_split

X_pf_train, X_pf_test, y_train, y_test=train_test_split(X_pf, y, test_size=0.3, random_state=345)
```

3.3 Standard Scale X

In [134]:
```
s=StandardScaler()
X_pf_train_s=s.fit_transform(X_pf_train)

X_pf_test_s=s.transform(X_pf_test)
```

3.4 Linear model on original X, polynomial X and standard scaled X

In [135]:
```
## Build three models on original X, polynomial X and scaled polynomial X
lr_origin=LinearRegression().fit(X_train, y_train_sqrt)

lr_pf=LinearRegression().fit(X_pf_train, y_train_sqrt)

lr_pf_s=LinearRegression().fit(X_pf_train_s, y_train_sqrt)
```

In [136]:
```
## model performance of original X

y_pred=(lr_origin.predict(X_test))**2
r2_score(y_pred, y_test)
```
Out[136]:
```
0.44840374471025446
```

In [137]:
```
## model performance of polynomial X

y_pred_pf=lr_pf.predict(X_pf_test)**2
r2_score(y_pred_pf, y_test)
```
Out[137]:
```
0.7245110645321935
```

In [138]:
```
## model performance of scaled polynomial X

y_pred_pf_s=lr_pf_s.predict(X_pf_test_s)**2
r2_score(y_pred_pf_s, y_test)
```
Out[138]:
```
0.7245110645279285
```

As we see above, adding polynomial features significantly increase the r^2 score. While scaled does change the score for linear regression. It may have different effect on regularization before and after scaling. In the following section, we will only build model on the data after polynomial features.

4 Linear Regression with regularization

4.1 Pipeline and cross_val_predict

In [139]:
```
from sklearn.model_selection import KFold, cross_val_predict
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

estimator=Pipeline([("polynomial_features", PolynomialFeatures()),
                    ("Scaler",StandardScaler()),
                    ("regression", LinearRegression())])

params = {
    'polynomial_features__degree': [1, 2, 3]}
```

In [140]:
```
## create k-fold split
kf=KFold(shuffle=True, random_state=72018, n_splits=3)

grid = GridSearchCV(estimator, params, cv=kf)

grid.fit(X_train,y_train_sqrt)
```
Out[140]:
```
GridSearchCV(cv=KFold(n_splits=3, random_state=72018, shuffle=True),
       error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('polynomial_features', PolynomialFeatures(degree=2, include_bias=True, interaction_only=
False)), ('Scaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('regression', LinearR
egression(copy_X=True, fit_intercept=True, n_jobs=None,
         normalize=False))]),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'polynomial_features__degree': [1, 2, 3]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

In [141]:
```
grid.best_score_, grid.best_params_
```
Out[141]:
```
(0.7159810786141326, {'polynomial_features__degree': 3})
```

In [142]:
```
y_predict = grid.predict(X_test)
r2_score(y_test, y_predict**2)
```
Out[142]:
```
0.7979886673001146
```

4.2 Pipeline with Lasso Regression

In [143]:
```
estimator=Pipeline([("polynomial_features", PolynomialFeatures()),
                    ("Scaler",StandardScaler()),
                    ("lasso_regression", Lasso())])

params = {
    'polynomial_features__degree': [1, 2, 3],
    'lasso_regression__alpha': np.geomspace(1e-9, 1, 10)}
```

In [144]:
```
grid = GridSearchCV(estimator, params, cv=kf)
grid.fit(X_train,y_train_sqrt)
```
Out[144]:
```
GridSearchCV(cv=KFold(n_splits=3, random_state=72018, shuffle=True),
       error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('polynomial_features', PolynomialFeatures(degree=2, include_bias=True, interaction_only=
False)), ('Scaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('lasso_regression', L
asso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False))]),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'polynomial_features__degree': [1, 2, 3], 'lasso_regression__alpha': array([1.e-0
9, 1.e-08, 1.e-07, 1.e-06, 1.e-05, 1.e-04, 1.e-03, 1.e-02,
       1.e-01, 1.e+00])},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

In [145]:
```
grid.best_score_, grid.best_params_
```
Out[145]:
```
(0.8549284748177066,
 {'lasso_regression__alpha': 0.001, 'polynomial_features__degree': 3})
```

In [146]:
```
y_predict = grid.predict(X_test)
r2_score(y_test, y_predict**2)
```
Out[146]:
```
0.8172137493885941
```

4.3 Pipeline with Ridge Regression

In [147]:
```
estimator=Pipeline([("polynomial_features", PolynomialFeatures()),
                    ("Scaler",StandardScaler()),
                    ("ridge_regression", Ridge())])

params = {
    'polynomial_features__degree': [1, 2, 3],
    'ridge_regression__alpha': np.geomspace(1, 20, 30)}
```

In [148]:
```
grid = GridSearchCV(estimator, params, cv=kf)
grid.fit(X_train,y_train_sqrt)
```
Out[148]:
```
GridSearchCV(cv=KFold(n_splits=3, random_state=72018, shuffle=True),
       error_score='raise-deprecating',
       estimator=Pipeline(memory=None,
     steps=[('polynomial_features', PolynomialFeatures(degree=2, include_bias=True, interaction_only=
False)), ('Scaler', StandardScaler(copy=True, with_mean=True, with_std=True)), ('ridge_regression', R
idge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001))]),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'polynomial_features__degree': [1, 2, 3], 'ridge_regression__alpha': array([ 1.
  ,  1.10883,  1.22949,  1.36329,  1.51165,  1.67616,
        1.85857,  2.06083,  2.2851 ,  2.53377,  2.80951,  3.11526,
        3.45428,  3.83019,  4.24701,  4.70919,  5.22167,  5.78992,
        6.42001,  7.11867,  7.89336,  8.75236,  9.70484, 10.76097,
       11.93203, 13.23054, 14.67036, 16.26686, 18.03711, 20.     ])},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

In [149]:
```
grid.best_score_, grid.best_params_
```
Out[149]:
```
(0.8477801686735869,
 {'polynomial_features__degree': 3, 'ridge_regression__alpha': 1.0})
```

In [150]:
```
y_predict = grid.predict(X_test)
r2_score(y_test, y_predict**2)
```
Out[150]:
```
0.8342387019846206
```

5 Conclusion

As we can see from above, the vanilla linear regression on original X and y had the r^2 score of 0.4. After we transformed y, the r^2 score increased to 0.45. We further transformed X by polynomial features, and r^2 score increased to 0.72. Further adding L2 regularization. The final model had the r^2 score of 0.83. We can further adjust the hyperparameters of regularization term.