Logistic Regression

Binary logistic regression: model formulation, interpretation, and evaluation.

Binary Logistic Regression

Logistic regression is a powerful statistical method that extends beyond the capabilities of simple linear regression, particularly when dealing with binary (yes/no, male/female, high/low) outcomes. Unlike linear regression, which struggles with dichotomous dependent variables, logistic regression excels by analyzing how various independent variables influence a binary outcome.

This technique simultaneously examines all independent variables in a single analysis. This approach not only evaluates the predictive strength of each variable but also accounts for the influence of other variables in the model, ensuring a comprehensive understanding of their effects on the binary outcome.

Binary logistic regression is a type of regression analysis used when the dependent variable is binary. The goal of binary logistic regression is to predict the probability that an observation falls into one of the two categories based on one or more independent variables.

Logistic Regression

Logistic regression is a statistical model that uses the logistic function to model the probability of the binary outcome. Unlike linear regression which predicts continuous outcomes logistic regression predicts the probability of the categorical outcome.

Probability and Odds in Logistic Regression

Logistic regression models the probability of the event occurring using the odds ratio. Odd ratio compares the probability of the success to the probability of the failure, providing insight into the relationship between variables and outcomes. Odds ratios greater than 1 indicate higher odds of the event occurring while those less than 1 suggest lower odds. The logistic function transforms linear regression output into the probabilities bounded between the 0 and 1.

For example, Predicting the likelihood of the customer buying a product based on the demographic variables.

Application: Widely used in the fields like medicine, finance and social sciences to the predict binary outcomes.

Model Fitting in Binary Logistic Regression

- **Parameter Estimation:** Fitting a binary logistic regression model involves estimating coefficients for the independent variables.
- **Maximum Likelihood Estimation (MLE):** Common method used to the find parameter estimates that maximize the likelihood of the observed data.
- **Gradient Descent:** Optimization algorithm used when MLE is computationally expensive or infeasible.

Logistic Regression

- **Iterative Process:** Model fitting is an iterative process where coefficients are adjusted until the model converges.
- **Goodness of Fit:** Measures like AIC and BIC help assess the fit of the model to the data.
- **Overfitting and Regularization:** Techniques like ridge and lasso regression are employed to the prevent overfitting and improve model generalization.

Model Evaluation and Validation

- **Cross-Validation:** Technique to the assess how well a model generalizes to the new data by splitting the dataset into the training and testing subsets.
- **ROC Curve Analysis:** Receiver Operating Characteristic curve evaluates the trade-off between the sensitivity and specificity.
- **Area Under Curve (AUC):** AUC measures the overall performance of the model in the distinguishing between the classes.
- **Confusion Matrix Analysis:** Evaluates the performance of the classification model by the comparing predicted and actual values.
- **Precision, Recall, and F1 Score:** Metrics used to the evaluate the performance of the binary classification models.
- **Validation Set Approach:** Divides data into the training, validation and test sets to the tune model hyperparameters and assess performance.

Binary Vs Multinomial Logistic Regression

Differences between binary logistic regression and multinomial logistic regression is shown in the table added below:

Ascept	Binary Logistic Regression	Multinomial Logistic Regression
Number of Outcome Categories	Binary logistic regression deals with the two outcome categories.	Multinomial logistic regression deals with more than two outcome categories.
Model Complexity	Binary logistic regression is simpler as it involves for single categories.	Multinomial logistic regression is more complex than binary as it accounts for the multiple categories.

Logistic Regression

Ascept	Binary Logistic Regression	Multinomial Logistic Regression
Interpretation of Coefficients	In binary logistic regression coefficients represent the log odds ratio of the event occurring	In multinomial they compare each category to the reference category.
Applications	Binary logistic regression is used when outcomes are dichotomous like yes/no or success/failure.	Multinomial is employed when there are multiple levels or categories.
Data Structure	Binary logistic regression deals with the binary outcomes.	Multinomial logistic regression requires the outcome variable to be nominal or ordinal.

Practical Applications of Binary Logistic Regression

Binary logistic regression finds applications in the various fields such as:

- Predicting the likelihood of the disease occurrence based on the risk factors in the medical research.
- Assessing the probability of the default on the loan in financial analysis.
- Determining customer churn in the marketing analytics.
- Identifying sentiment polarity in the text classification.

 # Exercise

Problems on Binary Logistic Regression

Q1. Predicting the probability of the customer buying a product based on the demographic information.

Q2. Estimating the likelihood of the patient having a specific disease based on the medical test results.

Q3. Analyzing the factors influencing employee attrition in a company.

Q4. Assessing the risk of credit card fraud based on the transaction patterns.

Conclusion

Logistic Regression

Binary logistic regression is a powerful statistical tool for the analyzing binary outcome variables and identifying the predictors associated with them. By understanding its methodology, interpretation and practical applications researchers and analysts can make informed the decisions and draw meaningful conclusions from the their data.

Multinomial logistic regression: handling multiple classes.

Logistic regression is a popular machine learning algorithm used for binary classification tasks. It models the probability of the output variable (also known as the dependent variable) given the input variables (also known as the independent variables). It is a linear algorithm that applies a logistic function to the output of a linear regression model, which transforms the continuous output into a probability between 0 and 1.

Logistic regression has several variants, including binary logistic regression, multinomial logistic regression, and ordinal logistic regression. Binary logistic regression is used for binary classification tasks, where there are only two possible outcomes for the output variable. Multinomial logistic regression, also known as softmax regression, is used for multi-class classification tasks, where there are more than two possible outcomes for the output variable. Ordinal logistic regression is used for ordered multi-class classification tasks, where the outcomes have a natural ordering (e.g. low, medium, high).

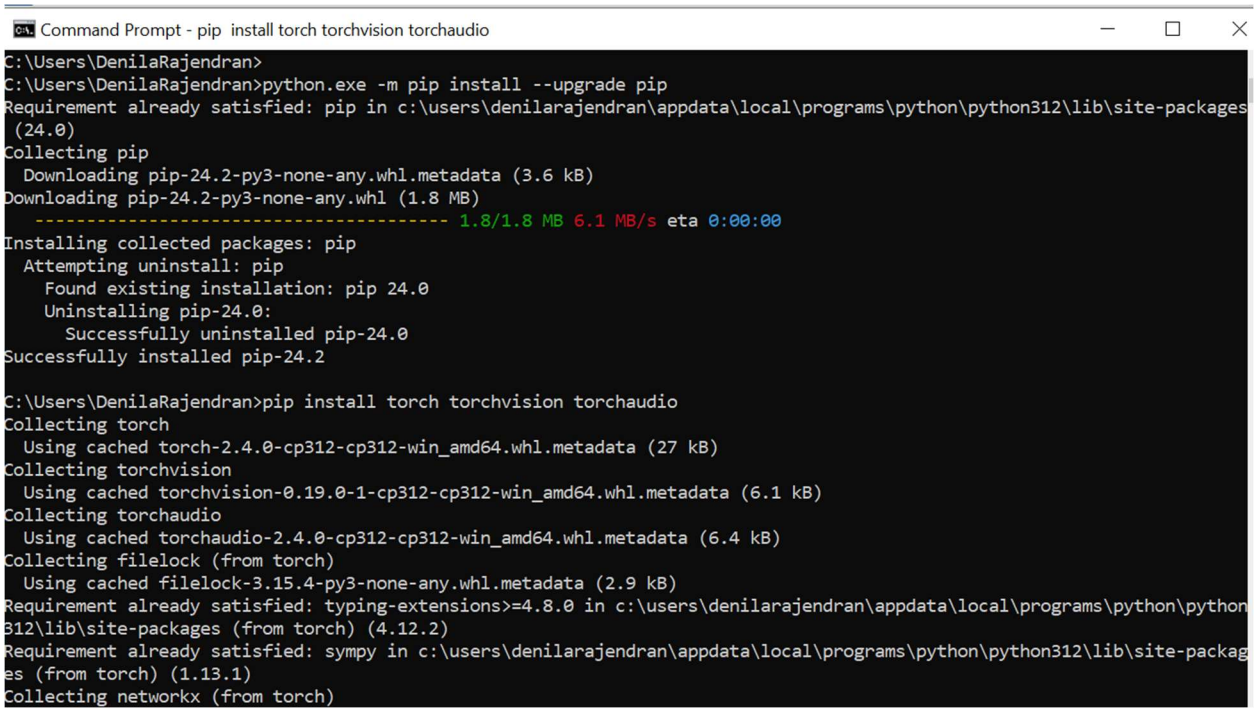
Data Preparation

For our example, we will be using the famous Iris dataset, which contains measurements of the sepal length, sepal width, petal length, and petal width for three species of iris flowers (Iris setosa, Iris versicolor, and Iris virginica). The goal is to predict the species of an iris flower based on these measurements.

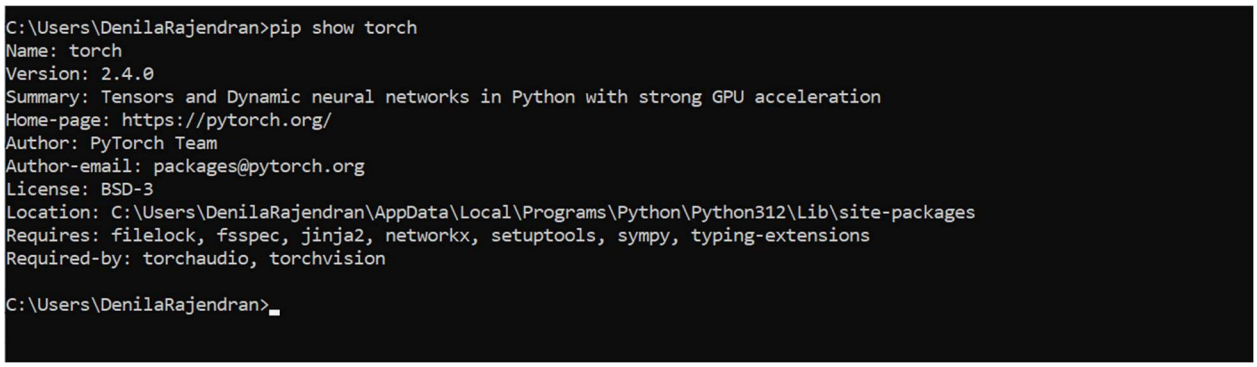
Before training our multinomial logistic regression model, we need to preprocess our data. In this case, we will normalize our input data to have **a mean of 0 and a standard deviation of 1.** This helps to ensure that our model trains more efficiently and effectively.

Pytorch: PyTorch

Logistic Regression



Pip show torch:




```
# Import the necessary libraies
import torch
from sklearn.datasets import load_iris
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import train_test_split
import torch.nn as nn
from torchinfo import summary
```

Logistic Regression

```
# Load the Iris dataset
iris = load_iris()

# Convert the data to PyTorch tensors
X = torch.tensor(iris.data, dtype=torch.float32)
y = torch.tensor(iris.target, dtype=torch.long)

# Normalize the input data
mean = torch.mean(X, dim=0)
std = torch.std(X, dim=0)
X = (X - mean) / std

# Split the dataset into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
random_state=42)

# Create PyTorch Datasets
train_dataset = TensorDataset(X_train, y_train)
val_dataset = TensorDataset(X_val, y_val)

# Define the data loaders
batch_size = 16
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
```

Logistic Regression

```
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
```

Model Creation

Multinomial logistic regression is a type of logistic regression that is used when there are three or more categories in the dependent variable. It models the probability of each category using a separate logistic regression equation, and then selects the category with the highest probability as the predicted outcome.

We can implement multinomial logistic regression using PyTorch by defining a neural network with a single linear layer and a softmax activation function. The linear layer takes in the input data and outputs a vector of logits (i.e. unnormalized log probabilities), which are then passed through the softmax function to obtain a vector of probabilities.

- Python3

```
class LogisticRegression(nn.Module):
    def __init__(self, input_size, num_classes):
        super(LogisticRegression, self).__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        out = self.linear(x)
        out = nn.functional.softmax(out, dim=1)
        return out
```

In the above code, we define a PyTorch module called LogisticRegression that inherits from nn.Module. We define a single linear layer with input size 4 (i.e. the number of input features) and output size num_classes (i.e. the number of output classes). We then define the forward() method, which takes in the input data x and passes it through the linear layer and softmax activation function.

Logistic Regression

Print the model summary

Instantiate our model by creating an instance of the LogisticRegression class with input_size=4 and num_classes=3.

If a GPU is available, we set the device to ‘cuda:0’, otherwise, we set it to ‘cpu’.Next, we move the model to the specified device using the to() method. This is done so that the model can utilize the hardware resources of the device during training.

Print the model summary.

- Python3

```
# Define the model
model = LogisticRegression(input_size=4, num_classes=3)


# Check for cuda
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
# Move the model to the device
model = model.to(device)
summary(model, input_size=(16,4))
```

Output:

```
==========================================================================
==================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
==================================
LogisticRegression                       [16, 3]                   --
├─Linear: 1-1                            [16, 3]                   15
==========================================================================
==================================
```

Logistic Regression

Total params: 15

Trainable params: 15

Non-trainable params: 0

Total mult-adds (M): 0.00

==========

==========

Input size (MB): 0.00

Forward/backward pass size (MB): 0.00

Params size (MB): 0.00

Estimated Total Size (MB): 0.00

==========

==========

Define a loss function and an optimizer

To train our model, we need to define a loss function and an optimizer. For this example, we will use cross-entropy loss and stochastic gradient descent (SGD) optimizer.

- Python3

```
# Define the loss function and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.SGD(model.parameters(), lr=0.002)
```

In the above code, we define our loss function as nn.CrossEntropyLoss(), which is commonly used for multi-class classification problems. We also define our optimizer as torch.optim.SGD(), which implements stochastic gradient descent with a learning rate of 0.002. We pass in our model parameters using model.parameters().

Now that we have defined our model, loss function, and optimizer, we can move on to training the model using our training dataset.

Logistic Regression

Training and Validation

To train our model, we need to define a few parameters such as the number of epochs, batch size, and the device to use for training (CPU or GPU).

First, we define the training parameters such as the number of epochs to train the model and the device to use for training.

Then, we start training the model using a nested loop. In the outer loop, we iterate over the specified number of epochs. In the inner loop, we iterate over the batches of data in the training DataLoader.

For each batch, we move the input data and labels to the specified device using the to() method. Then, we perform a forward pass through the model using the input data to get the model's predictions. We calculate the loss using the predicted outputs and the actual labels using the specified loss function.

We then perform backpropagation to compute the gradients of the loss with respect to the model's parameters using the backward() method. We zero out the gradients using optimizer.zero_grad() before performing backpropagation to prevent gradient accumulation. We then update the model's parameters using the optimizer's step() method.

After each epoch, we print the training loss for that epoch using the print() function.

- Python3

```
# Define training parameters
num_epochs = 1000


# Train the model
for epoch in range(num_epochs):
    for i, (inputs, labels) in enumerate(train_loader):
        # Move inputs and labels to the device
        inputs = inputs.to(device)
```

Logistic Regression

```
        labels = labels.to(device)


        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)


        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()


    # Print training loss for each epoch
    if (epoch+1)%100 == 0:
        print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs,
loss.item()))
```

Output:

Epoch [100/1000], Loss: 0.8293

Epoch [200/1000], Loss: 0.8145

Epoch [300/1000], Loss: 0.9410

Epoch [400/1000], Loss: 0.9489

Epoch [500/1000], Loss: 0.7749

Epoch [600/1000], Loss: 0.8242

Epoch [700/1000], Loss: 0.7823

Epoch [800/1000], Loss: 0.7413

Epoch [900/1000], Loss: 0.7628

Logistic Regression

Epoch [1000/1000], Loss: 0.7471

In this output, we can see that the training loss decreases with each epoch, indicating that our model is learning from the training data. We can see that the training loss starts at 0.8293 and gradually decreases over the epochs, indicating that the model is learning to make better predictions. The loss fluctuates a bit but generally trends downwards.

There are a few ways we can further improve our model's performance. One way is to adjust the hyperparameters such as the learning rate, number of layers, and number of neurons in each layer. We can also use different optimization algorithms such as Adam or SGD to update the model parameters. Additionally, we can try using different types of regularization such as L1 or L2 regularization to prevent overfitting. We can also use techniques such as data augmentation to increase the size of our training dataset and reduce overfitting.

Evaluations

After training our model, we can evaluate its performance on the validation set by looping over the validation dataset and computing the model's predictions for each example. We can then calculate the accuracy of the model by comparing its predictions to the true labels.

In this code block, we evaluate the performance of the trained model on the validation set.

We first wrap the evaluation code in a "torch.no_grad()" context, which tells PyTorch that we don't need to keep track of the gradients during this evaluation. This can save memory and computation time.

Next, we initialize "correct" and "total" counters to keep track of the number of correctly classified examples and the total number of examples, respectively. We then loop over the validation set using the "val_loader" created earlier.

For each batch in the validation set, we move the inputs and labels to the device (either CPU or GPU), just like we did during training. We then compute the model's predictions by passing the inputs through the model, and taking the argmax of the output tensor to get the predicted class labels.

We update the "total" counter with the number of examples in this batch, and add the number of correctly classified examples to the "correct" counter. Note that we use the ".item()" method to convert the PyTorch tensor to a scalar value.

Logistic Regression

Finally, we print the validation accuracy as a percentage, which is calculated as the number of correctly classified examples divided by the total number of examples in the validation set, multiplied by 100.

- Python3

```
# Evaluate the model on the validation set
with torch.no_grad():
    correct = 0
    total = 0
    for inputs, labels in val_loader:
        # Move inputs and labels to the device
        inputs = inputs.to(device)
        labels = labels.to(device)

        # Compute the model's predictions
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)

        # Compute the accuracy
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Validation Accuracy: {:.2f}%'.format(100 * correct / total))
```

Output:

Validation Accuracy: 90.00%

Logistic Regression

The validation accuracy is displayed as 90.00%. This means that when the model was evaluated on the validation set, it correctly classified 90.00% of the samples. This indicates that the model has learned to generalize well on unseen data.

Evaluation with AUC - ROC (Area Under the Receiver Operating Characteristic Curve)

What is the AUC-ROC curve?

The AUC-ROC curve, or Area Under the Receiver Operating Characteristic curve, is a graphical representation of the performance of a binary classification model at various classification thresholds. It is commonly used in machine learning to assess the ability of a model to distinguish between two classes, typically the positive class (e.g., presence of a disease) and the negative class (e.g., absence of a disease).

Let's first understand the meaning of the two terms **ROC** and **AUC**.

- **ROC**: Receiver Operating Characteristics
- **AUC**: Area Under Curve

Receiver Operating Characteristics (ROC) Curve

ROC stands for Receiver Operating Characteristics, and the ROC curve is the graphical representation of the effectiveness of the binary classification model. It plots the true positive rate (TPR) vs the false positive rate (FPR) at different classification thresholds.

Area Under Curve (AUC) Curve:

AUC stands for the Area Under the Curve, and the AUC curve represents the area under the ROC curve. It measures the overall performance of the binary classification model. As both TPR and FPR range between 0 to 1, So, the area will always lie between 0 and 1, and A greater value of AUC denotes better model performance. Our main goal is to maximize this area in order to have the highest TPR and lowest FPR at the given threshold. The AUC measures the probability that the model will assign a randomly chosen positive instance a higher predicted probability compared to a randomly chosen negative instance.

It represents the probability with which our model can distinguish between the two classes present in our target.

ROC-AUC Classification Evaluation Metric

Key terms used in AUC and ROC Curve

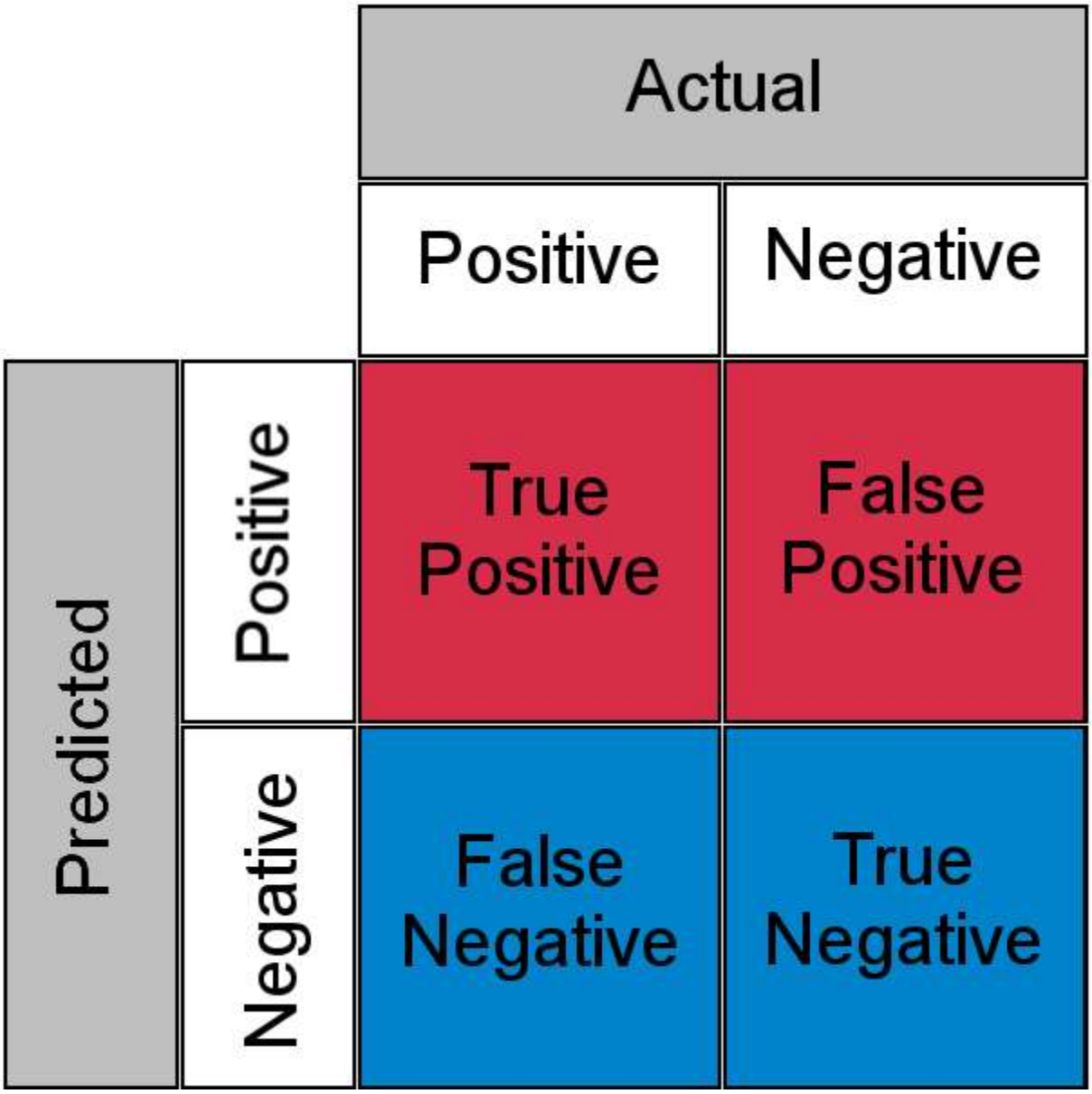
1. TPR and FPR

Logistic Regression

This is the most common definition that you would have encountered when you would Google AUC-ROC. Basically, the ROC curve is a graph that shows the performance of a classification model at all possible thresholds(threshold is a particular value beyond which you say a point belongs to a particular class). The curve is plotted between two parameters

- **TPR** – True Positive Rate
- **FPR** – False Positive Rate

Before understanding, TPR and FPR let us quickly look at the confusion matrix.



Confusion Matrix for a Classification Task

- **True Positive**: Actual Positive and Predicted as Positive

Logistic Regression

- **True Negative**: Actual Negative and Predicted as Negative
- **False Positive(Type I Error)**: Actual Negative but predicted as Positive
- **False Negative(Type II Error)**: Actual Positive but predicted as Negative

In simple terms, you can call False Positive a **false alarm** and False Negative a **miss**. Now let us look at what TPR and FPR are.

2. Sensitivity / True Positive Rate / Recall

Basically, TPR/Recall/Sensitivity is the ratio of positive examples that are correctly identified. It represents the ability of the model to correctly identify positive instances and is calculated as follows:

Sensitivity/Recall/TPR measures the proportion of actual positive instances that are correctly identified by the model as positive.

3. False Positive Rate

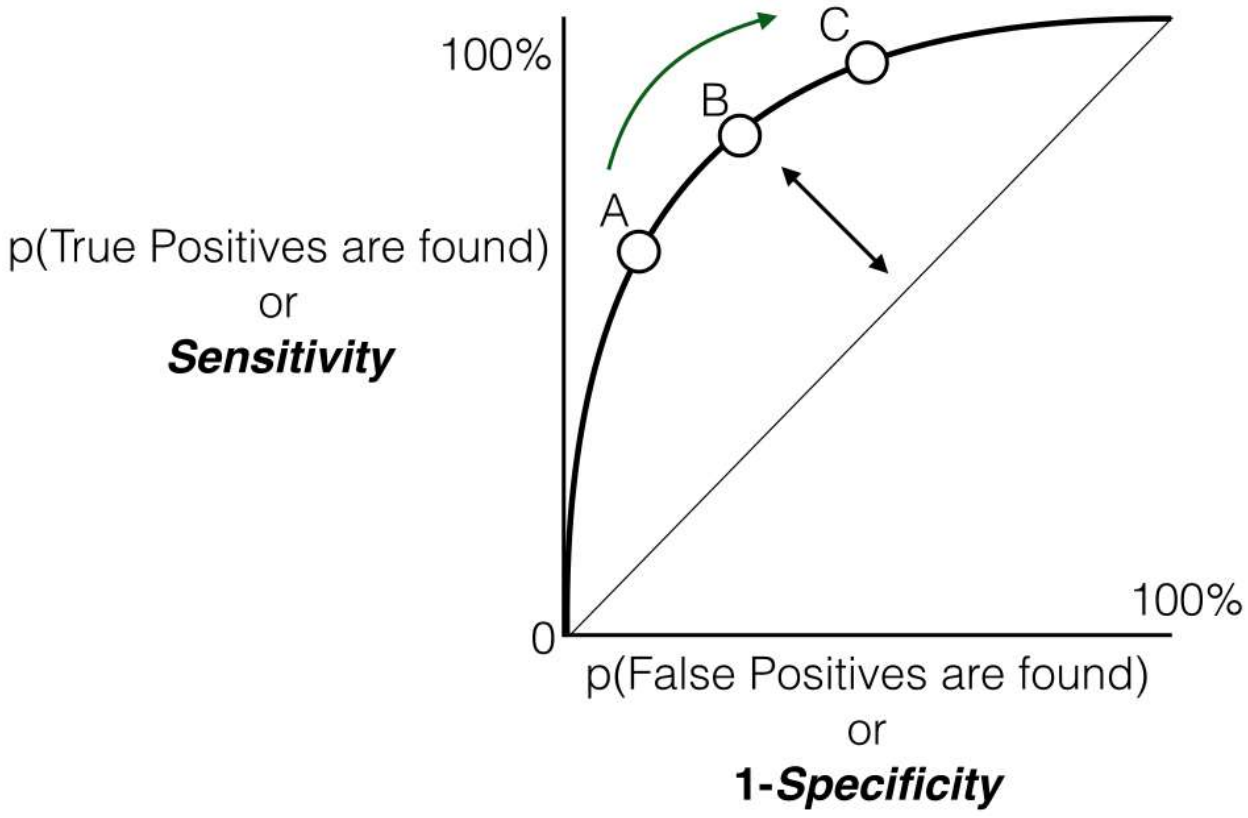
FPR is the ratio of negative examples that are incorrectly classified.

4. Specificity

Specificity measures the proportion of actual negative instances that are correctly identified by the model as negative. It represents the ability of the model to correctly identify negative instances

And as said earlier ROC is nothing but the plot between TPR and FPR across all possible thresholds and AUC is the entire area beneath this ROC curve.

Logistic Regression



Sensitivity versus False Positive Rate plot

Relationship between Sensitivity, Specificity, FPR, and Threshold.

Sensitivity and Specificity:

- **Inverse Relationship:** sensitivity and specificity have an inverse relationship. When one increases, the other tends to decrease. This reflects the inherent trade-off between true positive and true negative rates.
- **Tuning via Threshold:** By adjusting the threshold value, we can control the balance between sensitivity and specificity. Lower thresholds lead to higher sensitivity (more true positives) at the expense of specificity (more false positives). Conversely, raising the threshold boosts specificity (fewer false positives) but sacrifices sensitivity (more false negatives).

Threshold and False Positive Rate (FPR):

- **FPR and Specificity Connection:** False Positive Rate (FPR) is simply the complement of specificity (FPR = 1 – specificity). This signifies the direct relationship between them: higher specificity translates to lower FPR, and vice versa.
- **FPR Changes with TPR:** Similarly, as you observed, the True Positive Rate (TPR) and FPR are also linked. An increase in TPR (more true positives) generally leads to a rise in

FPR (more false positives). Conversely, a drop in TPR (fewer true positives) results in a decline in FPR (fewer false positives)

Ref link: https://www.geeksforgeeks.org/auc-roc-curve/

Thank you