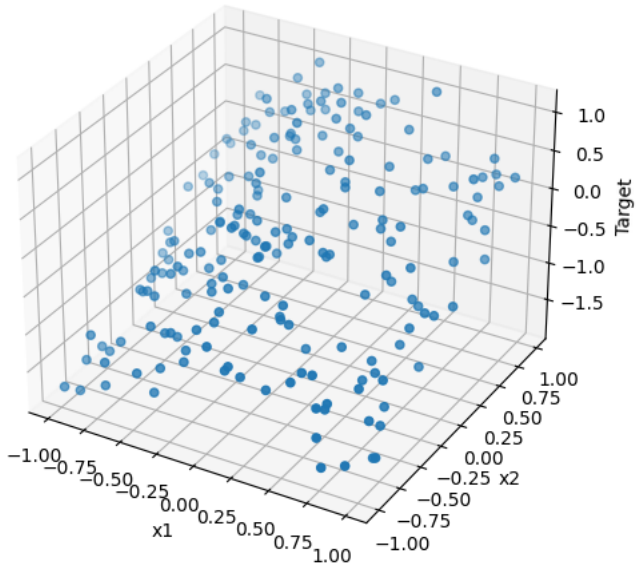
Week 3 Assignment

Efeosa Eguavoen - 17324649

October 26, 2020

1 (i) - id:2–2-2

1.1 a



To generate the graph, I first read in the data and placed it into a dataframe. I then used ax.scatter to generate the 3d graph and added labels using built in functions to label my axes.

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(df['x1'], df['x2'], df['label'])
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('Target')
```

The graph seems to have some sort of curve shape, as the data points towards the front of the graph go up in the centre then down, indicating some sort of quadratic shape roughly.

1.2 b

As the value of C increases the parameter values get smaller and smaller also, eventually becoming 0 when C is large enough.

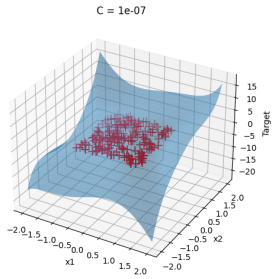
To get the features, I used PolynomialFeatures fitted to the data given to us, to the power of 5. I then placed this into a dataframe called features, with each feature labelled accordingly like in the photo below.

```
p = PolynomialFeatures(5).fit(df[['x1', 'x2']])
   features = pd.DataFrame(p.transform(df[['x1', 'x2']]), columns=p.get_feature_names(df.columns)
```

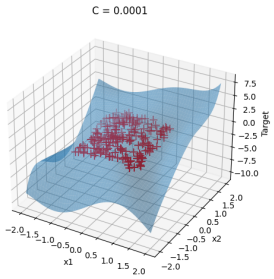
After this I then had a list of C vals between 0 and 1 as 1 was where I found my parameter values to all equal 0, so I used a range of values between 0 and 1 as my C values. From there I used a for loop to iterate through my values of C and created a model for each value of C, fitting the data and added each model to my list of models.

```
models = []
    c_vals = [1e-7, 1e-4, 1e-3, 1e-2, 1e-1, 1]
    for s in c_vals:
        model = Lasso(alpha=s)
        model.fit(features, df['label'])
        models.append((model, s))
```

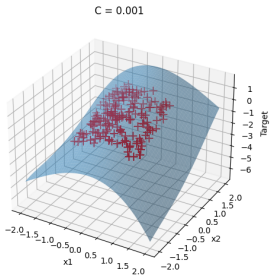
1.3 C



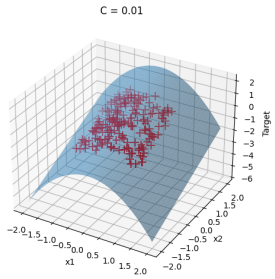
(a) C = 1e-07



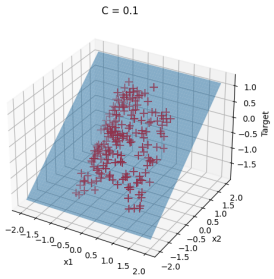
(b) C = 0.0001



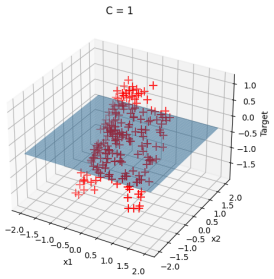
(c) C = 0.001



(d) C = .01



(e) C = .1



(f) C = 1

Red Dots: Training Data, Blue curves: Predictions (Adding a legend to a 3d plot isn't natively supported). From the above plots we can see that changing the value of C influences hugely our predictions. This is due to the fact that Lasso regression uses a L1 penalty that sets more and more of our parameters to 0 as we increase the value of C. This can be seen as the shapes of our graph change a lot, for example when C = 1e-07 the shape of the predictions is quite wild and fits the data closely, while when C = 1 it doesn't fit the data at all, as all our parameters are set to 0.
To generate these graphs, I created a list of points for x1 and x2, then I used np.meshgrid to generate a grid of points. From there I flattened the arrays to 1D and stacked them using np.ravel and np.vstack respectively. I got the transpose of this array which then acted as my base array to generate features from using Polynomial features.

```
x1vals = y1vals = np.array(np.linspace(-2, 2))
x, y = np.meshgrid(x1vals, y1vals)
positions = np.vstack([x.ravel(), y.ravel()])
xtest = (np.array(positions)).T
pdata = pd.DataFrame(xtest, columns=['x1', 'x2'])
p1 = PolynomialFeatures(5).fit(pdata[['x1', 'x2']])
mesh_features = pd.DataFrame(p1.transform(pdata[['x1', 'x2']]), columns=p.get_feature_names(pdata.
```

From here I iterated over my list of models and made predictions on the dataframe of data points and plotted each graph of predictions vs the training data.

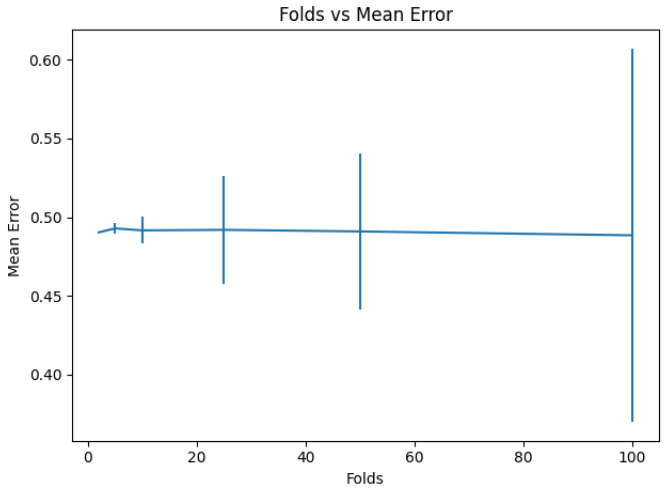
```
    for i in models:
        pred = i[0].predict(mesh_features)
        pred = pred.reshape(x.shape)
```

1.4 D

Overfitting refers to a model that matches the training data too closely, to the point where it's capturing all the noise and randomness in the graph. Underfitting refers to a model that doesn't capture the trends of the data at all and can't make accurate predictions of the data whatsoever. The parameter C enables us to manage between overfitting and underfitting our data by setting more of our parameter values to 0 in terms of Lasso Regression. This enables us to ignore certain parameters that aren't as important in capturing the general trend in the data. We can see this in the graph where C = 0.001 compared to when C = 0.0001. More of our parameters have been set to 0 or values very close to 0, which in turn enables us to capture the general shape of the data without overfitting like when C = 0.0001. When this penalty is too aggressive though we can get an underfit, like when C = .1 vs C = 0.01

2 (ii)

2.1 A



When selecting the number of folds, there's a trade off between the amount of computational time we can use and the level of bias we're willing to accept. Having higher values of K enables us to have less bias overall as the training set better represents the data and the variance increases as the training sets become more similar. But having too high a value of K becomes an issue as the test set might become too small which might not properly represent the data. For this dataset, I think using a value of K = 10 is appropriate as the data set isn't that large and using larger values of K reduces or test set too much to represent the data well. Using 5 increases the mean error slightly also.
To get the above graph, I set up a list of K values and a list for mean errors and their associated variances. I then iterated over this list of K values to generate different splits.

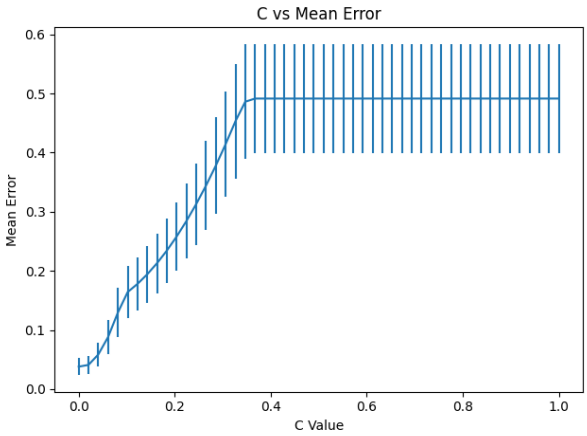
```
k_vals = [2, 5, 10, 25, 50, 100]
mean_list = []
variance_list = []
    for k in k_vals:
        error_list = []
        kf = KFold(n_splits=k)
```

I then split the training data into 2 sets, a training set and a test set. Following this I trained the model and got the mean squared error and appended this to the error list. Once I had all the errors, I got the mean and variance and appended them to their respective lists. I then plotted after.

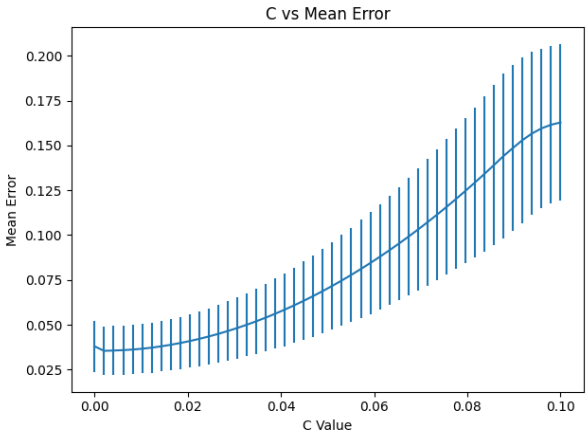
```
for train, test in kf.split(features):
   x_train, x_test = features.loc[train], features.loc[test]
   y_train, y_test = df.loc[train, 'label'], df.loc[test, 'label']
   model = Lasso(alpha=1)
   model.fit(x_train.values, y_train.values)
   pred = model.predict(x_test)
   error_list.append(mean_squared_error(y_test.values, pred))
```

```
error_list = np.array(error_list)
mean = error_list.mean()
mean_list.append(mean)
var = error_list.var()
variance_list.append(var)
```

2.2 B



(a) $0 < C < 1$



(b) $0 < C < 0.1$

I used 10-fold Cross validation for the above plots as I established it to work slightly better than 5 fold in an earlier section. For choosing values of C, I first started with mapping values between 0 and 1 like in the plot(a) as this spanned all values of C that kept my parameters non 0. From there I then reduced the range to scan values of C as I could see that towards the left of graph(a) was where my minimum was. From there I generated graph(b) which scans values in a much smaller range to find the optimum value of C.
To generate the graphs, I used the same code as for using different values of K, I just iterated over my list of C values instead of iterating through a list of values of K.

```
c_vals = np.linspace(0.00000001, 0.1)
    mean_list = []
    std_list = []
    kf = KFold(n_splits=10)
    for i in c_vals:
```

2.3 C

Based on the above graph, I'd recommend a value of C around 0.002 as the values of C seem to take a slight dip at the start around C = 0.02 then just increases from there. At this point, my mean square error is at it's lowest meaning the accuracy of my predictions is at it's best here as the parameters of my algorithm are the most optimised they can be and are as accurate to the trends in the data as possible.

2.4 D