Machine learning - Part 4

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

Welcome back!

In the last class we learned about

- applying cross-validation to discover optimal model accuracy
- using hyperparameters and GridSearch
- logistic regression and its applications

Today we will cover

- logistic regression on a training dataset and predict on test
- classification performance metrics
- transformation of categorical variables for implementation of logistic regression
- implementation of logistic regression on the data

Warm-up activity

- **Check out this *article* to see an example of a logistic regression project.**

Module completion checklist

Objective	Complete
Implement logistic regression on a training dataset and predict on test	
Review classification performance metrics and assess results of logistic model performance	
Transform categorical variables for implementation of logistic regression	
Implement logistic regression on the data and assess results of classification model performance	

Loading packages

- Load the packages we will be using

```
# Helper packages.
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pickle

# 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

# 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 `skillsoft-machine-learning-2021` folder
- `data_dir` be the variable corresponding to your `data` folder

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

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

Recap: logistic regression coefficients

- In **linear** regression, the coefficients in the equation can easily be interpreted

$$ax + b$$

- An increase in x will result in an increase in y and vice versa

BUT

- In **logistic** regression, the simplest way to interpret a positive coefficient is with an increase in likelihood
- A larger value of x increases the likelihood that $y = 1$

Datasets for logistic regression

- **We will be using two datasets total, we discussed each of the datasets and use cases already**

- **One dataset to learn the concepts in class**
  - Costa Rica household poverty data

- **One dataset for our in-class exercises**
  - Chicago census data

Costa Rican poverty recap

Costa Rican poverty level prediction: proposed solution

- To improve on PMT, the IDB built a competition for Kaggle participants to use methods beyond traditional econometrics

- The given dataset contains Costa Rican household characteristics with a target of four categories:
    - extreme poverty
    - moderate poverty
    - vulnerable households
    - non-vulnerable households

Load the dataset

- Let's load the entire dataset

```
household_poverty = pd.read_csv(data_dir + '/costa_rica_poverty.csv')
print(household_poverty.head())
```

```
  household_id        ind_id  rooms  ...  age  Target  monthly_rent
0    21eb7fcc1  ID_279628684      3  ...   43       4      190000.0
1    0e5d7a658  ID_f29eb3ddd      4  ...   67       4      135000.0
2    2c7317ea8  ID_68de51c94      8  ...   92       4           NaN
3    2b58d945f  ID_d671db89c      5  ...   17       4      180000.0
4    2b58d945f  ID_d56d6f5f5      5  ...   37       4      180000.0

[5 rows x 84 columns]
```

- The entire dataset consists of `9557` observations and `84` variables

Subsetting data

- In this module, we will run the model on a simple subset
- We don’t want to use `monthly_rent` as a variable right now because it has many NAs
- For our report, we want to see if **number of rooms** and **number of adults** would predict poverty level well
- Then we are going to predict the same target with whole dataset

Subsetting data

- Let's subset our data so that we have the variables we need for building our model
- We will drop the variables containing ID as they do not provide any significance for the model, along with `monthly_rent`
- Let's name this subset `household_logistic`

```
household_logistic = household_poverty.drop(['household_id', 'ind_id', 'monthly_rent'], axis = 1)
```

The data at first glance

- Look at the data types and the frequency table of the target variable

```
# The data types.
print(household_logistic.dtypes.head())
```

```
rooms               int64
tablet              int64
males_under_12      int64
males_over_12       int64
males_tot           int64
dtype: object
```

```
print(household_logistic['Target'].value_counts())
```

```
4    5996
2    1597
3    1209
1     755
Name: Target, dtype: int64
```

- The target variable is not well-balanced and has **four levels**

Converting the target variable

- Let's convert poverty to a binary target variable, which will help to balance it out
- The levels translate to 1, 2 and 3 as being **vulnerable** households
- Level 4 is **non-vulnerable**
- For this reason, we will convert all 1, 2 and 3 to `vulnerable` and 4 to `non_vulnerable`

```
household_logistic['Target'] = np.where(household_logistic['Target'] <= 3, 'vulnerable',
'non_vulnerable')
```

```
print(household_logistic['Target'].head())
```

```
0    non_vulnerable
1    non_vulnerable
2    non_vulnerable
3    non_vulnerable
4    non_vulnerable
Name: Target, dtype: object
```

Data prep: check for NAs

- Check for NAs

```
# Check for NAs.
print(household_logistic.isnull().sum().head())
```

```
rooms             0
tablet            0
males_under_12    0
males_over_12     0
males_tot         0
dtype: int64
```

- We do not have any NAs!

Data prep: numeric variables

- We try and use **numeric data** as predictors
- In some cases, we can **convert categorical data to integer values**
- However, in this simple example, our predictors are numeric by default
- Let's double check:

```
print(household_logistic.dtypes.head())
```

```
rooms              int64
tablet             int64
males_under_12     int64
males_over_12      int64
males_tot          int64
dtype: object
```

Data prep: target

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

```
print(household_logistic.Target.dtypes)
```

```
object
```

- We want to convert this to `bool` (Boolean type) so that it's a binary class

```
household_logistic["Target"] = np.where(household_logistic["Target"] == "non_vulnerable", True,
False)

# Check class again.
print(household_logistic.Target.dtypes)
```

```
bool
```

Split into train and test set

- For now, we are only going to use `rooms` and `num_adults` for a simple logistic regression model
- As we did previously, we split our data into training and test sets
- We run logistic regression initially on the training data

```
# Separate predictors from data.
X = household_logistic[['rooms', 'num_adults']]
```

```
# Separate target from data.
y = np.array(household_logistic['Target'])
```

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

# Split data into training 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)
```

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`)
- For all the parameters of the `LogisticRegression` function, visit ***scikit-learn's documentation***

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 data sets and multi-class problems	L2 or no penalty
newton-cg	Ideal for large data sets and multi-class problems	L2 or no penalty
sag	Works faster on large data sets and handles multi-class problems	L2 or no penalty
saga	Works faster on large data sets and handles multi-class problems	L1, L2, elastic net or no penalty

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

- To know more about solvers in Logistic regression, visit ***scikit-learn's documentation***

Logistic regression: build

- Let’s 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()
```

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

- 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,
                              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

`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.

Logistic regression: predict

- 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)
print(predicted_values)
```

```
[ True  True  True ...  True False  True]
```

Knowledge check 1

Exercise 1

Module completion checklist

Objective	Complete
Implement logistic regression on a training dataset and predict on test	✔
Review classification performance metrics and assess results of logistic model performance	
Transform categorical variables for implementation of logistic regression	
Implement logistic regression on the data and assess results of classification model performance	

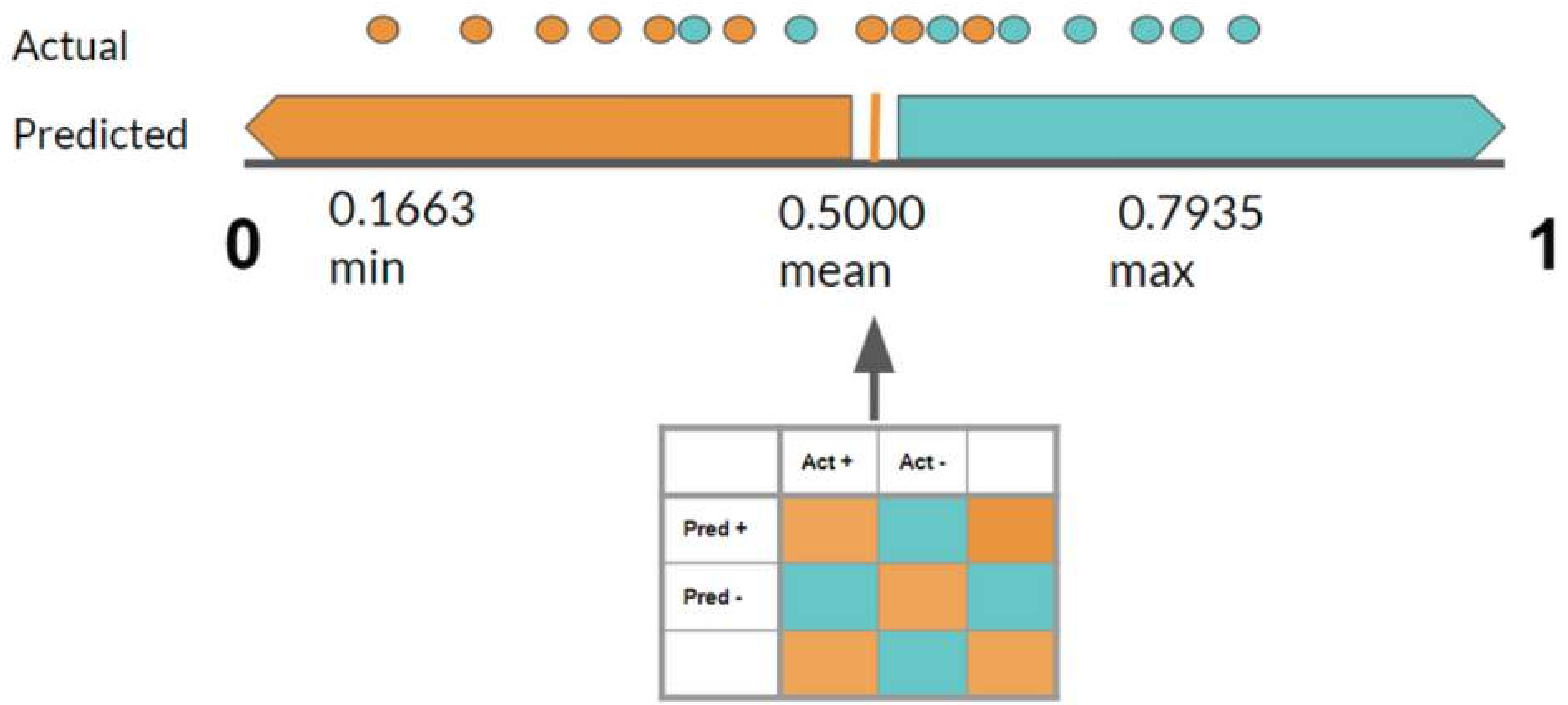
Recap: confusion matrix

	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

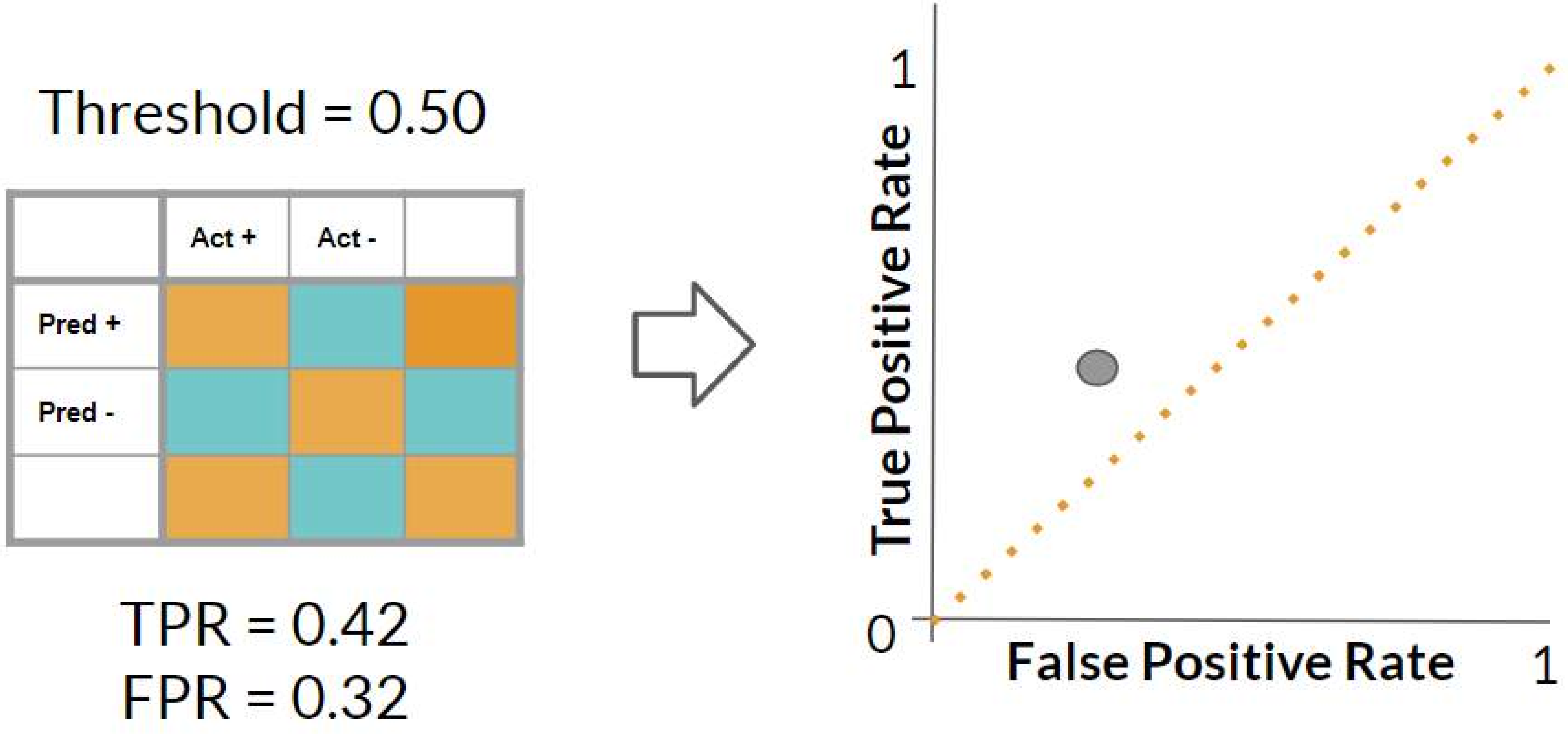
- In logistic regression, the output is a range of probabilities from `0` to `1`
- But how do you interpret that as a `1` / `0` or `High value` / `Low value` label?
- You set a **threshold** where everything above is predicted as `1` and everything below is predicted as `0`
- A typical threshold for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



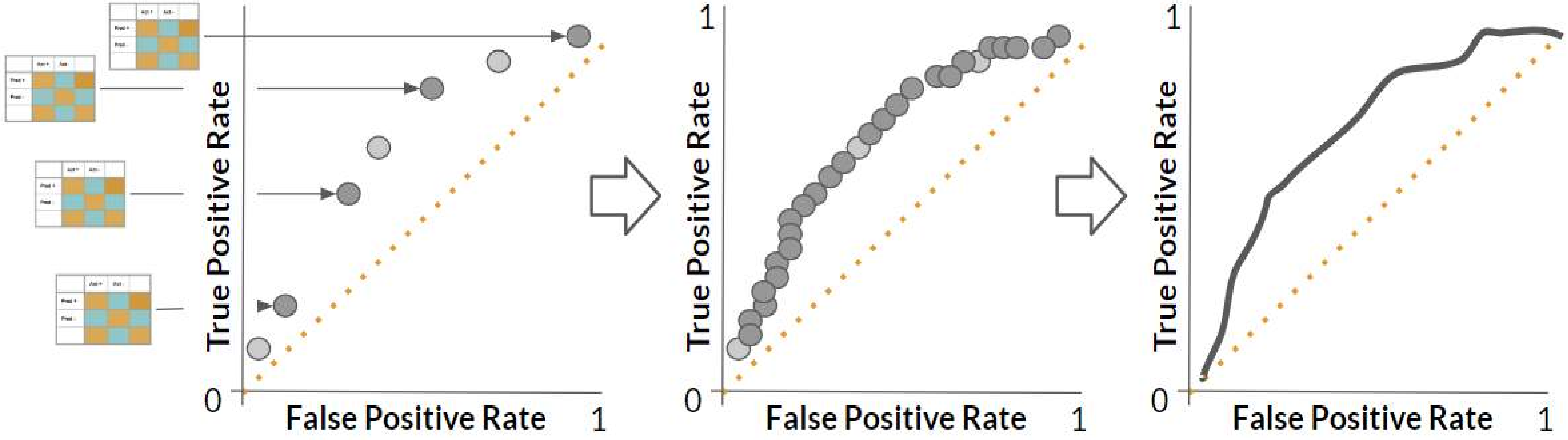
From metrics to a point

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



- 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

More about AUC and ROC curve

- ROC curve obtained by plotting the TPR vs FPR for various thresholds
- It is used to compare classification models to measure predictive accuracy
- The AUC should be above .5 to say the model is better than a random guess
- The function to obtain AUC by providing the FPR and TPR is

```
metrics.auc (fpr, tpr)
```

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***

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)
```

```
[[ 178  884]
 [ 161 1645]]
```

```
# 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:  0.6356345885634589
```

Classification report

- To make interpretation of the `classification_report` easier, in addition to the two arguments that `confusion_matrix` takes, we can add the actual class names for our target variable

```
# Create a list of target names to interpret class assignments.
target_names = ['vulnerable', 'non_vulnerable']
```

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

```
                precision    recall  f1-score   support

    vulnerable       0.53      0.17      0.25      1062
non_vulnerable       0.65      0.91      0.76      1806

      accuracy                           0.64      2868
     macro avg       0.59      0.54      0.51      2868
  weighted avg       0.60      0.64      0.57      2868
```

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. PPV - positive predicted value

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`

Add accuracy score to the final scores

- So we have it, let's add this score to the dataframe `model_final` that we created in the previous class
- Let's load the pickled dataset and append the score to it

```
model_final = pickle.load(open((data_dir + "/model_final.sav"),"rb"))
```

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

```
     metrics  values              model
0   accuracy  0.6046              knn_5
1   accuracy  0.6268  knn_GridSearchCV
2   accuracy  0.6287             knn_29
3   accuracy  0.6268  knn_GridSearchCV
4   accuracy  0.6287             knn_29
5   accuracy  0.6268  knn_GridSearchCV
6   accuracy  0.6287             knn_29
7   accuracy  0.6268  knn_GridSearchCV
8   accuracy  0.6287             knn_29
9   accuracy  0.6268  knn_GridSearchCV
10  accuracy  0.6287             knn_29
11  accuracy  0.6268  knn_GridSearchCV
12  accuracy  0.6287             knn_29
13  accuracy  0.6268  knn_GridSearchCV
```

Getting probabilities instead of class labels

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

```
[[0.28454513 0.71545487]
 [0.37631548 0.62368452]
 [0.16166044 0.83833956]
 [0.5294136  0.4705864 ]
 [0.3519399  0.6480601 ]]
```

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

```
[0.71545487 0.62368452 0.83833956 0.4705864  0.6480601 ]
```

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.         0.         0.         0.0047081]
```

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

```
True positive:  [0.         0.00387597 0.00609081 0.01052049 0.01162791]
```

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

```
Threshold:  [1.93016754 0.93016754 0.91552704 0.90664549 0.89774864]
```

Computing AUC

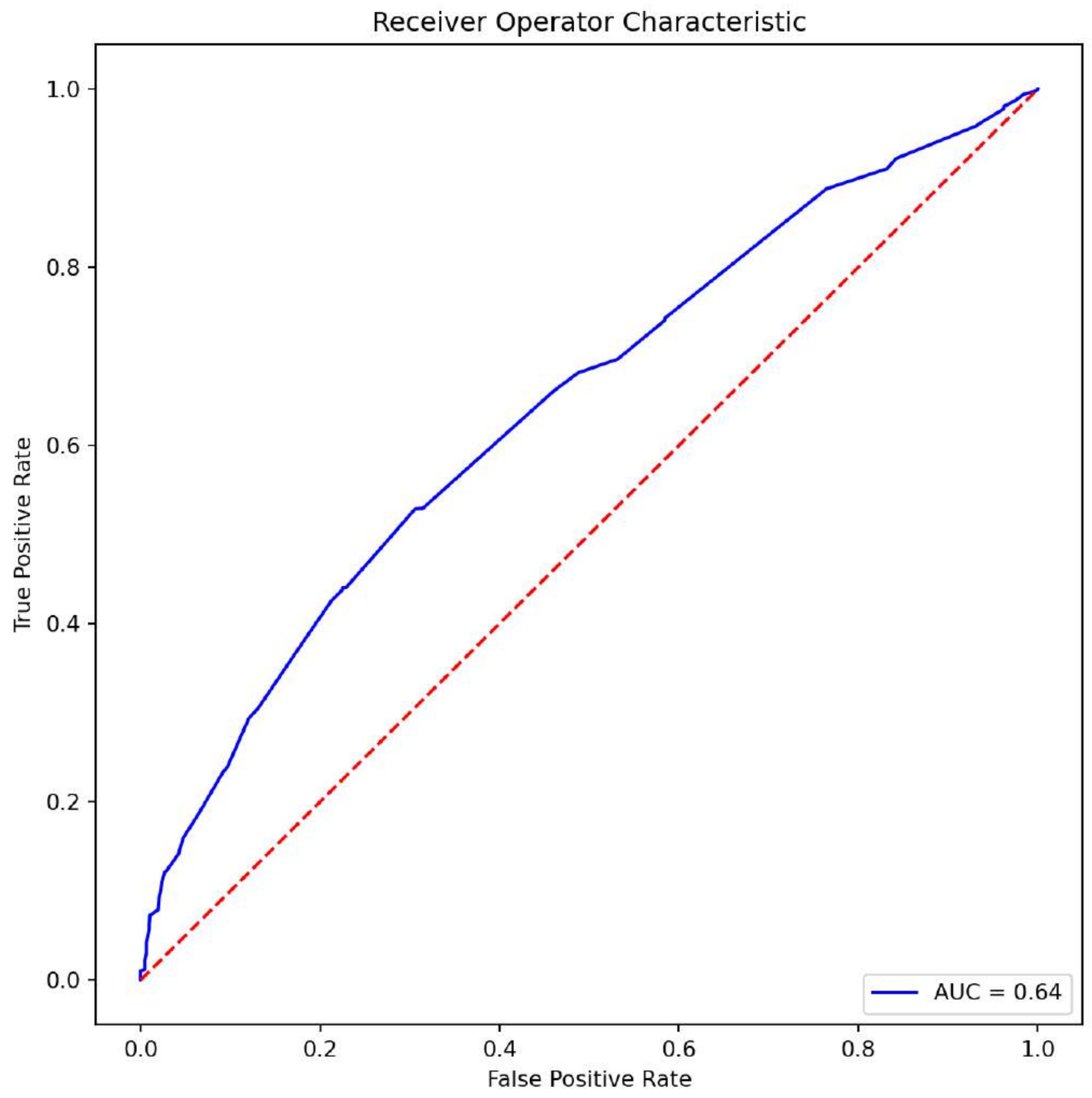
```
# Get AUC by providing the FPR and TPR.
auc = metrics.auc(fpr, tpr)
print("Area under the ROC curve: ", auc)
```

```
Area under the ROC curve:  0.6440758780628705
```

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 `0.635`, which is decent for a base model.
- Our estimated AUC is about `0.644`
- Given that we have not done any model tuning or data transformations, this is a fair baseline that we'll use to assess future models that we'll create



Knowledge check 2

Exercise 2

Module completion checklist

Objective	Complete
Implement logistic regression on a training dataset and predict on test	✔
Review classification performance metrics and assess results of logistic model performance	✔
Transform categorical variables for implementation of logistic regression	
Implement logistic regression on the data and assess results of classification model performance	

Working with categorical variables

- Let's take a look at numerical variable `age` from our dataset

```
print(household_logistic.age.head())
```

```
0     43
1     67
2     92
3     17
4     37
Name: age, dtype: int64
```

- We are going to convert `age` to a **categorical variable with 3 levels** to analyze varying poverty level between age groups

```
household_logistic['age'] = np.where(household_logistic['age'] <= 30, "30 or Below",
             np.where(household_logistic['age'] < 60, 'Between 30 and 60', '60 and above'))
```

Working with categorical variables

- Let's see the frequency of each level in `age`

```
household_logistic.age.value_counts()
```

```
30 or Below           4655
Between 30 and 60     3495
60 and above          1407
Name: age, dtype: int64
```

- As regression analysis is used with **numeric or continuous variables** to determine an outcome, how would we handle **categorical variables**?

Dummy variables: one hot encoding

- It is an **artificial variable** used to represent a variable with **two or more distinct levels or categories**
- It represents categorical predictors as binary values, **0 or 1**
- 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*

Transform `age` into dummies

- We need to transform `age`, which is categorical with 3 levels, into a dummy variable and save it into a dataframe

```
# Convert 'age' into dummy variables.
age_dummy = pd.get_dummies(household_logistic['age'], drop_first = True)
print(age_dummy.head())
```

```
   60 and above  Between 30 and 60
0             0                  1
1             1                  0
2             1                  0
3             0                  0
4             0                  1
```

- Notice that level `30 or below`, which has the highest count, has been removed and used as a reference category

Transform `age` into dummies

- Let's drop the original `age` column from our Costa Rica subset and concatenate the dummy variables `age_dummy`

```
# Drop `age` from the data.
household_logistic.drop(['age'], axis = 1, inplace = True)
```

```
# Concatenate `age_dummy` to our dataset.
household_logistic = pd.concat([household_logistic,age_dummy],axis=1)
print(household_logistic.head())
```

```
   rooms  tablet  males_under_12  ...  Target  60 and above  Between 30 and 60
0      3       0               0  ...    True             0                  1
1      4       1               0  ...    True             1                  0
2      8       0               0  ...    True             1                  0
3      5       1               0  ...    True             0                  0
4      5       1               0  ...    True             0                  1

[5 rows x 82 columns]
```

Module completion checklist

Objective	Complete
Implement logistic regression on a training dataset and predict on test	✔
Review classification performance metrics and assess results of logistic model performance	✔
Transform categorical variables for implementation of logistic regression	✔
Implement logistic regression on the data and assess results of classification model performance	

Split into train and test set

- Let's use the whole dataset this time
- We run logistic regression initially on the training data

```
# Separate predictors from data.
# We can just drop the target variable, as we are using all other variables as predictors.
X = household_logistic.drop('Target', axis = 1)
```

```
# Separate target from data.
y = np.array(household_logistic['Target'])
```

```
# Set the seed.
np.random.seed(1)
# Split data into training 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)
```

Logistic regression: build

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.

- Let's build our logistic regression model on our whole data set and use solver as liblinear.

```
# Set up the logistic regression model.
logistic_regression_model = linear_model.LogisticRegression(solver='liblinear')
print(logistic_regression_model)
```

```
LogisticRegression(solver='liblinear')
```

- The default logistic regression model contains `C = 1` and `penalty = 'l2'` as its parameters
- To know more about parameters in Logistic regression, visit ***scikit-learn's documentation***

Logistic regression: fit

- 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,
                              y_train)
```

```
LogisticRegression(solver='liblinear')
```

Logistic regression: predict

- 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)
print(predicted_values)
```

```
[ True False  True ...  True False False]
```

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)
```

```
[[ 687  375]
 [ 243 1563]]
```

```
# 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:  0.7845188284518828
```

Add accuracy score to the final scores

- So we have it, let's add this score to the dataframe `model_final` that we created earlier
- Let's load the pickled dataset and append the score to it

```
model_final = model_final.append({'metrics' : "accuracy" ,
                                  'values' : round(test_accuracy_score,4),
                                  'model':'logistic_whole_dataset'} ,
                                  ignore_index = True)
print(model_final)
```

```
     metrics  values                model
0    accuracy  0.6046               knn_5
1    accuracy  0.6268    knn_GridSearchCV
2    accuracy  0.6287              knn_29
3    accuracy  0.6268    knn_GridSearchCV
4    accuracy  0.6287              knn_29
5    accuracy  0.6268    knn_GridSearchCV
6    accuracy  0.6287              knn_29
7    accuracy  0.6268    knn_GridSearchCV
8    accuracy  0.6287              knn_29
9    accuracy  0.6268    knn_GridSearchCV
10   accuracy  0.6287              knn_29
11   accuracy  0.6268    knn_GridSearchCV
12   accuracy  0.6287              knn_29
13   accuracy  0.6268    knn_GridSearchCV
14   accuracy  0.6287              knn_29
15   accuracy  0.6268    knn_GridSearchCV
16   accuracy  0.6287              knn_29
```

Accuracy on train vs accuracy on test

- Take a look at the accuracy score for the training data

```
# Compute trained model accuracy score.
trained_accuracy_score = logistic_regression_model.score(X_train, y_train)
print("Accuracy on train data: " , trained_accuracy_score)
```

```
Accuracy on train data:  0.7808342054118702
```

- Did our model underperform?
- Is there a big difference in `train` and `test` accuracy?

Knowledge check 3

Exercise 3

Module completion checklist

Objective	Complete
Implement logistic regression on a training dataset and predict on test	✔
Review classification performance metrics and assess results of logistic model performance	✔
Transform categorical variables for implementation of logistic regression	✔
Implement logistic regression on the data and assess results of classification model performance	✔

Summary

- In this class we covered following topics:
  - logistic regression on a training dataset and predict on test
  - classification performance metrics
  - transformation of categorical variables for implementation of logistic regression
  - implementation of logistic regression on the data
- There are many more topics within machine learning that you might be interested in. Some of them are:
  - Decision trees and random forests
  - Support vector machines
  - Neural networks and deep learning

Congratulations on completing this module!