Decision Trees

Classifying with Histograms

Let's imagine that we want to classify observations based on a binary dependent variable, y.

- Two possible outcomes
- We classify each observation based on which outcome is most likely

$$p(y|x) \geq .5 \Rightarrow \hat{y} = 1$$

$$p(y|x) < .5 \Rightarrow \hat{y} = 0$$

Classifying with Histograms

Let's draw some classifiers on the white board.

- Divide the data in half for each of two x variables
- Each separate bin can then be classified separately
- We can increase the granularity of our classifier by adding more breaks to each x variable

Classifying with Histograms

Is there a more efficient way?

- Imagine we only divide variables in half
- How many discrete bins of data would exist if we looked at each of 16 variables in this way?

Classifying with Histograms

Is there a more efficient way?

- Imagine we only divide variables in half
- How many discrete bins of data would exist if we looked at each of 16 variables in this way?
  - $2^{16} = 65,536$ possible bins
  - We would then need 65,536 observations at the **minimum** to obtain **any** information about each cell
- This is **not** efficient

Instead, we use Decision Trees

But first, we need to learn about entropy...

Entropy is a measure of uncertainty (or information) about the world, or, more specifically, uncertainty about the true value of an outcome in a given model.

Lower entropy: less uncertainty

Higher entropy: greater uncertainty

Measuring Entropy

Entropy can be calculated using the following equation

$$H(x) = -\sum_{i=0}^{n} p(x_i)\ ln\ p(x_i)$$

This is *Nat* Entropy (Shannon entropy is calculated using log_2, and Hartley entropy uses log_{10}, but it doesn't actually matter which we use so long as we are **consistent**)

Measuring Entropy

1. More possible outcomes leads to higher entropy
2. Greater uncertainty among outcomes leads to higher entropy

The goal of all of our predictive measures will be to reduce entropy (or maximize information gain) at each step in our model

Remember: information gain should be relative to the universe, not our sample

Exercise

Write a function to estimate the Nat Entropy of a set of outcomes, given the observed probability for each outcome in an arbitrary set. Use your function to answer:

1. If the probability of 5 outcomes are .2, .3, .1, .1, and .3, then what is the entropy of the system?
2. If the probabilty of each of 5 outcomes is .2, then what is the entropy of the system?
3. If the probability of each of 10 outcomes is .1, then what is the entropy of the system?

Exercise Answer

```
import numpy as np

def natEnt(listP):
    entropy = 0
    n = len(listP)
    for i in range(n):
        entropy += listP[i] * np.log(listP[i])
    entropy *= -1
    return entropy

print(natEnt([.2,.3,.1,.1,.3]))
print(natEnt([.2]*5))
print(natEnt([.1]*10))
```

Using Entropy

We need to determine how we can reduce the entropy of our system by dividing data.

1. Choose the most informative Variable
2. Choose how to best divide the selected variable in order to gain information
3. Repeat until we reach a stopping point
    - We need to predetermine a stopping rule

Using Entropy

In order to choose the most informative variable, we need to first determine how informative each variable is

- Search across each variable for the optimal decision point
- Determine the information gain from that variable and decision point
- Compare the information gain across the available variables

Information Gain

We can define information gain from a binary split of our data as follows:

$$IG = H_0(x) - H_1(x)$$

Where H_0 is the original entropy, and H_1 is the entropy after the split.

Information Gain

We can calculate H_1 as

$$H_1(x) = \omega_1 \cdot H_{11}(x) + \omega_2 \cdot H_{12}(x)$$

- ω_1 and ω_2: the fraction of elements in each respective child node relative to the parent node (should add up to 1)
- H_{11} and H_{12}: the entropy of each child node

Where is the Cutoff?

Where do we draw the line when dividing observations based on a given variable?

Where is the Cutoff?

We need an algorithm that will **search** across possible cutoffs for our variable, and return the most advantageous split.

- Gradient Descent is frequently used on continuous variables
- For binary variables, we can simply separate the groups/classes
- For count variables, determine which cutoff will generate the greatest gain

Implementing a Decision Tree

We will be using the `sklearn` library today. It is the most robust machine learning library available, and allows us to implement many kinds of tests and algorithms. sci-kit learn documentation

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
```

This is all the extra code that we will need to start using our new Decision Tree Classifiers.

Fitting Data with Sci-kit Learn

```
# Our import statements for this problem
import pandas as pd
import numpy as np
import patsy as pt

from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
```

Fitting Data with Sci-kit Learn

```
# The code to implement a decision tree
data = pd.read_csv(
        "https://github.com/dustywhite7/Econ8310/"
  + "raw/master/DataSets/titanic.csv")


model = DecisionTreeClassifier()

# As usual, Patsy makes data prep easier
y, x = pt.dmatrices("Survived ~ -1 + Sex + Age
                + SibSp + Pclass", data=data)

x, xt, y, yt = train_test_split(x, y,
                test_size=0.33, random_state=42)

res = model.fit(x,y)

print("\n\nIn-sample accuracy: %s%%\n\n"
 % str(round(100*accuracy_score(y, model.predict(x)), 2)))
```

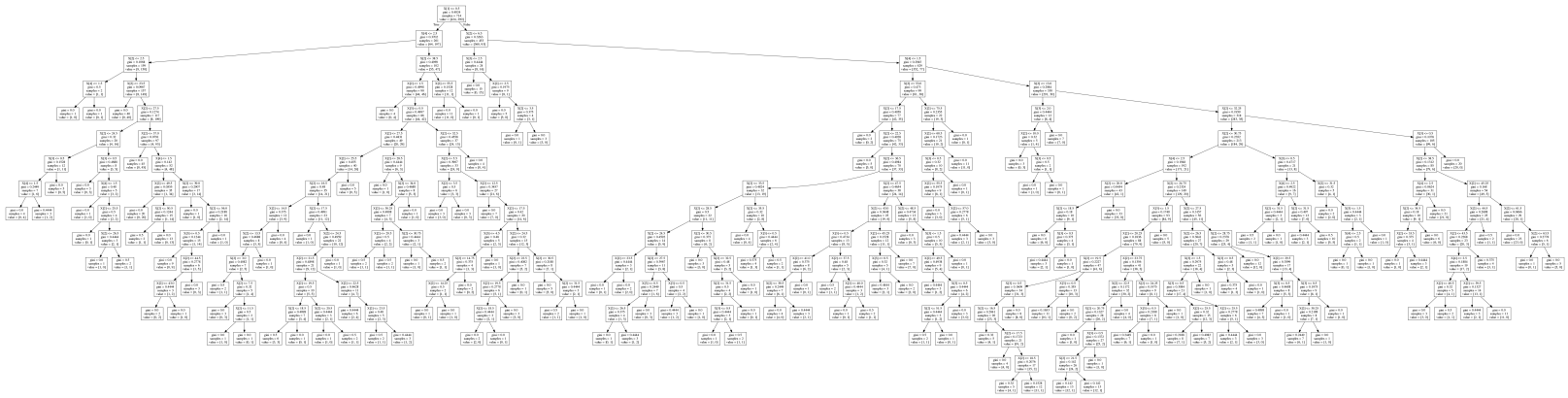
How does it do?

Using the Titanic dataset, and predicting survival with sex, age, siblings, and class (how fancy the passenger was traveling) results in the following printout:

In-sample accuracy: 94.35%

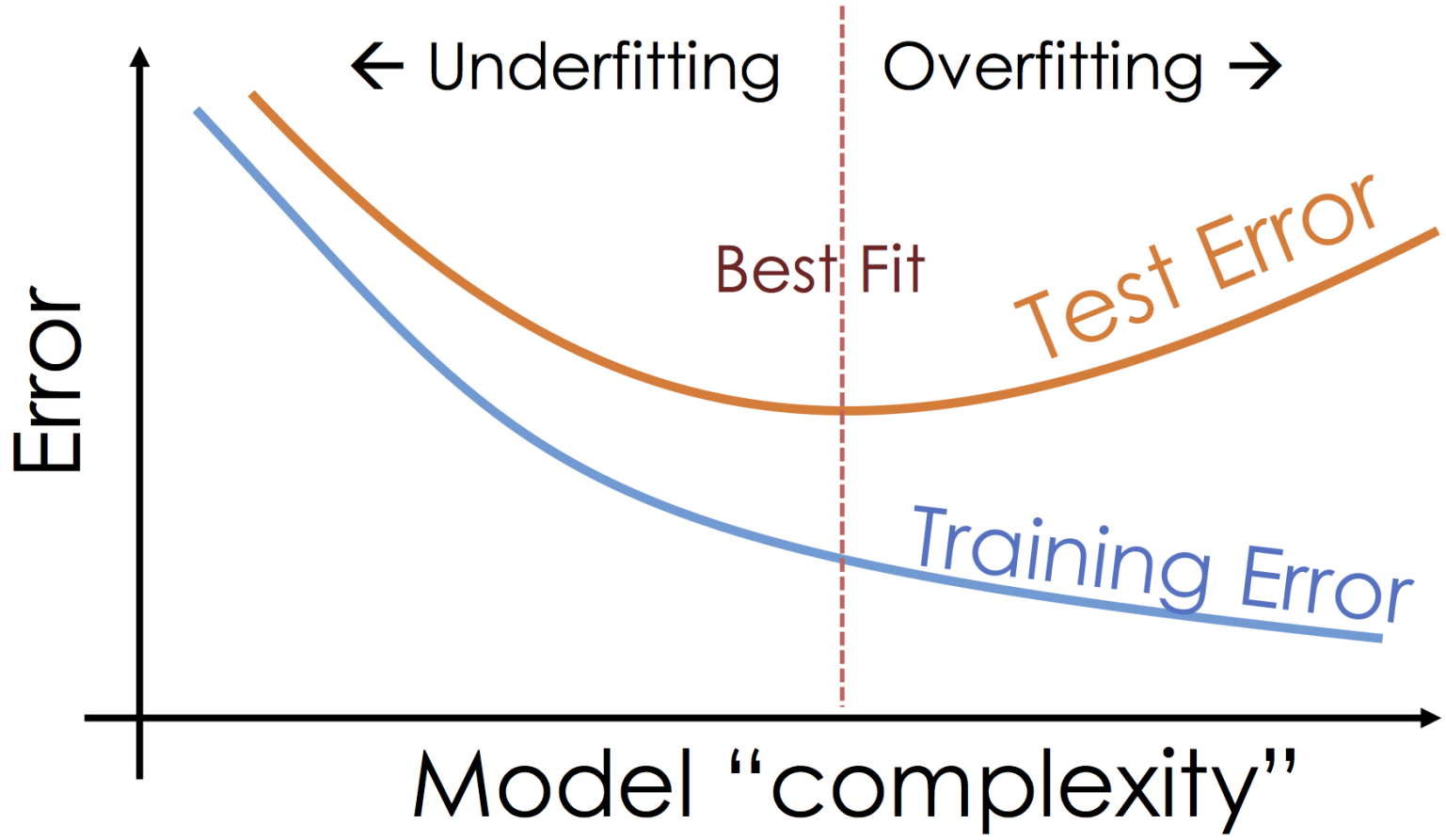
For being easy to implement, that is a pretty good prediction!

Visualizing the Model



Didn't you say that this would be **human** interpretable?

Bias and Variance



Bias and Variance

Bias: When we predict using a model, the bias is the difference between our predicted outcome and the true outcome

Variance: When we predict one observation using various models, the variance is the dispersion of outcomes for that single observation

- Do all models tend to say the same thing? That would indicate low variance

Bias and Variance

Typically, both bias and variance can be reduced by training models on a larger data set. This is unsurprising, since more information about an outcome should enable us to make better decisions regarding that outcome

- These models are designed to converge on truth
- Assuming that we have **representative** data

Bias and Variance

While more data is better for both, bias and variance are opposites when it comes to model complexity

- Bias declines as complexity increases
- Variance increases as complexity increases

Our job is to identify the sweet spot where the **combined** error is lowest

Overfitting

Overfitting is when we allow our model to overemphasize the random variation between observations in our sample. This practice will lead to higher in-sample accuracy (frequently we will even achieve 100% accuracy in-sample!), but reduce our accuracy out of sample.

Underfitting

Underfitting is when we fail to take advantage of available information, and induce higher errors both in- and out-of-sample

- If we don't make use of our data, then we can't make quality decisions

Overfitting in Decision Trees

Remember our crazy decision tree? We want a model to be readable for a human, we should probably try to keep the model simpler. This will also improve out-of-sample accuracy.

```
print("\n\nIn-sample accuracy: %s%%\n\n"
 % str(round(100*accuracy_score(y, model.predict(x)), 2)))
print("\n\nOut-of-sample accuracy: %s%%\n\n"
%str(round(100*accuracy_score(yt, model.predict(xt)), 2)))
```

In-sample accuracy: 94.35%

Out-of-sample accuracy: 75.85%

Performance is much worse out of sample

Overfitting Decision Trees

Let's restrict our tree to only 5 levels, and see what happens. We only need to modify one line of our code:

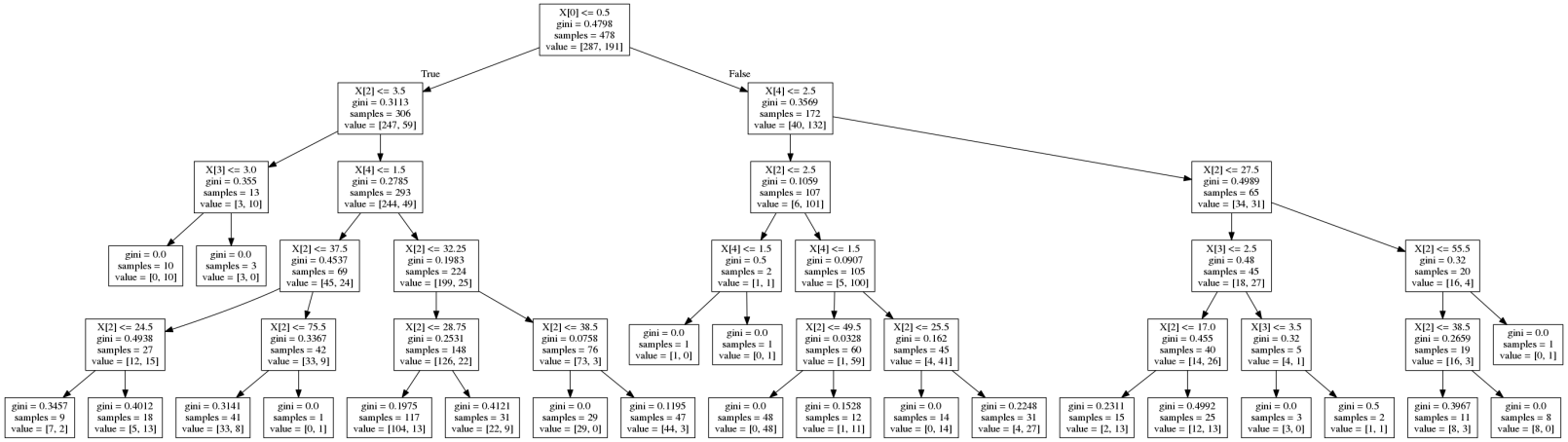
```
model = DecisionTreeClassifier(max_depth=5)
```

In-sample accuracy: 86.82%

Out-of-sample accuracy: 77.12%

By simplifying, we actually do **better** out of sample, even though training accuracy suffers!

The Tree


X[0] <= 0.5
gini = 0.4798
samples = 478
value = [287, 191]
True
False
X[2] <= 3.5
gini = 0.3113
samples = 306
value = [247, 59]
X[4] <= 2.5
gini = 0.3569
samples = 172
value = [40, 132]
X[3] <= 3.0
gini = 0.355
samples = 13
value = [3, 10]
X[4] <= 1.5
gini = 0.2785
samples = 293
value = [244, 49]
X[2] <= 2.5
gini = 0.1059
samples = 107
value = [6, 101]
X[2] <= 27.5
gini = 0.4989
samples = 65
value = [34, 31]
gini = 0.0
samples = 10
value = [0, 10]
gini = 0.0
samples = 3
value = [3, 0]
X[2] <= 37.5
gini = 0.4537
samples = 69
value = [45, 24]
X[2] <= 32.25
gini = 0.1983
samples = 224
value = [199, 25]
X[4] <= 1.5
gini = 0.5
samples = 2
value = [1, 1]
X[4] <= 1.5
gini = 0.0907
samples = 105
value = [5, 100]
X[3] <= 2.5
gini = 0.48
samples = 45
value = [18, 27]
X[2] <= 55.5
gini = 0.32
samples = 20
value = [16, 4]
X[2] <= 24.5
gini = 0.4938
samples = 27
value = [12, 15]
X[2] <= 75.5
gini = 0.3367
samples = 42
value = [33, 9]
X[2] <= 28.75
gini = 0.2531
samples = 148
value = [126, 22]
X[2] <= 38.5
gini = 0.0758
samples = 76
value = [73, 3]
gini = 0.0
samples = 1
value = [1, 0]
gini = 0.0
samples = 1
value = [0, 1]
X[2] <= 49.5
gini = 0.0328
samples = 60
value = [1, 59]
X[2] <= 25.5
gini = 0.162
samples = 45
value = [4, 41]
X[2] <= 17.0
gini = 0.455
samples = 40
value = [14, 26]
X[3] <= 3.5
gini = 0.32
samples = 5
value = [4, 1]
X[2] <= 38.5
gini = 0.2659
samples = 19
value = [16, 3]
gini = 0.0
samples = 1
value = [0, 1]
gini = 0.3457
samples = 9
value = [7, 2]
gini = 0.4012
samples = 18
value = [5, 13]
gini = 0.3141
samples = 41
value = [33, 8]
gini = 0.0
samples = 1
value = [0, 1]
gini = 0.1975
samples = 117
value = [104, 13]
gini = 0.4121
samples = 31
value = [22, 9]
gini = 0.0
samples = 29
value = [29, 0]
gini = 0.1195
samples = 47
value = [44, 3]
gini = 0.0
samples = 48
value = [0, 48]
gini = 0.1528
samples = 12
value = [1, 11]
gini = 0.0
samples = 14
value = [0, 14]
gini = 0.2248
samples = 31
value = [4, 27]
gini = 0.2311
samples = 15
value = [2, 13]
gini = 0.4992
samples = 25
value = [12, 13]
gini = 0.0
samples = 3
value = [3, 0]
gini = 0.5
samples = 2
value = [1, 1]
gini = 0.3967
samples = 11
value = [8, 3]
gini = 0.0
samples = 8
value = [8, 0]

Overfitting Decision Trees

Let's make one more change, and restrict our tree to leaves with 10 or more observations:

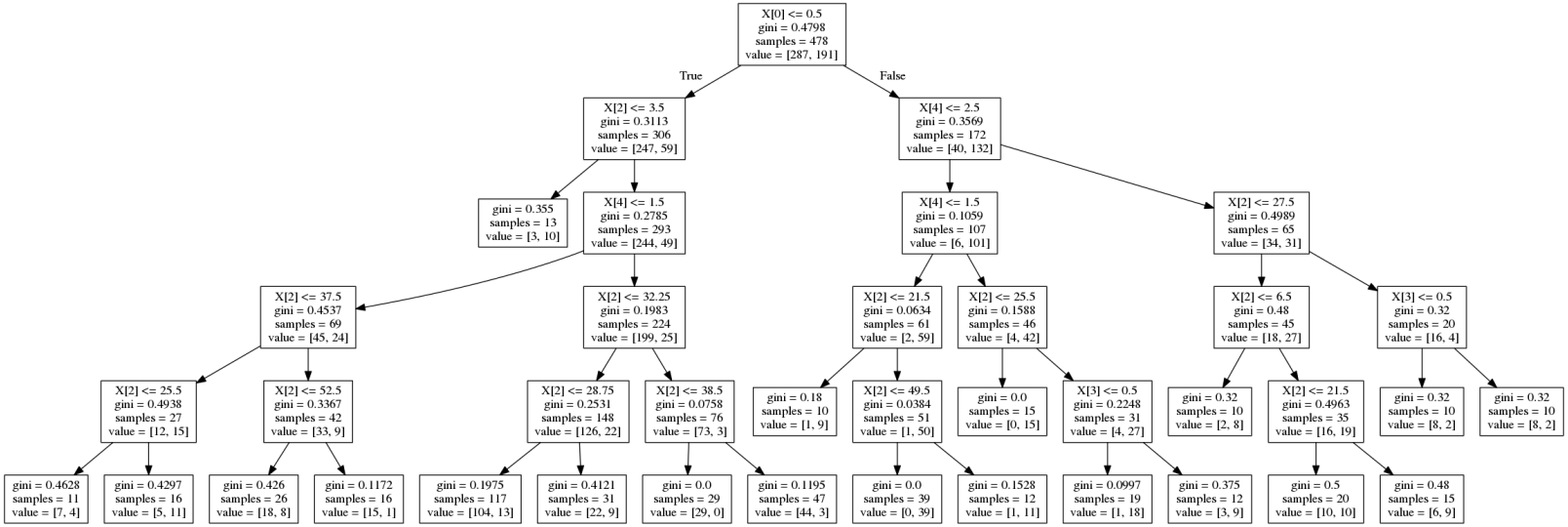
```
model = DecisionTreeClassifier(max_depth=5,
        min_samples_leaf=10)
```

In-sample accuracy: 84.52%

Out-of-sample accuracy: 78.39%

Again, we simplify and do **better** out of sample!

The Tree



It's small on the slide, but it is now a reasonably readable algorithm. At most, you have to ask 5 questions to arrive at an answer.

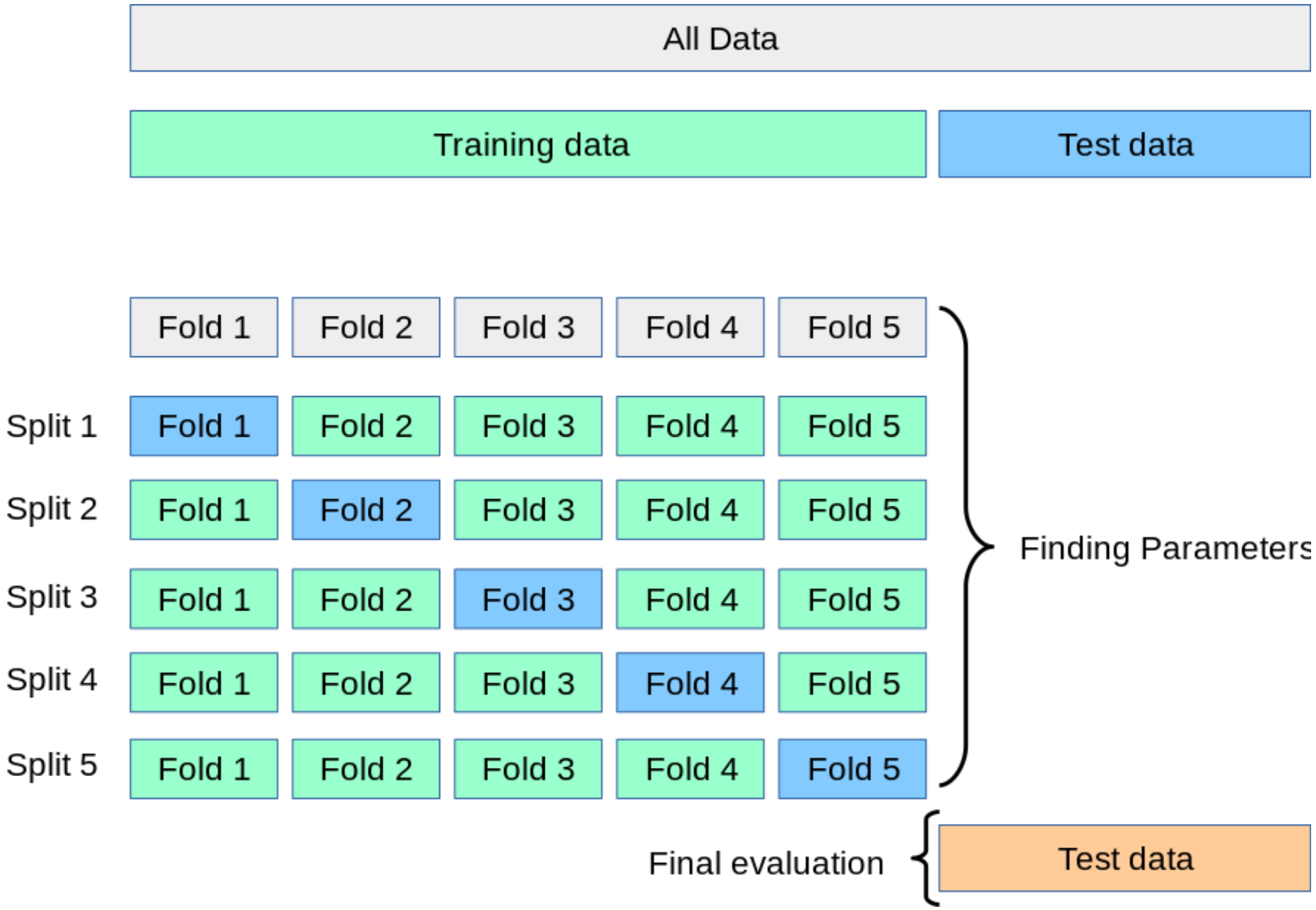
A Note on Cross-Validation

In order to maximize the information that we have about our model's performance, we will often test the model on many draws of our existing data.

This is called **cross-validation**. We resample our training and testing data k times, and compare the performance of the models.

If the models perform more or less equally well, then we can treat our model as well-specified. If not, then we know performance was sample-dependent.

Cross-Validation Diagram



Cross-Validation Code

```
from sklearn.model_selection import KFold

# If we have imported data and created x, y already:
kf = KFold(n_splits=10) # 10 "Folds"

models = [] # We will store our models here

for train, test in kf.split(x): # Iterate over folds
  model = model.fit(x[train], y[train]) # Fit model
  accuracy = accuracy_score(y[test],    # Store accuracy
    model.predict(x[test]))
  print("Accuracy: ", accuracy_score(y[test],
    model.predict(x[test])))            # Print results
  models.append([model, accuracy])      # Store it all

print("Mean Model Accuracy: ",          # Print aggregate
  np.mean([model[1] for model in models]))
```

One More Note

After we complete our cross-validation, we do not use the cross-validation models. When we are satisfied with the results of the cross-validation, we

- Recombine all training data
- Train the model on all training data
- Test the model on withheld testing data (should not have been used at all in cross-validation)
- If results are still positive, implement model!

Lab time!