Lecture 5: Classification

Classification

In regression, we try to predict a real-valued (quantitative) variable.

Examples:

- Predict house prices based on attributes
- Predict credit score rating based on income, balance, gender, education, etc...

In classification, we try to predict a categorical (qualitative) variable.

Examples:

- Predict whether a bank should issue a person a credit card (yes/no)
- Predict a hospital patient's diagnosis (stroke, heart attack,...) based on symptoms.

Classification

We assume a dataset with n samples $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, where x_i are attributes or features and y_i are categorical variables that you want to predict.

Our Goal is to develop a rule for predicting the categorical variable y based on the features x.

For example, can we predict whether or not a student will be admitted to a graduate program based on their undergraduate performance (GPA, GRE score, prestige of student's undergraduate university)?

Or, for example, the post office uses classification of hand-written zip codes to automatically sort mail. The digits of the zip code are photographed, the picture serves as your data vector, and the picture is then assigned to one of the *classes:* $0, 1, 2, \ldots, 9$.

K–Nearest Neighbors (k-NN)

Idea: To decide the class of a given point, find the k nearest neighbors of that point, and let them "vote" on the class. That is, we assign the class to the sample that is most common among its k nearest neighbors.

Considerations:

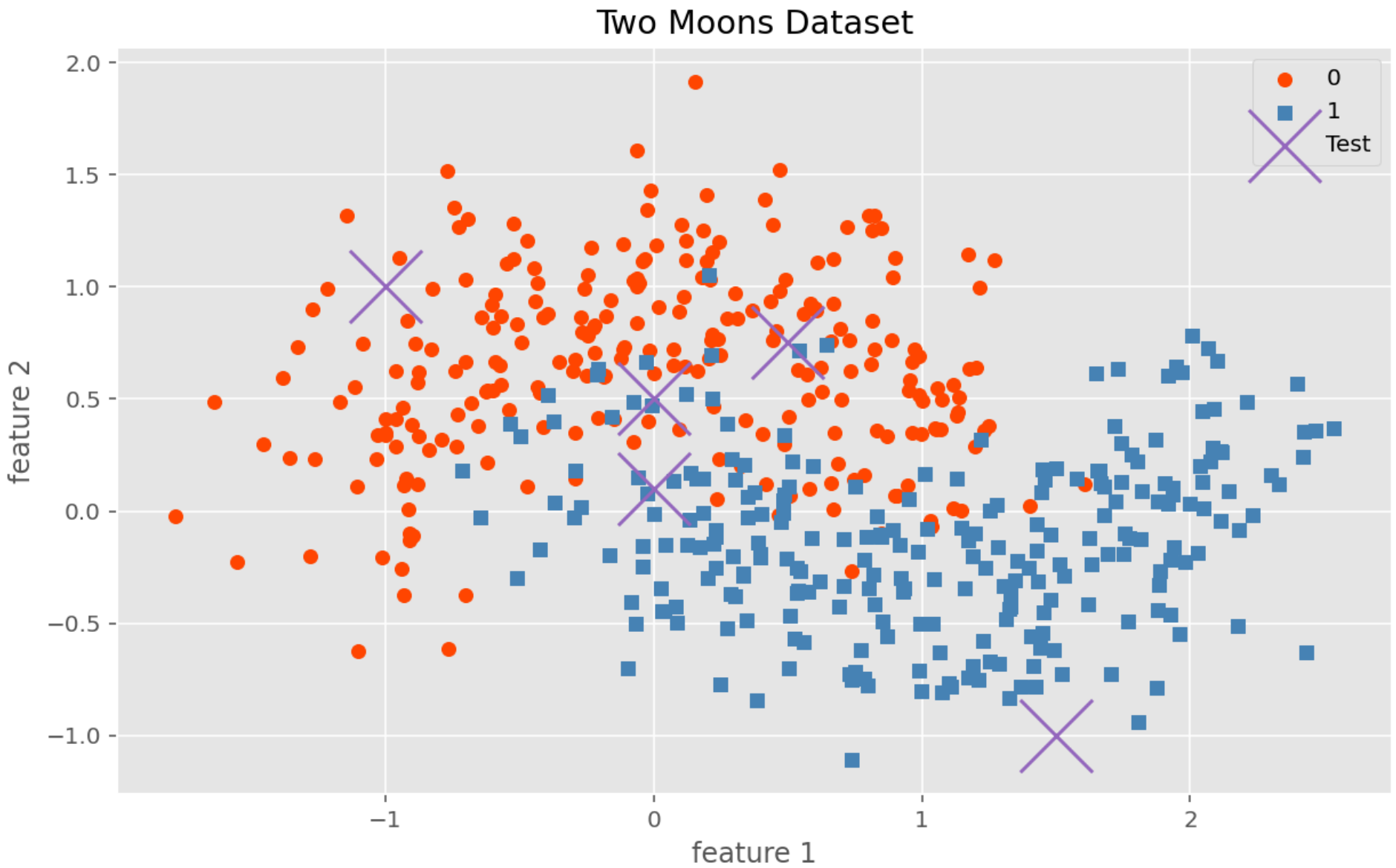
- We must pick k, the number of voting neighbors (typically a small number, say k=10)
- 'Nearest' means closest in distance, so there is some flexibility in defining the distance
- There are different ways to vote. For example, of the k nearest neighbors, I might give the closest ones more weight than farther ones.
- We have to decide how to break ties in the vote.

Example: Classifying Two Moons

Let's consider a synthetic dataset in the shape of "two moons". Here, each sample has two pieces of information:

- the *features*, denoted by x_i, which are just a two-dimensional coordinate and
- a *class*, denoted by y_i, which is either 0 and 1.

```
# there are two features contained in X and the labels are contained in y
X,y = make_moons(n_samples=500,random_state=1,noise=0.3)

plt.scatter(X[y == 0, 0], X[y == 0, 1], color="OrangeRed", marker="o",label="0")
plt.scatter(X[y == 1, 0], X[y == 1, 1], color="SteelBlue", marker="s",label="1")

test1 = [1.5,-1] # should be 1
test2 = [-1,1] # should be 0
test3 = [0,0.1] # borderline, probably 1
test4 = [0,0.5] # borderline, probably 0
test5 = [0.5,0.75] # borderline, probably 0
test = [test1, test2, test3, test4, test5]

plt.scatter([x[0] for x in test], [y[1] for y in test], color="tab:Purple", marker="x",
label="Test", s=1000)

plt.legend(scatterpoints=1)

plt.title('Two Moons Dataset')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.show()
```


Two Moons Dataset
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
feature 2
−1
0
1
2
feature 1
0
1
Test

KNeighborsClassifier

```
# The key parameter is the number of neighbors (k) to consider.
model = KNeighborsClassifier(n_neighbors = 100)
# Here we train the model with the data (X) and the labels (y)
model.fit(X, y)

# We can then use new, never beofre seen data to make predictions
print(test1, "expected 1, predicted:", model.predict([test1]))
print(test2, "expected 0, predicted:", model.predict([test2]))
print(test3, "borderline 1, predicted:", model.predict([test3]))
print(test4, "borderline 0, predicted:", model.predict([test4]))
print(test5, "borderline 0, predicted:", model.predict([test5]))
```

[1.5, -1] expected 1, predicted: [1]

[-1, 1] expected 0, predicted: [0]

[0, 0.1] borderline 1, predicted: [1]

[0, 0.5] borderline 0, predicted: [0]

[0.5, 0.75] borderline 0, predicted: [0]

```
# Here we initialize the model. The key parameter is the number of neighbors (k) to consider.
model = KNeighborsClassifier(n_neighbors = 1)
# Here we train the model with the data (X) and the labels (y)
model.fit(X, y)

# We can then use new, never beofre seen data to make predictions
print(test1, "expected 1, predicted:", model.predict([test1]))
print(test2, "expected 0, predicted:", model.predict([test2]))
print(test3, "borderline 1, predicted:", model.predict([test3]))
print(test4, "borderline 0, predicted:", model.predict([test4]))
print(test5, "borderline 0, predicted:", model.predict([test5]))
```

[1.5, -1] expected 1, predicted: [1]

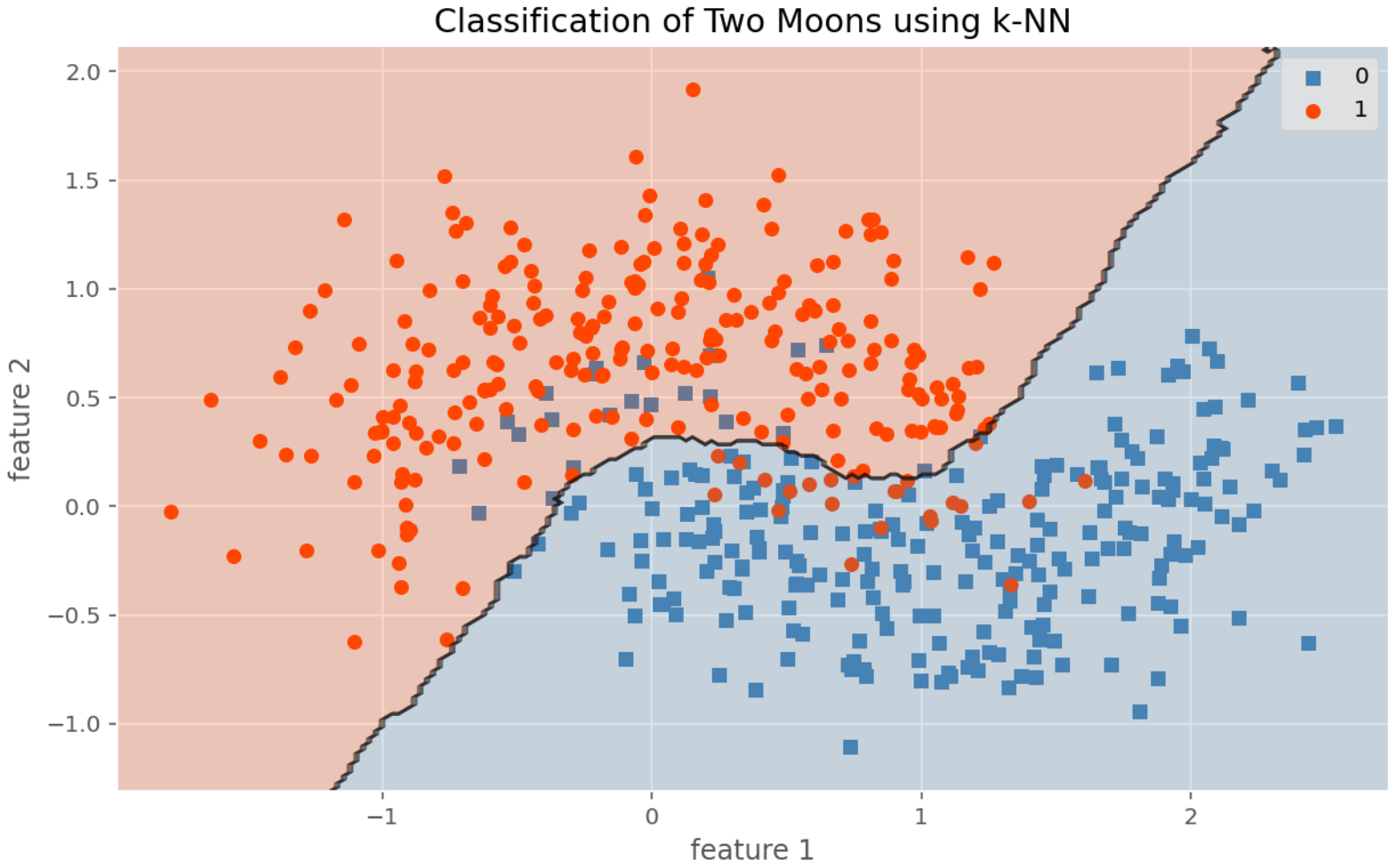
[-1, 1] expected 0, predicted: [0]

[0, 0.1] borderline 1, predicted: [1]

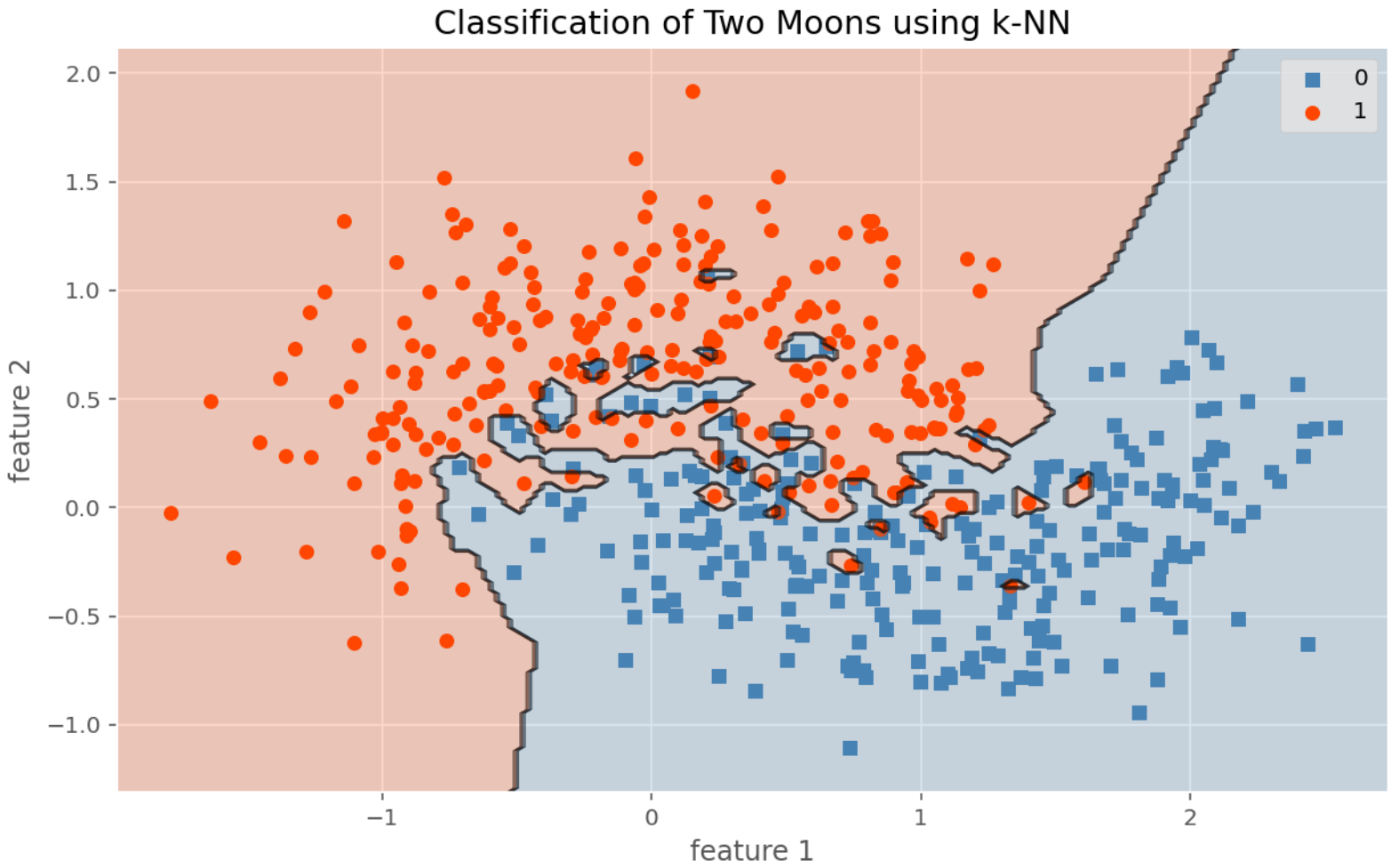
[0, 0.5] borderline 0, predicted: [1]

[0.5, 0.75] borderline 0, predicted: [1]

Classification of Two Moons using k-NN with k=100



Classification of Two Moons using k-NN with k=1



The larger the `k` values, the smoother / simpler the decision boundary becomes.

Evaluating a classification method

In regression methods, we had several methods for evaluating a particular model:

- R^2 value
- hypothesis tests associated with the model and individual predictor variables

How can we evaluate the performance of a classification method?

Confusion Matrix

The confusion matrix is a table where each column of the matrix represents the number of samples in each predicted class and each row represents the number of samples in each actual class.

Consider the results from a classifier that is trying to classify 27 images of cats, dogs, and rabbits. Here is an example of what the confusion matrix might look like

		Predicted		
		Cat	**Dog**	**Rabbit**
Actual class	**Cat**	5	3	0
	Dog	2	3	1
	Rabbit	0	2	11

This classifier is very good at distinguishing between cats and rabbits but lousy at recognizing dogs...half are misclassified!

Precision vs. Recall

Two key metrics that can be obtained from the confusion matrix for binary classification are precision and recall.

Precision measures how accurately our predicted set contains only the desired class,
i.e., a high false positive rate leads to low precision.

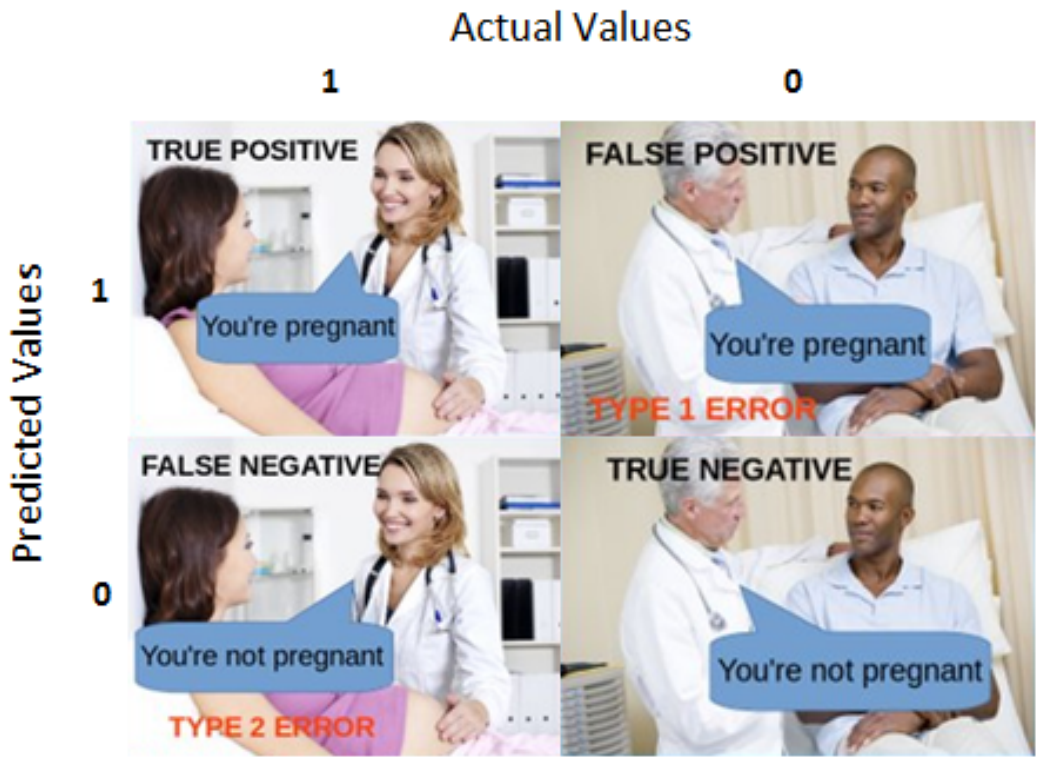
Recall measures whether we accurately pick up all the cases in the desired class,
i.e., a high false negative rate leads to low recall.

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Here, TP is the number of true positives, FP is the number of false positives, and FN is the number of false negatives.

Often, precision and recall are inversely related; it is not possible to increase one without decreasing the other. For example, a trivial way to have perfect precision (precision = 1 = 100%) is if you make one correct positive prediction. However, this is not very useful since there will be many false negatives, so recall will be low. Conversely, if you can make one correct negative prediction, recall will be 100%, while precision will be very low.

In a particular application, it might be more desirable to have either better precision or recall.



F-measure, ROC curve, and Jaccard similarity score

There have been some attempts to combine precision and recall into a single measure, such as the F-measure, which is the harmonic mean of precision and recall:

$$\text{F-measure} = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}.$$

The F-measure is large if precision and recall are close.

The ROC curve further illustrates the trade-off between precision and recall.

The Jaccard similarity score is another measure of accuracy, given by

$$\text{J} = \frac{\text{TP}}{\text{TP} + \text{FP} + \text{TN}}.$$

```
y_pred = model.predict(X)
print('Confusion Matrix:')
print(metrics.confusion_matrix(y_true = y, y_pred = y_pred))

print('Precision = ', metrics.precision_score(y_true = y, y_pred = y_pred))
print('Recall = ', metrics.recall_score(y_true = y, y_pred = y_pred))
print('F-score = ', metrics.f1_score(y_true = y, y_pred = y_pred))

print('Jaccard similarity score', metrics.jaccard_score(y_true = y, y_pred = y_pred))
```

Confusion Matrix:

[[228 22]

[23 227]]

Precision = 0.911646586345381 5

Recall = 0.908

F-score = 0.9098196392785571

Jaccard similarity score 0.8345588235294118

Dataset: The Iris dataset

Let's try kNN on a real dataset. We'll use the super-common Iris dataset for a demonstration. This dataset was introduced in 1936 by the statistician Sir Ronald A. Fisher.

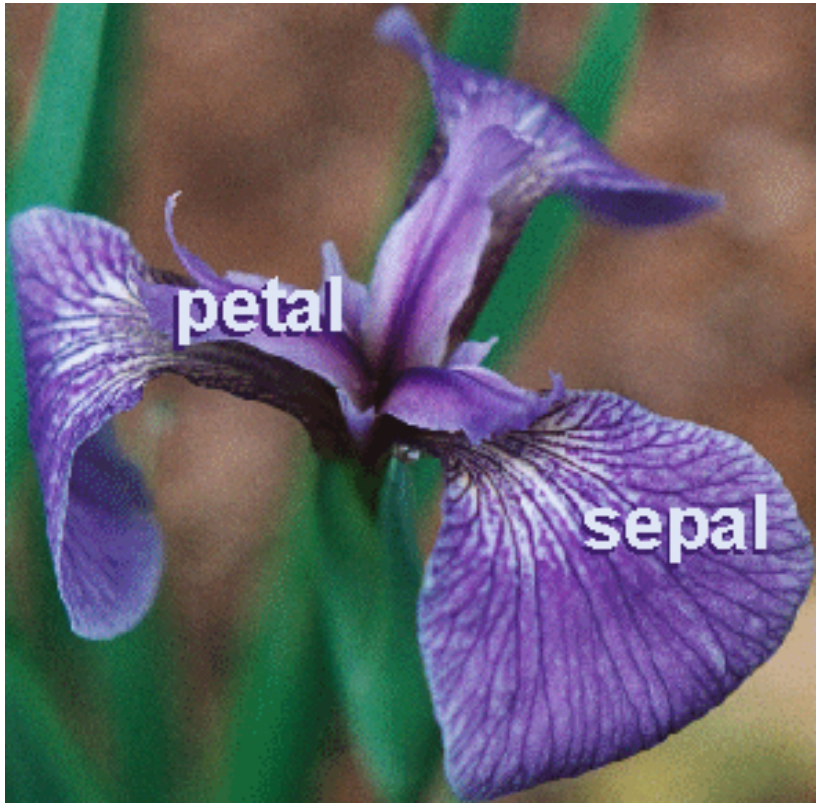
The dataset contains 4 features (attributes) of 50 samples containing 3 different species of iris plants. The goal is to classify the species of iris plant given the attributes.

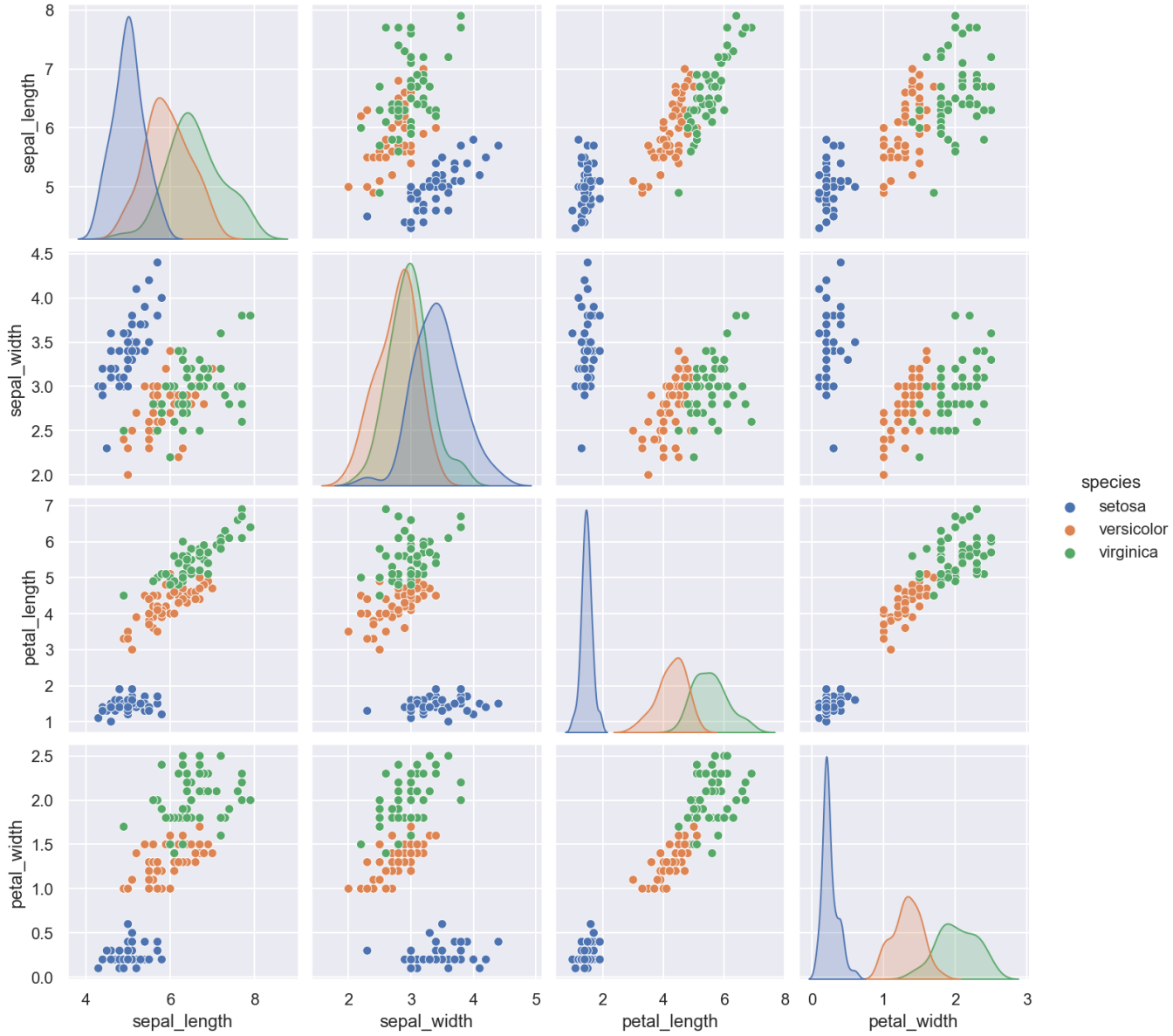
The species, or classes are:

1. Iris Setosa
2. Iris Versicolour
3. Iris Virginica

The features (attributes) we have are::

1. sepal length (cm)
2. sepal width (cm)
3. petal length (cm)
4. petal width (cm)

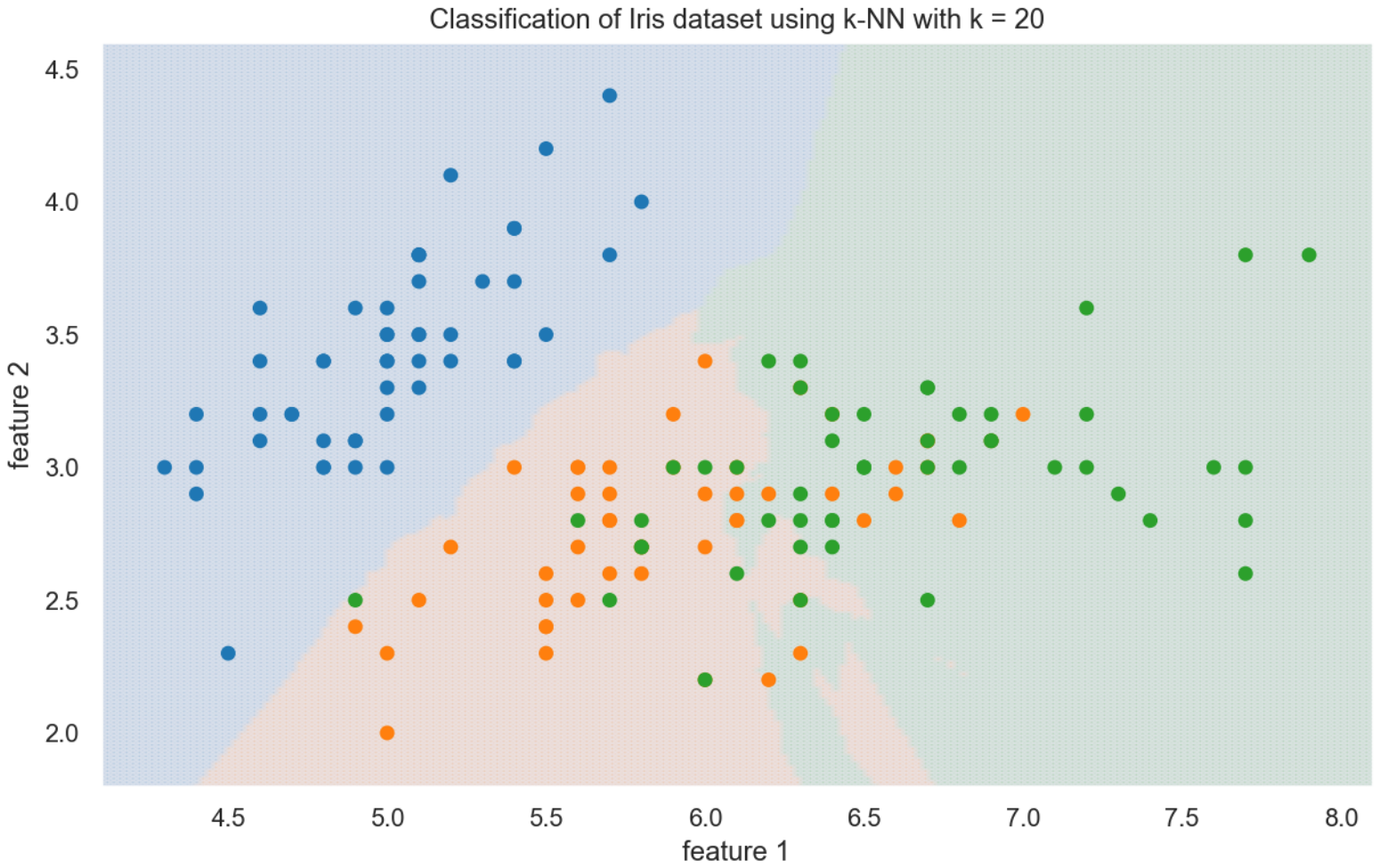

petal
sepal


sepal_length
sepal_width
petal_length
petal_width
species
setosa
versicolor
virginica

```
# import data, scikit-learn also has this dataset built-in
iris = load_iris()

# For easy plotting and interpretation, we only use first 2 features here.
# We're throwing away useful information - don't do this at home!
X = iris.data[:,:2]
y = iris.target

# Create color maps
clrs = sns.color_palette("tab10").as_hex();
cmap_bold = matplotlib.colors.ListedColormap(clrs[:3])

# plot data
plt.scatter(X[:, 0], X[:, 1], c=y,  marker="o", cmap=cmap_bold, s=30)

x_min, x_max = X[:, 0].min() - padding, X[:, 0].max() + padding
y_min, y_max = X[:, 1].min() - padding, X[:, 1].max() + padding
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.title('Iris dataset')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.show()
```

We see that it would be fairly easy to separate the "blue" irises from the two classes. However, separating the "green" and "orange" ones would be a challenge.

```
## set up the model, k-NN classification with k = ?
k = 20
clf = KNeighborsClassifier(n_neighbors=k)
clf.fit(X, y)

# plot classification
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),np.linspace(y_min, y_max, 200))
zz = clf.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
plt.pcolormesh(xx, yy, zz, cmap=cmap_bold, alpha=0.05, shading='gouraud')

# plot data
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap_bold, s=30)

plt.title('Classification of Iris dataset using k-NN with k = '+ str(k))
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.xlim(x_min, x_max)
plt.ylim(y_min, y_max)
plt.show()
```


Classification of Iris dataset using k-NN with k = 20
feature 2
4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
feature 1

```
print('Confusion Matrix:')
y_pred = clf.predict(X)
print(metrics.accuracy_score(y_true=y, y_pred=y_pred))
print(metrics.confusion_matrix(y_true = y, y_pred = y_pred))
```

Confusion Matrix:

0.7866666666666666

[[50 0 0]

[0 31 19]

[0 13 37]]

Comments on the parameter, k

- For k large (say $k = 100$), the *decision boundary* (boundary between classes) is smooth. The model is not very complex - it could basically be described by a few lines. The model has low variance in the sense that if the data were to change slightly, the model wouldn't change much. (There are many voters.) Since the model doesn't depend on the data very much, we might expect that it would *generalize* to new data points.
- For k small (say $k = 1$), the decision boundary is very wiggly. The model is very complex - it definitely can't be described by a few lines. The model has high variance in the sense that if the data were to change slightly, the model would change quite a bit. Since the model is very dependent on the dataset, we would say that it wouldn't generalize to new data points well. In this case, we would say that the model has overfit the data.

Model generalizability and cross-validation

In classification, and other prediction problems, we would like to develop a model on a dataset, the *training dataset*, that will not only perform well on that dataset but on similar data that the model hasn't yet seen, the *testing dataset*. If a model satisfies this criterion, we say that it is *generalizable*.

If a model has 100% accuracy on the training dataset ($k = 1$ in k-NN) but doesn't generalize to new data, then it isn't a very good model. We say that this model has *overfit* the data. On the other hand, it isn't difficult to see that we could also *underfit* the data (taking k large in k-NN). In this case, the model isn't complex enough to have good accuracy on the training dataset.

Cross-validation

Cross-validation is a general method for assessing how the results of a predictive model (classification, regression,...) will *generalize* to an independent data set. In classification, cross-validation is a method for assessing how well the classification model will predict the class of points that weren't used to *train* the model.

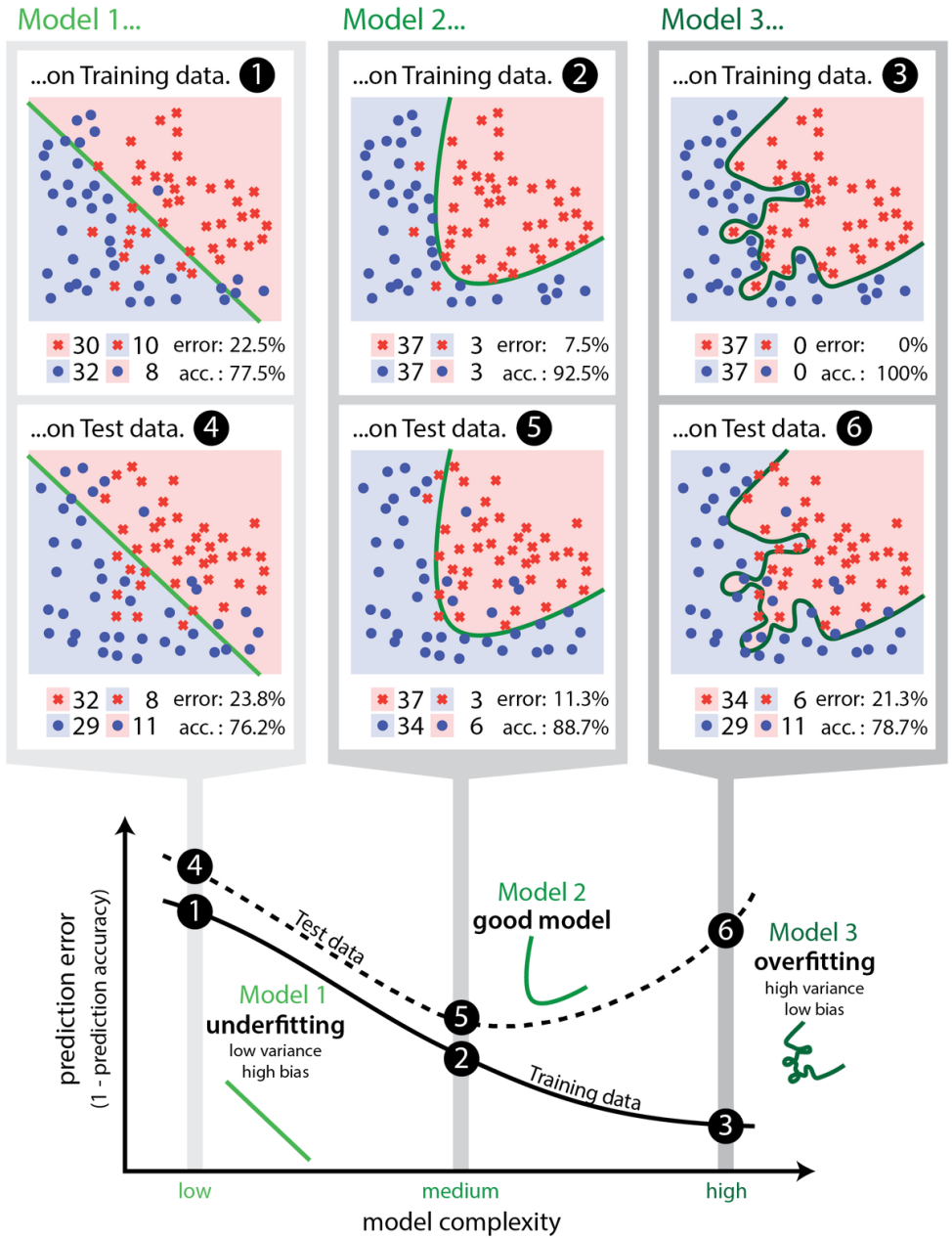
The idea of the method is simple:

1. Split the dataset into two groups: the training dataset and the test dataset.
2. Train the model on the training dataset.
3. Check the accuracy of the model on the test dataset.

In practice, you have to decide how to split the data into groups (i.e. how large the groups should be). You might also want to repeat the experiment so that the assessment doesn't depend on the way in which you split the data into groups.

```
# Split into training and test sets
X,y = make_moons(n_samples=1000,random_state=1,noise=0.5)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=2, test_size=0.7)

for k in [1, 2, 5, 10, 50, 100]:
    print("k is:", k)
    clf = KNeighborsClassifier(n_neighbors=k)
    clf.fit(X_train, y_train)

    y_pred = clf.predict(X_test)
    print("Accuracy Train:", metrics.accuracy_score(y_true=y_train, y_pred=clf.predict(X_train)))
    print("Accuracy Test:", metrics.accuracy_score(y_true=y_test, y_pred=y_pred))
    print(metrics.confusion_matrix(y_true=y_test, y_pred=y_pred))
    print()
```

```
k is: 1
Accuracy Train: 1.0
Accuracy Test: 0.7028571428571428
[[254 109]
 [ 99 238]]

k is: 2
Accuracy Train: 0.8433333333333334
Accuracy Test: 0.7071428571428572
[[310  53]
 [152 185]]

k is: 5
Accuracy Train: 0.84
Accuracy Test: 0.7685714285714286
[[275  88]
 [ 74 263]]

k is: 10
Accuracy Train: 0.8133333333333334
Accuracy Test: 0.7928571428571428
[[299  64]
 [ 81 256]]

k is: 50
Accuracy Train: 0.7866666666666666
Accuracy Test: 0.7971428571428572
[[278  85]
 [ 57 280]]

k is: 100
Accuracy Train: 0.77
Accuracy Test: 0.7842857142857143
[[270  93]
 [ 58 279]]
```

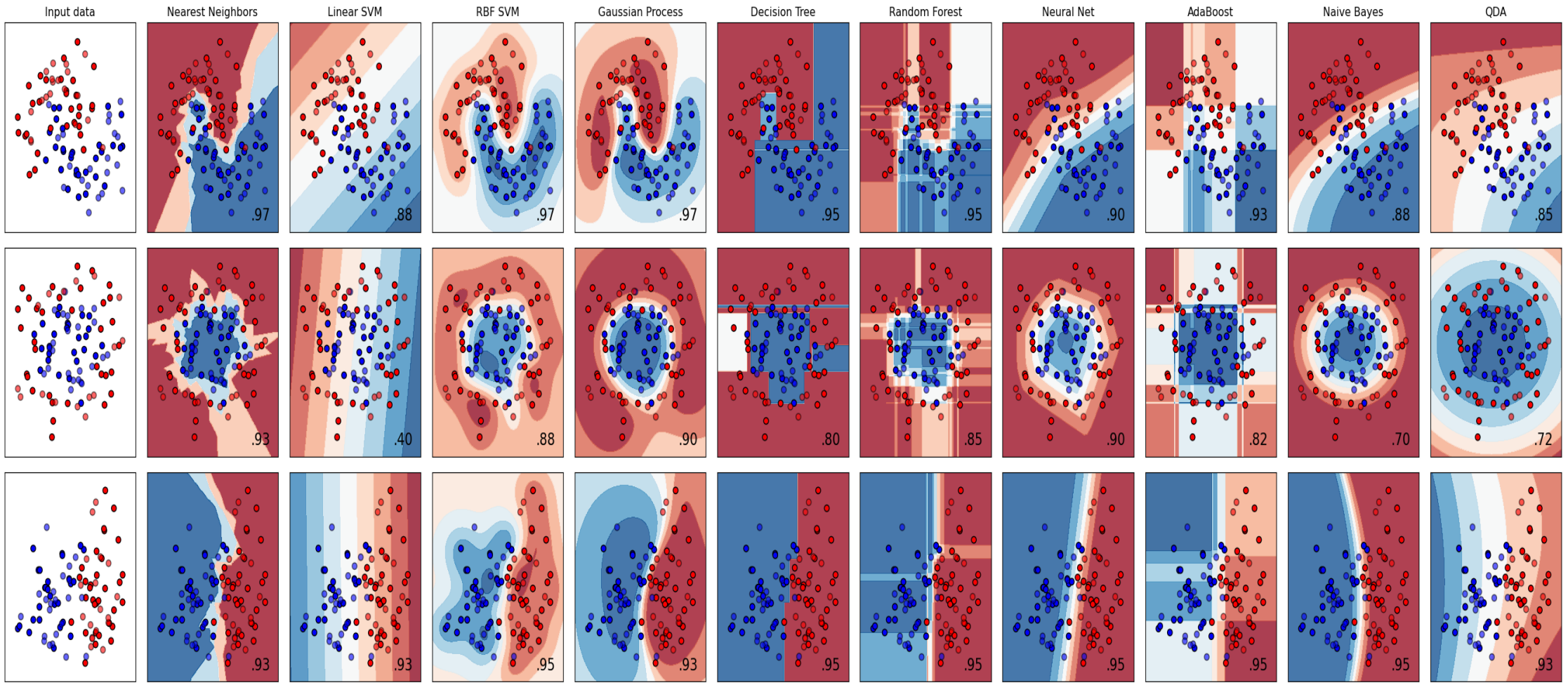

Model 1...
Model 2...
Model 3...
...on Training data. 1
30 10 error: 22.5%
32 8 acc.: 77.5%
...on Training data. 2
37 3 error: 7.5%
37 3 acc.: 92.5%
...on Training data. 3
37 0 error: 0%
37 0 acc.: 100%
...on Test data. 4
32 8 error: 23.8%
29 11 acc.: 76.2%
...on Test data. 5
37 3 error: 11.3%
34 6 acc.: 88.7%
...on Test data. 6
34 6 error: 21.3%
29 11 acc.: 78.7%
prediction error
(1 - prediction accuracy)
Test data
Training data
Model 1
underfitting
low variance
high bias
Model 2
good model
Model 3
overfitting
high variance
low bias
low
medium
high
model complexity

As the model becomes more complex (k decreases), the accuracy always increases for the training dataset. But, at some point, it starts to overfit the data and the accuracy decreases for the test dataset! Cross validation techniques will allow us to find the sweet-spot for the parameter k! (Think: Goldilocks and the Three Bears.)

Wrap-up: k-NN

1. k-NN is a very simple method that can be used for classification.
2. Model accuracy (measured on the training dataset) and generalizability (measured on the testing dataset) are both important and often in contention with one another.
3. Model accuracy can be measured using the confusion matrix, precision, recall, F-measure, or the Jaccard similarity score. Generalizability can be measured via cross validation.
4. Picking parameters in models (such as k in k-NN) is non-trivial, but can be done via cross validation.

Classification method preview



Data: Titanic Passengers

PassengerId	*Survived*	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.25		S
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.925		S
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1	0	113803	53.1	C123	S
5	0	3	Allen, Mr. William Henry	male	35	0	0	373450	8.05		S

Variable descriptions:

- **Survived:** (0 = No; 1 = Yes)
- **pclass:** Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
- **sibsp:** Number of Siblings/Spouses Aboard
- **parch:** Number of Parents/Children Aboard
- **ticket:** Ticket Number
- **fare:** Passenger Fare
- **cabin:** Cabin
- **embarked:** Port of Embarkation (C = Cherbourg; Q = Queenstown; S = Southampton)

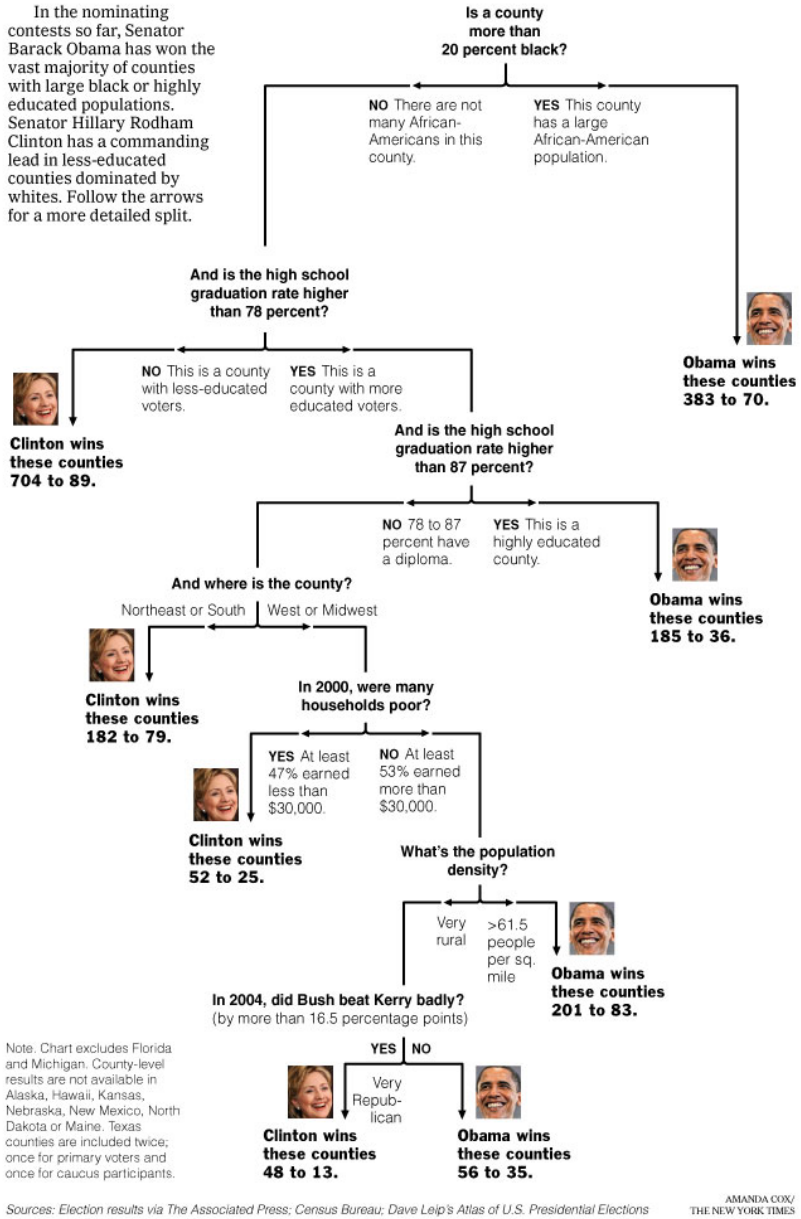
We're trying to decide whether a passenger with particular attributes will survive the Titanic disaster, so the **survival** variable is our label, the other columns are the **features**.

Decision Trees

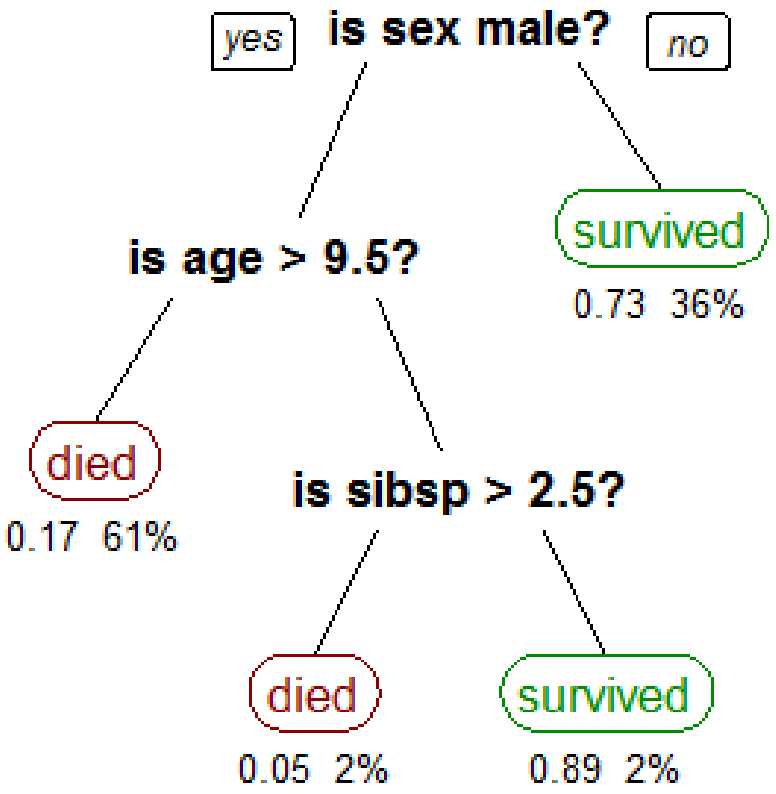
Decision trees are very intuitive classification and regression tools.

Consider the following decision tree from the New York Times,
which predicts whether a county voted for Obama or Clinton in the 2008 democratic primary.

Decision Tree: The Obama-Clinton Divide



Here is another decision tree for the survival of passengers on the titanic. The figures under the leaves show the probability of survival and the percentage of observations in the leaf.



The use of a decision tree

Suppose someone gives you this tree and a new person. In order to predict whether or not the person would have died on the titanic, you ask the following questions, in order:

- Is the person male? If no, we predict they would have survived. If yes, continue.
- Is the person older than 9.5 years? If yes, we predict they would have died. If no, continue.
- Did the person have 3 or more siblings? If yes, we predict they would have died. If no, they would have survived.

Many of these decisions make intuitive sense: social convention gave women and children preference for rescue. Some, however, are not as obvious: why is having more than 3 siblings or spouses a marker? Could this be a marker for a hidden variable?

The question we'll move to now is: How would one create such a tree?

Building a Decision Tree

Building a decision tree isn't really much harder than reading one. Here's the essential rundown of the ID3 algorithm:

- If the data all have the same label, create a leaf node that predicts that label, and you're done.
- If the list of attributes is empty (e.g., because you have used all already), create a leaf node that predicts the most common label.
- Else, partition the data by each of the attributes; choose the partition with the lowest error.
- Recursively continue on each partitioned subset using the remaining attributes.
- Terminate when no attributes are left (see above), or when desired depth or another termination criterion is reached.

Building a Decision Tree

So, how do you choose the partition with the lowest error? There are various approaches.

Let's say we're building a classification tree by considering a list of predictors. In our example above we want to be able to classify whether people have survived based on things like gender, age, the booked fare, etc. Let's call the variables $X_{i,p}$ (i for passengers, p for predictors).

Initially, for the first split, we consider all the passengers and all the predictors. We also have an observed label Y_i for each passenger, and a predicted label $\hat{Y}_i$.

We can calculate the *mean error*,

$$ME = \frac{1}{N}\sum_{i=1}^{N} \ell(\hat{Y}_i, Y_i),$$

where $\ell(\hat{Y}_i, Y_i)$ is the error for sample i. Here, the error would be

- squared error for regression, *i.e.*, $\ell(\hat{Y}_i, Y_i) = (\hat{Y}_i - Y_i)^2$ and
- either *cross-entropy* or *Gini impurity* for classification.

We want to achieve two things: pick the **best split** for the **best predictor**.

- At **each step** of the algorithm we consider a list of possible decisions or splits, *e.g.*, $X_{i,6} > 9$ (age is greater than 9), or $X_{i,5} = female$.
- For each possible decision we recalculate the predictor for that rule, for example $\hat{Y}_i = 1$ if $X_{i,6} > 9$ and 0 otherwise.
- We recalculate the mean error for each possible decision
- We choose the decision that reduces the error by the largest amount.
- Then continue with the next step on the reduced input set.

In building decision trees, it is easy to overfit the data. There are several methods for avoiding this. Simple strategies include limiting the depth of a tree or only splitting when we have more than N samples left.

Decision Trees with SciKit Learn

Scikit-learn has a nice decision tree implementation which we'll use to learn a tree for our Titanic dataset.

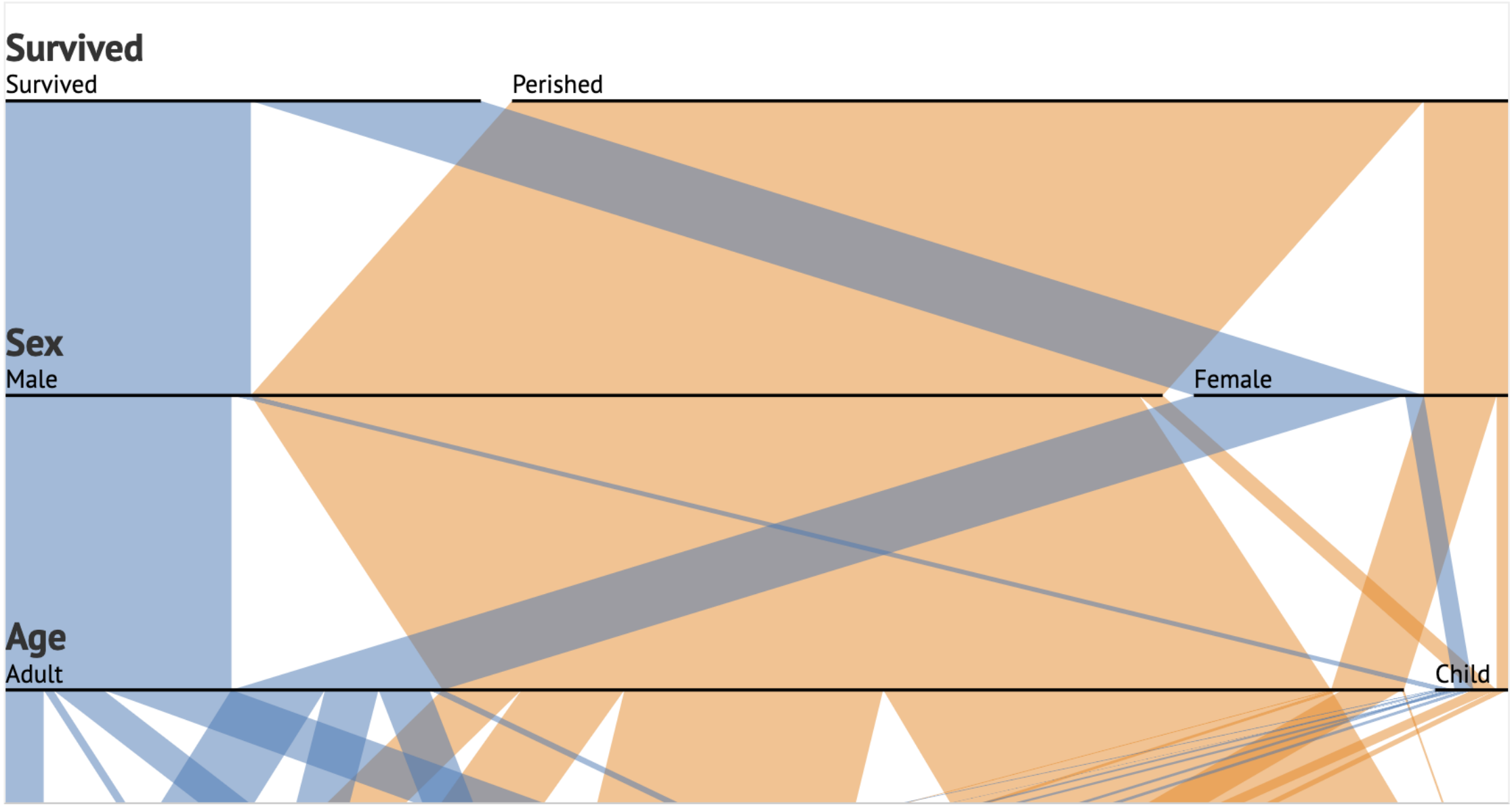
Data cleanup

- Age has missing values. Let's assume that a person with missing age is of mean age (this is not necessarily a good decision).
- Embarked has missing values, we add a dedicated category for unknown embarkation points.
- We need to convert the categorical values to numerical values.

```
titanic["Age"] = titanic["Age"].fillna(titanic["Age"].mean())

def sex_to_numeric(x):
    if x=='male':
        return 0
    if x=='female':
        return 1
    else:
        return x

titanic["Sex"] = titanic["Sex"].apply(sex_to_numeric)
```

```
def embarked_to_numeric(x):
    if x=="S":
        return 0
    if x=="C":
        return 1
    if x=="Q":
        return 2
    else:
        return 3

titanic["Embarked"] = titanic["Embarked"].apply(embarked_to_numeric)
```

Parallel Sets



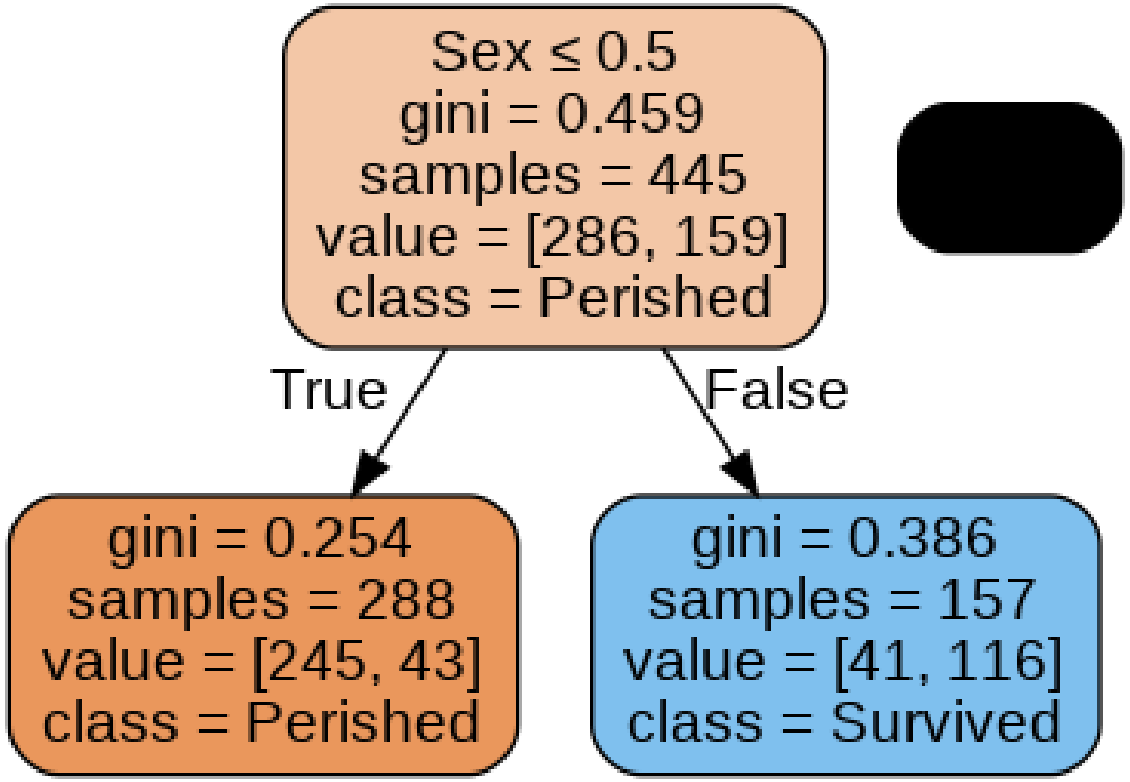
First Tree

Let's look at a decision tree that **ONLY operates on sex**!

```
decisionTree = tree.DecisionTreeClassifier()

XTrain, XTest, yTrain, yTest = splitData(["Sex"])
# fit the tree with the traing data
decisionTree = decisionTree.fit(XTrain, yTrain)

# predict with the training data
y_pred_train = decisionTree.predict(XTrain)
# measure accuracy
print('Accuracy on training data = ', metrics.accuracy_score(y_true = yTrain, y_pred = y_pred_train))

# predict with the test data
y_pred = decisionTree.predict(XTest)
# measure accuracy
print('Accuracy on test data = ', metrics.accuracy_score(y_true = yTest, y_pred = y_pred))

renderTree(decisionTree, ["Sex"])
```

Accuracy on training data = 0.8112359550561797

Accuracy on test data = 0.7623318385650224



~76% correct on the test set isn't bad! - sex seems to be a very good indicator of whether someone has survived or not.

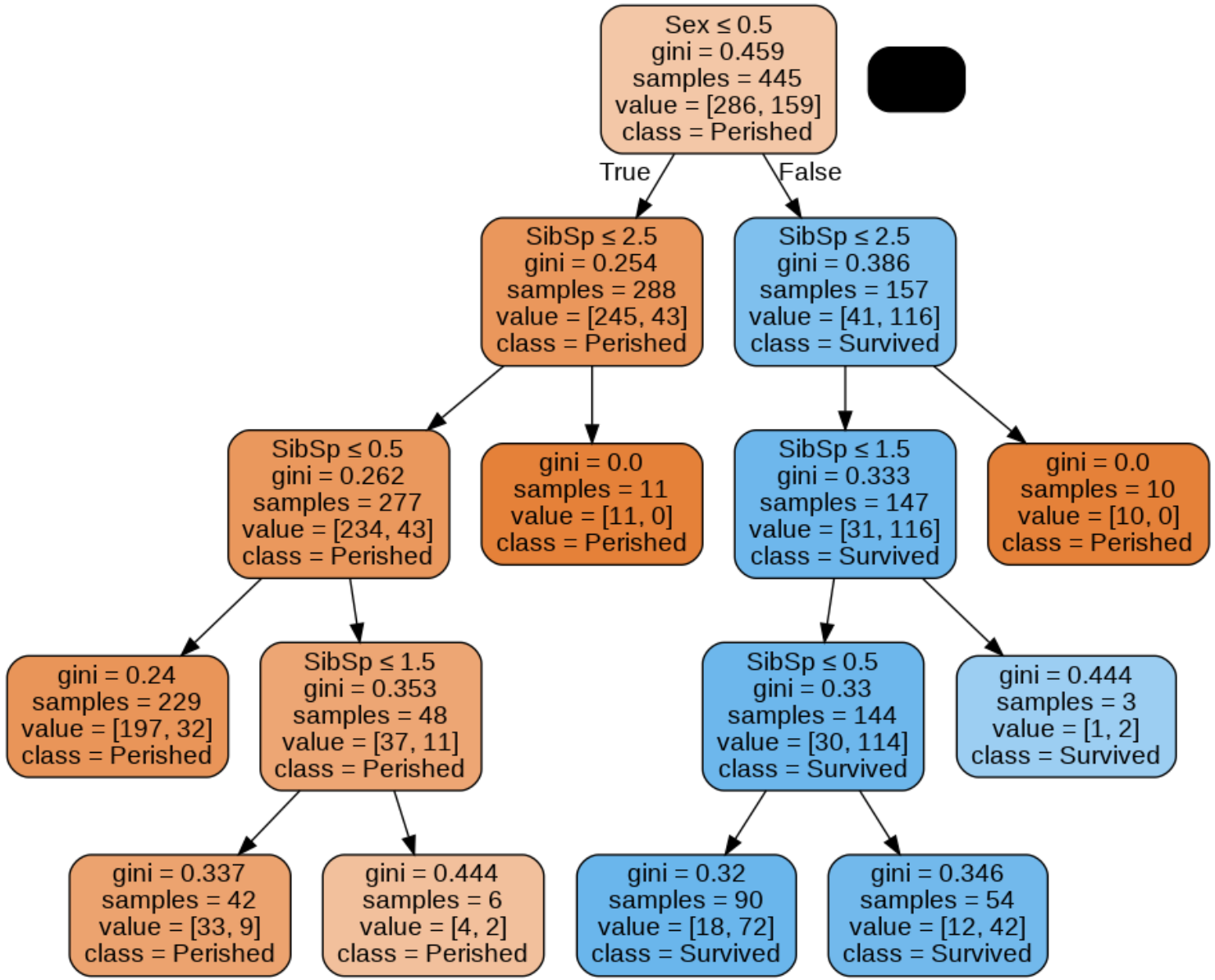
Adding more features

Let's add the number of siblings and spouses as a feature.

```
#  train tree on sex and the number of the number of siblings/spouses aboard
used_features = ["Sex", "SibSp"]
XTrain, XTest, yTrain, yTest = splitData(used_features)
decisionTree = tree.DecisionTreeClassifier()
decisionTree = decisionTree.fit(XTrain, yTrain)

y_pred_train = decisionTree.predict(XTrain)
print('Accuracy on training data= ', metrics.accuracy_score(y_true = yTrain, y_pred = y_pred_train))

y_pred = decisionTree.predict(XTest)
print('Accuracy on test data= ', metrics.accuracy_score(y_true = yTest, y_pred = y_pred))
renderTree(decisionTree, used_features)
```

Accuracy on training data= 0.8337078651685393

Accuracy on test data= 0.7600896860986547


Sex ≤ 0.5
gini = 0.459
samples = 445
value = [286, 159]
class = Perished
True
False
SibSp ≤ 2.5
gini = 0.254
samples = 288
value = [245, 43]
class = Perished
SibSp ≤ 2.5
gini = 0.386
samples = 157
value = [41, 116]
class = Survived
SibSp ≤ 0.5
gini = 0.262
samples = 277
value = [234, 43]
class = Perished
gini = 0.0
samples = 11
value = [11, 0]
class = Perished
SibSp ≤ 1.5
gini = 0.333
samples = 147
value = [31, 116]
class = Survived
gini = 0.0
samples = 10
value = [10, 0]
class = Perished
gini = 0.24
samples = 229
value = [197, 32]
class = Perished
SibSp ≤ 1.5
gini = 0.353
samples = 48
value = [37, 11]
class = Perished
SibSp ≤ 0.5
gini = 0.33
samples = 144
value = [30, 114]
class = Survived
gini = 0.444
samples = 3
value = [1, 2]
class = Survived
gini = 0.337
samples = 42
value = [33, 9]
class = Perished
gini = 0.444
samples = 6
value = [4, 2]
class = Perished
gini = 0.32
samples = 90
value = [18, 72]
class = Survived
gini = 0.346
samples = 54
value = [12, 42]
class = Survived

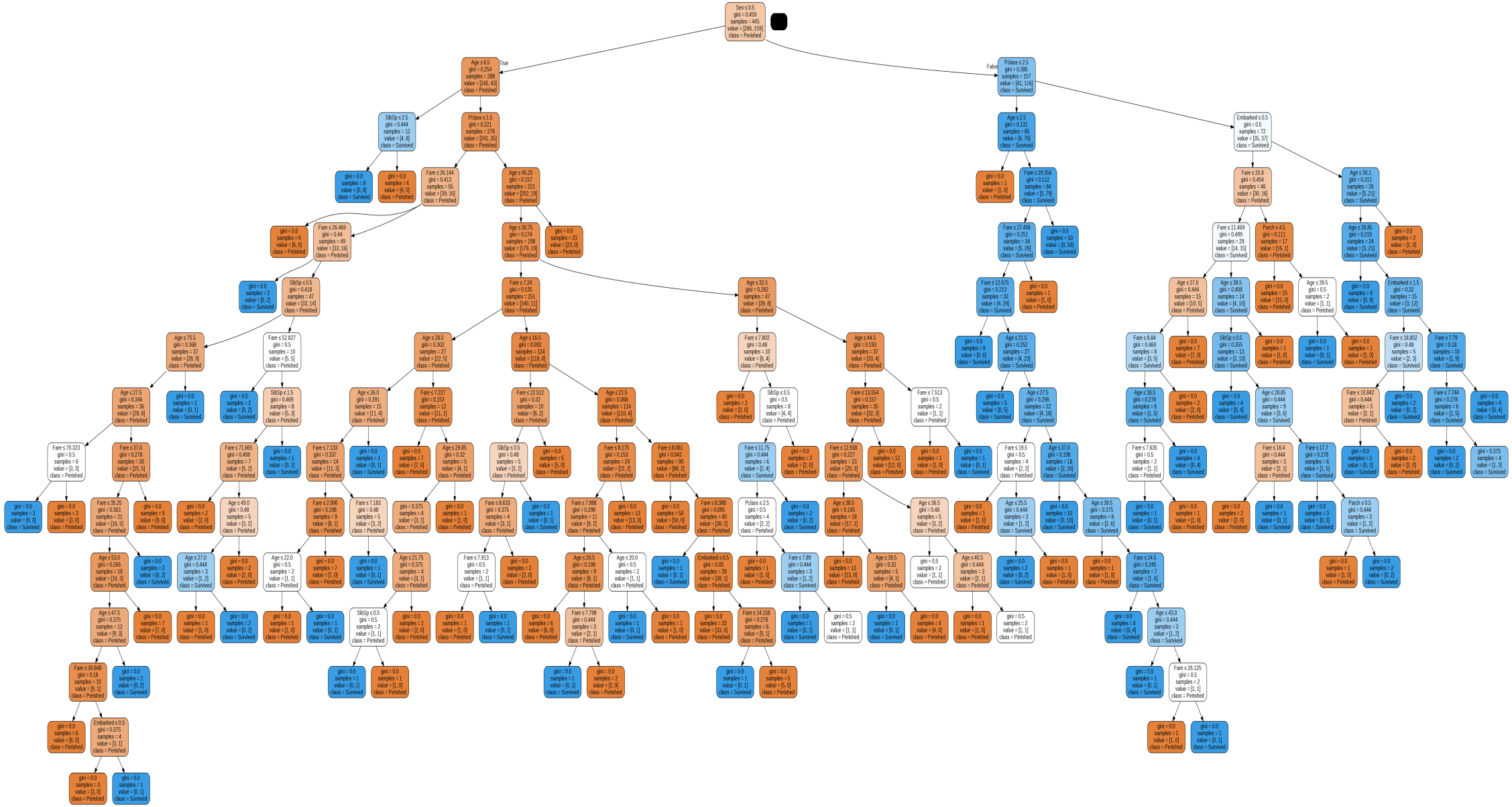
All features

Our accuracy on the training data has gone up, **but the accuracy on the test data has gone down**. It looks like we're overfitting. But maybe we just selected the wrong features? Let's just try all of them!

```
all_features = ["Pclass", "Sex", "Age", "SibSp", "Parch", "Fare", "Embarked"]

XTrain, XTest, yTrain, yTest = splitData(all_features)
decisionTree = tree.DecisionTreeClassifier()
decisionTree = decisionTree.fit(XTrain, yTrain)

y_pred_train = decisionTree.predict(XTrain)
print('Accuracy on training data= ', metrics.accuracy_score(y_true = yTrain, y_pred = y_pred_train))

y_pred = decisionTree.predict(XTest)
print('Accuracy on test data= ', metrics.accuracy_score(y_true = yTest, y_pred = y_pred))
renderTree(decisionTree, all_features)
```

Accuracy on training data= 0.9887640449438202

Accuracy on test data= 0.7556053811659192


Sex ≤ 0.5
gini = 0.459
samples = 445
value = [286, 159]
class = Perished
True
False
Age ≤ 6.5
gini = 0.254
samples = 288
value = [245, 43]
class = Perished
Pclass ≤ 2.5
gini = 0.396
samples = 157
value = [41, 116]
class = Survived
SibSp ≤ 2.5
gini = 0.444
samples = 12
value = [4, 8]
class = Survived
Pclass ≤ 1.5
gini = 0.221
samples = 276
value = [241, 35]
class = Perished
Age ≤ 2.5
gini = 0.131
samples = 85
value = [6, 79]
class = Survived
Embarked ≤ 0.5
gini = 0.5
samples = 72
value = [35, 37]
class = Survived
gini = 0.0
samples = 8
value = [0, 8]
class = Survived
gini = 0.0
samples = 4
value = [4, 0]
class = Perished
Fare ≤ 26.144
gini = 0.413
samples = 55
value = [39, 16]
class = Perished
Age ≤ 45.25
gini = 0.157
samples = 221
value = [202, 19]
class = Perished
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
Fare ≤ 29.356
gini = 0.112
samples = 84
value = [5, 79]
class = Survived
Fare ≤ 20.8
gini = 0.454
samples = 46
value = [30, 16]
class = Perished
Age ≤ 30.1
gini = 0.311
samples = 26
value = [5, 21]
class = Survived
gini = 0.0
samples = 6
value = [6, 0]
class = Perished
Fare ≤ 26.469
gini = 0.44
samples = 49
value = [33, 16]
class = Perished
Age ≤ 30.75
gini = 0.174
samples = 198
value = [179, 19]
class = Perished
gini = 0.0
samples = 23
value = [23, 0]
class = Perished
Fare ≤ 27.498
gini = 0.251
samples = 34
value = [5, 29]
class = Survived
gini = 0.0
samples = 50
value = [0, 50]
class = Survived
Fare ≤ 11.469
gini = 0.499
samples = 29
value = [14, 15]
class = Survived
Parch ≤ 4.5
gini = 0.111
samples = 17
value = [16, 1]
class = Perished
Age ≤ 26.85
gini = 0.219
samples = 24
value = [3, 21]
class = Survived
gini = 0.0
samples = 2
value = [2, 0]
class = Perished
gini = 0.0
samples = 2
value = [0, 2]
class = Survived
SibSp ≤ 0.5
gini = 0.418
samples = 47
value = [33, 14]
class = Perished
Fare ≤ 7.24
gini = 0.135
samples = 151
value = [140, 11]
class = Perished
Age ≤ 32.5
gini = 0.282
samples = 47
value = [39, 8]
class = Perished
Fare ≤ 12.675
gini = 0.213
samples = 33
value = [4, 29]
class = Survived
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
Age ≤ 27.0
gini = 0.444
samples = 15
value = [10, 5]
class = Perished
Age ≤ 38.5
gini = 0.408
samples = 14
value = [4, 10]
class = Survived
gini = 0.0
samples = 15
value = [15, 0]
class = Perished
Age ≤ 39.5
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
Embarked ≤ 1.5
gini = 0.32
samples = 15
value = [3, 12]
class = Survived
gini = 0.0
samples = 2
value = [2, 0]
class = Perished
gini = 0.0
samples = 2
value = [0, 2]
class = Survived
Age ≤ 75.5
gini = 0.368
samples = 37
value = [28, 9]
class = Perished
SibSp ≤ 1.5
gini = 0.469
samples = 8
value = [5, 3]
class = Perished
Age ≤ 28.0
gini = 0.302
samples = 27
value = [22, 5]
class = Perished
Age ≤ 16.5
gini = 0.092
samples = 124
value = [118, 6]
class = Perished
Fare ≤ 7.802
gini = 0.48
samples = 10
value = [6, 4]
class = Perished
Age ≤ 44.5
gini = 0.193
samples = 37
value = [33, 4]
class = Perished
gini = 0.0
samples = 6
value = [0, 6]
class = Survived
Age ≤ 21.5
gini = 0.252
samples = 27
value = [4, 23]
class = Survived
Fare ≤ 8.84
gini = 0.469
samples = 8
value = [3, 5]
class = Survived
gini = 0.0
samples = 7
value = [7, 0]
class = Perished
Fare ≤ 10.842
gini = 0.444
samples = 3
value = [2, 1]
class = Perished
gini = 0.0
samples = 2
value = [0, 2]
class = Survived
Fare ≤ 18.802
gini = 0.48
samples = 5
value = [2, 3]
class = Survived
Fare ≤ 7.79
gini = 0.18
samples = 10
value = [1, 9]
class = Survived
Age ≤ 27.5
gini = 0.346
samples = 36
value = [28, 8]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
gini = 0.0
samples = 2
value = [0, 2]
class = Survived
Fare ≤ 52.827
gini = 0.5
samples = 10
value = [5, 5]
class = Perished
Age ≤ 26.0
gini = 0.391
samples = 15
value = [11, 4]
class = Perished
Fare ≤ 7.227
gini = 0.153
samples = 12
value = [11, 1]
class = Perished
Fare ≤ 22.512
gini = 0.32
samples = 10
value = [8, 2]
class = Perished
Age ≤ 21.5
gini = 0.068
samples = 114
value = [110, 4]
class = Perished
SibSp ≤ 0.5
gini = 0.5
samples = 8
value = [4, 4]
class = Perished
Fare ≤ 13.554
gini = 0.157
samples = 35
value = [32, 3]
class = Perished
Fare ≤ 7.513
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
Age ≤ 18.5
gini = 0.278
samples = 6
value = [1, 5]
class = Survived
gini = 0.0
samples = 2
value = [2, 0]
class = Perished
Age ≤ 28.85
gini = 0.444
samples = 9
value = [3, 6]
class = Survived
Fare ≤ 7.744
gini = 0.278
samples = 6
value = [1, 5]
class = Survived
Fare ≤ 70.323
gini = 0.5
samples = 6
value = [3, 3]
class = Perished
Fare ≤ 37.0
gini = 0.278
samples = 30
value = [25, 5]
class = Perished
Fare ≤ 71.665
gini = 0.408
samples = 7
value = [5, 2]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
Fare ≤ 7.133
gini = 0.337
samples = 14
value = [11, 3]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
gini = 0.0
samples = 7
value = [7, 0]
class = Perished
Age ≤ 29.85
gini = 0.32
samples = 5
value = [4, 1]
class = Perished
SibSp ≤ 0.5
gini = 0.48
samples = 5
value = [3, 2]
class = Perished
gini = 0.0
samples = 5
value = [5, 0]
class = Perished
Fare ≤ 8.153
gini = 0.153
samples = 24
value = [22, 2]
class = Perished
Fare ≤ 8.081
gini = 0.043
samples = 90
value = [88, 2]
class = Perished
Fare ≤ 11.75
gini = 0.444
samples = 6
value = [2, 4]
class = Survived
gini = 0.0
samples = 2
value = [2, 0]
class = Perished
Fare ≤ 12.938
gini = 0.227
samples = 23
value = [20, 3]
class = Perished
gini = 0.0
samples = 12
value = [12, 0]
class = Perished
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
Fare ≤ 19.5
gini = 0.5
samples = 4
value = [2, 2]
class = Perished
Age ≤ 37.0
gini = 0.198
samples = 18
value = [2, 16]
class = Survived
Fare ≤ 7.635
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
gini = 0.0
samples = 4
value = [0, 4]
class = Survived
Age ≤ 16.4
gini = 0.444
samples = 3
value = [2, 1]
class = Perished
Fare ≤ 17.7
gini = 0.278
samples = 6
value = [1, 5]
class = Survived
Fare ≤ 35.25
gini = 0.363
samples = 21
value = [16, 5]
class = Perished
gini = 0.0
samples = 9
value = [9, 0]
class = Perished
gini = 0.0
samples = 2
value = [2, 0]
class = Perished
Age ≤ 49.0
gini = 0.48
samples = 5
value = [3, 2]
class = Perished
Fare ≤ 7.183
gini = 0.48
samples = 5
value = [3, 2]
class = Perished
gini = 0.375
samples = 4
value = [3, 1]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
Fare ≤ 8.633
gini = 0.375
samples = 4
value = [3, 1]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
Fare ≤ 7.988
gini = 0.298
samples = 11
value = [9, 2]
class = Perished
gini = 0.0
samples = 13
value = [13, 0]
class = Perished
gini = 0.0
samples = 50
value = [50, 0]
class = Perished
Fare ≤ 8.388
gini = 0.095
samples = 40
value = [38, 2]
class = Perished
Pclass ≤ 2.5
gini = 0.5
samples = 4
value = [2, 2]
class = Perished
gini = 0.0
samples = 2
value = [0, 2]
class = Survived
Age ≤ 38.5
gini = 0.105
samples = 18
value = [17, 1]
class = Perished
Age ≤ 36.5
gini = 0.48
samples = 5
value = [3, 2]
class = Perished
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
Age ≤ 25.5
gini = 0.444
samples = 3
value = [1, 2]
class = Survived
gini = 0.0
samples = 10
value = [0, 10]
class = Survived
Age ≤ 39.5
gini = 0.375
samples = 8
value = [2, 6]
class = Survived
Parch ≤ 0.5
gini = 0.444
samples = 3
value = [1, 2]
class = Survived
Age ≤ 47.5
gini = 0.375
samples = 12
value = [9, 3]
class = Perished
gini = 0.0
samples = 7
value = [7, 0]
class = Perished
Age ≤ 27.0
gini = 0.444
samples = 3
value = [1, 2]
class = Survived
gini = 0.0
samples = 2
value = [2, 0]
class = Perished
Age ≤ 22.0
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
gini = 0.0
samples = 7
value = [7, 0]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
Age ≤ 21.75
gini = 0.375
samples = 4
value = [3, 1]
class = Perished
Fare ≤ 7.913
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
gini = 0.0
samples = 2
value = [2, 0]
class = Perished
Age ≤ 20.5
gini = 0.198
samples = 9
value = [8, 1]
class = Perished
Age ≤ 20.0
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
Embarked ≤ 0.5
gini = 0.05
samples = 39
value = [38, 1]
class = Perished
Fare ≤ 7.89
gini = 0.444
samples = 3
value = [1, 2]
class = Survived
gini = 0.0
samples = 13
value = [13, 0]
class = Perished
Age ≤ 39.5
gini = 0.32
samples = 5
value = [4, 1]
class = Perished
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
Age ≤ 40.5
gini = 0.444
samples = 3
value = [2, 1]
class = Perished
gini = 0.0
samples = 2
value = [0, 2]
class = Survived
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
Fare ≤ 24.5
gini = 0.245
samples = 7
value = [1, 6]
class = Survived
Age ≤ 43.0
gini = 0.444
samples = 3
value = [1, 2]
class = Survived
Fare ≤ 26.125
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
Fare ≤ 30.848
gini = 0.18
samples = 10
value = [9, 1]
class = Perished
Embarked ≤ 0.5
gini = 0.375
samples = 4
value = [3, 1]
class = Perished
gini = 0.0
samples = 6
value = [6, 0]
class = Perished
gini = 0.0
samples = 3
value = [3, 0]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
SibSp ≤ 0.5
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
Fare ≤ 7.798
gini = 0.444
samples = 3
value = [2, 1]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
gini = 0.0
samples = 2
value = [2, 0]
class = Perished
Fare ≤ 14.158
gini = 0.278
samples = 6
value = [5, 1]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
gini = 0.0
samples = 5
value = [5, 0]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
gini = 0.0
samples = 4
value = [4, 0]
class = Perished
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived

Overfitting

Clearly, we're overfitting the data - 98% accuracy on the training data and only ~75% on the test data. Yet, we've created a complicated tree.

Decision trees are notorious for overfitting the data. There are two parameters that help us reign in overfitting:

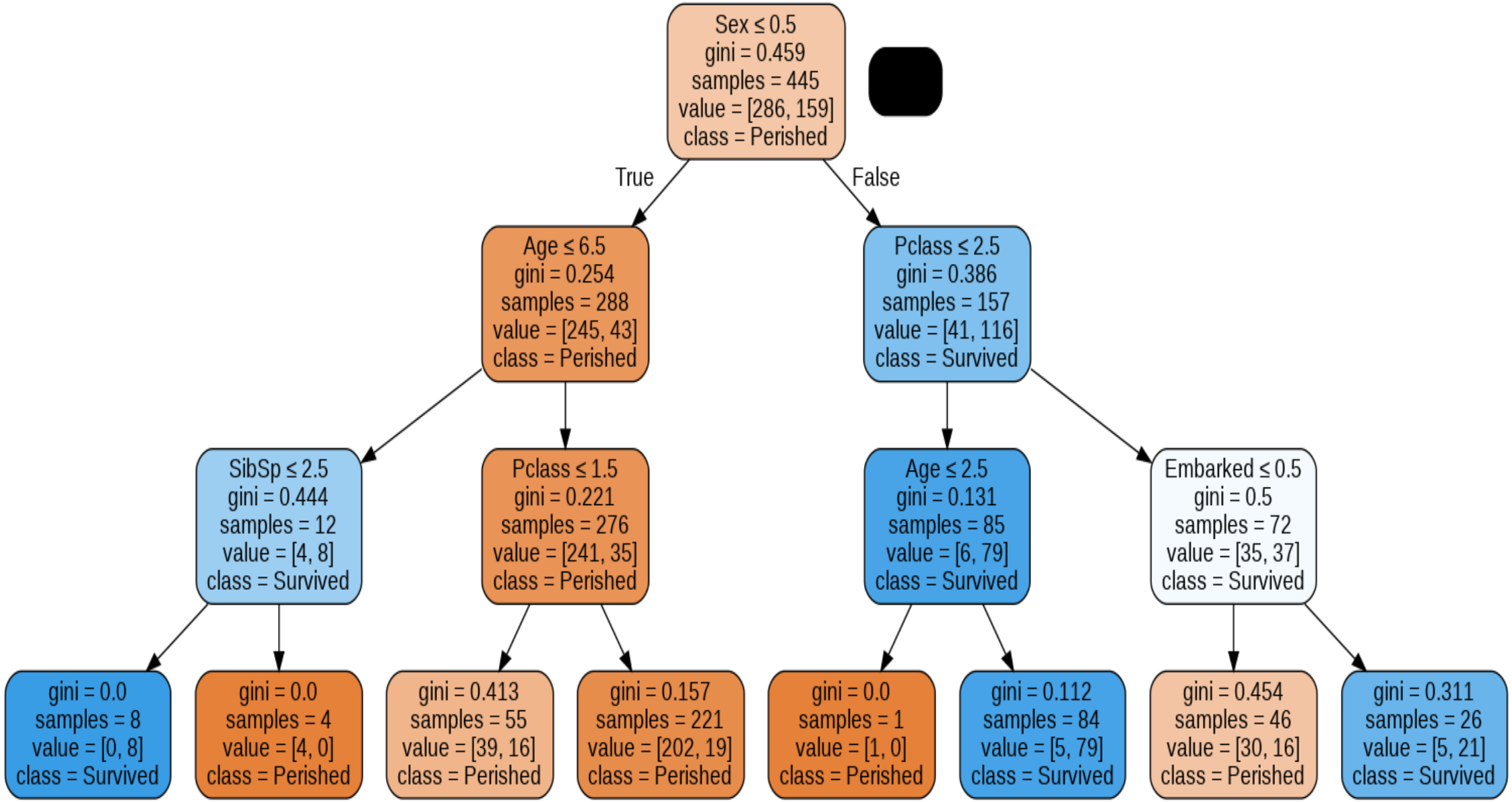
- **max_depth:** The maximum depth of the tree. If this is not set, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- **min_samples_split:** The minimum number of samples required to split an internal node: If the value is an integer, then consider `min_samples_split` as the minimum number. If it is float, then `min_samples_split` is a percentage and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Limiting Depth

```
# Limiting Depth!
decisionTree = tree.DecisionTreeClassifier(max_depth=3)

decisionTree = decisionTree.fit(XTrain, yTrain)

y_pred_train = decisionTree.predict(XTrain)
print('Accuracy on training data= ', metrics.accuracy_score(y_true = yTrain, y_pred = y_pred_train))

y_pred = decisionTree.predict(XTest)
print('Accuracy on test data= ', metrics.accuracy_score(y_true = yTest, y_pred = y_pred))
renderTree(decisionTree, all_features)
```

Accuracy on training data= 0.8629213483146068

Accuracy on test data= 0.7937219730941704

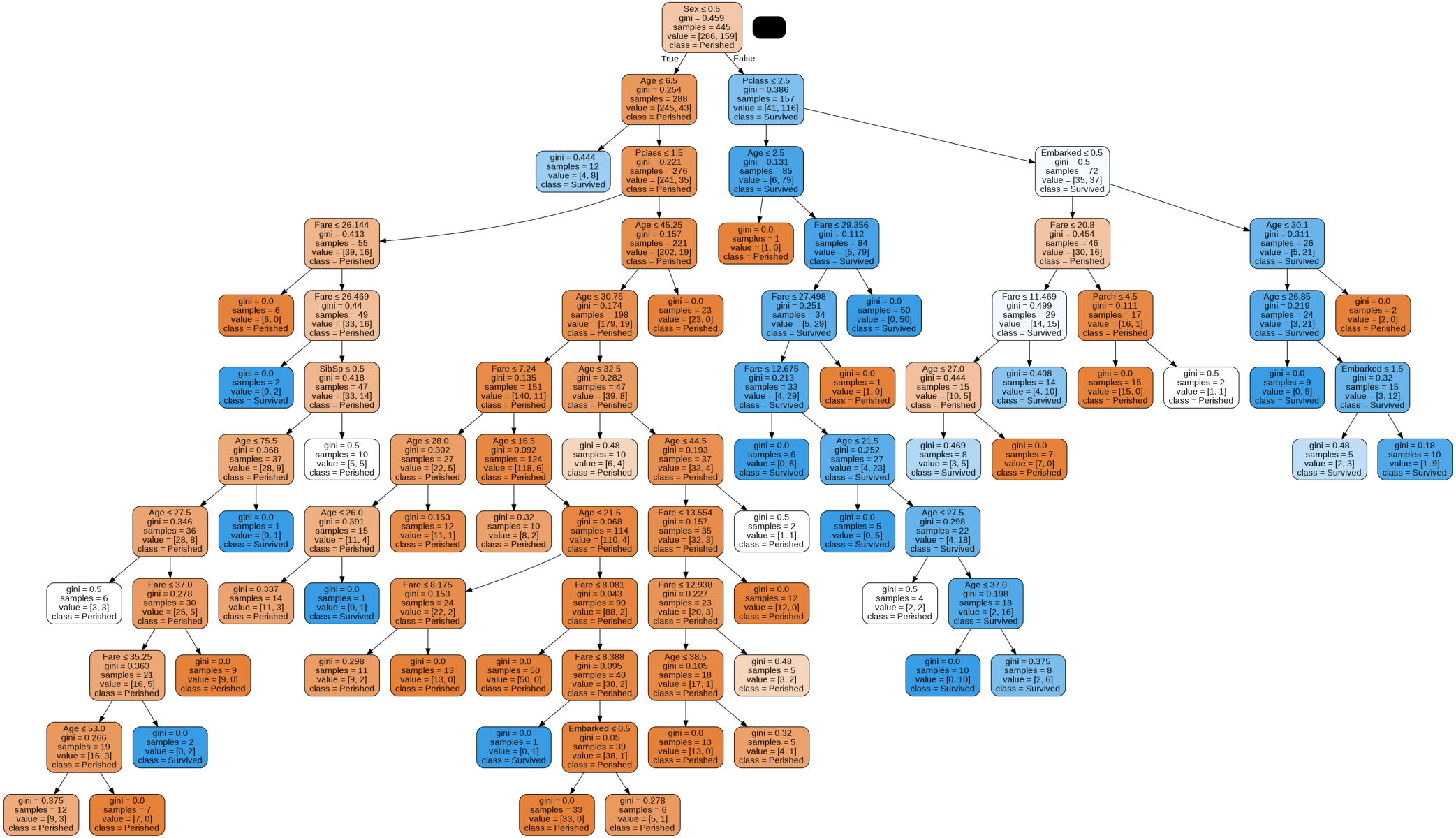

Sex ≤ 0.5
gini = 0.459
samples = 445
value = [286, 159]
class = Perished
True
False
Age ≤ 6.5
gini = 0.254
samples = 288
value = [245, 43]
class = Perished
Pclass ≤ 2.5
gini = 0.386
samples = 157
value = [41, 116]
class = Survived
SibSp ≤ 2.5
gini = 0.444
samples = 12
value = [4, 8]
class = Survived
Pclass ≤ 1.5
gini = 0.221
samples = 276
value = [241, 35]
class = Perished
Age ≤ 2.5
gini = 0.131
samples = 85
value = [6, 79]
class = Survived
Embarked ≤ 0.5
gini = 0.5
samples = 72
value = [35, 37]
class = Survived
gini = 0.0
samples = 8
value = [0, 8]
class = Survived
gini = 0.0
samples = 4
value = [4, 0]
class = Perished
gini = 0.413
samples = 55
value = [39, 16]
class = Perished
gini = 0.157
samples = 221
value = [202, 19]
class = Perished
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
gini = 0.112
samples = 84
value = [5, 79]
class = Survived
gini = 0.454
samples = 46
value = [30, 16]
class = Perished
gini = 0.311
samples = 26
value = [5, 21]
class = Survived

Lmiting the minimum samples used to split

```
# Lmiting the minimum samples used to split.
decisionTree = tree.DecisionTreeClassifier(min_samples_split=15)

decisionTree = decisionTree.fit(XTrain, yTrain)

y_pred_train = decisionTree.predict(XTrain)
print('Accuracy on training data= ', metrics.accuracy_score(y_true = yTrain, y_pred = y_pred_train))

y_pred = decisionTree.predict(XTest)
print('Accuracy on test data= ', metrics.accuracy_score(y_true = yTest, y_pred = y_pred))
renderTree(decisionTree, all_features)
```

Accuracy on training data= 0.8943820224719101

Accuracy on test data= 0.7780269058295964

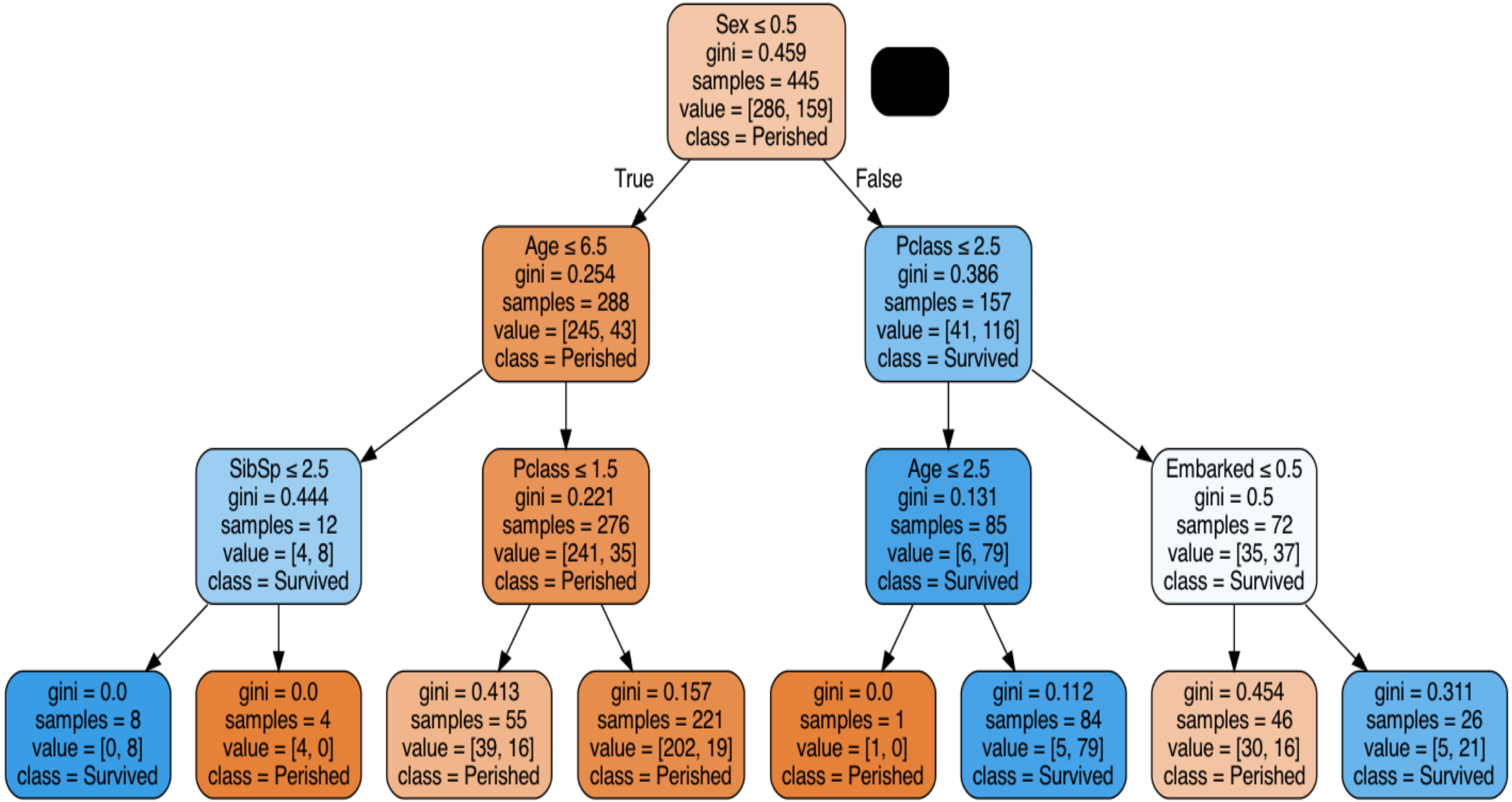

Sex ≤ 0.5
gini = 0.459
samples = 445
value = [286, 159]
class = Perished
True
False
Age ≤ 6.5
gini = 0.254
samples = 288
value = [245, 43]
class = Perished
Pclass ≤ 2.5
gini = 0.386
samples = 157
value = [41, 116]
class = Survived
gini = 0.444
samples = 12
value = [4, 8]
class = Survived
Pclass ≤ 1.5
gini = 0.221
samples = 276
value = [241, 35]
class = Perished
Age ≤ 2.5
gini = 0.131
samples = 85
value = [6, 79]
class = Survived
Embarked ≤ 0.5
gini = 0.5
samples = 72
value = [35, 37]
class = Survived
Fare ≤ 26.144
gini = 0.413
samples = 55
value = [39, 16]
class = Perished
Age ≤ 45.25
gini = 0.157
samples = 221
value = [202, 19]
class = Perished
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
Fare ≤ 29.356
gini = 0.112
samples = 84
value = [5, 79]
class = Survived
Fare ≤ 20.8
gini = 0.454
samples = 46
value = [30, 16]
class = Perished
Age ≤ 30.1
gini = 0.311
samples = 26
value = [5, 21]
class = Survived
gini = 0.0
samples = 6
value = [6, 0]
class = Perished
Fare ≤ 26.469
gini = 0.44
samples = 49
value = [33, 16]
class = Perished
Age ≤ 30.75
gini = 0.174
samples = 198
value = [179, 19]
class = Perished
gini = 0.0
samples = 23
value = [23, 0]
class = Perished
Fare ≤ 27.498
gini = 0.251
samples = 34
value = [5, 29]
class = Survived
gini = 0.0
samples = 50
value = [0, 50]
class = Survived
Fare ≤ 11.469
gini = 0.499
samples = 29
value = [14, 15]
class = Survived
Parch ≤ 4.5
gini = 0.111
samples = 17
value = [16, 1]
class = Perished
Age ≤ 26.85
gini = 0.219
samples = 24
value = [3, 21]
class = Survived
gini = 0.0
samples = 2
value = [2, 0]
class = Perished
gini = 0.0
samples = 2
value = [0, 2]
class = Survived
SibSp ≤ 0.5
gini = 0.418
samples = 47
value = [33, 14]
class = Perished
Fare ≤ 7.24
gini = 0.135
samples = 151
value = [140, 11]
class = Perished
Age ≤ 32.5
gini = 0.282
samples = 47
value = [39, 8]
class = Perished
Fare ≤ 12.675
gini = 0.213
samples = 33
value = [4, 29]
class = Survived
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
Age ≤ 27.0
gini = 0.444
samples = 15
value = [10, 5]
class = Perished
gini = 0.408
samples = 14
value = [4, 10]
class = Survived
gini = 0.0
samples = 15
value = [15, 0]
class = Perished
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
gini = 0.0
samples = 9
value = [0, 9]
class = Survived
Embarked ≤ 1.5
gini = 0.32
samples = 15
value = [3, 12]
class = Survived
Age ≤ 75.5
gini = 0.368
samples = 37
value = [28, 9]
class = Perished
gini = 0.5
samples = 10
value = [5, 5]
class = Perished
Age ≤ 28.0
gini = 0.302
samples = 27
value = [22, 5]
class = Perished
Age ≤ 16.5
gini = 0.092
samples = 124
value = [118, 6]
class = Perished
gini = 0.48
samples = 10
value = [6, 4]
class = Perished
Age ≤ 44.5
gini = 0.193
samples = 37
value = [33, 4]
class = Perished
gini = 0.0
samples = 6
value = [0, 6]
class = Survived
Age ≤ 21.5
gini = 0.252
samples = 27
value = [4, 23]
class = Survived
gini = 0.469
samples = 8
value = [3, 5]
class = Survived
gini = 0.0
samples = 7
value = [7, 0]
class = Perished
gini = 0.48
samples = 5
value = [2, 3]
class = Survived
gini = 0.18
samples = 10
value = [1, 9]
class = Survived
Age ≤ 27.5
gini = 0.346
samples = 36
value = [28, 8]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
Age ≤ 26.0
gini = 0.391
samples = 15
value = [11, 4]
class = Perished
gini = 0.153
samples = 12
value = [11, 1]
class = Perished
gini = 0.32
samples = 10
value = [8, 2]
class = Perished
Age ≤ 21.5
gini = 0.068
samples = 114
value = [110, 4]
class = Perished
Fare ≤ 13.554
gini = 0.157
samples = 35
value = [32, 3]
class = Perished
gini = 0.5
samples = 2
value = [1, 1]
class = Perished
gini = 0.0
samples = 5
value = [0, 5]
class = Survived
Age ≤ 27.5
gini = 0.298
samples = 22
value = [4, 18]
class = Survived
gini = 0.5
samples = 6
value = [3, 3]
class = Perished
Fare ≤ 37.0
gini = 0.278
samples = 30
value = [25, 5]
class = Perished
gini = 0.337
samples = 14
value = [11, 3]
class = Perished
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
Fare ≤ 8.175
gini = 0.153
samples = 24
value = [22, 2]
class = Perished
Fare ≤ 8.081
gini = 0.043
samples = 90
value = [88, 2]
class = Perished
Fare ≤ 12.938
gini = 0.227
samples = 23
value = [20, 3]
class = Perished
gini = 0.0
samples = 12
value = [12, 0]
class = Perished
gini = 0.5
samples = 4
value = [2, 2]
class = Perished
Age ≤ 37.0
gini = 0.198
samples = 18
value = [2, 16]
class = Survived
Fare ≤ 35.25
gini = 0.363
samples = 21
value = [16, 5]
class = Perished
gini = 0.0
samples = 9
value = [9, 0]
class = Perished
gini = 0.298
samples = 11
value = [9, 2]
class = Perished
gini = 0.0
samples = 13
value = [13, 0]
class = Perished
gini = 0.0
samples = 50
value = [50, 0]
class = Perished
Fare ≤ 8.388
gini = 0.095
samples = 40
value = [38, 2]
class = Perished
Age ≤ 38.5
gini = 0.105
samples = 18
value = [17, 1]
class = Perished
gini = 0.48
samples = 5
value = [3, 2]
class = Perished
gini = 0.0
samples = 10
value = [0, 10]
class = Survived
gini = 0.375
samples = 8
value = [2, 6]
class = Survived
Age ≤ 53.0
gini = 0.266
samples = 19
value = [16, 3]
class = Perished
gini = 0.0
samples = 2
value = [0, 2]
class = Survived
gini = 0.0
samples = 1
value = [0, 1]
class = Survived
Embarked ≤ 0.5
gini = 0.05
samples = 39
value = [38, 1]
class = Perished
gini = 0.0
samples = 13
value = [13, 0]
class = Perished
gini = 0.32
samples = 5
value = [4, 1]
class = Perished
gini = 0.375
samples = 12
value = [9, 3]
class = Perished
gini = 0.0
samples = 7
value = [7, 0]
class = Perished
gini = 0.0
samples = 33
value = [33, 0]
class = Perished
gini = 0.278
samples = 6
value = [5, 1]
class = Perished

Limiting both Depth and the minimum samples

```
# Limiting Both

decisionTree = tree.DecisionTreeClassifier(max_depth=3, min_samples_split=10)
decisionTree = decisionTree.fit(XTrain, yTrain)

y_pred_train = decisionTree.predict(XTrain)
print('Accuracy on training data= ', metrics.accuracy_score(y_true = yTrain, y_pred = y_pred_train))

y_pred = decisionTree.predict(XTest)
print('Accuracy on test data= ', metrics.accuracy_score(y_true = yTest, y_pred = y_pred))
renderTree(decisionTree, all_features)
```

Accuracy on training data= 0.8629213483146068

Accuracy on test data= 0.7937219730941704


Sex ≤ 0.5
gini = 0.459
samples = 445
value = [286, 159]
class = Perished
True
False
Age ≤ 6.5
gini = 0.254
samples = 288
value = [245, 43]
class = Perished
Pclass ≤ 2.5
gini = 0.386
samples = 157
value = [41, 116]
class = Survived
SibSp ≤ 2.5
gini = 0.444
samples = 12
value = [4, 8]
class = Survived
Pclass ≤ 1.5
gini = 0.221
samples = 276
value = [241, 35]
class = Perished
Age ≤ 2.5
gini = 0.131
samples = 85
value = [6, 79]
class = Survived
Embarked ≤ 0.5
gini = 0.5
samples = 72
value = [35, 37]
class = Survived
gini = 0.0
samples = 8
value = [0, 8]
class = Survived
gini = 0.0
samples = 4
value = [4, 0]
class = Perished
gini = 0.413
samples = 55
value = [39, 16]
class = Perished
gini = 0.157
samples = 221
value = [202, 19]
class = Perished
gini = 0.0
samples = 1
value = [1, 0]
class = Perished
gini = 0.112
samples = 84
value = [5, 79]
class = Survived
gini = 0.454
samples = 46
value = [30, 16]
class = Perished
gini = 0.311
samples = 26
value = [5, 21]
class = Survived

It looks like both, the minimum number of samples for splitting and the maximum depth help with overfitting and we achieve a 79-80% accuracy rate. That doesn't sound much better than just gender alone, but 4% improvement is a lot in classification.

Also, our last model is fairly simple yet quite accurate. The main point seems to be:

- Sex is the dominant factor at the root of the tree
- For females, if you're in class 1 or 2 you're almost sure to survive
- For males, if you were younger than 6.5 years old, you had a chance to survive.
- Also note that there are branches that predict the same thing, but with different certainty. For example, in the male / adult category, if you were in "first class", you still were likely to die, but less likely than in second or third.

Of course, here, we made these decisions on how to limit the tree manually. What would be a good approach to do this automatically?

Discussion

Advantages of decision trees

- Decision trees are simple to explain and interpret, and interpretability is a major issue in many applications. Think about credit decisions, or medical decisions.
- There is a nice graphical display for trees

Disadvantages of decision trees

- Decision trees generally don't have the predictive accuracy of other approaches as they tend to overfit the data.
- Decision trees are non-robust, *i.e.*, sensitive to small changes in the data.

Ensemble Methods based on Decision Trees

Ensemble Methods (Ensemble Learning) use multiple algorithms at the same time and then come to a consensus of a predictive label.

Bagging

One such method is **Bagging** (Bootstrap Aggregating). The idea of bagging is to generate several trained models (e.g., decision trees) based on subsets of the data and let the decision trees vote to arrive at a prediction.
Commomly the subset is chosen through bootstrapping, i.e., random sampling with replacement.
Since averaging a set of observations reduces variances (think CLT), this increases the predictive accuracy of the method.

Boosting

Boosting is similar to bagging, except that the trees are grown sequentially and are trained specifically to address previously mis-classified items.

When applied to decision trees, these methods are called Random Forests. Generally, random forests combine multiple decision trees that were generated with some randomness and let them vote on the result. Here the randomness comes from choosing a random sample of the prediction variables to build each tree.

Random Forests

An example with a random forest that uses *300* trees and bootstraping.

```
from sklearn.ensemble import RandomForestClassifier

forest =  RandomForestClassifier(bootstrap=True, n_estimators=300, random_state=0)

trained_forest = forest.fit(XTrain, yTrain)

y_pred_train = trained_forest.predict(XTrain)
print('Accuracy on training data= ', metrics.accuracy_score(y_true = yTrain, y_pred = y_pred_train))

y_pred = trained_forest.predict(XTest)
print('Accuracy on test data= ', metrics.accuracy_score(y_true = yTest, y_pred = y_pred))
```

Accuracy on training data= 0.9887640449438202

Accuracy on test data= 0.7914798206278026

Ensemble methods are generally of higher accuracy, but also take a lot longer to train.