Machine learning - Part 2

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

Welcome back!

In the last class we learned about

- the characteristics of supervised and unsupervised machine learning
- clustering and its applications
- using k-means for clustering

Today we will cover

- classification and its use cases
- summary and applications of knn algorithm
- implementation of the knn algorithm on training data
- cross-validation and its use cases

Classification in real life

Before we look into classification, look at this example of how retail industry uses it:

- In 2002, Target implemented data analytics to analyze buying patterns in customers.
- New parents often get bombarded with advertising offers, so Target wanted a way to anticipate who is expecting in order to get ahead of the competition.
- They were able to predict pregnancy of their customers based upon their purchases and sent out targeted coupons.


How Companies Learn Your Secrets

By CHARLES DUHIGG FEB. 16, 2012

Antonio Bolfo/Reportage for The New York Times

http://www.nytimes.com/2012/02/19/magazine/shopping-habits.html?_r=0

Module completion checklist

Objective	Complete
Understanding classification and its uses	
Summarize steps & application of kNN	
Clean and transform data to run kNN	
Define cross-validation and how and when it is used	
Implement kNN algorithm on the training data without cross-validation	
Identify performance metrics for classification algorithms and evaluate simple kNN model	

Loading packages

Let’s load the packages we will be using:

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# New today - we will introduce it when we use it
import pickle
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import scale
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn import metrics
```

Directory settings

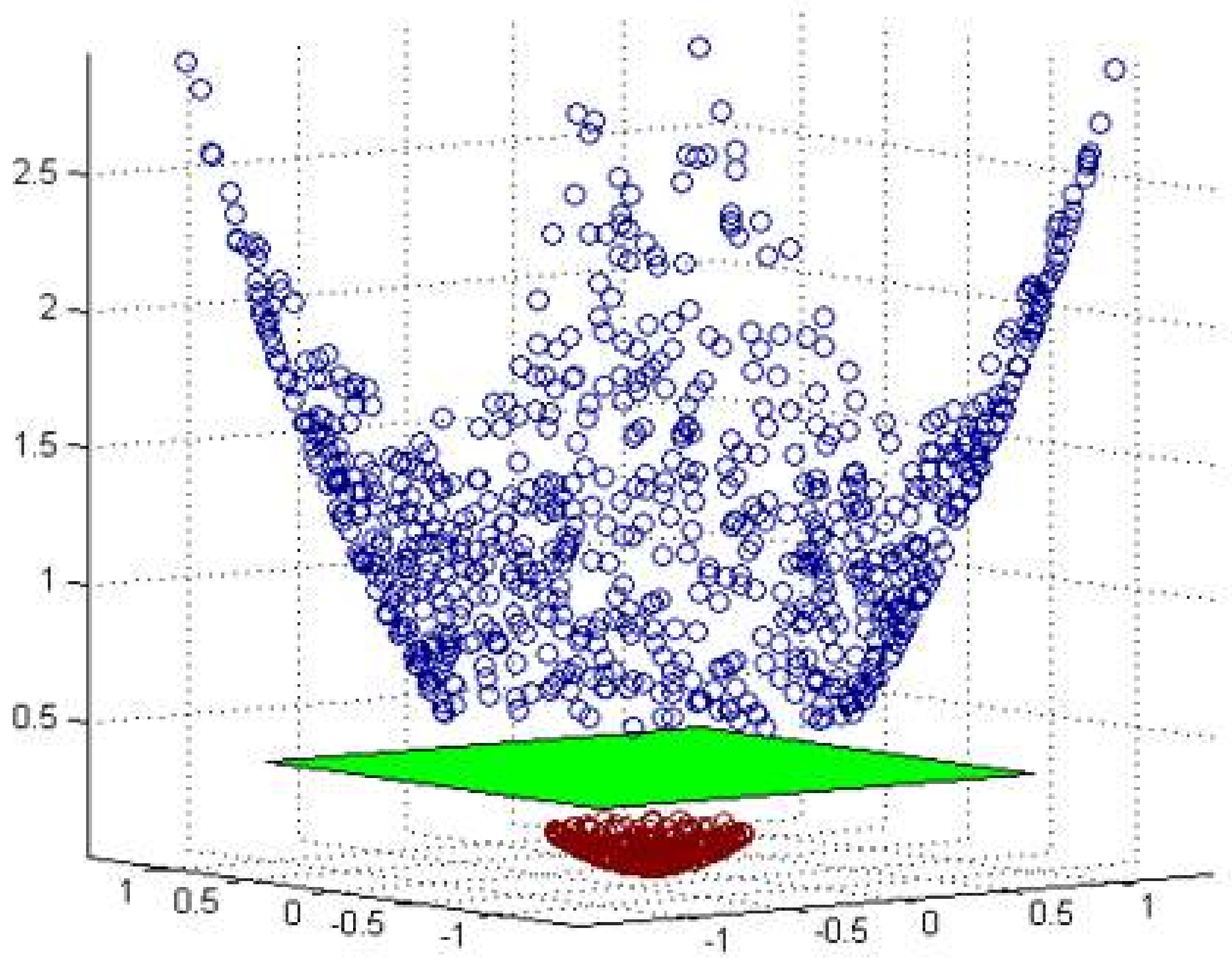
- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- We will use the `pathlib` library
- Let the `main_dir` be the variable corresponding to your `skillsoft-machine-learning-2021` folder
- `data_dir` be the variable corresponding to your `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)
```

Classification

- Classification is a type of supervised learning method It is the process of assigning new data points to known classes
- The assignment is done based on the similarity of new data points to existing data points with known class assignment (category or behavior pattern)
- In classification models, the target variable is categorical, usually binary
- It can also be multi-class like income levels - high, medium, low with n levels



Classification: use cases

- These are some examples of how you would apply classification algorithms in a business setting

Question	Example
What is this object like?	Selecting similar products at the lowest prices
Who is this person like?	Anticipating behavior or preferences of a person based on her similarities with others
What category is this in?	Anticipating if your customer is pregnant, remodeling, just got married, etc.
What is the probability that something is in a given category?	Determining the probability that a piece of equipment will fail; determining the probability that someone will buy your product

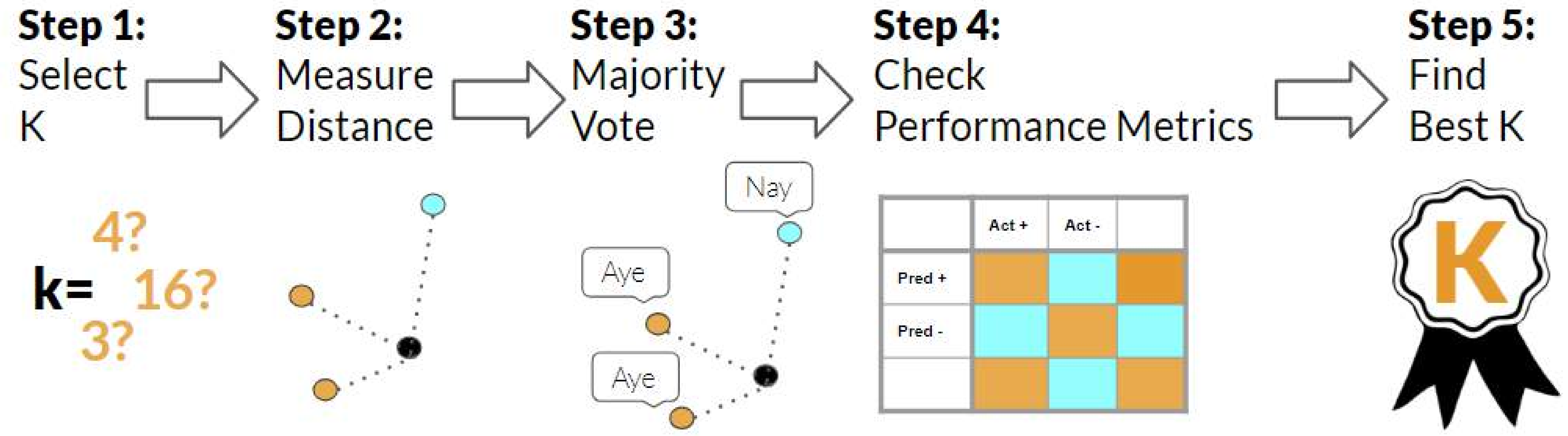
Module completion checklist

Objective	Complete
Understanding classification and its uses	✔
Summarize steps & application of kNN	
Clean and transform data to run kNN	
Define cross-validation and how and when it is used	
Implement kNN algorithm on the training data without cross-validation	

kNN: what is it?

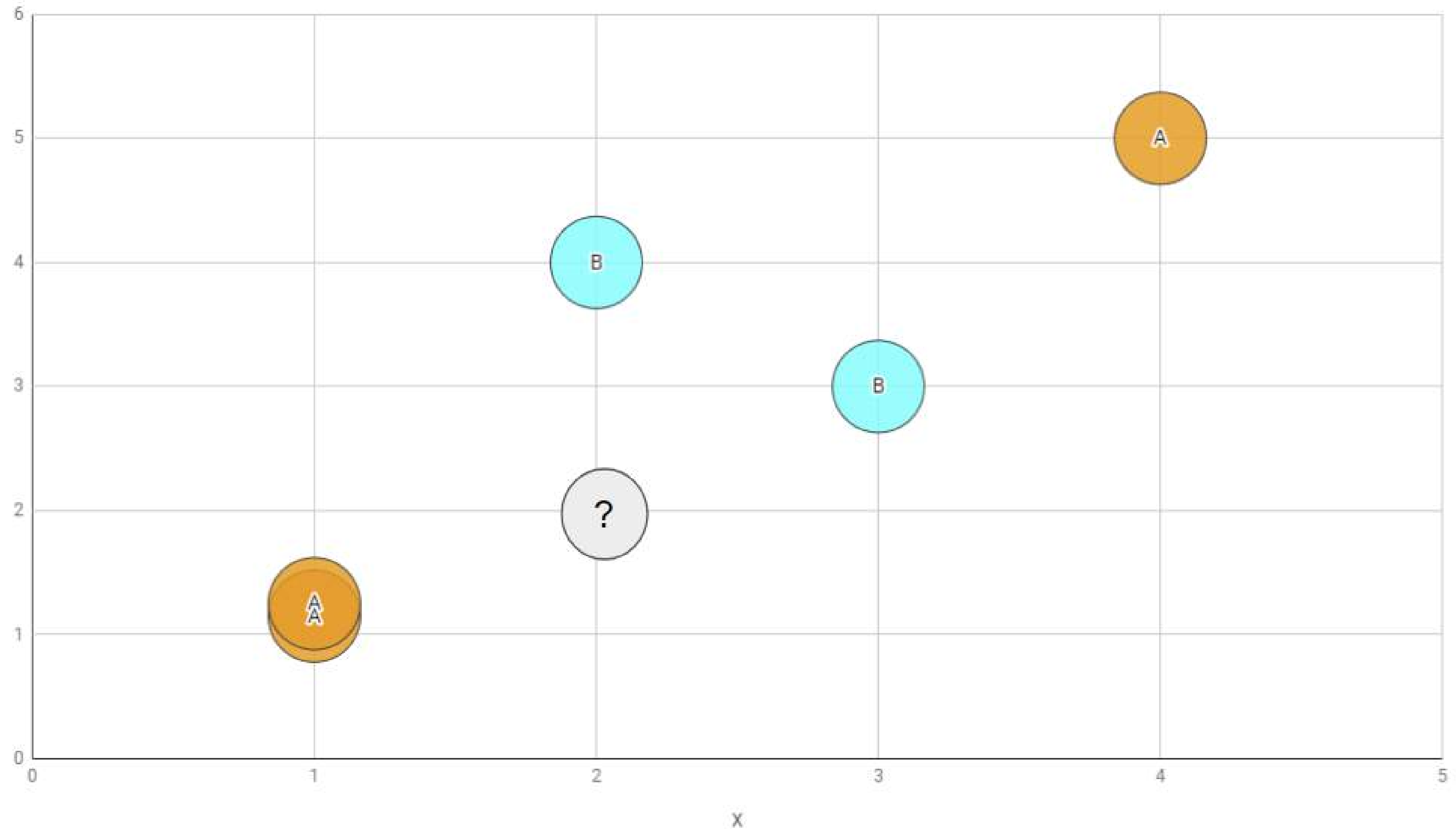
- The k-Nearest Neighbors (kNN) algorithm is a **supervised** algorithm
- It is primarily used for **classification**
- It takes **labeled points** and uses them to learn how to label other points
- It is based on an algorithm that involves distance calculation

Steps of kNN

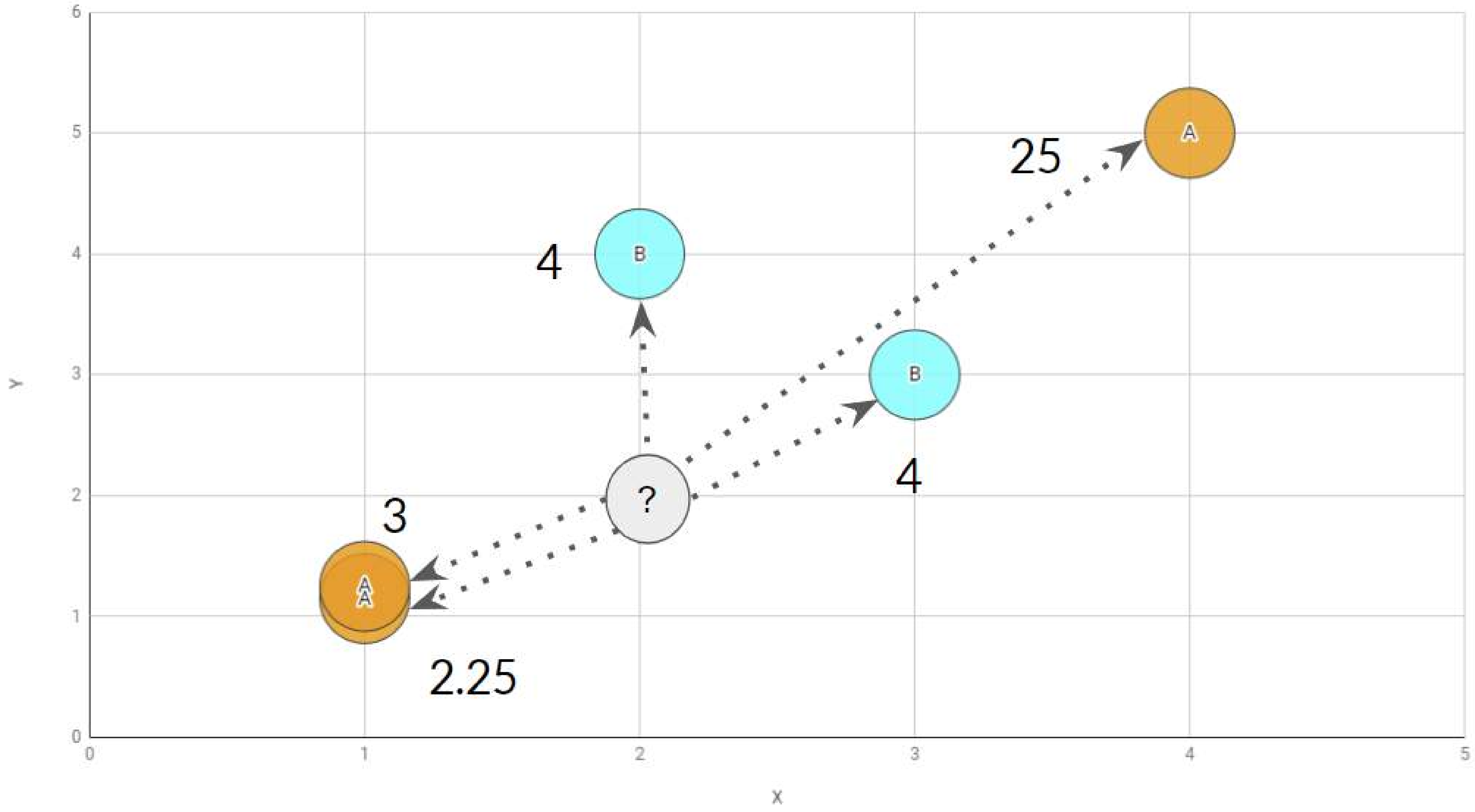


Step 1:
Select K
Step 2:
Measure Distance
Step 3:
Majority Vote
Step 4:
Check Performance Metrics
Step 5:
Find Best K
4?
k= 16?
3?
Nay
Aye
Aye
Act +
Act -
Pred +
Pred -
K

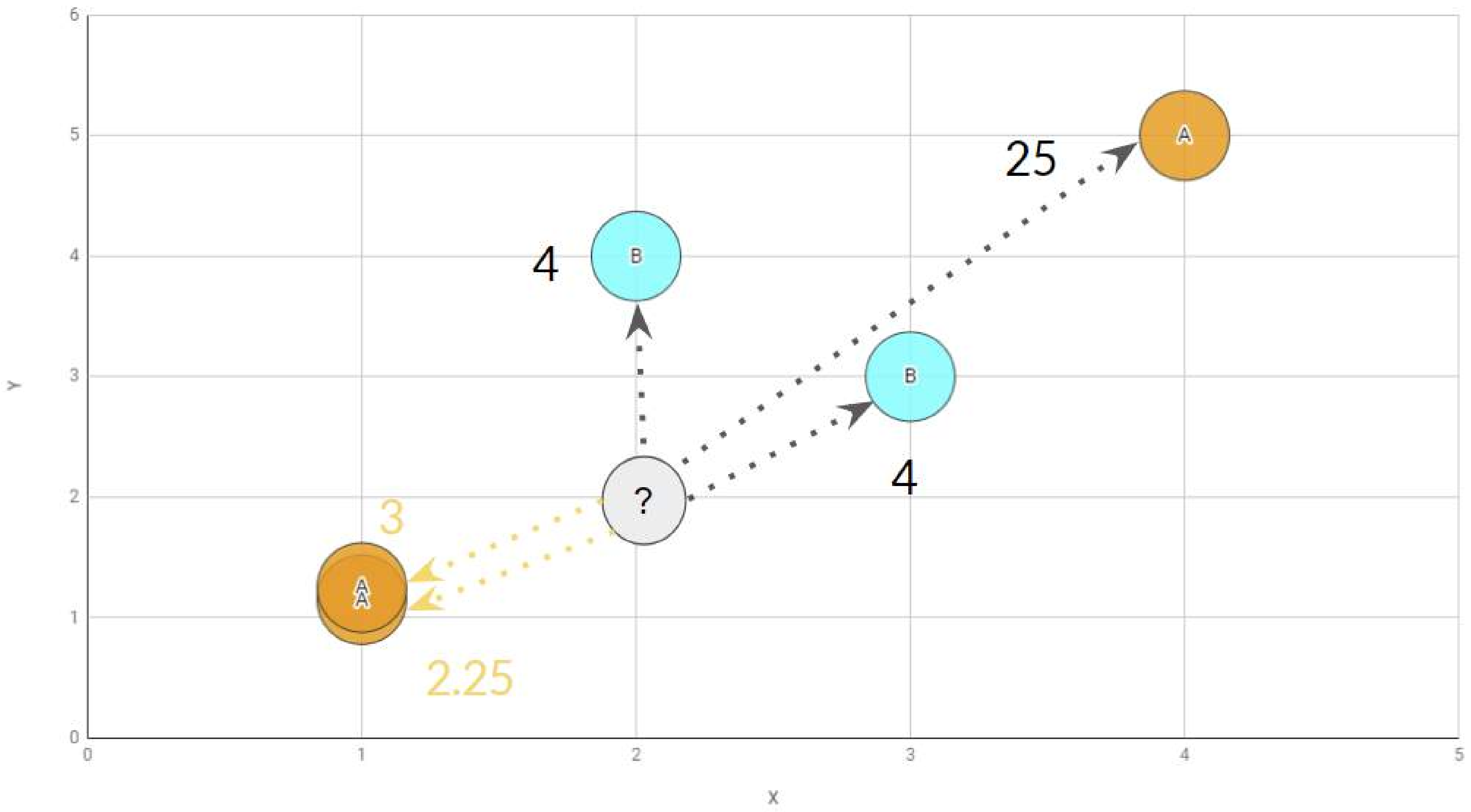
k-Nearest Neighbors: setup



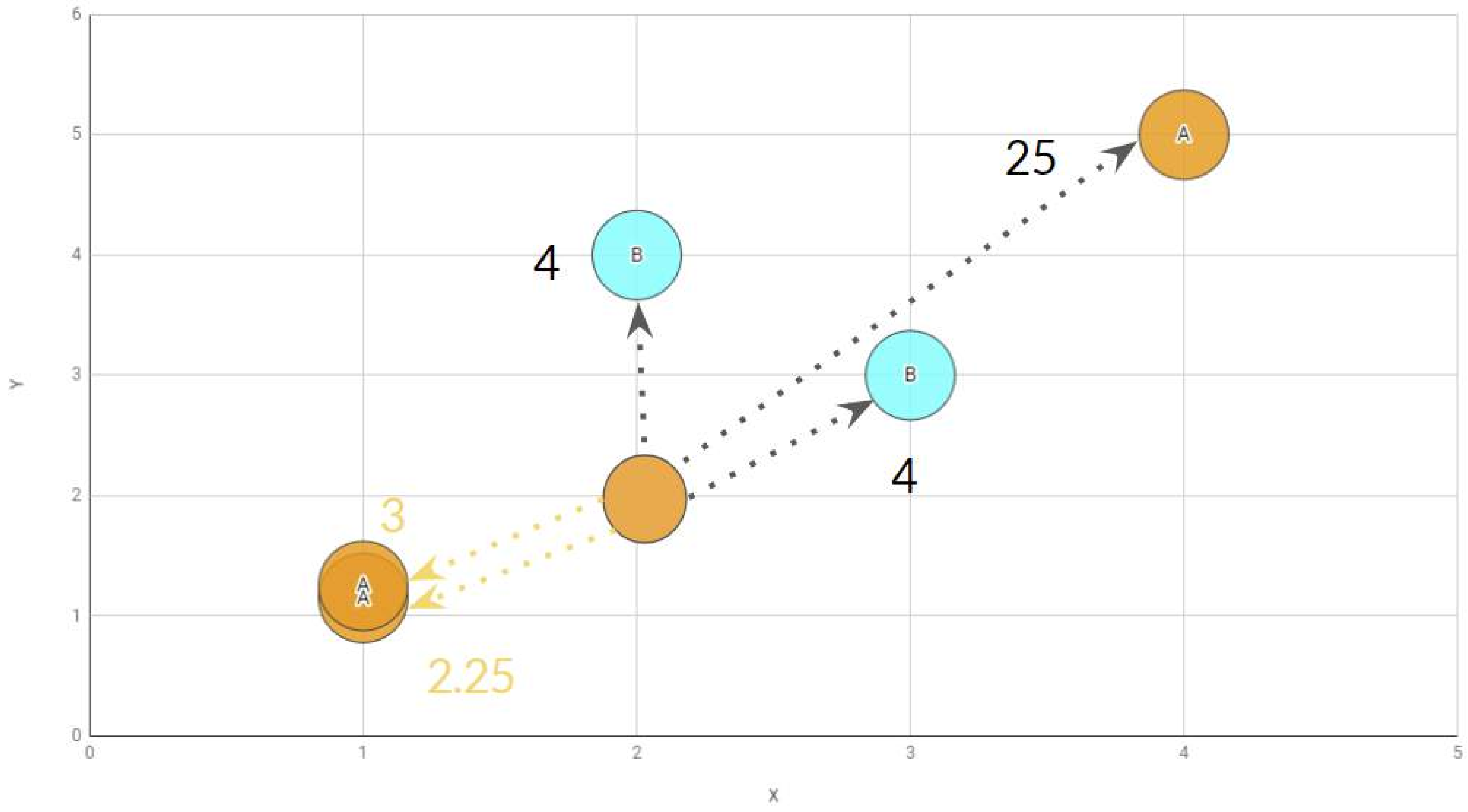
k-Nearest Neighbors: measure



k-Nearest Neighbors: 2-NN for majority vote



k-Nearest Neighbors: label point



Module completion checklist

Objective	Complete
Understanding classification and its uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	
Define cross-validation and how and when it is used	
Implement kNN algorithm on the training data without cross-validation	
Identify performance metrics for classification algorithms and evaluate simple kNN model	

Datasets for kNN

- We will be using two datasets in class today:
- One to learn the concepts: **Costa Rica household poverty data**
- One for our in-class exercises: **Chicago census data**

Costa Rican poverty: back story

- As stated by the Inter-American Development Bank (IDB):
  - Social programs have a hard time making sure the right people are given enough aid
  - It's especially tricky when a program focuses on the poorest segment of the population
  - The world's poorest typically can't provide the necessary income and expense records to prove that they qualify

Costa Rican poverty: back story

- In Latin America, one popular method to verify income qualification uses an algorithm
- It's called the **Proxy Means Test (PMT)**
- With the PMT, agencies use a model that considers a family's observable household attributes like the material of their walls and ceiling, or the assets found in the home, to classify them and predict their level of need
- While this is an improvement over other methods, accuracy remains a problem as the region's population grows and poverty declines

Costa Rican poverty: back story

- To improve the PMT, the IDB built a competition for Kaggle participants to build a model that uses methods beyond traditional econometrics

- The dataset provided contains Costa Rican household characteristics
- Four categories of poverty are targeted:
  - extreme poverty
  - moderate poverty
  - vulnerable households
  - non vulnerable households

Our goals

- Our goals with the Costa Rica household poverty data are to:
  - understand the patterns and groups within the dataset
  - predict the poverty levels of Costa Rican households
  - build a model that is also reproducible for other countries

Predicting poverty - kNN

- Today, we will be using kNN to predict poverty
- Because kNN works much better with fewer dimensions, we will be taking a small subset of the actual dataset
- As we move towards more complex machine learning algorithms, we will add more variables

Loading data into Python

- Let's load the entire dataset
- For KNN, we will be taking a specific subset
- We are now going to use the function `read_csv` to read in our `household_poverty` 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 `9,557` observations and `84` variables

Subsetting data

- In this module, we will once again subset data, however this time we will use some new variables:
  - **household id**
  - **rooms**
  - **num_adults**
  - *Target*
- We don't want to use `monthly_rent` as a variable right now because it has so many NAs
- We want to see if maybe the **number of rooms** and **number of adults** would predict the poverty level well

Subsetting data

- Let's subset our data so that we have the variables we need for building `kNN`
- Once again, we are keeping `household_id`, `rooms`, `num_adults`, and `Target`
- Let's name this subset `costa_knn`

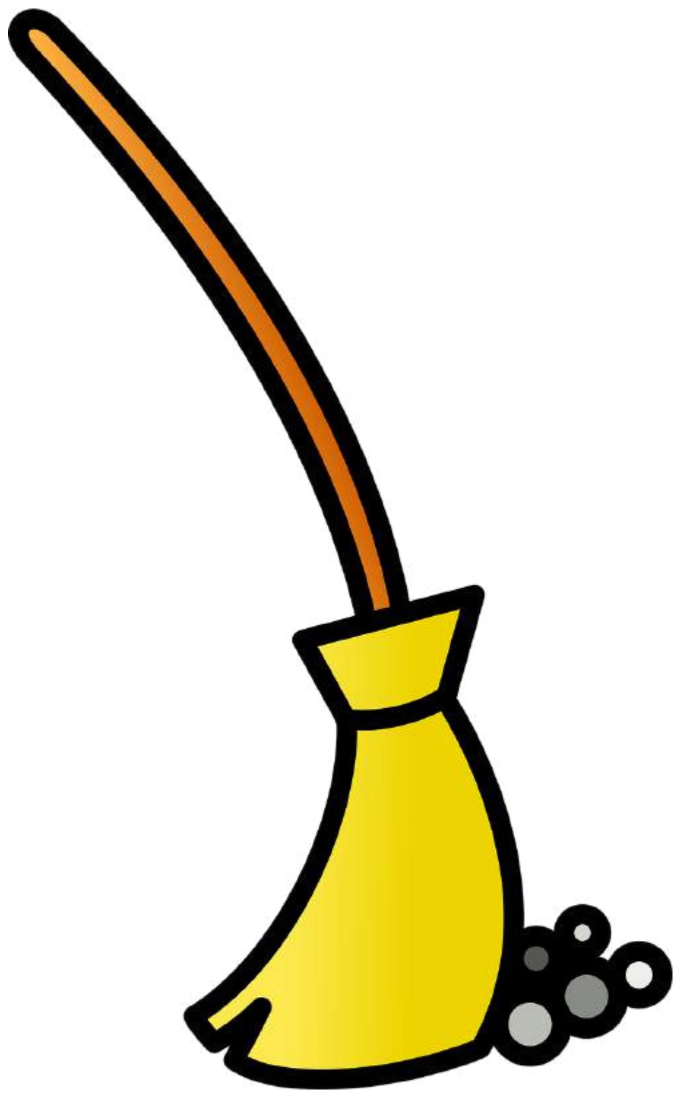
```
costa_knn = household_poverty[["household_id","rooms","num_adults","Target"]]
print(costa_knn.head())
```

```
  household_id  rooms  num_adults  Target
0    21eb7fcc1      3           1       4
1    0e5d7a658      4           1       4
2    2c7317ea8      8           1       4
3    2b58d945f      5           2       4
4    2b58d945f      5           2       4
```

Data cleaning steps for kNN

- 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 kNN
- 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. Scale the predictors



The data at first glance

- Look at the first 5 rows and the data types

```
# The first 5 rows.
print(costa_knn.head())
```

```
  household_id  rooms  num_adults  Target
0    21eb7fcc1      3           1       4
1    0e5d7a658      4           1       4
2    2c7317ea8      8           1       4
3    2b58d945f      5           2       4
4    2b58d945f      5           2       4
```

```
# The data types.
print(costa_knn.dtypes)
```

```
household_id    object
rooms            int64
num_adults       int64
Target           int64
dtype: object
```

- Frequency table of the target variable

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

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

- The target variable is not well balanced
- It also has **four levels**, we are going to make it binary for now
- This will also help balance it out

Converting the target variable

- Let's convert target to a binary target variable
- 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`

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

```
print(costa_knn['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(costa_knn.isnull().sum())
```

```
household_id    0
rooms           0
num_adults      0
Target          0
dtype: int64
```

- We do not have any NAs; we are now ready to scale our predictors!

Data prep: numeric variables

- In `kNN`, we 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(costa_knn.dtypes)
```

```
household_id     object
rooms             int64
num_adults        int64
Target           object
dtype: object
```

Data prep: ready for kNN

- 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(costa_knn.Target.dtypes)
```

```
object
```

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

```
costa_knn["Target"] = np.where(costa_knn["Target"] == "non_vulnerable", True, False)
# Check class again.
print(costa_knn.Target.dtypes)
```

```
bool
```

- Let's save this cleaned dataset as a .csv file (We will be using this dataset in our next module)

```
costa_knn.to_csv(data_dir + "/costa_knn_cleaned.csv", index = False)
```

Data prep: scaling variables

- Once the data is converted to `numeric` (if necessary), we **scale** the dataset to make sure that we can properly calculate the relationship between variables
- There are a few methods to scale data and we will use the `scale` function from `sklearn.preprocessing`
- A few things to remember about `scale`:
  - it is a generic function whose default method **centers** and/or scales the columns of a numeric matrix
  - it will convert your dataset to have a `mean` of `0` and a `standard deviation` of `1`

sklearn.preprocessing.scale

sklearn.preprocessing.**scale**(*X, axis=0, with_mean=True, with_std=True, copy=True*) [source]

Standardize a dataset along any axis

Center to the mean and component wise scale to unit variance.

Read more in the User Guide.

Parameters:

X : *{array-like, sparse matrix}*
The data to center and scale.

axis : *int (0 by default)*
axis used to compute the means and standard deviations along. If 0, independently standardize each feature, otherwise (if 1) standardize each sample.

with_mean : *boolean, True by default*
If True, center the data before scaling.

with_std : *boolean, True by default*
If True, scale the data to unit variance (or equivalently, unit standard deviation).

copy : *boolean, optional, default True*
set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSC matrix and if axis is 1).

Data prep: scaling variables

- To scale only our predictors, we split our data into `X` and `y`

```
# Split the data into X and y - y is categorical, so can't scale.
X = costa_knn[['rooms', 'num_adults']]
y = np.array(costa_knn['Target'])

# Scale X.
X_scaled = scale(X)
print(X_scaled[0:5])
```

```
[[-1.33182893 -1.3657179 ]
 [-0.65077114 -1.3657179 ]
 [ 2.07346003 -1.3657179 ]
 [ 0.03028665 -0.5080948 ]
 [ 0.03028665 -0.5080948 ]]
```

Knowledge check 1

Exercise 1

Module completion checklist

Objective	Complete
Understanding classification and its uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	
Implement kNN algorithm on the training data without cross-validation	
Identify performance metrics for classification algorithms and evaluate simple kNN model	

Introducing cross-validation

- Before applying any machine learning algorithms on the data, we usually need to split the data into a **training set** and a **test set**
- But now, we are doing this **multiple times**
- We have a new **test set** for each fold `n`
- The rest of the data is the **training set**

Why do we use cross-validation?

- Cross-validation is helpful in multiple ways:
  - It tunes our model better by running it multiple times on our data (instead of just once on the training set and once on the test set)
  - You get assurance that your model has most of the patterns from the data correct and it’s not picking up too much of the noise
  - It finds optimal parameters for your model because it runs multiple times

Cross-validation: train and test

Train

- This is the data that you train your model on
- Use a larger portion of the data to train so that the model gets a large enough sample of the population
- Usually about **70%** of your dataset
- **When there is not a large population to begin with, cross-validation techniques can be implemented**

Test

- This is the data that you test your model on
- Use a smaller portion to test your trained model on
- Usually about **30%** of your dataset
- **When cross-validation is implemented, small test sets will be held out multiple times**

Cross-validation: n-fold

Here is how cross-validation works:

1. Split the dataset into several subsets (“n” number of subsets) of equal size
2. **Use each subset as the test dataset** and **use the rest of the data as the training dataset**
3. Repeat the process for every subset you create

	Data	x	y	z
Test	1	...	...	...
	2	...	...	...
	3	...	...	...
Train	4	...	...	...
	5	...	...	...
	6	...	...	...

Data	x	y	z
1	...	...	...
2	...	...	...
3	...	...	...
4	...	...	...
5	...	...	...
6	...	...	...

Data	x	y	z
1	...	...	...
2	...	...	...
3	...	...	...
4	...	...	...
5	...	...	...
6	...	...	...

Train & test: small scale before n-fold

- Before we actually use n-fold cross-validation:
  - We split our data into a train and test set
  - We run kNN initially on the training data

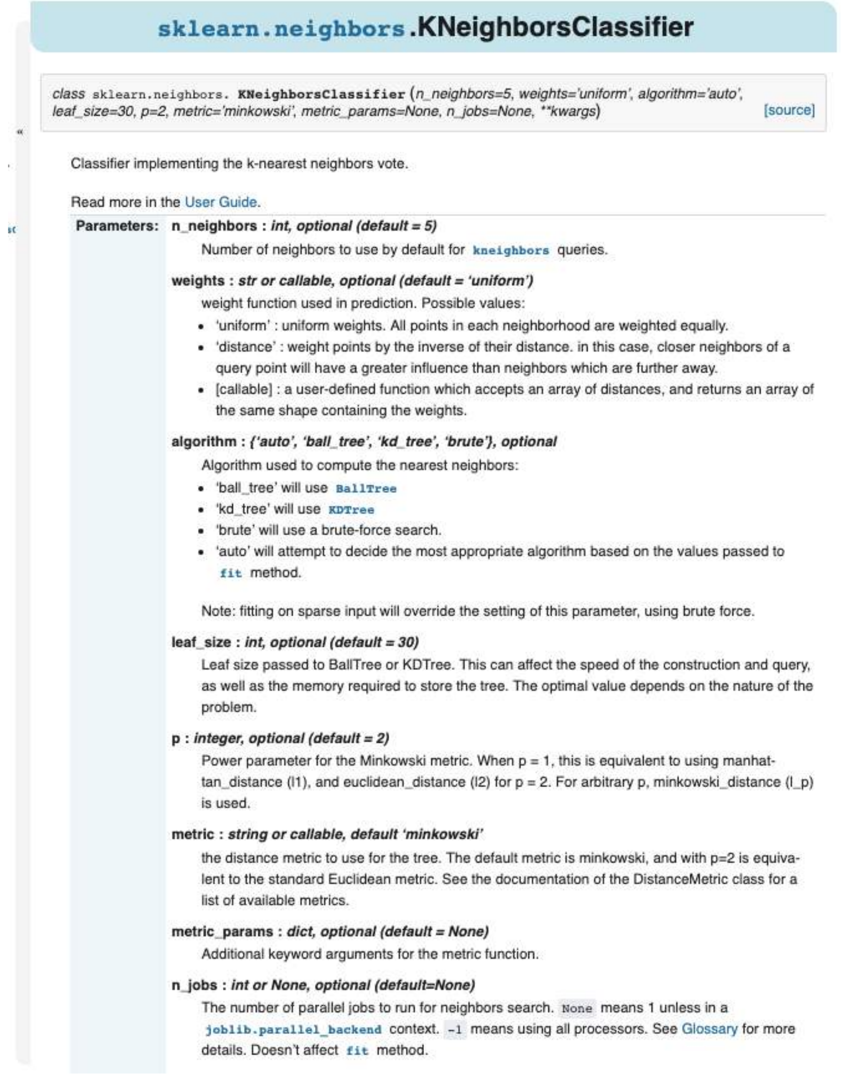
```
# Set the seed.
np.random.seed(1)

# Split into train and test.
X_train, X_test, y_train, y_test = train_test_split(X_scaled,
                                                     y,
                                                     test_size = 0.3)
```

Module completion checklist

Objective	Complete
Understanding classification and its uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	✔
Implement kNN algorithm on the training data without cross-validation	
Identify performance metrics for classification algorithms and evaluate simple kNN model	

kNN: modeling with KNeighborsClassifier

- We will use the `sklearn.neighbors` function, `KNeighborsClassifier`
- We will be using mostly `sklearn` modules and functions for classification and machine learning



sklearn.neighbors.KNeighborsClassifier

class `sklearn.neighbors.` **`KNeighborsClassifier`** (*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs*) [source]

Classifier implementing the k-nearest neighbors vote.

Read more in the User Guide.

Parameters:

n_neighbors : ***int, optional (default = 5)***

Number of neighbors to use by default for `kneighbors` queries.

weights : ***str or callable, optional (default = 'uniform')***

weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

algorithm : ***{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional***

Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size : ***int, optional (default = 30)***

Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p : ***integer, optional (default = 2)***

Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric : ***string or callable, default 'minkowski'***

the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics.

metric_params : ***dict, optional (default = None)***

Additional keyword arguments for the metric function.

n_jobs : ***int or None, optional (default=None)***

The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details. Doesn't affect `fit` method.

kNN: build model

- We now will instantiate our kNN model and run it on `X_train`
- At first, we will simply run the model on our training data and predict on test
- We set `n_neighbors = 5` as a random guess; usually we can use 3 or 5
- We will use cross-validation to optimize our model next time
- Using this process, we will also choose the best `n_neighbors` for an optimal result

```
# Create KNN classifier.
knn = KNeighborsClassifier(n_neighbors = 5)
# Fit the classifier to the data.
knn.fit(X_train, y_train)
```

```
KNeighborsClassifier()
```

Note that we typically choose an odd number of nearest neighbors to ensure that there are no 'ties'

kNN: predict on test

- Now we will take our trained model and predict on the test set

```
predictions = knn.predict(X_test)
```

- What we get is a vector of predicted values

```
print(predictions[0:5])
```

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

kNN: predict on test

- Let's quickly glance at our first five **actual observations** vs our first five **predicted observations**
- This is helpful because we have the actual values for this sample

```
actual_v_predicted = np.column_stack((y_test, predictions))
print(actual_v_predicted[0:5])
```

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

- At first glance, it looks like our model did well!

Knowledge check 2

Module completion checklist

Objective	Complete
Understanding classification and its uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	✔
Implement kNN algorithm on the training data without cross-validation	✔
Identify performance metrics for classification algorithms and evaluate simple kNN model	

Classification: assessing performance

- Our outcome variable is **binary**, and we need to understand how to measure error in classification problems

- The following terms are very important to measure performance of a classification algorithm
  - Confusion matrix
  - Accuracy
  - Receiver operating characteristic (ROC) curve
  - Area under the curve (AUC)

Classification: sklearn.metrics

- `sklearn.metrics` has many packages that are used to calculate metrics for various models
- We will be using metrics found within the *Classification metrics* section
- Here is an idea of what we can calculate using this library

Classification metrics

See the Classification metrics section of the user guide for further details.

`metrics.accuracy_score` (y_true, y_pred[, …])	Accuracy classification score.
`metrics.auc` (x, y[, reorder])	Compute Area Under the Curve (AUC) using the trapezoidal rule
`metrics.average_precision_score` (y_true, y_score)	Compute average precision (AP) from prediction scores
`metrics.balanced_accuracy_score` (y_true, y_pred)	Compute the balanced accuracy
`metrics.brier_score_loss` (y_true, y_prob[, …])	Compute the Brier score.
`metrics.classification_report` (y_true, y_pred)	Build a text report showing the main classification metrics
`metrics.cohen_kappa_score` (y1, y2[, labels, …])	Cohen's kappa: a statistic that measures inter-annotator agreement.
`metrics.confusion_matrix` (y_true, y_pred[, …])	Compute confusion matrix to evaluate the accuracy of a classification
`metrics.f1_score` (y_true, y_pred[, labels, …])	Compute the F1 score, also known as balanced F-score or F-measure
`metrics.fbeta_score` (y_true, y_pred, beta[, …])	Compute the F-beta score
`metrics.hamming_loss` (y_true, y_pred[, …])	Compute the average Hamming loss.
`metrics.hinge_loss` (y_true, pred_decision[, …])	Average hinge loss (non-regularized)
`metrics.jaccard_similarity_score` (y_true, y_pred)	Jaccard similarity coefficient score
`metrics.log_loss` (y_true, y_pred[, eps, …])	Log loss, aka logistic loss or cross-entropy loss.
`metrics.matthews_corrcoef` (y_true, y_pred[, …])	Compute the Matthews correlation coefficient (MCC)
`metrics.precision_recall_curve` (y_true, …)	Compute precision-recall pairs for different probability thresholds
`metrics.precision_recall_fscore_support` (…)	Compute precision, recall, F-measure and support for each class
`metrics.precision_score` (y_true, y_pred[, …])	Compute the precision
`metrics.recall_score` (y_true, y_pred[, …])	Compute the recall
`metrics.roc_auc_score` (y_true, y_score[, …])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.
`metrics.roc_curve` (y_true, y_score[, …])	Compute Receiver operating characteristic (ROC)
`metrics.zero_one_loss` (y_true, y_pred[, …])	Zero-one classification loss.

Confusion matrix: what is it

- A **confusion matrix** is what we use to measure error
- We use it to calculate Accuracy, Misclassification rate, True positive rate, False positive rate, and Specificity
- In the matrix overview of our data, let `Y1` be "non-vulnerable" and `Y2` be "vulnerable"

	Predicted Y1	**Predicted Y2**	**Actual totals**
Y1	True Negative (TN)	False Positive (FP)	Total negatives
Y2	False Negative (FN)	True Positive (TP)	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	Total

Confusion matrix: accuracy

- We will now review the metrics we are looking for from the confusion matrix, one at a time

Accuracy: overall, how often is the classifier correct?
TP + TN / **total**

	Predicted Y1	**Predicted Y2**	**Actual totals**
Y1	**True Negative (TN)**	False Positive (FP)	Total negatives
Y2	False Negative (FN)	**True Positive (TP)**	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	**Total**

Confusion matrix: misclassification rate

Misclassification rate (error rate) : overall, how often is the classifier wrong?

FP + FN / total

	Predicted Y1	Predicted Y2	Actual totals
Y1	True Negative (TN)	False Positive (FP)	Total negatives
Y2	False Negative (FN)	True Positive (TP)	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	Total

Confusion matrix: true positive rate

True positive rate (Sensitivity): how often does it predict yes?

TP / **actual yes**

	Predicted Y1	Predicted Y2	Actual totals
Y1	True Negative (TN)	False Positive (FP)	Total negatives
Y2	False Negative (FN)	**True Positive (TP)**	**Total positives**
Predicted totals	Total predicted negatives	Total predicted positives	Total

Confusion matrix: false positive rate

False positive rate: when it's actually no, how often does it predict yes?

FP / actual no

	Predicted Y1	Predicted Y2	Actual totals
Y1	True Negative (TN)	False Positive (FP)	Total negatives
Y2	False Negative (FN)	True Positive (TP)	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	Total

Confusion matrix: specificity

True Negative Rate (Specificity): when it's actually no, how often does it predict no?
TN / actual no

	Predicted Y1	Predicted Y2	Actual totals
Y1	True Negative (TN)	False Positive (FP)	Total negatives
Y2	False Negative (FN)	True Positive (TP)	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	Total

Confusion matrix: summary

- Here is a table with all the metrics in one place:

Metric name	Formula
Accuracy	True positive + True Negative / Overall total
Misclassification rate	False positive + False Negative / Overall total
True positive rate	True positive / Actual yes (True positive + False negative)
False positive rate	False positive / Actual no (False positive + True negative)
Specificity	True negative / Actual no (False positive + True negative)

Confusion matrix in Python

- Now that we know the metrics behind the madness, let's execute the code to build a confusion matrix in Python
- We use a function called `confusion_matrix` from `sklearn.metrics`

```
# Confusion matrix for knn.
cm_knn5 = confusion_matrix(y_test, predictions)
print(cm_knn5)
```

```
[[ 294  768]
 [ 366 1440]]
```

- We won't go through all of the metrics right now, but let's calculate accuracy because it's a metric used frequently to compare classification models

- **Accuracy = True positive + True Negative / Overall total**
- Using `accuracy_score` from `sklearn.metrics`, we calculate:

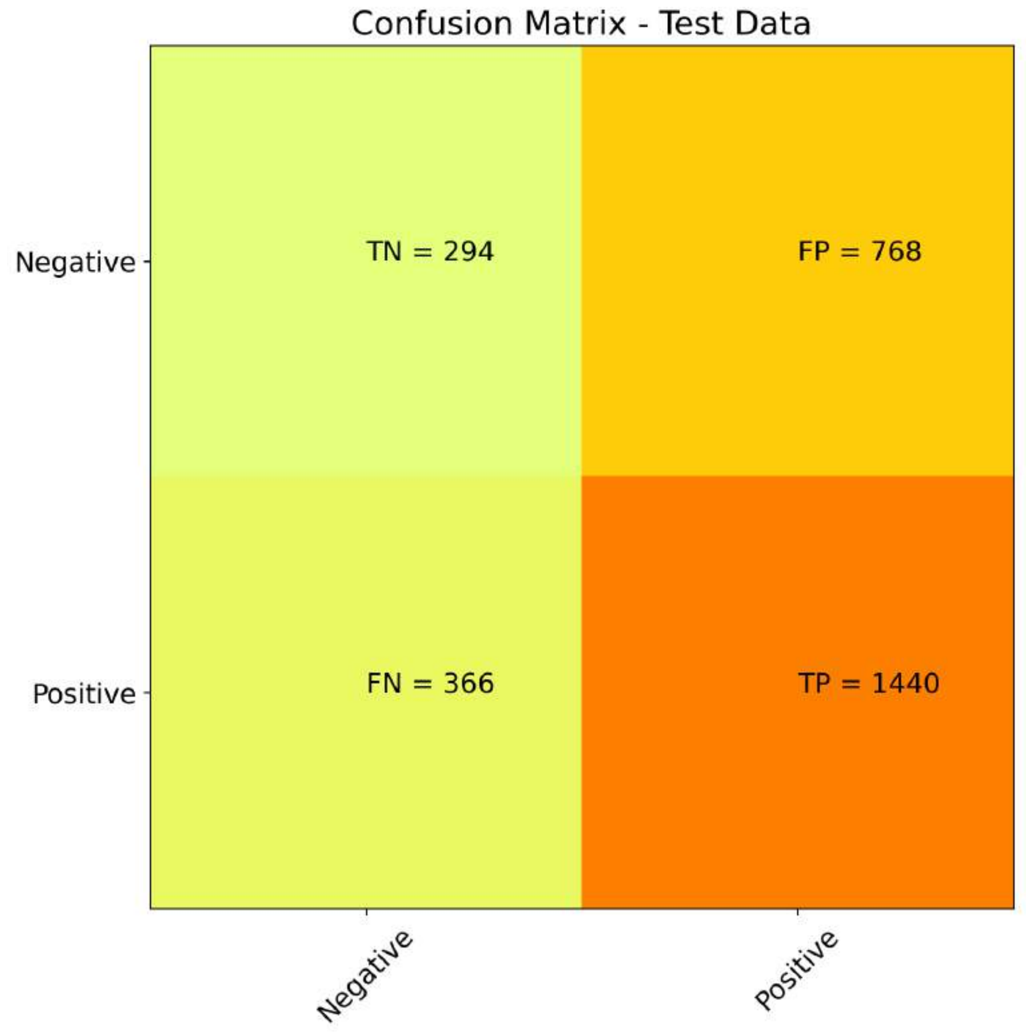
```
print(round(accuracy_score(y_test, predictions),
4))
```

```
0.6046
```

Confusion matrix: visualize

- Let’s visualize our confusion matrix

```
plt.imshow(cm_knn5, interpolation = 'nearest', cmap =
plt.cm.Wistia)
classNames = ['Negative', 'Positive']
plt.title('Confusion Matrix - Test Data')
plt.ylabel('True label')
plt.xlabel('Predicted label')
tick_marks = np.arange(len(classNames))
plt.xticks(tick_marks, classNames, rotation = 45)
plt.yticks(tick_marks, classNames)
s = [['TN', 'FP'], ['FN', 'TP']]
for i in range(2):
    for j in range(2):
        plt.text(j,i, str(s[i][j]) + " = " + str(cm_knn5[i][j]))
plt.show()
```



Evaluation of kNN with 5 neighbors

- Let's store the accuracy of this model:

```
# Create a dictionary with accuracy values for our knn model with k = 5.
model_final_dict = {'metrics': ["accuracy"],
                    'values':[round(accuracy_score(y_test, predictions), 4)],
                    'model':['knn_5']}
model_final = pd.DataFrame(data = model_final_dict)
print(model_final)
```

```
    metrics  values  model
0  accuracy  0.6046  knn_5
```

- Our model is not doing great, but we will now observe how it does compared to other models

Pickle library

- We are going to pause for a moment to learn about the library `pickle`
- When we have objects we want to carry over and do not want to rerun code, we can `pickle` these objects
- In other words, `pickle` will help us save objects from one script/ session and pull them up in new scripts
- How do we do that? We use a function in Python called `pickle`
- It is similar to **flattening** a file
  - **Pickle/saving: a Python object is converted into a byte stream**
  - **Unpickle/loading: the inverse operation where a byte stream is converted back into an object**

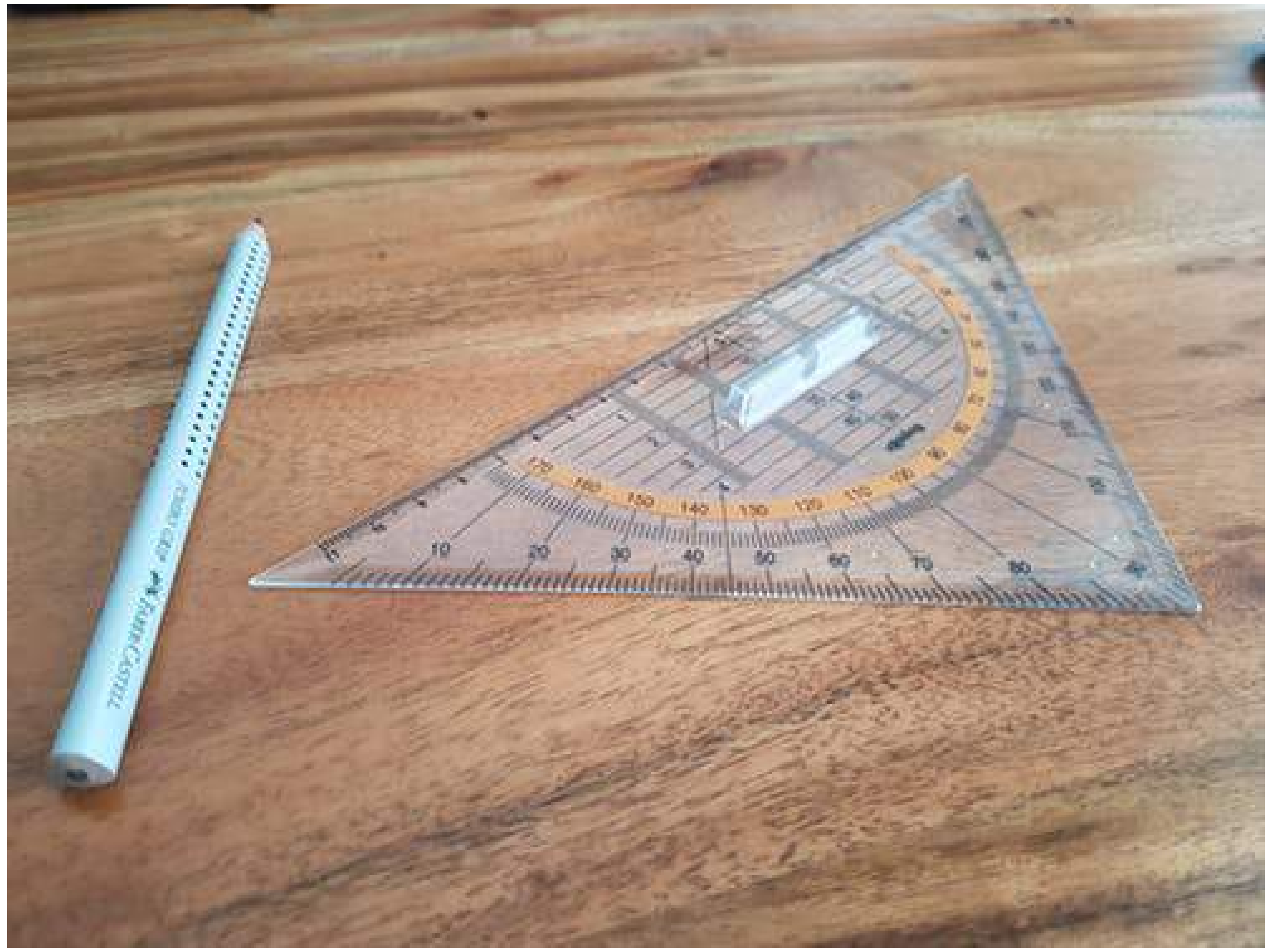
Saving the accuracy into a pickle file

- As we move forward, we will add our accuracy scores for each new model we build so that we can compare the models and evaluate which one seems to be the ***model champion***
- Pickle `model_final` dataframe as `model_final.sav`

```
pickle.dump(model_final, open(data_dir + "/model_final.sav","wb"))
```

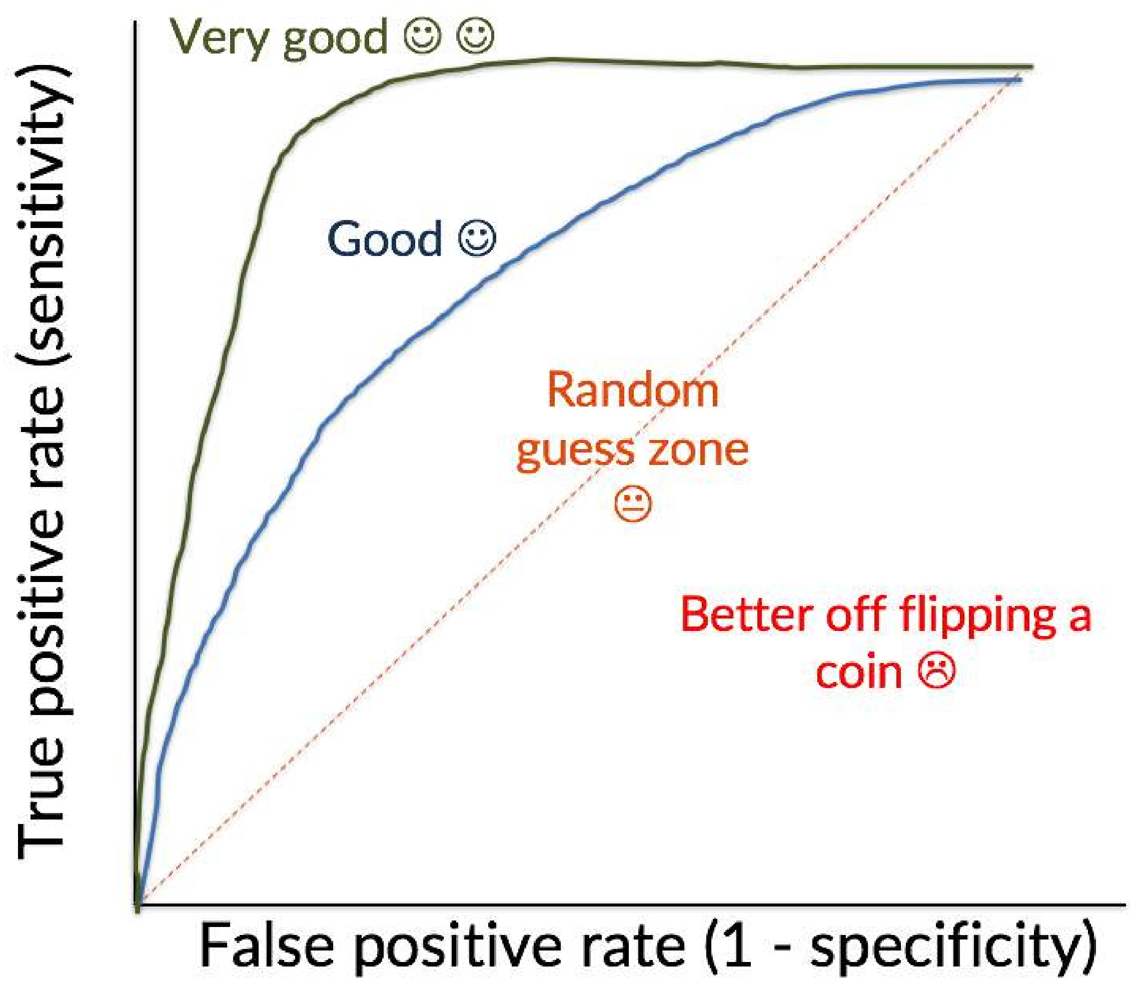
Performance of our kNN model

- The remaining metrics we want to look at to evaluate our model are:
  - Receiver operating characteristic (**ROC**) curve
  - Area under the curve (**AUC**)



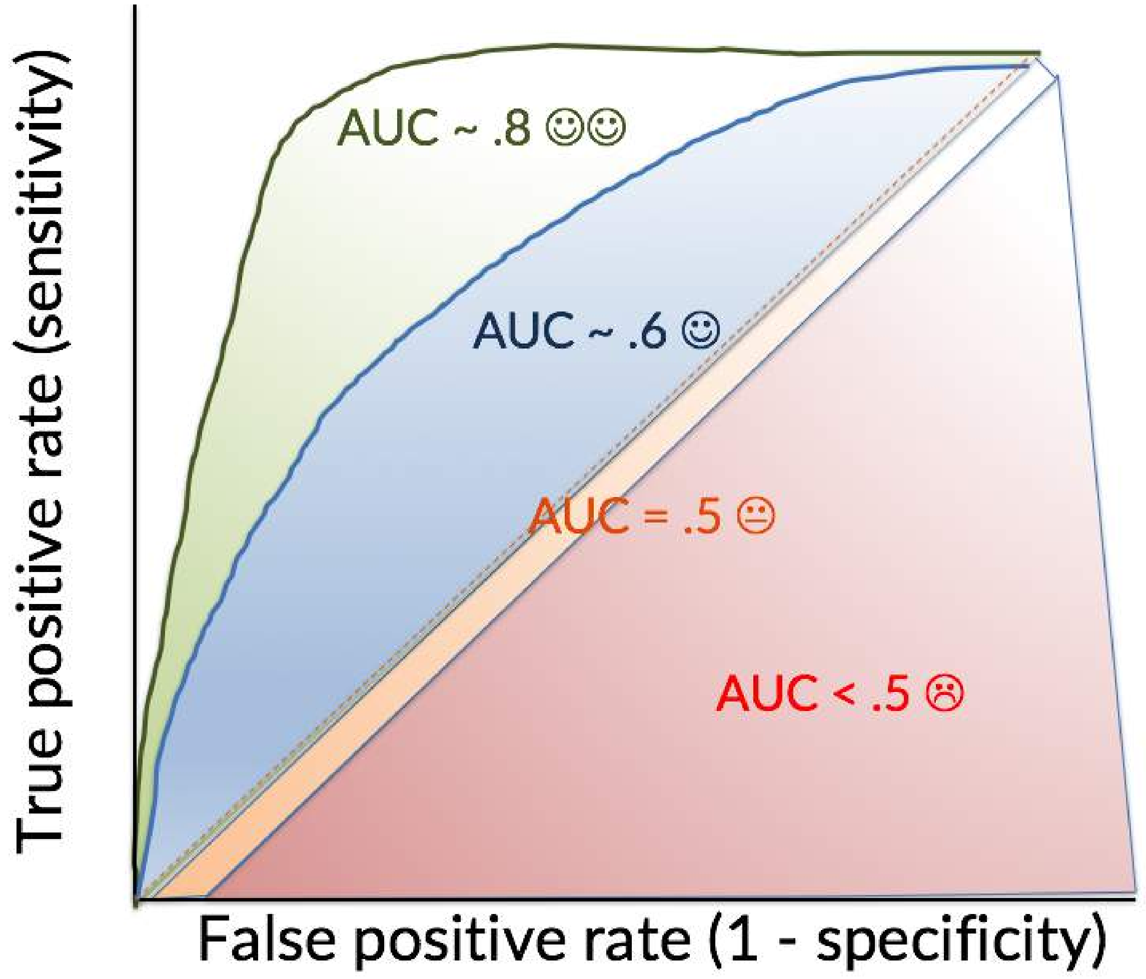
ROC: receiver operator characteristic

- ROC is a plot of the true positive rate (**TPR**) against the false positive rate (**FPR**)
- The plot illustrates the trade off between TPR and FPR
- Classification models produce them to show the performance of the model and allow us to choose which threshold to use



AUC: area under the curve

- The AUC is a **performance metric** 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 perfect AUC = `1` (you will never see this number working with real world data!)

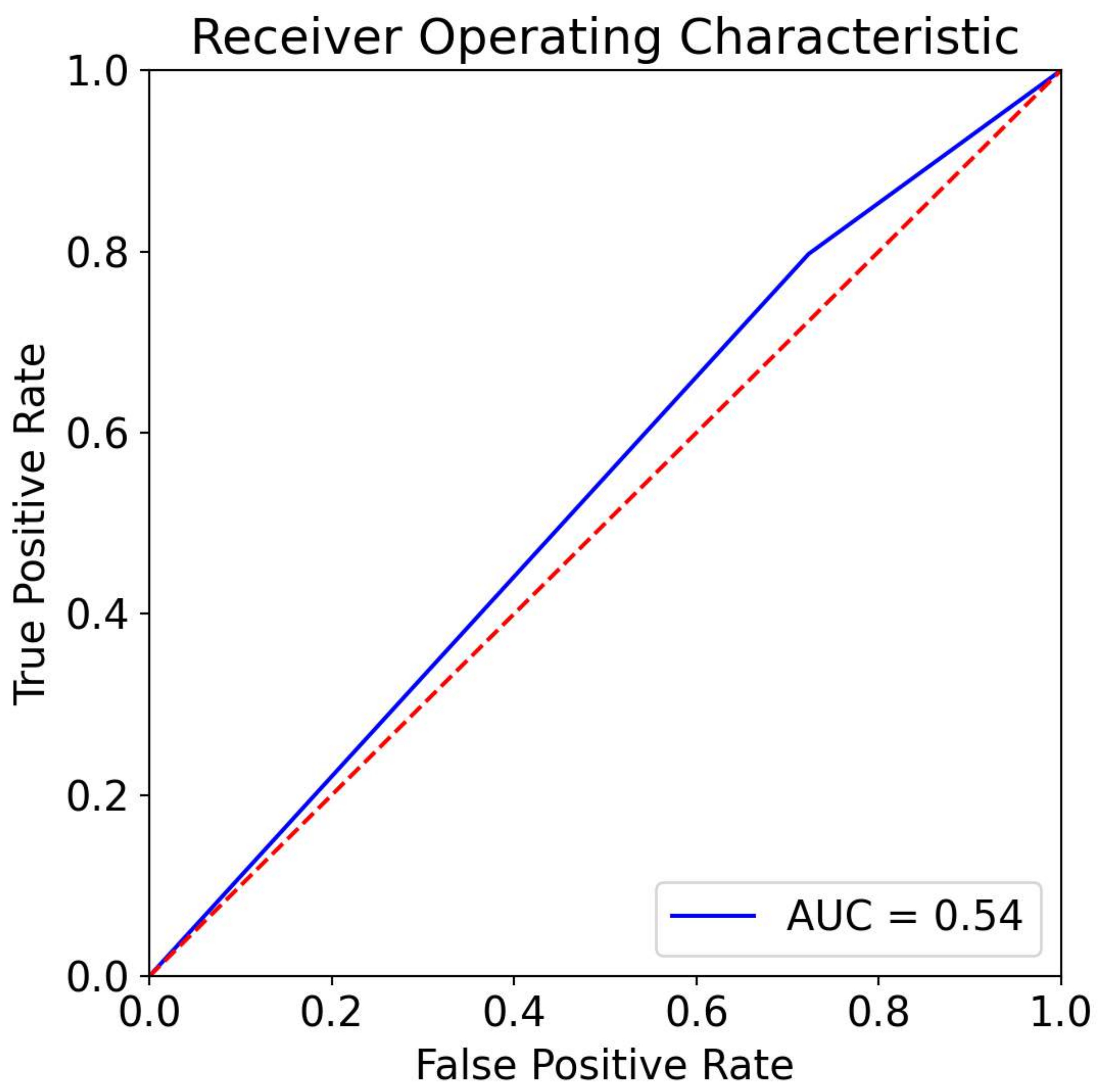


Plot ROC and calculate AUC

- Let's plot the **ROC** for our model and calculate the **AUC**

```
# Store FPR, TPR, and threshold as variables.
fpr, tpr, threshold = metrics.roc_curve(y_test,
predictions)
# Store the AUC.
roc_auc = metrics.auc(fpr, tpr)
```

```
plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' %
roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



Knowledge check 3

Exercise 2

Module completion checklist

Objective	Complete
Understanding classification and its uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	✔
Implement kNN algorithm on the training data without cross-validation	✔
Identify performance metrics for classification algorithms and evaluate simple kNN model	✔

Summary

Today we learned about

- classification and its use cases
- summary and applications of knn algorithm
- implementation of the knn algorithm on training data
- cross-validation and its use cases

Tomorrow we will - apply cross-validation to understand what is the optimal model accuracy - learn about hyperparameters and GridSearch - take a look into logistic regression ad its applications

Congratulations on completing this module!