Logistic Regression

In this exercise you will train a Logistic Regression model and analyze its performance. You will train the model on "cell2cell" -- a telecom company churn prediction data set.

Our problem is a binary classification problem. We are trying to predict whether a customer will leave their current telecom company or whether a customer will not leave the company. This problem is well suited for a Logistic Regression model.

Import Packages

Before you get started, import a few packages. Run the code cell below.

In [1]:

```
import pandas as pd
import numpy as np
import os
```

We will also import the scikit-learn linear model `LogisticRegression` classifier, the `train_test_split()` function for splitting the data into training and test sets, and the metrics `log_loss` and `accuracy_score` to evaluate our model.

In [2]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import log_loss
from sklearn.metrics import accuracy_score
```

Step 1. Load a 'ready-to-fit' Data Set

Load a Data Set and Save it as a Pandas DataFrame

We will work with a data set called "cell2celltrain." This data set is already pre-processed, with the proper formatting, outliers and missing values taken care of, and all numerical columns scaled to the [0, 1] interval.

In [3]:

```
filename = os.path.join(os.getcwd(), "data", "cell2celltrain.csv")
df = pd.read_csv(filename, header=0)
```

Step 2: Create Labeled Examples from the Data Set

Let's obtain columns from our data set to create labeled examples.

Identify Numeric Columns to use as Features

To implement a Logistic Regression model, we must use only the numeric columns.

In the code cell below: use the Pandas DataFrame `select_dtypes()` method to obtain all of the column names that have a dtype of "float64." Save the result to a list named `feature_list`.

In [4]:
```
feature_list = df.select_dtypes(include=['float64']).columns.tolist()
feature_list
```

Out[4]:
```
['MonthlyRevenue',
 'MonthlyMinutes',
 'TotalRecurringCharge',
 'DirectorAssistedCalls',
 'OverageMinutes',
 'RoamingCalls',
 'PercChangeMinutes',
 'PercChangeRevenues',
 'DroppedCalls',
 'BlockedCalls',
 'UnansweredCalls',
 'CustomerCareCalls',
 'ThreewayCalls',
 'ReceivedCalls',
 'OutboundCalls',
 'InboundCalls',
 'PeakCallsInOut',
 'OffPeakCallsInOut',
 'DroppedBlockedCalls',
 'CallForwardingCalls',
 'CallWaitingCalls',
 'MonthsInService',
 'UniqueSubs',
 'ActiveSubs',
 'Handsets',
 'HandsetModels',
 'CurrentEquipmentDays',
 'AgeHH1',
 'AgeHH2',
 'RetentionCalls',
 'RetentionOffersAccepted',
 'ReferralsMadeBySubscriber',
 'IncomeGroup',
 'AdjustmentsToCreditRating',
 'HandsetPrice']
```

Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

In [5]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testFeatureList

try:
    p, err = testFeatureList(df,feature_list)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

Create X and y Labeled Examples

In the code cell below carry out the following steps:

- Get the `Churn` column from DataFrame `df` and assign it to the variable `y`. This will be our label. The label will be either True or False.
- Get the columns listed in `feature_list` from DataFrame `df` and assign them to the variable `X`. These will be our features.

You should have 51047 labeled examples. Each example contains 35 features and one label (`Churn`).

In [6]:

```
y= df['Churn']
X = df[feature_list]

print("Number of examples: " + str(X.shape[0]))
print("\nNumber of Features:" + str(X.shape[1]))
print(str(list(X.columns)))
```

```
Number of examples: 51047

Number of Features:35
['MonthlyRevenue', 'MonthlyMinutes', 'TotalRecurringCharge', 'DirectorAssistedCalls', 'OverageMinutes', 'RoamingCalls', 'PercChangeMinutes', 'PercChangeRevenues', 'DroppedCalls', 'BlockedCalls', 'UnansweredCalls', 'CustomerCareCalls', 'ThreewayCalls', 'ReceivedCalls', 'OutboundCalls', 'InboundCalls', 'PeakCallsInOut', 'OffPeakCallsInOut', 'DroppedBlockedCalls', 'CallForwardingCalls', 'CallWaitingCalls', 'MonthsInService', 'UniqueSubs', 'ActiveSubs', 'Handsets', 'HandsetModels', 'CurrentEquipmentDays', 'AgeHH1', 'AgeHH2', 'RetentionCalls', 'RetentionOffersAccepted', 'ReferralsMadeBySubscriber', 'IncomeGroup', 'AdjustmentsToCreditRating', 'HandsetPrice']
```

Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

In [7]:

```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testXY

try:
    p, err = testXY(y,X,df)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

Step 3: Create Training and Test Data Sets ¶

In the code cell below, use the `train_test_split()` function to create training and test sets out of the labeled examples.

You will call `train_test_split()` function with the following arguments:

1. Variable `X` containing features.
2. Variable `y` containing the label.
3. A test set that is 33% (.33) of the size of the data set.
4. A `random_state` seed value of `1234`.

The `train_test_split()` function will return four outputs. Assign these outputs to the following variable names, using the following order: `X_train`, `X_test`, `y_train`, `y_test`.

Note that you will be able to accomplish this using one line of code.

Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

In [8]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33, random_state = 1234)
```

Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

In [8]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.33, random_state = 1234)
```

Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

In [9]:
```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testSplit

try:
    p, err = testSplit(X_train, X_test, y_train, y_test, df)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

Using the shape property, check the dimensions of the training and test data sets.

In [10]:
```
print(X_train.shape)
print(X_test.shape)
```

```
(34201, 35)
(16846, 35)
```

Step 4: Fit a Logistic Regression Classification Model and Evaluate the Model ¶

The code cell below contains code that must be completed to train a Logistic Regression classification model, analyze its performance and print the results. The code below will train a Logistic Regression model on the training data, test the resulting model on the test data, and compute and return (1) the log loss of the resulting probability predictions on the test data and (2) the accuracy score of the resulting predicted class labels on the test data.

Note: It is worth noting that evaluating a model's training loss and evaluating a model's accuracy is different. Accuracy measures what fraction of the examples are correctly predicted by the classifier, while training loss measures the average prediction error per training example over all training examples.

Your task is to fill in the code to make it work.

In the code cell below:

1. Use `LogisticRegression()` to create a model object, and assign the result to the variable `model`. You will not provide arguments for its hyperparameters, but will use Scikit-learn's default values.
2. Call the `model.fit()` method to fit the model to the training data. The first argument should be `X_train` and the second argument should be `y_train`.
3. Call the `model.predict_proba()` method with the argument `X_test` to use the fitted model to predict values for the test data. Store the outcome in the variable `probability_predictions`. Note that the `predict_proba()` method returns two columns, one column per class label. The first column contains the probability that an unlabeled example belongs to class `False` (Churn is "False) and the second column contains the probability that an unlabeled example belongs to class `True` (Churn is "True").
4. Call the `log_loss()` function; the first argument should be `y_test` and the second argument should be `probability_predictions`. Assign the result to variable `l_loss`. Recall that loss indicates how close the prediction probability is to the actual class label. The closer the probability is to the label (for example, a probability of 0.9 that the label is of class "False" when the actual class label is indeed "False"), the lower the loss.

5. Call the `model.predict()` method with the argument `X_test` to use the fitted model to predict the class labels for the test data. Store the outcome in the variable `class_label_predictions`. Note that the `predict()` method returns the class label (True or False) per unlabeled example.

6. Call the `accuracy_score()` function; the first argument should be `y_test` and the second argument should be `class_label_predictions`. Assign the result to variable `acc_score`. Accuracy refers to the number of class label predictions that are correct.

You might find it useful to consult the `LogisticRegression` Scikit-learn online documentation to see how to accomplish this task.

Graded Cell

The cell below will be graded. Remove the line "raise NotImplementedError()" before writing your code.

In [15]:

```
# 1. Create the LogisticRegression model object below and assign to variable 'model'

model = LogisticRegression()

# 2. Fit the model to the training data below

model.fit(X_train, y_train)

# 3. Make predictions on the test data using the predict_proba() method and assign the
# result to the variable 'probability_predictions' below

probability_predictions = model.predict_proba(X_test)

# print the first 5 probability class predictions
df_print = pd.DataFrame(probability_predictions, columns = ['Class: False', 'Class: True'])
print('Class Prediction Probabilities: \n' + df_print[0:5].to_string(index=False))

# 4. Compute the log loss on 'probability_predictions' and save the result to the variable
# 'l_loss' below

# YOUR CODE HERE
l_loss = log_loss(y_test, probability_predictions)
print('Log loss: ' + str(l_loss))


# 5. Make predictions on the test data using the predict() method and assign the result
# to the variable 'class_label_predictions' below

# YOUR CODE HERE
class_label_predictions = model.predict(X_test)

# print the first 5 class label predictions
print('Class labels: ' + str(class_label_predictions[0:5]))

# 6.Compute the accuracy score on 'class_label_predictions' and save the result
# to the variable 'acc_score' below

# YOUR CODE HERE
acc_score = accuracy_score(y_test, class_label_predictions)
print('Accuracy: ' + str(acc_score))
```

```
Class Prediction Probabilities:
 Class: False  Class: True
     0.745386     0.254614
     0.658797     0.341203
     0.724719     0.275281
     0.848821     0.151179
     0.749441     0.250559
Log loss: 0.5878612157234173
Class labels: [False False False False False]
Accuracy: 0.7097827377418972
```

Self-Check

Run the cell below to test the correctness of your code above before submitting for grading. Do not add code or delete code in the cell.

In [16]:

```
# Run this self-test cell to check your code;
# do not add code or delete code in this cell
from jn import testModel

try:
    p, err = testModel(probability_predictions, l_loss, class_label_predictions, acc_score, df)
    print(err)
except Exception as e:
    print("Error!\n" + str(e))
```

```
Correct!
```

Step 5: Thresholds: Map Probabilities to a Class Label

Examine the output of the code cell above.

Note that the `predict_proba()` method returns two columns. As stated, the first column contains the probability that an unlabeled example belongs to class `False` and the second column contains the probability that an unlabeled example belongs to class `True`.

The `predict()` method outputs the actual class label (`True` or `False`).

In order to determine the order of the classes in the `predict_proba()` output, you can inspect the model object's `classes_` property. Run the cell below and inspect the results. You'll see that indeed, column one gives the probability that an unlabeled example belongs to class `False` and column two gives the probability that an unlabeled example belongs to class `True`.

In [17]:

```
print(model.classes_)
```

```
[False  True]
```

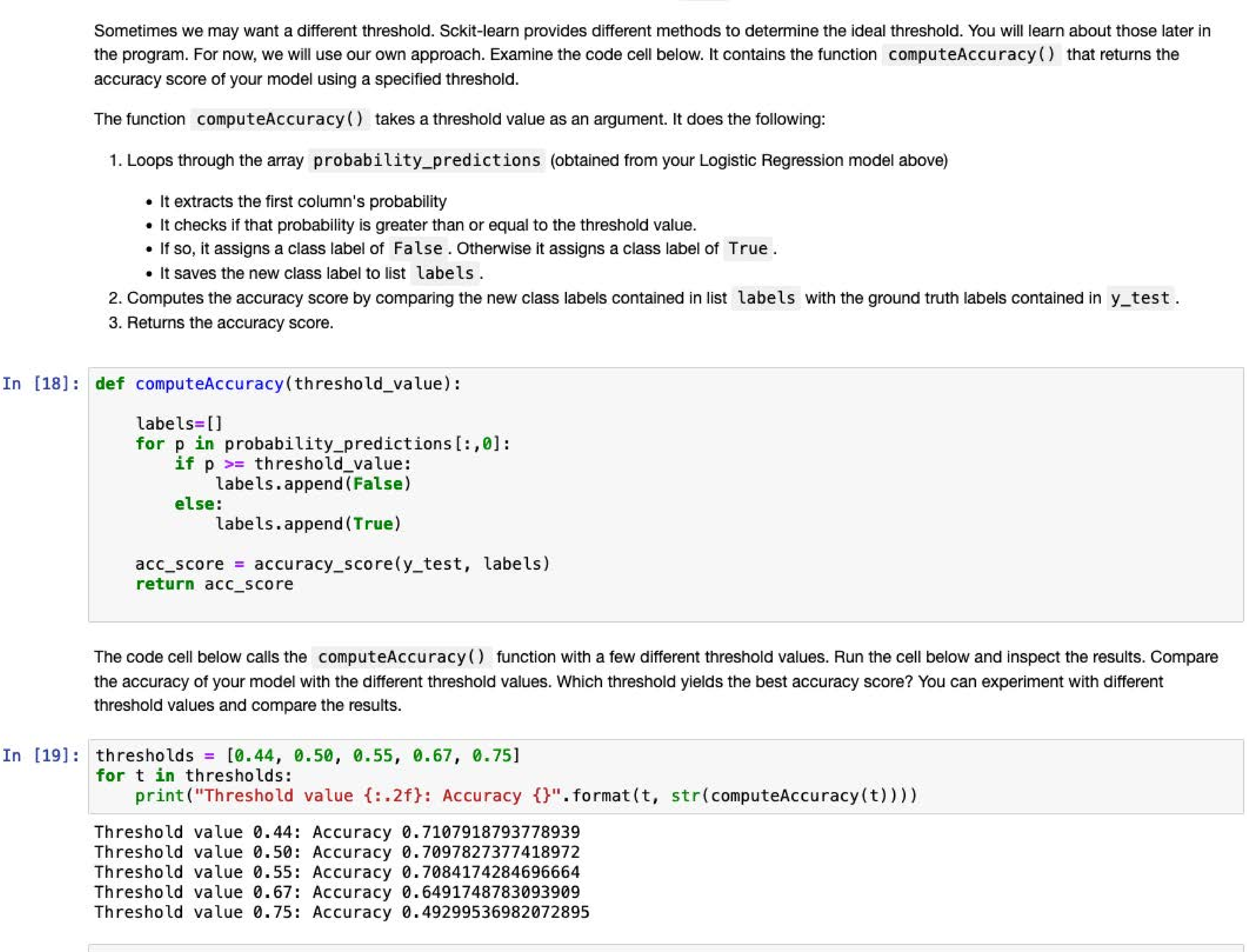
Notice how the probabilities map to labels in the table below. The table contains:

- 3 unlabeled examples
- the resulting class probability values from the logistic regression model's `predict_proba()` method
- the corresponding class label from the same logistic regression model's `predict()` method.

Inspect "Example 1". The probability that that unlabeled "Example 1" is of class `False` is 0.745386. The probability that unlabeled "Example 1" is of class `True` is 0.254614. The `predict()` method assigns "Example 1" the class label `False`.

	Class: False	Class: True	Class Label
Example 1	0.745386	0.254614	False
Example 2	0.745386	0.254614	False
Example 3	0.436033	0.563967	True

How does the Scikit-learn `predict()` method assign a class label based on probability values? For binary classification, the method defaults to a 0.5 threshold. If the resulting probability for class 0 is greater than or equal to 0.5, the unlabeled example is given a label of `False` On the other hand, if the probability for class 0 is less than 0.5, the unlabeled example is given a label of `True`.



Sometimes we may want a different threshold. Sckit-learn provides different methods to determine the ideal threshold. You will learn about those later in the program. For now, we will use our own approach. Examine the code cell below. It contains the function `computeAccuracy()` that returns the accuracy score of your model using a specified threshold.

The function `computeAccuracy()` takes a threshold value as an argument. It does the following:

1. Loops through the array `probability_predictions` (obtained from your Logistic Regression model above)
    - It extracts the first column's probability
    - It checks if that probability is greater than or equal to the threshold value.
    - If so, it assigns a class label of `False`. Otherwise it assigns a class label of `True`.
    - It saves the new class label to list `labels`.
2. Computes the accuracy score by comparing the new class labels contained in list `labels` with the ground truth labels contained in `y_test`.
3. Returns the accuracy score.

In [18]:

```
def computeAccuracy(threshold_value):
    
    labels=[]
    for p in probability_predictions[:,0]:
        if p >= threshold_value:
            labels.append(False)
        else:
            labels.append(True)
    
    acc_score = accuracy_score(y_test, labels)
    return acc_score
```

The code cell below calls the `computeAccuracy()` function with a few different threshold values. Run the cell below and inspect the results. Compare the accuracy of your model with the different threshold values. Which threshold yields the best accuracy score? You can experiment with different threshold values and compare the results.

In [19]:

```
thresholds = [0.44, 0.50, 0.55, 0.67, 0.75]
for t in thresholds:
    print("Threshold value {:.2f}: Accuracy {}".format(t, str(computeAccuracy(t))))
```

```
Threshold value 0.44: Accuracy 0.7107918793778939
Threshold value 0.50: Accuracy 0.7097827377418972
Threshold value 0.55: Accuracy 0.7084174284696664
Threshold value 0.67: Accuracy 0.6491748783093909
Threshold value 0.75: Accuracy 0.49299536982072895
```