Classical Machine Learning with TensorFlow

Machine learning is an area of computer science that involves research, development, and application of algorithms to make computing machines learn from data. The models learned by computing machines are used to make predictions and forecasts.

All of the machine learning problems are abstracted to the following equation in one form or another:

$$\mathbf{y = f(x)}$$

y is the output or target and **x** is the input or features. If **x** is a collection of features, we also call it a feature vector and denote with **X**. When we say model, we mean to find the function **f** that maps features to targets. Thus once we find **f**, we can use the new value of **x** to predict values of **y**.

Machine learning is centered around finding the function f that can be used to predict y from values of x.

The equation of the line is as follows:

$$\mathbf{y = mx + c}$$

We can rewrite the preceding simple equation as follows:

$$\mathbf{y = Wx + b}$$

Here, W is known as the weight and b is known as the bias.The machine learning problem can be stated as a problem of finding W and b from current values of X, such that the equation can be used to predict the values of y.

Regression analysis or regression modeling refers to the methods and techniques used to estimate relationships among variables. The variables that are input to regression models are called independent variables or predictors or features and the output variable from regression models is called dependent variable or target. Regression models are defined as follows:

$$\mathbf{Y = f(X, \beta)}$$

where Y is the target variable, X is a vector of features, and β is a vector of parameters.we use a very simple form of regression known as simple linear regression to estimate the parameter β.

In machine learning problems, we have to learn the model parameters **β0** and **β1** from the given data so that we have an estimated model to predict the value of **Y** from future values of **X**. We use the term **weight** for **β1** and **bias** for **β0** and represent them with **w** and **b** in the code, respectively.

Thus the model becomes as:

$$\mathbf{Y = X \times w + b}$$

Classification is one of the classical problems in machine learning. Data under consideration could belong to one or other classes, for example, if the images provided are data, they could be pictures of cats or dogs. Thus the classes, in this case, are cats and dogs. Classification means to identify or recognize the label or class of the data or objects under consideration. Classification falls under the umbrella of supervised machine learning. In classification problems, the training dataset is provided that has features or inputs and their corresponding outputs or labels. Using this training dataset, a model is trained; in other words, parameters of the model are computed. The trained model is then used on new data to find its correct labels.

Classification problems can be of two types: binary class or multiclass. Binary class means the data is to be classified in two distinct and discrete labels, for example, the patient has cancer or the patient does not have cancer, the images are of cats or dogs. Multiclass means the data is to be classified among multiple classes, for example, an email classification problem will divide emails into social media emails, work-related emails, personal email, family-related emails, spam emails, shopping offer emails, and so on. Another example would be the example of pictures of digits; each picture could be labeled between 0 to 9, depending on what digit the picture represents.

1. Simple linear regression
    a. Let's practice learning the simple linear regression model using TensorFlow.
    b. We will use generated datasets
    c. **Data preparation**
        i. To generate the dataset, we use the **make_regression** function from the **datasets** module of the **sklearn** library

```
from sklearn import datasets as skds
X, y = skds.make_regression(n_samples=200,
                            n_features=1,
                            n_informative=1,
                            n_targets=1,
                            noise = 20.0)
```

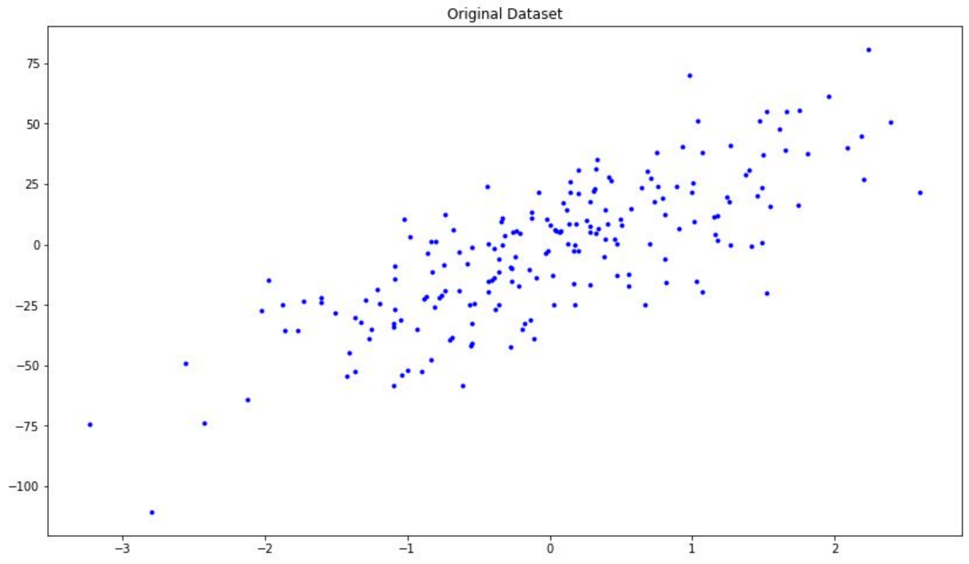
        ii. This generates a dataset for regression with 200 sample values for one feature and one target each, with some noise added. As we are generating only one target, the function generates **y** with a one-dimensional NumPy Array; thus, we reshape **y** to have two dimensions:

```
if (y.ndim == 1):
    y = y.reshape(len(y),1)
```

        iii. We plot the generated dataset to look at the data with the following code:

```
import matplotlib.pyplot as plt
plt.figure(figsize=(14,8))
plt.plot(X,y,'b.')
plt.title('Original Dataset')
plt.show()
```

iv. We get the following plot. As the data generated is random, you might get a different plot:



v. Now let's divide the data into train and test sets:

```
X_train, X_test, y_train, y_test = skms.train_test_split(X, y,
                                                 test_size=.4,
                                                 random_state=123)
```

d. Building a simple regression model

i. To build and train a regression model in TensorFlow, the following steps are taken in general:

1. Defining the inputs, parameters, and other variables
    a. Before we get into building and training the regression model using TensorFlow, let's define some important variables and operations. We find out the number of output and input variables from **X_train** and **y_train** and then use these numbers to define the **x**(*x_tensor*), **y** (*y_tensor*), *weights* (**w**), and *bias* (**b**):

```
num_outputs = y_train.shape[1]
num_inputs = X_train.shape[1]

x_tensor = tf.placeholder(dtype=tf.float32,
                   shape=[None, num_inputs],
                   name="x")
y_tensor = tf.placeholder(dtype=tf.float32,
                   shape=[None, num_outputs],
                   name="y")

w = tf.Variable(tf.zeros([num_inputs,num_outputs]),
                dtype=tf.float32,
                name="w")
b = tf.Variable(tf.zeros([num_outputs]),
                dtype=tf.float32,
                name="b")
```

b.
   i. **x_tensor** is defined as having a shape of variable rows and **num_inputs** columns and the number of columns is only one in our example
   ii. **y_tensor** is defined as having a shape of variable rows and **num_outputs** columns and the number of columns is only one in our example
   iii. **w** is defined as a variable of dimensions **num_inputs** x **num_outputs**, which is 1 x 1 in our example
   iv. **b** is defined as a variable of dimension **num_outputs**, which is one in our example

2. Defining the model
   a. we define the model as **(x_tensor × w) + b**:

```
model = tf.matmul(x_tensor, w) + b
```

3. Defining the loss function
   a. we define the loss function using the **mean squared error (MSE)**. MSE is defined as follows:

$$\frac{1}{n}\sum\left(y_i - \hat{y}_i\right)^2$$

   b. The difference in the actual and estimated value of y is known as **residual**. The loss function calculates the mean of squared residuals. We define it in TensorFlow in the following way:

```
loss = tf.reduce_mean(tf.square(model - y_tensor))
```

c.
   i. **model - y_tensor** calculates the residuals
   ii. **tf.square(model - y_tensor)** calculates the squares of each residual
   iii. **tf.reduce_mean(...)** finally calculates the mean of squares calculated in the preceding step

   d. We also define the **mean squared error (mse)** and **r-squared (rs)** functions to evaluate the trained model. We use a separate **mse** function.

```
# mse and R2 functions
mse = tf.reduce_mean(tf.square(model - y_tensor))
y_mean = tf.reduce_mean(y_tensor)
total_error = tf.reduce_sum(tf.square(y_tensor - y_mean))
unexplained_error = tf.reduce_sum(tf.square(y_tensor - model))
rs = 1 - tf.div(unexplained_error, total_error)
```

4. Defining the optimizer function
   a. we instantiate the **theGradientDescentOptimizer** function with a learning rate of 0.001 and set it to minimize the loss function:

```
learning_rate = 0.001
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

   b. TensorFlow offers many other optimizer functions such as Adadelta, Adagrad, and Adam.
5. Training the model
   a. Now that we have the model, loss function, and optimizer function defined, train the model to learn the parameters, **w**, and **b**. To train the model, define the following global variables:
   b.
      i. **num_epochs**: The number of iterations to run the training for. With every iteration, the model learns better parameters
      ii. **w_hat** and **b_hat**: To collect the estimated w and b parameters.
      iii. **loss_epochs**, **mse_epochs**, **rs_epochs**: To collect the total error value on the training dataset, along with the mse and r-squared values of the model on the test dataset in every iteration.
      iv. **mse_score** and **rs_score**: To collect mse and r-squared values of the final trained model.
   c.

```
num_epochs = 1500
w_hat = 0
b_hat = 0
loss_epochs = np.empty(shape=[num_epochs],dtype=float)
mse_epochs = np.empty(shape=[num_epochs],dtype=float)
rs_epochs = np.empty(shape=[num_epochs],dtype=float)
```

```
mse_score = 0
rs_score = 0
```

d. After initializing the session and the global variables, run the training loop for **num_epoch** times:

```
with tf.Session() as tfs:
   tf.global_variables_initializer().run()
   for epoch in range(num_epochs):
```

e. Within each iteration of the loop, run the optimizer on the training data:

```
tfs.run(optimizer, feed_dict={x_tensor: X_train, y_tensor: y_train})
```

f. Using the learned **w** and **b** values, calculate the error and save it in **loss_val** to plot it later:

```
loss_val = tfs.run(loss,feed_dict={x_tensor: X_train, y_tensor: y_train})
loss_epochs[epoch] = loss_val
```

g. Calculate the mean squared error and r-squared value for the predicted values of the test data:

```
mse_score = tfs.run(mse,feed_dict={x_tensor: X_test, y_tensor: y_test})
mse_epochs[epoch] = mse_score

rs_score = tfs.run(rs,feed_dict={x_tensor: X_test, y_tensor: y_test})
rs_epochs[epoch] = rs_score
```

h. Finally, once the loop is finished, save the values of **w** and **b** to plot them later:

```
w_hat,b_hat = tfs.run([w,b])
w_hat = w_hat.reshape(1)
```

i. Let's print the model and final mean squared error on the test data after 2,000 iterations:

```
print('model : Y = {0:.8f} X + {1:.8f}'.format(w_hat[0],b_hat[0]))
print('For test data : MSE = {0:.8f}, R2 = {1:.8f} '.format(
    mse_score,rs_score))
```

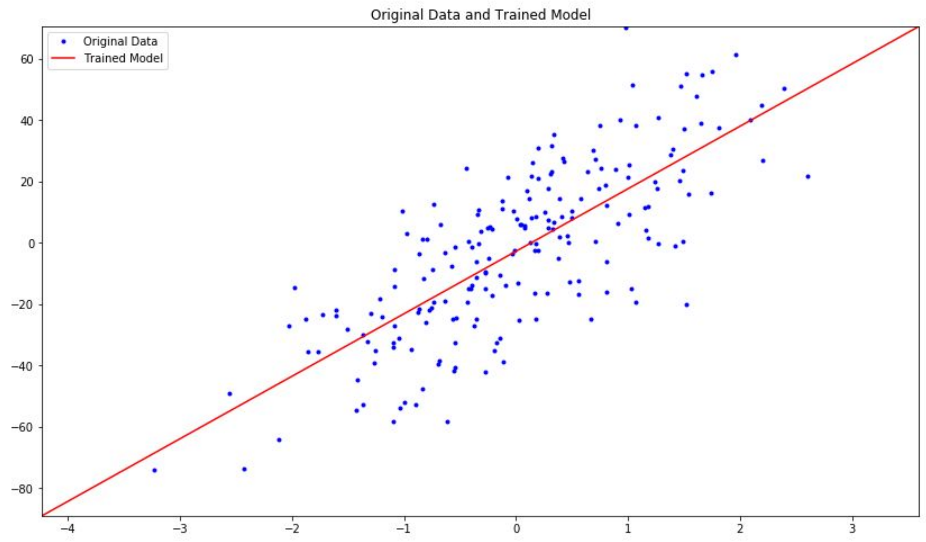
j. This gives us the following output:
   i. **model : Y = 20.37448120 X + -2.75295663**
   ii. **For test data : MSE = 297.57995605, R2 = 0.66098368**

k. Thus, the model that we trained is not a very good model, but we will see can improve it using neural networks

l. Let's plot the estimated model along with the original data:

```
plt.figure(figsize=(14,8))
plt.title('Original Data and Trained Model')
x_plot = [np.min(X)-1,np.max(X)+1]
y_plot = w_hat*x_plot+b_hat
plt.axis([x_plot[0],x_plot[1],y_plot[0],y_plot[1]])
plt.plot(X,y,'b.',label='Original Data')
plt.plot(x_plot,y_plot,'r-',label='Trained Model')
plt.legend()
plt.show()
```

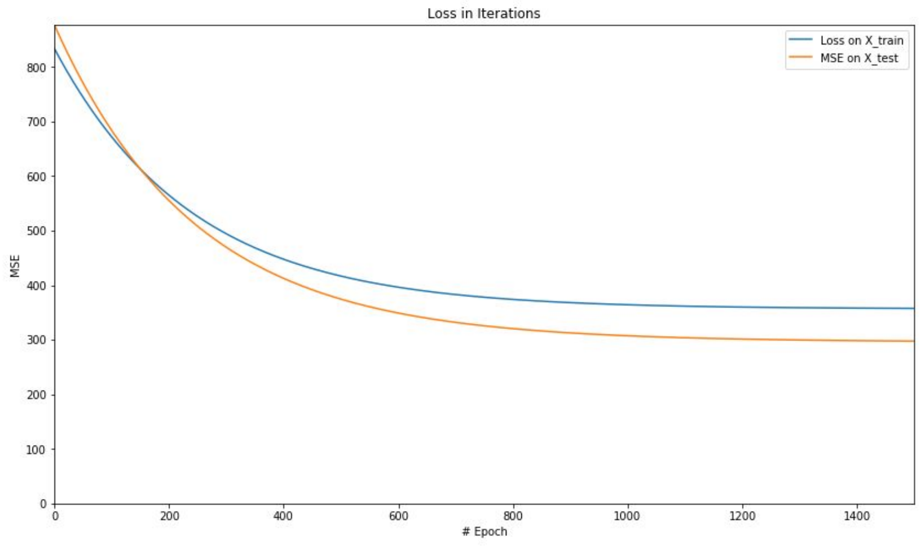
m. We get the following plot of the original data vs. the data from the trained model:



n. Let's plot the mean squared error for the training and test data in each iteration:

```
plt.figure(figsize=(14,8))

plt.axis([0,num_epochs,0,np.max(loss_epochs)])
plt.plot(loss_epochs, label='Loss on X_train')
plt.title('Loss in Iterations')
plt.xlabel('# Epoch')
plt.ylabel('MSE')

plt.axis([0,num_epochs,0,np.max(mse_epochs)])
plt.plot(mse_epochs, label='MSE on X_test')
plt.xlabel('# Epoch')
plt.ylabel('MSE')
plt.legend()
plt.show()
```

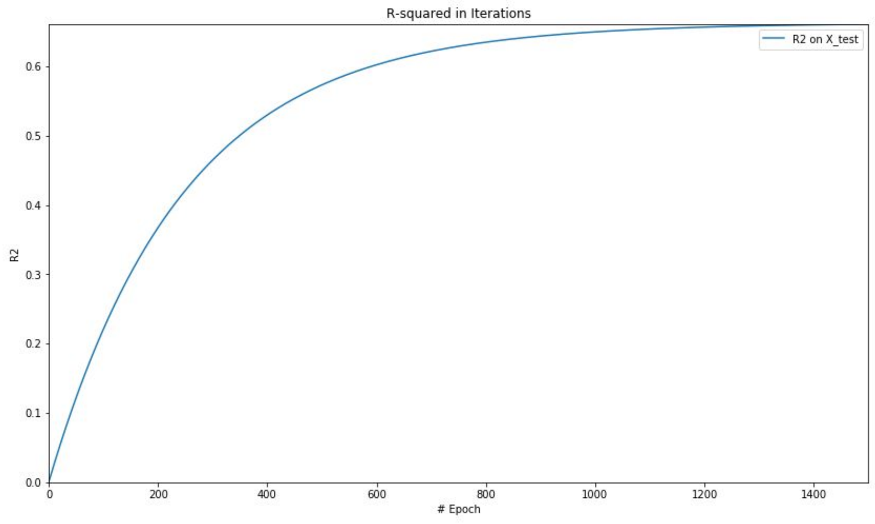
o. We get the following plot that shows that with each iteration, the mean squared error reduces and then remains at the same level near 500:



p. Let's plot the value of r-squared:

```
plt.figure(figsize=(14,8))
plt.axis([0,num_epochs,0,np.max(rs_epochs)])
plt.plot(rs_epochs, label='R2 on X_test')
plt.xlabel('# Epoch')
plt.ylabel('R2')
plt.legend()
plt.show()
```

q. We get the following plot when we plot the value of r-squared over epochs:



r. This basically shows that the model starts with a very low value of r-squared, but as the model gets trained and reduces the error, the value of r-squared starts getting higher and finally becomes stable at a point little higher than 0.6.

6. Using the trained model to predict
   a. Now that we have the trained model, it can be used to make predictions about new data. The predictions from the linear model are made with the understanding of some minimum mean squared error that we saw in the previous plot because the straight line may not fit the data perfectly.
   b. To get a better fitting model, we have to extend our model using different methods such as adding the linear combination of variables.

2. Multi-regression
   a. The dataset that we generated as an example dataset is univariate, namely, the target was dependent only on one feature.
   b. Most of the datasets, in reality, are multivariate. To emphasize a little more, the target depends on multiple variables or features, thus the regression model is called **multi-regression** or **multidimensional regression**.
   c. We first start with the most popular Boston dataset. This dataset contains 13 attributes of 506 houses in Boston such as the average number of rooms per dwelling, nitric oxide concentration, weighted distances to five Boston employment centers, and so on. The target is the median value of owner-occupied homes. Let's dive into exploring a regression model for this dataset.
   d. Load the dataset from the **sklearn** library and look at its description:

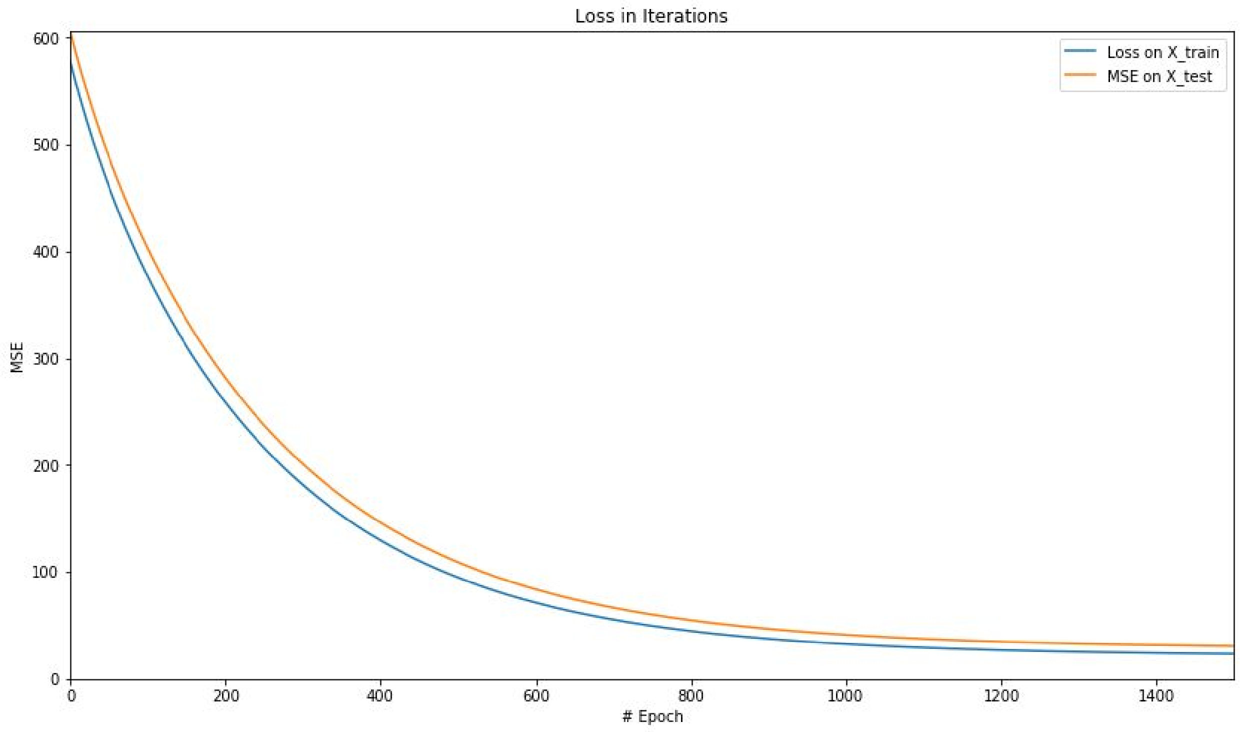
```
boston=skds.load_boston()
print(boston.DESCR)
X=boston.data.astype(np.float32)
y=boston.target.astype(np.float32)
if (y.ndim == 1):
    y = y.reshape(len(y),1)
X = skpp.StandardScaler().fit_transform(X)
```

   e. We also extract **X**, a matrix of features, and **y**, a vector of targets in the preceding code. We reshape **y** to make it two-dimensional and scale the features in **x** to have a mean of zero and standard deviation of one. Now let's use this **X** and **y** to train the regression model

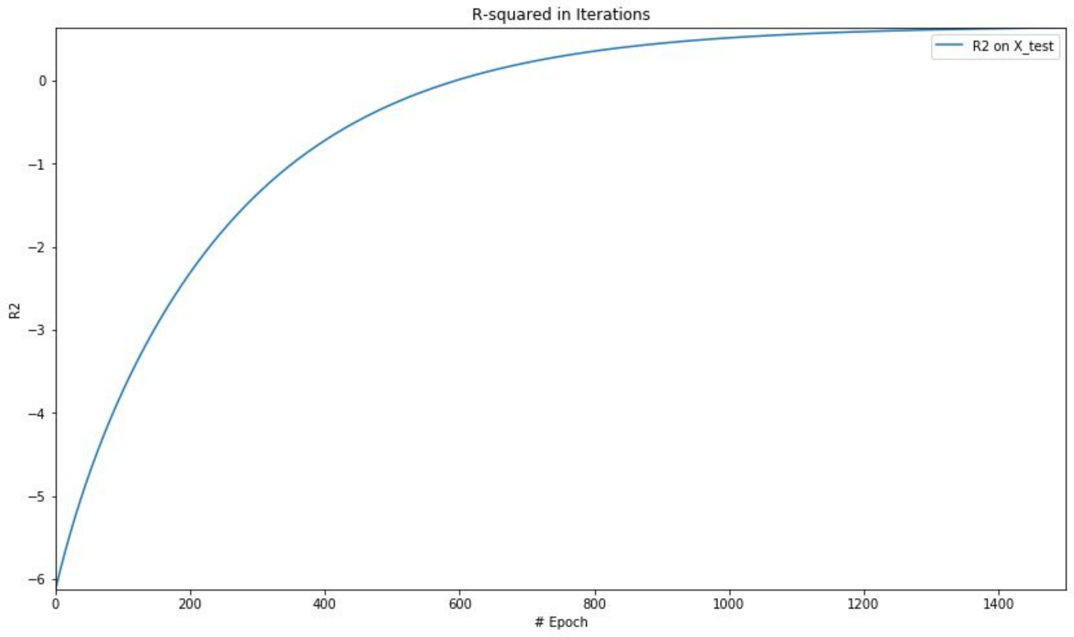
```
X_train, X_test, y_train, y_test = skms.train_test_split(X, y,
    test_size=.4, random_state=123)
num_outputs = y_train.shape[1]
```

```
num_inputs = X_train.shape[1]

x_tensor = tf.placeholder(dtype=tf.float32,
    shape=[None, num_inputs], name="x")
y_tensor = tf.placeholder(dtype=tf.float32,
    shape=[None, num_outputs], name="y")

w = tf.Variable(tf.zeros([num_inputs,num_outputs]),
    dtype=tf.float32, name="w")
b = tf.Variable(tf.zeros([num_outputs]),
    dtype=tf.float32, name="b")

model = tf.matmul(x_tensor, w) + b
loss = tf.reduce_mean(tf.square(model - y_tensor))
# mse and R2 functions
mse = tf.reduce_mean(tf.square(model - y_tensor))
y_mean = tf.reduce_mean(y_tensor)
total_error = tf.reduce_sum(tf.square(y_tensor - y_mean))
unexplained_error = tf.reduce_sum(tf.square(y_tensor - model))
rs = 1 - tf.div(unexplained_error, total_error)

learning_rate = 0.001
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

num_epochs = 1500
loss_epochs = np.empty(shape=[num_epochs],dtype=np.float32)
mse_epochs = np.empty(shape=[num_epochs],dtype=np.float32)
rs_epochs = np.empty(shape=[num_epochs],dtype=np.float32)

mse_score = 0
rs_score = 0

with tf.Session() as tfs:
    tfs.run(tf.global_variables_initializer())
    for epoch in range(num_epochs):
        feed_dict = {x_tensor: X_train, y_tensor: y_train}
        loss_val, _ = tfs.run([loss, optimizer], feed_dict)
        loss_epochs[epoch] = loss_val

        feed_dict = {x_tensor: X_test, y_tensor: y_test}
        mse_score, rs_score = tfs.run([mse, rs], feed_dict)
        mse_epochs[epoch] = mse_score
        rs_epochs[epoch] = rs_score

print('For test data : MSE = {0:.8f}, R2 = {1:.8f} '.format(
    mse_score, rs_score))
```

f. We get the following output from the model:
    i. **For test data : MSE = 30.48501778, R2 = 0.64172244**
g. Let's plot the MSE and R-squared values.
h. The following image shows the plotting of MSE:



i. The following image shows the plotting of R-squared values:



j. We see a similar pattern for MSE and r-squared

3. Regularized regression
   a. In linear regression, the model that we trained returns the best-fit parameters on the training data. However, finding the best-fit parameters on the training data may lead to overfitting.
   b. **Overfitting** means that the model fits best to the training data but gives a greater error on the test data. Thus, we generally add a penalty term to the model to obtain a simpler model.
   c. This penalty term is called a **regularization** term, and the regression model thus obtained is called a regularized regression model. There are three main types of regularization models:
      i. **Lasso regression**: In lasso regularization, also known as L1 regularization, the regularization term is the lasso parameter α multiplied with the sum of absolute values of the weights **w**. Thus, the loss function is as follows:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2+\alpha\frac{1}{n}\sum_{i=1}^{n}|w_i|$$

      ii. **Ridge regression**: In ridge regularization, also known as L2 regularization, the regularization term is the ridge parameter α multiplied with the ith sum of the squares of the weights **w**. Thus, the loss function is as follows:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2+\alpha\frac{1}{n}\sum_{i=1}^{n}w_i^2$$

iii. **ElasticNet regression**: When we add both lasso and ridge regularization terms, the resulting regularization is known as the ElasticNet regularization. Thus, the loss function is as follows:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2+\alpha_1\frac{1}{n}\sum_{i=1}^{n}|w_i|+\alpha_2\frac{1}{n}\sum_{i=1}^{n}w_i^2$$

d. A simple rule of thumb is to use L1 or Lasso when we want to remove some features, thus reducing computation time, but at the cost of reduced accuracy.

e. Let's see these regularization loss functions implemented in TensorFlow. We will continue with the Boston dataset that we used in the previous example.

i. Lasso regularization

1. We define the lasso parameter to have the value 0.8:

```
lasso_param = tf.Variable(0.8, dtype=tf.float32)
lasso_loss = tf.reduce_mean(tf.abs(w)) * lasso_param
```

2. Setting the lasso parameter as zero means no regularization as the term becomes zero. Higher the value of the regularization term, higher the penalty. The following is the complete code for lasso regularized regression to train the model in order to predict Boston house pricing:

```
num_outputs = y_train.shape[1]
num_inputs = X_train.shape[1]

x_tensor = tf.placeholder(dtype=tf.float32,
                          shape=[None, num_inputs], name='x')
y_tensor = tf.placeholder(dtype=tf.float32,
                          shape=[None, num_outputs], name='y')

w = tf.Variable(tf.zeros([num_inputs, num_outputs]),
                dtype=tf.float32, name='w')
b = tf.Variable(tf.zeros([num_outputs]),
                dtype=tf.float32, name='b')

model = tf.matmul(x_tensor, w) + b
```

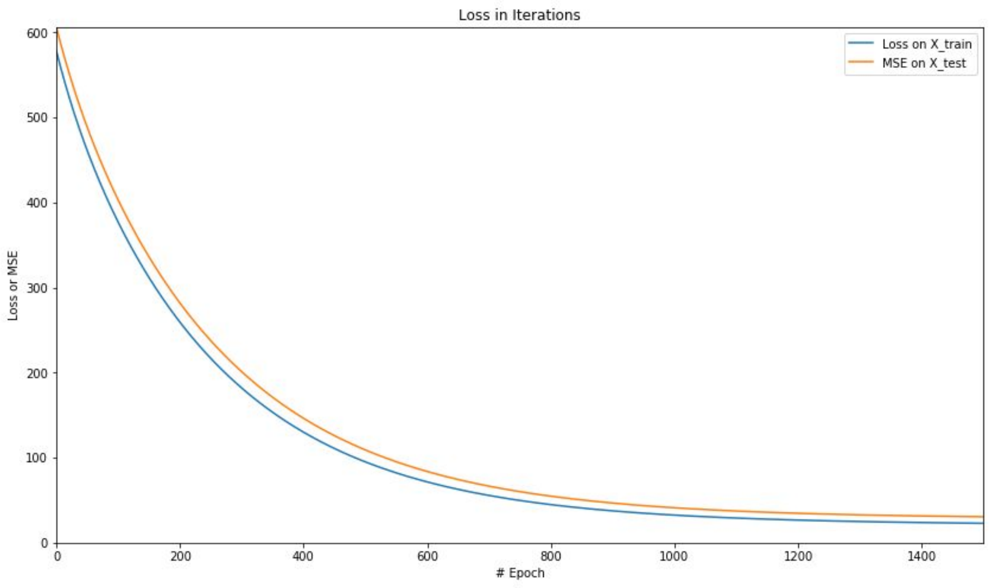
```
lasso_param = tf.Variable(0.8, dtype=tf.float32)
lasso_loss = tf.reduce_mean(tf.abs(w)) * lasso_param

loss = tf.reduce_mean(tf.square(model - y_tensor)) + lasso_loss

learning_rate = 0.001
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

mse = tf.reduce_mean(tf.square(model - y_tensor))
y_mean = tf.reduce_mean(y_tensor)
total_error = tf.reduce_sum(tf.square(y_tensor - y_mean))
unexplained_error = tf.reduce_sum(tf.square(y_tensor - model))
rs = 1 - tf.div(unexplained_error, total_error)

num_epochs = 1500
loss_epochs = np.empty(shape=[num_epochs],dtype=np.float32)
mse_epochs = np.empty(shape=[num_epochs],dtype=np.float32)
rs_epochs = np.empty(shape=[num_epochs],dtype=np.float32)

mse_score = 0.0
rs_score = 0.0

num_epochs = 1500
loss_epochs = np.empty(shape=[num_epochs], dtype=np.float32)
mse_epochs = np.empty(shape=[num_epochs], dtype=np.float32)
rs_epochs = np.empty(shape=[num_epochs], dtype=np.float32)

mse_score = 0.0
rs_score = 0.0

with tf.Session() as tfs:
    tfs.run(tf.global_variables_initializer())
    for epoch in range(num_epochs):
        feed_dict = {x_tensor: X_train, y_tensor: y_train}
        loss_val,_ = tfs.run([loss,optimizer], feed_dict)
        loss_epochs[epoch] = loss_val

        feed_dict = {x_tensor: X_test, y_tensor: y_test}
        mse_score,rs_score = tfs.run([mse,rs], feed_dict)
        mse_epochs[epoch] = mse_score
        rs_epochs[epoch] = rs_score
```

```
print('For test data : MSE = {0:.8f}, R2 = {1:.8f} '.format(
    mse_score, rs_score))
```

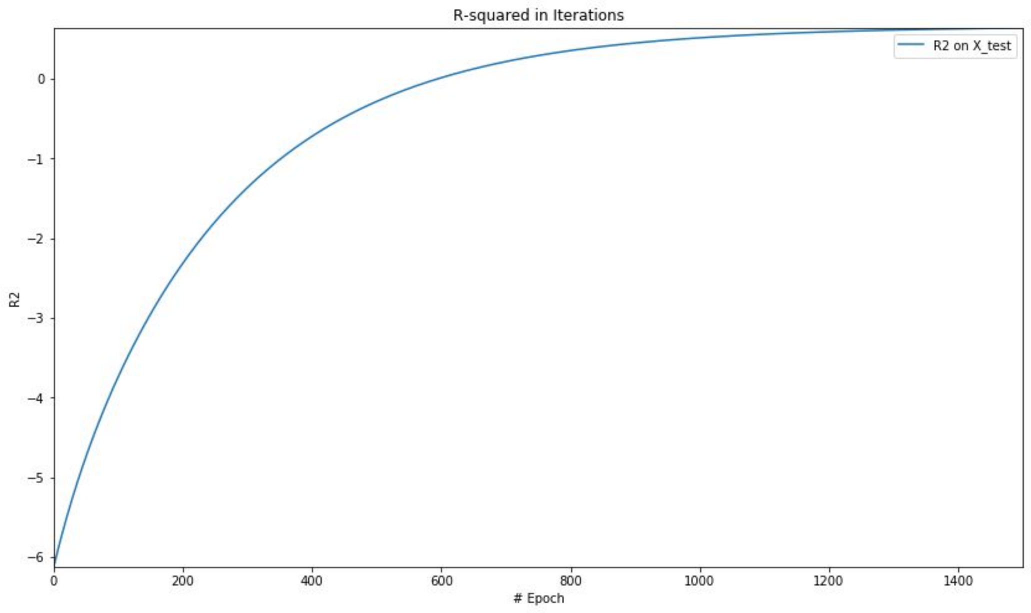
3. We get the following output:
    a. **For test data : MSE = 30.48978233, R2 = 0.64166653**
4. Let's plot the values of MSE and r-squared using the following code:

```
plt.figure(figsize=(14,8))

plt.axis([0,num_epochs,0,np.max([loss_epochs,mse_epochs])])
plt.plot(loss_epochs, label='Loss on X_train')
plt.plot(mse_epochs, label='MSE on X_test')
plt.title('Loss in Iterations')
plt.xlabel('# Epoch')
plt.ylabel('Loss or MSE')
plt.legend()

plt.show()

plt.figure(figsize=(14,8))

plt.axis([0,num_epochs,np.min(rs_epochs),np.max(rs_epochs)])
plt.title('R-squared in Iterations')
plt.plot(rs_epochs, label='R2 on X_test')
plt.xlabel('# Epoch')
plt.ylabel('R2')
plt.legend()

plt.show()
```

5. We get the following plot for loss:



6. The plot for R-squared in iterations is as follows:



7. Let's repeat the same example with ridge regression.

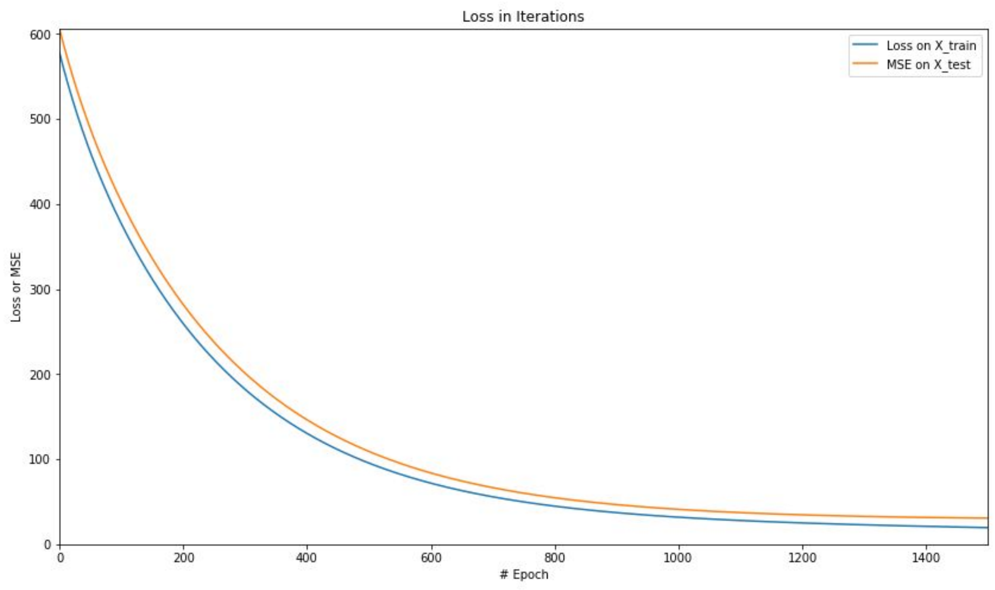
ii. Ridge regularization

1. The following is the complete code for ridge regularized regression to train the model in order to predict Boston house pricing:

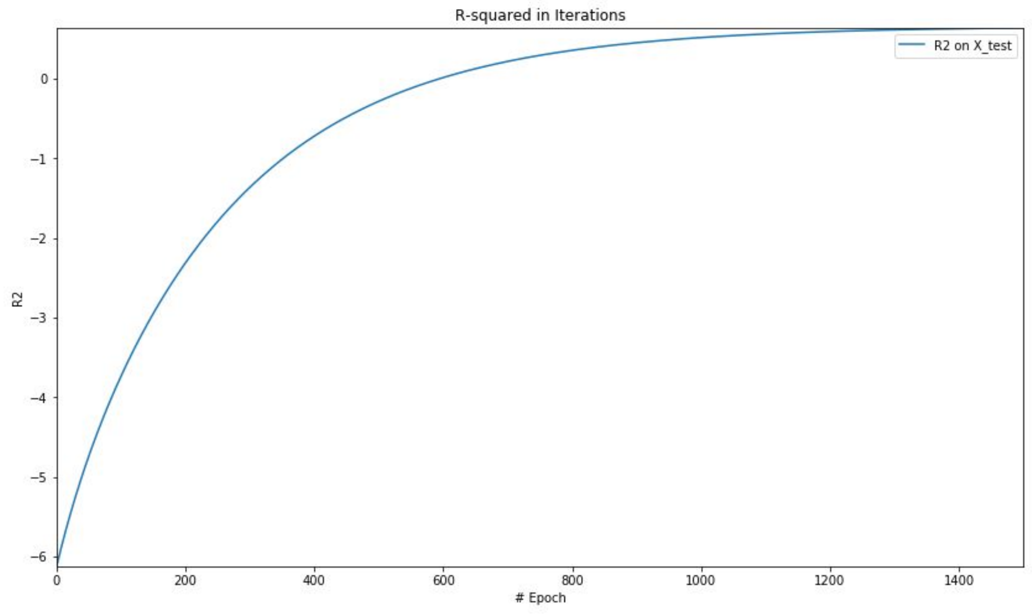
```
num_outputs = y_train.shape[1]
num_inputs = X_train.shape[1]

x_tensor = tf.placeholder(dtype=tf.float32,
                          shape=[None, num_inputs], name='x')
```

```
y_tensor = tf.placeholder(dtype=tf.float32,
                          shape=[None, num_outputs], name='y')

w = tf.Variable(tf.zeros([num_inputs, num_outputs]),
                dtype=tf.float32, name='w')
b = tf.Variable(tf.zeros([num_outputs]),
                dtype=tf.float32, name='b')

model = tf.matmul(x_tensor, w) + b

ridge_param = tf.Variable(0.8, dtype=tf.float32)
ridge_loss = tf.reduce_mean(tf.square(w)) * ridge_param

loss = tf.reduce_mean(tf.square(model - y_tensor)) + ridge_loss

learning_rate = 0.001
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

mse = tf.reduce_mean(tf.square(model - y_tensor))
y_mean = tf.reduce_mean(y_tensor)
total_error = tf.reduce_sum(tf.square(y_tensor - y_mean))
unexplained_error = tf.reduce_sum(tf.square(y_tensor - model))
rs = 1 - tf.div(unexplained_error, total_error)

num_epochs = 1500
loss_epochs = np.empty(shape=[num_epochs],dtype=np.float32)
mse_epochs = np.empty(shape=[num_epochs],dtype=np.float32)
rs_epochs = np.empty(shape=[num_epochs],dtype=np.float32)

mse_score = 0.0
rs_score = 0.0

with tf.Session() as tfs:
    tfs.run(tf.global_variables_initializer())
    for epoch in range(num_epochs):
        feed_dict = {x_tensor: X_train, y_tensor: y_train}
        loss_val, _ = tfs.run([loss, optimizer], feed_dict=feed_dict)
        loss_epochs[epoch] = loss_val

        feed_dict = {x_tensor: X_test, y_tensor: y_test}
        mse_score, rs_score = tfs.run([mse, rs], feed_dict=feed_dict)
        mse_epochs[epoch] = mse_score
        rs_epochs[epoch] = rs_score

print('For test data : MSE = {0:.8f}, R2 = {1:.8f} '.format(
    mse_score, rs_score))
```

2. We get the following result:
    a. **For test data : MSE = 30.64177132, R2 = 0.63988018**
3. Plotting the values of loss and MSE, we get the following plot for loss:



4. We get the following plot for R-squared:



5. Let's look at the combination of lasso and ridge regularization methods.

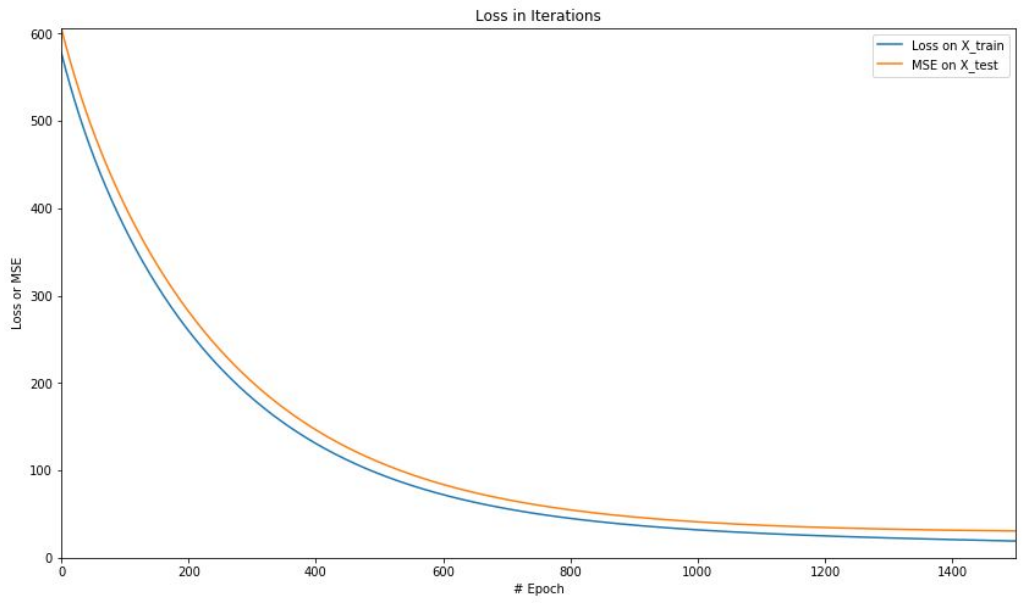
iii. ElasticNet regularization

1. The following is the complete code to train the model in order to predict Boston house pricing:

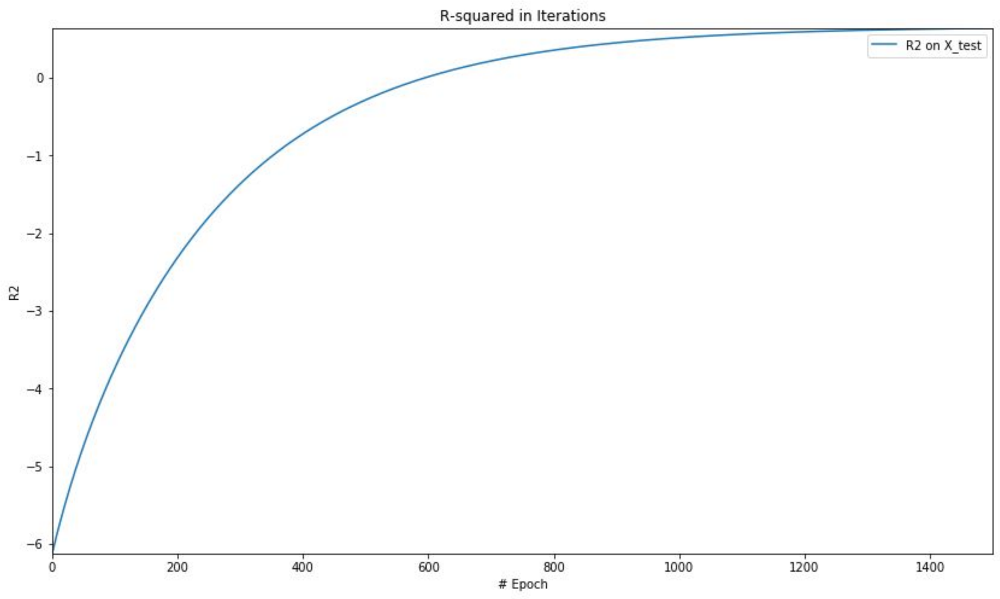
```
num_outputs = y_train.shape[1]
num_inputs = X_train.shape[1]

x_tensor = tf.placeholder(dtype=tf.float32,
                          shape=[None, num_inputs], name='x')
y_tensor = tf.placeholder(dtype=tf.float32,
                          shape=[None, num_outputs], name='y')

w = tf.Variable(tf.zeros([num_inputs, num_outputs]),
                dtype=tf.float32, name='w')
b = tf.Variable(tf.zeros([num_outputs]),
                dtype=tf.float32, name='b')

model = tf.matmul(x_tensor, w) + b

ridge_param = tf.Variable(0.8, dtype=tf.float32)
ridge_loss = tf.reduce_mean(tf.square(w)) * ridge_param
lasso_param = tf.Variable(0.8, dtype=tf.float32)
lasso_loss = tf.reduce_mean(tf.abs(w)) * lasso_param

loss = tf.reduce_mean(tf.square(model - y_tensor)) + \
    ridge_loss + lasso_loss

learning_rate = 0.001
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)

# mse and R2 functions
mse = tf.reduce_mean(tf.square(model - y_tensor))
y_mean = tf.reduce_mean(y_tensor)
total_error = tf.reduce_sum(tf.square(y_tensor - y_mean))
unexplained_error = tf.reduce_sum(tf.square(y_tensor - model))
rs = 1 - tf.div(unexplained_error, total_error)
num_epochs = 1500
loss_epochs = np.empty(shape=[num_epochs], dtype=np.float32)
mse_epochs = np.empty(shape=[num_epochs], dtype=np.float32)
rs_epochs = np.empty(shape=[num_epochs], dtype=np.float32)

mse_score = 0.0
rs_score = 0.0
```

```
with tf.Session() as tfs:
    tfs.run(tf.global_variables_initializer())
    for epoch in range(num_epochs):
        feed_dict = {x_tensor: X_train, y_tensor: y_train}
        loss_val, _ = tfs.run([loss, optimizer], feed_dict=feed_dict)
        loss_epochs[epoch] = loss_val

        feed_dict = {x_tensor: X_test, y_tensor: y_test}
        mse_score, rs_score = tfs.run([mse, rs], feed_dict=feed_dict)
        mse_epochs[epoch] = mse_score
        rs_epochs[epoch] = rs_score

print('For test data : MSE = {0:.8f}, R2 = {1:.8f} '.format(
    mse_score, rs_score))
```

2. we get the following result:
    a. **For test data : MSE = 30.64861488, R2 = 0.63979971**
3. Plotting the values of loss and MSE, we get the following plots:



4. We get the following plot for R-squared:



4. Classification using logistic regression
    a. The most common method for classification is using logistic regression. Logistic regression is a probabilistic and linear classifier. The probability that vector of input features is a member of a specific class can be written formally as the following equation:

$$P(Y = i | x, w, b) = \phi(z)$$

    b. In the above equation:
        i. **Y** represents the output,
        ii. **i** represents one of the classes
        iii. **x** represents the inputs
        iv. **w** represents the weights
        v. **b** represents the biases
        vi. **z** represents the regression equation

$$z = w \times x + b$$

        vii. ϕ represents the smoothing function or model in our case
    c. The equation represents that probability that **x** belongs to class **i** when **w** and **b** are given, is represented by function **ϕ(z)**. Thus the model has to be trained to maximize the value of probability.
    d. **Logistic regression for binary classification**

i. For binary classification, we define the model function **ϕ(z)** to be the sigmoid function, written as follows:

$$\phi(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-(w \times x + b)}}$$

ii. The sigmoid function produces the value of y to lie between the range [0,1]. Thus we can use the value of **y=ϕ(z)** to predict the class: if **y** > 0.5 then class is equal to 1, else class is equal to 0.

iii. For linear regression, the model can be trained by finding parameters that minimize the loss function and loss function could be the sum of squared error or mean squared error. For logistic regression, we want to maximize the likelihood:

$$L(w) = P(y|x, w, b)$$

iv. However, as it is easier to maximize the log-likelihood, thus we use the log-likelihood **l(w)** as the cost function. The loss function **(J(w))** is thus written as **-l(w)** that can be minimized using the optimization algorithms such as gradient descent.

v. The loss function for binary logistic regression is written mathematically as follows:

$$J(w) = -\sum_{i=1}^{n}[(y_i \times log(\phi(z_i))) + ((1-y_i) \times (1 - log(\phi(z_i))))]$$

vi. where ϕ(z) is the sigmoid function.

e. Logistic regression for multiclass classification

i. When there are more than two classes involved, the logistic regression is known multinomial logistic regression. In multinomial logistic regression, instead of sigmoid, we use softmax function that is one of the most popular functions. Softmax can be represented mathematically as follows:

$$softmax \;\; \phi_i(z) = \frac{e_i^z}{\sum_j e_j^z} = \frac{e_i^{(w \times x + b)}}{\sum_j e_j^{(w \times x + b)}} =$$

ii. Softmax function produces the probabilities for each class, and the probabilities vector adds to 1. While predicting, the class with highest softmax value becomes the output or predicted class. The loss function, is the negative log-likelihood function **-l(w)** that can be minimized by the optimizers such as gradient descent.

iii. The loss function for multinomial logistic regression is written formally as follows:

$$J(w) = -\sum_{i=1}^{n} [y_i \times log(\phi(z_i))]$$

iv. where φ(z) is the softmax function.

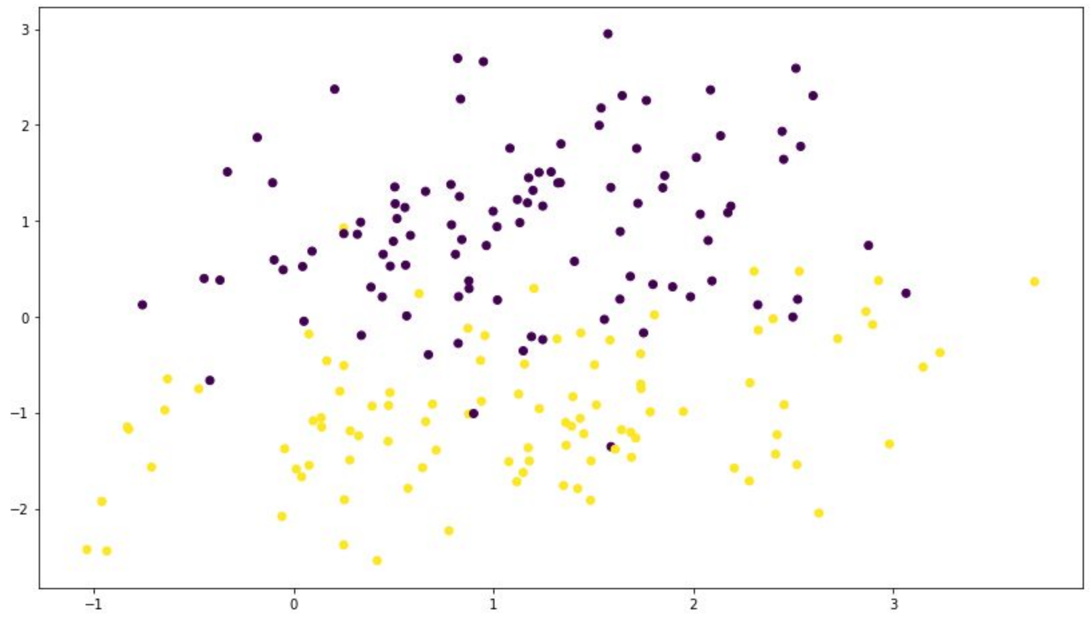
5. Binary classification
    a. Binary classification refers to problems with only two distinct classes. We will generate a dataset using the convenience function, **make_classification()**, in the SciKit Learn library:

```
X, y = skds.make_classification(n_samples=200,
   n_features=2,
   n_informative=2,
   n_redundant=0,
   n_repeated=0,
   n_classes=2,
   n_clusters_per_class=1)
if (y.ndim == 1):
    y = y.reshape(-1,1)
```

    b. The arguments to **make_classification()** are self-explanatory; **n_samples** is the number of data points to generate, **n_features** is the number of features to be generated, and **n_classes** is the number of classes, which is 2:
        i. **n_samples** is the number of data points to generate. We have kept it to 200 to keep the dataset small.
        ii. **n_features** is the number of features to be generated; we are using only two features so that we can keep it a simple problem to understand the TensorFlow commands.
        iii. **n_classes** is the number of classes, which is 2 as it is a binary classification problem.
    c. Let's plot the data using the following code:

```
plt.scatter(X[:,0],X[:,1],marker='o',c=y)
plt.show()
```

d. We get the following plot:



e. Then we use the NumPy **eye** function to convert **y** to one-hot encoded targets:

```
print(y[0:5])
y=np.eye(num_outputs)[y]
print(y[0:5])
```

f. The one-hot encoded targets appear as follows:

   i. **[1 0 0 1 0]**
   ii. **[[0. 1.]**
   iii. **[1. 0.]**
   iv. **[1. 0.]**
   v. **[0. 1.]**
   vi. **[1. 0.]]**

g. Divide the data into train and test categories:

```
X_train, X_test, y_train, y_test = skms.train_test_split(
    X, y, test_size=.4, random_state=42)
```

h. In classification, we use the sigmoid function to quantify the value of model such that the output value lies between the range [0,1]. The following equations denote the sigmoid function indicated by $\phi(z)$, where z is the equation

   i. **W X x + b**

i. The loss function now changes to the one indicated by $J(\theta)$, where θ represents the parameters

$$z_i = w_i \times x_i + b$$

$$\phi(z) = \frac{1}{1+e^{-z}}$$

$$J(w) = -\sum_{i=1}^{n}[(y_i \times log(\phi(z_i))) + ((1-y_i) \times (1-log(\phi(z_i))))]$$

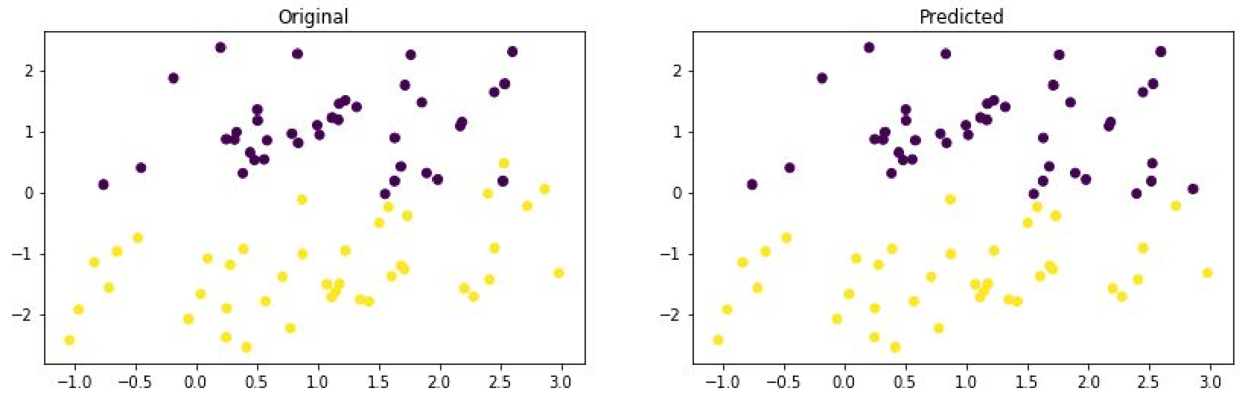
j. We implement the new model and loss function using the following code:

```
num_outputs = y_train.shape[1]
num_inputs = X_train.shape[1]

learning_rate = 0.001

# input images
x = tf.placeholder(dtype=tf.float32, shape=[None, num_inputs], name="x")
# output labels
y = tf.placeholder(dtype=tf.float32, shape=[None, num_outputs], name="y")

# model parameters
w = tf.Variable(tf.zeros([num_inputs,num_outputs]), name="w")
b = tf.Variable(tf.zeros([num_outputs]), name="b")
model = tf.nn.sigmoid(tf.matmul(x, w) + b)

loss = tf.reduce_mean(-tf.reduce_sum(
    (y * tf.log(model)) + ((1 - y) * tf.log(1 - model)), axis=1))
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate).minimize(loss)
```

k. Finally, we run our classification model:

```
num_epochs = 1
with tf.Session() as tfs:
    tf.global_variables_initializer().run()
    for epoch in range(num_epochs):
        tfs.run(optimizer, feed_dict={x: X_train, y: y_train})
        y_pred = tfs.run(tf.argmax(model, 1), feed_dict={x: X_test})
        y_orig = tfs.run(tf.argmax(y, 1), feed_dict={y: y_test})

        preds_check = tf.equal(y_pred, y_orig)
        accuracy_op = tf.reduce_mean(tf.cast(preds_check, tf.float32))
        accuracy_score = tfs.run(accuracy_op)
        print("epoch {0:04d} accuracy={1:.8f}".format(
            epoch, accuracy_score))

        plt.figure(figsize=(14, 4))
        plt.subplot(1, 2, 1)
        plt.scatter(X_test[:, 0], X_test[:, 1], marker='o', c=y_orig)
        plt.title('Original')
        plt.subplot(1, 2, 2)
        plt.scatter(X_test[:, 0], X_test[:, 1], marker='o', c=y_pred)
        plt.title('Predicted')
        plt.show()
```

l. We get a pretty good accuracy of about 96 percent and the original and predicted data graphs look like this:



6. Multiclass classification
    a. One of the popular examples of multiclass classification is to label the images of handwritten digits. The classes or labels in this examples are {0,1,2,3,4,5,6,7,8,9}.
    b. In the following example, we will use MNIST. Let's load the MNIST images as we did in the earlier chapter with the following code:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets(os.path.join(
    datasetslib.datasets_root, 'mnist'), one_hot=True)
```

    c. If the MNIST dataset is already downloaded,then we would get the following output:
        i. **Extracting /Users/armando/datasets/mnist/train-images-idx3-ubyte.gz**
        ii. **Extracting /Users/armando/datasets/mnist/train-labels-idx1-ubyte.gz**
        iii. **Extracting /Users/armando/datasets/mnist/t10k-images-idx3-ubyte.gz**
        iv. **Extracting /Users/armando/datasets/mnist/t10k-labels-idx1-ubyte.gz**
    d. Now let's set some parameters, as shown in the following code:

```
num_outputs = 10 # 0-9 digits
num_inputs = 784 # total pixels
```

```
learning_rate = 0.001
num_epochs = 1
batch_size = 100
num_batches = int(mnist.train.num_examples/batch_size)
```

e. The parameters in the above code are as follows:
   i. **num_outputs**: As we have to predict that image represents which digit out of the ten digits, thus we set the number of outputs as 10. The digit is represented by the output that is turned on or set to one.
   ii. **num_inputs**: We know that our input digits are 28 x 28 pixels, thus each pixel is an input to the model. Thus we have a total of 784 inputs.
   iii. **learning_rate**: This parameter represents the learning rate for the gradient descent optimizer algorithm. We set the learning rate arbitrarily to 0.001.
   iv. **num_epochs**: We will run our first example only for one iteration, hence we set the number of epochs to 1.
   v. **batch_size**: In the real world, we might have a huge dataset and loading the whole dataset in order to train the model may not be possible. Hence, we divide the input data into batches that are chosen randomly. We set the **batch_size** to 100 images that can be selected at a time using TensorFlow's inbuilt algorithm.
   vi. **num_batches**: This parameter sets the number of times the batches should be selected from the total dataset; we set this to be equal to the number of items in the dataset divided by the number of items in a batch.

f. Now let's define the inputs, outputs, parameters, model, and loss function using the following code:

```
# input images
x = tf.placeholder(dtype=tf.float32, shape=[None, num_inputs], name="x")
# output labels
y = tf.placeholder(dtype=tf.float32, shape=[None, num_outputs], name="y")

# model parameters
w = tf.Variable(tf.zeros([784, 10]), name="w")
b = tf.Variable(tf.zeros([10]), name="b")
model = tf.nn.softmax(tf.matmul(x, w) + b)

loss = tf.reduce_mean(-tf.reduce_sum(y * tf.log(model), axis=1))
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=learning_rate).minimize(loss)
```

g. The code is similar to the binary classification example with one significant difference: we use **softmax** instead of **sigmoid** function. **Softmax** is used for multiclass classification whereas **sigmoid** is used for binary class classification. **Softmax** function is a generalization of the **sigmoid** function that converts an n-dimensional vector z of arbitrary real values to an n-dimensional vector σ(z) of real values in the range (0, 1] that add up to 1.

h. Now let's run the model and print the accuracy:

```
with tf.Session() as tfs:
    tf.global_variables_initializer().run()
    for epoch in range(num_epochs):
        for batch in range(num_batches):
            batch_x, batch_y = mnist.train.next_batch(batch_size)
            tfs.run(optimizer, feed_dict={x: batch_x, y: batch_y})
        predictions_check = tf.equal(tf.argmax(model, 1), tf.argmax(y, 1))
        accuracy_function = tf.reduce_mean(
            tf.cast(predictions_check, tf.float32))
        feed_dict = {x: mnist.test.images, y: mnist.test.labels}
        accuracy_score = tfs.run(accuracy_function, feed_dict)
        print("epoch {0:04d} accuracy={1:.8f}".format(
            epoch, accuracy_score))
```

i. We get the following accuracy:
   i. **epoch 0000 accuracy=0.76109999**

j. Let's try training our model in multiple iterations, such that it learns with different batches in each iteration. We build two supporting functions to help us with this:

```
def mnist_batch_func(batch_size=100):
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    return [batch_x, batch_y]
```

k. The preceding function takes the number of examples in a batch as input and uses the **mnist.train.next_batch()** function to return a batch of features (**batch_x**) and targets (**batch_y**).

```
def tensorflow_classification(num_epochs, num_batches, batch_size,
                              batch_func, optimizer, test_x, test_y):
    accuracy_epochs = np.empty(shape=[num_epochs], dtype=np.float32)
    with tf.Session() as tfs:
        tf.global_variables_initializer().run()
        for epoch in range(num_epochs):
            for batch in range(num_batches):
                batch_x, batch_y = batch_func(batch_size)
                feed_dict = {x: batch_x, y: batch_y}
                tfs.run(optimizer, feed_dict)
            predictions_check = tf.equal(
                tf.argmax(model, 1), tf.argmax(y, 1))
            accuracy_function = tf.reduce_mean(
                tf.cast(predictions_check, tf.float32))
            feed_dict = {x: test_x, y: test_y}
            accuracy_score = tfs.run(accuracy_function, feed_dict)
            accuracy_epochs[epoch] = accuracy_score
            print("epoch {0:04d} accuracy={1:.8f}".format(
                epoch, accuracy_score))

    plt.figure(figsize=(14, 8))
    plt.axis([0, num_epochs, np.min(
        accuracy_epochs), np.max(accuracy_epochs)])
    plt.plot(accuracy_epochs, label='Accuracy Score')
    plt.title('Accuracy over Iterations')
    plt.xlabel('# Epoch')
    plt.ylabel('Accuracy Score')
    plt.legend()
    plt.show()
```

l. The preceding function takes the parameters and performs the training iterations, printing the accuracy score for each iteration and prints the accuracy scores. It also saves the accuracy scores for each epoch in the **accuracy_epochs** array. Later, it plots the accuracy in each epoch. Let's run this function for 30 epochs using the parameters we set previously, using the following code:

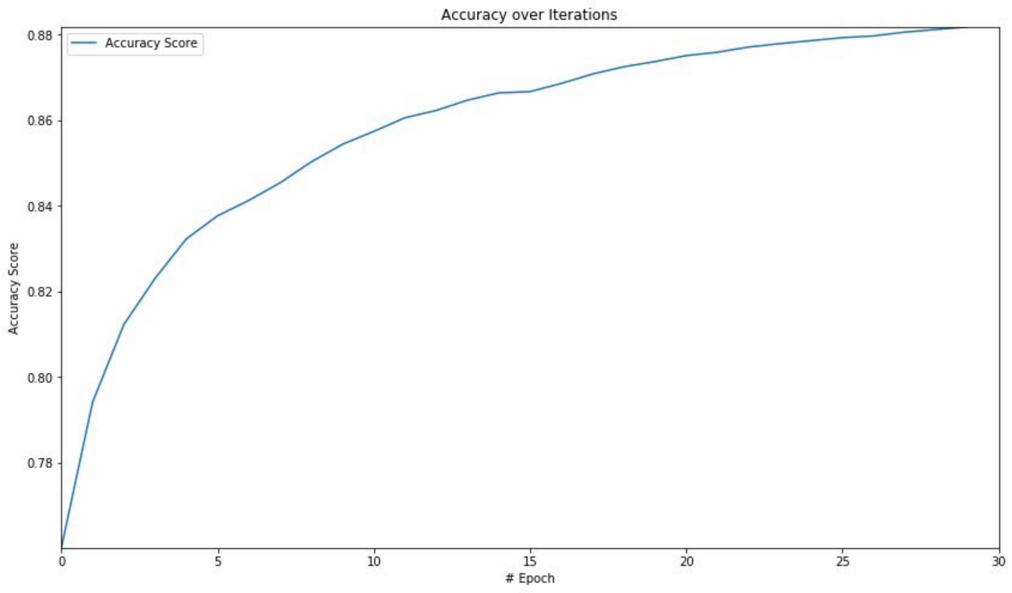
```
num_epochs=30
tensorflow_classification(num_epochs=num_epochs,
    num_batches=num_batches,
    batch_size=batch_size,
    batch_func=mnist_batch_func,
    optimizer=optimizer,
    test_x=mnist.test.images,test_y=mnist.test.labels)
```

m. We get the following accuracy and graph:

    i. **epoch 0000 accuracy=0.76020002**
    ii. **epoch 0001 accuracy=0.79420000**
    iii. **epoch 0002 accuracy=0.81230003**
    iv. **epoch 0003 accuracy=0.82309997**
    v. **epoch 0004 accuracy=0.83230001**
    vi. **epoch 0005 accuracy=0.83770001**
    vii.
    viii. **--- epoch 6 to 24 removed for brevity ---**
    ix.
    x. **epoch 0025 accuracy=0.87930000**
    xi. **epoch 0026 accuracy=0.87970001**
    xii. **epoch 0027 accuracy=0.88059998**

xiii. **epoch 0028 accuracy=0.88120002**

xiv. **epoch 0029 accuracy=0.88180000**



n. As we can see from the graph, accuracy improves very sharply in initial iterations and then the rate of improvement in accuracy slows down.