Python for Machine Learning and Data Analysis

Dr. Handan Liu

h.liu@northeastern.edu

Northeastern University

Spring 2019

Machine Learning 04

Lecture 10

Content

- Support Vector Machines
    - Motivating Support Vector Machines
    - Support Vector Machines: Maximizing the Margin
    - Example: Face Recognition
- Decision Trees and Random Forests
    - Motivating Random Forests: Decision Trees
    - Ensembles of Estimators: Random Forests
    - Random Forest Regression
    - Example: Random Forest for Classifying Digits

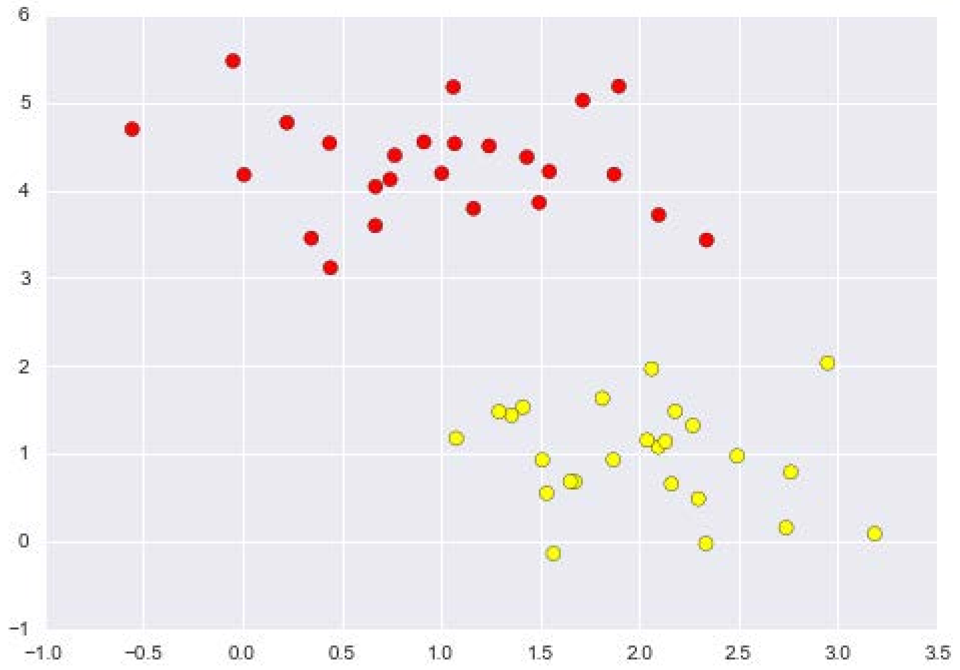
Support Vector Machines

- Support vector machines (SVMs) are a particularly powerful and flexible class of supervised algorithms for both classification and regression.
- In this section, we will develop the intuition behind support vector machines and their use in classification problems.

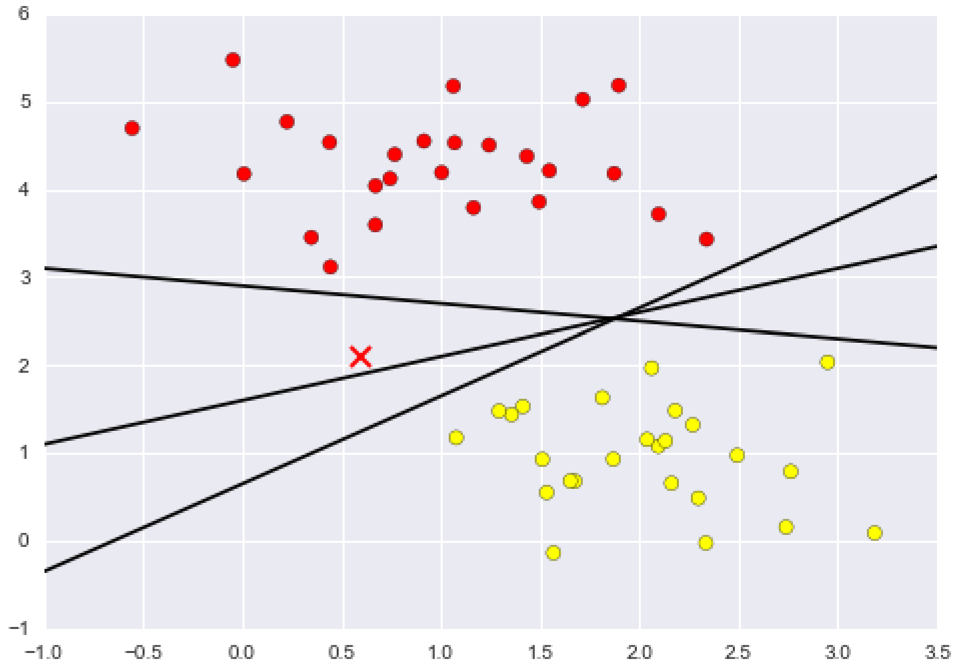
1. Motivating Support Vector Machines

- As part of our discussion of Bayesian classification, we learned:
  - a simple model describing the distribution of each underlying class, and
  - used these generative models to probabilistically determine labels for new points.
  - That was an example of generative classification;
- Here we will consider instead discriminative classification:
  - Rather than modeling each class, we simply find a line or curve (in two dimensions) or manifold (in multiple dimensions) that divides the classes from each other.

As an example of this, consider the simple case of a classification task, in which the two classes of points are well separated:



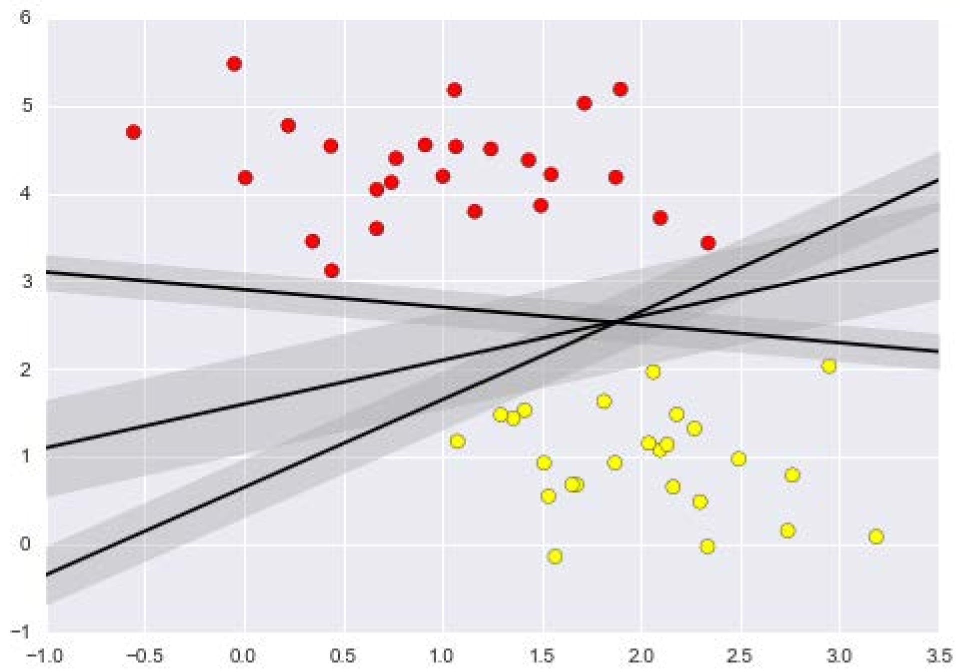
- A linear discriminative classifier would attempt to draw a straight line separating the two sets of data, and thereby create a model for classification.
- For two dimensional data like that shown here, this is a task we could do by hand. But immediately we see a problem: there is more than one possible dividing line that can perfectly discriminate between the two classes! These are three very different separators which perfectly discriminate between these samples.



- Depending on which you choose, a new data point (e.g., the one marked by the "X" in this plot) will be assigned a different label!
- Evidently our simple intuition of "drawing a line between classes" is not enough, and we need to think a bit deeper.

2. Support Vector Machines: Maximizing the Margin

- Support vector machines offer one way to improve on this.
- The intuition is this: rather than simply drawing a zero-width line between the classes, we can draw around each line a margin of some width, up to the nearest point.
- Here is an example of how this might look:



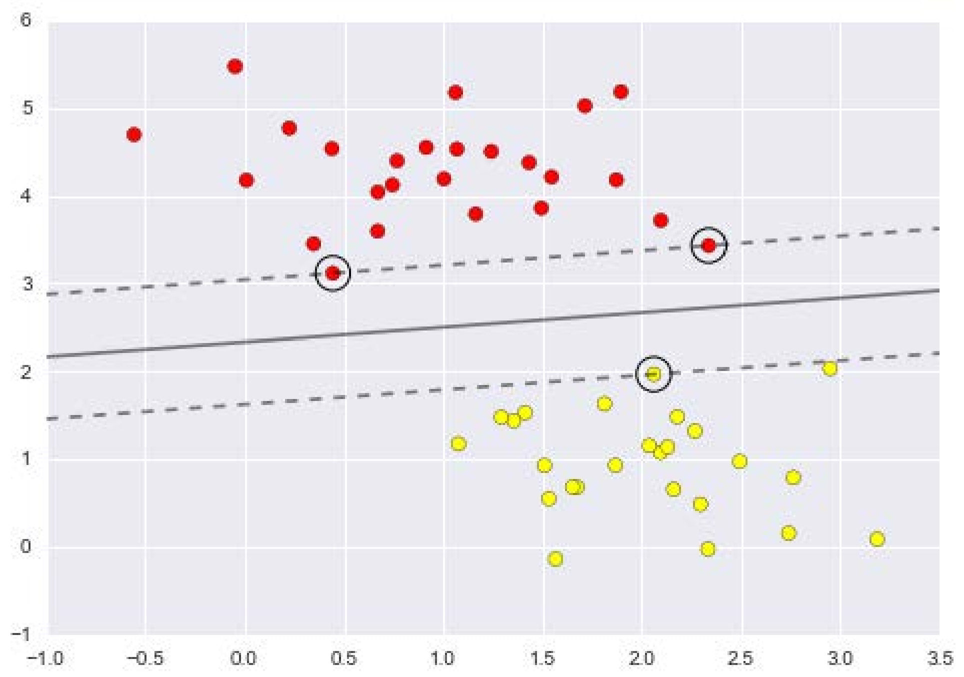
Maximizing the margin of this line is important one we will choose as the optimal model in support vector machines.

2.1 Fitting a support vector machine

- Use Scikit-Learn's support vector classifier to train an SVM model on this data.
- Use a linear kernel and set the *C* parameter to a very large number.

```
from sklearn.svm import SVC        # "Support vector classifier"
model = SVC(kernel='linear', C=1E10)
model.fit(X, y)
```

- Plot SVM decision boundaries.



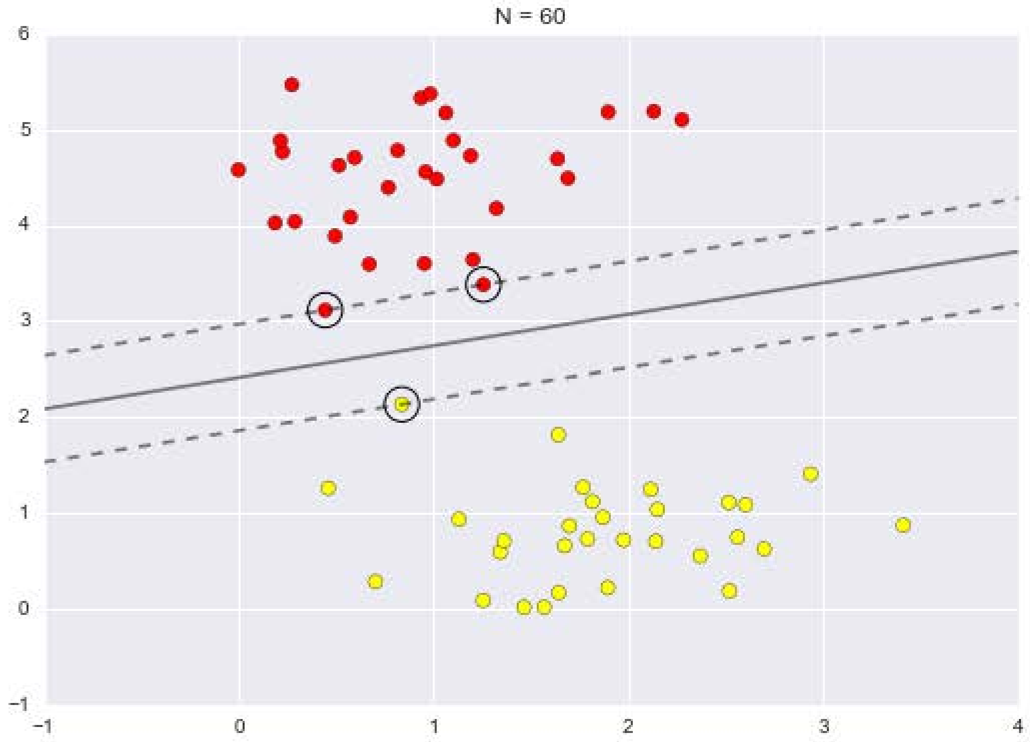
- This is the dividing line that maximizes the margin between the two sets of points.
- Notice that a few of the training points just touch the margin: they are indicated by the black circles in this figure.
- These points are the pivotal elements of this fit, and are known as **the support vectors**, and give the algorithm its name.

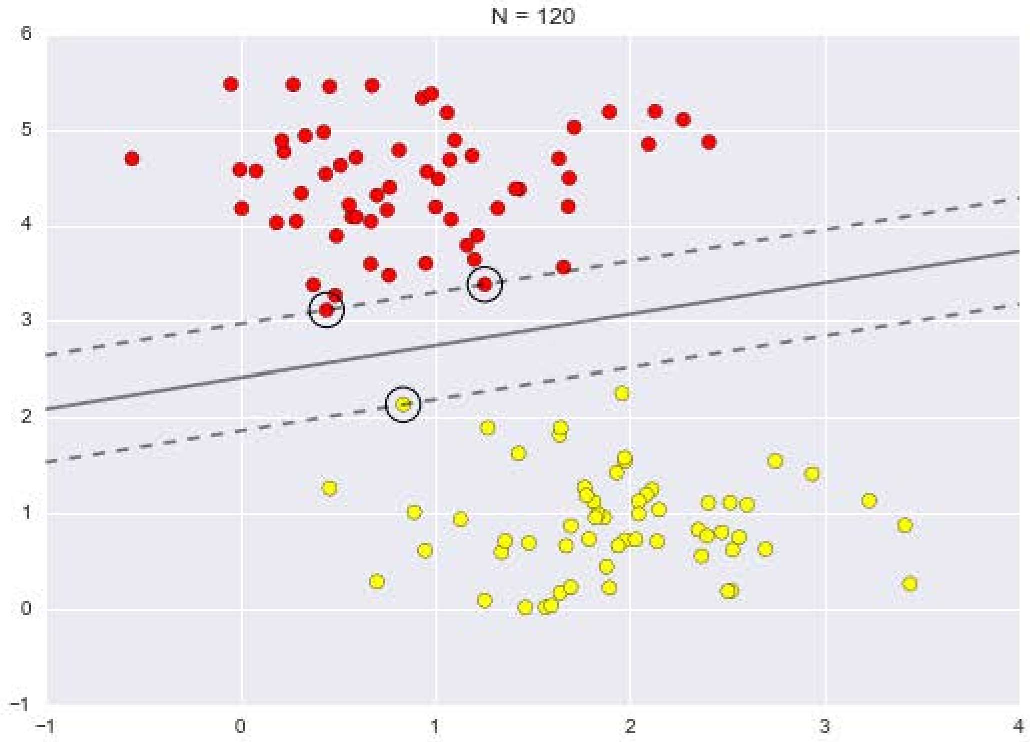
In Scikit-Learn, the identity of these points are stored in the support_vectors_ attribute of the classifier:

```
model.support_vectors_
array([[ 0.44359863,  3.11530945],
       [ 2.33812285,  3.43116792],
       [ 2.06156753,  1.96918596]])
```

- A key to this classifier's success is that for the fit, only the position of the support vectors matter; any points further from the margin which are on the correct side do not modify the fit!
- Technically, this is because these points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.

For example, plot the model learned from the first 60 points and first 120 points of this dataset:





In the left panel, the model and the support vectors are for 60 training points.

In the right panel, the number of training points are doubled, but the model has not changed: the three support vectors from the left panel are still the support vectors from the right panel.

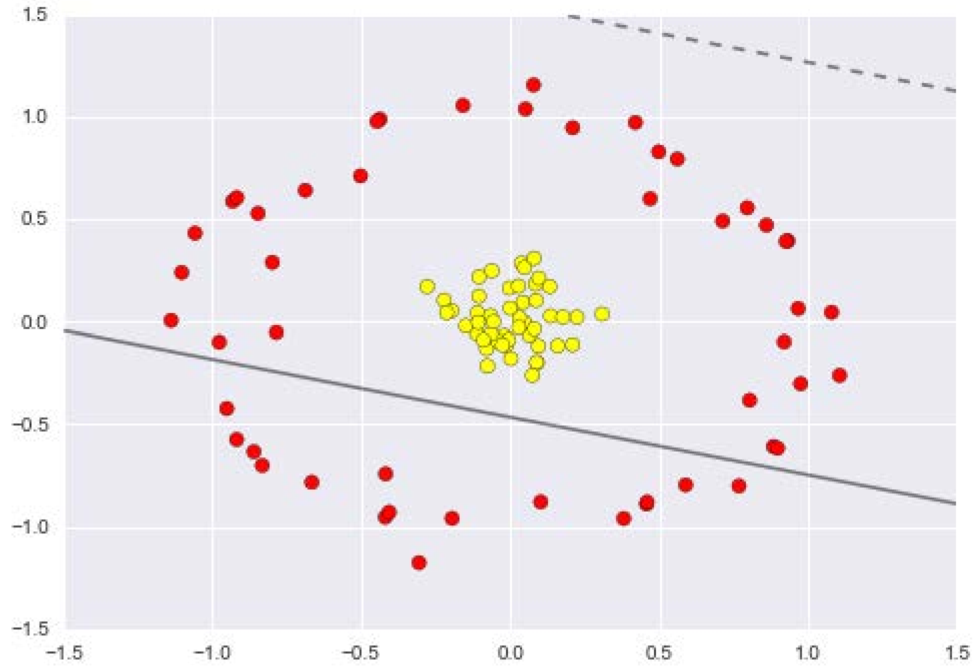
This insensitivity to the exact behavior of distant points is one of the strengths of the SVM model.

2.2 Beyond linear boundaries: Kernel SVM

Where SVM becomes extremely powerful is when it is combined with kernels.

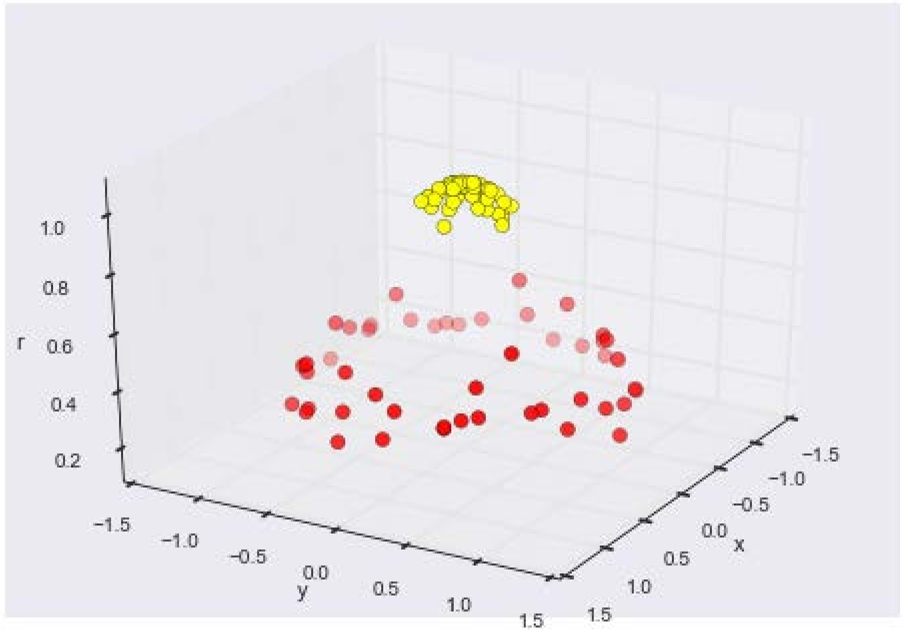
- We have seen a version of kernels before, in the basis function regressions in last class. There we projected our data into higher-dimensional space defined by polynomials and Gaussian basis functions, and thereby were able to fit for nonlinear relationships with a linear classifier.

  - In SVM models, we can use a version of the same idea. To motivate the need for kernels, let's look at some data that is not linearly separable:



It is clear that no linear discrimination will ever be able to separate this data. But we can draw a lesson from the basis function regressions, and think about how we might project the data into a higher dimension such that a linear separator would be sufficient.

For example, one simple projection would be used to compute a radial basis function centered on the middle clump; visualize this extra data dimension using a three-dimensional plot:



With this additional dimension, the data becomes trivially linearly separable, by drawing a separating plane at, say, r=0.7.

Choose and carefully tune the projection: if we had not centered our radial basis function in the right location, we would not have seen such clean, linearly separable results.

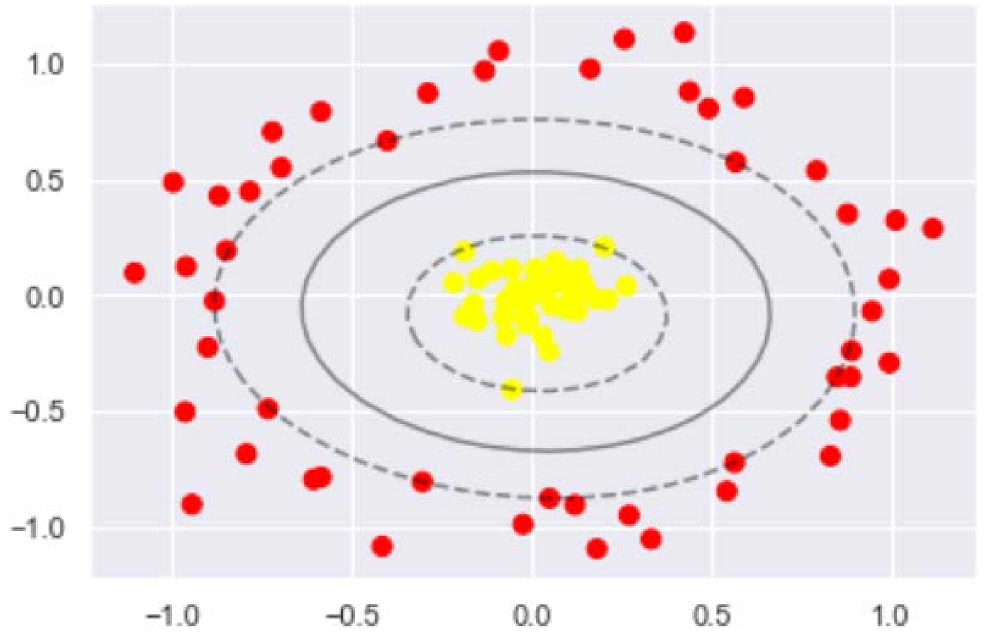
In general, the need to make such a choice is a problem: we would like to somehow automatically find the best basis functions to use.

- One strategy to this end is to compute a basis function centered at every point in the dataset, and let the SVM algorithm sift through the results.
- This type of basis function transformation is known as *a kernel transformation*, as it is based on a similarity relationship (or kernel) between each pair of points.

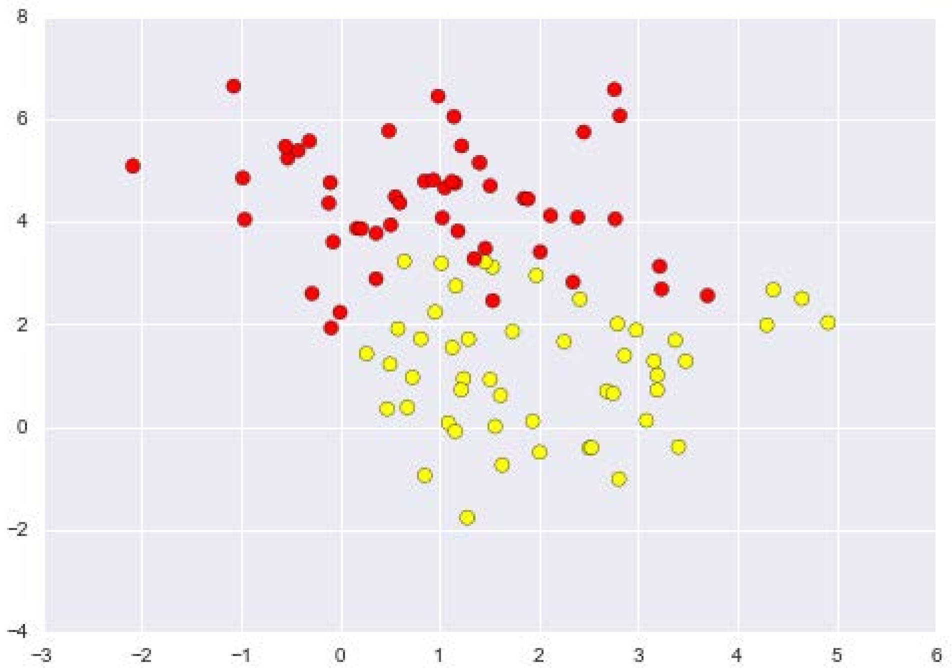
- This strategy is projecting *N* points into *N* dimensions. A potential problem with this strategy is that it might become very computationally intensive as *N* grows large.
- However, because of a neat little procedure known as the *kernel trick*, a fit on *kernel-transformed* data can be done implicitly — that is, without ever building the full *N*-dimensional representation of the kernel projection!
- This *kernel trick* is built into the SVM, and is one of the reasons the method is so powerful.
- In Scikit-Learn, we can apply kernelized SVM simply by changing our linear kernel to an RBF (radial basis function) kernel, using the kernel model hyperparameter:

Using this kernelized support vector machine, we learn a suitable nonlinear decision boundary.

This kernel transformation strategy is used often in machine learning to turn fast linear methods into fast nonlinear methods, especially for models in which the kernel trick can be used.



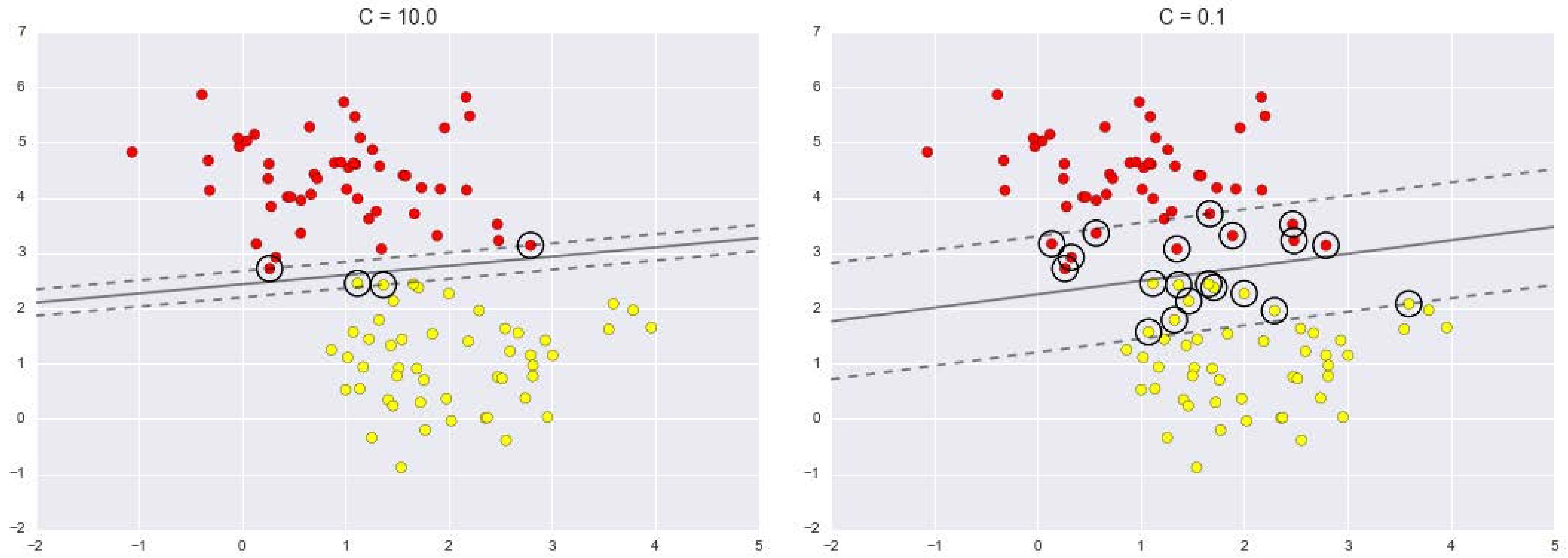
Our discussion thus far has centered around very clean datasets, in which a perfect decision boundary exists. But what if your data has some amount of overlap?



2.3 Tuning the SVM: Softening Margins

- To handle this case, the SVM implementation has a bit of a fudge-factor which "softens" the margin: that is, it allows some of the points to creep into the margin if that allows a better fit.
- The hardness of the margin is controlled by a tuning parameter, most often known as *C*.
- For very large *C*, the margin is hard, and points cannot lie in it.
- For smaller *C*, the margin is softer, and can grow to encompass some points.

The plot shown below gives a visual picture of how a changing *C* parameter affects the final fit, via the softening of the margin:



The optimal value of the *C* parameter will depend on your dataset, and should be tuned using cross-validation or a similar procedure.

Example: Face Recognition

- As an example of support vector machines in action: the facial recognition problem.
- Use the Labeled Faces in the Wild dataset, which consists of several thousand collated photos of various public figures.
- A fetcher for the dataset is built into Scikit-Learn:

```
from sklearn.datasets import fetch_lfw_people
faces = fetch_lfw_people(min_faces_per_person=60)
```

- Each image contains [62×47] or nearly 3,000 pixels. We could proceed by simply using each pixel value as a feature, but often it is more effective to use some sort of preprocessor to extract more meaningful features;
- Use a principal component analysis (PCA) to extract 150 fundamental components to feed into our support vector machine classifier; and do this most straightforwardly by packaging the preprocessor and the classifier into a single pipeline:

- For the sake of testing our classifier output, we will split the data into a training and testing set:
- Finally, we can use a grid search cross-validation to explore combinations of parameters. Here we will adjust
  - *C* -- controls the margin hardness, and
  - gamma -- controls the size of the radial basis function kernel, and
  - determine the best model:
- The optimal values fall toward the middle of our grid; if they fell at the edges, we would want to expand the grid to make sure we have found the true optimum.

- Now with this cross-validated model, predict the labels for the test data, which the model has not yet seen.
- Take a look at a few of the test images along with their predicted values.
- Out of this small sample, our optimal estimator mislabeled only a single face (Bush’s face in the bottom row was mislabeled as Blair).
- To get a better sense of our estimator's performance using the classification report, which lists recovery statistics label by label.
- We might also display the confusion matrix between these classes. This helps us get a sense of which labels are likely to be confused by the estimator.

- For a real-world facial recognition task, in which the photos do not come pre-cropped into nice grids, the only difference in the facial classification scheme is the feature selection:
  - You would need to use a more sophisticated algorithm to find the faces, and
  - Extract features that are independent of the pixellation.
  - For this kind of application, one good option is to make use of OpenCV, which includes pre-trained implementations of state-of-the-art feature extraction tools for images in general and faces in particular.

Support Vector Machine Summary

We have seen here a brief intuitive introduction to the principals behind support vector machines. These methods are a powerful classification method for a number of reasons:

- Their dependence on relatively few support vectors means that they are very compact models, and take up very little memory.
- Once the model is trained, the prediction phase is very fast.
- Because they are affected only by points near the margin, they work well with high-dimensional data — even data with more dimensions than samples, which is a challenging regime for other algorithms.
- Their integration with kernel methods makes them very versatile, able to adapt to many types of data.

However, SVMs have several disadvantages as well:

- For large numbers of training samples, this computational cost can be prohibitive.
- The results are strongly dependent on a suitable choice for the softening parameter *C*. This must be carefully chosen via cross-validation, which can be expensive as datasets grow in size.
- The results do not have a direct probabilistic interpretation. This can be estimated via an internal cross-validation (see the probability parameter of SVC), but this extra estimation is costly.

With those traits in mind, I generally only turn to SVMs once other simpler, faster, and less tuning-intensive methods have been shown to be insufficient for my needs. But, if you have the CPU cycles to commit to training and cross-validating an SVM on your data, the method can lead to excellent results.

Decision Trees and Random Forests

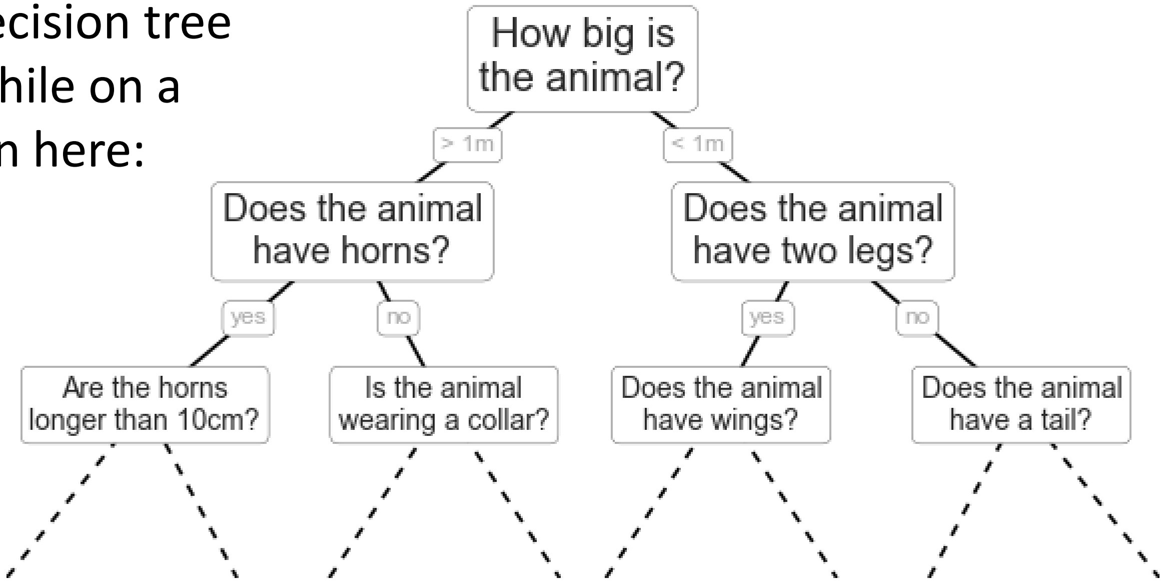
Motivating another powerful algorithm — a non-parametric algorithm called random forests.

Motivating Random Forests: Decision Trees

- Random forests are an example of an ensemble learner built on decision trees. For this reason we'll start by discussing decision trees themselves.

- Decision trees are extremely intuitive ways to classify or label objects: you simply ask a series of questions designed to zero-in on the classification.

For example, if you wanted to build a decision tree to classify an animal you come across while on a hike, you might construct the one shown here:

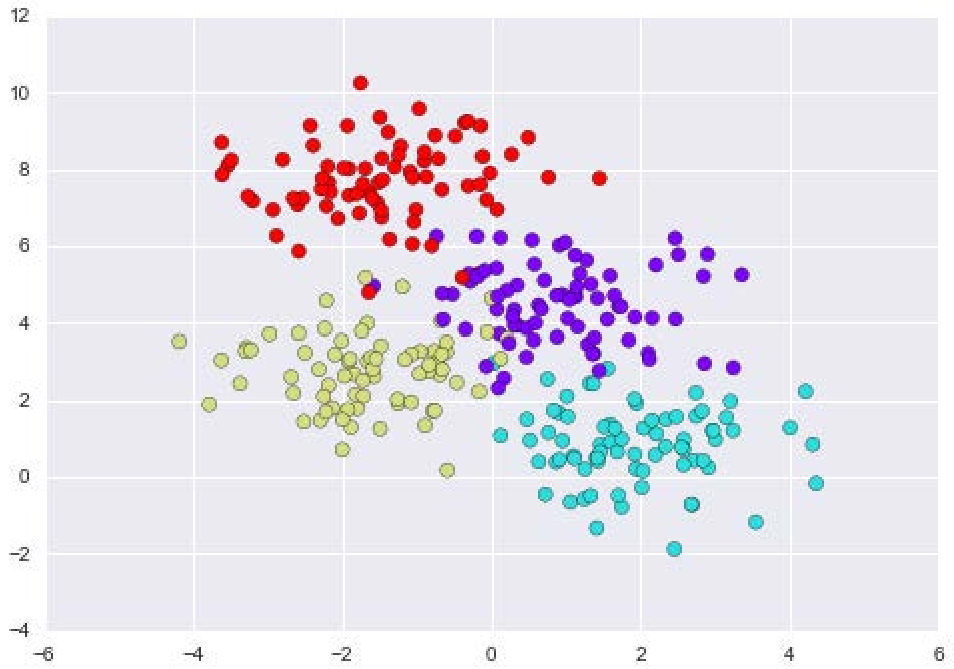

How big is the animal?
> 1m
< 1m
Does the animal have horns?
Does the animal have two legs?
yes
no
yes
no
Are the horns longer than 10cm?
Is the animal wearing a collar?
Does the animal have wings?
Does the animal have a tail?

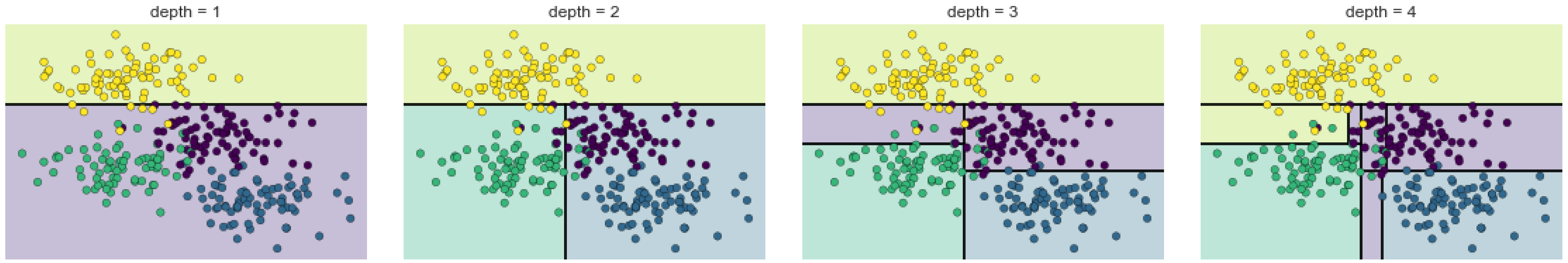
- The binary splitting makes this extremely efficient: in a well-constructed tree, each question will cut the number of options by approximately half, very quickly narrowing the options even among a large number of classes.
- The trick comes in deciding which questions to ask at each step.
- In machine learning implementations of decision trees, the questions generally take the form of axis-aligned splits in the data: that is, each node in the tree splits the data into two groups using a cutoff value within one of the features.

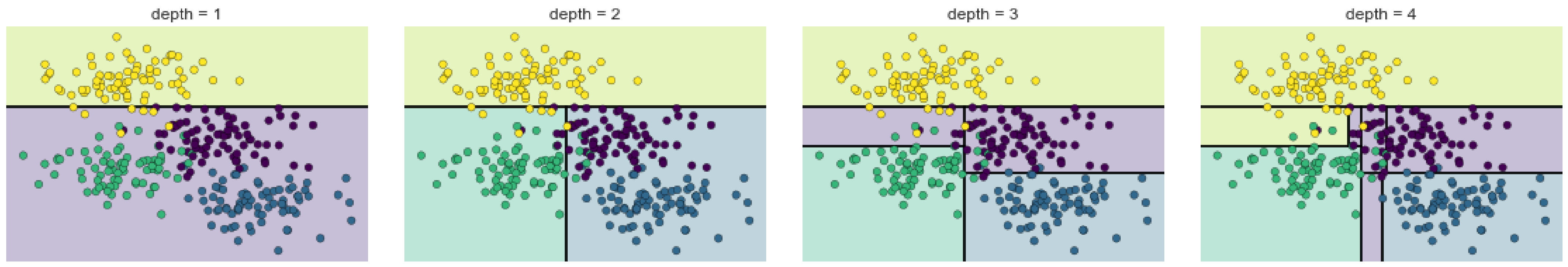
2.1 Creating a decision tree

Consider the following two-dimensional data, which has one of four class labels:

- A simple decision tree built on this data will iteratively split the data along one or the other axis according to some quantitative criterion, and at each level assign the label of the new region according to a majority vote of points within it.
- This figure presents a visualization of the first four levels of a decision tree classifier for this data:





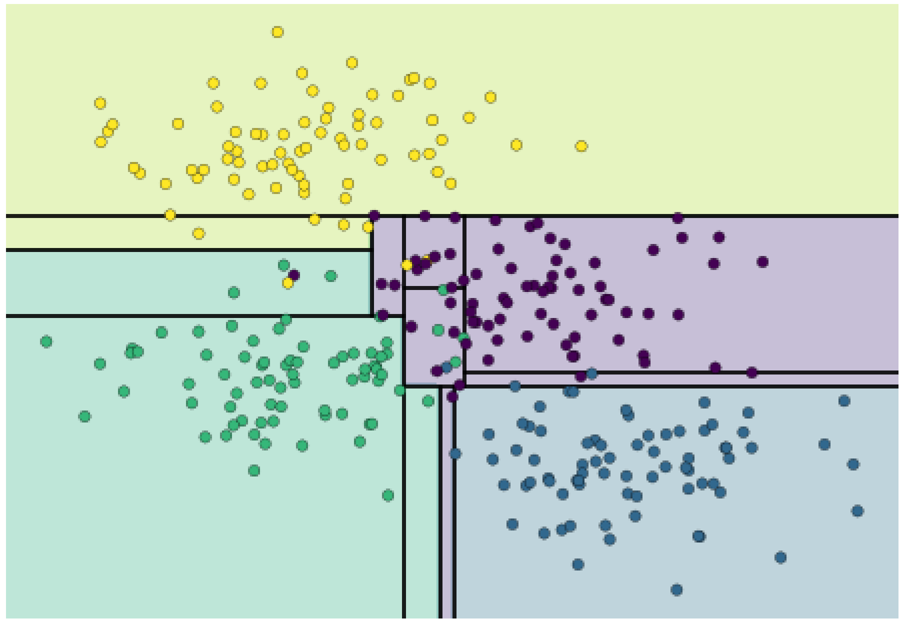


Notice that:

- After the first split, every point in the upper branch remains unchanged, so there is no need to further subdivide this branch.
- Except for nodes that contain all of one color, at each level *every* region is again split along one of the two features.

This process of fitting a decision tree to our data can be done in Scikit-Learn with the DecisionTreeClassifier estimator.

- Examine what the decision tree classification looks like.



Notice that:

- As the depth increases, we tend to get very strangely shaped classification regions;
- For example, at a depth of five, there is a tall and skinny purple region between the yellow and blue regions.

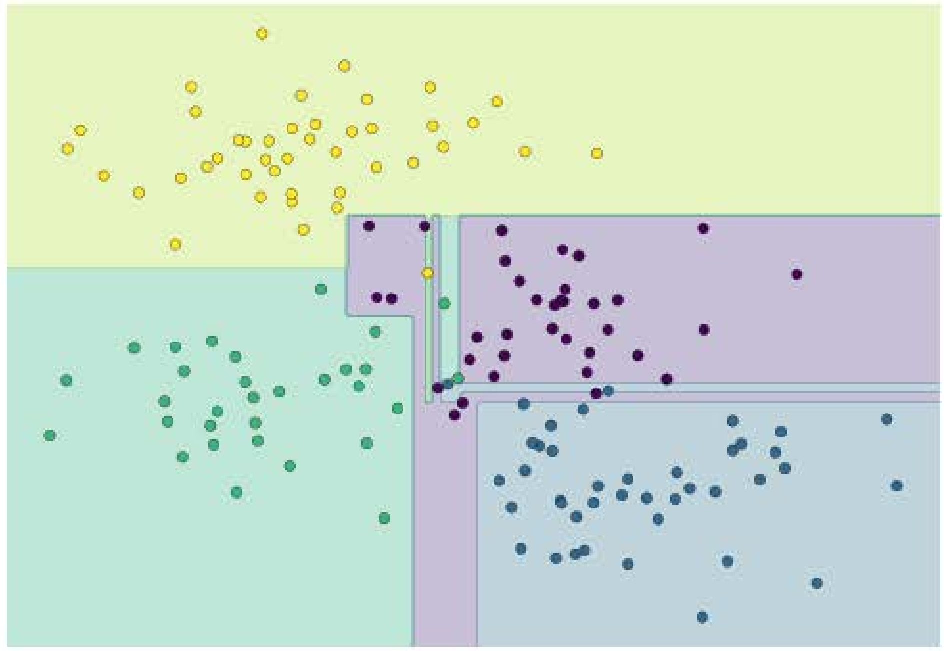
It's clear that this is less a result of the true, intrinsic data distribution, and more a result of the particular sampling or noise properties of the data.

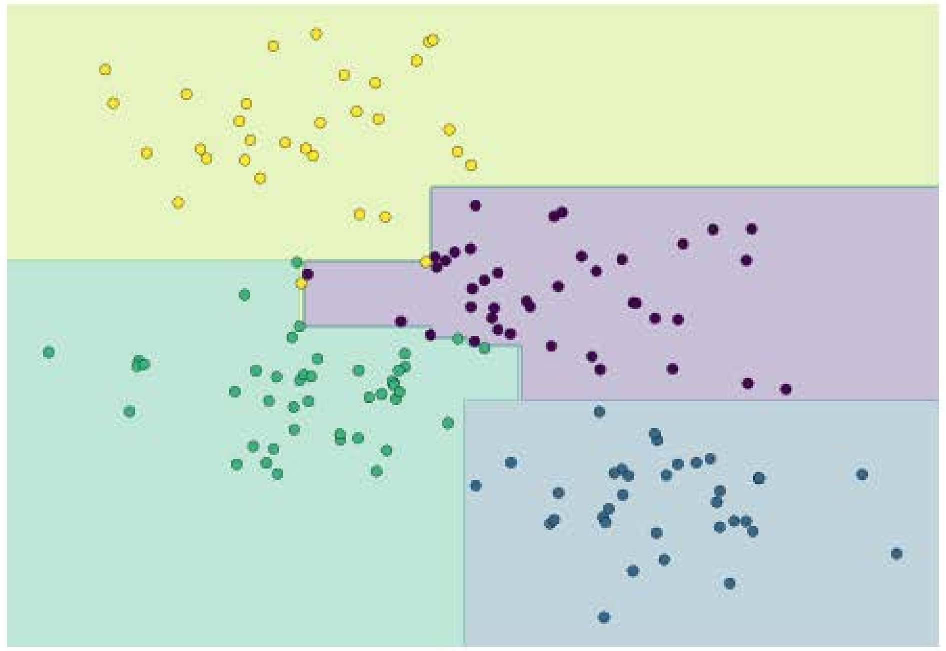
That is, this decision tree, even at only five levels deep, is clearly over-fitting our data.

2.2 Decision trees and over-fitting

Such over-fitting turns out to be a general property of decision trees:

- it is very easy to go too deep in the tree, and thus to fit details of the particular data rather than the overall properties of the distributions they are drawn from.
- Another way to see this over-fitting is to look at models trained on different subsets of the data — for example, in this figure we train two different trees, each on half of the original data:



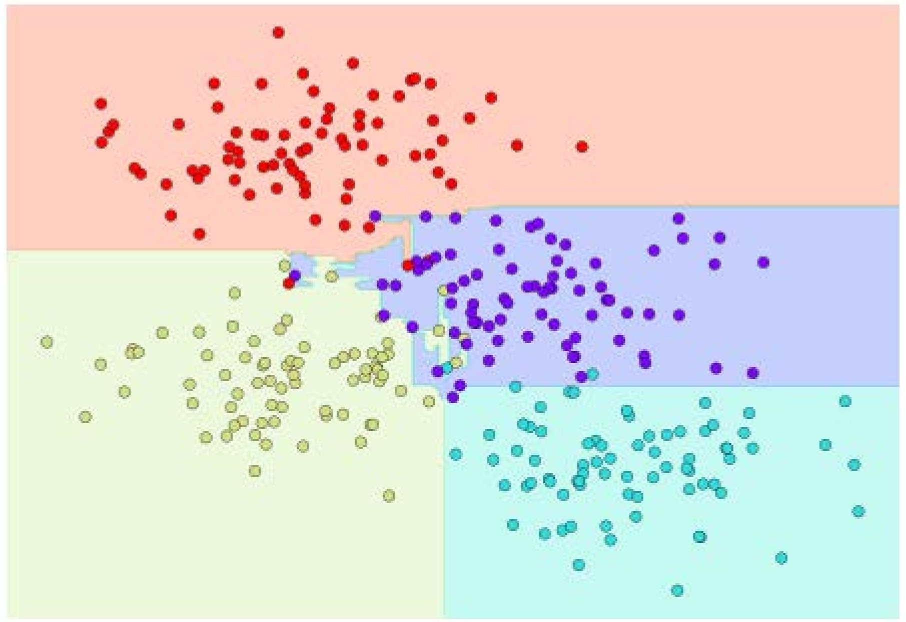


- It is clear that in some places, the two trees produce consistent results (e.g., in the four corners), while in other places, the two trees give very different classifications (e.g., in the regions between any two clusters).
- The key observation is that the inconsistencies tend to happen where the classification is less certain, and thus by using information from both of these trees, we might come up with a better result!
- Just as using information from two trees improves our results, we might expect that using information from many trees would improve our results even further.

2.3 Ensembles of Estimators: Random Forests

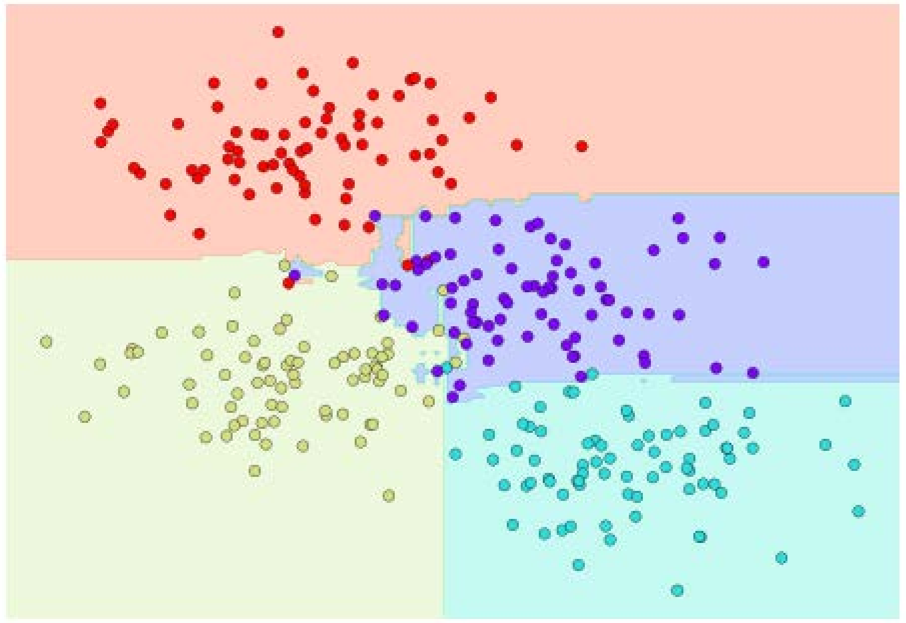
- Multiple overfitting estimators can be combined to reduce the effect of this overfitting.
- This notion is what underlies an ensemble method called ***bagging***.
- Bagging makes use of an ensemble (a grab bag, perhaps) of parallel estimators, each of which over-fits the data, and averages the results to find a better classification.
- An ensemble of randomized decision trees is known as a ***random forest***.

This type of bagging classification can be done manually using Scikit-Learn's BaggingClassifier meta-estimator, as shown here:



- In this example, we have randomized the data by fitting each estimator with a random subset of 80% of the training points.
- In practice, decision trees are more effectively randomized by injecting some stochasticity in how the splits are chosen:
    - this way all the data contributes to the fit each time, but the results of the fit still have the desired randomness.
- For example, when determining which feature to split on, the randomized tree might select from among the top several features.
- You can read more technical details about these randomization strategies in the Scikit-Learn documentation and references within.

- In Scikit-Learn, such an optimized ensemble of randomized decision trees is implemented in the RandomForestClassifier estimator, which takes care of all the randomization automatically.
- All you need to do is select a number of estimators, and it will very quickly (in parallel, if desired) fit the ensemble of trees:

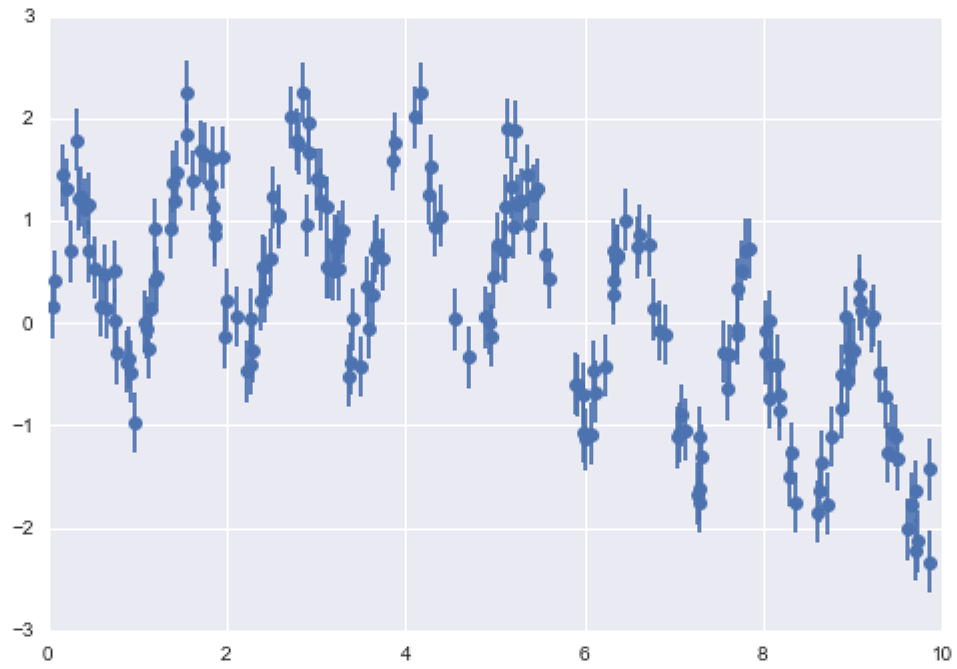


We see that by averaging over 100 randomly perturbed models, we end up with an overall model that is much closer to our intuition about how the parameter space should be split.

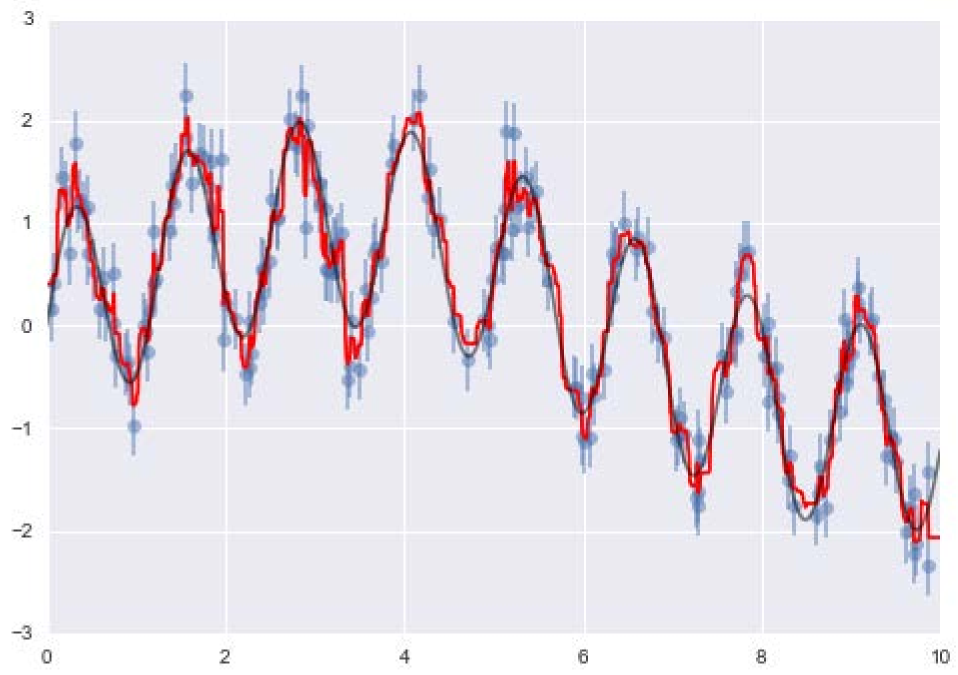
2.4 Random Forest Regression

- In the previous section we considered random forests within the context of classification.
- Random forests can also be made to work in the case of regression (that is, continuous rather than categorical variables).

- The estimator to use for this is the RandomForestRegressor, and the syntax is very similar to what we saw earlier.
- Consider the following data, drawn from the combination of a fast and slow oscillation:



Using the random forest regressor, we can find the best fit curve as follows:



Here the true model is shown in the smooth gray curve, while the random forest model is shown by the jagged red curve. As you can see, the non-parametric random forest model is flexible enough to fit the multi-period data, without needing to specifying a multi-period model!

Example: Random Forest for Classifying Digits

- Earlier we took a quick look at the hand-written digits data. Let's use that again here to see how the random forest classifier can be used in this context.

Summary of Random Forests

- This section contained a brief introduction to the concept of ensemble estimators, and in particular the random forest – an ensemble of randomized decision trees. Random forests are a powerful method with several advantages:
  - Both training and prediction are very fast, because of the simplicity of the underlying decision trees. In addition, both tasks can be straightforwardly parallelized, because the individual trees are entirely independent entities.
  - The multiple trees allow for a probabilistic classification: a majority vote among estimators gives an estimate of the probability (accessed in Scikit-Learn with the predict_proba() method).
  - The nonparametric model is extremely flexible, and can thus perform well on tasks that are under-fit by other estimators.

- A primary disadvantage of random forests is that the results are not easily interpretable: that is, if you would like to draw conclusions about the meaning of the classification model, random forests may not be the best choice.

The End!

Assignment:

1. Practice the example code I ran in this class.
2. Homework 3 will be assigned after the next class.