Intro to classification - kNN - 3

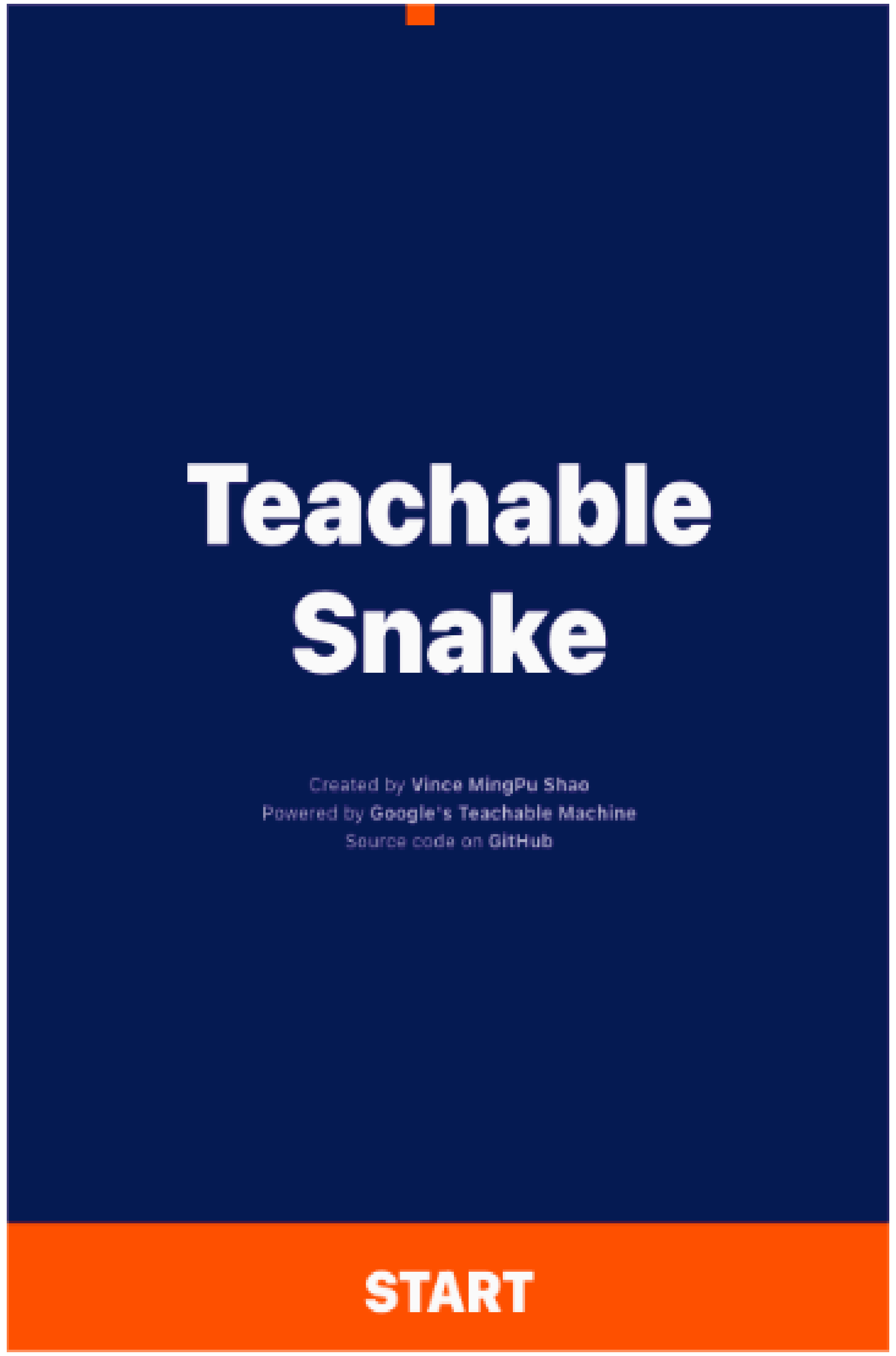
One should look for what is and not what he thinks should be. (Albert Einstein)

Field trip: Teachable Snake

- Classifying with **kNN** can seem really simple due to the fact that it just stores training data rather than actively performing calculations
- Despite its simplicity, it is often at the core of complex classification tasks
- On a **white sheet of paper** or an **index card, draw an arrow**
- Then, visit ***https://teachable-snake.netlify.app/*** to play a quick game of **Teachable Snake**

Teachable Snake debrief

- Believe it or not, kNN is at the core of Google Creative Lab's Teachable Machine, a tool for creating ML models – learn more **here**
- Google's boilerplate documentation, available via GitHub **here**, explains how to combine kNN with a neural network for image recognition
- Read more about Teachable Snake at the creator's website by clicking **here**



Module completion checklist

Objective	Complete
Implement kNN algorithm on the training data without cross-validation	
Identify performance metrics for classification algorithms and evaluate a 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.
default = 5
kNN = KNeighborsClassifier(n_neighbors = default)
# 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 a test set

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

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

- What we get is a vector of predicted values

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

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

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])
```

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

Module completion checklist

Objective	Complete
Implement kNN algorithm on the training data without cross-validation	✔
Identify performance metrics for classification algorithms and evaluate a 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

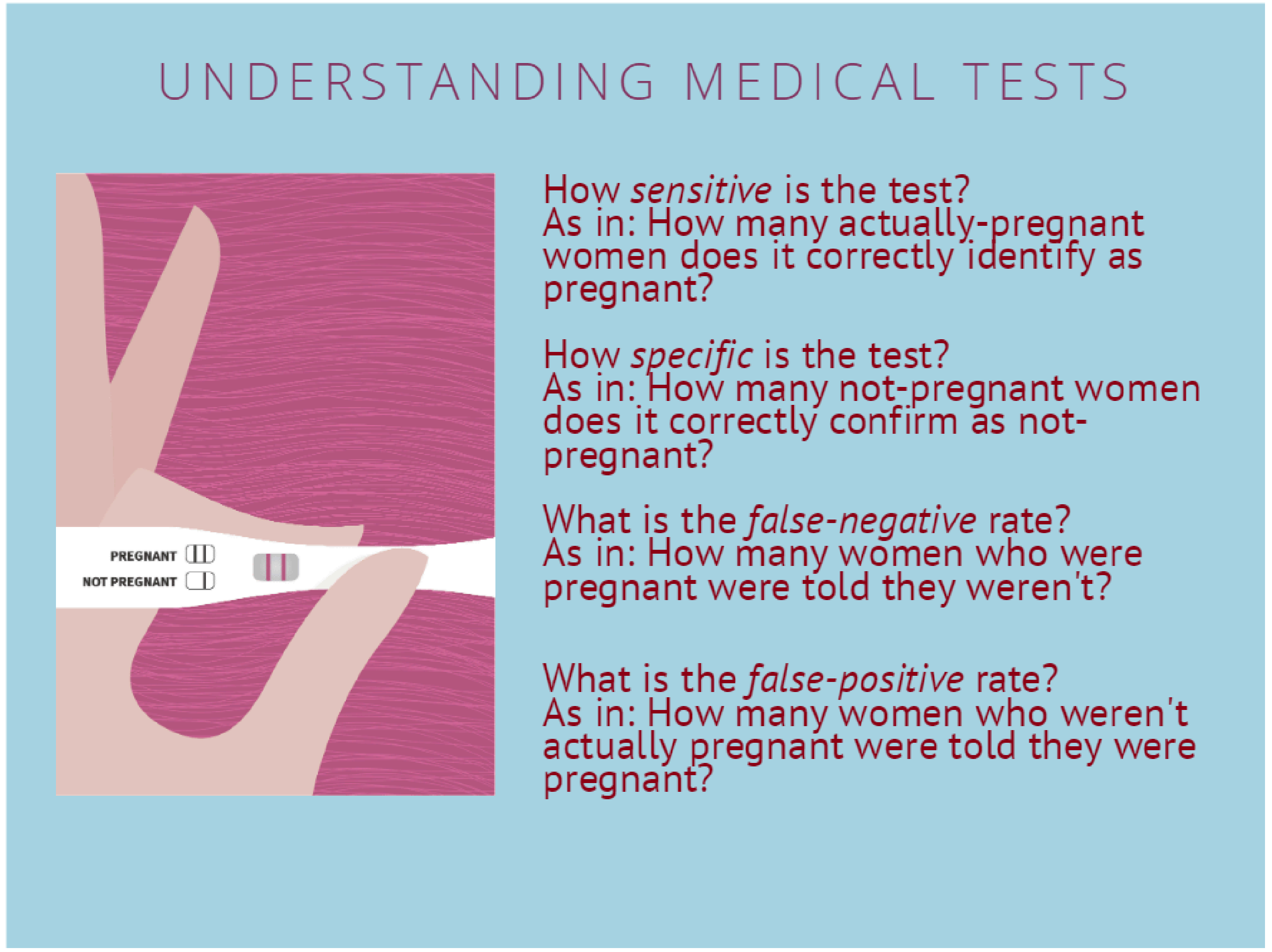
- Here is a table with all the metrics in one place:

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)

Understanding medical tests: sensitivity, specificity, and positive predictive value

- A test that's highly sensitive will flag almost everyone who has the disease and not generate many false-negative results. (Example: a test with 90% sensitivity will correctly return a positive result for 90% of people who have the disease, but will return a negative result — a false-negative — for 10% of the people who have the disease and should have tested positive.)
- A high-specificity test will correctly rule out almost everyone who doesn't have the disease and won't generate many false-positive results. (Example: a test with 90% specificity will correctly return a negative result for 90% of people who don't have the disease, but will return a positive result — a false-positive — for 10% of the people who don't have the disease and should have tested negative.)

Understanding medical tests: Pregnancy test results



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_kNN = confusion_matrix(y_test, predictions)
print(cm_kNN)
```

```
[[1443    7]
 [  82    1]]
```

- 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:

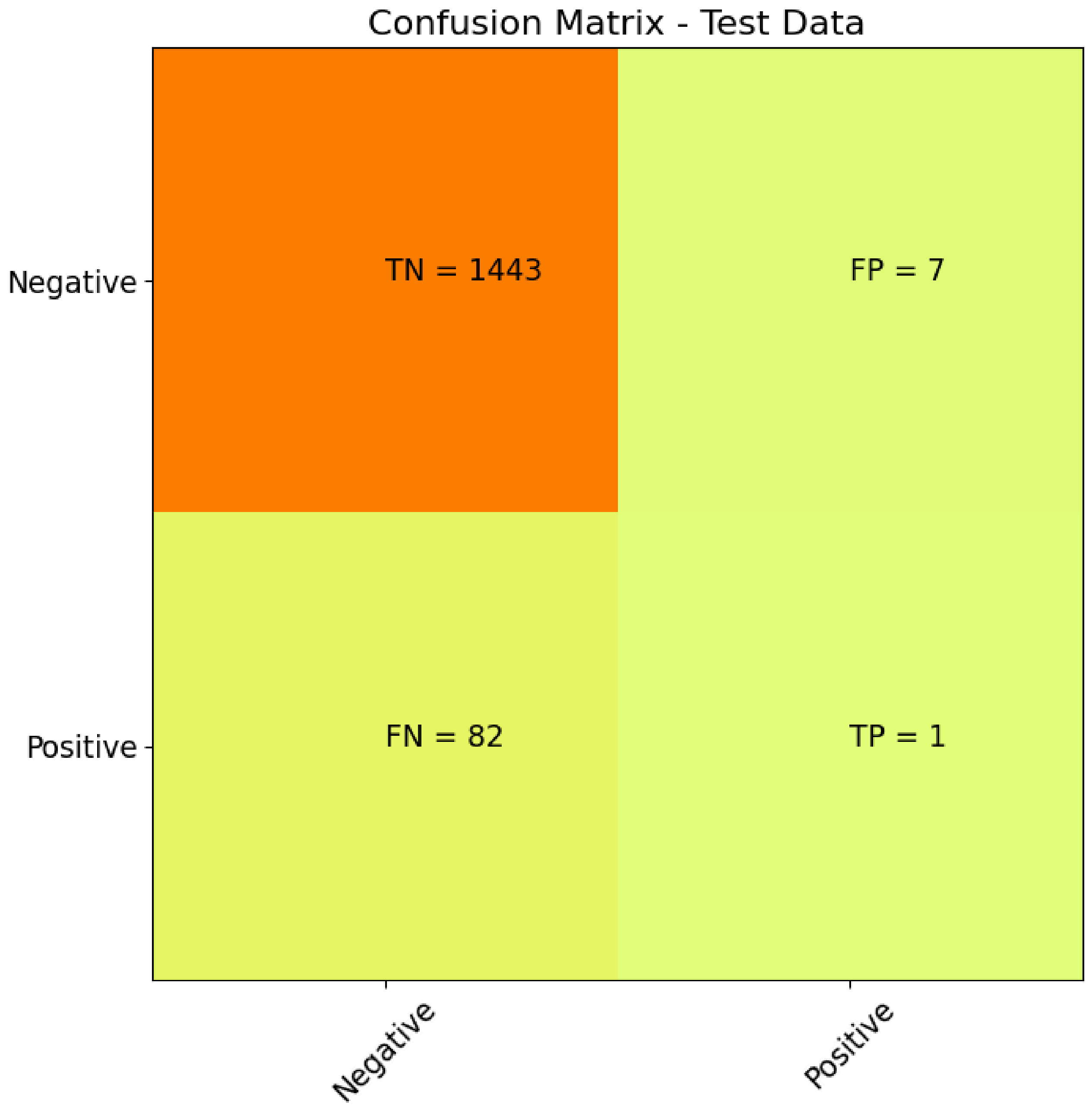
```
print(round(accuracy_score(y_test, predictions),
4))
```

```
0.9419
```

Confusion matrix: visualize

- Let's visualize our confusion matrix

```
plt.imshow(cm_kNN, 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_kNN[i]
[j]))
plt.show()
```



Evaluation of kNN with k neighbors

- Let's store the accuracy of this model. This way we can access it later to compare.

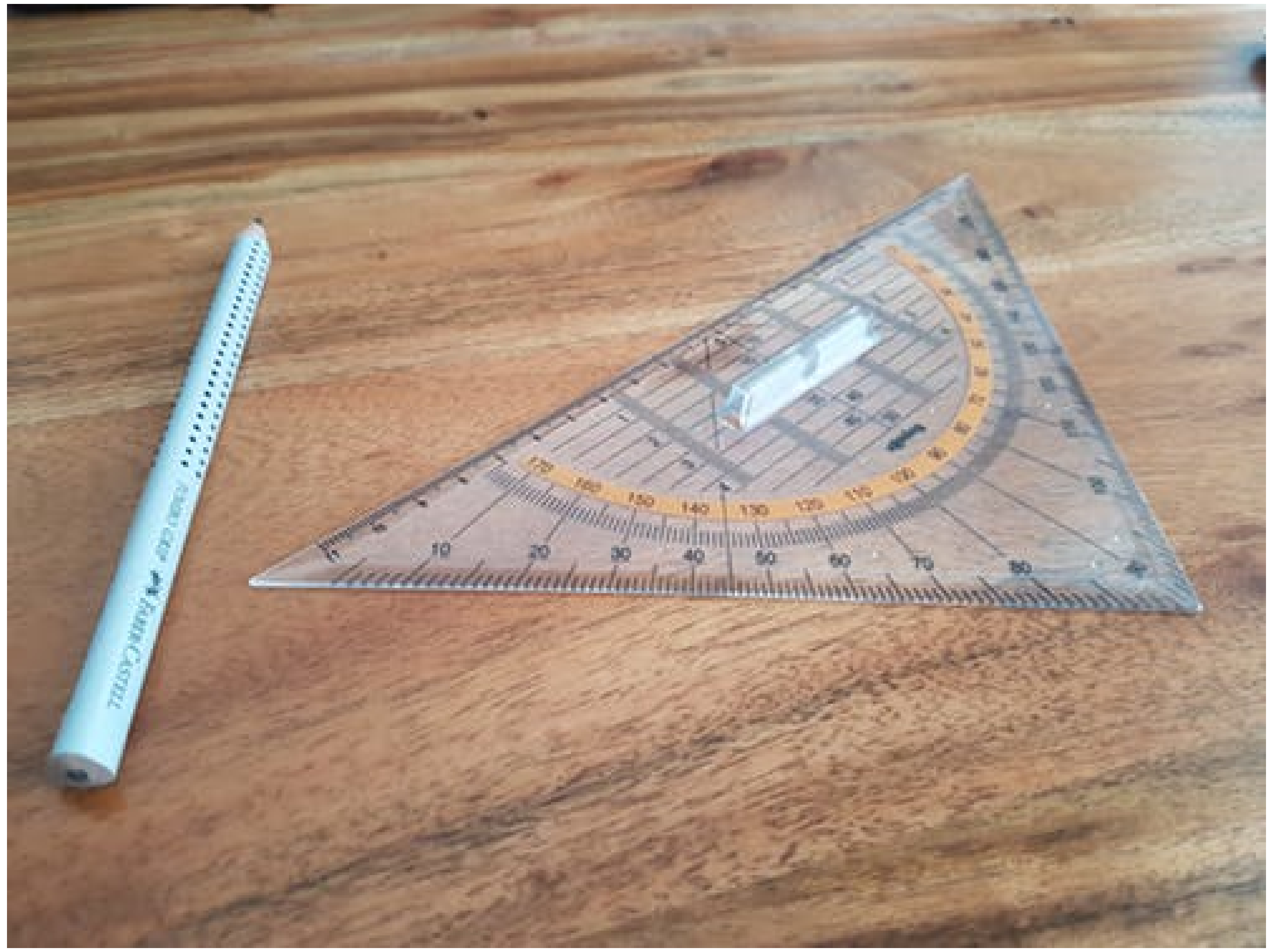
```
# Create a dictionary with accuracy values for our kNN model with k.
model_final_dict = {'metrics': ["accuracy"],
                    'values':[round(accuracy_score(y_test, predictions), 4)],
                    'model':['kNN_k']}
model_final = pd.DataFrame(data = model_final_dict)
print(model_final)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

```
    metrics  values  model
0  accuracy  0.9419  kNN_k
```

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

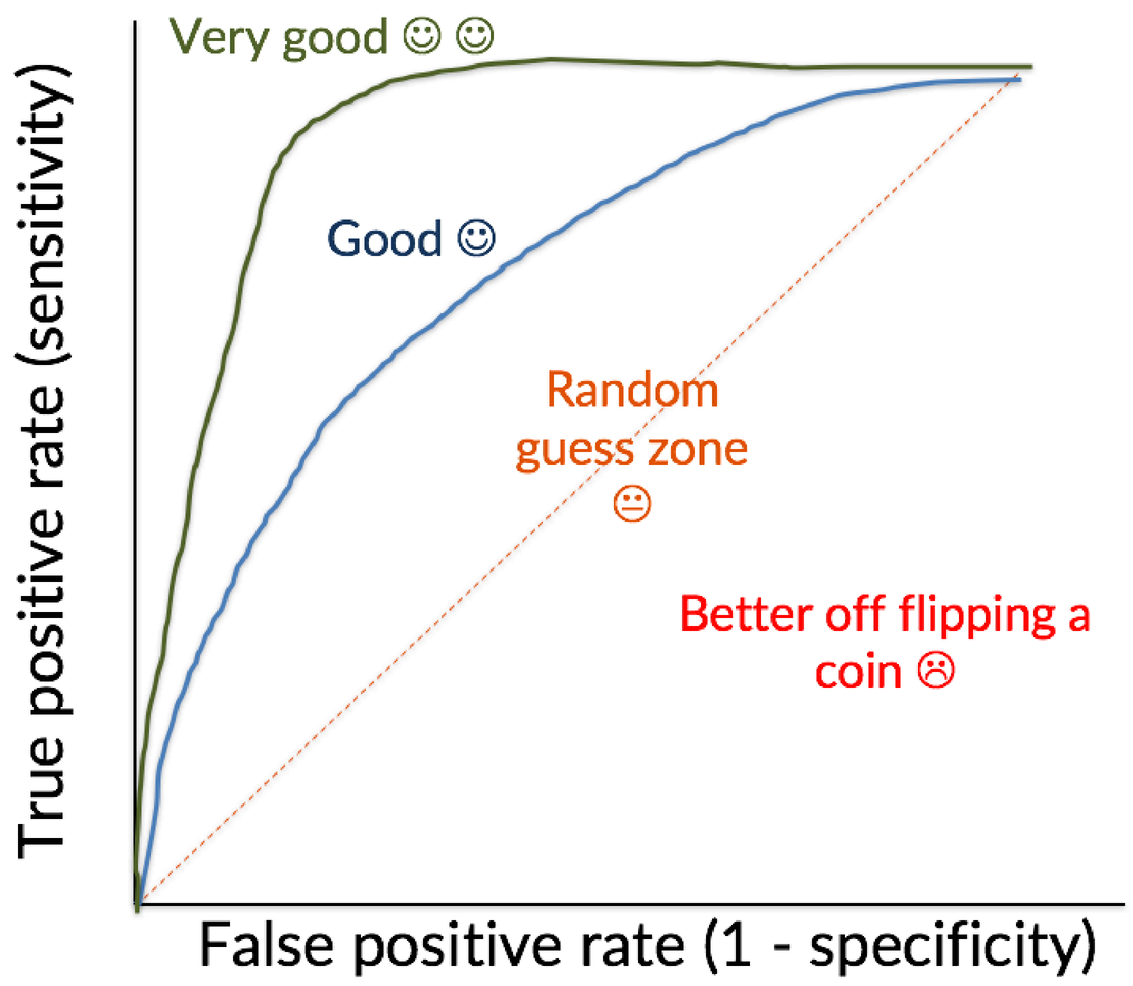
Performance of our kNN model

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



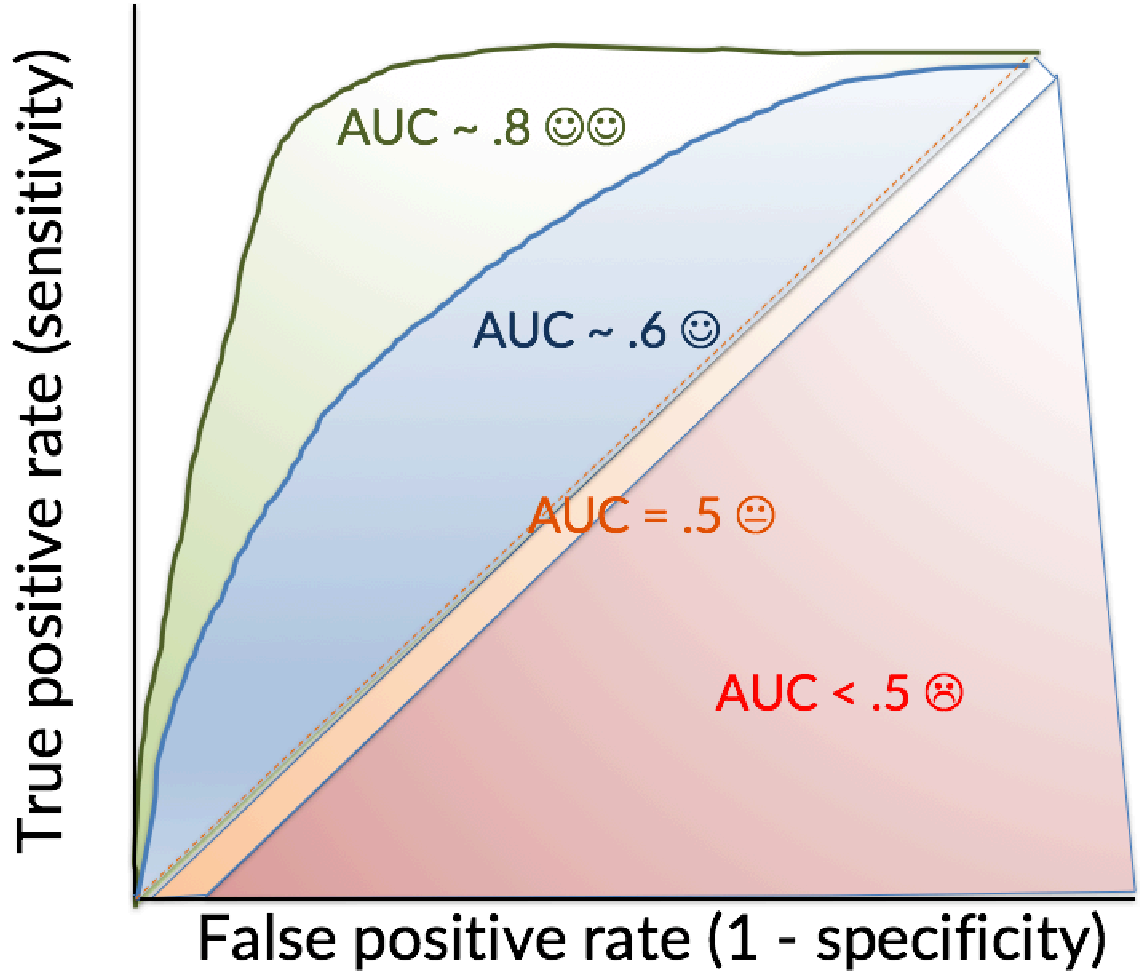
ROC: receiver operator characteristic

- ROC is a plot of the true positive rate (**TPR**) against the false positive rate (**FPR**)
- The plot illustrates the trade off between the 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!)

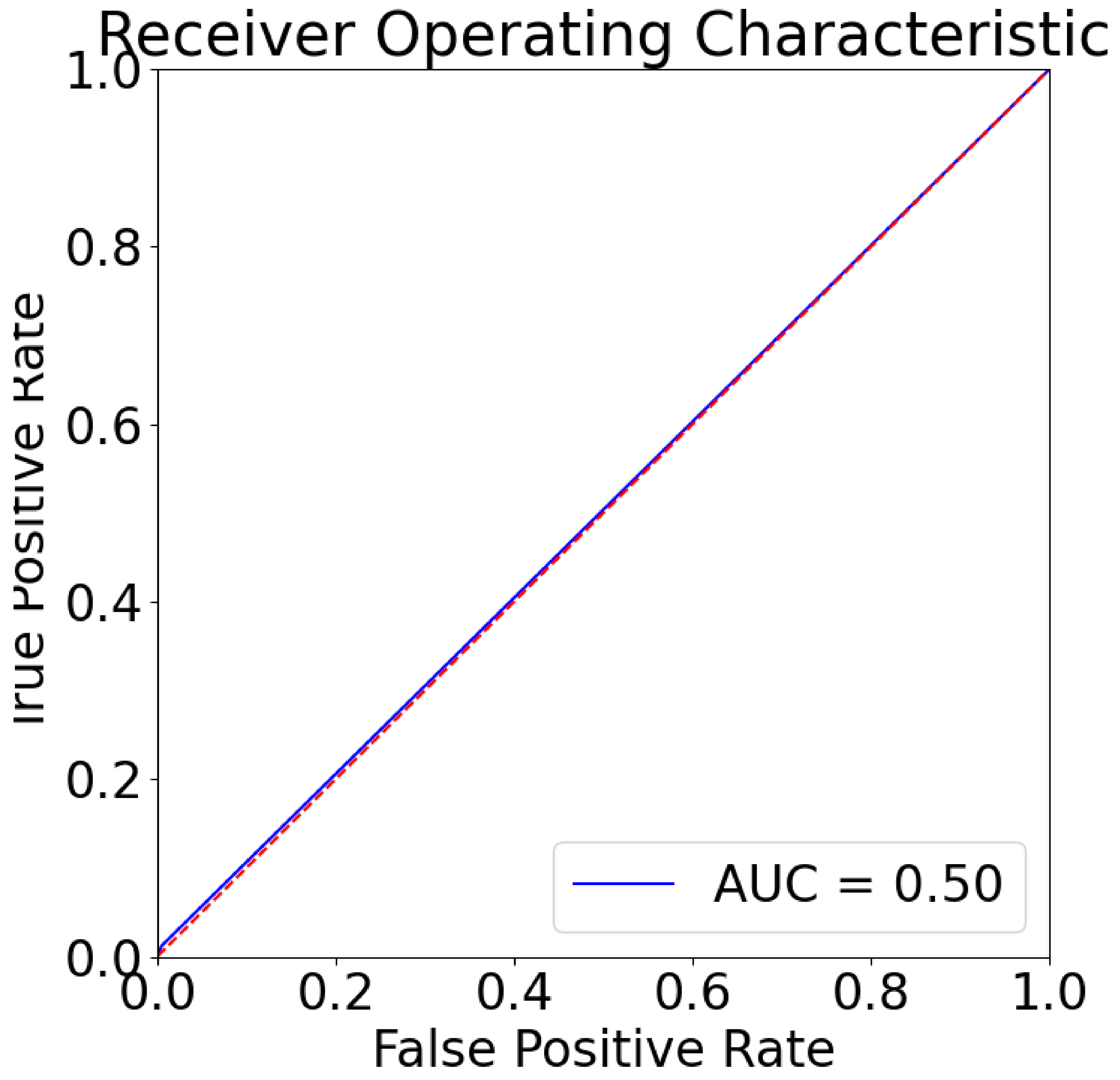


Plot ROC and calculate AUC

- 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

Module completion checklist

Objective	Complete
Implement kNN algorithm on the training data without cross-validation	✔
Identify performance metrics for classification algorithms and evaluate a simple kNN model	✔

Congratulations on completing this module!

You are now ready to try Tasks 9-13 in the Exercise for this topic