ID5030 Assignment 1

Haran Rajkumar

ED15B020

1. Linear Regression with gradient descent

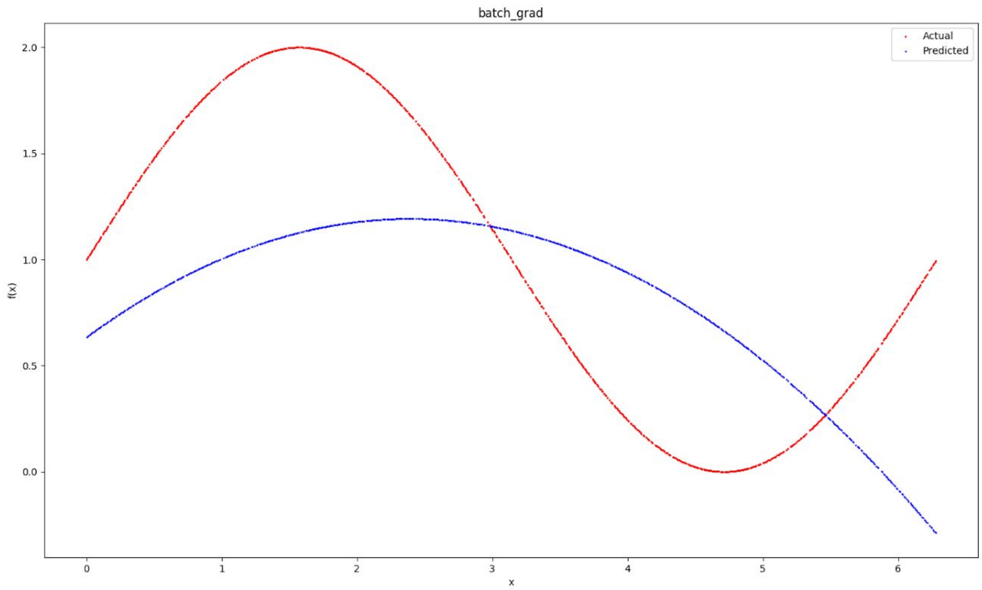
1. General Code

```
from matplotlib import pyplot as plt
import random as rd
import pandas as pd
from functions import *

#Generates a sample hypothesis
def sample_hypothesis1(data_size,noise=0):
	x = np.arange(0,2*np.pi,2*np.pi/data_size)
	y = np.sin(x)
	if noise!=0:
		print("noise added")
		mean=np.std(y)
		for i in rd.sample(range(0,len(x)),int(len(x)*noise)):
			# y[i]+=mean*3*(i%2-0.5)
			y[i]=np.cos(x[i])
	return x,y

#Weight Initialisation
def initialize_weights(number_of_features,val=10**-5):
	w=np.full(number_of_features,val)
	return w

#Extrapolates Polynomial Features from the given 2D function
def create_features(x,number_of_features):
	X = np.zeros((len(x),number_of_features))
	for j in range(0,number_of_features):
		for i,a in enumerate(x):
			X[i][j]=a**j
	return X

#Separate Dataset into training and test datasets
```

```
def separate_datasets(X,y,train_percent=0.8):
      train_indices =
rd.sample(range(0,len(X)),int(train_percent*len(X)))
      test_indices = [i for i in range(0,len(X)) if i not in
train_indices]
      x_training = np.ones((len(train_indices),X.shape[1]))
      x_test = np.ones((len(test_indices),X.shape[1]))
      y_test = np.ones(len(test_indices))
      y_training = np.ones(len(train_indices))
      # print(len(train_indices),X.shape)
      for idx,i in enumerate(train_indices):
            x_training[idx,:]=X[i,:]
            y_training[idx]=y[i]

      for idx,i in enumerate(test_indices):
            x_test[idx,:]=X[i,:]
            y_test[idx]=y[i]

      return x_training,y_training,x_test,y_test

def batch_grad(X_training,y_training,alpha,w,grad_threshold):
      number_of_features=len(w)
      data_size=len(y_training)
      grad_w=np.full(number_of_features,11)
      L=np.array([0,1])
      count = 1
      while (abs(L[count]-L[count-1])>grad_threshold):
            Y_training=np.dot(X_training,w.T)
            grad_w=np.dot(-1*2*(y_training-Y_training).T,X_training)
            w=w-alpha*grad_w
            count+=1

val=sum((y_training-np.dot(X_training,w.T))**2)/float(data_size)
            L=np.append(L,val)
      return w

def calc_error(X_test,y_test,w):
      return sum((y_test-np.dot(X_test,w.T))**2)/len(y_test)

#Main function
alpha=10**-6
number_of_features=4
data_size = 10000
grad_threshold=10**-4
```

```
x,y= sample_hypothesis1(data_size,0)
X= create_features(x,number_of_features)
X_training,y_training,X_test,y_test= separate_datasets(X,y)

w = initialize_weights(number_of_features)
w=batch_grad(X_training,y_training,alpha,w,grad_threshold)
mean_squared_error=calc_error(X_test,y_test,w)
print("Error: %f"%(mean_squared_error))
```

2. I used a 2D - curve as the sample hypothesis (function 'sample_hypothesis'). I extrapolated features by taking the polynomial values of the samples (function 'create_features()'). Therefore, I am I trying to fit the 2D curve with an nth order polynomial with n being the number of features. I plotted the 2D curve along with the predicted curve to see if it fits properly. I used the 'mean squared error'(function calc_error') as the test metric. It's a general test metric because it is normalised to the number of observations.



3. I used a criterion which computes the mean squared error and stops if the difference in errors of the current iteration to the previous one is lesser than a threshold (typically around 10^-3 - 10^-6). I tried various learning rates and found that above a certain a value the prediction blows up and below a certain value it takes a very long time to converge. For the above hypothesis, a learning rate between [10^-5,10^-6] gave reasonable results.

2. Stochastic gradient descent and variants

I used the same test I used for testing Batch Gradient Descent, i.e. check if the predicted curve fits the graph and check if the error is below a reasonable limit. I've provided the graphs which compares the actual function to the predicted function

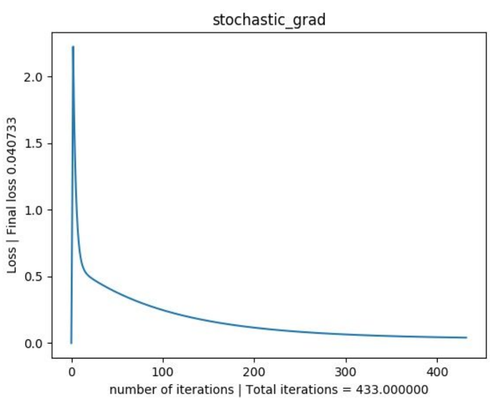
a. Stochastic gradient descent

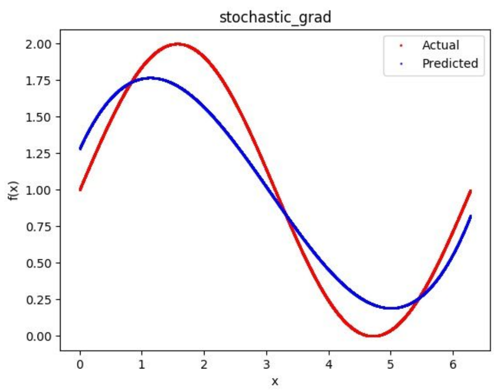
Formulae

Code

```
def stochastic_grad(X_training,y_training,alpha,w,grad_threshold):
      number_of_features=len(w)
      data_size=len(y_training)
      grad_w=np.full(number_of_features,11)
      L=np.array([0,1])
      count = 1
      while (abs(L[count]-L[count-1])>grad_threshold):
            for idx in range(0,data_size):
                  Y_training=np.dot(X_training[idx,:],w.T)
                  grad_w = 
np.dot(-1*2*(y_training[idx]-Y_training).T,X_training[idx,:])
                  w=w-alpha*grad_w
            count+=1
            
val=sum((y_training-np.dot(X_training,w.T))**2)/float(data_size)
            L=np.append(L,val)
      return w
```

Plot





Merits

- It's more accurate compared to Batch gradient descent
-

Demerits

- Slower compared to the algorithms due to the static learning rate

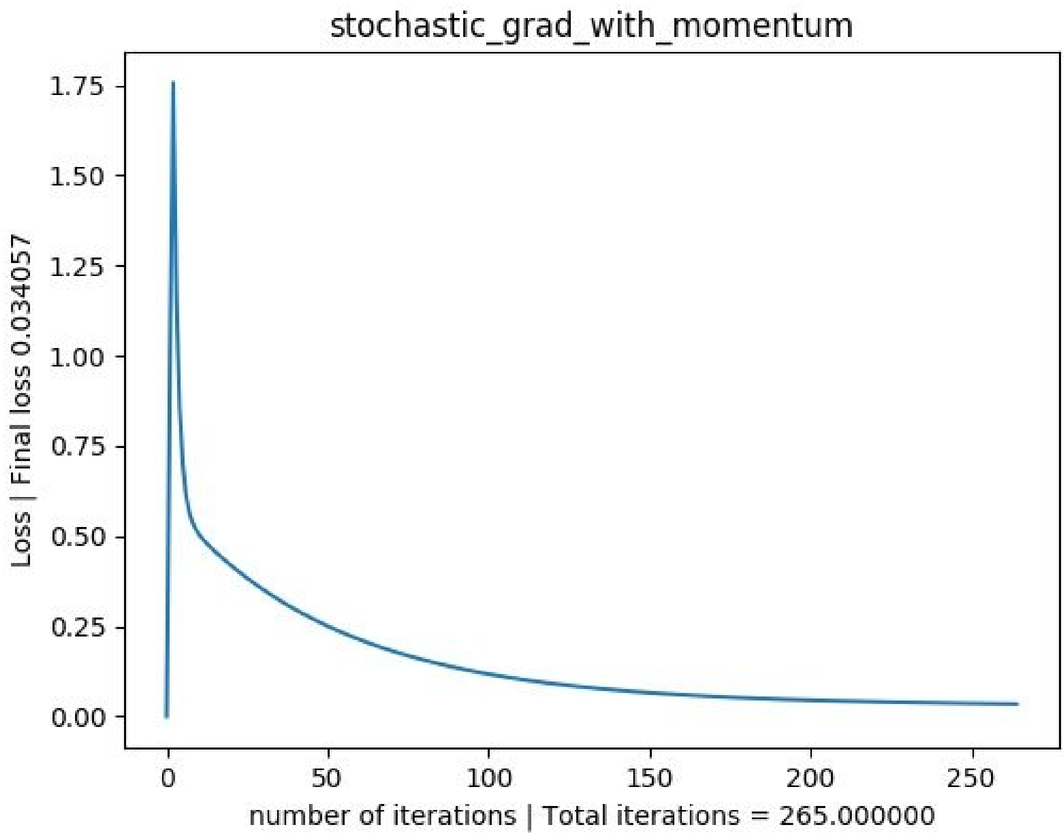
b. Stochastic gradient descent with momentum

Formulae

Code

```
def
stochastic_grad_with_momentum(X_training,y_training,alpha,w,grad_thresho
ld,eta=1):
	number_of_features=len(w)
	data_size=len(y_training)
	grad_w_dash=0
	grad_w=np.full(number_of_features,11)

	L=np.array([0,1])
	count = 1
	while (abs(L[count]-L[count-1])>grad_threshold):
		for idx in range(0,data_size):
			Y_training=np.dot(X_training[idx,:],w.T)
			grad_w =
np.dot(-1*2*(y_training[idx]-Y_training).T,X_training[idx,:])
			delta_w=-alpha*grad_w
			w=w + delta_w*eta - alpha*grad_w
		count+=1

val=sum((y_training-np.dot(X_training,w.T))**2)/float(data_size)
		L=np.append(L,val)
	return w
```

Plots



Merits

-

Demerits

-

c. Adagrad

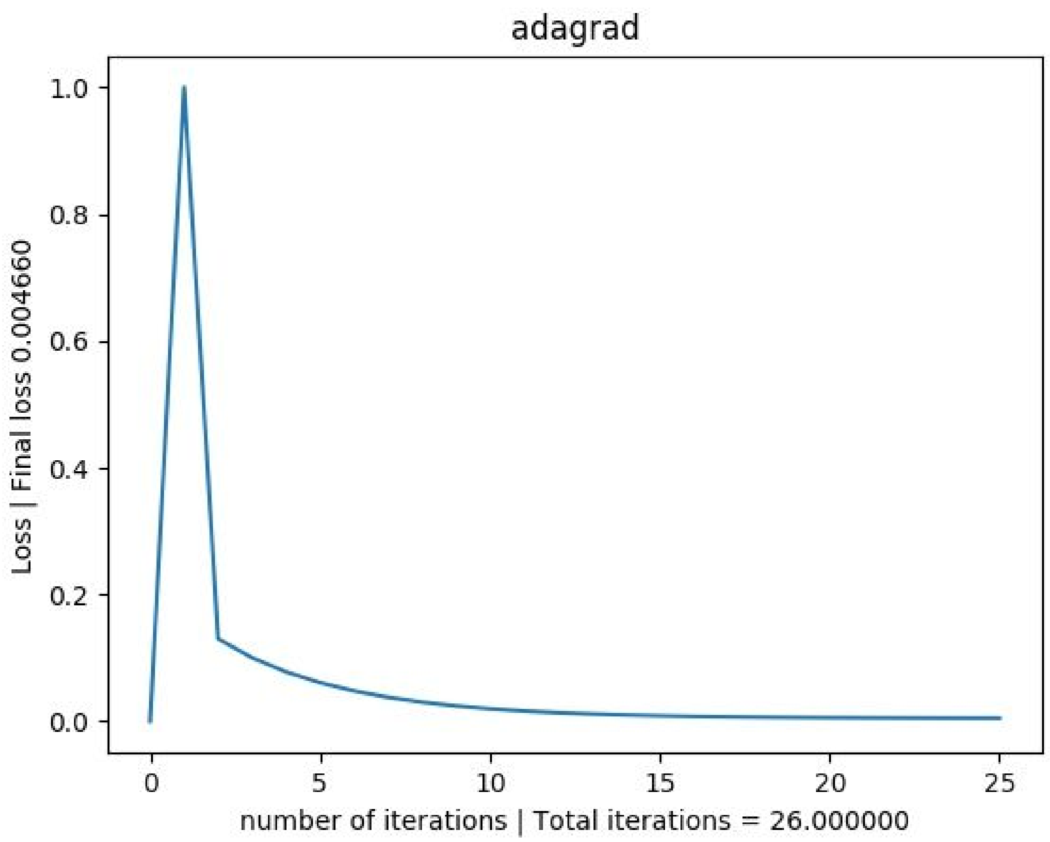
Formulae

Code

```
def adagrad(X_training,y_training,alpha,w,grad_threshold,eta=1):
      number_of_features=len(w)
      data_size=len(y_training)
      ada_g=np.zeros(number_of_features)
      grad_w=np.zeros(number_of_features)

      L=np.array([0,1])
      count = 1
      while (abs(L[count]-L[count-1])>grad_threshold):
            for idx in range(0,data_size):
```

```
                  Y_training=np.dot(X_training[idx,:],w.T)
                  grad_w =
np.dot(-1*2*(y_training[idx]-Y_training).T,X_training[idx,:])
                  ada_g+=grad_w**2
                  w = w - eta*grad_w/np.sqrt(ada_g)

            count+=1

val=sum((y_training-np.dot(X_training,w.T))**2)/float(data_size)
            L=np.append(L,val)
      return w
```

Plots



Merits

-

Demerits

-

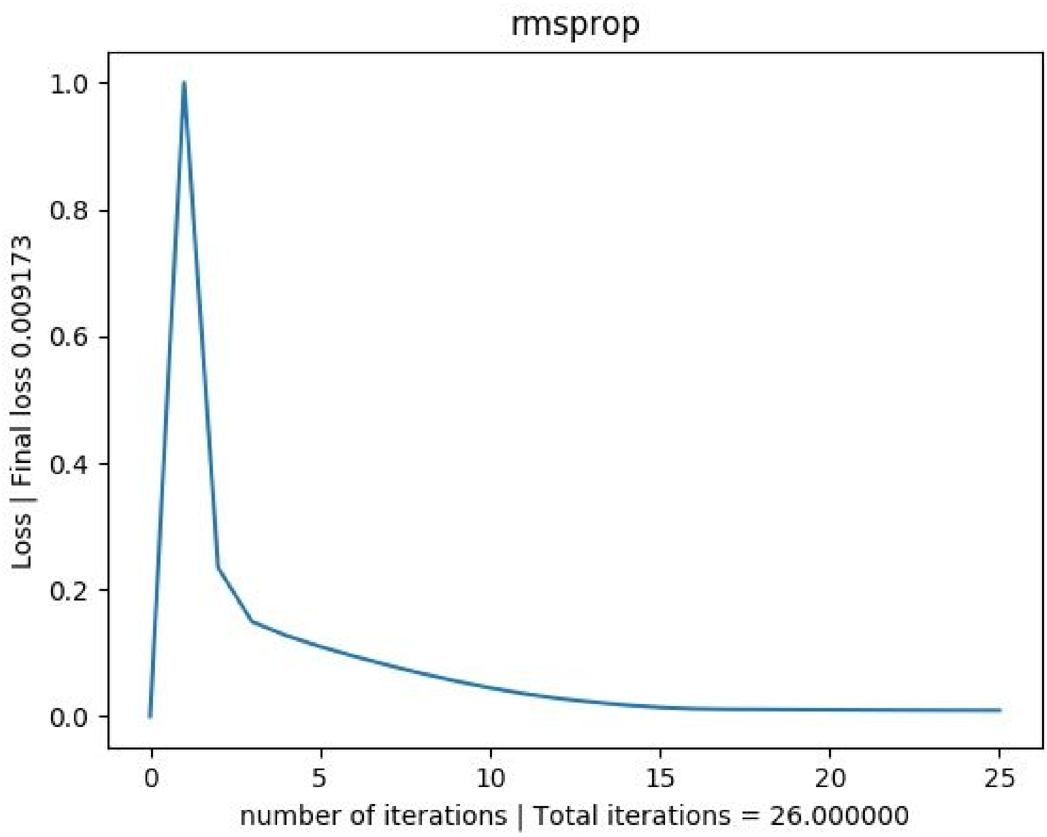
d. RMSprop

Formulae

Code

```
def
rmsprop(X_training,y_training,alpha,w,grad_threshold,eta=10**-3,gamma=0.
9):
      number_of_features=len(w)
      data_size=len(y_training)
      v=np.zeros(number_of_features)
      grad_w=np.zeros(number_of_features)

      L=np.array([0,1])
      count = 1
      while (abs(L[count]-L[count-1])>grad_threshold):
            for idx in range(0,data_size):
                  Y_training=np.dot(X_training[idx,:],w.T)
                  grad_w =
np.dot(-1*2*(y_training[idx]-Y_training).T,X_training[idx,:])
                  v=gamma*v+(1-gamma)*(grad_w**2)
                  w = w - eta*grad_w/np.sqrt(v)
            count+=1

val=sum((y_training-np.dot(X_training,w.T))**2)/float(data_size)
            L=np.append(L,val)
      return w
```

Plots



Merits

-

Demerits

-