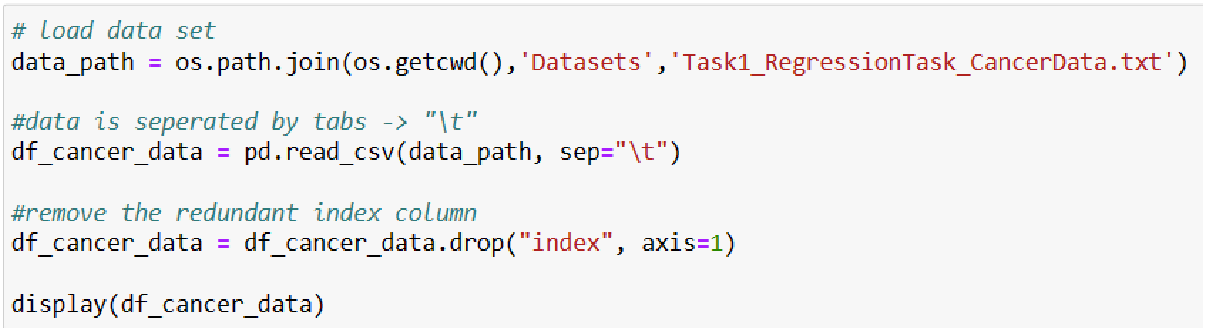
CS4049 Assessment 1 Report

Ben Temming

Contents

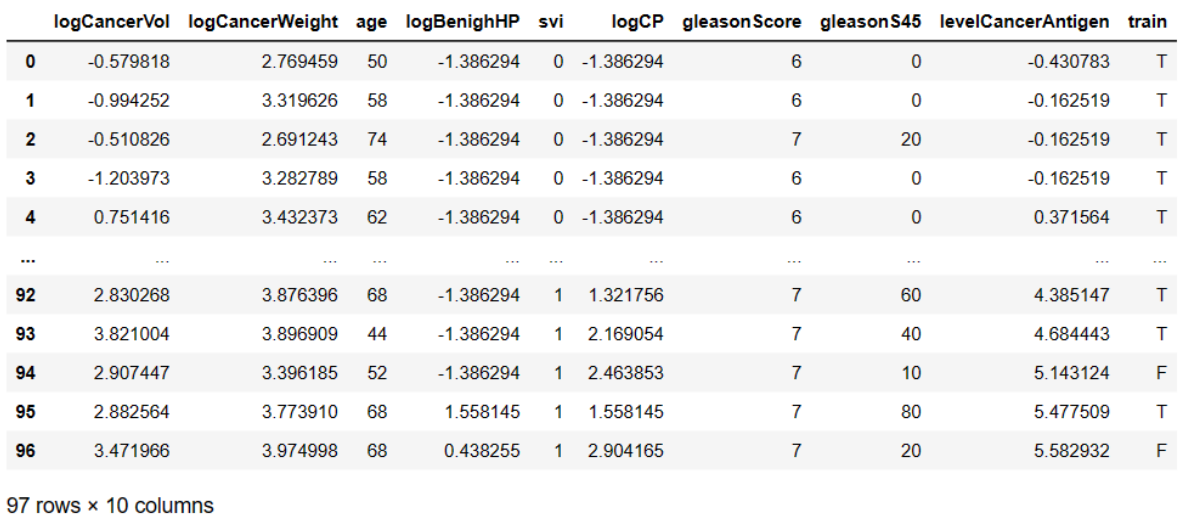
1. The Data

The dataset contains several clinical measures and the level of cancer antigen of patients who were due to receive radical treatment. The aim of this experiment is to explore the relationship between the clinical measures and cancer antigen levels and develop a predictive model for a patient's cancer antigen level. Figure *1* shows how the data can be loaded and Figure *2* shows the loaded dataframe.



```python
# load data set
data_path = os.path.join(os.getcwd(),'Datasets','Task1_RegressionTask_CancerData.txt')

#data is seperated by tabs -> "\t"
df_cancer_data = pd.read_csv(data_path, sep="\t")

#remove the redundant index column
df_cancer_data = df_cancer_data.drop("index", axis=1)

display(df_cancer_data)
```

Figure 1: Code for loading the cancer data.

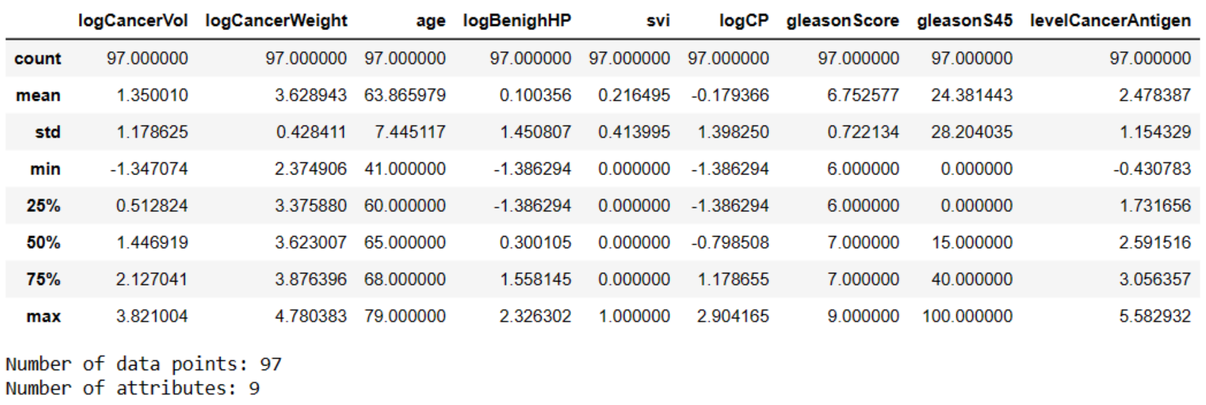


	logCancerVol	logCancerWeight	age	logBenighHP	svi	logCP	gleasonScore	gleasonS45	levelCancerAntigen	train
0	-0.579818	2.769459	50	-1.386294	0	-1.386294	6	0	-0.430783	T
1	-0.994252	3.319626	58	-1.386294	0	-1.386294	6	0	-0.162519	T
2	-0.510826	2.691243	74	-1.386294	0	-1.386294	7	20	-0.162519	T
3	-1.203973	3.282789	58	-1.386294	0	-1.386294	6	0	-0.162519	T
4	0.751416	3.432373	62	-1.386294	0	-1.386294	6	0	0.371564	T
...	...	...	...	...	...	...	...	...	...	...
92	2.830268	3.876396	68	-1.386294	1	1.321756	7	60	4.385147	T
93	3.821004	3.896909	44	-1.386294	1	2.169054	7	40	4.684443	T
94	2.907447	3.396185	52	-1.386294	1	2.463853	7	10	5.143124	F
95	2.882564	3.773910	68	1.558145	1	1.558145	7	80	5.477509	T
96	3.471966	3.974998	68	0.438255	1	2.904165	7	20	5.582932	F

97 rows × 10 columns

Figure 2: Overview of cancer data.

A table with essential statistical properties of the data can be generated using the built-in .describe() function from the Pandas library.



	logCancerVol	logCancerWeight	age	logBenighHP	svi	logCP	gleasonScore	gleasonS45	levelCancerAntigen
count	97.000000	97.000000	97.000000	97.000000	97.000000	97.000000	97.000000	97.000000	97.000000
mean	1.350010	3.628943	63.865979	0.100356	0.216495	-0.179366	6.752577	24.381443	2.478387
std	1.178625	0.428411	7.445117	1.450807	0.413995	1.398250	0.722134	28.204035	1.154329
min	-1.347074	2.374906	41.000000	-1.386294	0.000000	-1.386294	6.000000	0.000000	-0.430783
25%	0.512824	3.375880	60.000000	-1.386294	0.000000	-1.386294	6.000000	0.000000	1.731656
50%	1.446919	3.623007	65.000000	0.300105	0.000000	-0.798508	7.000000	15.000000	2.591516
75%	2.127041	3.876396	68.000000	1.558145	0.000000	1.178655	7.000000	40.000000	3.056357
max	3.821004	4.780383	79.000000	2.326302	1.000000	2.904165	9.000000	100.000000	5.582932

```
Number of data points: 97
Number of attributes: 9
```

Figure 3: Statistical summary of cancer data.

Creating a pair plot, Figure *4*, also helps to get a better understanding of the relationship between the features and the target variable.

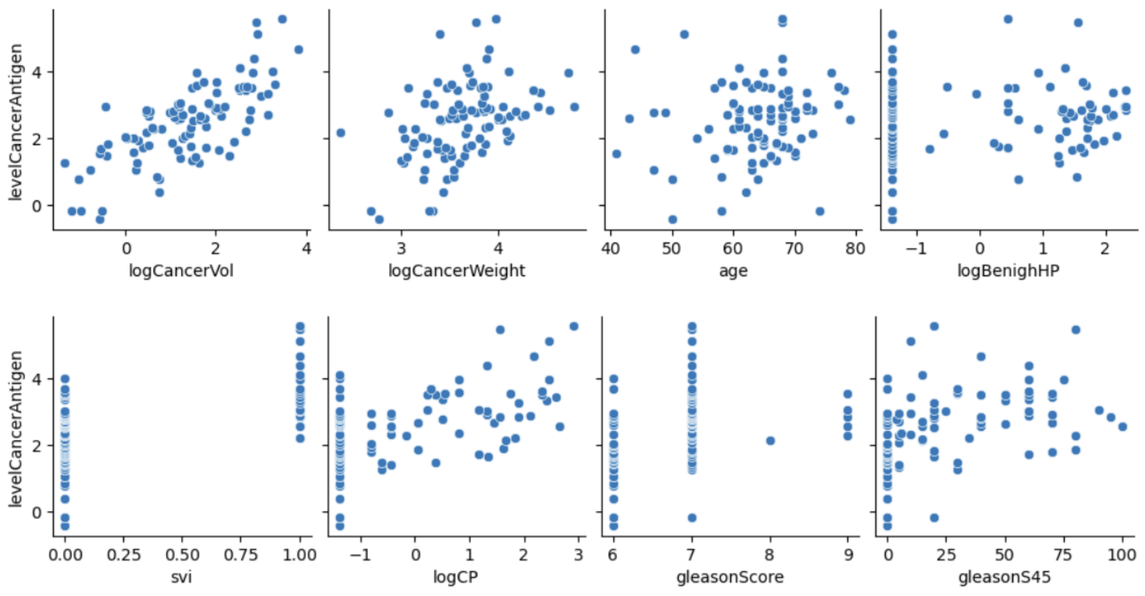
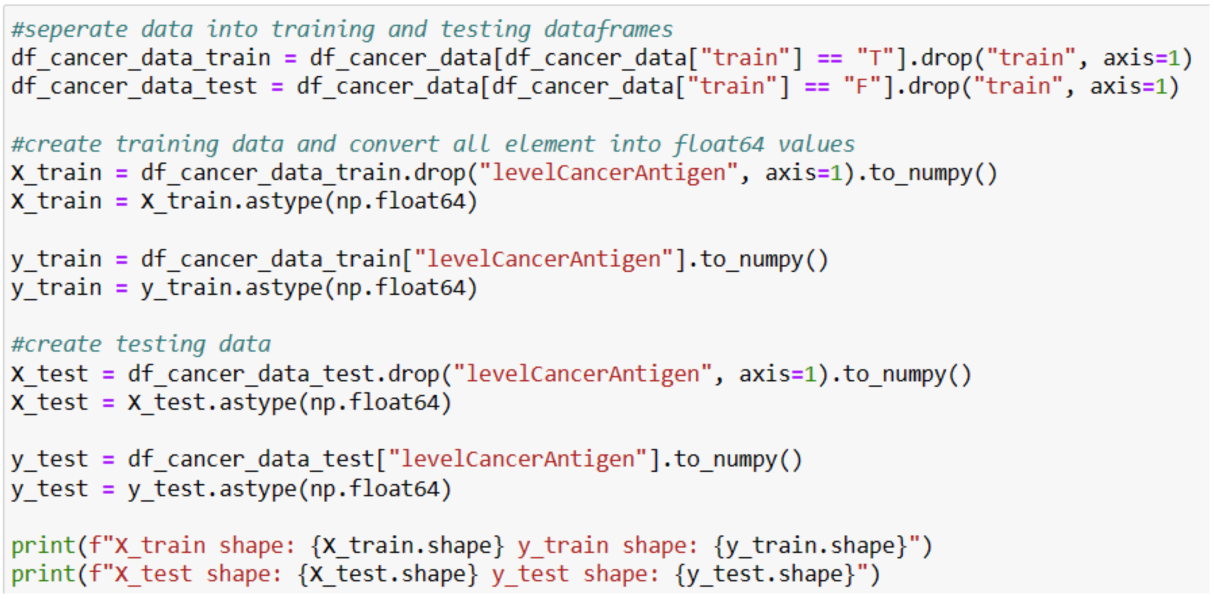


Figure 4: Pair plot of levelCancerAntigen against each feature.

In summary, as indicated in Figure *3*, the dataset comprises of 97 data points, each containing 9 attributes. 8 of these attributes serve as features for predicting the cancer antigen level in patients. The dataset ranges from a minimum cancer antigen level of 0.43 to a maximum of 5.58 and an average level of 2.478. Figure *4* reveals a distinct relationship between cancer volume, cancer weight, CP and the level of cancer antigen.

2. Data Preprocessing

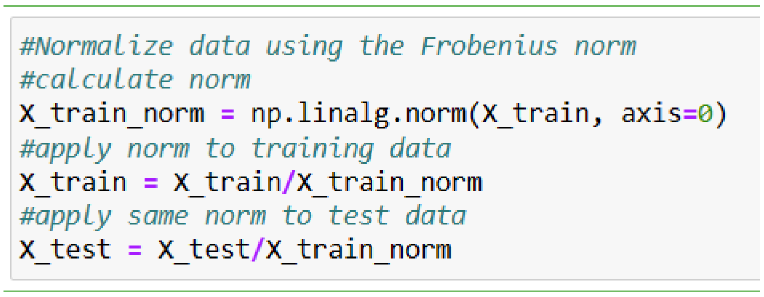
To prepare the data for training the dataset must be split into training and testing data. This is implemented in Figure *5*, using the “train” label of the dataset.



```python
#seperate data into training and testing dataframes
df_cancer_data_train = df_cancer_data[df_cancer_data["train"] == "T"].drop("train", axis=1)
df_cancer_data_test = df_cancer_data[df_cancer_data["train"] == "F"].drop("train", axis=1)

#create training data and convert all element into float64 values
X_train = df_cancer_data_train.drop("levelCancerAntigen", axis=1).to_numpy()
X_train = X_train.astype(np.float64)

y_train = df_cancer_data_train["levelCancerAntigen"].to_numpy()
y_train = y_train.astype(np.float64)

#create testing data
X_test = df_cancer_data_test.drop("levelCancerAntigen", axis=1).to_numpy()
X_test = X_test.astype(np.float64)

y_test = df_cancer_data_test["levelCancerAntigen"].to_numpy()
y_test = y_test.astype(np.float64)

print(f"X_train shape: {X_train.shape} y_train shape: {y_train.shape}")
print(f"X_test shape: {X_test.shape} y_test shape: {y_test.shape}")
```

Figure 5: Splitting data into training data and testing data.

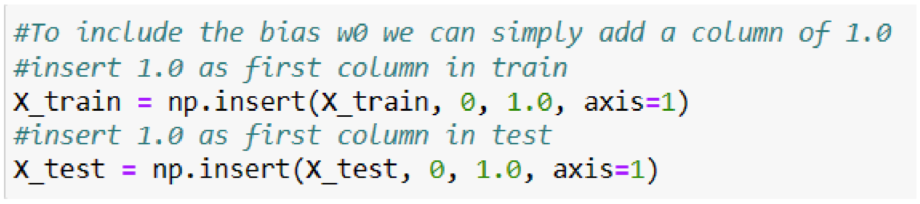
To achieve faster convergence when training the model and protect against large outliner values the data is normalized. Figure 6 shows normalization with the Frobenius norm using the numpy .norm() function. It is important to note that both the training and testing data must be normalised using the same norm.



```
#Normalize data using the Frobenius norm
#calculate norm
X_train_norm = np.linalg.norm(X_train, axis=0)
#apply norm to training data
X_train = X_train/X_train_norm
#apply same norm to test data
X_test = X_test/X_train_norm
```

Figure 6: Normalising training and testing data.

Finally, an additional column of 1s is added to the data, serving as the bias term during training. This simplifies the process of computing derivatives for each weight.



```
#To include the bias w0 we can simply add a column of 1.0
#insert 1.0 as first column in train
X_train = np.insert(X_train, 0, 1.0, axis=1)
#insert 1.0 as first column in test
X_test = np.insert(X_test, 0, 1.0, axis=1)
```

Figure 7: Adding an additional column of 1s to data.

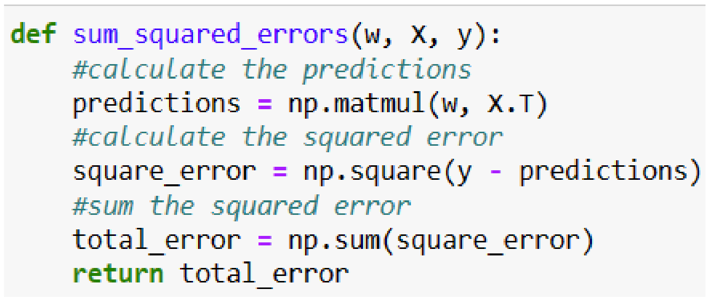
3. Model Implementation

3.1. Least Square Estimation (LSE) Regression Implementation

The first step of implementing LSE regression is defining the loss function. The sum of squared estimate of errors (SSE) error function is defined as:

$$SSE = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \text{ , } where\ \hat{y}_i = wx_i$$

Figure 8 shows the Python implementation of this formula.



```
def sum_squared_errors(w, X, y):
    #calculate the predictions
    predictions = np.matmul(w, X.T)
    #calculate the squared error
    square_error = np.square(y - predictions)
    #sum the squared error
    total_error = np.sum(square_error)
    return total_error
```

Figure 8: SSE implemented as a Python function.

To update the weights the derivate of SSE with respect to the weights is needed. The derivative can be computed as follows:

$$\nabla_w E(w) = \sum_{i=1}^{N} \nabla_w (y_i - \hat{y}_i)^2 = \sum_{i=1}^{N} \nabla_w (y_i - wx_i)^2 = \sum_{i=1}^{N} -2(y_i - wx_i)x_i$$

Figure 9 shows the Python implementation to compute the gradient for a single data point and the average gradient over the full dataset.

```python
def compute_gradient(w, xi, yi):
    #calculate the predictions
    prediction = np.inner(w,xi)
    #calculate the error
    error = yi - prediction
    #compute the gradient
    gradient = -2.0*error*xi
    return gradient

def compute_full_gradient(w, X, y):
    full_gradient = 0.0
    #calculate the gradient for every datapoint
    for i, x in enumerate(X):
        full_gradient = full_gradient + compute_gradient(w, x, y[i])

    #divide by the number of samples to get the average gradient over the dataset
    full_gradient = full_gradient/X.shape[0]
    return full_gradient
```

Figure 9: Gradient computation implemented as Python functions.

Finally, LSE can be implemented as shown in Figure 10, computing the gradient at every step and updating the weights with gradient scaled by the learning rate.

```python
def LSE(X, y, w, num_epochs, learning_rate, testing=False):
    #do gradient descent to find the optimal weights

    #for testing
    weights = []
    errors = []

    for epoch in range(num_epochs):
        #compute gradient for each of the weights for this epoch
        gradient_epoch = compute_full_gradient(w, X, y)
        #update each of the weights
        w = w - (learning_rate*gradient_epoch)

        if testing:
            #compute error for this epoch
            error_epoch = sum_squared_errors(w, X, y)
            #append values
            weights.append(w)
            errors.append(error_epoch)
            #print to see progress
            if epoch % 100 == 0:
                print(f"Epoch: {epoch} Sum Squared Error: {error_epoch}")

    if testing:
        return w, weights, errors

    return w
```

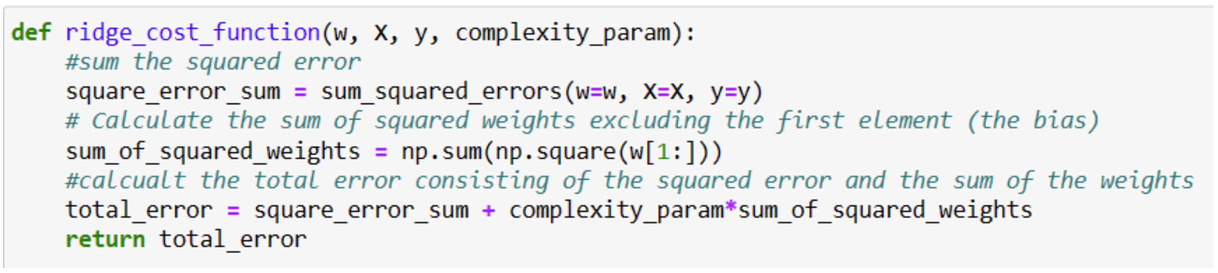
Figure 10: LSE regression implemented as a Python function.

3.2. Ridge regression implementation

The cost function used for ridge regression is similar to the squared error function but has an additional term to penalize large weights. Note that the bias, w0, is not penalized and can have any size without affecting the cost.

$$E(w) = \sum_{i=1}^{N}(y_i - \hat{y}_i)^2 + \lambda\sum_{j=1}^{p} {w_j}^2 \text{ ,} where\ \hat{y}_i = wx_i = \sum_{j=1}^{p} x_{ij}w_j \ + \ w_0$$

Figure *11* shows the Python implementation of this formula.



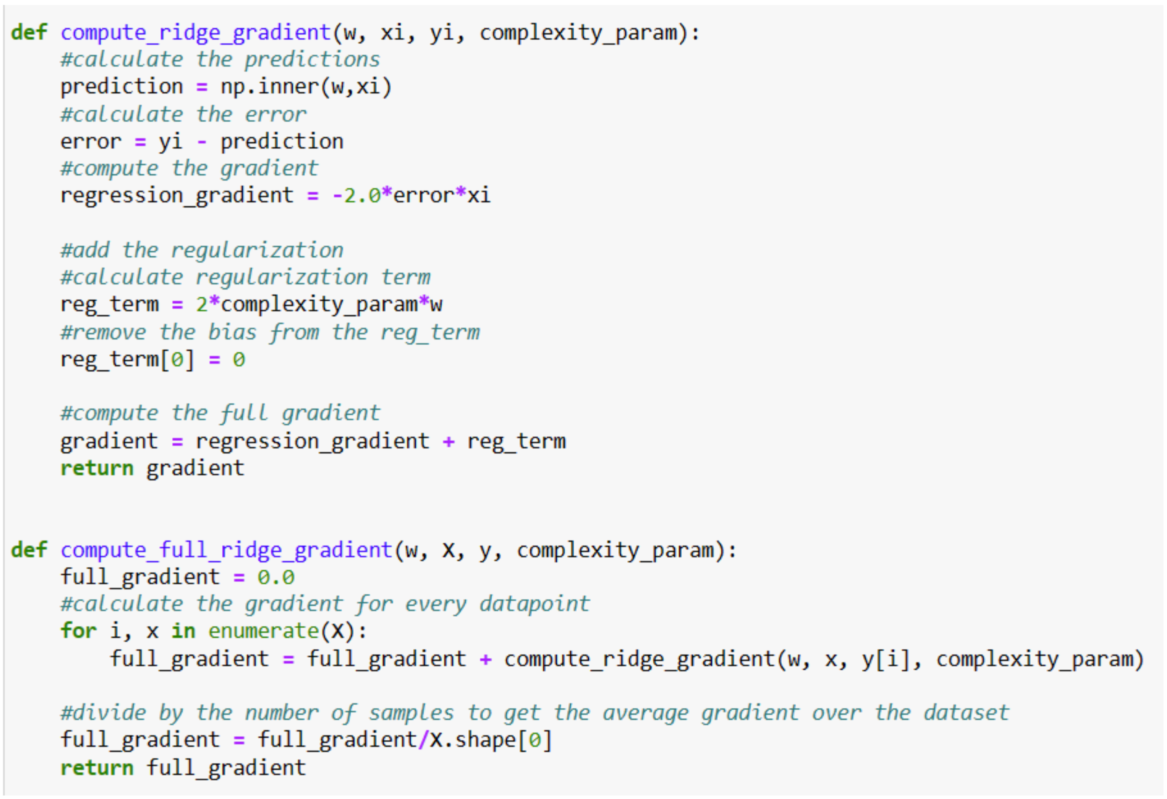
```
def ridge_cost_function(w, X, y, complexity_param):
    #sum the squared error
    square_error_sum = sum_squared_errors(w=w, X=X, y=y)
    # Calculate the sum of squared weights excluding the first element (the bias)
    sum_of_squared_weights = np.sum(np.square(w[1:]))
    #calcualt the total error consisting of the squared error and the sum of the weights
    total_error = square_error_sum + complexity_param*sum_of_squared_weights
    return total_error
```

Figure 11: Ridge cost function implemented as Python function.

Again, the derivative of the cost function is needed:

$$\nabla_w E(w) = \sum_{i=1}^{N} \nabla_w (y_i - \hat{y}_i)^2 + \lambda \sum_{j=1}^{p} \nabla_w {w_j}^2 = \sum_{i=1}^{N} -2(y_i - wx_i)x_i + 2\,\lambda \mathrm{w}$$

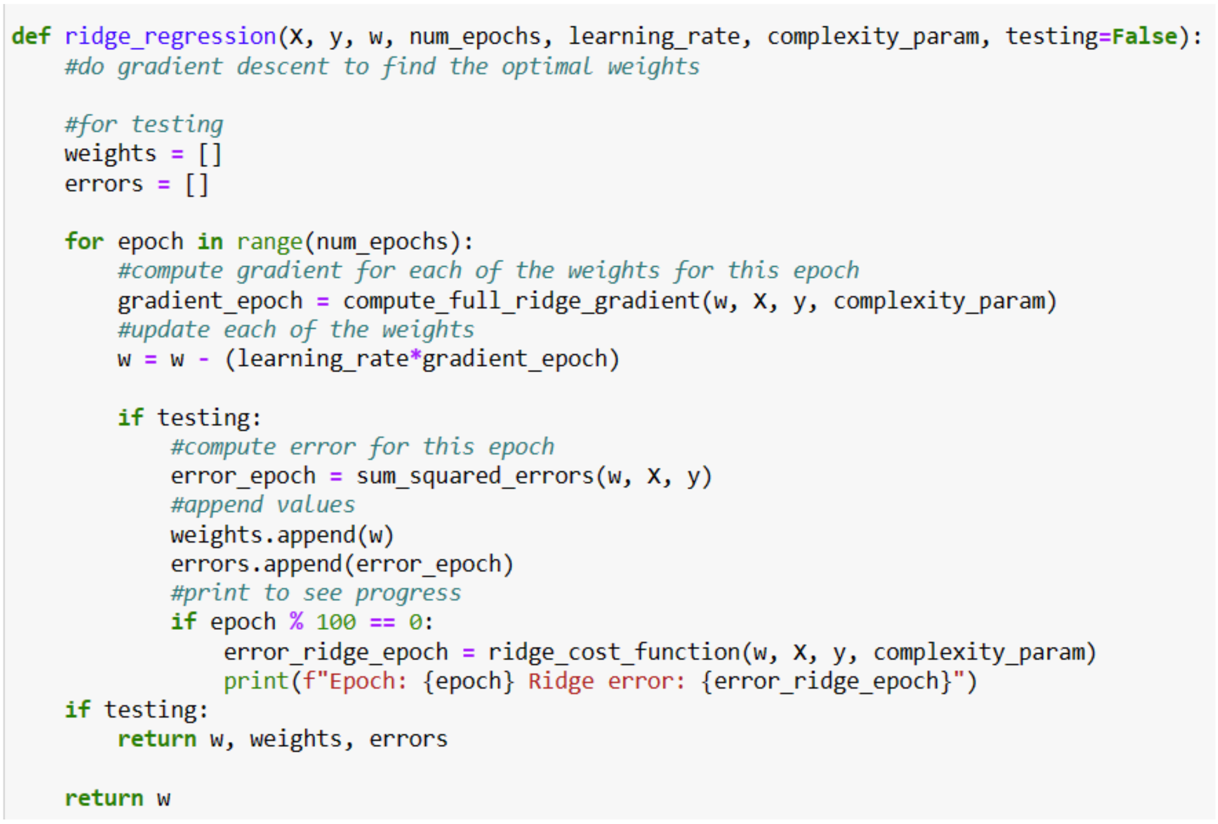
Figure *12* shows the Python implementation to compute the gradient for a single data point and the average gradient over the full dataset.



```
def compute_ridge_gradient(w, xi, yi, complexity_param):
    #calculate the predictions
    prediction = np.inner(w,xi)
    #calculate the error
    error = yi - prediction
    #compute the gradient
    regression_gradient = -2.0*error*xi

    #add the regularization
    #calculate regularization term
    reg_term = 2*complexity_param*w
    #remove the bias from the reg_term
    reg_term[0] = 0

    #compute the full gradient
    gradient = regression_gradient + reg_term
    return gradient


def compute_full_ridge_gradient(w, X, y, complexity_param):
    full_gradient = 0.0
    #calculate the gradient for every datapoint
    for i, x in enumerate(X):
        full_gradient = full_gradient + compute_ridge_gradient(w, x, y[i], complexity_param)

    #divide by the number of samples to get the average gradient over the dataset
    full_gradient = full_gradient/X.shape[0]
    return full_gradient
```

Figure 12: Ridge gradient computation implemented as Python functions.

Finally, ridge regression can be implemented as shown in Figure *13*.



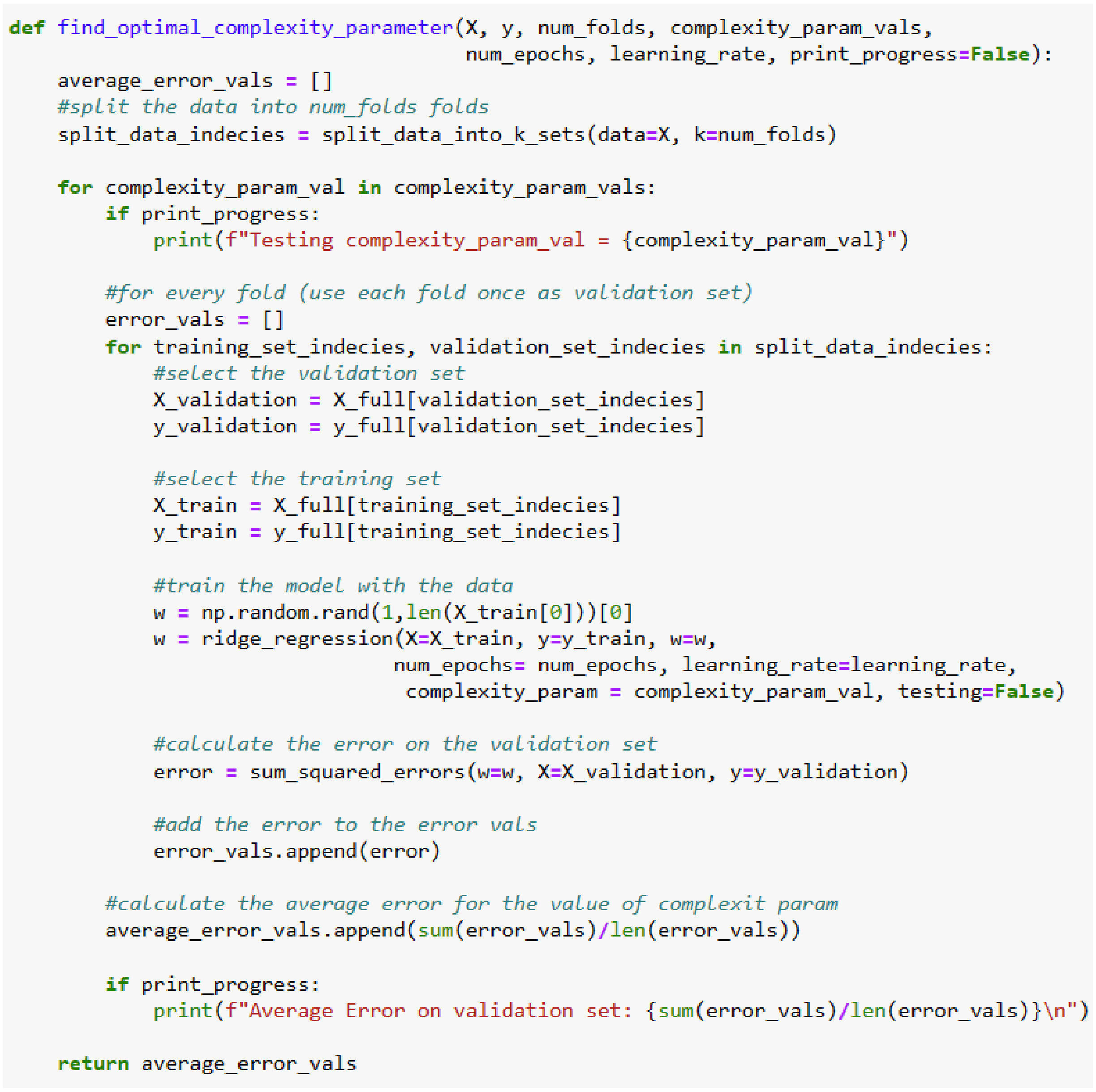
```python
def ridge_regression(X, y, w, num_epochs, learning_rate, complexity_param, testing=False):
    #do gradient descent to find the optimal weights

    #for testing
    weights = []
    errors = []

    for epoch in range(num_epochs):
        #compute gradient for each of the weights for this epoch
        gradient_epoch = compute_full_ridge_gradient(w, X, y, complexity_param)
        #update each of the weights
        w = w - (learning_rate*gradient_epoch)

        if testing:
            #compute error for this epoch
            error_epoch = sum_squared_errors(w, X, y)
            #append values
            weights.append(w)
            errors.append(error_epoch)
            #print to see progress
            if epoch % 100 == 0:
                error_ridge_epoch = ridge_cost_function(w, X, y, complexity_param)
                print(f"Epoch: {epoch} Ridge error: {error_ridge_epoch}")
    if testing:
        return w, weights, errors

    return w
```

Figure 13: Ridge regression implemented as a Python function.

4. Model Selection

4.1. Tuning the complexity parameter

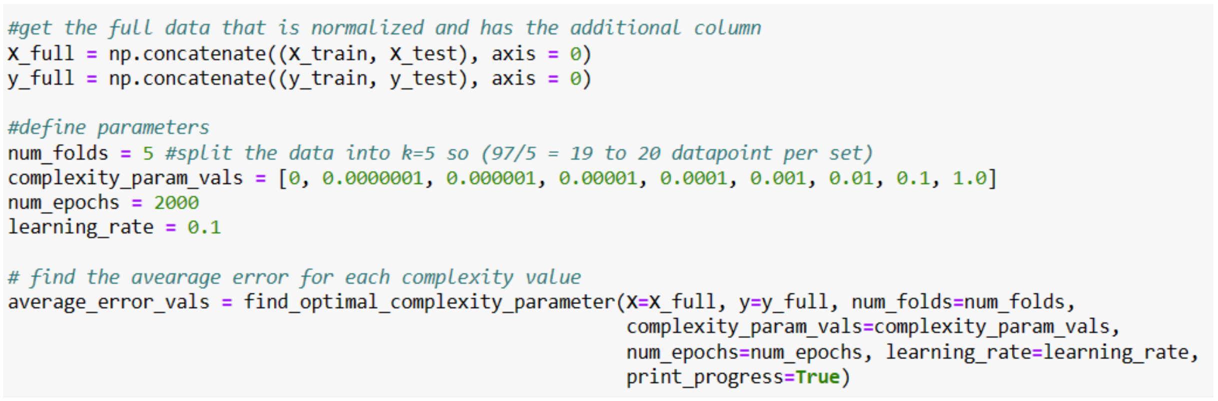
Before comparing the two regression models, its essential to tune the complexity parameter, also referred to as alpha. To determine the optimal alpha value, k-fold cross-validation is used. This method involves dividing the dataset into k-folds and iteratively using each fold as the validation set, while the remaining data serves as the training data. By varying the alpha value, its optimal value can be identified. This is implemented in Figure 14.



```python
def find_optimal_complexity_parameter(X, y, num_folds, complexity_param_vals,
                                      num_epochs, learning_rate, print_progress=False):
    average_error_vals = []
    #split the data into num_folds folds
    split_data_indecies = split_data_into_k_sets(data=X, k=num_folds)

    for complexity_param_val in complexity_param_vals:
        if print_progress:
            print(f"Testing complexity_param_val = {complexity_param_val}")

        #for every fold (use each fold once as validation set)
        error_vals = []
        for training_set_indecies, validation_set_indecies in split_data_indecies:
            #select the validation set
            X_validation = X_full[validation_set_indecies]
            y_validation = y_full[validation_set_indecies]

            #select the training set
            X_train = X_full[training_set_indecies]
            y_train = y_full[training_set_indecies]

            #train the model with the data
            w = np.random.rand(1,len(X_train[0]))[0]
            w = ridge_regression(X=X_train, y=y_train, w=w,
                                num_epochs= num_epochs, learning_rate=learning_rate,
                                 complexity_param = complexity_param_val, testing=False)

            #calculate the error on the validation set
            error = sum_squared_errors(w=w, X=X_validation, y=y_validation)

            #add the error to the error vals
            error_vals.append(error)

        #calculate the average error for the value of complexit param
        average_error_vals.append(sum(error_vals)/len(error_vals))

        if print_progress:
            print(f"Average Error on validation set: {sum(error_vals)/len(error_vals)}\n")

    return average_error_vals
```

Figure 14: K-fold cross-validation to find the optimal complexity parameter.

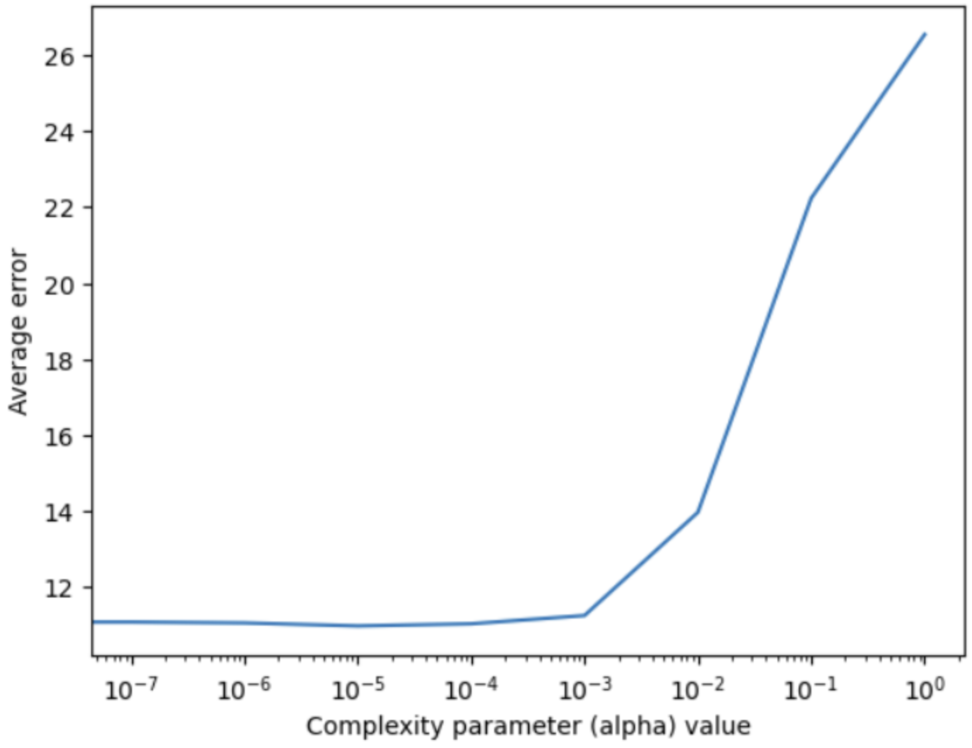
Initially, the optimal value is unknown, so a wide range of values are tested.



```python
#get the full data that is normalized and has the additional column
X_full = np.concatenate((X_train, X_test), axis = 0)
y_full = np.concatenate((y_train, y_test), axis = 0)

#define parameters
num_folds = 5 #split the data into k=5 so (97/5 = 19 to 20 datapoint per set)
complexity_param_vals = [0, 0.0000001, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1.0]
num_epochs = 2000
learning_rate = 0.1

# find the avearage error for each complexity value
average_error_vals = find_optimal_complexity_parameter(X=X_full, y=y_full, num_folds=num_folds,
                                                       complexity_param_vals=complexity_param_vals,
                                                       num_epochs=num_epochs, learning_rate=learning_rate,
                                                       print_progress=True)
```

Figure 15: Code to test a wide range of complexity parameter values.

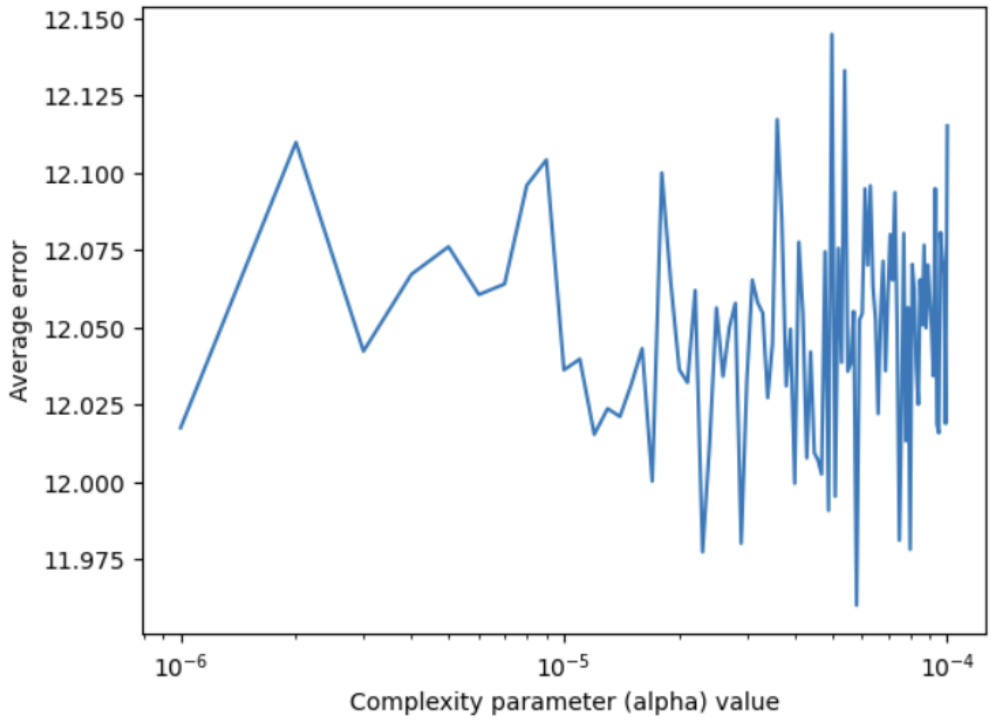
Plotting the error against the value of alpha gives the plot shown in Figure *16*.



```
Optimal complexity_param: 1e-05, error: 10.985016856268
```

Figure 16: Plot of average error against complexity parameter value.

Figure *16* shows that a small complexity parameter gives a smaller error than a larger value. By printing the complexity parameter value corresponding to the smallest error an optimal value of 1e-05 is found. To get a better result, the search can be repeated with a smaller range of values.



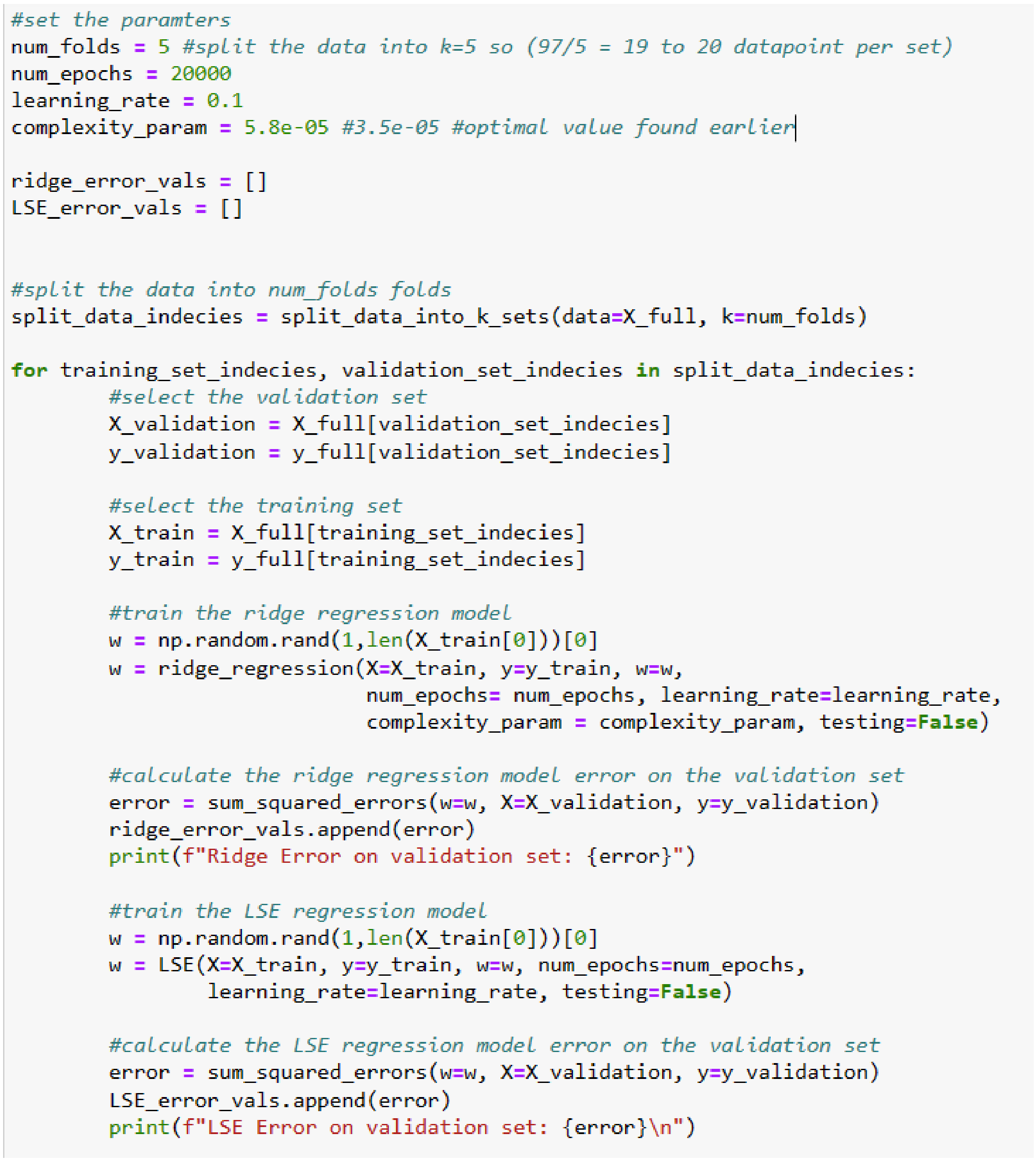
```
Optimal complexity_param (alpha): 5.7999999999999994e-05, error: 11.959950691173972
```

Figure 17: Plot of average error against complexity parameter value for the reduced range.

Figure *17* shows the average error over a range from 1e-06 to 1e-04. While the average error remains relatively stable, it highlights an optimal value of 5.8e-05.

4.2. Selecting the best model

With the optimal complexity parameter value determined, k-fold cross-validation can be employed to compare the performance of both models and select the best.



```python
#set the paramters
num_folds = 5 #split the data into k=5 so (97/5 = 19 to 20 datapoint per set)
num_epochs = 20000
learning_rate = 0.1
complexity_param = 5.8e-05 #3.5e-05 #optimal value found earlier

ridge_error_vals = []
LSE_error_vals = []


#split the data into num_folds folds
split_data_indecies = split_data_into_k_sets(data=X_full, k=num_folds)

for training_set_indecies, validation_set_indecies in split_data_indecies:
        #select the validation set
        X_validation = X_full[validation_set_indecies]
        y_validation = y_full[validation_set_indecies]

        #select the training set
        X_train = X_full[training_set_indecies]
        y_train = y_full[training_set_indecies]

        #train the ridge regression model
        w = np.random.rand(1,len(X_train[0]))[0]
        w = ridge_regression(X=X_train, y=y_train, w=w,
                             num_epochs= num_epochs, learning_rate=learning_rate,
                             complexity_param = complexity_param, testing=False)

        #calculate the ridge regression model error on the validation set
        error = sum_squared_errors(w=w, X=X_validation, y=y_validation)
        ridge_error_vals.append(error)
        print(f"Ridge Error on validation set: {error}")

        #train the LSE regression model
        w = np.random.rand(1,len(X_train[0]))[0]
        w = LSE(X=X_train, y=y_train, w=w, num_epochs=num_epochs,
                learning_rate=learning_rate, testing=False)

        #calculate the LSE regression model error on the validation set
        error = sum_squared_errors(w=w, X=X_validation, y=y_validation)
        LSE_error_vals.append(error)
        print(f"LSE Error on validation set: {error}\n")
```

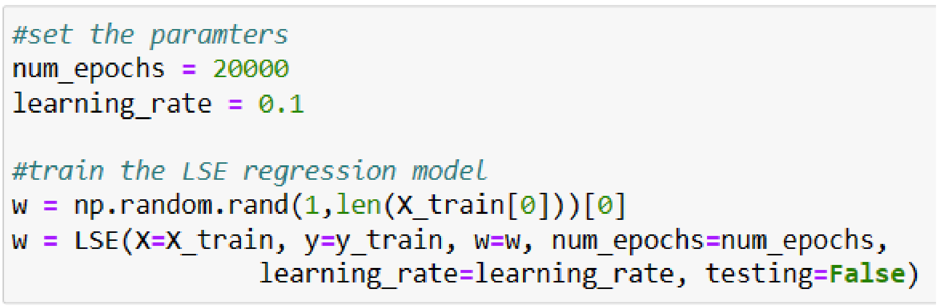
Figure 18: Code to perform k-fold cross-validation to select the best model.

Comparing the average error for both rounded to 3 decimal figures gives an average error on the validation set of 10.664 for both models. The average error varies slightly for different runs of the cross-validation, because of the random selection of training and validation sets, but it's clear that the errors are very close together.

Considering that both models are identical, the LSE model is the preferred choice for making the final predictions. The limited dataset size could mean that the regularization term in ridge regression could potentially dominate the optimization process, resulting in excessive weight shrinkage and information loss. Although the small dataset makes LSE susceptible to overfitting, it is the best choice due to its simplicity and lack of complexity parameter tuning. Ideally, the dataset would be much larger, allowing for better fitting by training on more examples and more effective cross-validation by increasing the number of elements in each fold.

5. Interpretation of results

Now the final model can be trained.



```
#set the paramters
num_epochs = 20000
learning_rate = 0.1

#train the LSE regression model
w = np.random.rand(1,len(X_train[0]))[0]
w = LSE(X=X_train, y=y_train, w=w, num_epochs=num_epochs,
                learning_rate=learning_rate, testing=False)
```

Figure 19: Code to train the final LSE regression model.

The weights learned by running the code in Figure *19* can be used to predict the level of cancer antigen for new data. To visualize its effectiveness, the predicted level of cancer antigen is plotted against the true values from the test dataset.

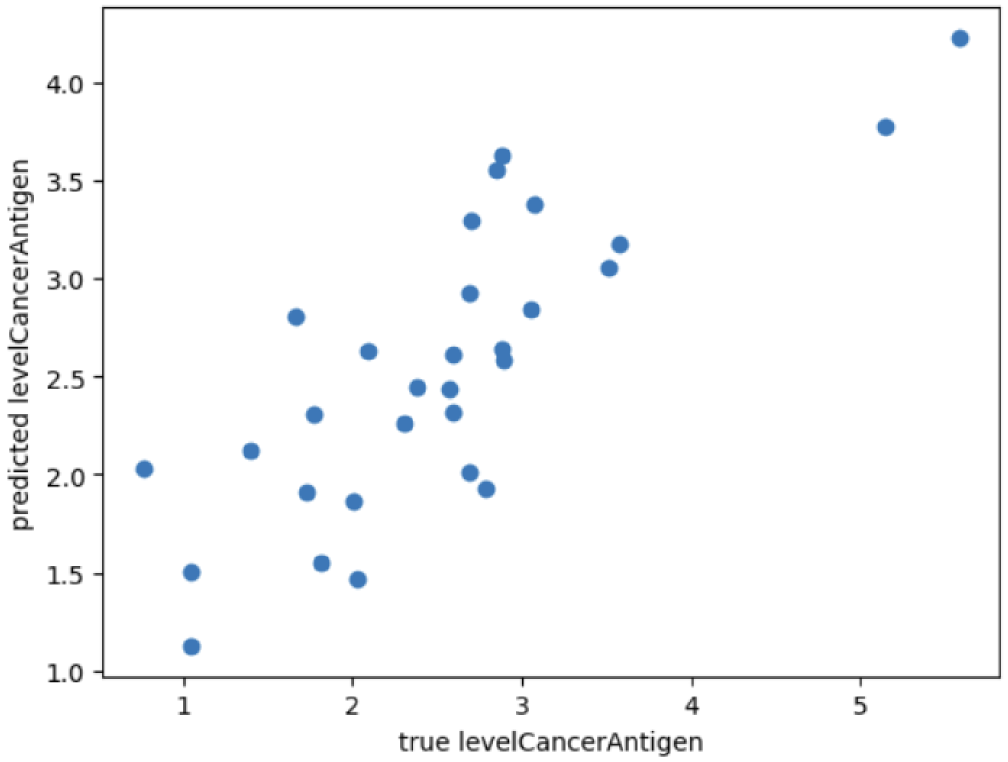


Figure 20: Predicted levelCancerAntigen against true levelCancerAntigen.

Figure *20* shows that the points lie around a straight line, indicating that the model gives reasonable predictions. For a perfect model, all the points would lie on a single straight line, but in real-world data, a model like this does not exist. Considering the lack of training data, the model performs as expected.

The weights can also help identify which clinical measures have the greatest influence on the level of cancer antigen.

Table 1: The model weights and their corresponding features.

Weight	Value	Corresponding feature
w0	1.385	
w1	8.508	logCancerVol
w2	5.434	logCancerWeight
w3	-4.714	age
w4	1.133	logBenighHP
w5	2.977	svi
w6	-0.557	logCP
w7	0.255	gleasonScore
w8	0.286	gleasonS45

Table 1 shows that w1, w2 and w4 have the greatest value, signifying that logCancerVol, logCancerWeight and svi are the clinical measures with the most significant impact on a patient's cancer antigen level. Age, with negative weight, displays an inverse relationship, while the remaining features have relatively small weights, indicating a small influence on cancer antigen levels.