Assignment-01

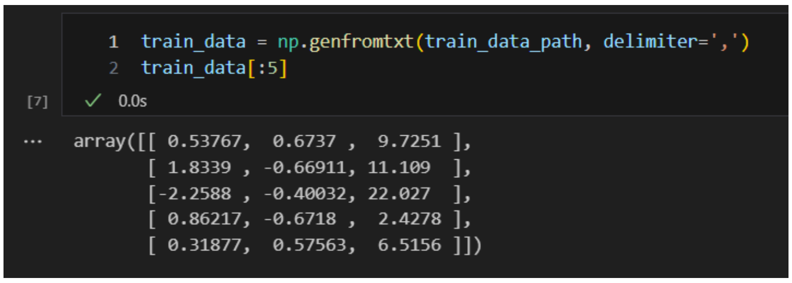
Loading Data

Before starting coding algorithms let me explain how I load data in python after downloading data files from drive. I uses two modules(libraries), os and NumPy in python to load data. From those 'os' is used for handling operating system dependent functionality and NumPy is used for handling any operations for matrix like matrix multiplication, inverse of matrix and etc.

1. **Getting Paths for Required files**
   I first found the path of directory which stores current file using `os.path.dirname(__file__)` then I joined this path of directory with name of files which contain training and testing data. To get path of those files and to join files name with path I used `os.path.join` function in 'os' module. And finally I will get paths to required files. The complete code will be as below:

   ```
   directory_name = os.path.dirname(__file__)
   train_data_path = os.path.join(directory_name, 'FMLA1Q1Data_train.csv
   test_data_path = os.path.join(directory_name, 'FMLA1Q1Data_test.csv')
   ```

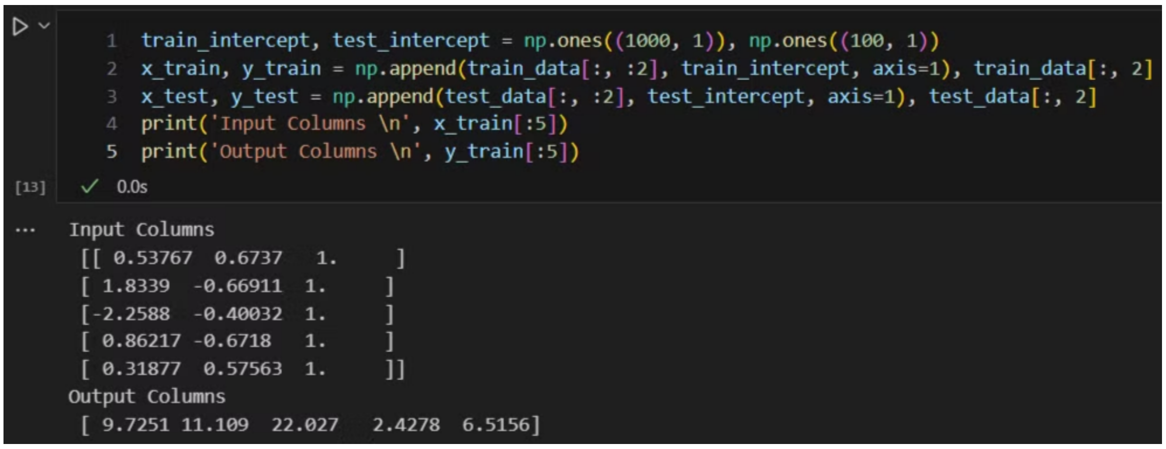
2. For loading data from given files to matrix I used NumPy. To read .csv files I used NumPy function `genfromtxt` , given the files paths and delimiter as comma it load data in matrix form. Below I have shown first 5 entries of dataset.

   ```
   train_data = np.genfromtxt(train_data_path, delimiter=',')
   train_data[:5]
   [7]  ✓ 0.0s
   ...  array([[ 0.53767,  0.6737 ,  9.7251 ],
              [ 1.8339 , -0.66911, 11.109  ],
              [-2.2588 , -0.40032, 22.027  ],
              [ 0.86217, -0.6718 ,  2.4278 ],
              [ 0.31877,  0.57563,  6.5156 ]])
   ```

   Same will be done for test data and get it loaded.

3. Now I have split the loaded data into input columns and target column. Which can be done easily in NumPy. As I have also added bias in columns of training and testing data. So, first I created an array of all one's with similar number of data point for train and test respectively and add them in columns of input data of test and train respectively using `numpy.append` function. First five rows of train data for slitted columns with added bias are shown below :



```python
train_intercept, test_intercept = np.ones((1000, 1)), np.ones((100, 1))
x_train, y_train = np.append(train_data[:, :2], train_intercept, axis=1), train_data[:, 2]
x_test, y_test = np.append(test_data[:, :2], test_intercept, axis=1), test_data[:, 2]
print('Input Columns \n', x_train[:5])
print('Output Columns \n', y_train[:5])
```

```
Input Columns
 [[ 0.53767  0.6737   1.     ]
 [ 1.8339  -0.66911  1.     ]
 [-2.2588  -0.40032  1.     ]
 [ 0.86217 -0.6718   1.     ]
 [ 0.31877  0.57563  1.     ]]
Output Columns
 [ 9.7251 11.109  22.027   2.4278  6.5156]
```

The reason to add bias term is that the models are able to generalize data well and able to understand how data is distributed which will eventually give less error for predicted data points.

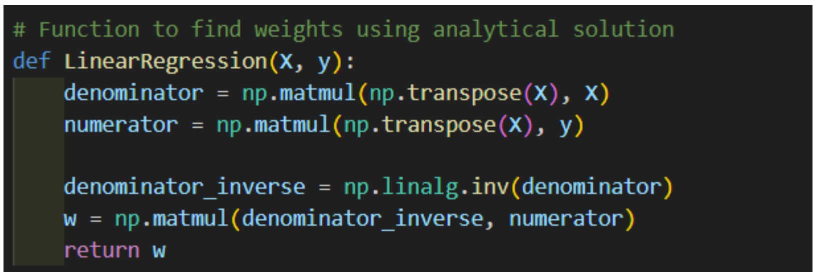
i. **Write a piece of code to obtain the least squares solution w_{ML} to the regression problem using the analytical solution.**

- As we already know the analytical solution of regression problem is given by finding minimum weights of features to get least loss. That will be:

$$error(h) = \sum_{i=1}^{n} (h(x_i) - y_i)^2$$
$$f(\mathrm{w}) = \min_{\mathrm{w} \in \mathbb{R}^d} \sum_{i=1}^{n} (x_i.\mathrm{w} - y_i)^2 = \min_{\mathrm{w} \in \mathbb{R}^d} ||X\mathrm{w} - Y||^2$$
$$\nabla f(\mathrm{w}) = 2((X^T X)\mathrm{w} - X^T Y) = 0$$
$$\mathrm{w}^* = (X^T X)^{-1} X^T Y$$

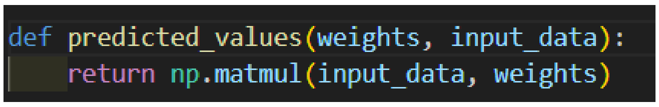
In above equation X is nxd, Y is $\mathrm{nx1}$ and w is $\mathrm{dx1}$ matrices. And same will be considered for all.

By using above solution I calculated W^*. I first found $X^T X$ which is `denominator` variable in my code then inversed it which is `denominator_inverse` and multiplied that with $X^T Y$ which is calculated in `numerator` and matrix multiplication was done using `numpy.matmul`. And by doing that I have solved it for W^*. The respective code is:



```python
# Function to find weights using analytical solution
def LinearRegression(X, y):
    denominator = np.matmul(np.transpose(X), X)
    numerator = np.matmul(np.transpose(X), y)

    denominator_inverse = np.linalg.inv(denominator)
    w = np.matmul(denominator_inverse, numerator)
    return w
```

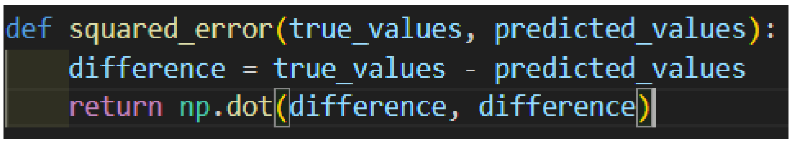
- For predicting new values I have done XW^* in code it will be:



```
def predicted_values(weights, input_data):
    return np.matmul(input_data, weights)
```

This same prediction function will be used for all models except Kernel Regression.

- For finding error I used sum of squared error as my error function which will be $SSE(X) = ||XW^* - Y_{true}||^2$.
  code for the same will be:



```
def squared_error(true_values, predicted_values):
    difference = true_values - predicted_values
    return np.dot(difference, difference)
```

This same error function will be used for all models.

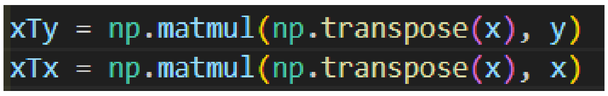
- The sum of squared error for model with bias is **6600.56** and the error without bias is **14276.61 .** So, I will be using input data with bias for all model.

ii. **Code the gradient descent algorithm with suitable step size to solve the least squares algorithms and plot $||\mathrm{w}^t - \mathrm{w}_{ML}||_2$ as a function of t. What do you observe?**

- As we know from above derivation the gradient of my loss function will be is:

$$\nabla f(\mathrm{w}) = 2((X^T X)\mathrm{w} - X^T Y)$$

In my code $X^T X$ is named as `xTx` and $X^T Y$ is named as `xTy` and the same will be considered for all following algorithms. In code it looks like:



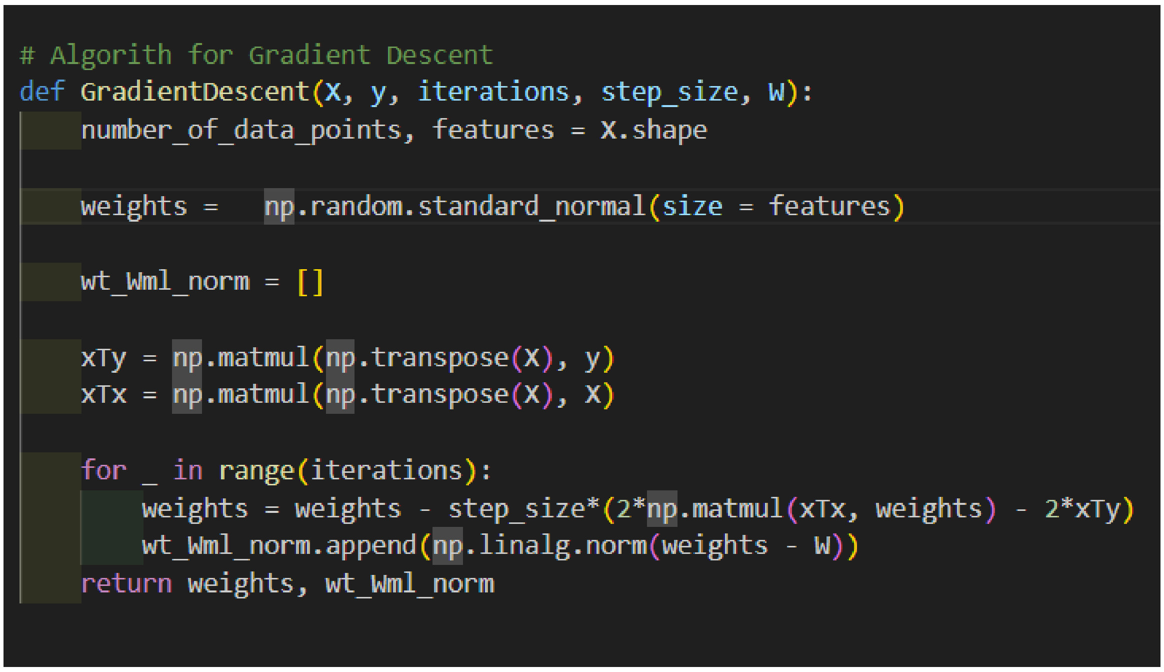
```
xTy = np.matmul(np.transpose(x), y)
xTx = np.matmul(np.transpose(x), x)
```

and my weights for each iterations will be updated as:

$$\mathrm{w}^{t+1} = \mathrm{w}^t - \eta^t \nabla f(\mathrm{w}^t)$$

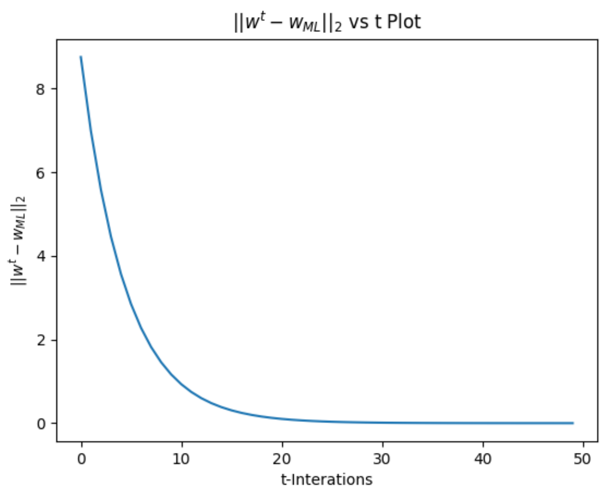
Here η^t is step size and w^{t+1} is updated weights.

- I first initialize w^t as `weights` in code which is some random data drawn from standard normal distribution using NumPy.
  Then I found
  $X^T X$ and $X^T Y$ and iterated over $\nabla f(\mathrm{w}^t)$ to get optimal weights. So, code for it is:



```python
# Algorith for Gradient Descent
def GradientDescent(X, y, iterations, step_size, W):
    number_of_data_points, features = X.shape

    weights =   np.random.standard_normal(size = features)

    wt_Wml_norm = []

    xTy = np.matmul(np.transpose(X), y)
    xTx = np.matmul(np.transpose(X), X)

    for _ in range(iterations):
        weights = weights - step_size*(2*np.matmul(xTx, weights) - 2*xTy)
        wt_Wml_norm.append(np.linalg.norm(weights - W))
    return weights, wt_Wml_norm
```

- And the plot of $||\mathrm{w}^t - \mathrm{w}_{ML}||_2$ as function of t for step size of 0.0001 and 50 iterations will be plot using values from `wt_Wml_norm` variable:



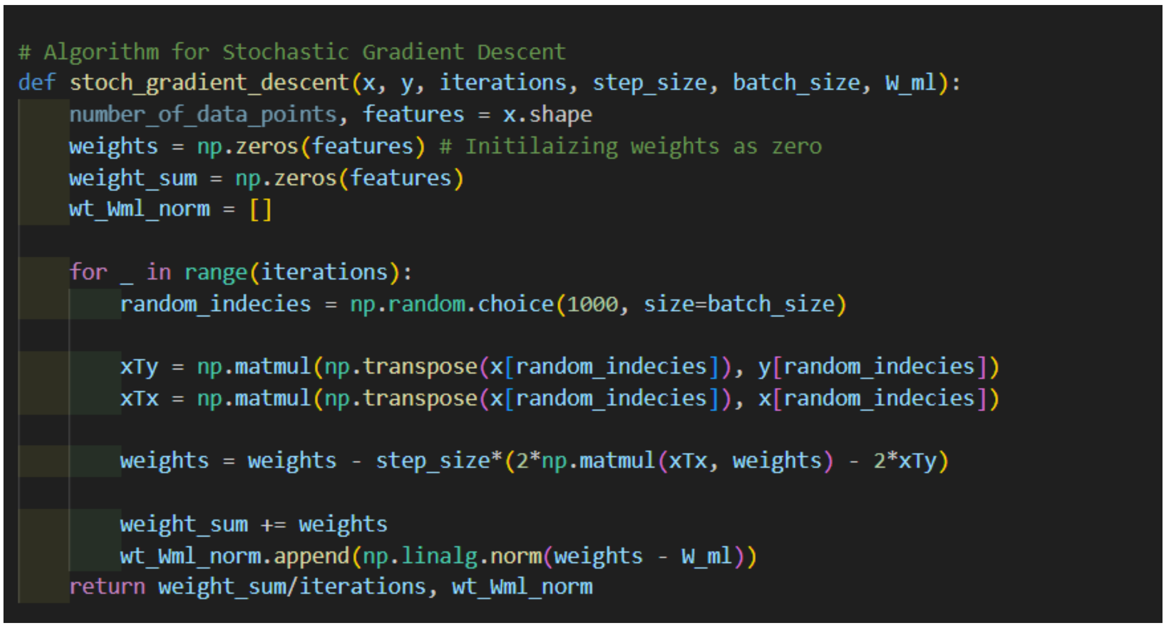
- **Observations:**
  1. The observed error for test data are almost same for gradient descent and analytical solution i.e. 6600.55 and 6600.51 for analytical solution and gradient descent respectively.
  2. The weights of gradient descent algorithm are gradually getting closer to the weights of analytical solution and at 50th iteration the weights of both algorithm are similar.

iii. **Code the stochastic gradient descent algorithm using batch size of 100 and plot $||\mathrm{w}^t - \mathrm{w}_{ML}||_2$ as a function of t. What are your observations?**

- For stochastic gradient I considered 100 uniformly random picked data pointes from given dataset and perform gradient descent on it. If I consider $\tilde{\mathrm{x}}$ and $\tilde{\mathrm{y}}$ as my sampled data points consider them as my dataset them update rule for weights will be:

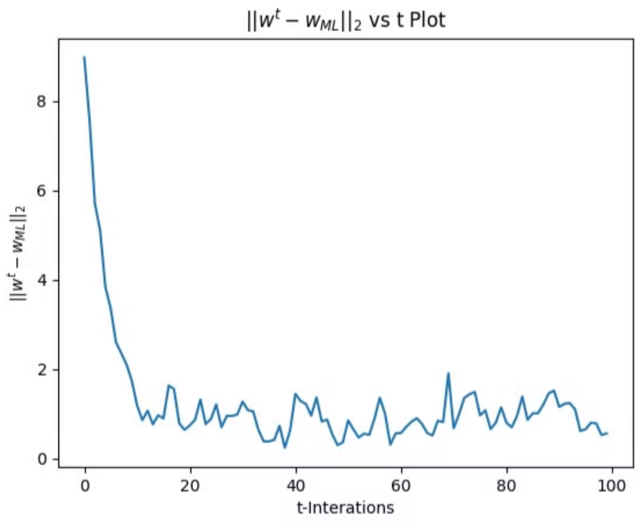
$$\nabla f(\mathrm{w}) = 2((\tilde{\mathrm{x}}^T\tilde{\mathrm{x}})\mathrm{w} - \tilde{\mathrm{x}}^T\tilde{\mathrm{y}})$$
$$\mathrm{w}^{t+1} = \mathrm{w}^t - \eta^t\nabla f(\mathrm{w}^t)$$
$$\mathrm{w}_{SGD} = \frac{1}{T}\sum_{t=1}^{T}\mathrm{w}_t$$

- I started by initializing weights as zeros then over t iterations I randomly picked 100 (given batch size) random indices from 0 to 999 and applied gradient descend on them to updated the weights. To get final weights I stored sum of all the weights in `weight_sum` variable and for each iteration new weights will be added to it and return average of them at end. The code for similar is:



```
# Algorithm for Stochastic Gradient Descent
def stoch_gradient_descent(x, y, iterations, step_size, batch_size, W_ml):
    number_of_data_points, features = x.shape
    weights = np.zeros(features) # Initilaizing weights as zero
    weight_sum = np.zeros(features)
    wt_Wml_norm = []

    for _ in range(iterations):
        random_indecies = np.random.choice(1000, size=batch_size)

        xTy = np.matmul(np.transpose(x[random_indecies]), y[random_indecies])
        xTx = np.matmul(np.transpose(x[random_indecies]), x[random_indecies])

        weights = weights - step_size*(2*np.matmul(xTx, weights) - 2*xTy)

        weight_sum += weights
        wt_Wml_norm.append(np.linalg.norm(weights - W_ml))
    return weight_sum/iterations, wt_Wml_norm
```

- The plot of $||\mathrm{w}^t - \mathrm{w}_{ML}||_2$ as function of t for step size of 0.001, 100 iterations and batch size of 100 will be:



- Observations:
  1. Compared to error of analytical solution which is 6600.55 and error from stochastic gradient descent is 6535.76 which are almost same. But this error will change every time algorithm is run as random data points are draw every time.
  2. In compare to gradient descent smooth fall and which stop updating weights after some point, the stochastic gradient descent algorithm due to it's random samples doesn't take a smooth path and never stops updating weights but remains in a range of weights that give less error.

iv. **Ridge Regression**

1. **Code the gradient descent algorithm for ridge regression.**
   - As we know loss function to minimize in ride regression is:

$$f(\mathrm{w}) = \min_{\mathrm{w} \in \mathbb{R}^d} \sum_{i=1}^{n} (x_i.\mathrm{w} - y_i)^2 + \lambda \sum_{i=1}^{d} \mathrm{w}_i^2$$
$$\text{Similar can written in matrix form as :}$$
$$f(\mathrm{w}) = \min_{\mathrm{w} \in \mathbb{R}^d} ||X\mathrm{w} - Y||^2 + \lambda ||\mathrm{w}||^2$$

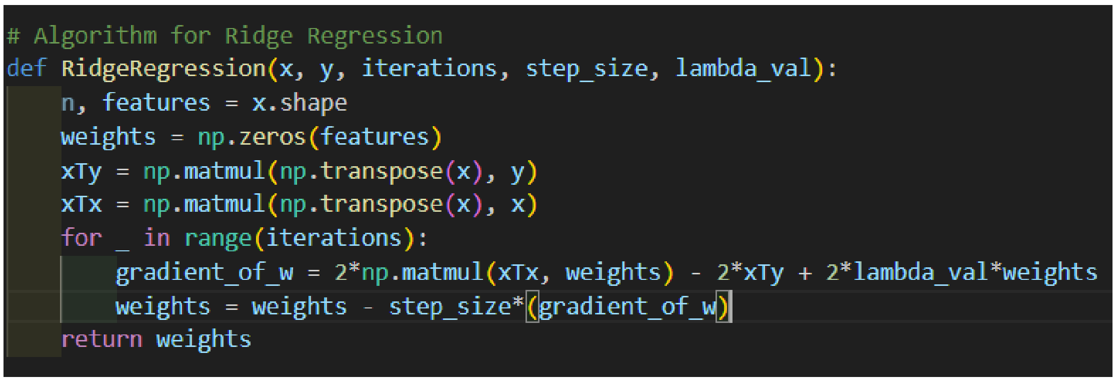
For ridge regression using gradient descent I will take gradient of this function. Then we will get:

$$\nabla f(\mathrm{w}) = 2((X^T X)\mathrm{w} - X^T Y) + 2\lambda \mathrm{w}$$

So weights of ridge regression will be updated as:

$$\mathrm{w}^{t+1} = \mathrm{w}^t - 2\eta^t((X^T X)\mathrm{w}^t - X^T Y + \lambda \mathrm{w}^t)$$

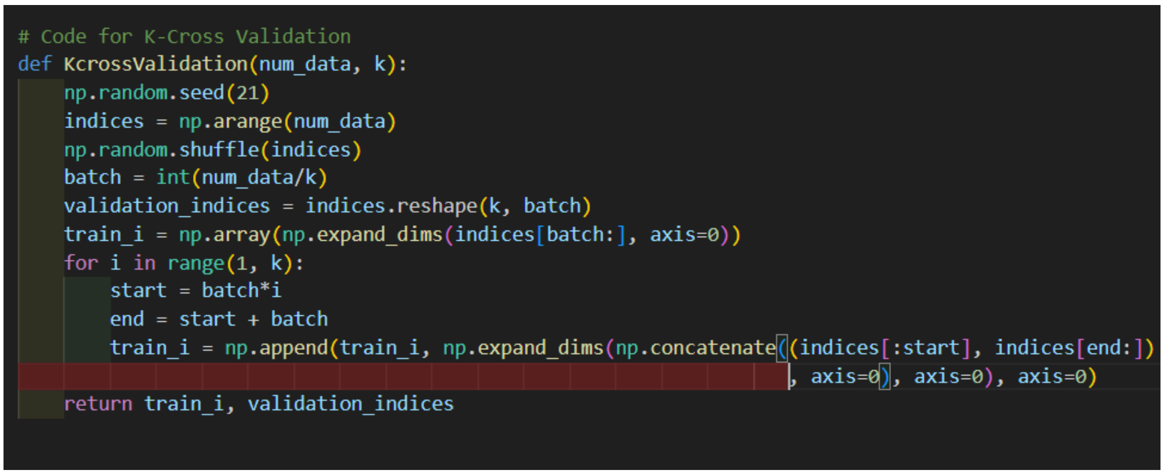
I first initialized weights as zeros into `weights` variable then calculate X^TX, X^TY and stored in variables `xTx` and `xTy` respectively. Then in each iterations I updated weights using above w^{t+1} equation. In code it will be:



```
# Algorithm for Ridge Regression
def RidgeRegression(x, y, iterations, step_size, lambda_val):
    n, features = x.shape
    weights = np.zeros(features)
    xTy = np.matmul(np.transpose(x), y)
    xTx = np.matmul(np.transpose(x), x)
    for _ in range(iterations):
        gradient_of_w = 2*np.matmul(xTx, weights) - 2*xTy + 2*lambda_val*weights
        weights = weights - step_size*(gradient_of_w)
    return weights
```

2. **Algorithm for K Fold Cross-validation**

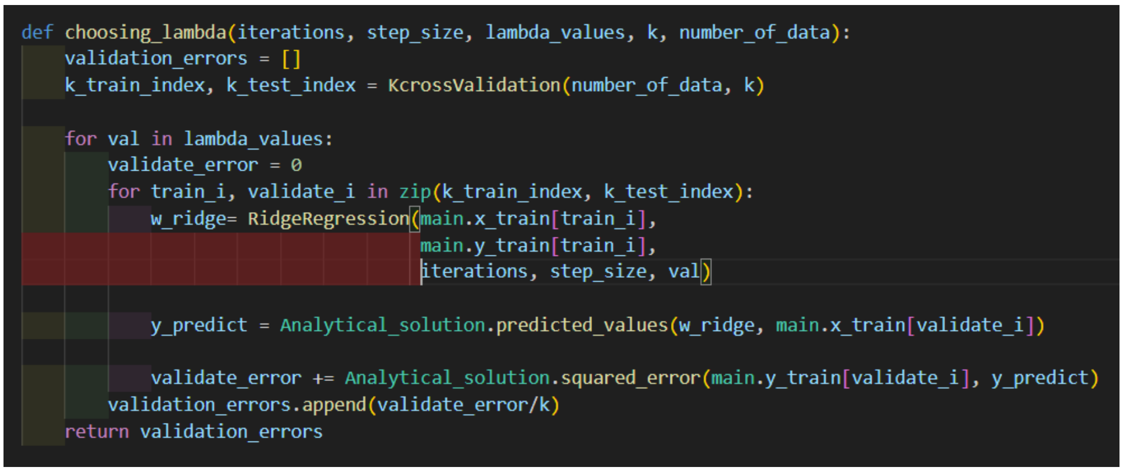
- I first initialize a array with name `indices` which contains integers similar to indices of stored data points. Then I shuffled this array randomly using NumPy function `numpy.random.shuffle()` which takes an array and shuffles it's in-place. To keep shuffle same for all run I used `numpy.random.seed()` which takes an integer and give same random values for all run of code. Then with respect to k-value I found how many data points should be there in one fold which is $\frac{number\ of\ data\ points}{k-value}$ which I took as `num_data/k` . After that I found indices for validation just by reshaping the `indices` array in k-folds. For training indices I assigned `train_i` variable in which I just divided indices in two parts for each fold, which are indices before validation indices, indices after validation indices and merge them to form training indices and looping it for k time will give training indices for k-folds. The code for same will be



```
# Code for K-Cross Validation
def KcrossValidation(num_data, k):
    np.random.seed(21)
    indices = np.arange(num_data)
    np.random.shuffle(indices)
    batch = int(num_data/k)
    validation_indices = indices.reshape(k, batch)
    train_i = np.array(np.expand_dims(indices[batch:], axis=0))
    for i in range(1, k):
        start = batch*i
        end = start + batch
        train_i = np.append(train_i, np.expand_dims(np.concatenate((indices[:start], indices[end:])
                                                                  , axis=0), axis=0), axis=0)
    return train_i, validation_indices
```

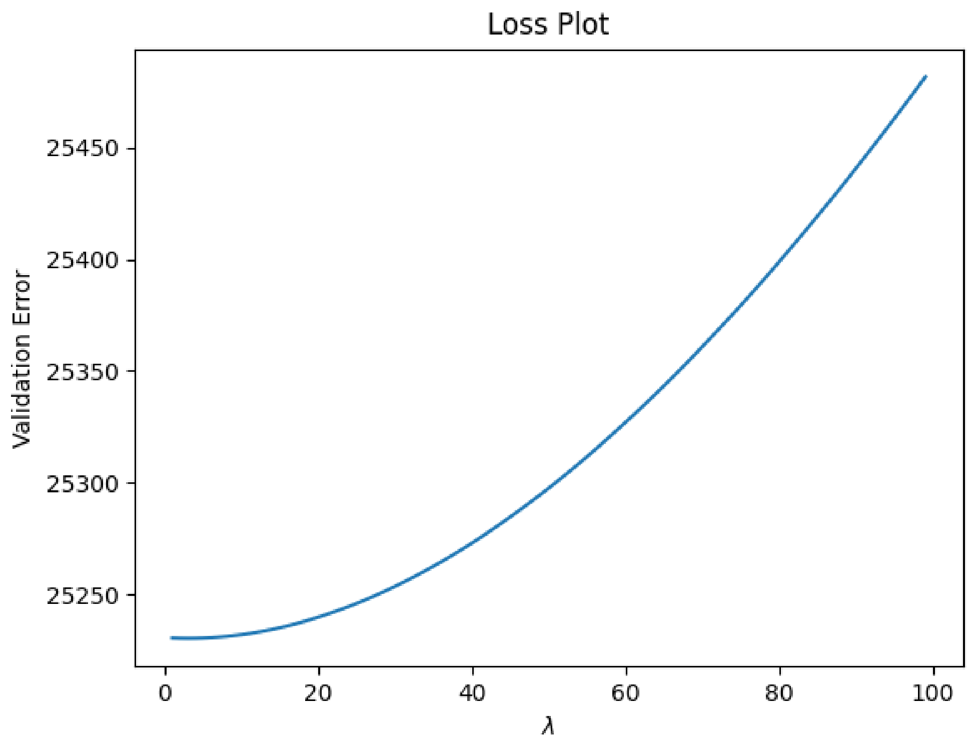
3. **Cross-validating for various choices of λ and plotting error in the validation set as a function of λ.**

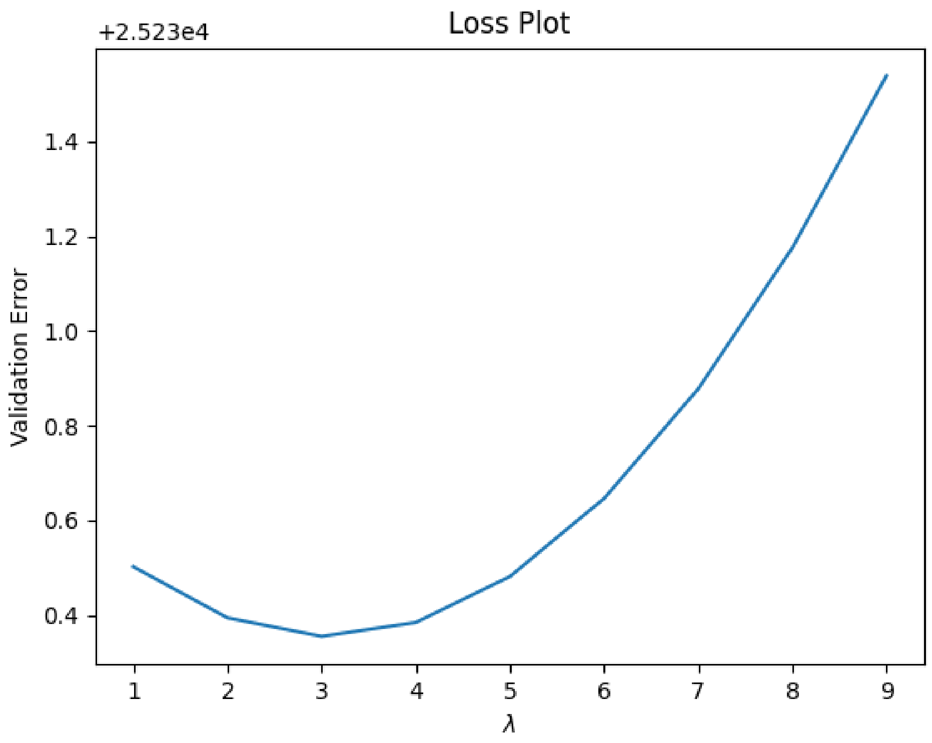
- First I created a function which used previous two function to cross validate λ. It takes input as number of iteration, step size, different lambda values as list for ridge regression with gradient descent and k-values, total number of data points for K-Cross validation. After k-folds it averages the mean and store in `validation_errors` to form plot. So, the code for the same is:

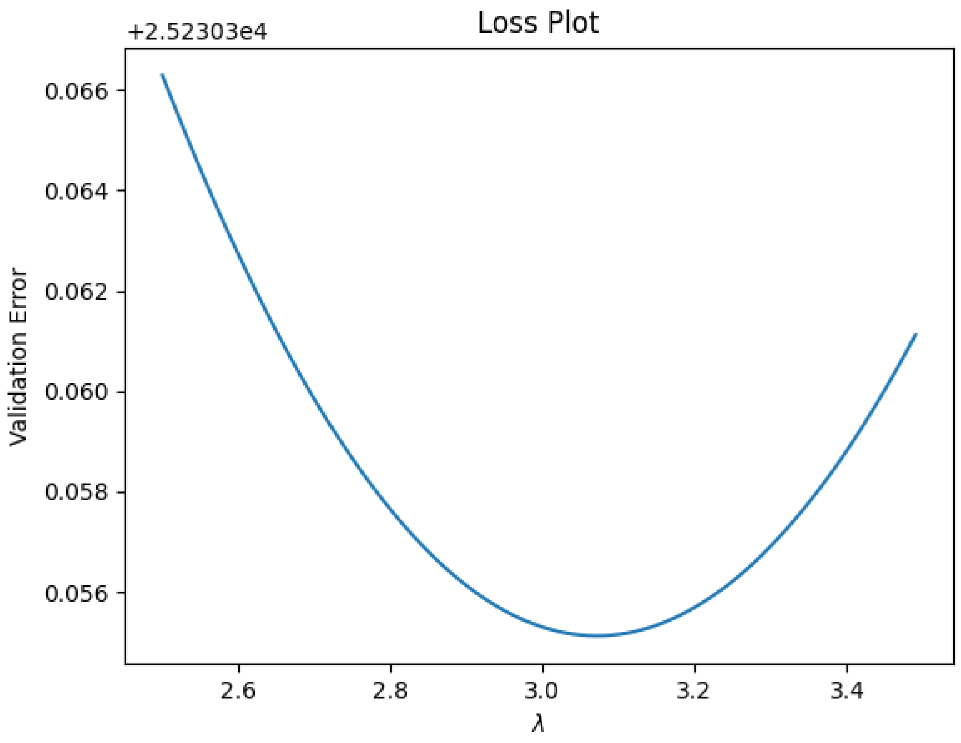


```python
def choosing_lambda(iterations, step_size, lambda_values, k, number_of_data):
    validation_errors = []
    k_train_index, k_test_index = KcrossValidation(number_of_data, k)

    for val in lambda_values:
        validate_error = 0
        for train_i, validate_i in zip(k_train_index, k_test_index):
            w_ridge= RidgeRegression(main.x_train[train_i],
                                     main.y_train[train_i],
                                     iterations, step_size, val)

            y_predict = Analytical_solution.predicted_values(w_ridge, main.x_train[validate_i])

            validate_error += Analytical_solution.squared_error(main.y_train[validate_i], y_predict)
        validation_errors.append(validate_error/k)
    return validation_errors
```

- I found λ for which validation error is least and graph for the same are:







And I found best λ value at $\lambda = 3.07$ and for this value of λ I calculate

$\mathrm{w}_R = [1.76, 3.51, 9.86]$ i.e. weight for first feature is 1.76, second feature is 3.51 and bias (which is all ones column) is 9.86.

- **Comparing test error for w_R and w_{ML}.**

  Sum of squared for w_{ML} is 6600.55 and for w_R is 6595.88. For me there was not large enough difference in both but still Ridge regression some what small error. But w_R is better because it keeps weights of features under check by penalizing them. More the weights for features are then the grater regulation term. Which is lambda time squared of it's features weight. Which helps in decreasing bias.

v. **Kernel Regression**

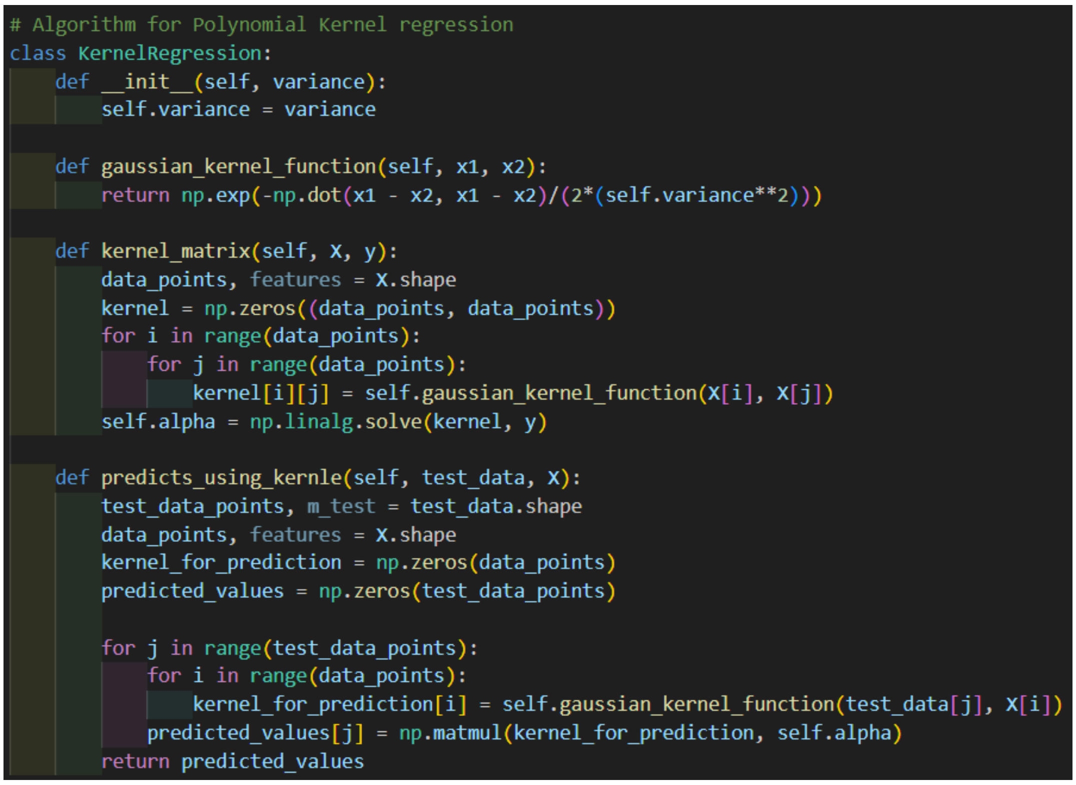
1. **Which Kernel would you choose and why?**
   - I would choose gaussian kernel because this kernel function maps data points in infinite dimension. Due to this mapping of data points our data becomes linear in higher dimension. Even if our data doesn't has a linear relationship in 1 or 2-dimension it will have linear relation with target feature in infinite dimension and Gaussian kernel function can be shown to be a valid function. So, I think choosing gaussian kernel is better choice.
2. **Explaining Code of kernel regression algorithm with predicting for test data.**
   - For kernel regression there is no need of bias in input data, so first I considered input data without bias term in it and named them as `x_train_kernel` which is input features of training data, `y_train_kernel` which is target feature for training data, and similarly `x_test_kernel` and `y_test_kernel` are for input features for testing model over prediction and true values for predictions respectively.
   - Let's first understand mathematics behind kernel function as we know:
     We considered $\mathrm{w_{ML}} = (X^TX)^{-1}(X^TY) = X^T\alpha$. Then $\alpha = (XX^T)^{-1}Y$. Then we will convert XX^T into kernel to get $\alpha = K^{-1}Y$ after getting α we do predictions for new values which will be $\hat{y} = \sum_{i=1}^{n} \alpha_i \mathrm{k}(t, x_i)$. As I am using gaussian kernel the kernel function will be:

     $$\mathrm{k}(x, x') = \exp(\frac{-||x - x'||^2}{2\sigma^2})$$

     and for predictions I will be doing as following:

     $$\mathrm{k(t, x_i) = K_{pred,i}}$$
     $$\mathrm{Then,\ \hat{y} = K_{pred}}\alpha$$
     $$\mathrm{where,\ K_{pred}\ is\ 1xn\ and\ }\alpha\mathrm{\ is\ nx1\ matrix.}$$

   - Coming to algorithm as we know for kernel 'K' will be an nxn matrix with each element being values of kernel function for a data point to every data point in training data. First I initiated with creating class called `KernelRegression` because there are many value which will be same for multiple functions. So, first I took input as variance as for Gaussian kernel my hyperparameter will be variance and wrote function for Gaussian kernel function which is `gaussian_kernel_function` in my code. Then I wrote code for creating kernel matrix in function called `kernel_matrix`. Kernel matrix will form by traversing through all elements in matrix and finding there values from kernel function and at the end of this function I calculate for α using `numpy.linalg.solve` which will solve equation $K\alpha = Y$ for α which will be `self.alpha` in code. After finding α we can do predictions which will be done in function `predicts_using_kernel`. For each value to be predicted I found K_{pred} as mentioned above which is `kernel_for_prediction` in code, which will contain list of values of $k(x_{\text{test}}, x_i)$ where x_{test} will be one of input data to be predicted. And all the predicted values will be stored in `predicted_values` variable which will be given as output. In code same will be:



```python
# Algorithm for Polynomial Kernel regression
class KernelRegression:
    def __init__(self, variance):
        self.variance = variance

    def gaussian_kernel_function(self, x1, x2):
        return np.exp(-np.dot(x1 - x2, x1 - x2)/(2*(self.variance**2)))

    def kernel_matrix(self, X, y):
        data_points, features = X.shape
        kernel = np.zeros((data_points, data_points))
        for i in range(data_points):
            for j in range(data_points):
                kernel[i][j] = self.gaussian_kernel_function(X[i], X[j])
        self.alpha = np.linalg.solve(kernel, y)

    def predicts_using_kernle(self, test_data, X):
        test_data_points, m_test = test_data.shape
        data_points, features = X.shape
        kernel_for_prediction = np.zeros(data_points)
        predicted_values = np.zeros(test_data_points)

        for j in range(test_data_points):
            for i in range(data_points):
                kernel_for_prediction[i] = self.gaussian_kernel_function(test_data[j], X[i])
            predicted_values[j] = np.matmul(kernel_for_prediction, self.alpha)
        return predicted_values
```

3. **Predictions and Argue.**

   - Sum of squared error for kernel regression using Gaussian kernel is 876.04 which is the least compared to all other algorithms which I have used. All other algorithm gave error similar to 6600.
   - The Gaussian kernel is better than the standard least squares regression because if data is not in linear pattern then the standard least squares regression will fail to capture data pattern on the other hand kernel regression using Gaussian kernel will map the data in infinite dimension and in higher dimension data will be in linear pattern which will be helpful in capturing data pattern for better predictions of unseen data and eventually giving better results.