LOGISTIC REGRESSION

In the linear regression module, you have learned that machine learning models broadly serve three purposes:

- Regression
- Classification
- Clustering

You have already learned about linear regression, where the predictions are numeric. In this module, you learned about a classification model, logistic regression. As you might already know, the classification models output categories of the target variables.

You will understand various aspects of logistic regression in this module.

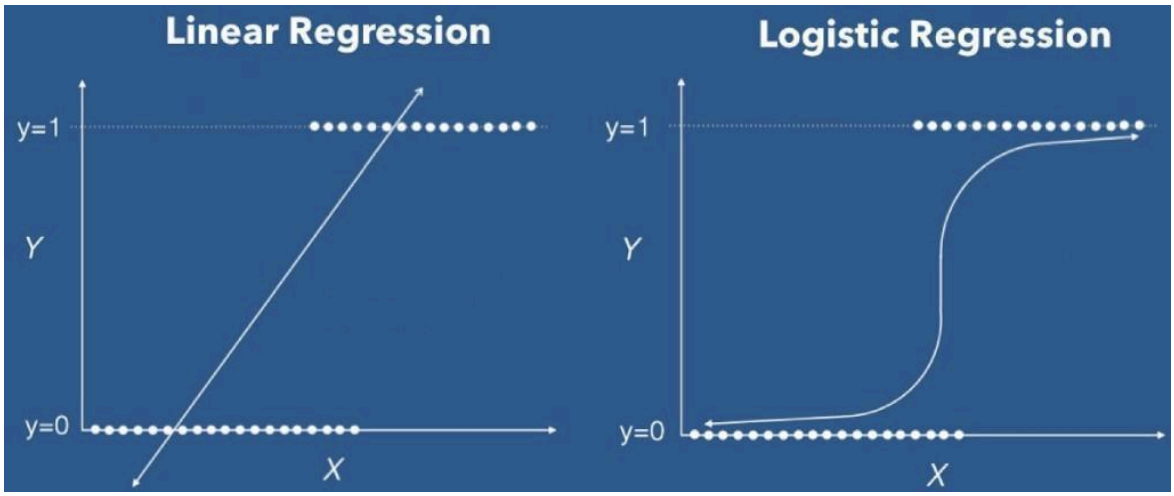
Mathematics behind Logistic Regression

Linear Regression Vs Logistic Regression:

All of the mathematics that we discussed or will discuss in this session are handled by the python library we are going to use. So, it is fine even if you do not grasp all of the functions. Try to focus on the approach. Now, let us look at how we derived the log loss function.

You saw that the dependent variable y is a continuous variable for linear regression, whereas y is a discrete variable(0 and 1 in general) in logistic regression. Earlier in the module, you have seen that a machine learning model tries to learn the function f that maps independent variables X to dependent variable y. Also, you need to provide a family of functions to narrow the search for this function.

In linear regression, we chose straight lines as the required family of functions. But when your target variable is categorical, there is a separation in data points and a straight line might not fit the requirements. If you observe the image below, you can see how a straight line does not fit the data.



So, it is difficult to use straight lines for classification models to fit the data. Now, it might seem that a step function can be a good fit here, but as we discussed earlier, most of the classification models return the probability of a positive class instead of the class(0 or 1) itself. And these probabilities are between 0 and 1 or maybe close to 0 or 1 but not equal to 0 or 1.

To summarize, the function we use to predict probabilities must satisfy the following criteria:

1. It is bound between **0** and 1
2. The function is **asymptotic**, that is, it can never take a value of 0 or 1.
3. The function must accommodate the separation in data points that arises in classification.

Once you have the probabilities, the final classes are derived based on whether the probability of a positive class for a data point is greater than the threshold value or not.

A function that satisfies all of these conditions is sigmoid, it can be expressed as:

$$y = \frac{1}{1+e^{-x}}$$

4.As x tends to - ∞, y tends to 0, and as x tends to +∞, y tends to 1. Moreover, for x = 0, y is 0.5. So overall, the sigmoid function satisfies the above criteria for a classification function.

In our use case, this equation transforms to:

$$p(b) = \frac{1}{1+e^{-(\beta_0+\beta_1 b)}}$$

Where p(b) is the probability of the positive class for an independent variable b. β_1 is the 'weight' of variable b.

Log Loss Function

As you already know, the probability of a positive class can be found using the sigmoid function.

$$p(b) = \frac{1}{1+e^{-(\beta_0+\beta_1 b)}}$$

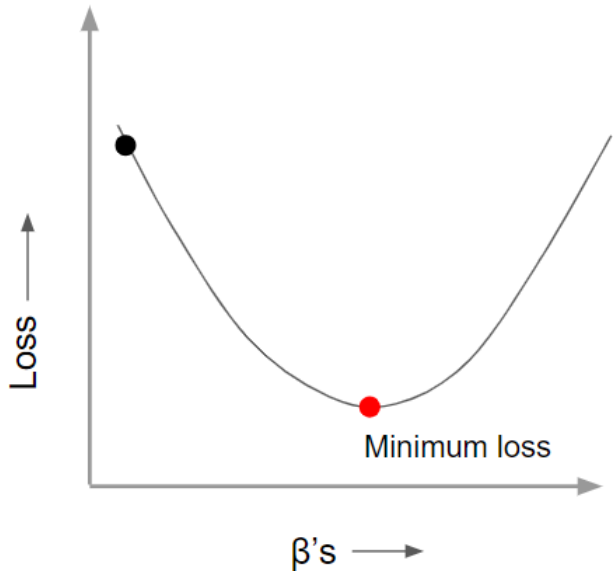
But before that, you need to estimate the values of β_1 and β_0. If you recall, in linear regression, the aim of the model is to find optimal beta values to minimize the sum of squared errors. In linear regression, the sum of squared errors indicated how different the model is compared to the actual data. Similar role is played by a function called log loss function. Following equation represents the log loss function.

$$L(y|p) = -\frac{1}{n}\sum_{i=1}^{n}[y_i log(p_i) \times (1-y_i) log(1-p_i)]$$

Here, y_i is the target variable containing the final class and p_i is the corresponding probability.

Gradient Descent Optimization of the Log Loss

We have already discussed that in order to complete our logistic regression model, we need to find the values of β that minimize the log loss function. However, there is no open solution for this. We need to calculate different values of the log loss function for different β's to find the minimum value of the log loss function.



We will start with a random value of β, and use a gradient descent approach to iteratively move toward the minimum loss, and in each iteration, we calculate a new β. The new β is calculated by using the following formula.

$$\beta_{new} = \beta_{old} - \eta . \frac{\partial L}{\partial \beta}$$

As the iterations move further, the log loss decreases and betas move towards optimisation.

If the gradient value is high, the step that we are gonna take towards the minimum is going to be large, and a smaller one if the gradient value is low. We can control the step size through the learning rate η. Both these ideas are captured in the term $-\eta . \frac{\partial L}{\partial w}$. Learning rate for logistic regression is a hyperparameter that can be controlled.

THe iteration process will continue until one of the following conditions is satisfied:

1. If the value updates are marginal; that is, they are less than a certain ϵ.
2. If you crossed the threshold on the number of iterations you wanted at the beginning of the method; For example, you want to stop after you've moved 10,000 times.

Till now, we have discussed all the functions we discussed containing only one variable as it is easier to visualize. For multiple variables, the sigmoid function transforms as follows:

$$p_i = \frac{1}{1+e^{-(\beta_0+\beta_1 b_{1i}+....)}}$$

All the other functions take p_i as an input, so they will not change much in terms of notation.

Let us summarize the mathematics behind logistic regression we have learned till now:

1. The sigmoid function can be used to find the function that relates inputs to outputs in case of classification problems.

$$p_i = \frac{1}{1+e^{-(\beta_0+\beta_1 b_{1i}+....)}}$$

2.s in the sigmoid are found by minimizing the log loss function

$$L(y|p) = -\frac{1}{n}\sum_{i=1}^{n}[y_i log(p_i) \times (1-y_i) log(1-p_i)]$$

2. There is no open solution to find the minimum of the log loss function, so we try to find different sets of betas and the corresponding values of the log loss function to find the minimum. After initializing the process with a random set of betas, we find new betas using the gradient descent approach.

$$\beta_{new} = \beta_{old} - \eta . \frac{\partial L}{\partial \beta}$$

As mentioned earlier, all of the mathematical parts are taken care of by python in the background. You just need to understand the functions and procedures that are involved.

Evaluating the model:

Evaluating the model refers to testing model predictability. How accurate are the model predictions as compared to actual labels.

A model must perform at the same level for both training and testing data sets. This phenomenon is called generalizability.

Generalizability refers to the ability of a model to make an accurate prediction even in

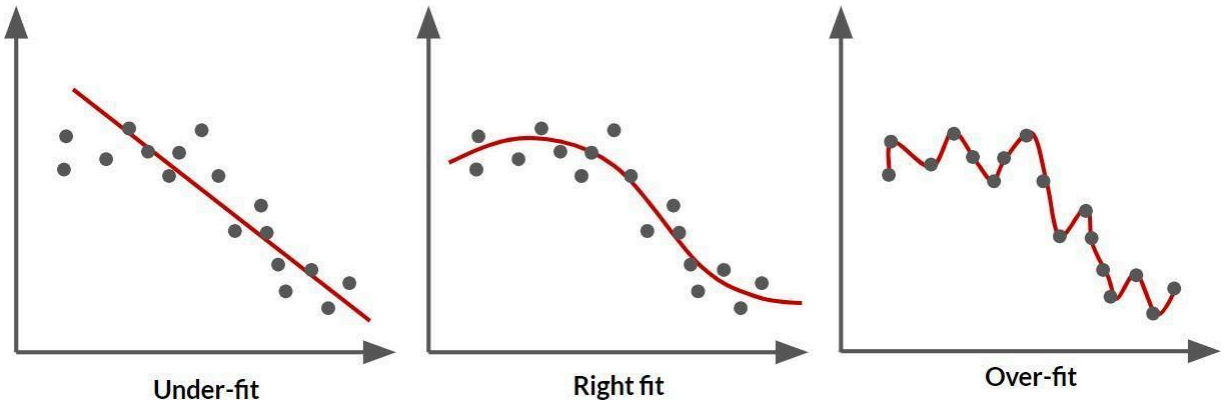
unfamiliar situations.

When discussing the generalizability of an ML model, we must consider two terms associated with model performance—overfitting and underfitting. Here is a summary of these terms:

Overfitting: Overfitting occurs when a model tries to map every input-output pair in a data set. Any noise in the data set is not accounted for, which reduces the model's efficiency and accuracy. When a model is overfitted, it loses generalizability.

Underfitting: Underfitting is the opposite of overfitting. It occurs when a model cannot identify the underlying trend in the data set. That prevents the model from learning anything from the data set, thus resulting in less accuracy. Like an overfitted model, an under-fitted model does not have good generalizability.

The image below illustrates the concepts of overfitting and underfitting.



You can effectively test an ML model by measuring its performance on unseen data. You may not always have access to unseen data, so it is common to divide your data set into these three parts:

Training data set: A training data set is used to fit the model to the data to estimate the model parameters. This is the most significant chunk of the data set available.

Validation data set: A validation data set is used to fine-tune the model further and simplify it if it is too complex.

Test data set: After training and validating a model, you can test the model against this data

set to evaluate its performance.

The ratio in which you split the data varies case by case.

Using the train and test split to refine your model is a bad practice. The process of repeatedly testing and fine-tuning a model using a test data set is called multiple hypothesis testing. It would be best to avoid this at all costs because it may result in overfitting the model to the data set. Test set is to be used only when you have a final model to test against unseen data.

Let's recall what we can do to avoid multiple hypothesis testing:

1. Gather a new test data set, and test the model on it.
2. Split the test data set into smaller parts, and use each part separately.
3. When we have limited data, we can reshuffle the data that forms the training, validation, and test data sets to get a fresh perspective on our data every time we test the model.

While splitting the data, you take random samples from a data set. This may give rise to variability that was not present initially. To avoid this, you can use one of the following methods.

1. Collect more unseen data for testing
2. Cross-validate the training data(will be discussed in a later module)

Evaluation of Classification models

For binary classification models, outcomes are divided into two classes:

- Positive Class: This is a phenomenon of interest
  Usually denoted by 1
  Examples:
  - For the credit approval model, defaulting on a loan is denoted by 1.
  - In a model built for identifying a cat in an image, the presence of a cat is denoted by 1
- Negative Class: The phenomenon that does not interest us
  Usually denoted by 0

Accuracy:

Accuracy is the number of correct predictions divided by the total number of predictions. It can be calculated as:

$$\frac{\textit{Number of correct predictions}}{\textit{Total number of predictions}}$$

To use accuray effectively you need to set a baseline. For example, assume that your training data has 97% of positive class. If your classifier predicts every case as a positive outcome, then your accuracy will still come out as 97% which might seem like a very good accuracy.

But 97% can be your baseline accuracy on which you can try to improve. This baseline accuracy can be found using a dummy classifier.

Confusion Matrix:

For building a confusion matrix, a table is created where:

- The columns are the ground truth (actually positive and actually negative); and
- The rows are the predicted values (predicted positive and predicted negative).

The configuration of the confusion matrix can be changed, so instead of relying on the position of the quadrants rely on the interaction of rows and columns.

This divides your data into four quadrants:

1. True positive quadrant: This is the top-left quadrant on the confusion matrix. It indicates the number of times the model has predicted that the applicant would default, and they did.
2. True negative quadrant: This quadrant is diagonally opposite to the true positive quadrant on the lower-right side of the matrix. It indicates the number of times the model has predicted that the applicant would not default, and they did not.
3. False-positive quadrant: This is the top-right quadrant on the confusion matrix. It indicates the number of times the algorithm has predicted that the applicant would not default, but they did.
4. False-negative quadrant: This is the lower-left quadrant on the confusion matrix. It indicates the number of times the algorithm has predicted that the applicant would default, but they did not

Following is an example of a confusion matrix.

Sample Size: 10,000		Ground Truth	
		Actually Positive	Actually Negative
Predicted	Predicted Positives	True Positives 81	False Positives 23
	Predicted Negatives	False Negatives 252	True Negatives 9644

You need to notice that the position of these quadrants might change for different formats and usages.

- Precision: The proportion of actual positives, out of all the samples that our model has predicted as positive.

$$Precision = \frac{TP}{TP + FP}$$

- Recall (Sensitivity): The proportion of correctly predicted positives, out of all the actually positive samples.

$$Recall = \frac{TP}{TP + FN}$$

- F1 measure: If your application demands that both precision and recall be high you can use a combined metric, i.e., the F1 score. The F1 measure is the harmonic mean of precision and recall.

$$\text{F1 Score} = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

Probability Predictions:

Many classification algorithms do not return explicit labels, i.e., 0s and 1s. Instead they return probabilities of a data point belonging to one of the cases. You can set a threshold based on

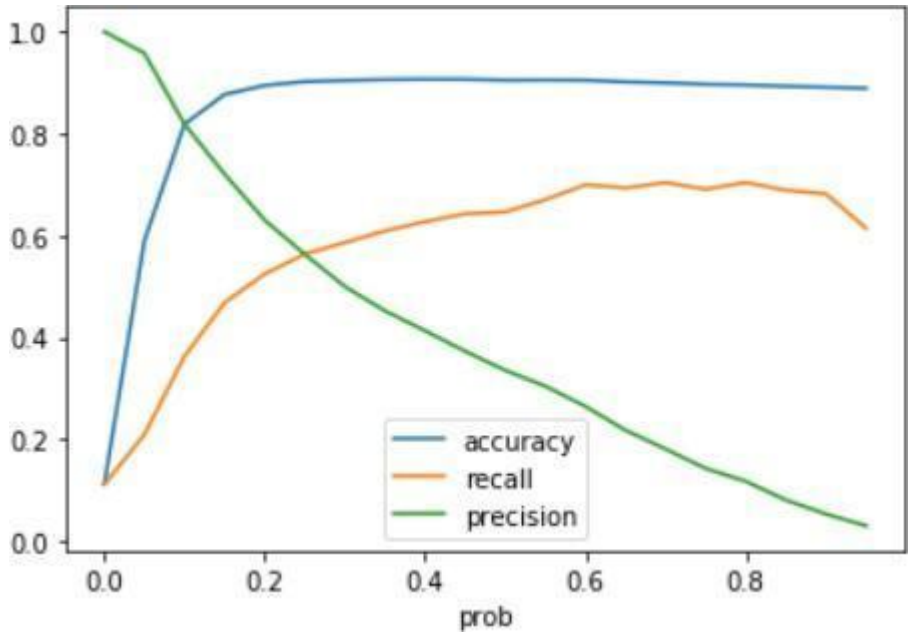
which you can make the final decision. For instance

If the probability of belonging to the positive class > threshold, then class = 1.

If the probability of belonging to the positive class < threshold, then class = 0.

As the threshold increases, recall decreases and precision increases, and vice versa. Depending on which measure is more important to your model, you can set the threshold accordingly.

Given below is a graph indicating what accuracy, precision, and recall values look like at varying thresholds.



Note that the exact precision and recall trade-off graph will be different for different models, but it generally follows a similar trend.

ROC Curve:

A receiver operating characteristic ROC curve signifies the model's ability to discriminate between classes. It helps in determining the threshold probability. It combines the true positive rates and false-positive rates from the confusion matrix and plots them on a graph.

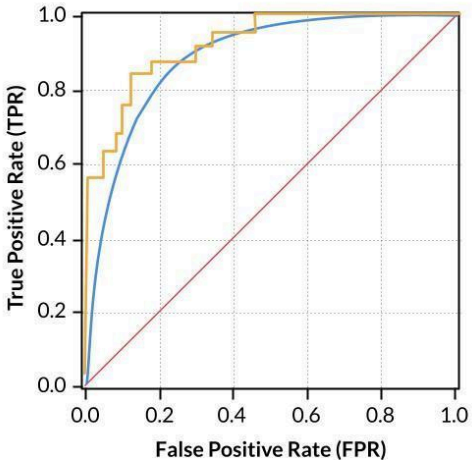
Here are the steps to follow for plotting a ROC curve:

1. Consider various threshold values.
2. For each threshold, calculate TPR and FPR.
3. Plot the data on an FPR (x-axis) vs. TPR (y-axis) grid.

All ROC curves start from the origin and end at the (1,1) point. Between these two points, the ideal ROC curve is a right-angle line passing through the point (0,1). The worst model is a straight line passing through (0,0) and (1,1). All the other real models lie between these two lines.

In most cases, we use discrete threshold values to plot ROC curves, so we get stepwise functions, as shown in the figure below in yellow, instead of a smooth curve like the blue line.

ROC CURVE CREATED WITH A FEW THRESHOLDS



There are two ways to use the ROC curve to evaluate a model's performance:

- You can look at the nature of the curve. A model that discriminates better will be close to the ideal curve, whereas a poorer model will be closer to the straight line.
- You can calculate the area under the ROC curve, popularly known as the ROC AUC.

An ideal curve will have a ROC AUC of 1, whereas a random guess will have a ROC AUC of 0.5.

Choosing Evaluation Metrics:

Misclassification has more implications in some cases than others. Based on the problem you are trying to solve, you will have to choose the metric for evaluating the model.

The choice of the modifications you are making to the model will be affected by the evaluation metric.

Imbalanced Data Sets:

A data set is said to have an imbalance when one class significantly outnumbers the other. Data imbalance causes certain problems:

- In the case of data with class imbalance, accuracy may not be the right metric, as discussed earlier.
- When a model considers the positive class as a rare event and does not learn the patterns associated with this class. As a result, identifying the rare class becomes even more difficult.

To deal with the imbalance in data sets, you need to apply one of the following measures:

- Subsample by reducing samples in the majority class.
- Oversample the minority class.
- Correct the model to compensate for the bias (case-control sampling).

However, you need to be cautious about the artificial balancing of data sets. You might lose the variance while removing samples in the majority class. And conversely, if you upsample the minority class, you might reinforce certain patterns related to the positive class. This can cause the model to make overconfident predictions. It is advised to deal with data imbalance in an iterative process where you try different ratios of different classes.

Python Demonstration

There are examples throughout this module related to loan approvals based on the personal and financial information of the customers. In the python demonstration, we used the same problem. It is rare to get clean data ready for analysis. So, like in the linear regression module, you started with data preparation and visualization.

We checked for null values and found that no columns have null values in them.

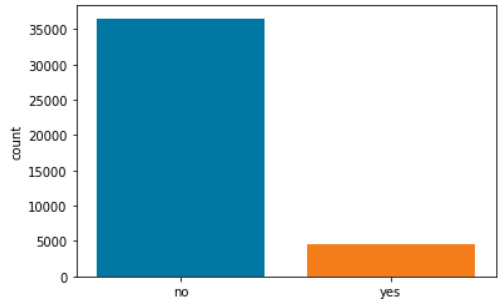
```
print(bankData.isnull().sum())
```

```
age               0
job               0
marital           0
education         0
default           0
housing           0
loan              0
contact           0
month             0
day_of_week       0
duration          0
campaign          0
pdays             0
previous          0
poutcome          0
emp.var.rate      0
cons.price.idx    0
cons.conf.idx     0
euribor3m         0
nr.employed       0
y                 0
dtype: int64
```

You saw that the data is imbalanced with a lot of responses in the column being yes.

```
# Labels against frequency
plt.figure()
sb.countplot(x='y', data=bankData)

plt.show()
```



Also, you created dummy variables for categorical variables, and removed the respective original columns such that there was no loss in information.

```
# Create dummy variables for all the categorical variables that you have kept.
df2 = pd.get_dummies(bankData, columns=['job', 'marital', 'default', 'housing', 'loan', 'poutcome', 'education','contact'])
```

```
# Drop one variable from each of the sets of dummy vraibles. No information will be lost.
df2.drop( ['job_unknown', 'marital_unknown' , 'default_unknown', 'housing_unknown', 'loan_unknown' , 'poutcome_nonexistent'
          ,'education_unknown','contact_telephone'], axis=1, inplace = True )

#Drop variables that do might not have an impact on the output
df2.drop( ['month', 'day_of_week'], axis=1, inplace = True )
```

After that, you split the data between training and testing data in the same way as you did for simple and multiple linear regression demonstrations.

```
# converting y from 'yes' and 'no' to 1 and 0
df2['y'] = df2.y.map(dict(yes=1, no=0))
```

```
# Split the data into a training and test set.

# All columns are features execpt for y
X = df2.loc[:, ['age', 'duration', 'campaign', 'pdays', 'previous', 'emp.var.rate',
       'cons.price.idx', 'cons.conf.idx', 'euribor3m', 'nr.employed',
       'job_admin.', 'job_blue-collar', 'job_entrepreneur', 'job_housemaid',
       'job_management', 'job_retired', 'job_self-employed', 'job_services',
       'job_student', 'job_technician', 'job_unemployed', 'marital_divorced',
       'marital_married', 'marital_single', 'default_no', 'default_yes',
       'housing_no', 'housing_yes', 'loan_no', 'loan_yes', 'poutcome_failure',
       'poutcome_success', 'education_basic.4y', 'education_basic.6y',
       'education_basic.9y', 'education_high.school', 'education_illiterate',
       'education_professional.course', 'education_university.degree',
       'contact_cellular']]

# y is the target variable
y = df2.loc[:, ['y']]
```

```
#import a prebuilt function to split the data
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

Let us recall the steps involved in model building.

1. First, we choose the family of models.
2. Next, we estimate the parameters.
3. Then we reduce the complexity of the model.

Here, you imported the LogisticRegression function from the sklearn.linear_model library and created an object. Refer to the code below:

```
# Fit your training data to a Logistic Regression Model
from sklearn.linear_model import LogisticRegression

# diclare the logistic model
model = LogisticRegression()
```

You used the fit() to train the model using the training data set and then used the coef method to get the β values of all the independent variables. Here is what the output looked like.

```
model.coef_
```

```
array([[ 3.36681685e-03,  4.48562022e-03, -7.03026507e-02,
        -1.68222953e-03,  1.76680568e-02, -2.60561800e-01,
         1.91184340e-01,  4.33534203e-02, -2.76033640e-01,
        -3.42029337e-03,  1.01936882e-02, -2.24273101e-02,
        -8.67140719e-04, -3.38207740e-04,  2.25035609e-03,
         8.62107704e-03, -3.41854422e-04, -5.59794817e-03,
         8.41992440e-03,  1.61874005e-03,  1.46498182e-03,
        -2.64274853e-03, -1.44393758e-02,  1.96592230e-02,
         2.77328703e-02, -3.21739554e-06, -2.01043579e-03,
         5.07868679e-03,  4.76553535e-03, -1.69728434e-03,
         7.30231646e-03,  3.19401959e-03, -4.59417031e-03,
        -3.91532749e-03, -9.81446191e-03, -2.24170139e-03,
         1.34212574e-04,  1.38894505e-03,  1.94208626e-02,
         4.03311904e-02]])
```

In order to gauge the performance of the model, we first look at the training set error. Here, you see that you can use the LogisticRegression_Object_Name.predict(X_train) command and look at the accuracy of the model.

```
# Use the testing data to make predictions and determine the accuracy of your predictions.
y_pred = model.predict(X_test)

from sklearn import metrics
print(metrics.accuracy_score(y_test, y_pred))
```

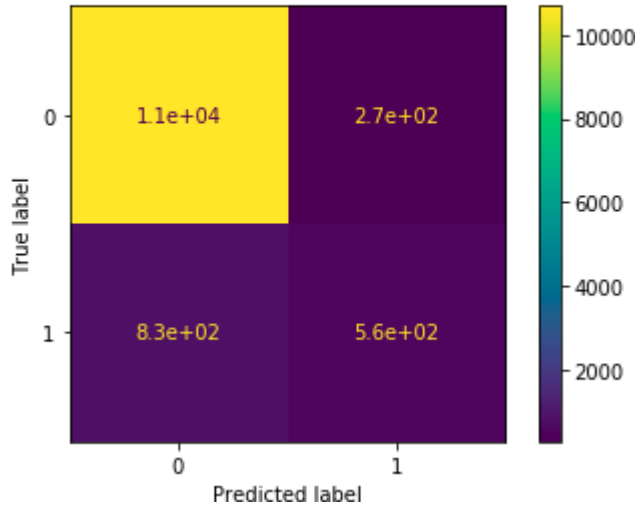
The argument strategy = "most_frequent" refers to the strategy where the most frequent class is the prediction for all the rows.

You also plotted a confusion matrix to help with the evaluation of your model.

```
# Plot and print confusion matrix
cnf_matrix = metrics.confusion_matrix(y_test, y_pred)
print(cnf_matrix)
metrics.plot_confusion_matrix(model, X_test, y_test)
plt.show()
```

Here's the output.

```
[[10701   268]
 [  829   559]]
```



Finally, you evaluated the model using test data where the accuracy score did not improve much. You also evaluated the model using roc_auc_score from sklearn.metrics.