Intro to classification - Logistic regression - 2

One should look for what is and not what he thinks should be. (Albert Einstein)

Chat question

- Below are the steps for data cleaning for logistic regression. In the chat, type the words to fill in the blanks:
    i. Make sure the _____ is labeled
    ii. Check for ____
    iii. Encode categorical data into ______ data
    iv. Split into ____ and test sets
    v. Scale _____

Chat question

- Below are the steps for data cleaning for logistic regression. In the chat, type the words to fill in the blanks:
    1. Make sure the **target** is labeled
    2. Check for **NAs**
    3. Encode categorical data into **numerical** data
    4. Split into **train** and test sets
    5. Scale **features**

Loading packages

Let's load the packages we will be using:

```
import os
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
# Helper packages.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path
# Scikit-learn package for logistic regression.
from sklearn import linear_model
# Model set up and tuning packages from scikit-learn.
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
# Scikit-learn packages for evaluating model performance.
from sklearn import metrics
# Scikit-learn package for data preprocessing.
from sklearn import preprocessing
```

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your `course` folder
- Let `data_dir` be the variable corresponding to your `data` folder

```
# Set 'main_dir' to location of the project folder
home_dir = Path(".").resolve()
main_dir = home_dir.parent.parent
print(main_dir)
```

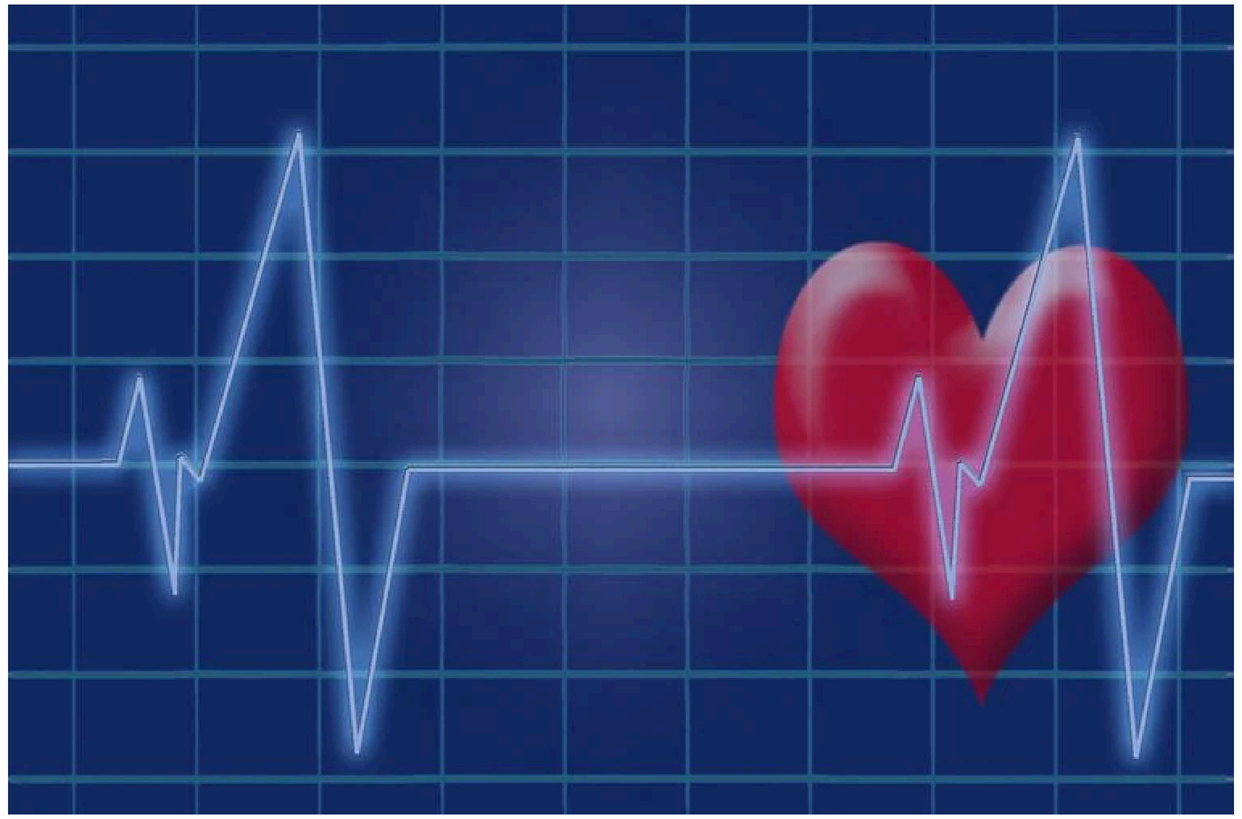
```
data_dir = str(main_dir) + "/data"
print(data_dir)
```

Module completion checklist

Objectives	Complete
Transform categorical variables for implementation of logistic regression	
Implement logistic regression on the data and assess results of classification model performance	

Stroke Prediction survey: case study

- According to the World Health Organization (WHO), stroke is the 2nd leading cause of death globally
- ***Click here*** to see a dataset showing the results of a clinical trial of a heart-disease drug survey on a sample of US adults
- Each row in the data provides relevant information about the adult, including whether they had a stroke or not
- We would like to use this data to predict whether a patient is likely to have a stroke based on their demographic information and medical history



Dataset

- In order to implement what you learn in this course, we will be using the `healthcare-dataset-stroke-data.csv` dataset
- We will be working with columns from the dataset such as:
  - stroke
  - gender
  - age
  - hypertension
  - heart_disease
  - ever_married
- We will be using different columns of the dataset to predict `stroke` as the target variable

Loading data into Python

- Let's load the entire dataset
- We are now going to use the function `read_csv` to read in our `healthcare-dataset-stroke-data.csv` dataset

```
df = pd.read_csv(str(data_dir)+"/"+ 'healthcare-dataset-stroke-data.csv')
print(df.head())
```

```
      id  gender   age  ...   bmi   smoking_status stroke
0   9046    Male  67.0  ...  36.6  formerly smoked      1
1  51676  Female  61.0  ...   NaN     never smoked      1
2  31112    Male  80.0  ...  32.5     never smoked      1
3  60182  Female  49.0  ...  34.4           smokes      1
4   1665  Female  79.0  ...  24.0     never smoked      1

[5 rows x 12 columns]
```

Subset data

- Remove any columns from the dataframe that are not numeric or categorical, as we will not be using them in our models

```
df_subset = df[['age', 'avg_glucose_level', 'heart_disease', 'ever_married', 'hypertension',
'Residence_type', 'gender', 'smoking_status', 'work_type', 'stroke', 'id']]
print(df_subset.head())
```

```
    age  avg_glucose_level  heart_disease  ...      work_type  stroke     id
0  67.0             228.69              1  ...        Private       1   9046
1  61.0             202.21              0  ...  Self-employed       1  51676
2  80.0             105.92              1  ...        Private       1  31112
3  49.0             171.23              0  ...        Private       1  60182
4  79.0             174.12              0  ...  Self-employed       1   1665

[5 rows x 11 columns]
```

Convert target to binary

- Let's convert the target so that it is a binary class if it is not already

```
# Target is binary
print(df_subset['stroke'].head())
```

```
0     1
1     1
2     1
3     1
4     1
Name: stroke, dtype: int64
```

- We want to convert this to `bool` so that it is a binary class

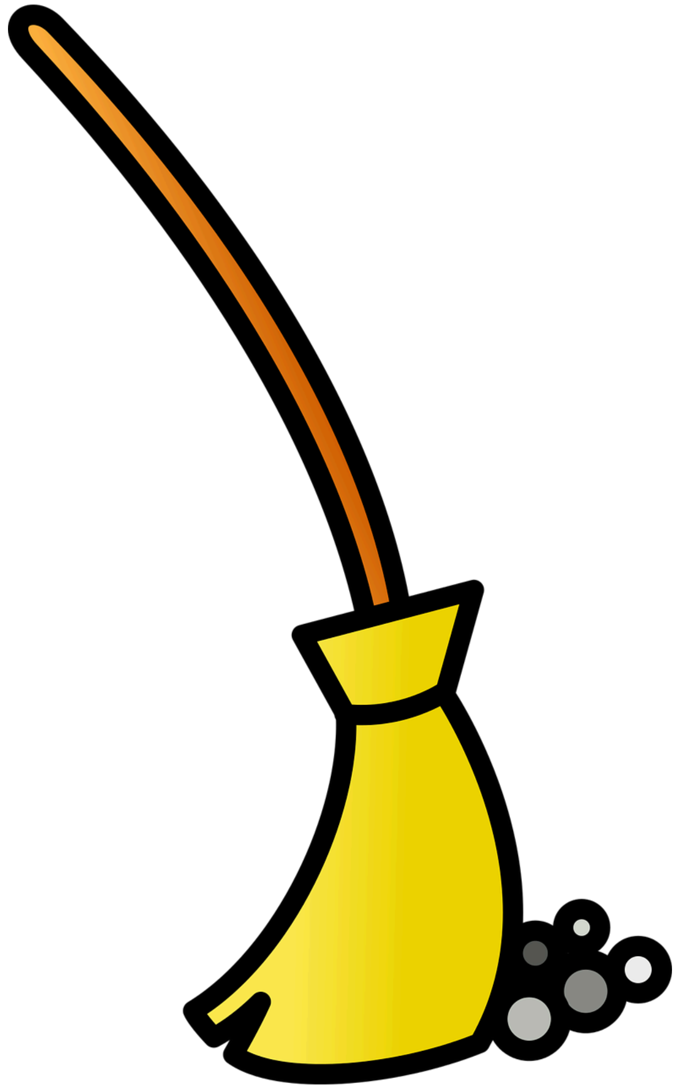
```
# Identify the the two unique classes
unique_values = sorted(df_subset['stroke'].unique())
df_subset['stroke'] = np.where(df_subset['stroke'] == unique_values[0],  False,True)
```

ID variables

- We will not use certain columns like ID variables in our dataset like **`id`** or variables with more than 50% NAs
- We want to see if the independent variables would predict `stroke` well

Data cleaning steps for logistic regression

- There are a few steps to remember to take before jumping into splitting the data and training the model
- Let's look at what it means to scale our predictors, and why it's necessary with logistic regression
- We will also talk through why we need to make sure the target variable is labeled



i. Make sure the target is labeled
ii. Check for NAs
iii. Encode categorical data into numerical data
iv. Split into train and test
v. Scale features

Data prep: target variable

- The first step of our data cleanup is to ensure that target variable is a binary class and has a label
- Let's look at the `dtype` of `stroke`

```
print(df_subset['stroke'].dtypes)
```

```
int64
```

- We want to convert this to `bool` so that is a binary class

```
# Identify the the two unique classes
unique_values = sorted(df_subset['stroke'].unique())
df_subset['stroke'] = np.where(df_subset['stroke'] == unique_values[0],  False,True)
# Check class again.
```

```
/opt/conda/envs/sdaia-python-classification/bin/python:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

```
print(df_subset['stroke'].dtypes)
```

Data prep: check for NAs

- We now check for NAs, and there are multiple methods to deal with them

```
# Check for NAs.
print(df_subset.isnull().sum())
```

```
age                    0
avg_glucose_level      0
heart_disease          0
ever_married           0
hypertension           0
Residence_type         0
gender                 0
smoking_status      1544
work_type              0
stroke                 0
id                     0
dtype: int64
```

- If we do have NA, we could replace them with a mean or 0

```
percent_missing = df_subset.isnull().sum() * 100 /
len(df_subset)
print(percent_missing)
```

```
age                   0.000000
avg_glucose_level     0.000000
heart_disease         0.000000
ever_married          0.000000
hypertension          0.000000
Residence_type        0.000000
gender                0.000000
smoking_status       30.215264
work_type             0.000000
stroke                0.000000
id                    0.000000
dtype: float64
```

Data prep: check for NAs (cont'd)

- We will delete the columns which contain either 50% or more than 50% missing data from the subset

```
# Delete columns containing either 50% or more than 50% NaN Values
perc = 50.0
min_count =  int(((100-perc)/100)*df_subset.shape[0] + 1)
df_subset = df_subset.dropna(axis=1,
               thresh=min_count)
print(df_subset.shape)
```

```
(5110, 11)
```

Data prep: check for NAs (cont'd)

- We will impute the numerical columns which have less than 50% missing data with mean and categorical columns with mode respectively

```
# Function to impute NA in both numeric and categorical columns
def fillna(df):
    # Fill numeric columns with mean value
    numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns
    df[numeric_cols] = df[numeric_cols].fillna(df[numeric_cols].mean())

    # Fill categorical columns with mode value
    categorical_cols = df.select_dtypes(include=['object']).columns
    df[categorical_cols] = df[categorical_cols].fillna(df[categorical_cols].mode().iloc[0])

    return df

df_subset = fillna(df_subset)
```

Data prep: split data

- We'll now split the data subset into a dataframe consisting of features and an target variable array

```
# Split the data into X and y
columns_to_drop_from_X = ['stroke'] + ['id']
X = df_subset.drop(columns_to_drop_from_X, axis = 1)
y = np.array(df_subset['stroke'])
```

- We are now ready to convert our data to numerical values

Dummy variables: one hot encoding

Dummy variables:

- are **artificial variables** used to represent variables with **two or more distinct levels or categories**
- represent categorical predictors as binary values, **0 or 1** and are often used for **regression analysis**

ID	Pet
1	Dog
2	Cat
3	Cat
4	Dog
5	Fish

ID	Dog	Cat	Fish
1	1	0	0
2	0	1	0
3	0	1	0
4	1	0	0
5	0	0	1

Dummy variables: reference category

- The number of dummy variables necessary to represent a single attribute variable is equal to the **number of levels (categories) in that variable minus one**
- One of the categories is omitted and used as a **base or reference category**
- The reference category, which is not coded, is the category to which **all other categories will be compared**
- The biggest group / category will often be the reference category

Dummy variables in Python

```
pd.get_dummies(dataframe['Column'],
               drop_first = ,
               ...)
```

- `data` is a `pandas` series or dataframe
- `drop_first` indicates whether to get `k-1` dummies out of `k` categorical levels

pandas.get_dummies

`pandas.get_dummies`(*data, prefix=None, prefix_sep='_', dummy_na=False, columns=None, sparse=False, drop_first=False, dtype=None*) [source]

Convert categorical variable into dummy/indicator variables

Parameters:

data : *array-like, Series, or DataFrame*

prefix : *string, list of strings, or dict of strings, default None*
String to append DataFrame column names. Pass a list with length equal to the number of columns when calling get_dummies on a DataFrame. Alternatively, *prefix* can be a dictionary mapping column names to prefixes.

prefix_sep : *string, default '_'*
If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with *prefix*.

dummy_na : *bool, default False*
Add a column to indicate NaNs, if False NaNs are ignored.

columns : *list-like, default None*
Column names in the DataFrame to be encoded. If *columns* is None then all the columns with *object* or *category* dtype will be converted.

sparse : *bool, default False*
Whether the dummy-encoded columns should be be backed by a `SparseArray` (True) or a regular NumPy array (False).

drop_first : *bool, default False*
Whether to get k-1 dummies out of k categorical levels by removing the first level.
New in version 0.18.0.

dtype : *dtype, default np.uint8*
Data type for new columns. Only a single dtype is allowed.
New in version 0.23.0.

Returns:

dummies : *DataFrame*

Data prep: convert categorical data columns to dummies

- In `logistic regression`, we use **numeric data** as predictors
- Let's double check:

```
print(X.dtypes)
```

```
age                  float64
avg_glucose_level    float64
heart_disease          int64
ever_married          object
hypertension           int64
Residence_type        object
gender                object
smoking_status        object
work_type             object
dtype: object
```

- Let's convert the categorical data to dummy variables

```
X = pd.get_dummies(X, columns = ['heart_disease',
'ever_married', 'hypertension', 'Residence_type',
'gender', 'smoking_status', 'work_type'],
dtype=float, drop_first=True)
print(X.dtypes)
```

```
age                               float64
avg_glucose_level                 float64
heart_disease_1                   float64
ever_married_Yes                  float64
hypertension_1                    float64
Residence_type_Urban              float64
gender_Male                       float64
gender_Other                      float64
smoking_status_never smoked       float64
smoking_status_smokes             float64
work_type_Never_worked            float64
work_type_Private                 float64
work_type_Self-employed           float64
work_type_children                float64
dtype: object
```

Split into train and test set

- We can now split our data into train and test sets
- We'll run logistic regression initially on the training data

```
# Set the seed.
np.random.seed(1)

# Split data into train and test sets, use a 70 train - 30 test split.
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size = .3)
```

Scale the features

- **Feature scaling** is an essential step of many machine learning algorithms
- Algorithms that use **gradient descent**, like logistic regression, are sensitive to the range of data points
- Performing feature scaling speeds up the gradient descent process used to calculate optimal coefficients
- ***Click here to learn more about gradient descent***

Scale the features (cont'd)

- `sklearn`'s implementation of ***`LogisticRegression`***, by default, implements **regularization** to prevent the model from overfitting
  - Regularization makes the model dependent on the scale of features
  - Features with larger magnitudes get **penalized** more in regularization than features with smaller magnitudes
  - Scaling features before fitting the model ensures that all features are penalized equally
- For more information about `sklearn.linear_model.LogisticRegression`, ***click here***

Scale the features (cont'd)

- We will use `sklearn`'s `MinMaxScaler` for feature scaling in this module

sklearn.preprocessing.MinMaxScaler¶

class sklearn.preprocessing. **MinMaxScaler** (*feature_range=(0, 1), copy=True*) [source]

Transforms features by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one.

The transformation is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where min, max = feature_range.

The transformation is calculated as:

```
X_scaled = scale * X + min - X.min(axis=0) * scale
where scale = (max - min) / (X.max(axis=0) - X.min(axis=0))
```

This transformation is often used as an alternative to zero mean, unit variance scaling.

```
# Initialize scaler.
scaler = preprocessing.MinMaxScaler()

# Fit on training data.
scaler.fit(X_train)

# Scale training and test data.
```

```
MinMaxScaler()
```

```
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Module completion checklist

Objectives	Complete
Transform categorical variables for implementation of logistic regression	✔
Implement logistic regression on the data and assess results of classification model performance	

scikit-learn: logistic regression

- We will be using the `LogisticRegression` library from `scikit-learn.linear_model` package

sklearn.linear_model.LogisticRegression

class `sklearn.linear_model.` **`LogisticRegression`** (*penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='liblinear', max_iter=100, multi_class='ovr', verbose=0, warm_start=False, n_jobs=1*) [source]

Logistic Regression (aka logit, MaxEnt) classifier.

- All inputs are optional arguments, but we will concentrate on two key inputs:
  - `penalty`: a regularization technique used to tune the model (either `l1`, a.k.a. *Lasso*, or `l2`, a.k.a. *Ridge*, default is `l2`)
  - `C`: a regularization constant used to amplify the effect of the regularization method (a value between $[0, \infty]$ default is `1`)

Logistic regression: solvers and their penalties

Solver	Behavior	Penalty
liblinear	Ideal for small datasets and one vs rest schemes	L1 and L2
lbfgs	Default solver, ideal for large datasets and multi-class problems	L2 or no penalty
newton-cg	Ideal for large datasets and multi-class problems	L2 or no penalty
sag	Works faster on large datasets and handles multi-class problems	L2 or no penalty
saga	Works faster on large datasets and handles multi-class problems	L1, L2, elastic net or no penalty

- Note: We'll be using `liblinear` and `lfbgs` solvers in this module

Logistic regression: build

- We're ready to build our logistic regression model
- We'll use all default parameters for now as our baseline model

```
# Set up logistic regression model.
logistic_regression_model = linear_model.LogisticRegression()
print(logistic_regression_model)
```

```
LogisticRegression()
```

- We can see that the default model contains `C = 1` and `penalty = 'l2'`
- We will discuss what that means later in more detail when we **tune** our model

Logistic regression: fit

The two main arguments are the same as with most classifiers in `scikit-learn`:

1. `X`: a `pandas` DataFrame or a `numpy` array of training data predictors
2. `y`: a `pandas` series or a `numpy` array of training labels

fit (*X, y, sample_weight=None*) [source]

Fit the model according to the given training data.

Parameters:	**X** : {array-like, sparse matrix}, shape (n_samples, n_features) Training vector, where n_samples is the number of samples and n_features is the number of features. **y** : array-like, shape (n_samples,) Target vector relative to X. **sample_weight** : array-like, shape (n_samples,) optional Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight. *New in version 0.17: sample_weight* support to LogisticRegression.
Returns:	**self** : object Returns self.

Logistic regression: fit (cont'd)

- We fit the logistic regression model with `X_train` and `y_train`
- We will run the model on our training data and predict on test data

```
# Fit the model.
logistic_regression_model.fit(X_train_scaled,
                              y_train)
```

```
LogisticRegression()
```

Logistic regression: predict

The main argument is the same as with most classifiers in `scikit-learn`:

1. `X`: a `pandas` dataframe or a `numpy` array of test data predictors

- We will predict on the test data using our trained model
- The result is a **vector** of the predictions

```
# Predict on test data.
predicted_values =
logistic_regression_model.predict(X_test_scaled)
print(predicted_values[:20])
```

```
[False False False False False False False False False
False False False
 False False False False False False False False]
```

`predict(X)` [source]

Predict class labels for samples in X.

Parameters:	**X** : {array-like, sparse matrix}, shape = [n_samples, n_features] Samples.
Returns:	**C** : array, shape = [n_samples] Predicted class label per sample.

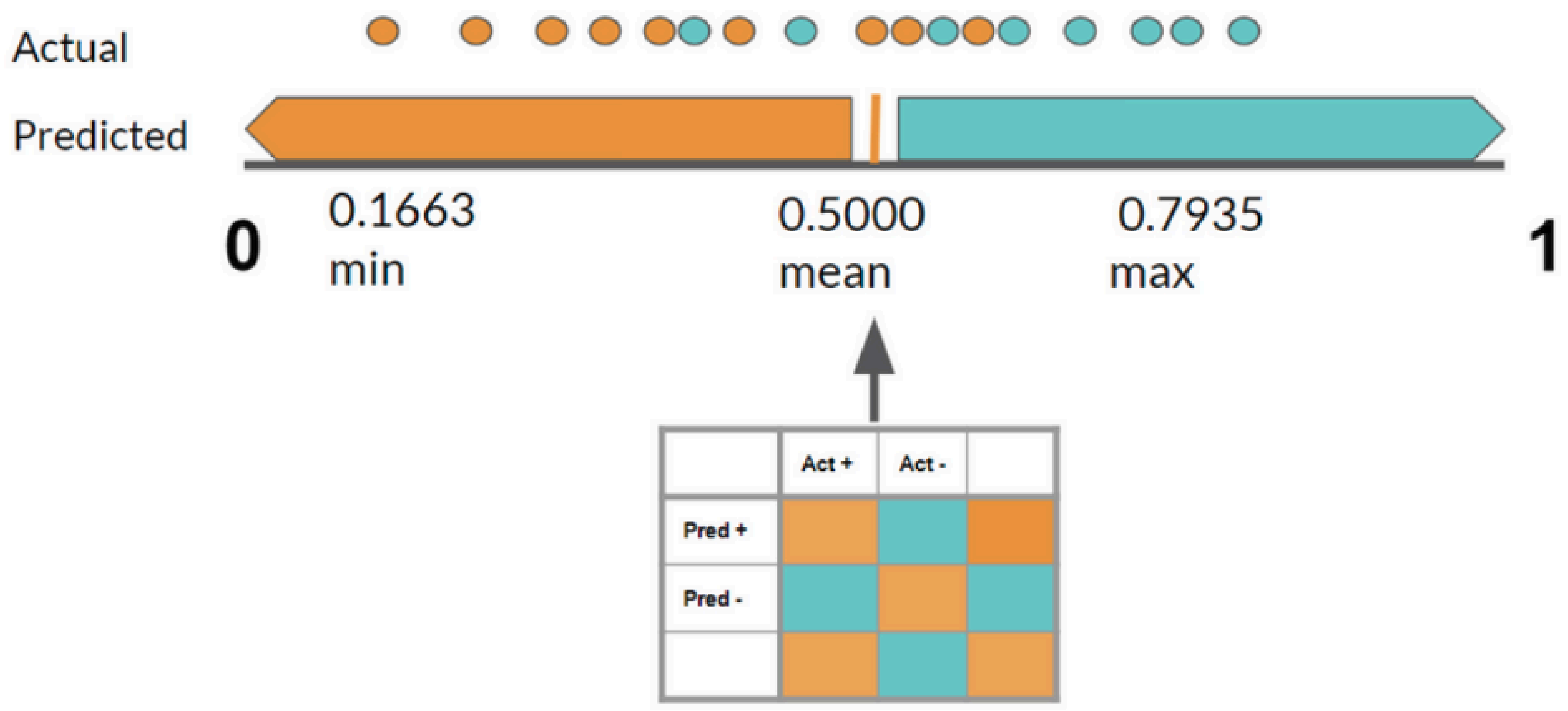
Confusion matrix: overview

	Predicted Low value	Predicted High value	Actual totals
Actual low value	**True negative (TN)**	**False positive (FP)**	Total negatives
Actual high value	**False negative (FN)**	**True positive (TP)**	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	**Total**

- **True positive rate (TPR)** (a.k.a. *Sensitivity*, *Recall*) = **TP** / Total positives
- **True negative rate (TNR)** (a.k.a. *Specificity*) = **TN** / Total negatives
- **False positive rate (FPR)** (a.k.a. *Fall-out*, *Type I Error*) = **FP** / Total negatives
- **False negative rate (FNR)** (a.k.a. *Type II Error*) = **FN** / Total positives
- **Accuracy** = **TP** + **TN** / **Total**
- **Misclassification rate** = **FP** + **FN** / **Total**

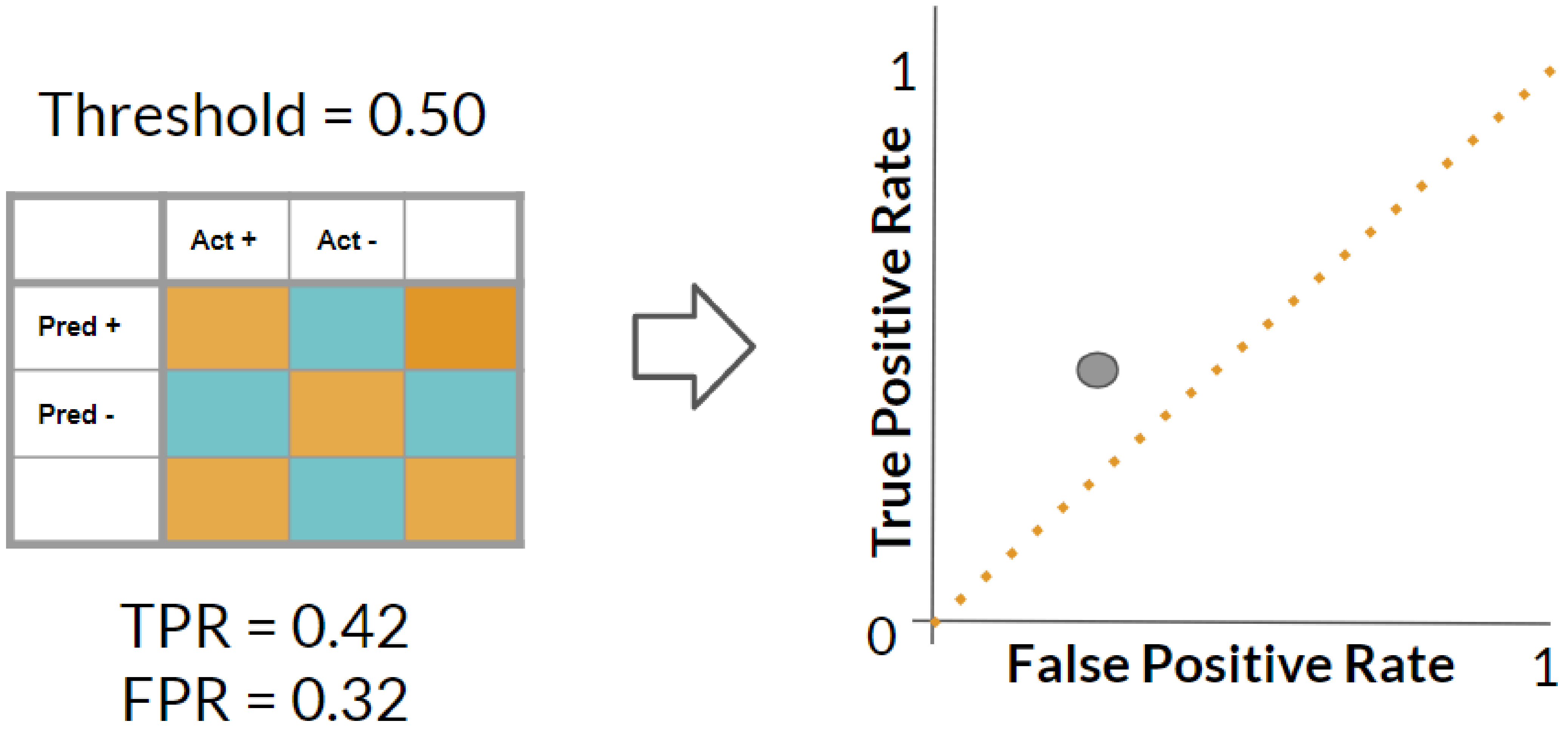
From threshold to metrics

- In logistic regression, the output is a range of probabilities from `0` to `1`
- But how do you interpret that as a `1` / `0` or `High value` / `Low value` label?
- You set a **threshold** where everything above is predicted as `1` and everything below is predicted as `0`
- A typical threshold for logistic regression is `0.5`



From metrics to a point

- Each threshold can create a confusion matrix, which can be used to calculate a point in space defined by:
  - **True positive rate (TPR)** on the `y-axis`
  - **False positive rate (FPR)** on the `x-axis`

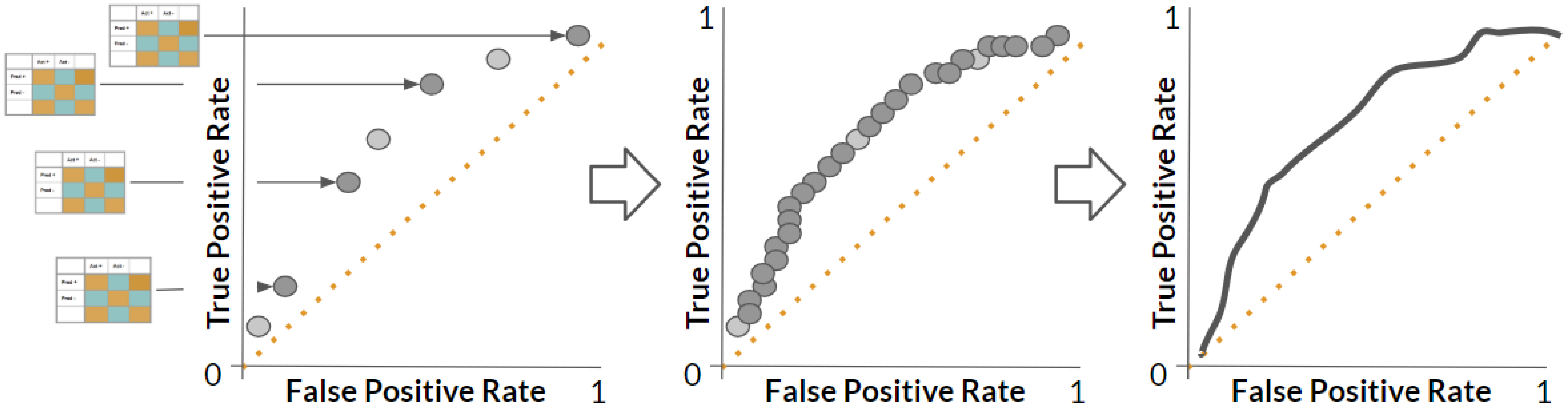


From points to a curve

- The **ROC curve** is a performance metric used to compare classification models to measure predictive accuracy
- The **AUC** **(Area under the ROC Curve)** should be above 0.5 to say the model is better than a random guess
- We can obtain the AUC by providing the FPR and TPR using the function `metrics.auc(fpr, tpr)`

From points to a curve (cont'd)

- When we move thresholds, we re-calculate our metrics and create confusion matrices for **every threshold**
- Each time, we plot a new point in the **TPR** vs. **FPR** space



scikit-learn: metrics package

`sklearn.metrics` : Metrics

See the Model evaluation: quantifying the quality of predictions section and the Pairwise metrics, Affinities and Kernels section of the user guide for further details.

The `sklearn.metrics` module includes score functions, performance metrics and pairwise metrics and distance computations.

- We will use the following methods from this library:
  - `confusion_matrix`
  - `accuracy_score`
  - `classification_report`
  - `roc_curve`
  - `auc`
- For all the methods and parameters of the `metrics` package, visit scikit-learn's documentation ***by clicking here***

Confusion matrix and accuracy

Both `confusion_matrix` and `accuracy_score` take **two** arguments:

1. Original data labels
2. Predicted labels

```
# Take a look at test data confusion matrix.
conf_matrix_test = metrics.confusion_matrix(y_test, predicted_values)
print(conf_matrix_test)
```

```
[[1450    0]
 [  83    0]]
```

```
# Compute test model accuracy score.
test_accuracy_score = metrics.accuracy_score(y_test, predicted_values)
print("Accuracy on test data: ", test_accuracy_score)
```

```
Accuracy on test data:  0.9458577951728636
```

Classification report

- We can more easily interpret the output `classification_report` by adding the target variable's **class names** to the two arguments that `confusion_matrix` takes

```
# Create a list of target names to interpret class assignments.
target_names = df_subset['stroke'].unique()
target_names=target_names.tolist()
target_names = [str(x) for x in target_names]
```

```
# Print an entire classification report.
class_report = metrics.classification_report(y_test,
                                             predicted_values,
                                             target_names = target_names)
```

```
/opt/conda/envs/sdaia-python-classification/lib/python3.7/site-
packages/sklearn/metrics/_classification.py:1245: UndefinedMetricWarning: Precision and F-
score are ill-defined and being set to 0.0 in labels with no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/envs/sdaia-python-classification/lib/python3.7/site-
packages/sklearn/metrics/_classification.py:1245: UndefinedMetricWarning: Precision and F-
score are ill-defined and being set to 0.0 in labels with no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/envs/sdaia-python-classification/lib/python3.7/site-
packages/sklearn/metrics/_classification.py:1245: UndefinedMetricWarning: Precision and F-
score are ill-defined and being set to 0.0 in labels with no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Classification report (cont'd)

```
print(class_report)
```

```
              precision    recall  f1-score   support

        True       0.95      1.00      0.97      1450
       False       0.00      0.00      0.00        83

    accuracy                           0.95      1533
   macro avg       0.47      0.50      0.49      1533
weighted avg       0.89      0.95      0.92      1533
```

Precision

	Positive	Negative
Positive	TP	FP
Negative	FN	TN

- $PR = \frac{(TP)}{(TP+FP)}$
- A proportion of values that is truly positive out of all predicted positive values
- A.K.A. positive predicted value (PPV)

Recall

	Positive	Negative
Positive	TP	FP
Negative	FN	TN

- $RE = \frac{(TP)}{(TP+FN)}$
- Proportion of actual positives that is classified correctly
- A.K.A. sensitivity, hit rate, or true positive rate (TPR)

F1: precision vs. recall

- A score that gives us a numeric value of the precision vs recall tradeoff
- `f1-score` is calculated as a weighted harmonic mean of `precision` and `recall`
- $F1 = 2 \times \frac{(PR*RE)}{(PR+RE)}$
- The higher the $F1$ score, the better (the score can be a value between 0 and 1)
- **Support** is the actual number of occurrences of each class in `y_test`

Save accuracy score

- So we have it, let's add this score to `model_final` in case we need to use it again to compare against other models

```
model_final = {'metrics' : "accuracy" ,
                'values' : round(test_accuracy_score,4),
                'model':'logistic' }
print(model_final)
```

```
{'metrics': 'accuracy', 'values': 0.9459, 'model': 'logistic'}
```

Getting probabilities instead of class labels

- Now we can start gathering the various components to build our ROC curve and calculate the AUC
- Again, we are looking to ensure that our model has better predictive ability than making a random guess

```
# Get probabilities instead of predicted values.
test_probabilities = logistic_regression_model.predict_proba(X_test_scaled)
print(test_probabilities[0:5, :])
```

```
[[0.98291824 0.01708176]
 [0.82575294 0.17424706]
 [0.98913898 0.01086102]
 [0.83706934 0.16293066]
 [0.96511303 0.03488697]]
```

```
# Get probabilities of test predictions only.
test_predictions = test_probabilities[:, 1]
print(test_predictions[0:5])
```

```
[0.01708176 0.17424706 0.01086102 0.16293066 0.03488697]
```

Computing FPR, TPR, and threshold

```
# Get FPR, TPR, and threshold values.
fpr, tpr, threshold = metrics.roc_curve(y_test,             #<- test data labels
                                        test_predictions)   #<- predicted probabilities
print("False positive: ", fpr[:5])
```

```
False positive:  [0.         0.00068966 0.00068966 0.00137931 0.00137931]
```

```
print("True positive: ", tpr[:5])
```

```
True positive:  [0.         0.         0.01204819 0.01204819 0.02409639]
```

```
print("Threshold: ", threshold[:5])
```

```
Threshold:  [1.41962279 0.41962279 0.40005488 0.38013817 0.37207471]
```

Computing AUC

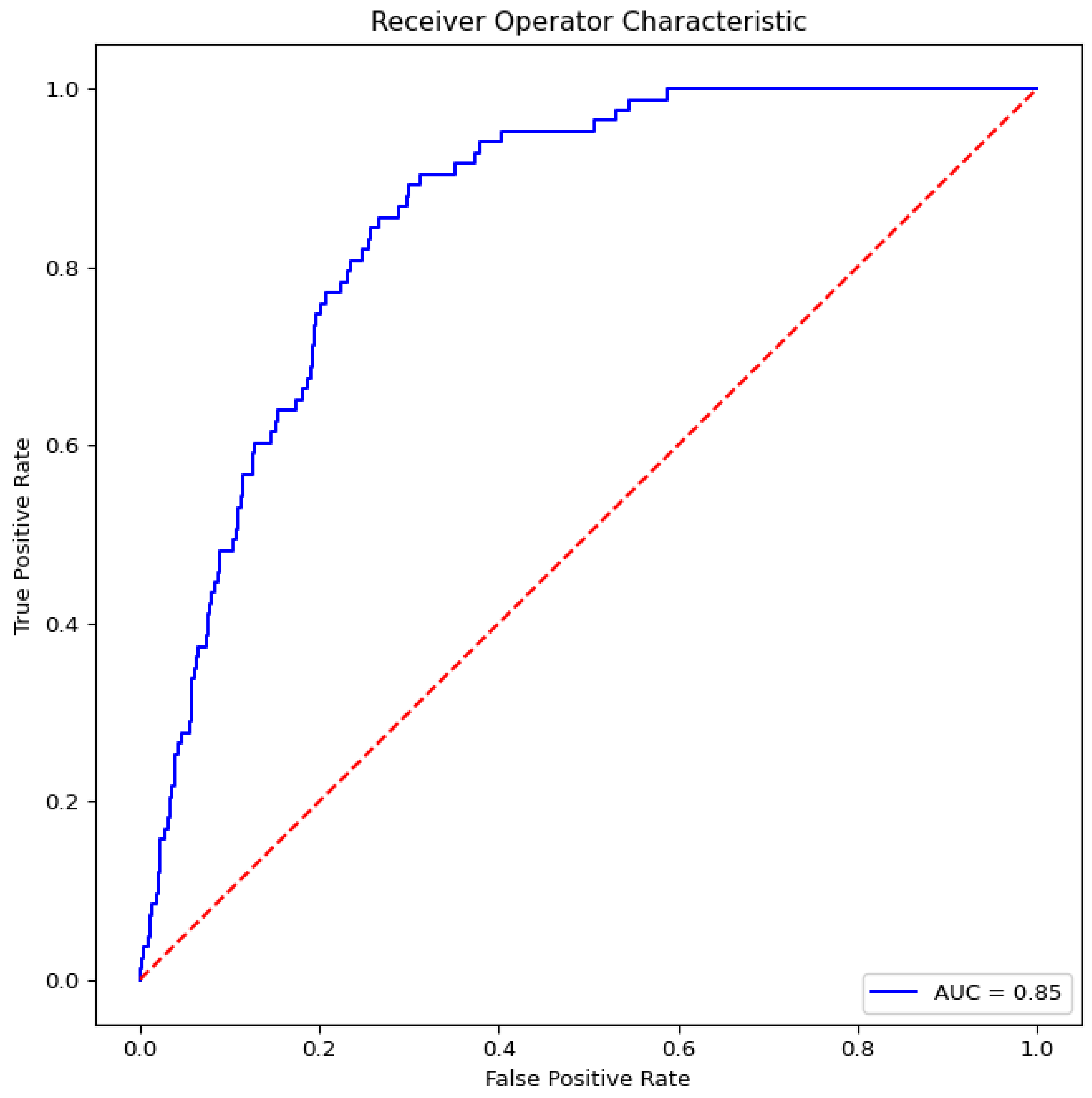
```
# Get AUC by providing the FPR and TPR.
auc = metrics.auc(fpr, tpr)
print("Area under the ROC curve: ", auc)
```

```
Area under the ROC curve:  0.8549646863315331
```

Putting it all together: ROC plot

```
# Make an ROC curve plot.
plt.title('Receiver Operator Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

- Our model achieved the accuracy of about `0.95`
- Our estimated AUC is about `0.85`
- **How would you rate this model as a baseline?**



Knowledge check

Module completion checklist

Objectives	Complete
Transform categorical variables for implementation of logistic regression	✔
Implement logistic regression on the data and assess results of classification model performance	✔

Congratulations on completing this module!

You are now ready to try Tasks 1-8 in the Exercise for this topic