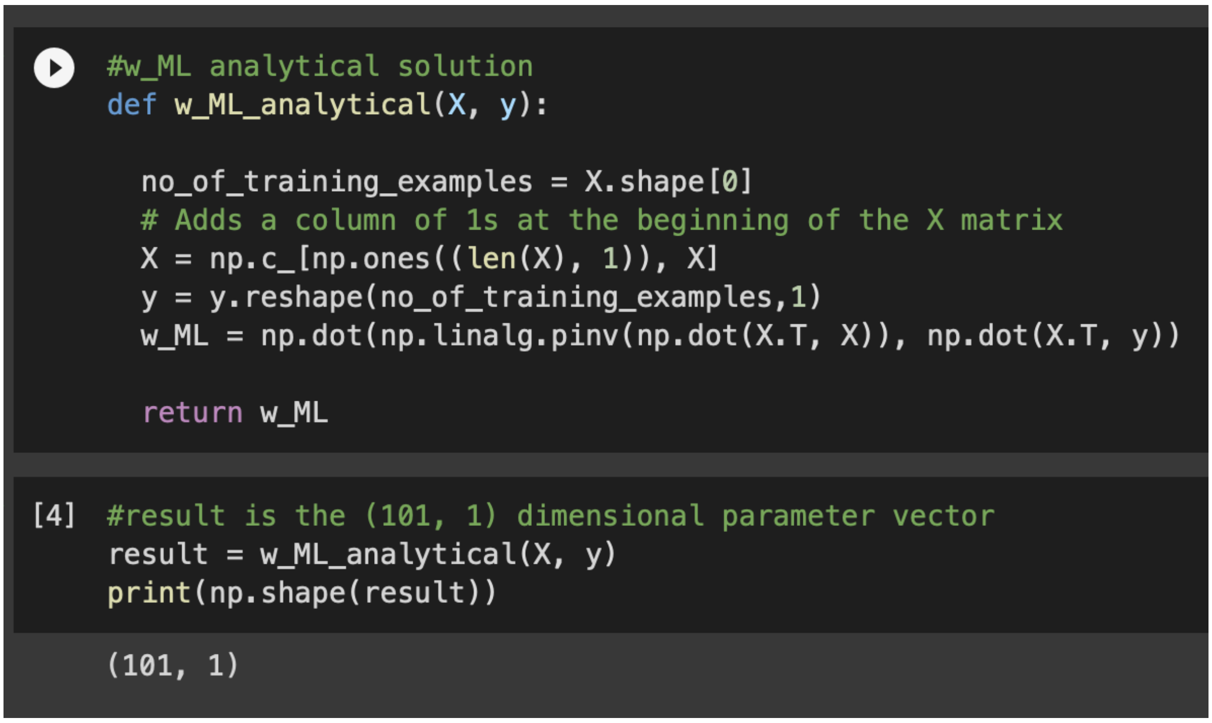
Question No. 2

i)

I implemented the following function which computes the least squares solution “w_ml” using the analytical solution.



```
#w_ML analytical solution
def w_ML_analytical(X, y):

  no_of_training_examples = X.shape[0]
  # Adds a column of 1s at the beginning of the X matrix
  X = np.c_[np.ones((len(X), 1)), X]
  y = y.reshape(no_of_training_examples,1)
  w_ML = np.dot(np.linalg.pinv(np.dot(X.T, X)), np.dot(X.T, y))

  return w_ML
```

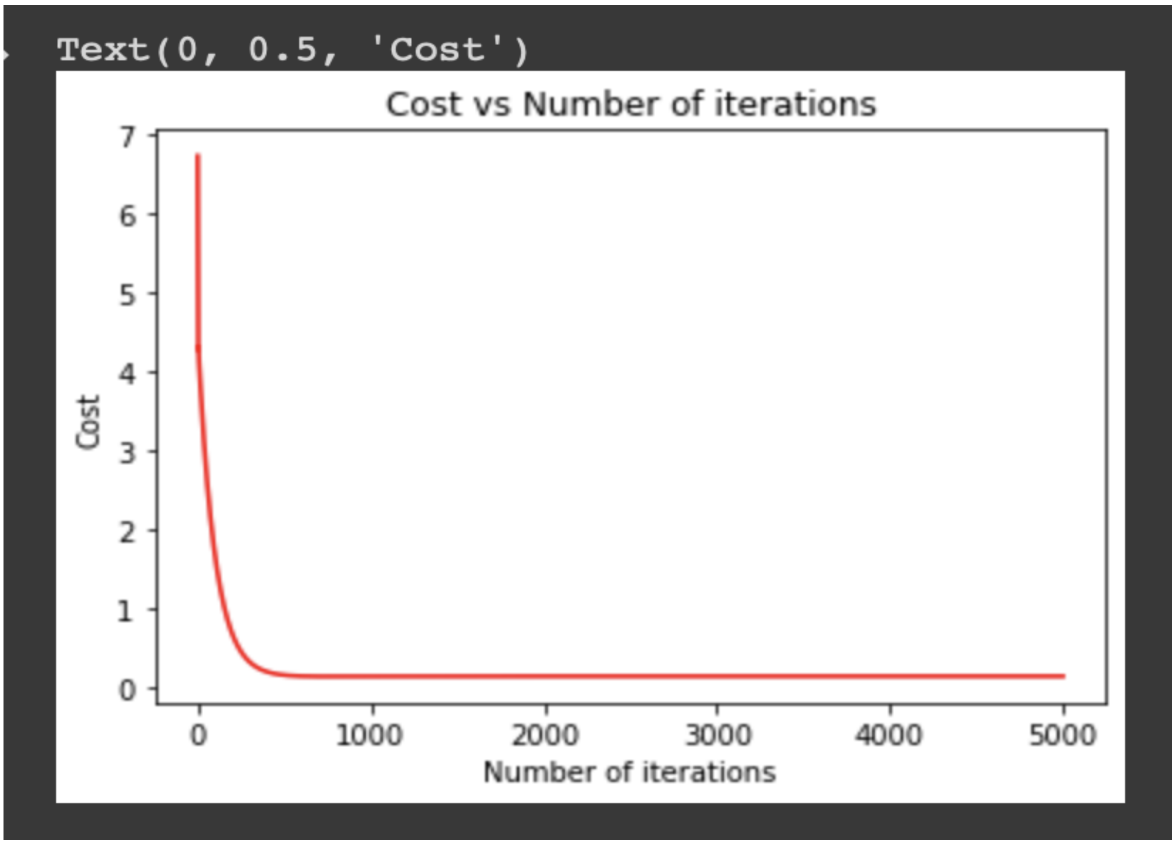
```
[4] #result is the (101, 1) dimensional parameter vector
    result = w_ML_analytical(X, y)
    print(np.shape(result))
```

```
(101, 1)
```

I used the “numpy.linalg.pinv” function to compute the (Moore-Penrose) pseudo-inverse.

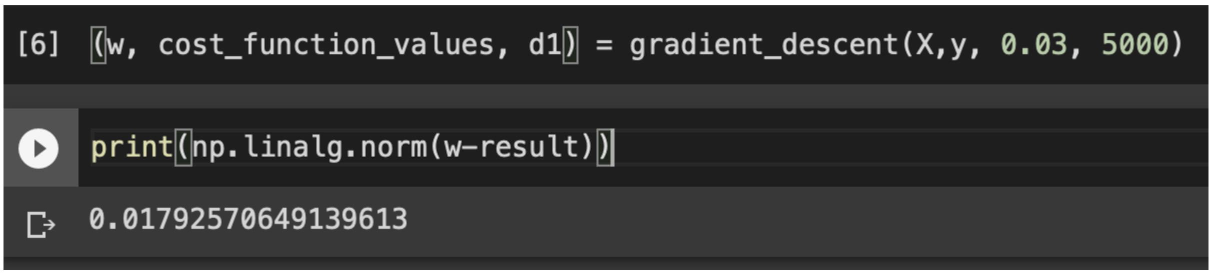
ii)

I implemented the gradient descent algorithm to solve the least squares algorithm. Please refer to the function “gradient_descent()” in my code. I used learning rate = 0.03 and ran the gradient descent algorithm for 5000 iterations. After every step of gradient descent the cost function value decreased. Please refer to the following plot. It is clear from the plot that initially the cost function value decreased very rapidly. And after approximately 500 iterations we were able to reach very close to the minima.



```
Text(0, 0.5, 'Cost')
```

Please refer to the screenshot below. Here ‘w’ is the parameter vector obtained after running gradient descent with learning rate = 0.03 and ‘result’ is the analytical solution ‘w_ml’. I have calculated the norm of (w-result) which captures the distance between them. The distance is very close to zero.

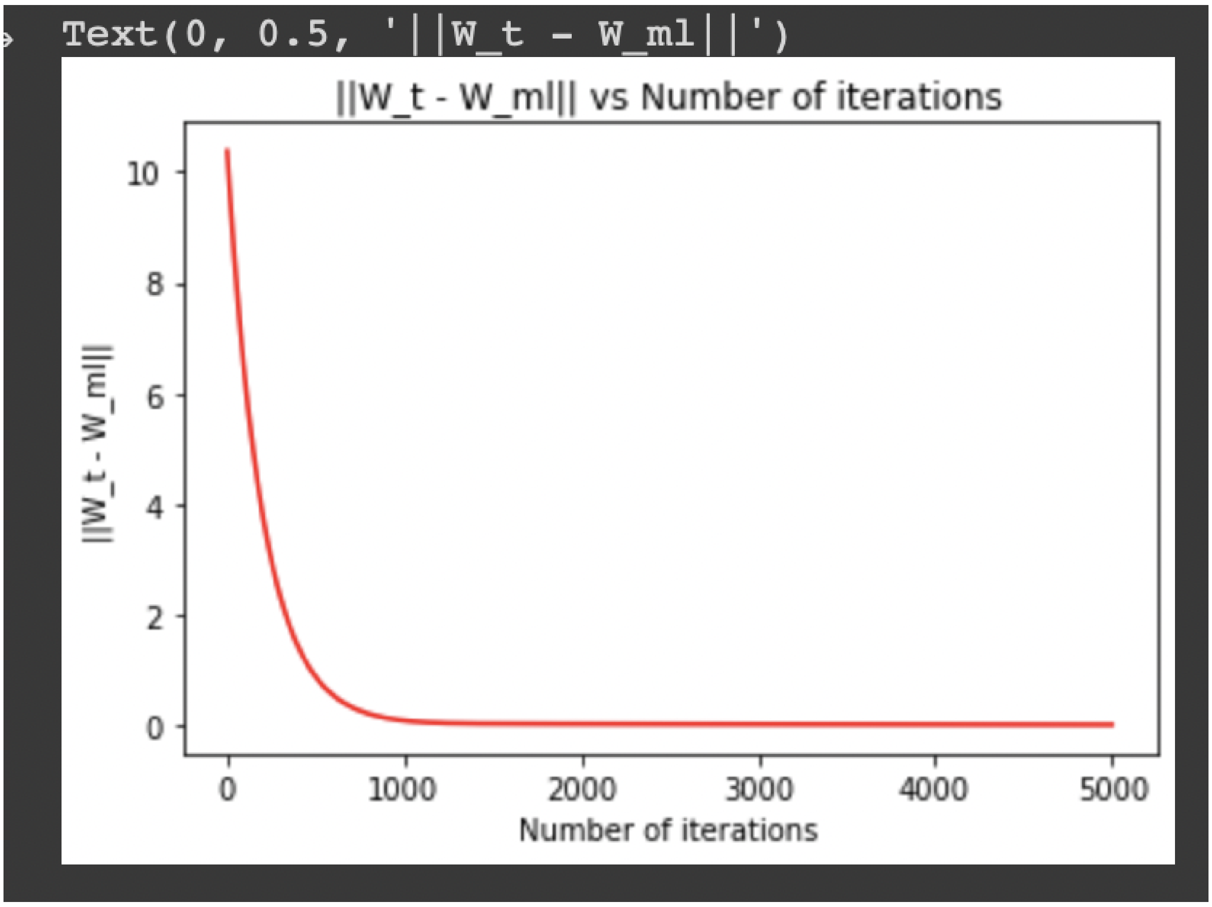


```
[6] (w, cost_function_values, d1) = gradient_descent(X,y, 0.03, 5000)

print(np.linalg.norm(w-result))

0.01792570649139613
```

The cost function in linear regression is a convex function. It has a single global minimum. After every step of gradient descent we move one step closer to the minima. Hence after every iteration the distance between ‘w_t’ and ‘w_ml’ decreases. It is clear from the below plot as well. After approximately 1,000 iterations the distance is almost zero.



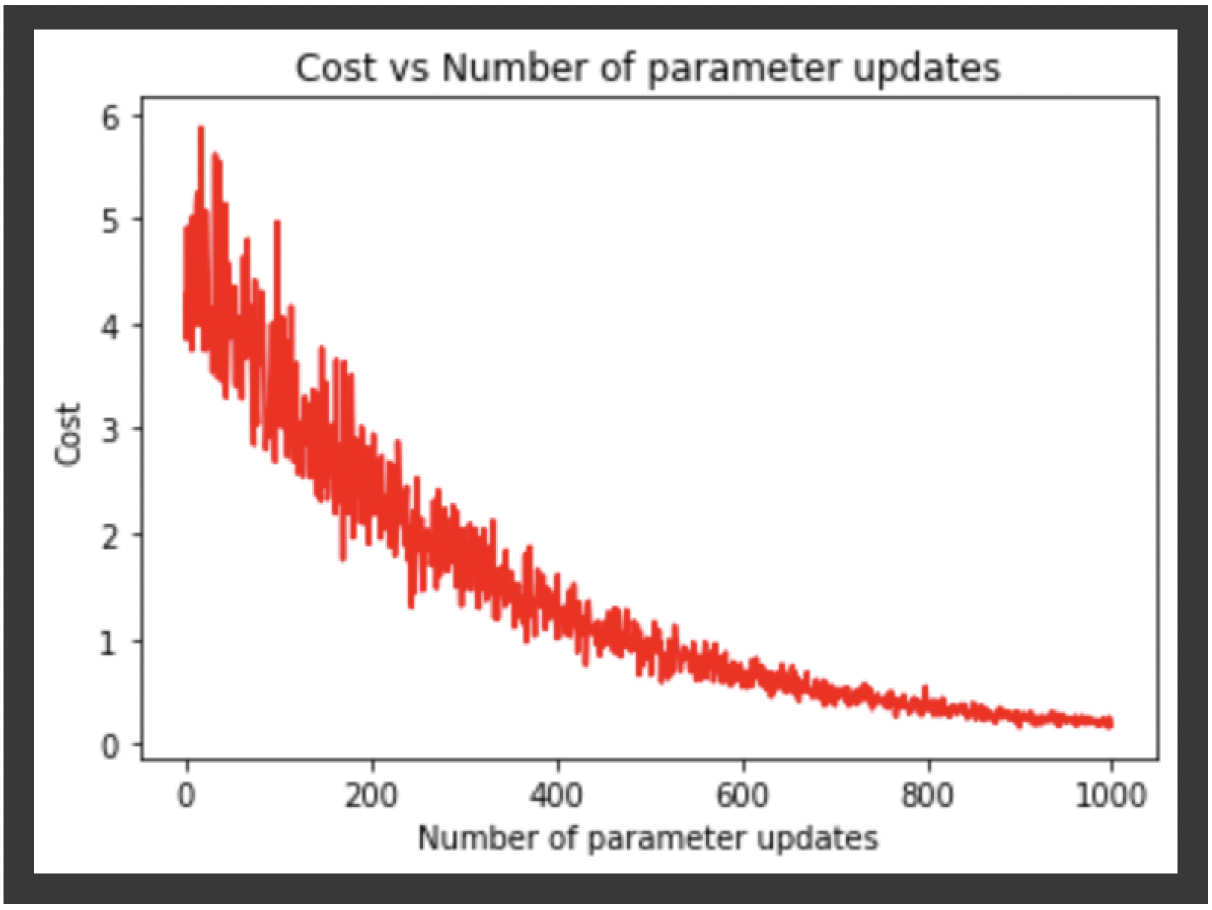
```
Text(0, 0.5, '||W_t - W_ml||')
```

iii)

Implemented the stochastic gradient descent algorithm using batch size of 100. Please refer to the following two functions in my code - mini_batch_generation() and mini_batch_gradient_descent().

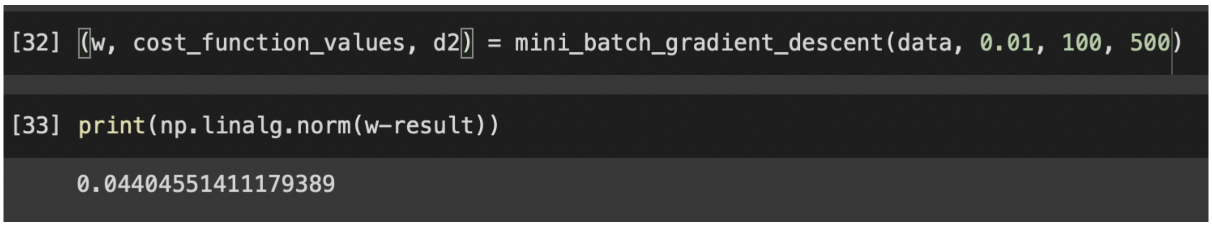
mini_batch_generation() : shuffles the data and creates random mini batches.
mini_batch_gradient_descent() : implements the mini batch gradient descent algorithm.

I used learning rate = 0.01 and batch size = 100. I ran the mini batch gradient descent algorithm for 500 iterations. In mini batch gradient descent we use only a subset of the training examples(here we have used batch size = 100) and update the parameters. So after every parameter update the cost function value does not decrease which is also clear from the below plot. In the case of gradient descent after every parameter update the cost function value decreases. It is clear from the plot that after 1000 parameter updates(10 iterations) we were able to reach very close to the minimum.



In the case of gradient descent we needed approximately 500 iterations to reach very close to the minimum. Mini Batch gradient descent needed only 10 iterations. So mini batch gradient descent provides much faster convergence than gradient descent.

Please refer to the screenshot below -

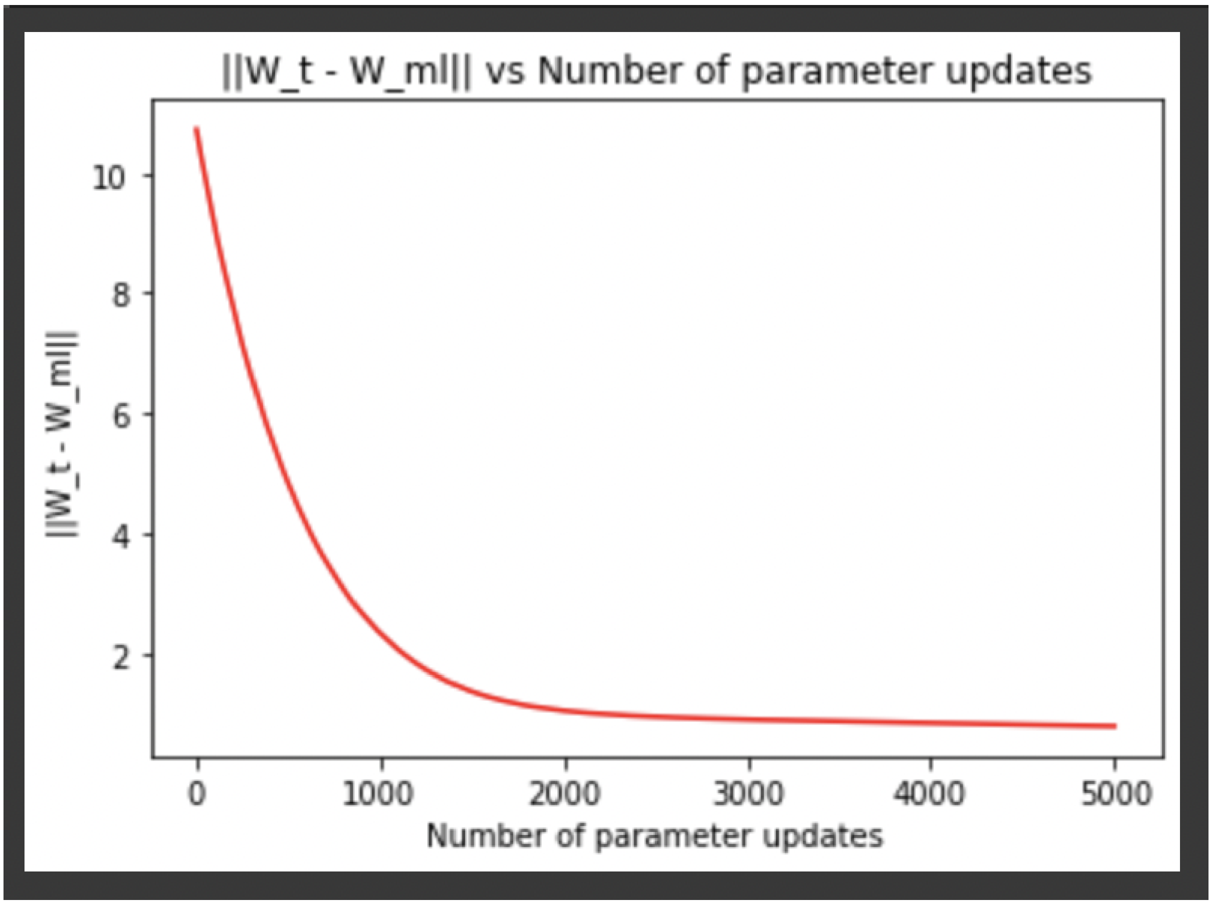


```
[32] (w, cost_function_values, d2) = mini_batch_gradient_descent(data, 0.01, 100, 500)

[33] print(np.linalg.norm(w-result))

     0.04404551411179389
```

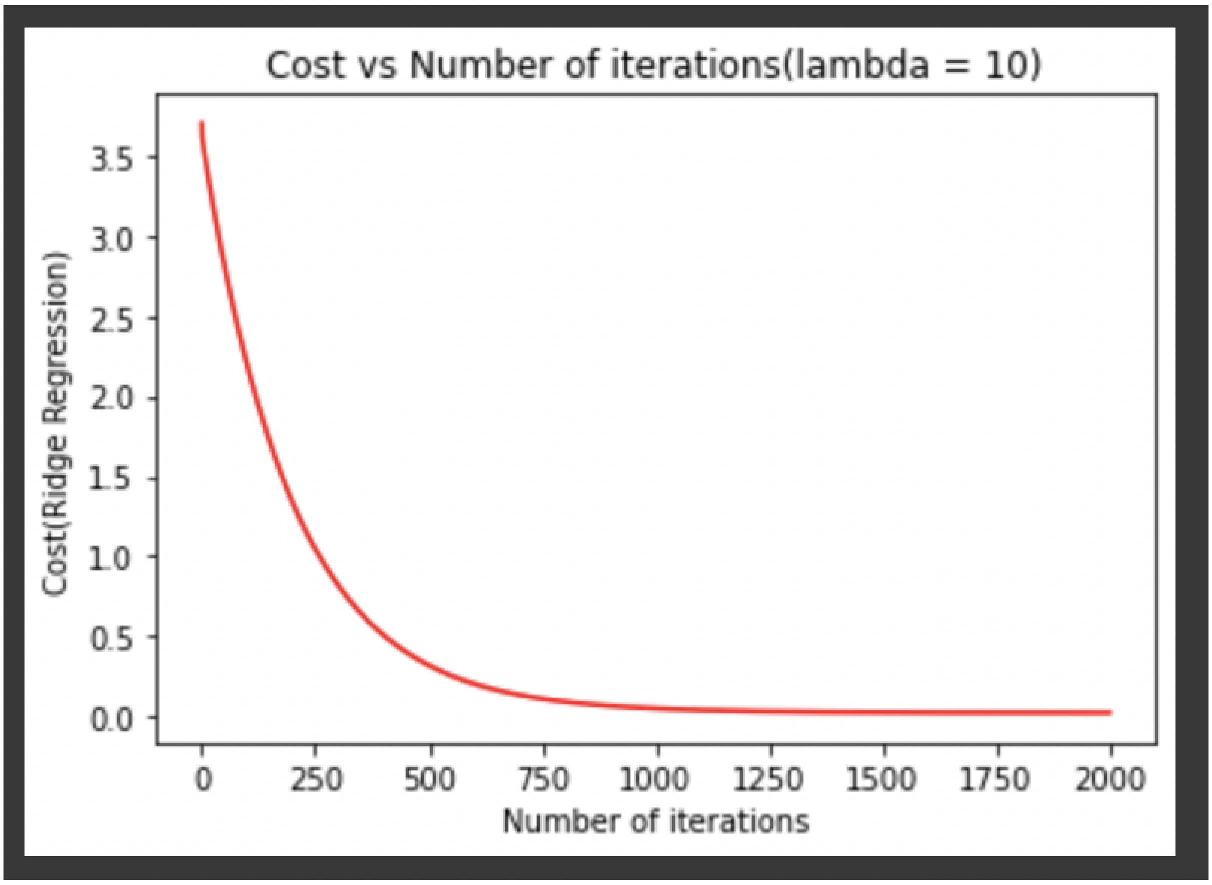
Here ‘w’ is the parameter vector obtained after running mini batch gradient descent with learning rate = 0.01 and ‘result’ is the analytical solution ‘w_ml’. I have calculated the norm of (w-result) which captures the distance between them. The distance is very close to zero. We know that the cost function in linear regression is a convex function. It has a single global minimum. So both gradient descent and mini batch gradient descent converge to the same solution(approximately) but using mini batch gradient descent we were able to reach the global minimum much faster.



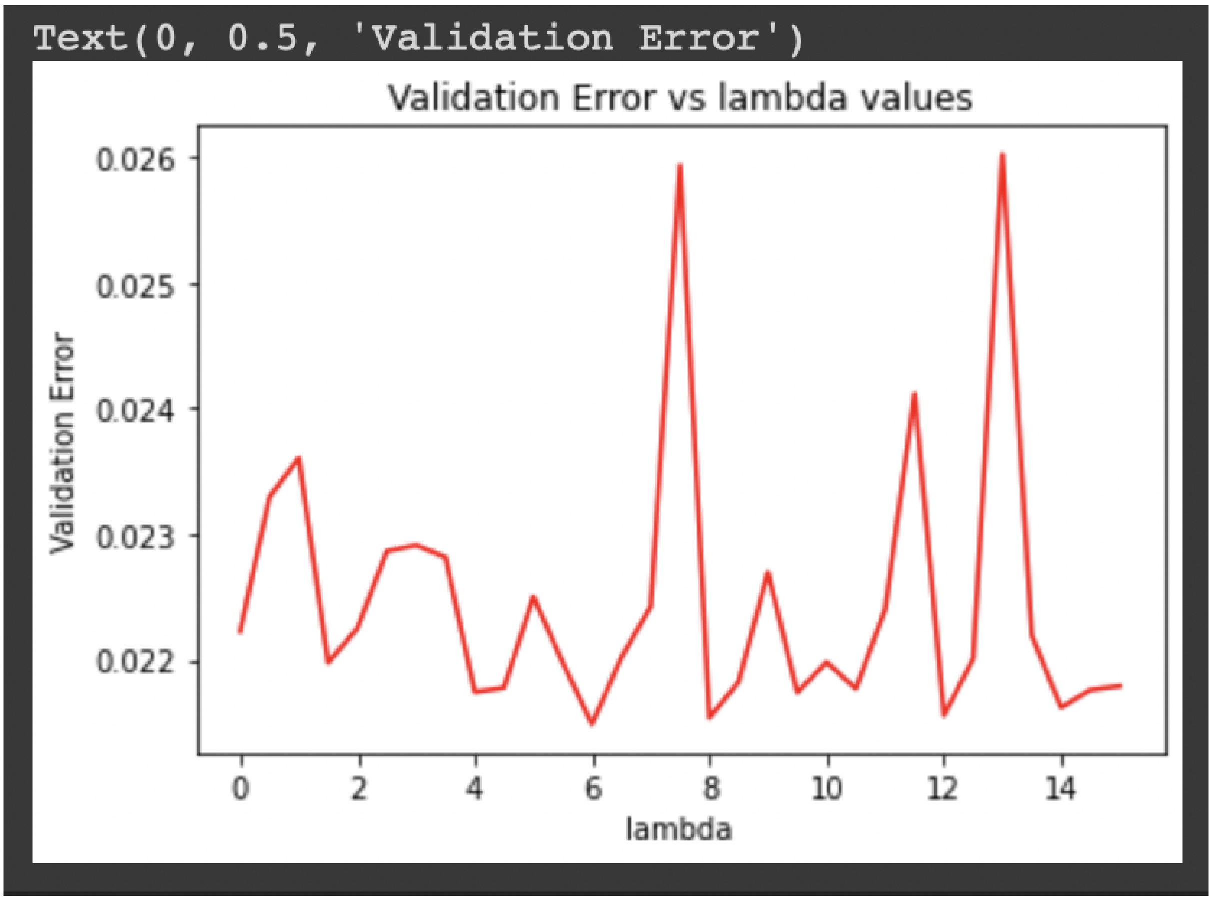
iv)

Implemented the gradient descent algorithm for ridge regression. Please refer to the following two functions in my code - ridge_cost_function() and ridge_gradient_descent().

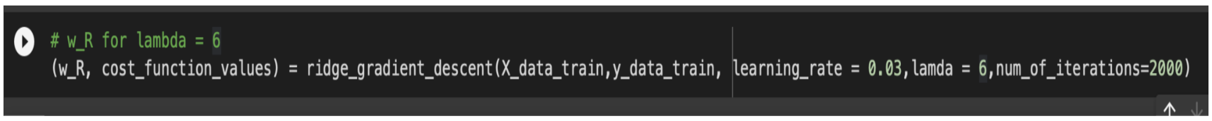
Please refer to the following plot. It shows that after every step of gradient descent the ridge regression cost function value decreases. I used learning rate = 0.03 and lambda = 10 for this particular plot.

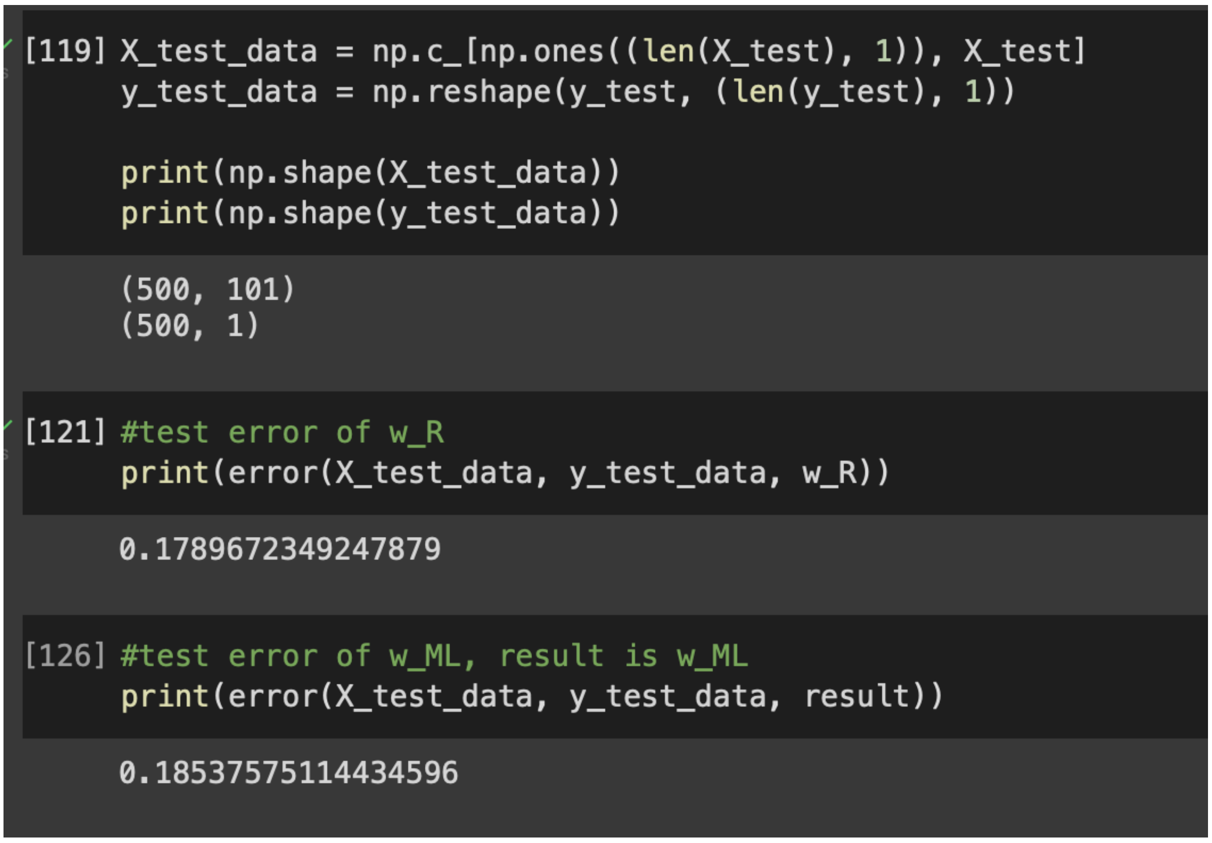


Dataset contains 10,000 data points. I kept 9,000 data points as training data and set aside 1,000 data points as validation data. I used numpy.linspace(0.0, 15.0, num=31) to generate 31 numbers and used them as various choices of lambda. Please refer to the following plot. It presents validation error values for different choices of lambda.


```
Text(0, 0.5, 'Validation Error')
```

The validation error values are very close to each other for different lambda values. However, it is clear from the above plot that for lambda = 6, the validation error is minimum. Please refer to the screenshot below. It calculates ‘w_R’ for lambda = 6.


```
# w_R for lambda = 6
(w_R, cost_function_values) = ridge_gradient_descent(X_data_train,y_data_train, learning_rate = 0.03,lamda = 6,num_of_iterations=2000)
```



```
[119] X_test_data = np.c_[np.ones((len(X_test), 1)), X_test]
      y_test_data = np.reshape(y_test, (len(y_test), 1))

      print(np.shape(X_test_data))
      print(np.shape(y_test_data))

      (500, 101)
      (500, 1)

[121] #test error of w_R
      print(error(X_test_data, y_test_data, w_R))

      0.1789672349247879

[126] #test error of w_ML, result is w_ML
      print(error(X_test_data, y_test_data, result))

      0.18537575114434596
```

Please refer to the screenshot above, it presents test errors of w_R and w_ML. It is clear that w_R has lesser test error. Since lesser test error means better generalisation, in this case w_R gives better performance on the test data. So w_R is better for this particular problem.