

# Linear Regression

## 1. Linear Regression

```
class LinearRegression:
    """
    This class implements Linear Regression using both
    Batch Gradient Descent and Stochastic Gradient descent.
    Class attributes:
        W          : current set of weights
        W_arr       : list of weights at each iteration
        Cost        : current cost
        cost_arr    : list of costs at each iteration
    """

    def init_weights(self, s):
        """
        This method initializes the weight matrix
        as a column vector with shape = (X rows+1, 1)
        """
        np.random.seed(2)
        self.W = np.random.randn(s[1],1)
        self.W_arr = []
        self.cost_arr = []
        self.cost = 0

    def get_cost(self, X, y, W):
        """
        This function returns the cost with the given set of weights
        using the formula
        
$$J = \frac{1}{2m} \sum_{i=0}^{m-1} (h_w(x^i) - y^i)^2$$

        """
        total_cost = sum(np.square(np.matmul(X,W)-y.reshape(-1,1)))[0]
        return (0.5/X.shape[0])*total_cost

    def add_bias(self, X):
        """
        This function adds bias (a column of ones) to the feature vector X.
        """
        bias = np.ones((X.shape[0],1))
        return np.concatenate((bias,X), axis=1)

    def get_h_i(self, X, i, W):
        """
        This function returns the hypothesis of ith feature vector
        with the given weights W.
        
$$h_w(x^i) = \sum_{j=0}^n w_j x_j^i = x^i w$$

        """
        h_i = 0
        h_i = np.matmul(X[i].reshape(1,-1),W)
        return h_i[0][0]
```

```
def batch_grad_descent(self, X, y, alpha, max_iter):
    """
    This function implements the Batch Gradient Descent algorithm.
    It runs for multiple iterations until either the weights converge or
    iterations reach max_iter. At each iteration the weights are updated using
    the following rule
        repeat until convergence{
            
$$w_j^{f+1} = w_j^f - \frac{\alpha}{m} \sum_{i=1}^m (h_w(x^i) - y^i) x_j^i$$

        }
    """
    W_new = self.W.copy()
    for iteration in range(max_iter):
        temp = np.matmul(X, self.W) - y.reshape(-1,1)
        for j in range(X.shape[1]):
            W_new[j][0] = self.W[j][0] - (alpha/X.shape[0])*(sum(temp*X[:,j:j+1])[0])
        self.W = W_new.copy()
        self.cost_arr.append(self.get_cost(X, y, self.W))
        self.W_arr.append(self.W)
    return W_new
```

```
def stochastic_grad_descent(self, X, y, alpha, max_iter):
    """
    This function implements the Stochastic Gradient Descent algorithm.
    It runs for multiple iterations until either the weights converge or
    iterations reach max_iter. Weights are updated for every row of the
    training set.

        repeat until convergence{
            randomly shuffle the feature matrix rows
            for each feature vector  $x^i$  {
                update all weights  $j \rightarrow 0$  to  $n+1$ 
                
$$w_j^{f+1} = w_j^f - \alpha (h_w(x^i) - y^i) x_j^i$$

            }
        }
    """

    mat = np.concatenate((X,y.reshape(-1,1)), axis=1)
    for iteration in range(max_iter):
        W_new = self.W.copy()
        np.random.shuffle(mat)
        X = mat[:,0:3]
        y = mat[:,3]
        for i in range(X.shape[0]):
            temp = np.matmul(X[i,:],self.W) - y[i]
            for j in range(X.shape[1]):
                W_new[j][0] = self.W[j][0] - (alpha)*(temp[0]*X[i,j])
            self.W = W_new.copy()
        self.cost_arr.append(self.get_cost(X, y, self.W))
        self.W_arr.append(self.W)
    return self.W
```

```
def train(self, X, y, alpha, max_iter=100, option="batch"):
    """
    This function initiates the training process.
    It runs batch gradient descent by default and can also run
    Stochastic gradient descent if the argument is passed.

    returns the cost list which has costs at every training iteration.
    """
    # adding bias column to feature matrix X.
    X = self.add_bias(X)
    self.init_weights(X.shape)
    if option=="batch":
        self.batch_grad_descent(X,y,alpha,max_iter)
    elif option=="stochastic":
        self.stochastic_grad_descent(X,y,alpha,max_iter)
    self.cost = self.cost_arr[-1]
    return self.cost_arr
```

```
def test(self,X,W=""):
    """
    This function takes a feature matrix as test data and
    predicts the target values using the trained weights.

    returns the predicted target values.
    """
    if W=="":W = self.W

    X = self.add_bias(X)
    y_pred = np.ones(X.shape[0])
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            y_pred[i] += X[i][j]*W[j][0]
    return y_pred
```

```
if __name__ == "__main__":
    model = LinearRegression()

    # data input
    data = pd.read_csv("./data.csv", header=None)
    X = data.loc[:,0:1].values
    y = data.loc[:,2].values

    # data preprocessing (Normal scaling)
    mscaler = NormalScaler()
    mscaler.fit(X[:,0])
    X[:,0] = mscaler.transform(X[:,0])
    mscaler.fit(X[:,1])
    X[:,1] = mscaler.transform(X[:,1])

    # Training the model by choosing alpha and max_iter values.
    # gradient descent algorithm can be set as either 'batch' or 'stochastic'
    # in this function call.
    arr = model.train(X,y,0.19,250,"batch")

    print("weights: ",model.W)
    print("Total Cost: ",model.cost)
```

```

# visualization of cost function.
W_arr = np.array(model.W_arr)
res = 100

xx = np.linspace(np.min(W_arr[:,1])-10, np.max(W_arr[:,1])+10, res)
yy = np.linspace(np.min(W_arr[:,2])-10, np.max(W_arr[:,2])+10, res)
minw0 = W_arr[-1][0][0]

r = np.ndarray((res,res))
s = np.ndarray((res,res))
z = np.ndarray((res,res))

for i in range(res):
    for j in range(res):
        z[i][j] = model.get_cost(model.add_bias(X), y, np.array([minw0,xx[i],yy[j]]).reshape(-1,1))
        r[i][j] = xx[i]
        s[i][j] = yy[j]

# 3d surface plot of cost function and learning curve
ax = plt.axes(projection='3d')
ax.plot_surface(r, s, z,cmap='coolwarm')
ax.plot(W_arr[:,1], W_arr[:,2], model.cost_arr,c='red')
plt.show()

# 2d contour plot of cost function
plt.title("2d contour plot of cost function")
plt.contour(r,s,z.reshape(res,res),levels=25)
plt.scatter(W_arr[:,1].ravel(),W_arr[:,2].ravel(),c=model.cost_arr)
plt.show()

# 2d Line plot of cost vs iteration
plt.plot(model.cost_arr)
plt.show()

```

Results:

#### Batch Gradient Descent

weights:

```

w0 = 14.9047221
w1 = 0.36752656
w2 = 1.6965344

```

Total Cost: 9.403235190130703

#### Stochastic Gradient Descent

weights:

```

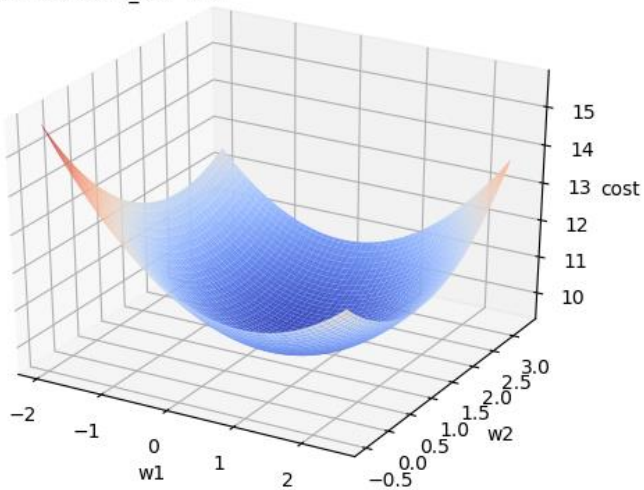
w0 = 14.95107574
w1 = 0.39873559
w2 = 1.75867644

```

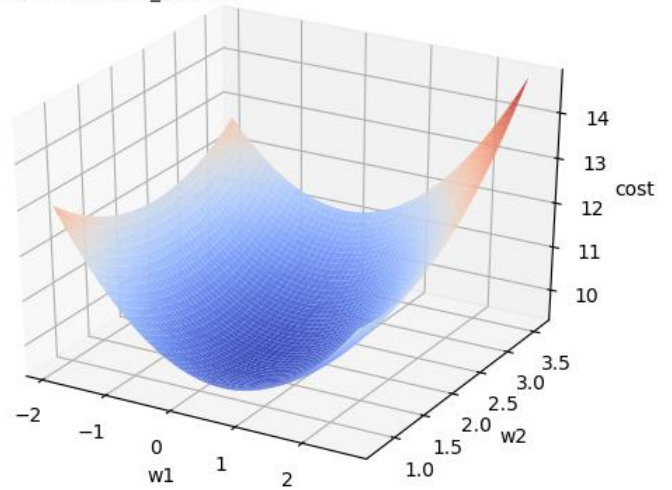
Total Cost: 9.407225072389698

Plots:

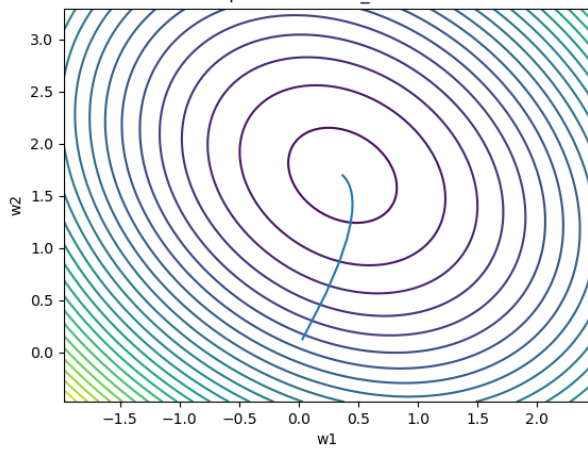
3D surface plot of cost function (batch)  
 $\alpha=0.26$  max\_iter=300



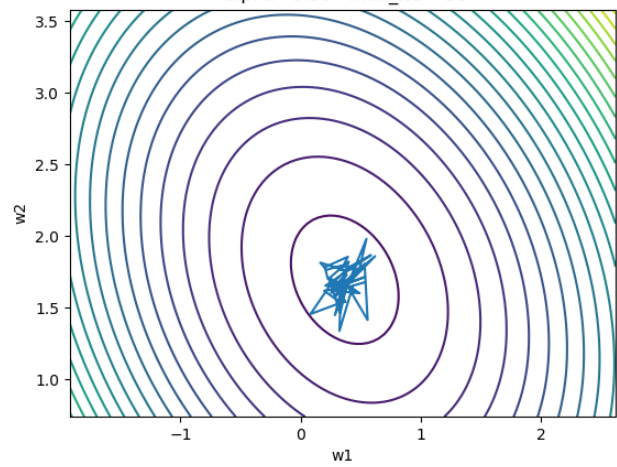
3D surface plot of cost function (stochastic)  
 $\alpha=0.007$  max\_iter=50



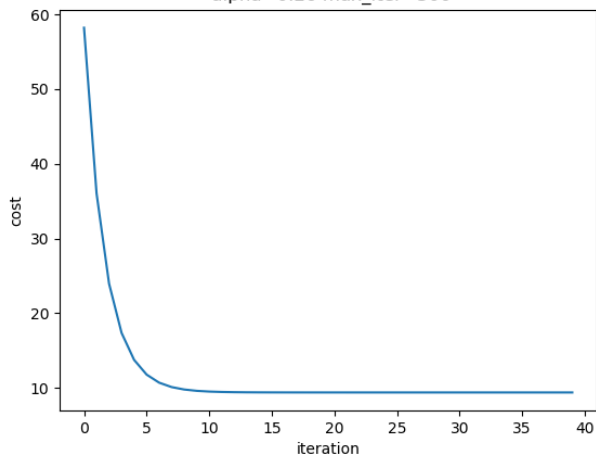
2d contour plot of cost function (batch)  
 $\alpha=0.26$  max\_iter=300



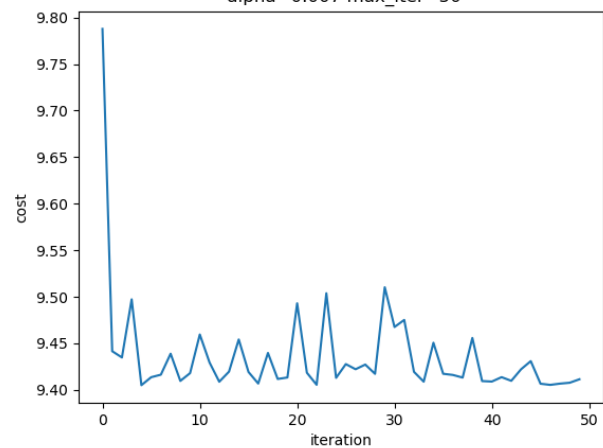
2d contour plot of cost function (stochastic)  
 $\alpha=0.007$  max\_iter=50



Cost Function vs iteration plot (batch)  
 $\alpha=0.26$  max\_iter=300



Cost Function vs iteration plot (stochastic)  
 $\alpha=0.007$  max\_iter=50



## 2. Ridge Regression

```
def get_cost(self, X, y, W):  
    """  
    This function returns the cost with the given set of weights  
    using the formula. Regularization term (sum of squares of weights)  
    is added to the cost.  
    
$$J = \frac{1}{2m} \sum_{i=0}^m (h_w(x^i) - y^i)^2 + \frac{\eta}{2} \sum_{i=0}^m w_j^2$$
  
    """  
    reg = 0  
    for i in range(1, W.shape[0]):  
        reg += W[i][0]**2  
    total_cost = sum(np.square(np.matmul(X, W) - y.reshape(-1, 1)))[0]  
  
    return (0.5/X.shape[0])*total_cost + 0.5*self.eta*reg
```

```
def batch_grad_descent(self, X, y, alpha, eta, max_iter):  
    """  
    This function implements the Batch Gradient Descent algorithm.  
    It runs for multiple iterations until either the weights converge or  
    iterations reach max_iter. At each iteration the weights are updated using  
    the following rule  
    repeat until convergence{  
        
$$w_j^{t+1} = w_j^t(1 - \eta\alpha) - \frac{\alpha}{m} \sum_{i=1}^m (h_w(x^i) - y^i)x_j^i$$
  
    }  
    """  
    self.eta = eta  
    for _ in range(max_iter):  
        W_new = np.ndarray(self.W.shape)  
        for j in range(X.shape[1]):  
            grad = 0  
            for i in range(X.shape[0]):  
                grad += (self.get_h_i(X, i, self.W) - y[i])*X[i][j]  
            W_new[j][0] = self.W[j][0]*(1-eta*alpha) - (alpha/X.shape[0])*grad  
        self.W = W_new.copy()  
        self.cost_arr.append(self.get_cost(X, y, self.W))  
        self.W_arr.append(self.W)  
        if len(self.W_arr)>1:  
            if sum(abs(self.W_arr[-2]-self.W_arr[-1]))<0.0001:  
                break  
    return W_new
```

```

def stochastic_grad_descent(self, X, y, alpha, eta, max_iter):
    """
    This function implements the Stochastic Gradient Descent algorithm.
    It runs for multiple iterations until either the weights converge or
    iterations reach max_iter. Weights are updated for every row of the
    training set.

    repeat until convergence{
        randomly shuffle the feature matrix rows
        for each feature vector  $x^i$  {
            update all weights  $j \rightarrow 0$  to  $n+1$ 
             $w_j^{t+1} = w_j^t(1 - \eta\alpha) - \alpha(h_w(x^i) - y^i)x_j^i$ 
        }
    }
    """
    mat = np.concatenate((X,y.reshape(-1,1)), axis=1)
    for _ in range(max_iter):
        W_new = self.W.copy()
        np.random.shuffle(mat)
        X = mat[:,0:3]
        y = mat[:,3]
        for i in range(X.shape[0]):
            temp = np.matmul(X[i,:],self.W) - y[i]
            for j in range(X.shape[1]):
                W_new[j][0] = self.W[j][0]*(1-eta*alpha) - (alpha)*temp[0]*X[i,j]
            self.W = W_new.copy()
        self.cost_arr.append(self.get_cost(X, y, self.W))
        self.W_arr.append(self.W)
        if len(self.W_arr)>1:
            if sum(abs(self.W_arr[-2]-self.W_arr[-1]))<0.0001:
                break
    return self.W

```

```

if __name__ == "__main__":
    model = RidgeRegression()

    # data input
    data = pd.read_excel("./data.xlsx", header=None)
    X = data.loc[:,0:1].values
    y = data.loc[:,2].values

    # data preprocessing (MinMax scaling)
    mscaler = NormalScaler()
    mscaler.fit(X[:,0])
    X[:,0] = mscaler.transform(X[:,0])
    mscaler.fit(X[:,1])
    X[:,1] = mscaler.transform(X[:,1])

    # Training the model by choosing alpha and max_iter values.
    # gradient descent algorithm can be set as either 'batch' or 'stochastic'
    # in this function call.
    alpha = 0.1
    eta = 0.1
    max_iter = 150
    algo = 'batch'

    arr = model.train(X,y,alpha,eta,max_iter,algo)

    print("weights: ",model.W)
    print("Total Cost: ",model.cost)

```



## Results:

### Batch Gradient Descent

weights:

$w_0 = 14.75716074$   
 $w_1 = 0.36824758$   
 $w_2 = 1.67956428$

Total Cost: 9.429059509401734

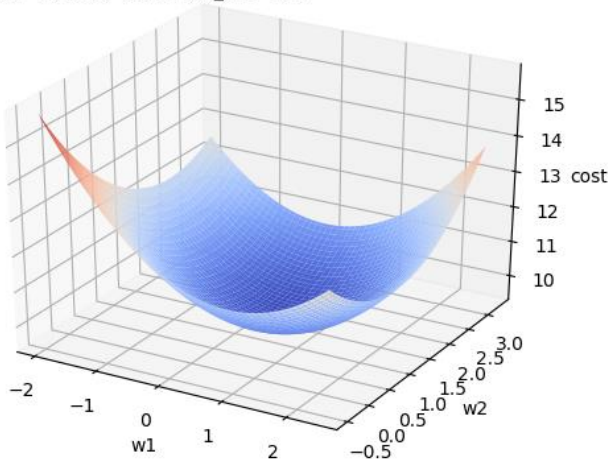
### Stochastic Gradient Descent

weights:

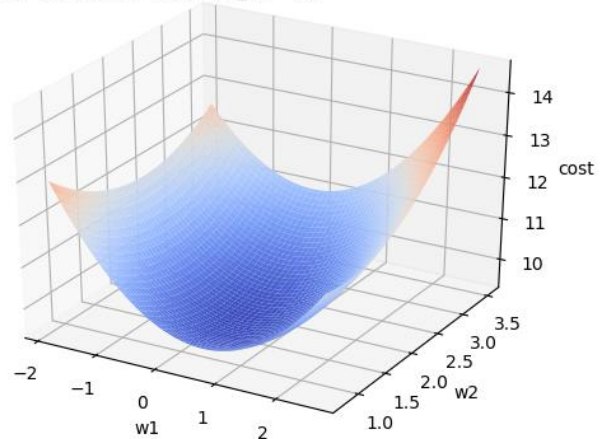
$w_0 = 14.8376096$   
 $w_1 = 0.18609906$   
 $w_2 = 1.76646287$

Total Cost: 9.421065885290048

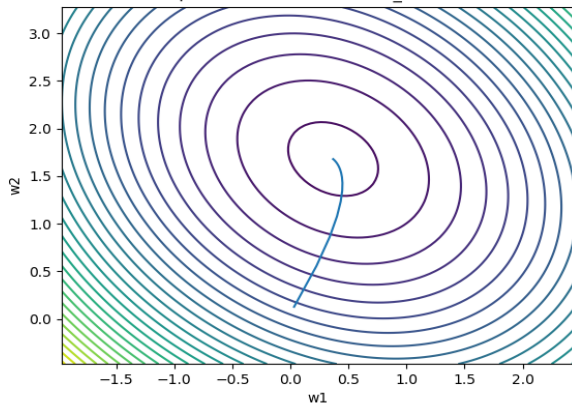
3D surface plot of cost function (batch)  
 $\alpha=0.26$   $\eta=0.01$   $\max\_iter=300$



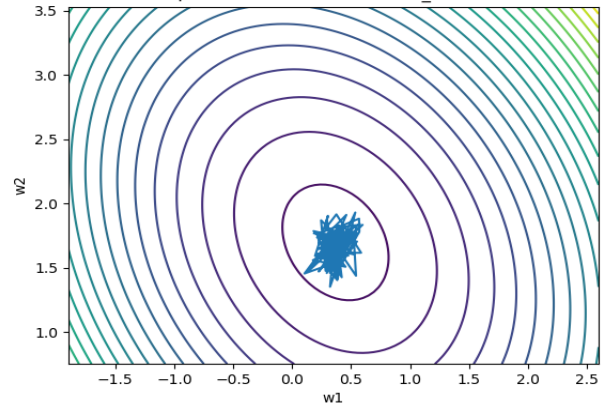
3D surface plot of cost function (stochastic)  
 $\alpha=0.0065$   $\eta=0.01$   $\max\_iter=150$



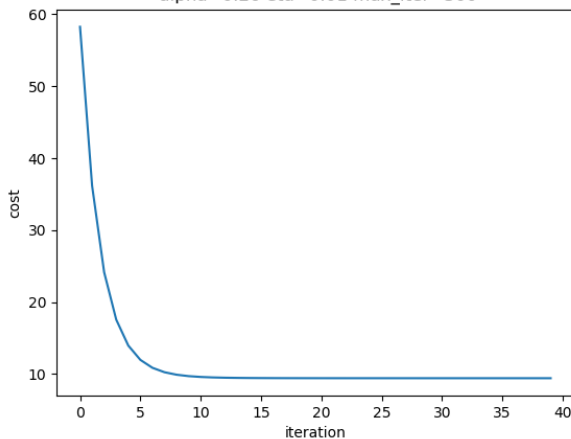
2d contour plot of cost function (batch)  
 $\alpha=0.26$   $\eta=0.01$   $\max\_iter=300$



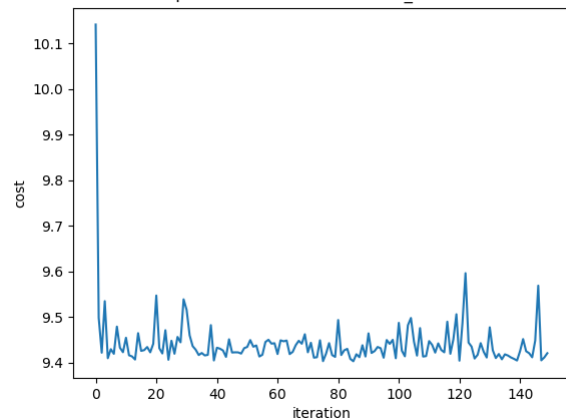
2d contour plot of cost function (stochastic)  
 $\alpha=0.0065$   $\eta=0.01$   $\max\_iter=150$



Cost Function vs iteration plot (batch)  
 $\alpha=0.26$   $\eta=0.01$   $\max\_iter=300$



Cost Function vs iteration plot (stochastic)  
 $\alpha=0.0065$   $\eta=0.01$   $\max\_iter=150$





### 3. Least Angle Regression

```
class LeastAngleRegression:
    """
    This class implements Linear Regression using both
    Batch Gradient Descent and Stochastic Gradient descent.
    Class attributes:
        W          : current set of weights
        W_arr       : list of weights at each iteration
        Cost        : current cost
        cost_arr     : list of costs at each iteration
    """

def get_cost(self, X, y, W):
    """
    This function returns the cost with the given set of weights
    using the formula
    
$$J = \frac{1}{2m} \sum_{i=0}^{m-1} (h_w(x^i) - y^i)^2 + \frac{\eta}{2} \sum_{j=1}^m |w_j|$$

    """
    total_cost = sum(np.square(np.matmul(X,W)-y.reshape(-1,1)))[0]
    reg = 0
    for i in range(1,W.shape[0]):
        reg += abs(W[i][0])
    return (0.5/X.shape[0])*total_cost + 0.5*self.eta*reg

def batch_grad_descent(self, X, y, alpha, eta, max_iter):
    """
    This function implements the Batch Gradient Descent algorithm.
    It runs for multiple iterations until either the weights converge or
    iterations reach max_iter. At each iteration the weights are updated using
    the following rule
    repeat until convergence{
        
$$w_j^{t+1} = w_j^t - \frac{\alpha}{m} \sum_{i=1}^m (h_w(x^i) - y^i) x_j^i - \frac{1}{2} \eta \alpha \operatorname{sgn}(w_j)$$

    }
    """
    self.eta = eta
    for _ in range(max_iter):
        W_new = np.ndarray(self.W.shape)
        for j in range(X.shape[1]):
            grad = 0
            for i in range(X.shape[0]):
                grad += (self.get_h_i(X, i, self.W) - y[i])*X[i][j]
            W_new[j][0] = self.W[j][0]-0.5*eta*alpha*np.sign(self.W[j][0]) - (alpha/X.shape[0])*grad
        self.W = W_new.copy()
        self.cost_arr.append(self.get_cost(X, y, self.W))
        self.W_arr.append(self.W)
        if len(self.W_arr)>1:
            if sum(abs(self.W_arr[-2]-self.W_arr[-1]))<0.0001:
                break
    return W_new
```

```

def stochastic_grad_descent(self, X, y, alpha, eta, max_iter):
    """
    This function implements the Stochastic Gradient Descent algorithm.
    It runs for multiple iterations until either the weights converge or
    iterations reach max_iter. Weights are updated for every row of the
    training set.

    repeat until convergence{
        randomly shuffle the feature matrix rows
        for each feature vector x^i {
            update all weights j -> 0 to n+1
             $w_j^{t+1} = w_j^t(1 - \eta\alpha) - \alpha(h_w(x^i) - y^i)x_j^i$ 
        }
    }
    """
    mat = np.concatenate((X, y.reshape(-1, 1)), axis=1)
    for _ in range(max_iter):
        W_new = self.W.copy()
        np.random.shuffle(mat)
        X = mat[:, 0:3]
        y = mat[:, 3]
        for i in range(X.shape[0]):
            temp = np.matmul(X[i, :], self.W) - y[i]
            for j in range(X.shape[1]):
                W_new[j][0] = self.W[j][0] - 0.5*eta*alpha*np.sign(self.W[j][0]) - (alpha/X.shape[0])*temp*X[i, j]
            self.W = W_new.copy()
        self.cost_arr.append(self.get_cost(X, y, self.W))
        self.W_arr.append(self.W)
        if len(self.W_arr)>1:
            if sum(abs(self.W_arr[-2]-self.W_arr[-1]))<0.0001:
                break
    return self.W

```

```

if __name__ == "__main__":
    model = LeastAngleRegression()

    # data input
    data = pd.read_csv("./data.csv", header=None)
    X = data.loc[:, 0:1].values
    y = data.loc[:, 2].values

    # data preprocessing (MinMax scaling)
    mscaler = MinMaxScaler()
    mscaler.fit(X[:, 0])
    X[:, 0] = mscaler.transform(X[:, 0])
    mscaler.fit(X[:, 1])
    X[:, 1] = mscaler.transform(X[:, 1])

    # Training the model by choosing alpha and max_iter values.
    # gradient descent algorithm can be set as either 'batch' or 'stochastic'
    # in this function call.
    alpha = 0.1
    eta = 0.01
    max_iter = 150
    algo = 'batch'

    arr = model.train(X, y, alpha, eta, max_iter, algo)
    print("weights: ", model.W)
    print("Total Cost: ", model.cost)

```

## Results:

### Batch Gradient Descent

weights:

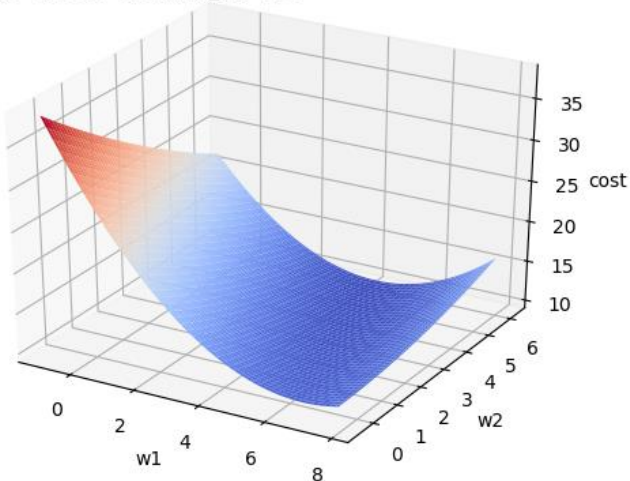
$w_0 = 8.70076276$

$w_1 = 8.70076276$

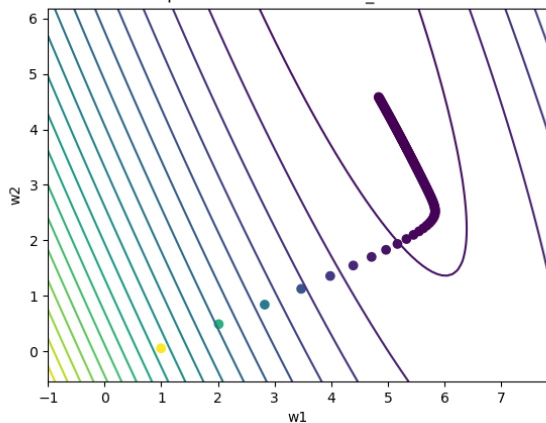
$w_2 = 4.58278454$

Total Cost: 9.80262939490788

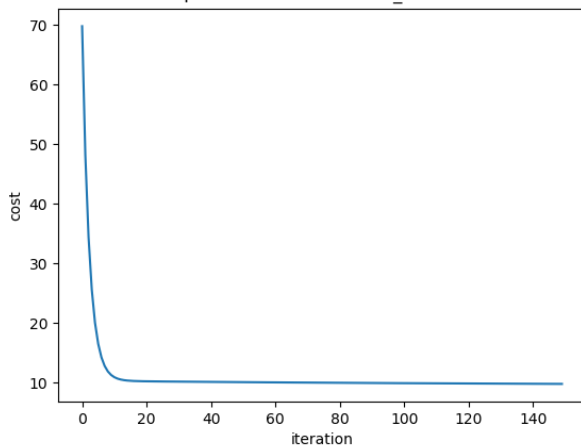
3D surface plot of cost function (batch)  
alpha=0.1 eta=0.01 max\_iter=150



2d contour plot of cost function (batch)  
alpha=0.1 eta=0.01 max\_iter=150



Cost Function vs iteration plot (batch)  
alpha=0.1 eta=0.01 max\_iter=150



### Stochastic Gradient Descent

weights:

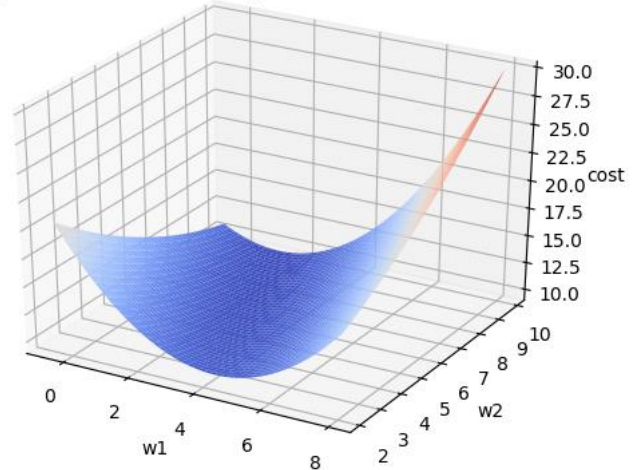
$w_0 = 10.11557396$

$w_1 = 1.71982718$

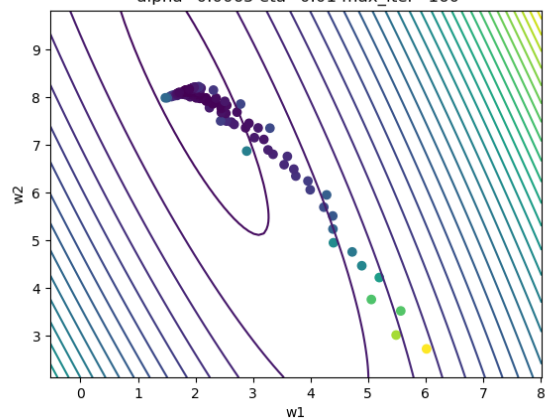
$w_2 = 8.10088578$

Total Cost: 9.42485150686

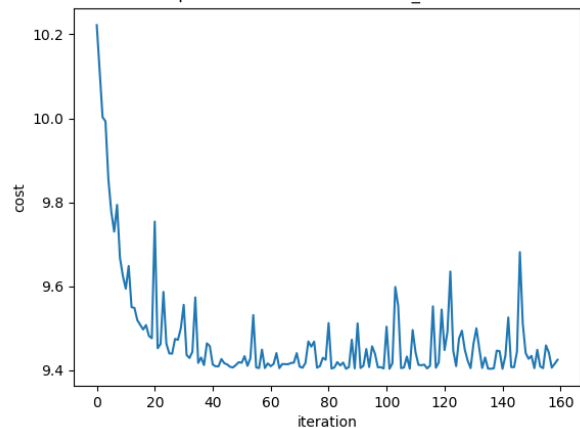
3D surface plot of cost function (stochastic)  
alpha=0.0065 eta=0.01 max\_iter=160



2d contour plot of cost function (stochastic)  
alpha=0.0065 eta=0.01 max\_iter=160



Cost Function vs iteration plot (stochastic)  
alpha=0.0065 eta=0.01 max\_iter=160



## 4. Vectorized Linear Regression

```
class LinearRegression_Vectorized:
    """
    This class implements Linear Regression using both
    Batch Gradient Descent and Stochastic Gradient descent.
    Class attributes:
        W          : current set of weights
        W_arr       : list of weights at each iteration
        Cost        : current cost
        cost_arr    : list of costs at each iteration
    """

    def train(self, X, y):
        """
        This function uses the vectorized version of linear regression
        and obtains the optimal weights with the given feature matrix
        and target values.
        
$$W = (X^T X)^{-1} X^T Y$$

        returns the optimal weights.
        """
        X = self.add_bias(X)
        self.init_weights(X.shape)
        self.W = np.matmul(np.matmul(np.linalg.inv(np.matmul(X.T, X)), X.T), y.reshape(-1, 1))
        self.cost = self.get_cost(X, y, self.W)
        return self.W

    def test(self, X):
        """
        This function takes a feature matrix as test data and
        predicts the target values using the trained weights.

        returns the predicted target values.
        """
        X = self.add_bias(X)
        return np.matmul(X, self.W)

if __name__ == "__main__":
    model = LinearRegression_Vectorized()

    # data input
    data = pd.read_excel("./data.xlsx", header=None)
    X = data.loc[:, 0:1].values
    y = data.loc[:, 2].values

    # data preprocessing (Normal Scaling)
    print(X[:, 0].mean())
    mscler = NormalScaler()
    mscler.fit(X[:, 0])
    X[:, 0] = mscler.transform(X[:, 0])
    mscler.fit(X[:, 1])
    X[:, 1] = mscler.transform(X[:, 1])

    # training the model
    arr = model.train(X, y)
    print("weights: ", model.W)
    print("Total Cost: ", model.cost)
```

Results:

Total Cost: 9.80262939490788

weights:

$w_0 = 14.90479943$

$w_1 = 0.36737803$

$w_2 = 9.403235170782$

3D surface plot of cost function

