Carnegie Mellon University
Heinz College

95-865 Unstructured Data Analytics

Recitation: More on prediction & PyTorch

Slides by George H. Chen and Yubo Li

Outline

- Parameter counting for CNNs (more details on lecture demo)

- More score functions for classification (including a demo)

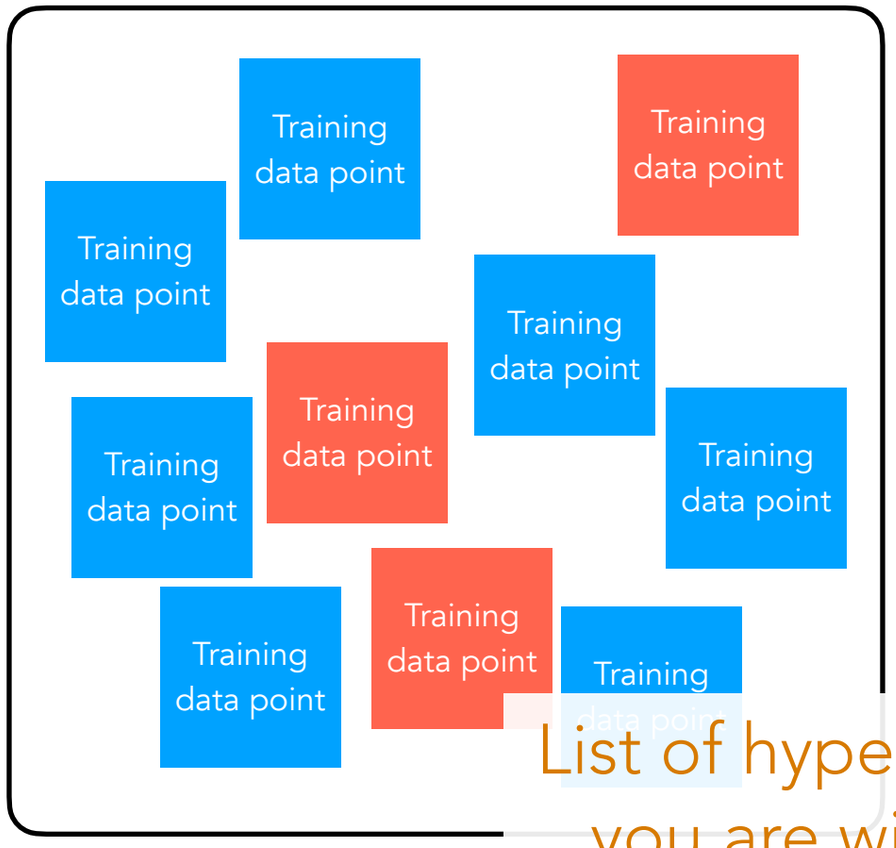
- Using `nn.Module` to create a PyTorch model instead of nn.Sequential (we'll need the nn.Module approach for our time series analysis demos)
  - You'll see that the code involved is sort of like the NumPy code shown in lecture for basic neural nets

Parameter Counting with CNNs

Demo

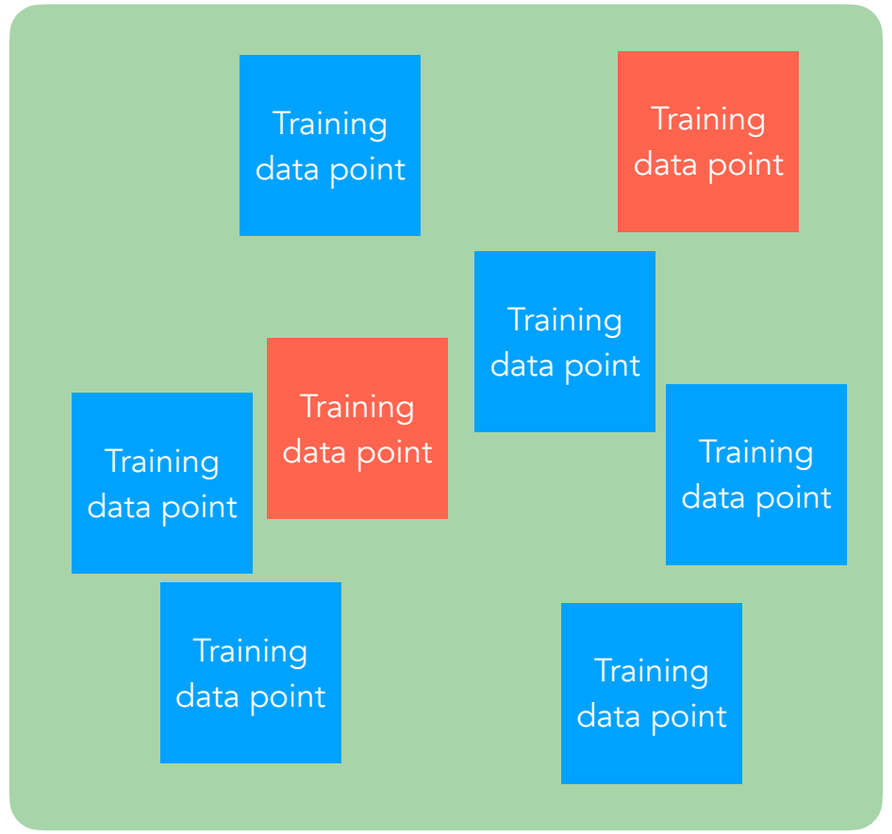
(Flashback)

Training data



Randomly split into two portions (example: 80% / 20%)

"Proper training data"



List of hyperparameters you are willing to try

For θ in Θ

1. Train prediction model on proper training data with hyperparameter setting θ
2. Use a score function to evaluate how well the trained model predicts on validation data

Use whichever value of θ achieves the best score

"Validation data"

There are many score functions possible

Example: fraction of validation points correctly predicted (raw accuracy)

Which score function is used for measuring accuracy matters!

What we already saw in lecture:

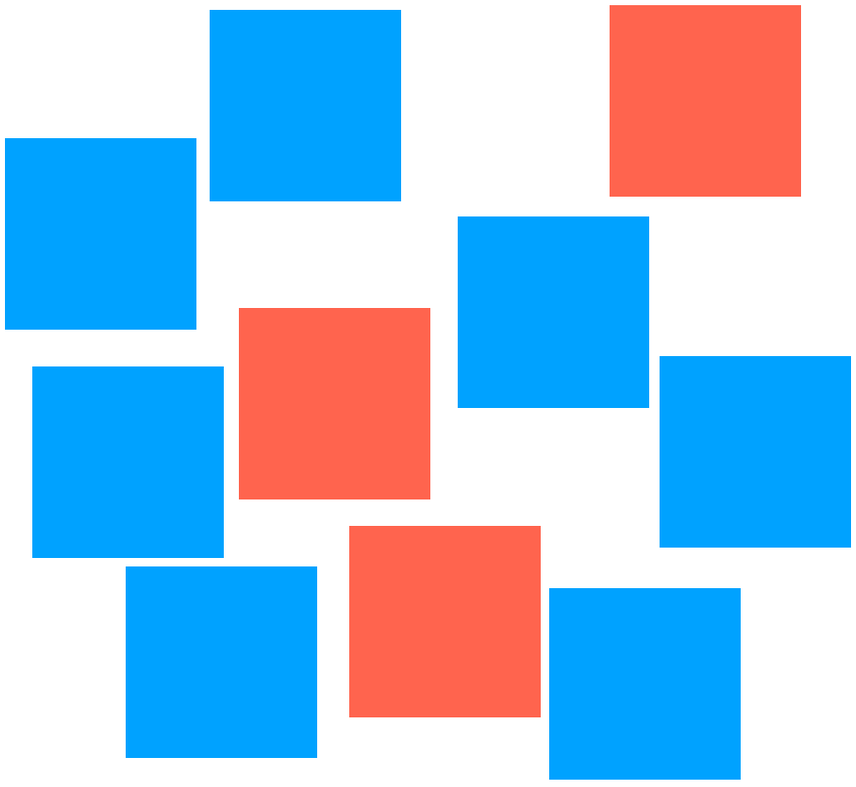
- **Raw accuracy:** fraction of predicted labels that are correct

Score Functions for Classification

In “binary” classification (there are 2 classes such as spam/ham) when 1 class is considered “positive” and the other “negative”:

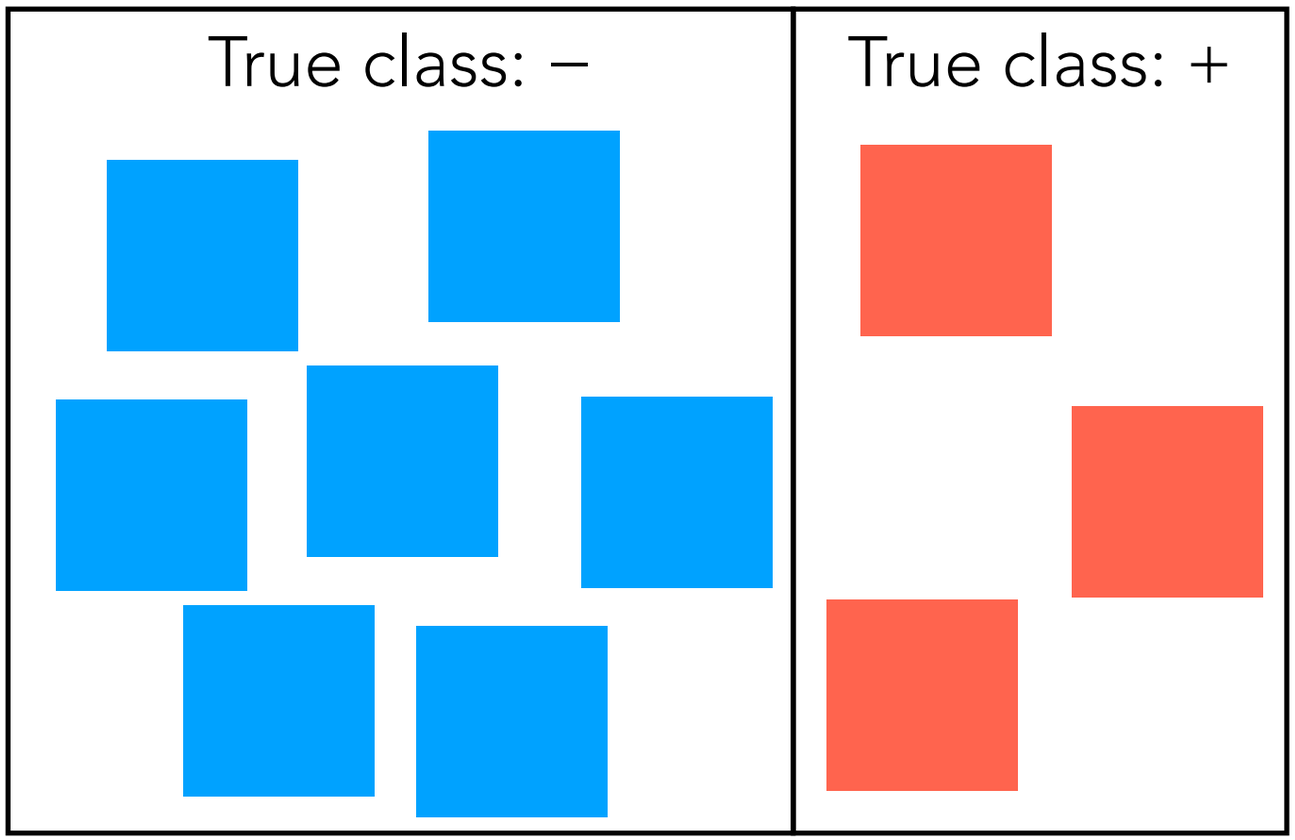
Score Functions for Classification

In “binary” classification (there are 2 classes such as spam/ham) when 1 class is considered “positive” and the other “negative”:



Score Functions for Classification

In “binary” classification (there are 2 classes such as spam/ham) when 1 class is considered “positive” and the other “negative”:

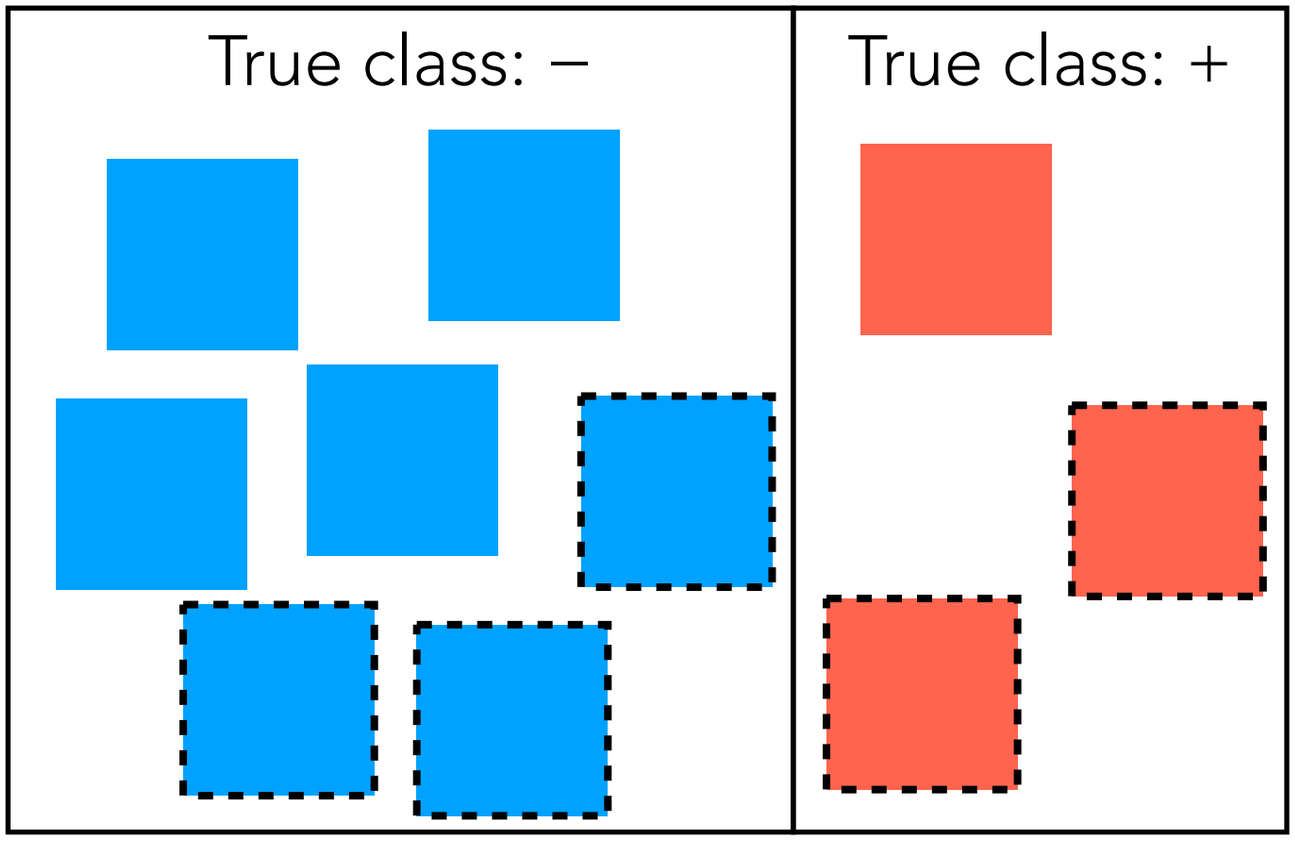


Score Functions for Classification

In "binary" classification (there are 2 classes such as spam/ham) when 1 class is considered "positive" and the other "negative":

Outlined in dotted black: predicted label +

(all other points predicted to be –)



Recall/True Positive Rate: fraction of red points correctly predicted = 2/3

Precision: fraction of dotted points correctly predicted = 2/5

False Positive Rate: fraction of blue points incorrectly predicted = 3/7

F1 score: $\dfrac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$ = 1/2

Some Use Cases

Use **recall** in applications where it's very costly to have *false negatives* (data points that are positive but predicted as negative)

- medical screening (positive class: having a nasty disease)
- fraud detection (positive class: fraud)
- fire alarms (positive class: real fire)

Use **precision** if it's very costly to have *false positives* (data points that are negative but predicted as positive)

- spam filtering (positive class: spam)

Use **F1** when you care about both precision and recall

- Can be particularly helpful when classes are highly imbalanced (e.g., positives are very rare), when raw accuracy alone can be misleading

Generalizing F1 Score to More Than 2 Classes

For each class $c \in \mathcal{C}$: ($\mathcal{C}$: set of possible classes)

- Treat class c as the positive class and compute the F1 score

  Denote the resulting F1 score as: $F_1^{(c)}$

How do we aggregate across the different classes' F1 scores to produce a single number as an overall score?

Option #1 (called "macro" in scikit-learn): report an equally weighted average across classes

$$F_1^{\text{equally weighted}} = \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} F_1^{(c)}$$

Option #2 (called "weighted" in scikit-learn): weight each class by how often it appears in the data that we're evaluating the F1 score for

$$F_1^{\text{weighted}} = \sum_{c \in \mathcal{C}} [\text{fraction of points in class } c] \times F_1^{(c)}$$

“Receiver Operating Characteristic” (ROC) Curves

Probability Thresholding

Many classifiers output the probability of each class for any test feature vector ***x***

(MNIST: for any test image, we predict probabilities for all 10 digits)

To get final predicted class of test feature vector ***x***:
pick whichever class has the highest probability

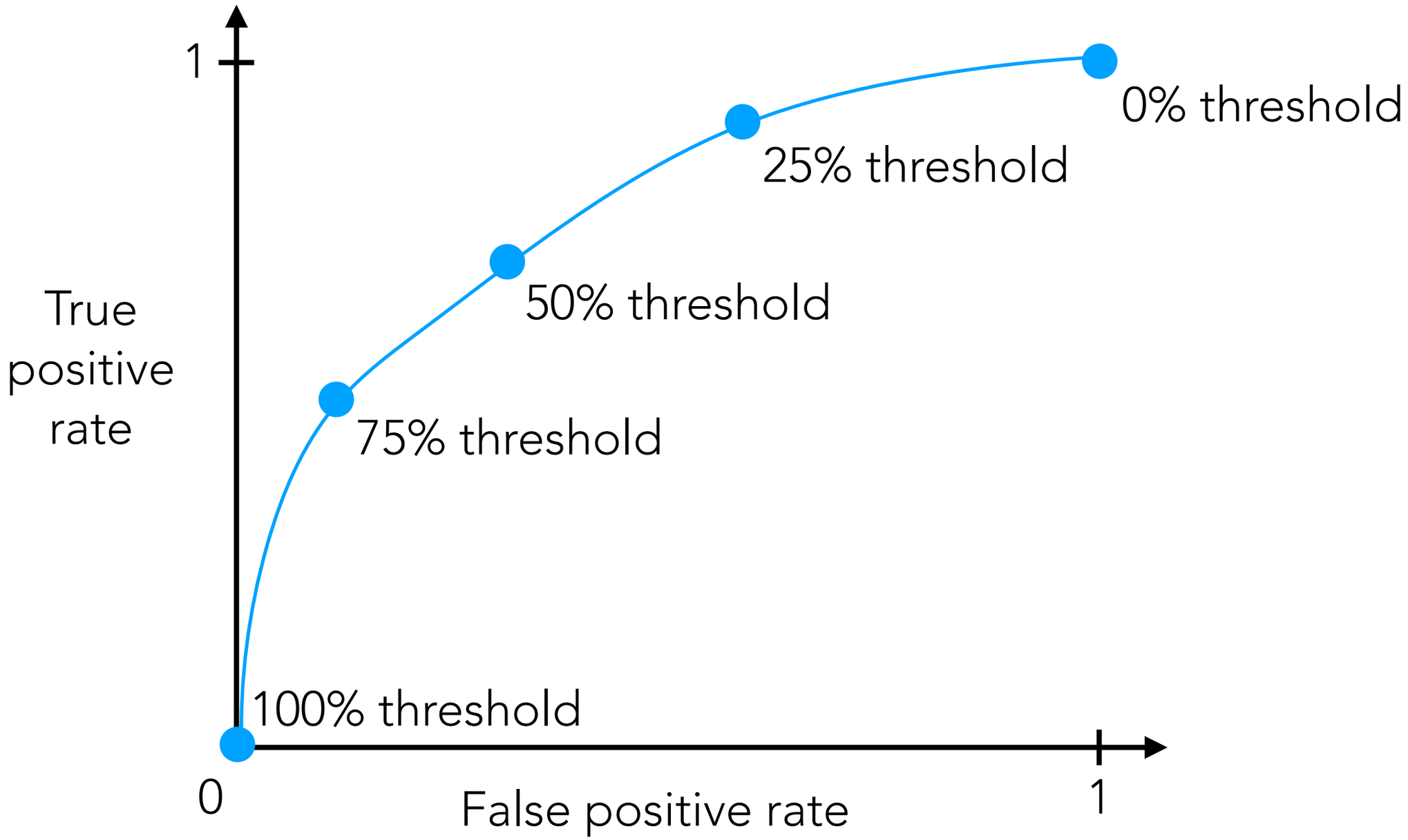
When there are 2 classes positive and negative

Predict positive if P(positive | test feature vector ***x***) ≥ 0.5

Predict negative otherwise

We can vary this 50% threshold!

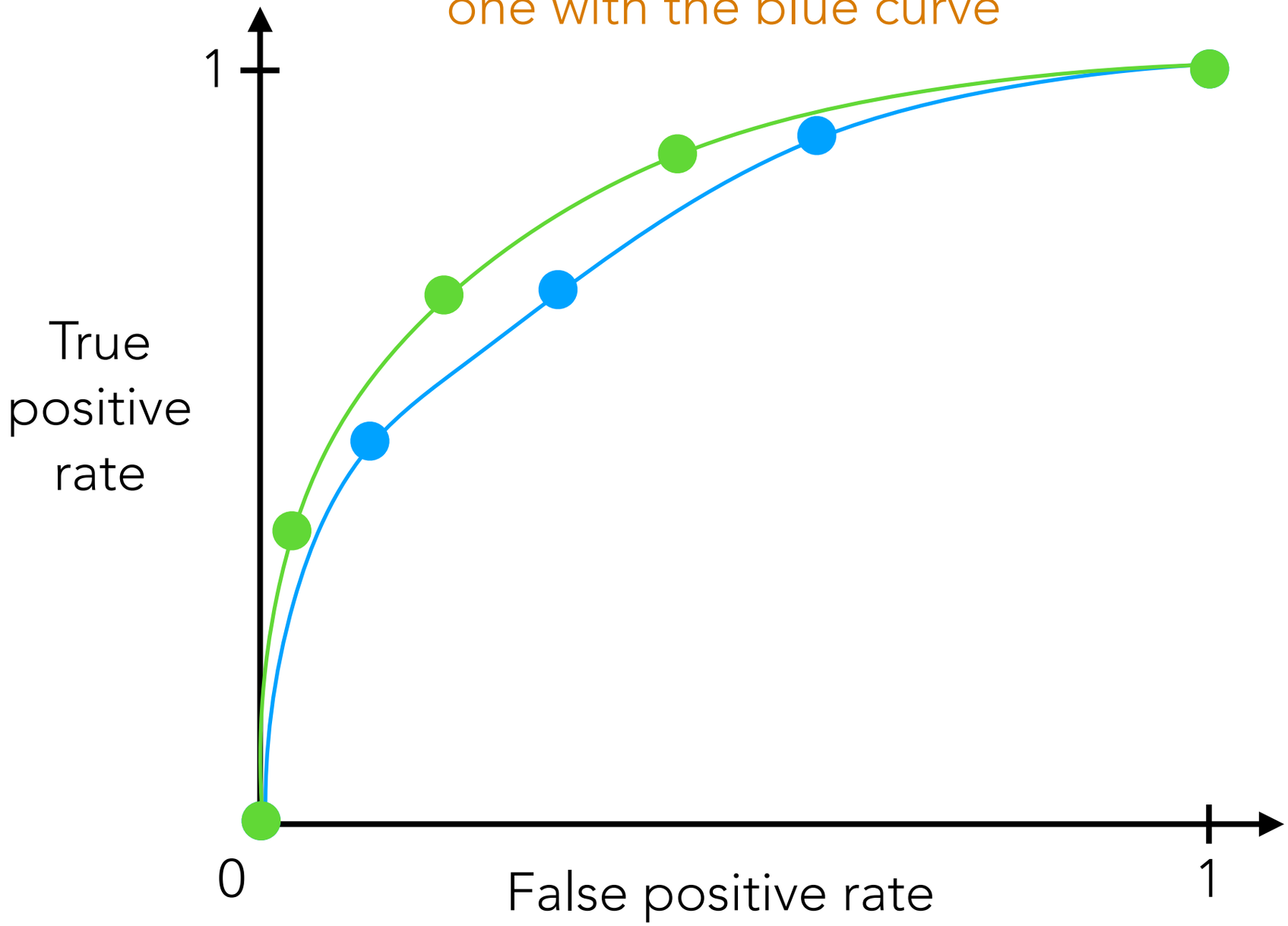
Binary Classification: ROC Curves



TPR and FPR are computed using test data

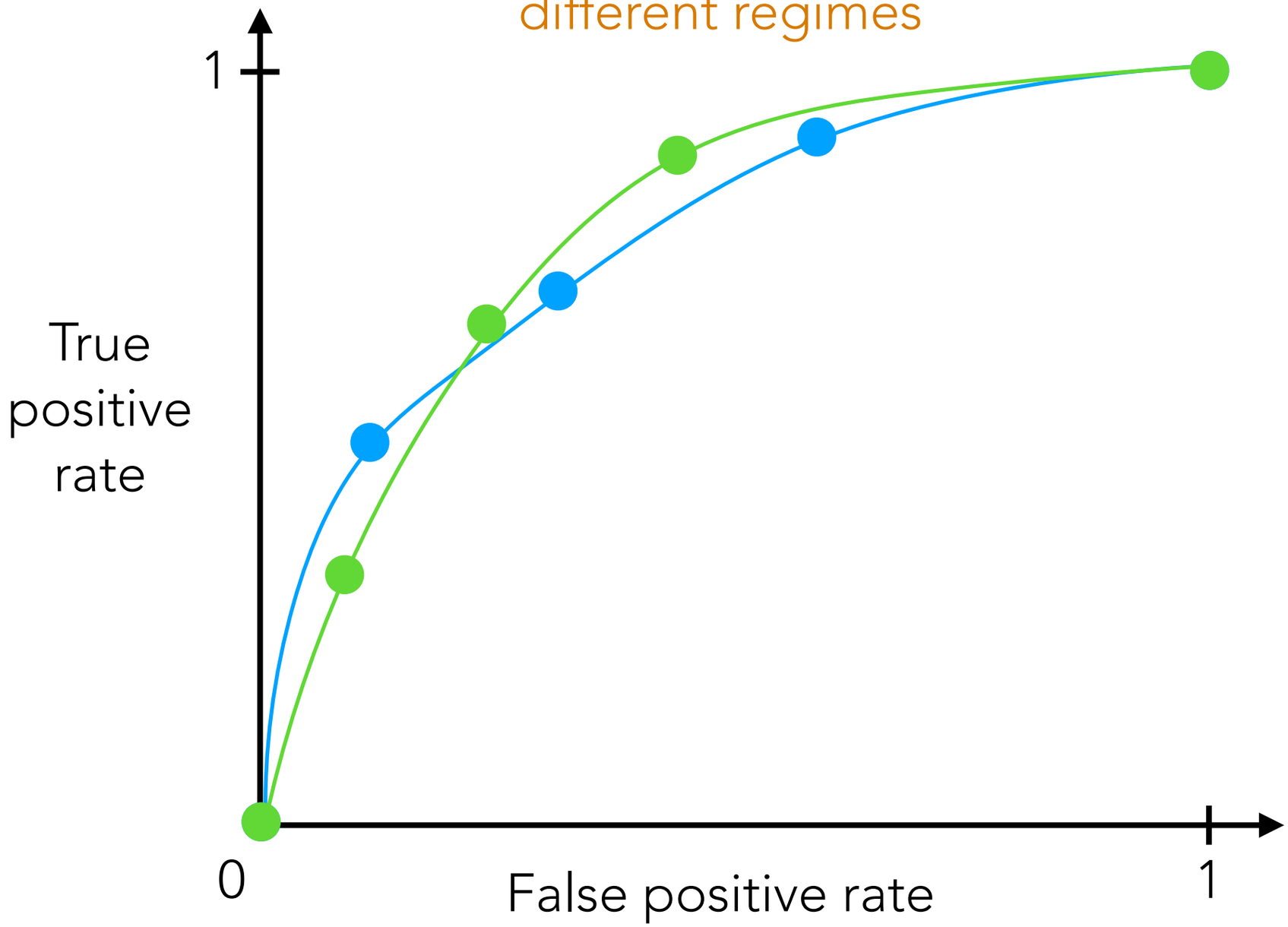
Binary Classification: ROC Curves

A classifier with the green curve is better than the one with the blue curve

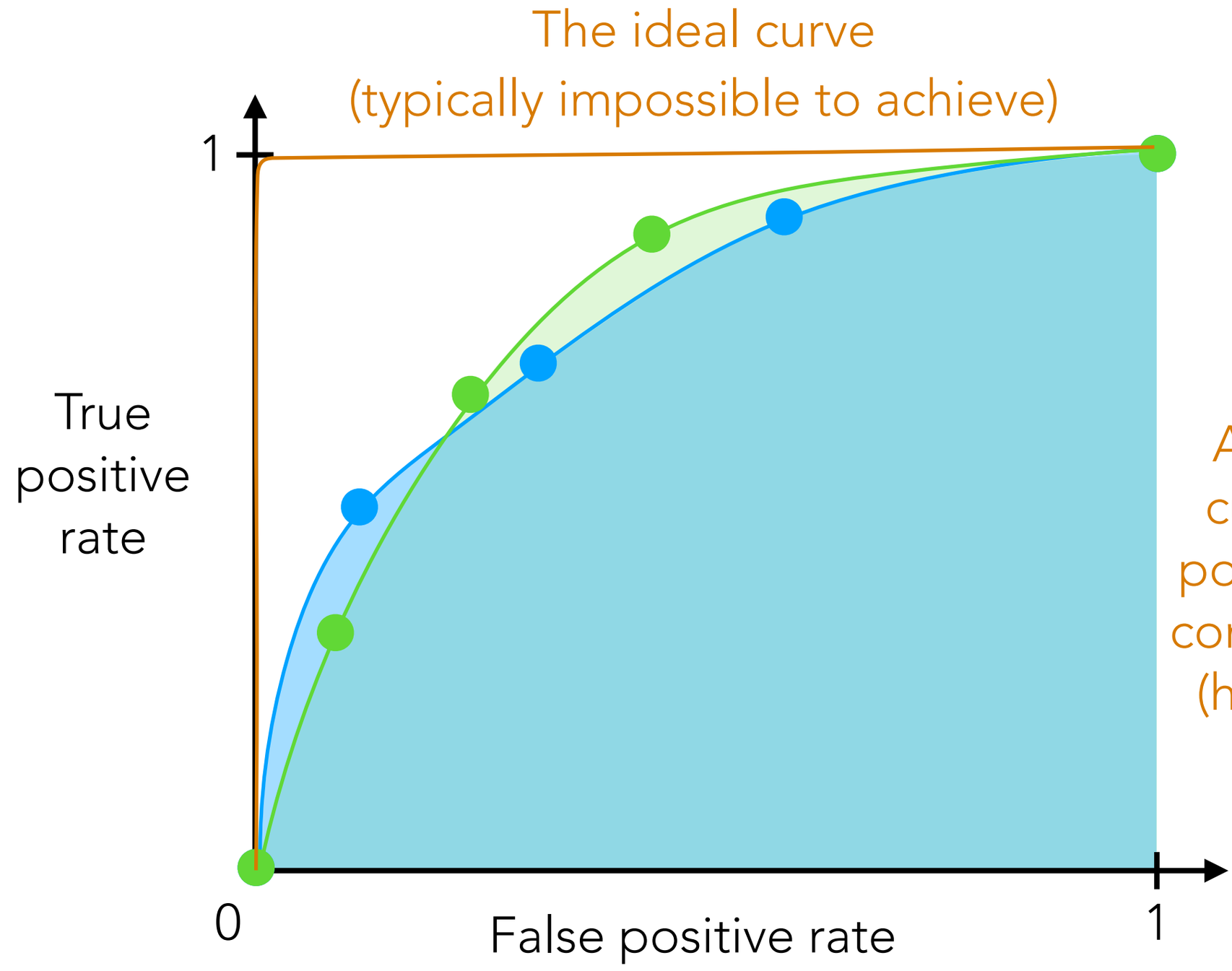


Binary Classification: ROC Curves

It's possible that different models are better in different regimes



Binary Classification: ROC Curves



Area under the curve (AUC) is a popular metric for comparing models (higher is better)

Binary Classification: ROC Curves

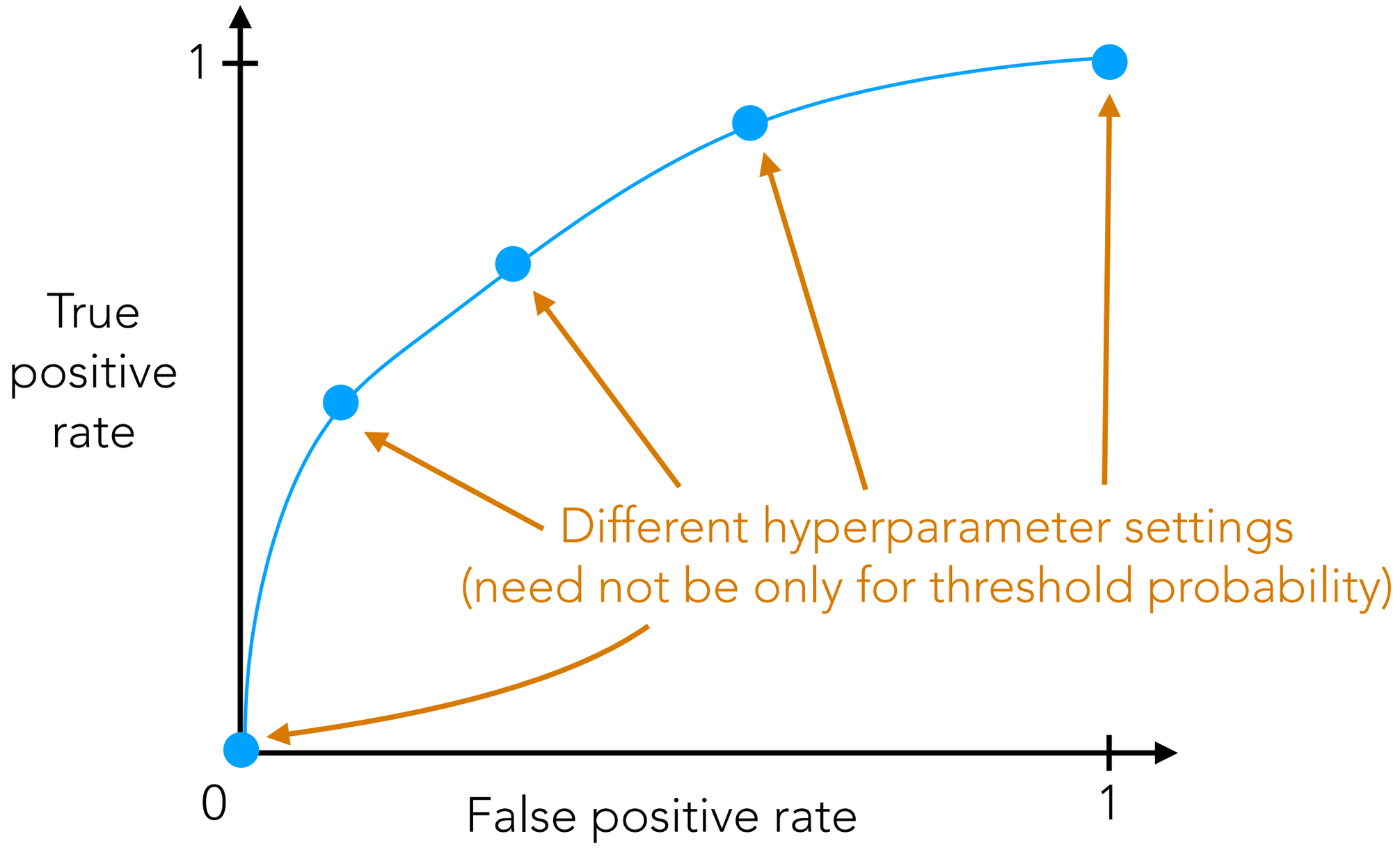
What we just saw:

- For a classifier that we can set the threshold probability to different values, we can plot an ROC curve
- True positive rate (TPR) and false positive rate (FPR) are evaluated on test data

Other variants are possible:

- Plot precision vs recall instead of TPR vs FPR
- Can actually plot ROC/precision-recall curves sweeping over hyperparameters *aside from threshold probability*!
- For ROC/precision-recall, rather than evaluating on test data, can evaluate on validation data during training *to help choose hyperparameters*

Binary Classification: ROC Curves



Can also be computed on validation data instead of test data!

Score Functions, ROC Curves

Demo

(Flashback) Neural Net as Function Approximation

Given `input`, learn a computer program that computes `output`

Multinomial logistic regression:

```
linear = np.dot(input, W.T) + b
output = softmax(linear)
```

Today's recitation coverage will reveal where this kind of code shows up in PyTorch

Multilayer perceptron:

```
linear1 = np.dot(input, W1.T) + b1
relu = np.maximum(0, linear1)  # ReLU
linear2 = np.dot(relu, W2.T) + b2
output = softmax(linear2)
```

Learning a neural net: learning a simple computer program that maps inputs (raw feature vectors) to outputs (predictions)

More layers ⇒ computer program has more lines of code

Constructing a neural net in PyTorch using `nn.Module` (instead of `nn.Sequential`)

(we'll need this level of detail in this upcoming Monday's lecture demo)

Constructing PyTorch Models with `nn.Module`

What you've already seen previously:

```
deeper_model = nn.Sequential(nn.Flatten(),
                             nn.Linear(in_features=784, out_features=512),
                             nn.ReLU(),
                             nn.Linear(in_features=512, out_features=10))
```

Another way to write this:

```
class DeeperModel(nn.Module):
    def __init__(self, num_in_features, num_intermediate_features, num_out_features):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(num_in_features, num_intermediate_features)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(num_intermediate_features, num_out_features)

    def forward(self, inputs):
        flatten_output = self.flatten(inputs)
        linear1_output = self.linear1(flatten_output)
        relu_output = self.relu(linear1_output)
        linear2_output = self.linear2(relu_output)
        return linear2_output

deeper_model = DeeperModel(784, 512, 10)
```

This is like the NumPy code presented earlier for the multilayer perceptron (softmax is excluded as it's part of the loss calculation)