Project Report

Team:

Sumaanyu Maheshwari, Ebru Kaşıkaralar, Shannon Nakamura, Yamaç Karakuş, Ferdinand Calisir

Table of Contents

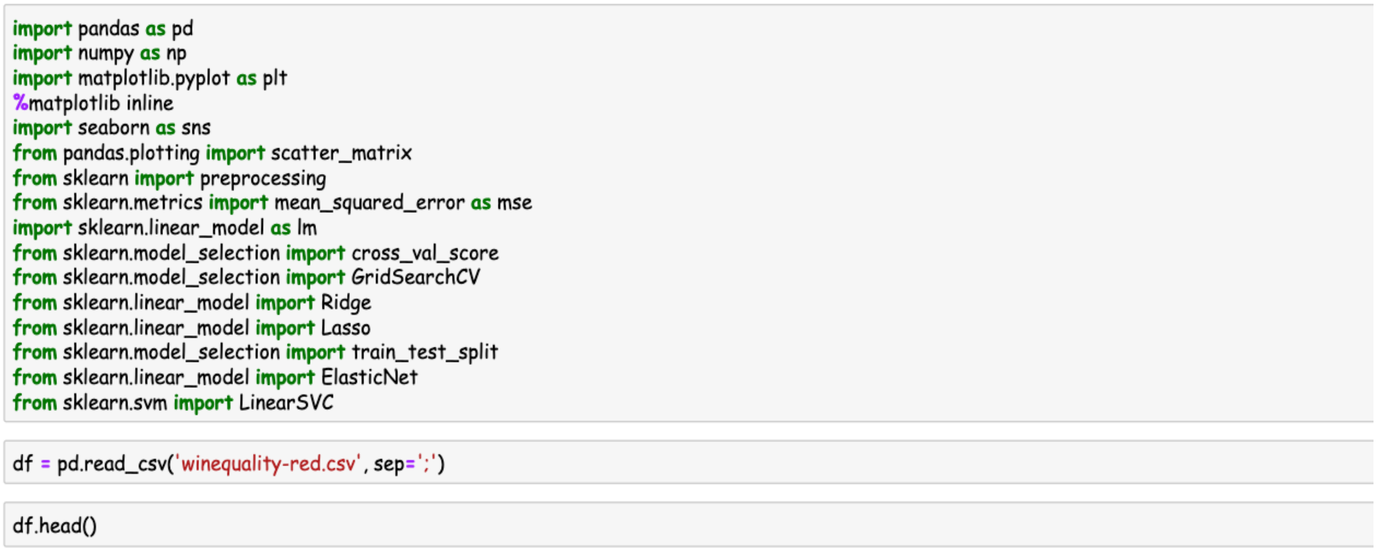
PART 1

Model Background

In this project, we will be using various regression methods to identify the coefficients of a linear model relating wine quality to different features of the wine: fixed acidity, volatile, acidity citric acid, residual sugar, chlorides, free sulfur dioxide, total sulfur dioxide, density, pH, sulphates and alcohol. Our design matrix (X) is composed of the wine features and our response variable will be the quality of wine. We will be using Pandas and ML libraries (Scikit-learn) to build our models. Our models will be based on the methods of ordinary least square, ridge regression, lasso regression, and elastic net. Moreover, we will be including the plots of tuning parameter versus cross-validation error, coefficients labeled by parameter and the minimum squared error or cross-validation error based on the choice of optimal parameters. Throughout our explanation, we will also be attaching our source code.

Data from: P. Cortez, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, Modeling wine preferences by data mining from physicochemical properties, Decision Support Systems, vol. 47, no. 4:547-553, 2009.

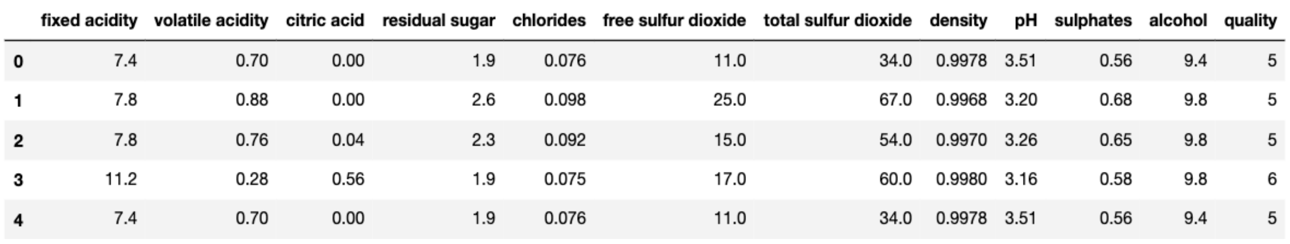
Exploratory Data Analysis



```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from pandas.plotting import scatter_matrix
from sklearn import preprocessing
from sklearn.metrics import mean_squared_error as mse
import sklearn.linear_model as lm
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split
from sklearn.linear_model import ElasticNet
from sklearn.svm import LinearSVC
```

```
df = pd.read_csv('winequality-red.csv', sep=';')
```

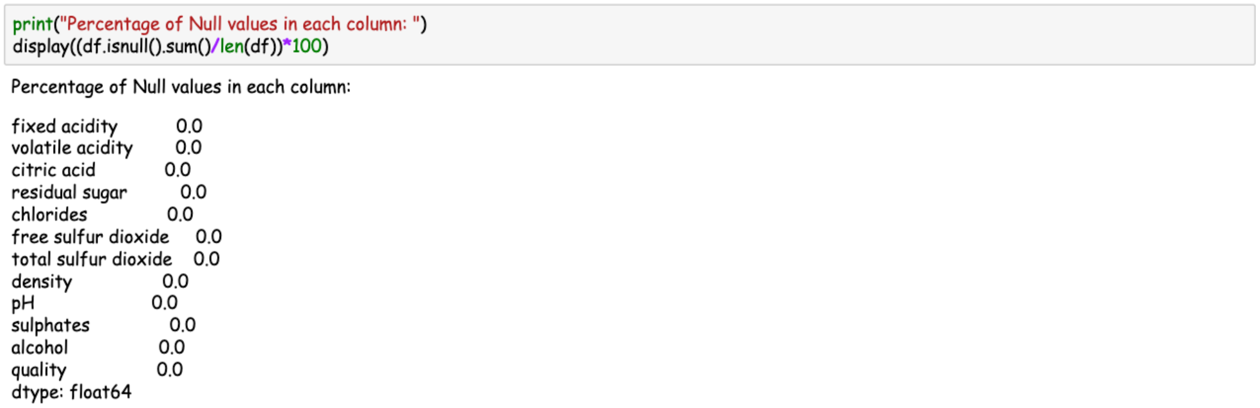
```
df.head()
```



	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5

We have imported all the relevant library for data analysis and building models. Above, we have also imported the data frame into our Jupyter notebook. Following is exploratory data analysis of the data provided to us for the project.

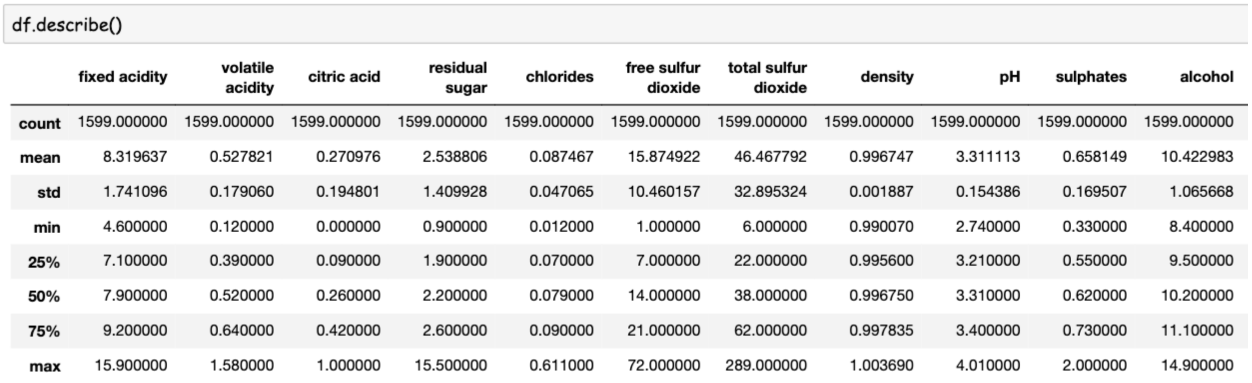
We have first checked whether there are any Null values in the data frame. If there were any Null values in the data frame, we would first have to handle them without changing the structure of the data. We have realized there is no Null value in the data frame.



```
print("Percentage of Null values in each column: ")
display((df.isnull().sum()/len(df))*100)
```

```
Percentage of Null values in each column:

fixed acidity          0.0
volatile acidity       0.0
citric acid            0.0
residual sugar         0.0
chlorides              0.0
free sulfur dioxide    0.0
total sulfur dioxide   0.0
density                0.0
pH                     0.0
sulphates              0.0
alcohol                0.0
quality                0.0
dtype: float64
```

We also wanted to check the statistics of the features. Following command has given us the mean, min, max, median, 25th percentile and 75th percentile and standard variation of the data regarding each feature.

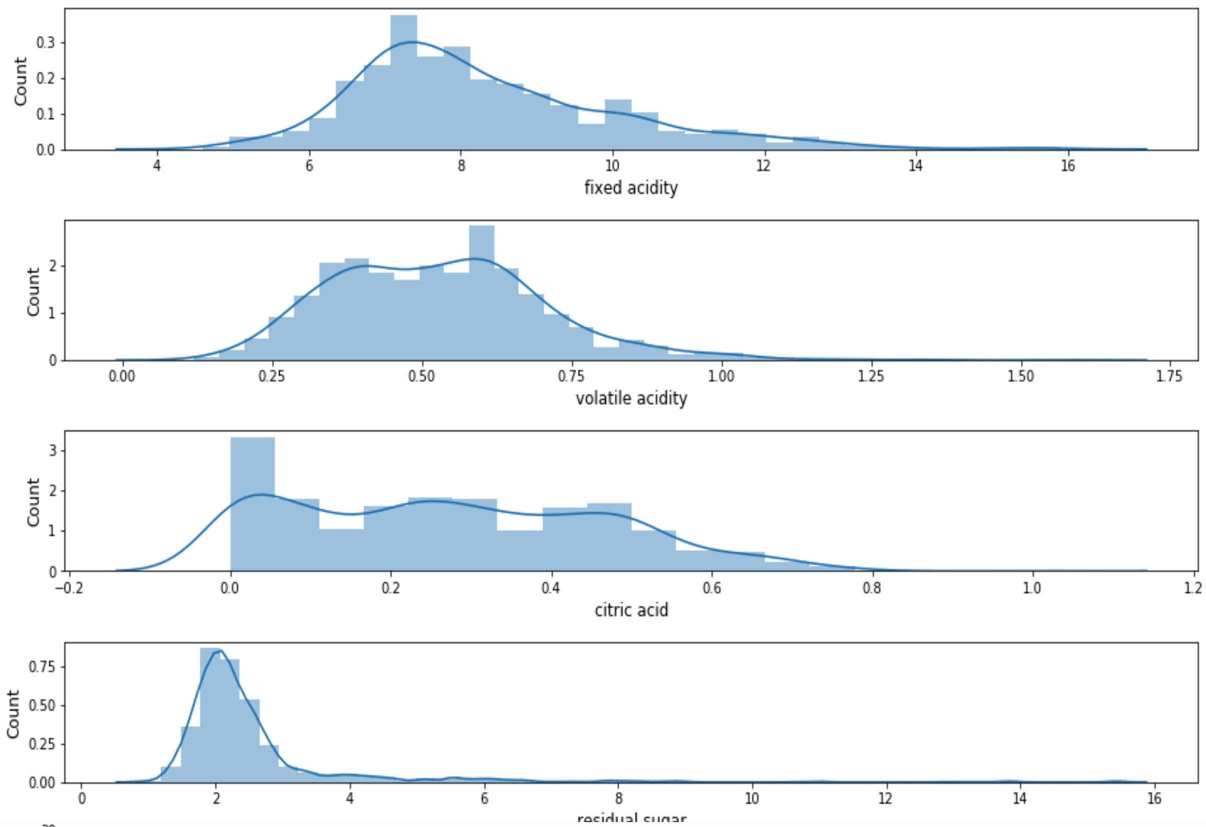


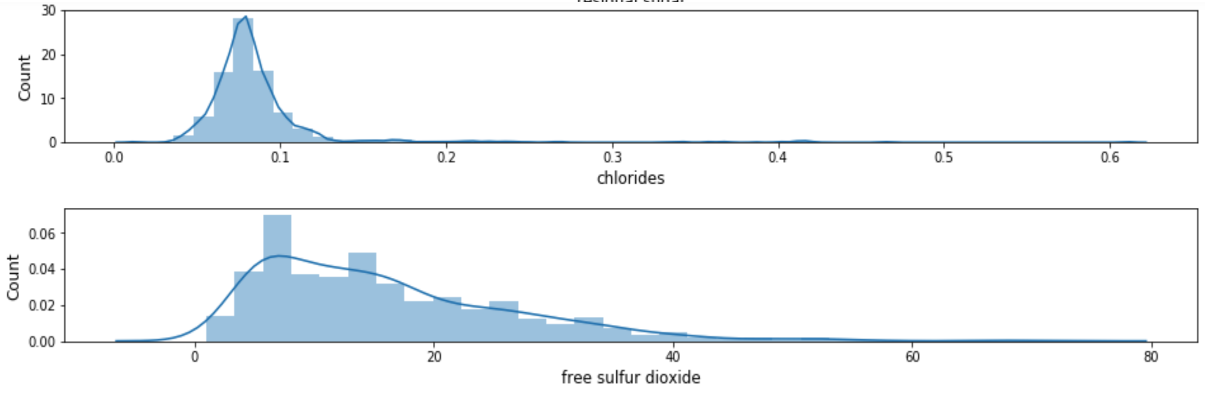
```
df.describe()
```

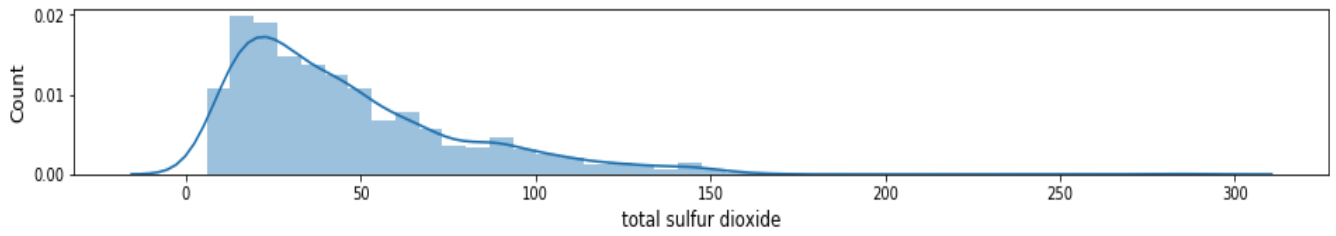
	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	46.467792	0.996747	3.311113	0.658149	10.422983
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	32.895324	0.001887	0.154386	0.169507	1.065668
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.000000	0.990070	2.740000	0.330000	8.400000
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.000000	0.995600	3.210000	0.550000	9.500000
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.000000	0.996750	3.310000	0.620000	10.200000
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62.000000	0.997835	3.400000	0.730000	11.100000
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.000000	1.003690	4.010000	2.000000	14.900000

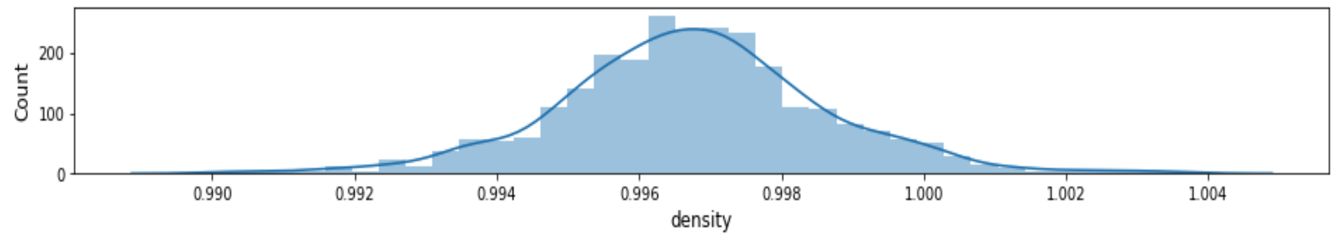
Following are the density plots showing the distribution of all of the features:

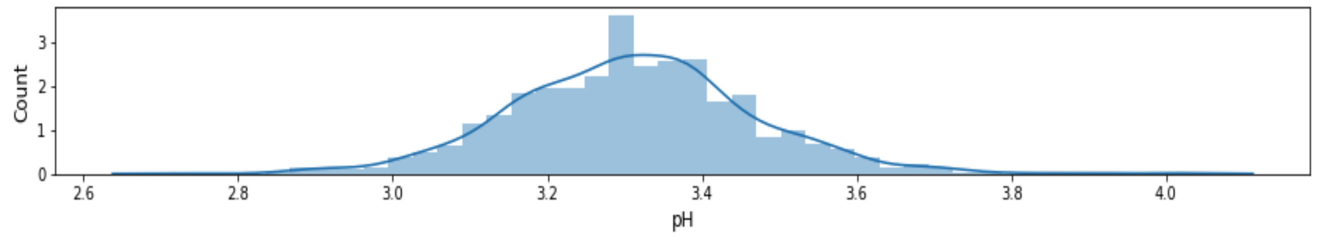
```
# check for skewness
col_names = df.columns

fig, ax = plt.subplots(len(col_names), figsize=(16,12))

for i, col_val in enumerate(col_names):

    sns.distplot(df[col_val], hist=True, ax=ax[i])
    ax[i].set_xlabel(col_val, fontsize=12)
    ax[i].set_ylabel('Count', fontsize=12)

plt.subplots_adjust(bottom = 0.01, top = 2, hspace = 0.5 )
plt.show()
```

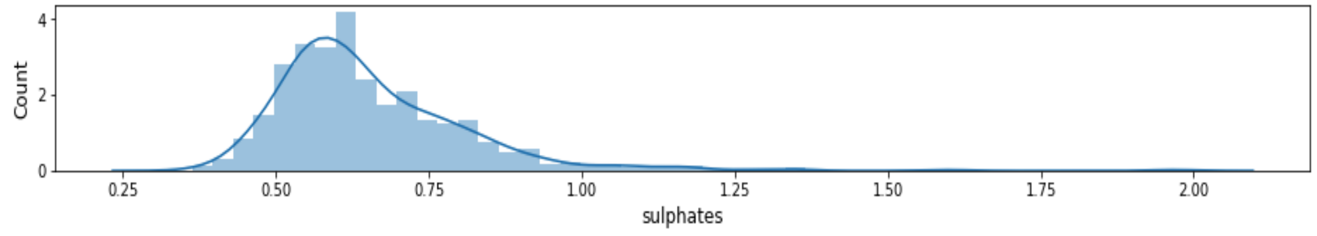


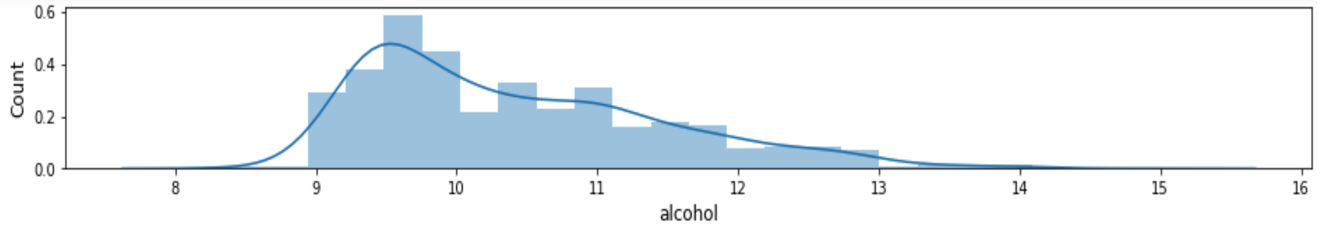


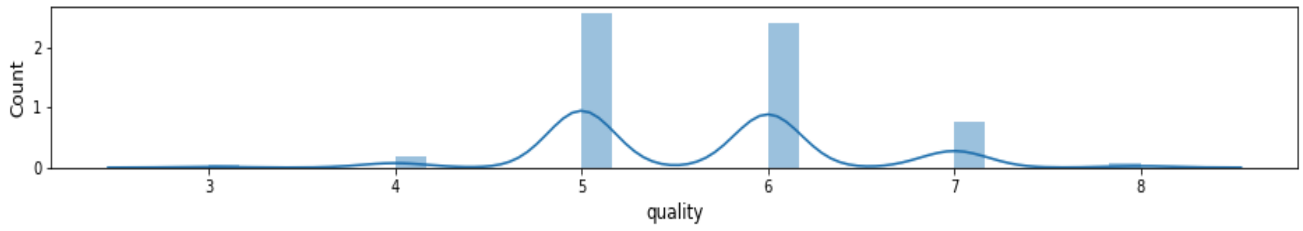

Count
0.02
0.01
0.00
0
50
100
150
200
250
300
total sulfur dioxide


Count
200
100
0
0.990
0.992
0.994
0.996
0.998
1.000
1.002
1.004
density


Count
3
2
1
0
2.6
2.8
3.0
3.2
3.4
3.6
3.8
4.0
pH

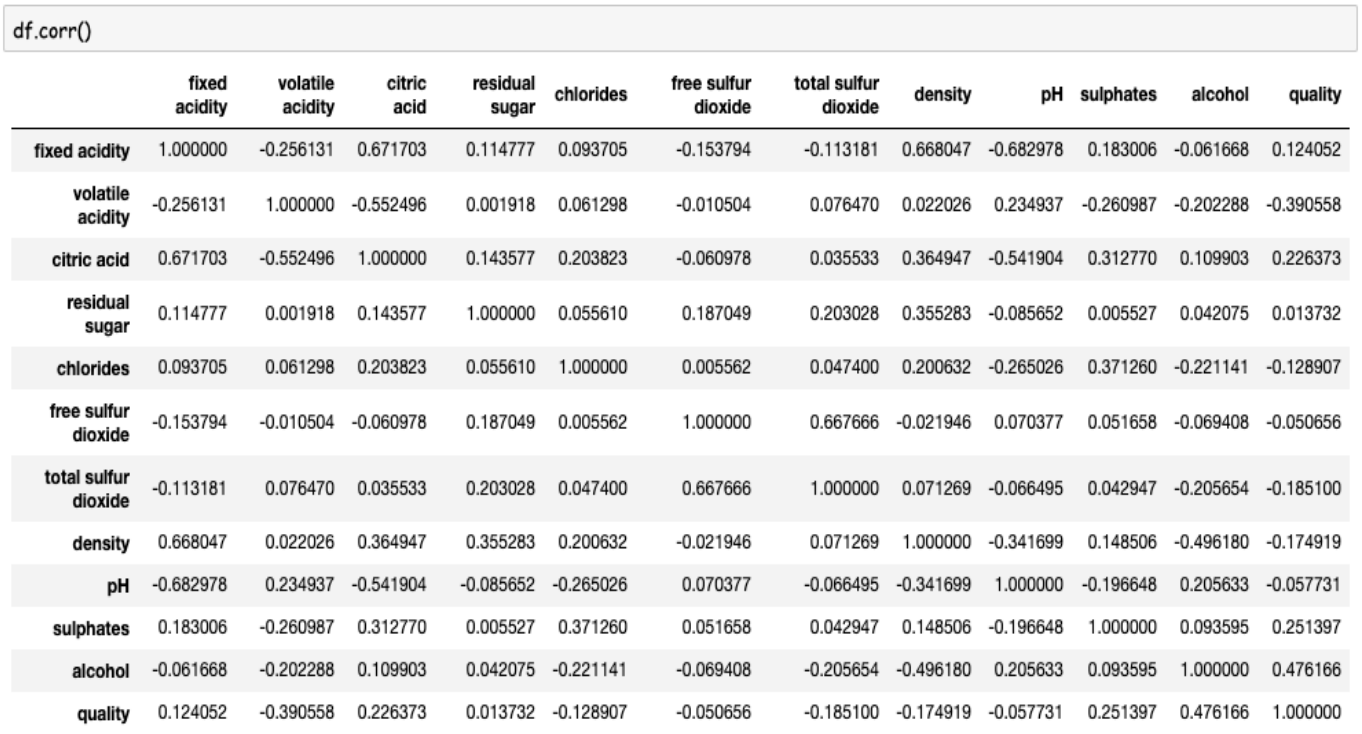

Count
4
2
0
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
sulphates


Count
0.6
0.4
0.2
0.0
8
9
10
11
12
13
14
15
16
alcohol


Count
2
1
0
3
4
5
6
7
8
quality

From the distributions above, we can see that each feature is measured using a different scale. Therefore, the means and standard deviations of each feature varies significantly from each other.

Following data frame tells us the correlations between the features of wine. We wanted to look at the correlation in order to understand whether there is any multicollinearity between some features of the models.



```
df.corr()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
fixed acidity	1.000000	-0.256131	0.671703	0.114777	0.093705	-0.153794	-0.113181	0.668047	-0.682978	0.183006	-0.061668	0.124052
volatile acidity	-0.256131	1.000000	-0.552496	0.001918	0.061298	-0.010504	0.076470	0.022026	0.234937	-0.260987	-0.202288	-0.390558
citric acid	0.671703	-0.552496	1.000000	0.143577	0.203823	-0.060978	0.035533	0.364947	-0.541904	0.312770	0.109903	0.226373
residual sugar	0.114777	0.001918	0.143577	1.000000	0.055610	0.187049	0.203028	0.355283	-0.085652	0.005527	0.042075	0.013732
chlorides	0.093705	0.061298	0.203823	0.055610	1.000000	0.005562	0.047400	0.200632	-0.265026	0.371260	-0.221141	-0.128907
free sulfur dioxide	-0.153794	-0.010504	-0.060978	0.187049	0.005562	1.000000	0.667666	-0.021946	0.070377	0.051658	-0.069408	-0.050656
total sulfur dioxide	-0.113181	0.076470	0.035533	0.203028	0.047400	0.667666	1.000000	0.071269	-0.066495	0.042947	-0.205654	-0.185100
density	0.668047	0.022026	0.364947	0.355283	0.200632	-0.021946	0.071269	1.000000	-0.341699	0.148506	-0.496180	-0.174919
pH	-0.682978	0.234937	-0.541904	-0.085652	-0.265026	0.070377	-0.066495	-0.341699	1.000000	-0.196648	0.205633	-0.057731
sulphates	0.183006	-0.260987	0.312770	0.005527	0.371260	0.051658	0.042947	0.148506	-0.196648	1.000000	0.093595	0.251397
alcohol	-0.061668	-0.202288	0.109903	0.042075	-0.221141	-0.069408	-0.205654	-0.496180	0.205633	0.093595	1.000000	0.476166
quality	0.124052	-0.390558	0.226373	0.013732	-0.128907	-0.050656	-0.185100	-0.174919	-0.057731	0.251397	0.476166	1.000000

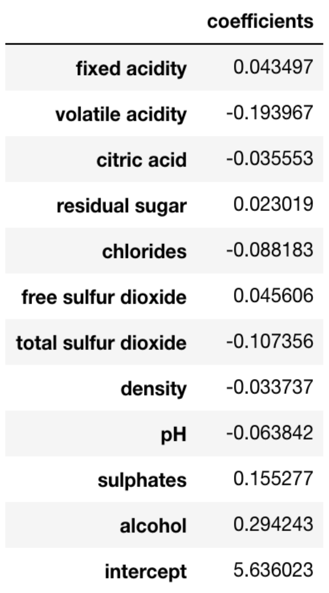
Ordinary Least Squares (OLS)

Ordinary Least Squares requires us to solve the following optimization problem:

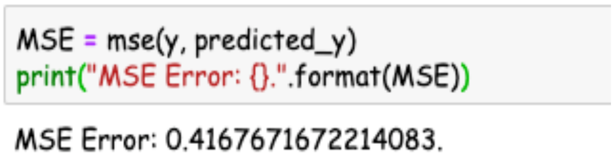
$$\hat{\beta} = \arg\min_{\beta} \|Y - X\beta\|_2^2,$$

We are trying to find the linear model with an intercept, so normally we would append the X data with a column of 1's, but the package we used already does that for us. To set up the OLS model we used the linear model package of Sklearn. We used the mean squared error to calculate the score.

Results



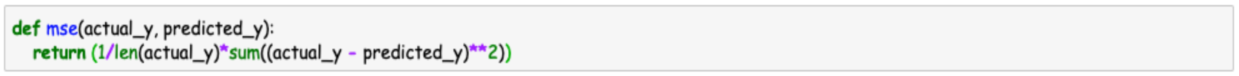
	coefficients
fixed acidity	0.043497
volatile acidity	-0.193967
citric acid	-0.035553
residual sugar	0.023019
chlorides	-0.088183
free sulfur dioxide	0.045606
total sulfur dioxide	-0.107356
density	-0.033737
pH	-0.063842
sulphates	0.155277
alcohol	0.294243
intercept	5.636023



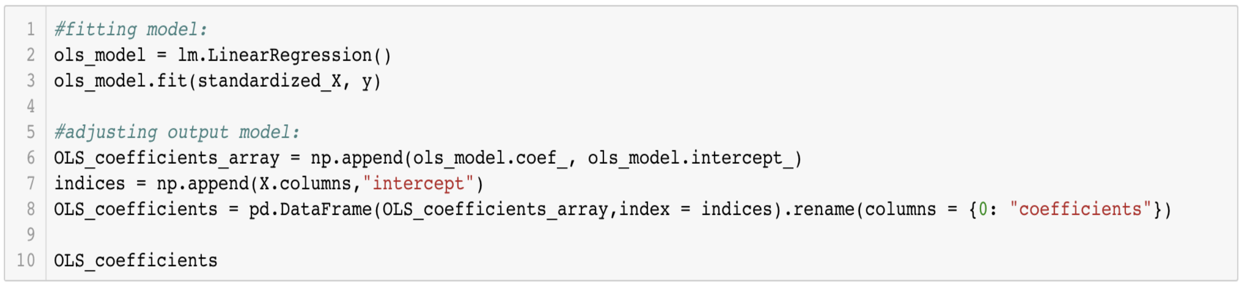
```
MSE = mse(y, predicted_y)
print("MSE Error: {}.".format(MSE))
```

```
MSE Error: 0.4167671672214083.
```

Source Code



```
def mse(actual_y, predicted_y):
    return (1/len(actual_y)*sum((actual_y - predicted_y)**2))
```



```
#fitting model:
ols_model = lm.LinearRegression()
ols_model.fit(standardized_X, y)

#adjusting output model:
OLS_coefficients_array = np.append(ols_model.coef_, ols_model.intercept_)
indices = np.append(X.columns,"intercept")
OLS_coefficients = pd.DataFrame(OLS_coefficients_array,index = indices).rename(columns = {0: "coefficients"})

OLS_coefficients
```

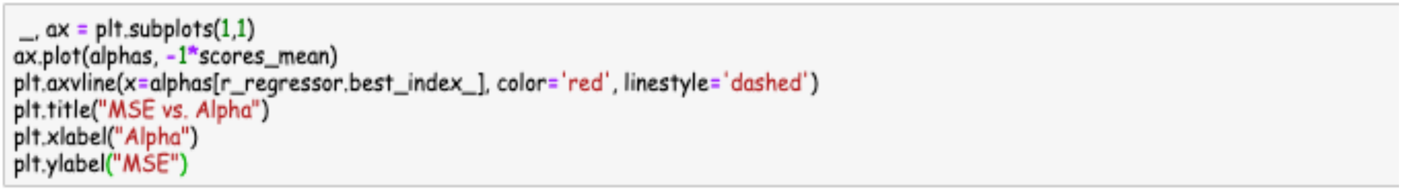
Ridge Regression

Ridge Regression or L2 – regularization is done by solving the following optimization problem with a parameter λ.

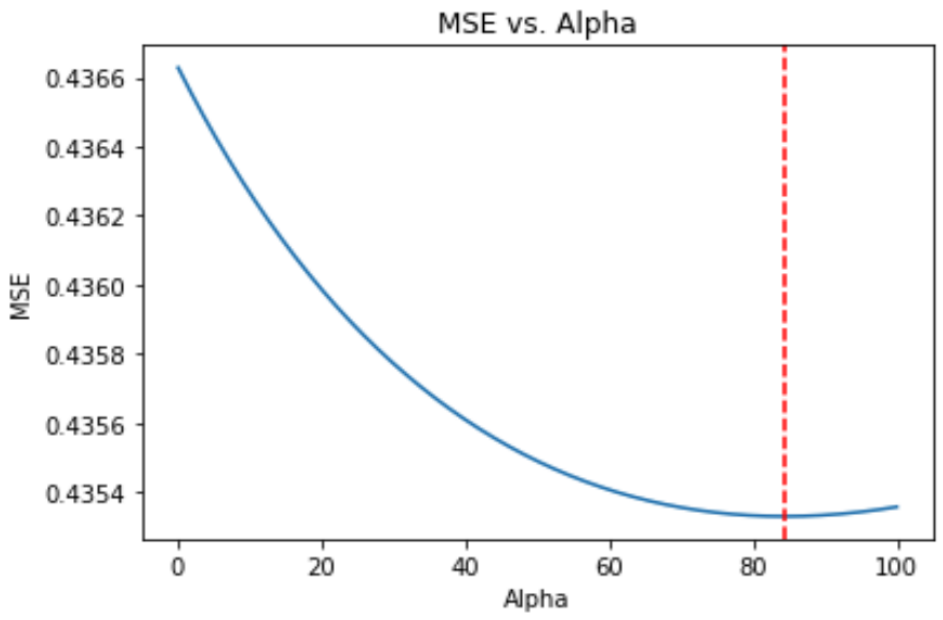
$$\hat{\beta} = argmin||Y - X\beta||_2^2 + \lambda||\beta||_2^2$$

In the equation, lambda is the tuning parameter. In order to find the coefficients that minimize the mean squared error, we tried to optimize the value of lambda. First, we built a ridge regression model using Scikit-learn Ridge package. Later on, we created a range of lambdas starting from 0.1 and ending at 100 with increments of 0.05. In order to find the optimal lambda that would result in the minimum mean squared error, we used Scikit-learn.model_selection's GridSearchCV package to generate a 5 fold cross-validation. We also standardized the X values in order to achieve a more accurate model. After building the model, we used .fit function to fit the model on our X and Y values and selecting the optimal lambda based on the result GridSearchCV function returned, we used a fit command to fit the model to our data. The function outputted an array of length 1998 of cross-validation errors for each value of lambda. We then used this to plot cross-validation error vs. value of tuning parameter. This gave us an optimal tuning parameter choice of 84.35. Please refer to the plot below.

Code for the plot:



```
_, ax = plt.subplots(1,1)
ax.plot(alphas, -1*scores_mean)
plt.axvline(x=alphas[r_regressor.best_index_], color='red', linestyle='dashed')
plt.title("MSE vs. Alpha")
plt.xlabel("Alpha")
plt.ylabel("MSE")
```



Using the optimal value of parameter lambda, which is 84.35, we get the lowest MSE of 0.4353. The coefficients selected by the Ridge Regressor with the optimal lambda are as follows:

	coefficients
fixed acidity	0.053098
volatile acidity	-0.181537
citric acid	-0.015915
residual sugar	0.026914
chlorides	-0.085085
free sulfur dioxide	0.038459
total sulfur dioxide	-0.101091
density	-0.053216
pH	-0.047636
sulphates	0.150007
alcohol	0.269769
intercept	5.636023

Source Code:

```
ridge_model = Ridge(alpha= alphas[r_regressor.best_index_])
```

```
ridge_model.fit(standardized_X, y)
```

```
Ridge(alpha=84.35000000000002, copy_X=True, fit_intercept=True, max_iter=None,
   normalize=False, random_state=None, solver='auto', tol=0.001)
```

```
Ridge_coefficients_array = np.append(ridge_model.coef_, ridge_model.intercept_)
indices = np.append(X.columns,"intercept")
Ridge_coefficients = pd.DataFrame(Ridge_coefficients_array,index = indices).rename(columns = {0: "coefficients"})
```

```
ridge = Ridge()
```

```
alphas = np.arange(0.1, 100, 0.05)
```

```
parameters = {'alpha': alphas}
```

```
r_regressor = GridSearchCV(ridge, parameters, scoring='neg_mean_squared_error', cv = 5)
```

```
r_regressor.fit(standardized_X, y)
```

```
GridSearchCV(cv=5, error_score='raise',
       estimator=Ridge(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=True, n_jobs=1,
       param_grid={'alpha': array([ 0.1 ,  0.15, ..., 99.9 , 99.95])},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='neg_mean_squared_error', verbose=0)
```

```
r_regressor.best_score_
```

```
-0.43533034290183703
```

```
r_regressor.best_index_
```

```
1685
```

```
alphas[r_regressor.best_index_]
```

```
84.35000000000002
```

```
scores_mean = r_regressor.cv_results_['mean_test_score']
```

```
mse = min(-1*scores_mean)
mse
```

```
0.43533034290183703
```

Lasso Regression

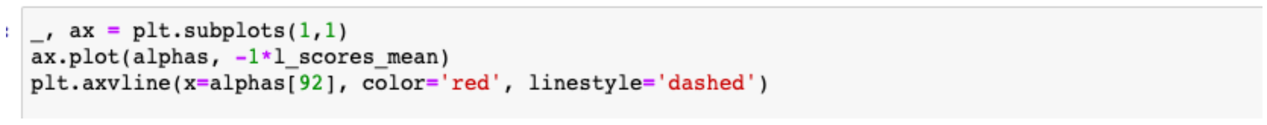
Lasso Regression or L1-regularization is done by solving the following optimization problem with a parameter λ.

$$\hat{\beta} = \arg\min \|Y - X\beta\|_2^2 + \lambda\|\beta\|_1$$

Lasso stands for Least Absolute Shrinkage and Selection Operator. It is very similar to the previously described Ridge Regression, except that Lasso penalizes the sum of the absolute weight values instead of the sum of squared values. Intuitively, the exponents in each procedure correspond to the names - L1 and L2 regularization.

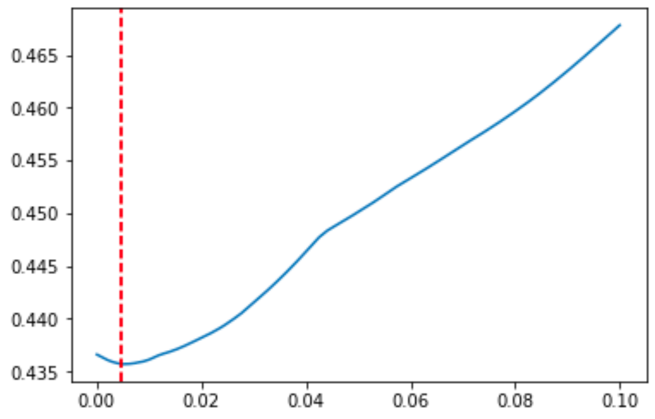
In the same way as before, lambda is the tuning parameter in this equation. In order to find the coefficients that minimize the mean squared error, we optimized the value of lambda. First, we built a lasso regression model using Scikit-learn Ridge package. Later on, we created a range of lambdas starting from 0.0001 and ending at 0.1 with increments of 0.0005. The rest of the procedure is the same as it was for Ridge -- in order to find the optimal lambda that would result in the minimum mean squared error, we used Scikit-learn.model_selection's GridSearchCV package to generate a 5 fold cross-validation. We also standardized the X values in order to achieve a more accurate model. After building the model, we used .fit function to fit the model on our X and Y values and selecting the optimal lambda based on the result GridSearchCV function returned, we used a fit command to fit the model to our data. The function outputted an array of length 1998 of cross-validation errors for each value of lambda. We then used this to plot cross-validation error vs. value of tuning parameter. This gave us an optimal tuning parameter choice of 0.0047. Please refer to the plot below.

Code for the plot:



```
_, ax = plt.subplots(1,1)
ax.plot(alphas, -1*l_scores_mean)
plt.axvline(x=alphas[92], color='red', linestyle='dashed')
```

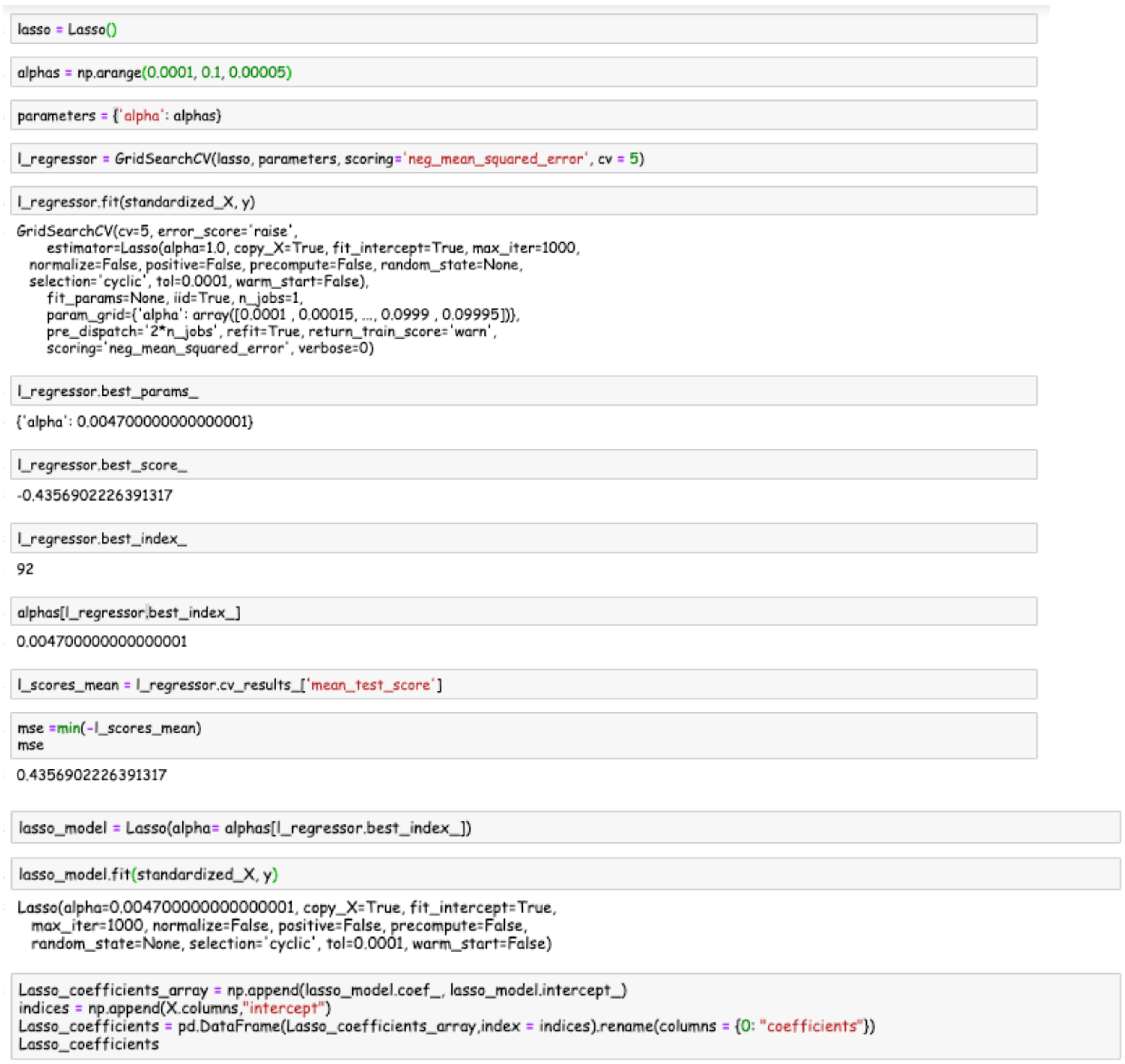
```
<matplotlib.lines.Line2D at 0x7f45786af748>
```



Using the optimal value of parameter lambda, which is 0.0047, we get the lowest MSE of 0.4357. The coefficients selected by the Lasso Regressor with the optimal lambda are as follows:

	coefficients
fixed acidity	0.002904
volatile acidity	-0.187431
citric acid	-0.011654
residual sugar	0.006605
chlorides	-0.085485
free sulfur dioxide	0.035767
total sulfur dioxide	-0.098463
density	-0.000000
pH	-0.068041
sulphates	0.144442
alcohol	0.306785
intercept	5.636023

Source Code:



```
lasso = Lasso()
```

```
alphas = np.arange(0.0001, 0.1, 0.00005)
```

```
parameters = {'alpha': alphas}
```

```
l_regressor = GridSearchCV(lasso, parameters, scoring='neg_mean_squared_error', cv = 5)
```

```
l_regressor.fit(standardized_X, y)
```

```
GridSearchCV(cv=5, error_score='raise',
       estimator=Lasso(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=True, n_jobs=1,
       param_grid={'alpha': array([0.0001 , 0.00015, ..., 0.0999 , 0.09995])},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring='neg_mean_squared_error', verbose=0)
```

```
l_regressor.best_params_
```

```
{'alpha': 0.004700000000000001}
```

```
l_regressor.best_score_
```

```
-0.4356902226391317
```

```
l_regressor.best_index_
```

```
92
```

```
alphas[l_regressor.best_index_]
```

```
0.004700000000000001
```

```
l_scores_mean = l_regressor.cv_results_['mean_test_score']
```

```
mse =min(-l_scores_mean)
mse
```

```
0.4356902226391317
```

```
lasso_model = Lasso(alpha= alphas[l_regressor.best_index_])
```

```
lasso_model.fit(standardized_X, y)
```

```
Lasso(alpha=0.004700000000000001, 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)
```

```
Lasso_coefficients_array = np.append(lasso_model.coef_, lasso_model.intercept_)
indices = np.append(X.columns,"intercept")
Lasso_coefficients = pd.DataFrame(Lasso_coefficients_array,index = indices).rename(columns = {0: "coefficients"})
Lasso_coefficients
```

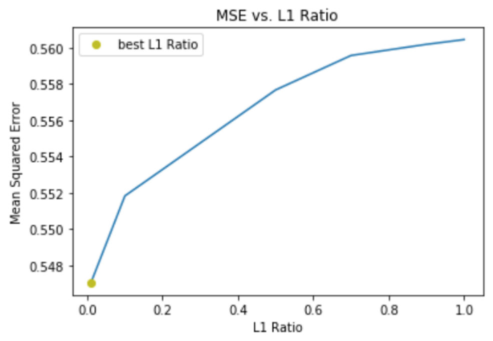
Elastic Net

For Elastic Nets, we have two parameters that we must optimize for - L1 ratio and the Alpha. We choose 40 evenly spaced alpha values from 10^{-10} to 10^{10}. We use ElasticNetCV to cross-validate and get the optimum values.

```
In [48]: en = ElasticNet()

In [54]: aph = np.logspace(-10, 10, 40)      #np.arange(.001, 0.1, .001)

In [64]: L1_vals = [.01, .1, .5, .7, .9, .95, .99, 1]

In [65]: en = lm.ElasticNetCV(alphas = aph, l1_ratio= L1_vals, fit_intercept = True, cv = 5)

In [66]: en.fit(standardized_X, np.ravel(y))

Out[66]: ElasticNetCV(alphas=array([1.00000e-10, 3.25702e-10, 1.06082e-09, 3.45511e-09, 1.12534e-08,
                3.66524e-08, 1.19378e-07, 3.88816e-07, 1.26638e-06, 4.12463e-06,
                1.34340e-05, 4.37548e-05, 1.42510e-04, 4.64159e-04, 1.51178e-03,
                4.92388e-03, 1.60372e-02, 5.22335e-02, 1.70125e-01, 5.54102e-01,
              ..., 8.37678e+06, 2.72833e+07,
                8.88624e+07, 2.89427e+08, 9.42668e+08, 3.07029e+09, 1.00000e+10]),
                copy_X=True, cv=5, eps=0.001, fit_intercept=True,
                l1_ratio=[0.01, 0.1, 0.5, 0.7, 0.9, 0.95, 0.99, 1], max_iter=1000,
                n_alphas=100, n_jobs=None, normalize=False, positive=False,
                precompute='auto', random_state=None, selection='cyclic',
                tol=0.0001, verbose=0)
```

Below is a code snippet and graph for the L1 ratio parameter. It turns out that the L1 ratio is 0.01, meaning our results will be similar to L2 regression.

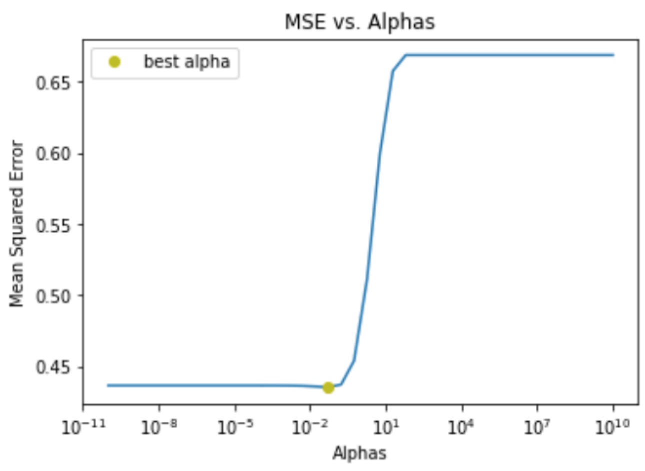
```
In [78]: l1_mse = []

In [92]: for e in en.mse_path_:
             l1_mse.append(np.mean(e))

In [80]: plt.plot(L1_vals, l1_mse)
         plt.ylabel("Mean Squared Error")
         plt.xlabel("L1 Ratio")
         plt.title("MSE vs. L1 Ratio")
         plt.plot(en.l1_ratio_, min(l1_mse), "yo", label = "best L1 Ratio")
         plt.legend();
```



```
In [81]: en.l1_ratio_

Out[81]: 0.01

In [83]: min(l1_mse)

Out[83]: 0.5470344344375718
```

The MSE we get with this ratio is 0.54. Using the best L1 ratio of 0.01, we now look at the most optimum values for Alpha.

```
In [99]:  a_mse = []

In [100]: L1_PATH = en.mse_path_[np.argmin(l1_mse)]

In [101]: for a in L1_PATH:
              a_mse.append(np.mean(a))

In [110]: plt.semilogx(en.alphas_, a_mse)
          plt.ylabel("Mean Squared Error")
          plt.xlabel("Alphas")
          plt.title("MSE vs. Alphas")
          plt.plot(en.alpha_, min(a_mse), "yo", label = "best alpha")
          plt.legend();
```



```
In [107]: en.alpha_

Out[107]: 0.05223345074266832

In [108]: min(a_mse)

Out[108]: 0.4354410093715669
```

We get alpha as 0.05 and L1 ratio as 0.01. Using these parameters we get the lowest MSE of 0.435

Results

Now, we look at the coefficient terms and the intercept that the model with the above-mentioned parameter values provide.

```
en_coefficients_array = np.append(en.coef_, en.intercept_)
indices = np.append(X.columns,"intercept")
en_coefficients = pd.DataFrame(en_coefficients_array,index = indices).rename(columns = {0: "coefficients"})
en_coefficients
```

	coefficients
fixed acidity	0.050472
volatile acidity	-0.181117
citric acid	-0.014110
residual sugar	0.025651
chlorides	-0.084713
free sulfur dioxide	0.037505
total sulfur dioxide	-0.100176
density	-0.050883
pH	-0.047515
sulphates	0.149181
alcohol	0.270673
intercept	5.636023

Final Comparisons

OLS

	coefficients
fixed acidity	0.043497
volatile acidity	-0.193967
citric acid	-0.035553
residual sugar	0.023019
chlorides	-0.088183
free sulfur dioxide	0.045606
total sulfur dioxide	-0.107356
density	-0.033737
pH	-0.063842
sulphates	0.155277
alcohol	0.294243
intercept	5.636023

RIDGE

	coefficients
fixed acidity	0.053098
volatile acidity	-0.181537
citric acid	-0.015915
residual sugar	0.026914
chlorides	-0.085085
free sulfur dioxide	0.038459
total sulfur dioxide	-0.101091
density	-0.053216
pH	-0.047636
sulphates	0.150007
alcohol	0.269769
intercept	5.636023

LASSO

	coefficients
fixed acidity	0.002904
volatile acidity	-0.187431
citric acid	-0.011654
residual sugar	0.006605
chlorides	-0.085485
free sulfur dioxide	0.035767
total sulfur dioxide	-0.098463
density	-0.000000
pH	-0.068041
sulphates	0.144442
alcohol	0.306785
intercept	5.636023

ELASTIC NET

	coefficients
fixed acidity	0.050472
volatile acidity	-0.181117
citric acid	-0.014110
residual sugar	0.025651
chlorides	-0.084713
free sulfur dioxide	0.037505
total sulfur dioxide	-0.100176
density	-0.050883
pH	-0.047515
sulphates	0.149181
alcohol	0.270673
intercept	5.636023

PART 2: Sales Channel Classification

Model Background

The data for this chapter was obtained from the publication, N. Abreu, Analise do perl do cliente Recheio e Desenvolvimento de um sistema promocional, Mestrado em Marketing, ISCTE-IUL, Lisbon, 2011. The data shows the sales amount of a variety of products, the channel and the region of each retailer. We are constructing various SVMs with different types of kernels in order to predict the sales channel. In this project we are applying three different methods: linear SVM, SVM with Polynomial kernel, and SVM with Gaussian kernel in order to classify data. We are cross-validating parameters for each SVM so that we choose the best parameters that would return models that maximize accuracy or in other words, minimizes cross-validation error.

Linear SVM

We start the problem first by importing the relevant libraries.

```
from sklearn import svm, grid_search
import numpy as np
from sklearn.model_selection import cross_val_score
from sklearn import datasets, svm
```

Later on, we create two different data frames, one for the design matrix (X) and one for the response variable (Y).

```
X = df_svm.iloc[:,1:]
y = df_svm.iloc[:,:1]
```

```
X.head()
```

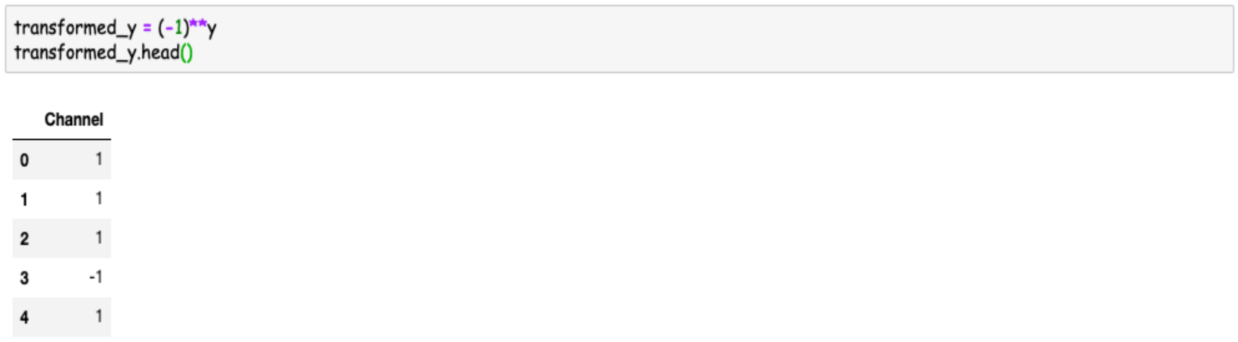
	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	3	12669	9656	7561	214	2674	1338
1	3	7057	9810	9568	1762	3293	1776
2	3	6353	8808	7684	2405	3516	7844
3	3	13265	1196	4221	6404	507	1788
4	3	22615	5410	7198	3915	1777	5185

```
y.head()
```

	Channel
0	2
1	2
2	2
3	1
4	2

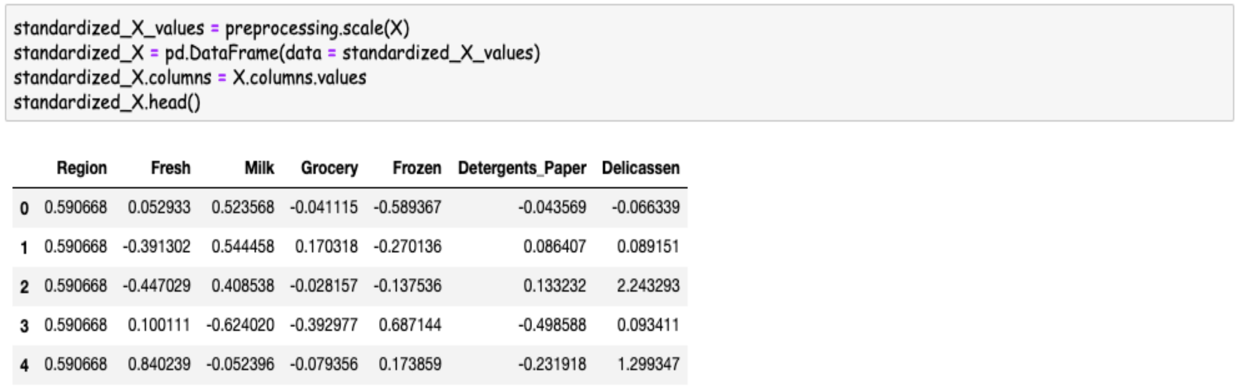
We transform the response variable Y so that the values are no longer composed of 1 and 2s but instead composed of -1 and 1s. In order to convert the values into -1 and +1 for Ys, the transformation we apply is $(-1)^{\text{Channel Label}}$.

Also, the values in the feature/design matrix (X) were standardized using Sklearn.preprocessing.scale package.



```
transformed_y = (-1)**y
transformed_y.head()
```

	Channel
0	1
1	1
2	1
3	-1
4	1



```
standardized_X_values = preprocessing.scale(X)
standardized_X = pd.DataFrame(data = standardized_X_values)
standardized_X.columns = X.columns.values
standardized_X.head()
```

	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	0.590668	0.052933	0.523568	-0.041115	-0.589367	-0.043569	-0.066339
1	0.590668	-0.391302	0.544458	0.170318	-0.270136	0.086407	0.089151
2	0.590668	-0.447029	0.408538	-0.028157	-0.137536	0.133232	2.243293
3	0.590668	0.100111	-0.624020	-0.392977	0.687144	-0.498588	0.093411
4	0.590668	0.840239	-0.052396	-0.079356	0.173859	-0.231918	1.299347

For Linear SVM, we have to optimize for one tuning parameter - the penalty parameter for the error term, C. We set a range of C values by using np.logspace(-10,0,10) which returns numbers starting from 10^{-10} and stopping at 10^{0} spaced evenly on a log scale. We then employ the 10 fold cross validation over different values of C to separately determine the optimum parameter, which maximizes the accuracy or minimizes the cross-validation error.

```
svc = svm.SVC(kernel='linear')
C_s = np.logspace(-10, 0, 10)

scores = list()
#scores_std = list()
for C in C_s:
    svc.C = C
    cv_scores = cross_val_score(svc, standardized_X, np.ravel(y), cv=10, n_jobs=-1)
    scores.append(np.mean(cv_scores))
```

```
scores
```

```
[0.677307963354475,
 0.677307963354475,
 0.677307963354475,
 0.677307963354475,
 0.677307963354475,
 0.677307963354475,
 0.6795806906272024,
 0.8318287526427062,
 0.8999741602067184,
 0.9022515856236787]
```

```
errors = []
for i in range(len(scores)):
    errors.append(1-scores[i])
errors
```

```
[0.32269203664552504,
 0.32269203664552504,
 0.32269203664552504,
 0.32269203664552504,
 0.32269203664552504,
 0.32269203664552504,
 0.3204193093727976,
 0.16817124735729383,
 0.10002583979328161,
 0.09774841437632131]
```

The following command gives the optimum C which maximizes the accuracy or minimum accuracy error of the model. The maximum accuracy is 0.90225158 and minimum accuracy error is 0.09774842.

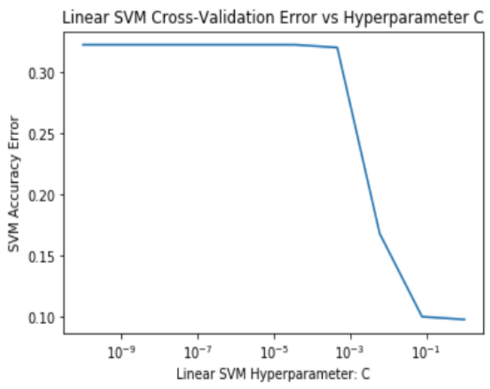
```
best_c = Cs[np.argmax(scores)]
best_c
```

```
1.0
```

From above, we can see that the optimal C value is 1. Also, the following plot shows how setting different values of C changes the cross-validation error.

```
_, ax = plt.subplots(1,1)
ax.semilogx(C_s, errors)
plt.ylabel("SVM Accuracy Error")
plt.xlabel("Linear SVM Hyperparameter: C")
plt.title("Linear SVM Cross-Validation Error vs Hyperparameter C")
```

```
Text(0.5, 1.0, 'Linear SVM Cross-Validation Error vs Hyperparameter C')
```



Using the optimal value of C, which is 1, we set the model and use .fit command to fit the model to our data. The coefficients and the intercept of the model that has the minimum accuracy error are as follows:

```
l_svm = svm.SVC(kernel='linear', C= 1)
model = l_svm.fit(standardized_X, np.ravel(y))
```

```
linearsvm_coefficients_array = np.append(model.coef_, model.intercept_)
indices = np.append(standardized_X.columns,"intercept")
linearsvm_coefficients = pd.DataFrame(linearsvm_coefficients_array,index = indices).rename(columns = {0: "coefficier
linearsvm_coefficients
```

	coefficients
Region	0.236901
Fresh	0.035426
Milk	0.346780
Grocery	0.512052
Frozen	-0.281653
Detergents_Paper	2.556616
Delicassen	-0.220582
intercept	0.023486

SVM with Polynomial Kernel

Importing the relevant libraries

In [1]:
```
import sklearn as skl
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import sklearn.model_selection
from sklearn.model_selection import cross_val_score
import sklearn.svm as svm
from sklearn import preprocessing
from sklearn.metrics import mean_squared_error as mse
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

Creating a Dataframe and Checking for Null Values

In [2]:
```
df = pd.read_csv('wholesale-customers.csv', sep=',')
df .head()
```

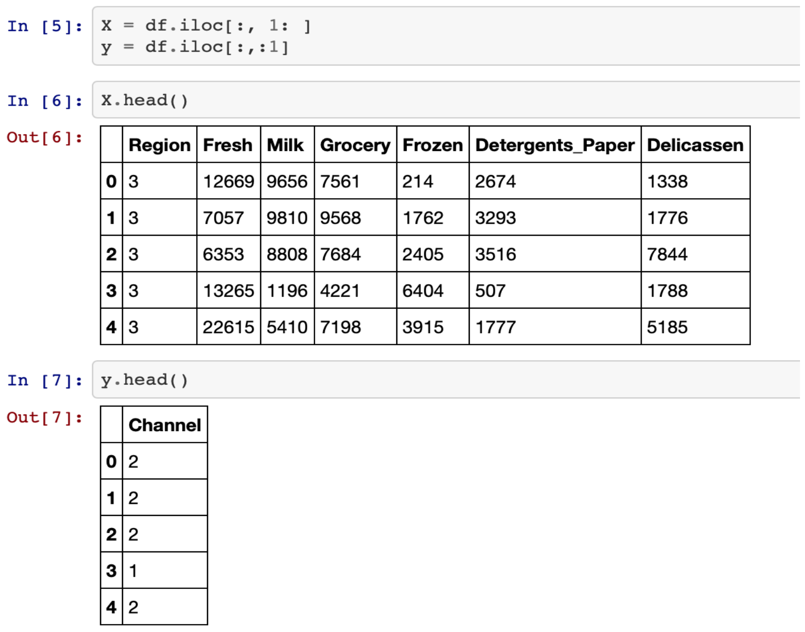
Out[2]:

	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	2	3	12669	9656	7561	214	2674	1338
1	2	3	7057	9810	9568	1762	3293	1776
2	2	3	6353	8808	7684	2405	3516	7844
3	1	3	13265	1196	4221	6404	507	1788
4	2	3	22615	5410	7198	3915	1777	5185

In [3]:
```
print("Percentage of Null values in each column: ")
display((df.isnull().sum()/len(df))*100)
```

```
Percentage of Null values in each column:
Channel             0.0
Region              0.0
Fresh               0.0
Milk                0.0
Grocery             0.0
Frozen              0.0
Detergents_Paper    0.0
Delicassen          0.0
dtype: float64
```

Creating Dataframe for Predictor and Response Variables



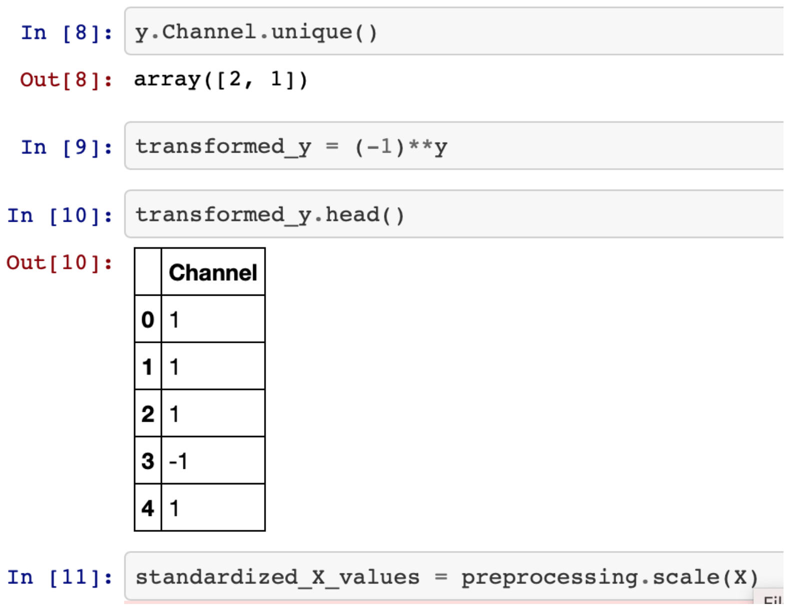
```
In [5]: X = df.iloc[:, 1: ]
        y = df.iloc[:,:1]

In [6]: X.head()

Out[6]:
```

	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	3	12669	9656	7561	214	2674	1338
1	3	7057	9810	9568	1762	3293	1776
2	3	6353	8808	7684	2405	3516	7844
3	3	13265	1196	4221	6404	507	1788
4	3	22615	5410	7198	3915	1777	5185

```
In [7]: y.head()

Out[7]:
```

	Channel
0	2
1	2
2	2
3	1
4	2

Processing the data to fit the SVM paradigm. This means that the labels for the Response variable must be between +1 and -1. The math works out much more cleanly if we do things thing way. Since our Channel labels are 1 and 2, we must transform them to give +1 and -1. The transformation we apply is $(-1)^{\text{Channel Label.}}$ This way the new label for retail becomes 1. We also standardize our results. When we are running SVMs, the estimated weights will update similarly rather than at different rates during the build process. This will give us more accurate results.



```
In [8]: y.Channel.unique()

Out[8]: array([2, 1])

In [9]: transformed_y = (-1)**y

In [10]: transformed_y.head()

Out[10]:
```

	Channel
0	1
1	1
2	1
3	-1
4	1

```
In [11]: standardized_X_values = preprocessing.scale(X)
In [13]: standardized_X_values[:,1][:5]

Out[13]: array([ 0.05293319, -0.39130197, -0.44702926,  0.10011141,  0.84023948])

In [14]: standardized_X = pd.DataFrame(data = standardized_X_values)

In [15]: standardized_X.columns = X.columns.values

In [16]: standardized_X.head()
```

Out[16]:

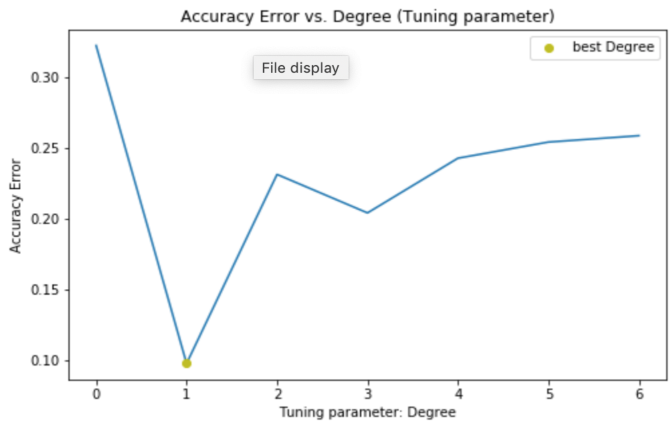
	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	0.590668	0.052933	0.523568	-0.041115	-0.589367	-0.043569	-0.066339
1	0.590668	-0.391302	0.544458	0.170318	-0.270136	0.086407	0.089151
2	0.590668	-0.447029	0.408538	-0.028157	-0.137536	0.133232	2.243293
3	0.590668	0.100111	-0.624020	-0.392977	0.687144	-0.498588	0.093411
4	0.590668	0.840239	-0.052396	-0.079356	0.173859	-0.231918	1.299347

For Polynomial Kernels, we have to optimize for 2 tuning parameters - the penalty parameter for the error term, C, and the degree of the polynomial kernel function, degree. We then employ the 10 fold cross-validation to separately determine the optimum tuning parameters. We also start with default C = 1. We use np.ravel with transformed_y to get a contiguous flattened array for smoother processing.

```
In [24]: degrees = [0,1,2,3,4,5,6]

In [25]: scores = []
         for d in degrees:
             n_scores = cross_val_score(svm.SVC(kernel='poly', C=1, degree = d, gamma = 'scale'), standardi
         zed_X, np.ravel(transformed_y), cv = 10)
             mean_score = np.mean(n_scores)
             scores.append(mean_score)

In [26]: scores

Out[26]: [0.677307963354475,
          0.9021458773784354,
          0.7684402161146346,
          0.7956647874089734,
          0.7569708715057553,
          0.7455543810194973,
          0.7410594315245478]
```

We now find the errors and plot them

```
In [27]: errors = (np.array(1) - scores)

In [28]: errors

Out[28]: array([0.32269204, 0.09785412, 0.23155978, 0.20433521, 0.24302913,
                0.25444562, 0.25894057])

In [29]: best_degree = degrees[np.argmin(errors)]

In [40]: plt.figure(figsize=(8,5))
         plt.plot(degrees, errors)
         plt.plot(best_degree, min(errors), "yo", label = "best Degree")
         plt.title("Accuracy Error vs. Degree (Tuning parameter)")
         plt.xlabel("Tuning parameter: Degree")
         plt.ylabel("Accuracy Error")
         plt.legend();
```

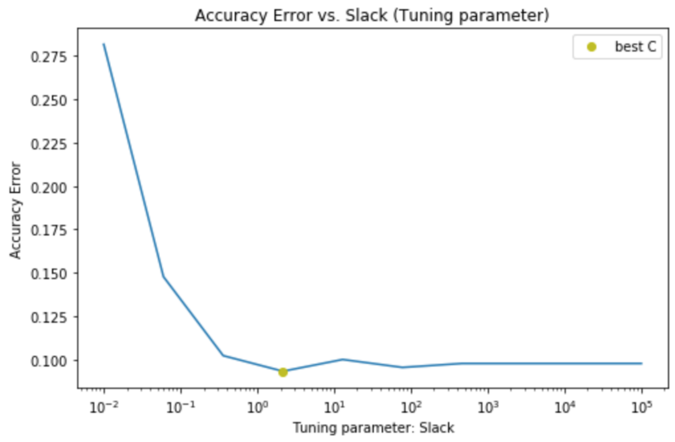


From the above graph, it is clear the best degree is 1. The cross-validation error associated with degree 1 was 0.09785412. We will use this value and find the most optimum penalty parameter for the error term, C. We choose the most optimum C from Cs = np.logspace(-2,5,10).

```
In [36]: s = []
         for c in Cs:
             n_scores = cross_val_score(svm.SVC(kernel='poly', C = c, degree = 1, gamma = 'scale'), standar
         dized_X, np.ravel(transformed_y), cv = 10)
             mean_n_scores = np.mean(n_scores)
             s.append(mean_n_scores)

In [37]: n_errors = (np.array(1) - s)

In [38]: n_errors

Out[38]: array([0.28167489, 0.14776486, 0.10229857, 0.09330632, 0.100074  ,
                0.09552854, 0.09780127, 0.09780127, 0.09780127, 0.09780127])

In [41]: min(n_errors)

Out[41]: 0.09330631900399344

In [42]: best_C = Cs[np.argmin(n_errors)]

In [43]: best_C

Out[43]: 2.1544346900318843
```

We can see that the best C is 2.154 with the lowest cross-validation error of 0.0933. Below is a visual representation of this.

```
In [44]: plt.figure(figsize=(8,5));
         plt.semilogx(Cs, n_errors)
         plt.plot(best_C, min(n_errors), "yo", label = "best C");
         plt.legend()
         plt.title("Accuracy Error vs. Slack (Tuning parameter)")
         plt.xlabel("Tuning parameter: Slack")
         plt.ylabel("Accuracy Error") ;
```



Results

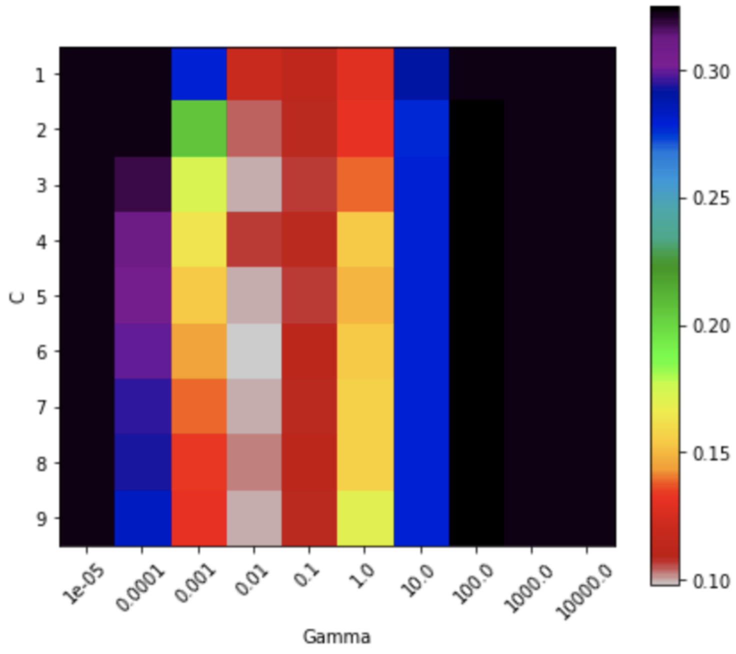
We found that SVM with degree 1, with default C = 1 gave the smallest cross-validation error of 0.09785412.

Using this optimized value of degree = 1, we optimized for C. We found that SVM with degree 1, and C = 2.154 gave us the lowest cross-validation error of 0.0933.

SVM with Gaussian Kernel

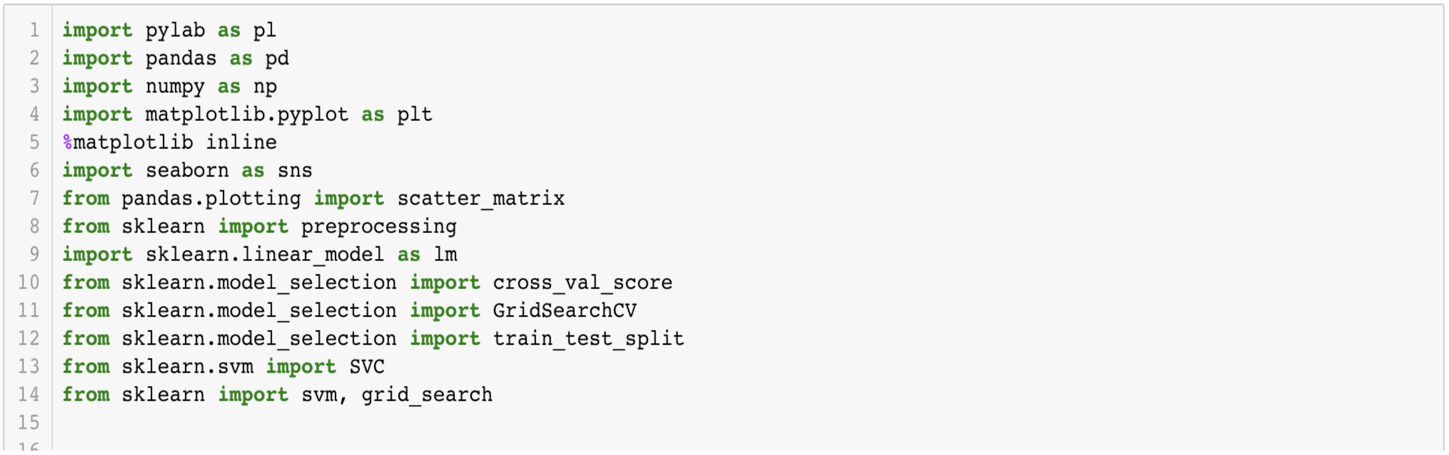
The X data was standardized already in the previous parts. Classifications are 1, and -1 as in the aforementioned processes. SVM with Gaussian kernel has two parameters (C and gamma) and these were tuned for minimum accuracy error using cross-validation by performing a grid search. Initially, we performed cross-validation across a larger range of C values, once we had established that C should be in a range of 10^0 *to* 10^1 we performed another grid search on this interval. The results were displayed as a heatmap as we would multiple lines would have been needed in a 2D graph.

Results:



```
Min Accuracy Error:
0.09772727272727277
{'C': 6, 'gamma': 0.01}
```

Source Code:



```
import pylab as pl
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from pandas.plotting import scatter_matrix
from sklearn import preprocessing
import sklearn.linear_model as lm
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn import svm, grid_search

```

```python
#Cs = 10. ** np.arange(-2,5)
Cs = [1,2,3,4,5,6,7,8,9]
g = np.logspace(-5, 4, 10)

def split_selector(X, y):
    X_train, X_test, y_train, y_test =train_test_split(X, y , test_size=0.3)
cv = split_selector(X_svm,y_svm)

X_train, X_test, y_train, y_test = train_test_split(X_svm, y_svm , test_size=0.3)

p_grid = {'C': Cs ,'gamma' : g}
cross_val = GridSearchCV(SVC(), p_grid, scoring = "accuracy",cv=cv)
cross_val.fit(X_svm, y_svm)
cross_val.best_params_

score = cross_val.grid_scores_
scores = [x[1] for x in score]

scores = 1 - np.array(scores).reshape(len(Cs), len(g))

print('Min Accuracy Error:')
print(np.min(scores))
print(cross_val.best_params_)

pl.figure(figsize=(7, 6))
pl.imshow(scores, cmap=pl.cm.spectral_r)
pl.ylabel('C')
pl.xlabel('Gamma')
pl.xticks(np.arange(len(g)), g, rotation=45)
pl.yticks(np.arange(len(Cs)), Cs)
pl.colorbar()
pl.show()
```

Final Comparisons

Type of SVM	Tuning Parameter: C	Tuning Parameter: Degree	Tuning Parameter: Gamma	Cross Validation Error
Linear Kernel	1.0	-	Scale: 1 / (n_features * X.var())	0.0977
Polynomial Kernel	2.154	1	Scale: 1 / (n_features * X.var())	0.0933
Gaussian Kernel	6.0	-	0.01	0.0977