Logistic Regression

Loading Data and Libraries

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
plt.style.use("ggplot")
%matplotlib inline
```

In [2]:
```
from pylab import rcParams
rcParams["figure.figsize"] = 20, 8
```

In [3]:
```
df_dmv = pd.read_csv("DMV_Written_Tests.csv")
df_dmv.head()
```

Out[3]:

	DMV_Test_1	DMV_Test_2	Results
0	34.623660	78.024693	0
1	30.286711	43.894998	0
2	35.847409	72.902198	0
3	60.182599	86.308552	1
4	79.032736	75.344376	1

In [4]:
```
df_dmv.shape
```

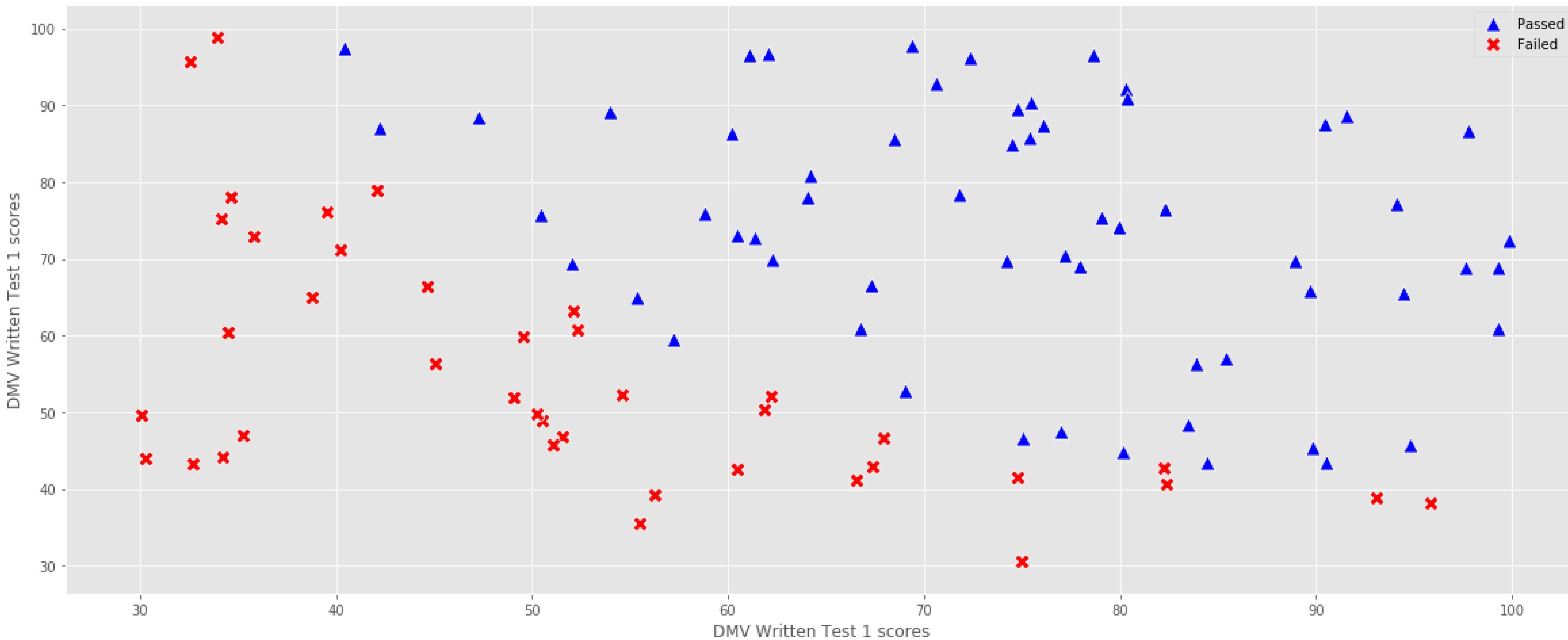
Out[4]: (100, 3)

In [5]:
```
test_scores = df_dmv[["DMV_Test_1", "DMV_Test_2"]].values
results = df_dmv["Results"].values
```

Data Visualization

In [6]:
```
passed = (results == 1).reshape(100, 1)
failed = (results == 0).reshape(100, 1)

ax = sns.scatterplot(x = test_scores[passed[:, 0], 0],
                     y = test_scores[passed[:, 0], 1],
                     marker = "^",
                     color = "blue", s = 120)
sns.scatterplot(x = test_scores[failed[:, 0], 0],
                y = test_scores[failed[:, 0], 1],
                marker = "X",
                color = "red", s = 120)
ax.set(xlabel = "DMV Written Test 1 scores", ylabel = "DMV Written Test 1 scores")
ax.legend(["Passed", "Failed"])
plt.show()
```



Logistic Sigmoid Function $\sigma(z)$

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

In [7]:
```
def sigmoid_function(x):
    return 1 / (1 + np.exp(-x))
```

In [8]:
```
sigmoid_function(0)
```

Out[8]: 0.5

Computing the Cost Function $J(\theta)$ and Gradient

The objective of logistic regression is to minimize the cost function

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}log(h_\theta(x^{(i)})) + (1 - y^{(i)})log(1 - (h_\theta(x^{(i)}))]$$

where the gradient of the cost function is given by

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

In [9]:
```
def cost_function(theta, x, y):
    m = len(y)
    y_pred = sigmoid_function(np.dot(x, theta))
    error = (y * np.log(y_pred) + (1 - y) * np.log(1 - y_pred))
    cost = - 1 / m * sum(error)
    gradient = 1 / m * np.dot(x.transpose(), (y_pred - y))
    return cost[0], gradient
```

Cost and Gradient at Initialization

In [10]:
```
mean_scores = np.mean(test_scores, axis = 0)
std_scores = np.std(test_scores, axis = 0)
test_scores = (test_scores - mean_scores) / std_scores

rows = test_scores.shape[0]
cols = test_scores.shape[1]

X = np.append(np.ones((rows, 1)), test_scores, axis = 1)
y = results.reshape(rows, 1)

theta_init = np.zeros((cols + 1, 1))
cost, gradient = cost_function(theta_init, X, y)

print(" Cost at Initialization is", cost)
print(" Gradients at Initialization is", gradient)
```

```
 Cost at Initialization is 0.69314718056
 Gradients at Initialization is [[-0.1        ]
 [-0.28122914]
 [-0.25098615]]
```

Gradient Descent

Minimize the cost function $J(\theta)$ by updating the below equation and repeat until convergence $\theta_j := \theta_j - \alpha\frac{\partial J(\theta)}{\partial \theta_j}$ (simultaneously update θ_j for all j)

In [11]:
```
def gradient_descent(X, y, theta, alpha, iterations):
    costs = []
    for i in range(iterations):
        cost, gradient = cost_function(theta, X, y)
        theta -= (alpha * gradient)
        costs.append(cost)
    return theta, costs
```

In [12]:
```
theta, costs = gradient_descent(X, y, theta_init, 1, 200)
```

In [13]:
```
print("Theta after gradient descent: ", theta)
print("Resulting cost: ", costs[-1])
```

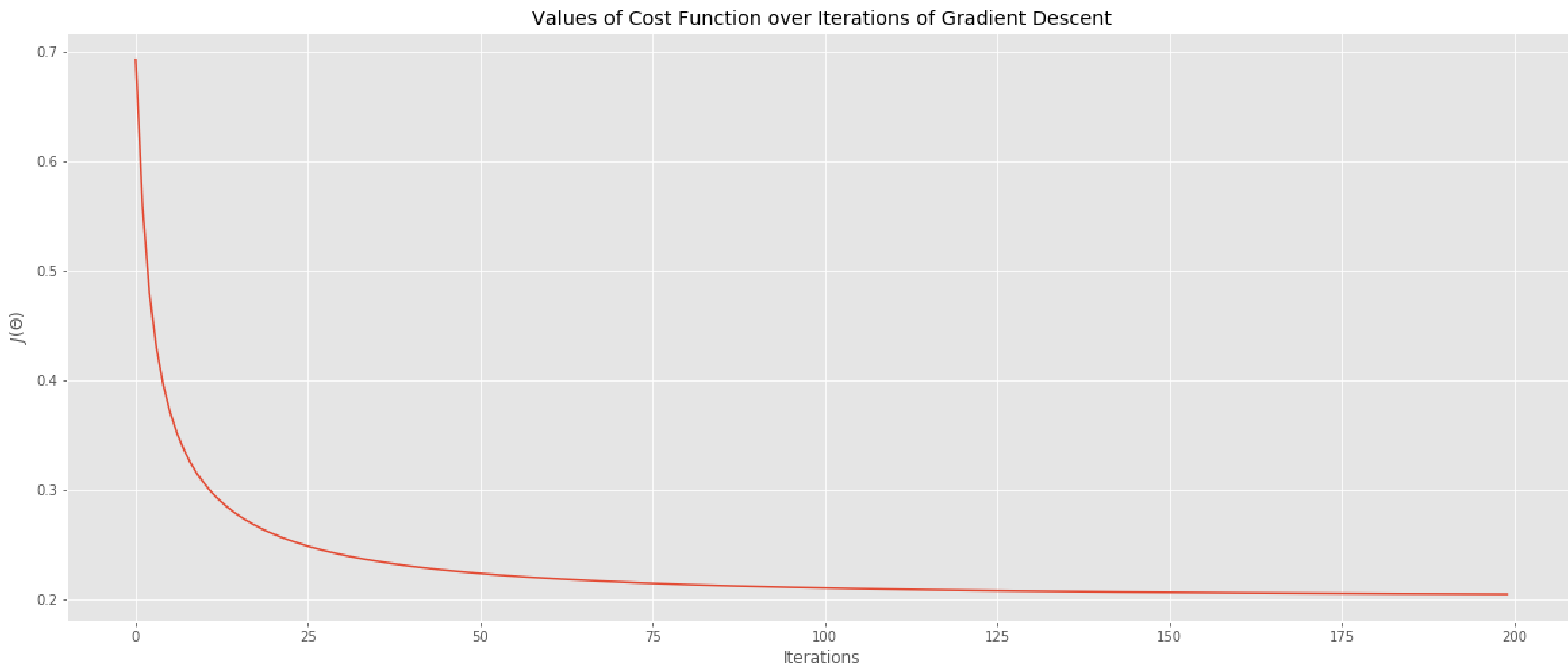
```
Theta after gradient descent:  [[ 1.50850586]
 [ 3.5468762 ]
 [ 3.29383709]]
Resulting cost:  0.204893820351
```

Plotting the Convergence of $J(\theta)$

Plot $J(\theta)$ against the number of iterations of gradient descent:

In [14]:
```
plt.plot(costs)
plt.xlabel("Iterations")
plt.ylabel("$J(\Theta)$")
plt.title("Values of Cost Function over Iterations of Gradient Descent")
```

Out[14]:
```
<matplotlib.text.Text at 0x8cdc110>
```



Plotting the decision boundary

$h_\theta(x) = \sigma(z)$, where σ is the logistic sigmoid function and $z = \theta^T x$

When $h_\theta(x) \geq 0.5$ the model predicts class "1":

$\implies \sigma(\theta^T x) \geq 0.5$

$\implies \theta^T x \geq 0$ predict class "1"

Hence, $\theta_1 + \theta_2 x_2 + \theta_3 x_3 = 0$ is the equation for the decision boundary, giving us

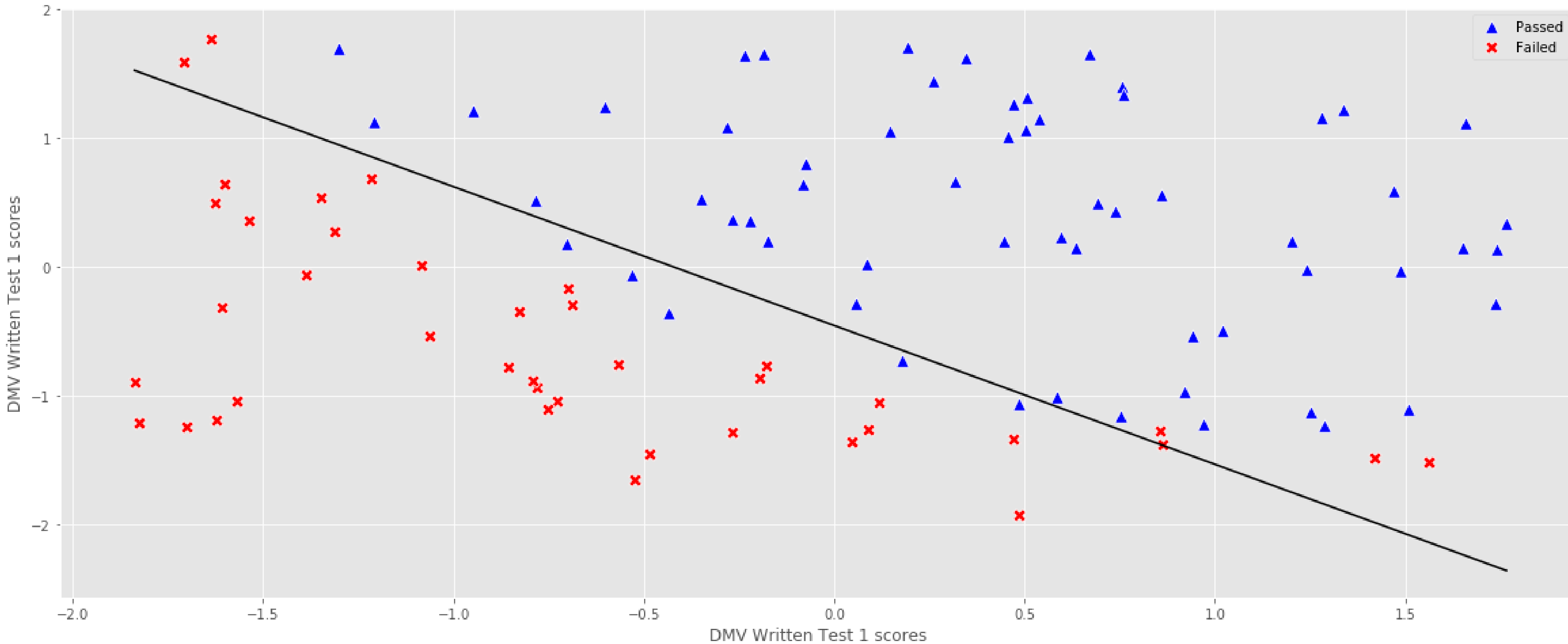
$x_3 = \frac{-(\theta_1 + \theta_2 x_2)}{\theta_3}$

In [15]:
```
ax = sns.scatterplot(x = X[passed[:, 0], 1],
                     y = X[passed[:, 0], 2],
                     marker = "^",
                     color = "blue", s = 90)
sns.scatterplot(x = X[failed[:, 0], 1],
                y = X[failed[:, 0], 2],
                marker = "X",
                color = "red", s = 90)

ax.set(xlabel = "DMV Written Test 1 scores", ylabel = "DMV Written Test 1 scores")
ax.legend(["Passed", "Failed"])

x_boundary = np.array([np.min(X[:, 1]), np.max(X[:, 1])])
y_boundary = -(theta[0] + theta[1] * x_boundary) / theta[2]

sns.lineplot(x = x_boundary, y = y_boundary, color = "black")
plt.show()
```



Predictions using the optimized θ values

$h_\theta(x) = x\theta$

In [16]:
```
def predict(theta, x):
    results = x.dot(theta)
    return results > 0
```

In [17]:
```
p = predict(theta, X)
print("Training Accuracy: ", sum(p == y)[0], "%")
```

```
Training Accuracy:  89 %
```

In [18]:
```
test = np.array([60, 79])
test = (test - mean_scores) / std_scores
test = np.append(np.ones(1), test)
probability = sigmoid_function(test.dot(theta))

print("A person who scores 60 and 79 on their DMV written test have a", np.round(probability[0], 2), "probability of passing")
```

```
A person who scores 60 and 79 on their DMV written test have a 0.94 probability of passing
```