Chapter 2: Classification

Classification uses models called **classifiers** to predict **categorical (discrete, unordered)** class labels. Table below shows examples for classification tasks and figure below shows a schematic illustration of a classification task:

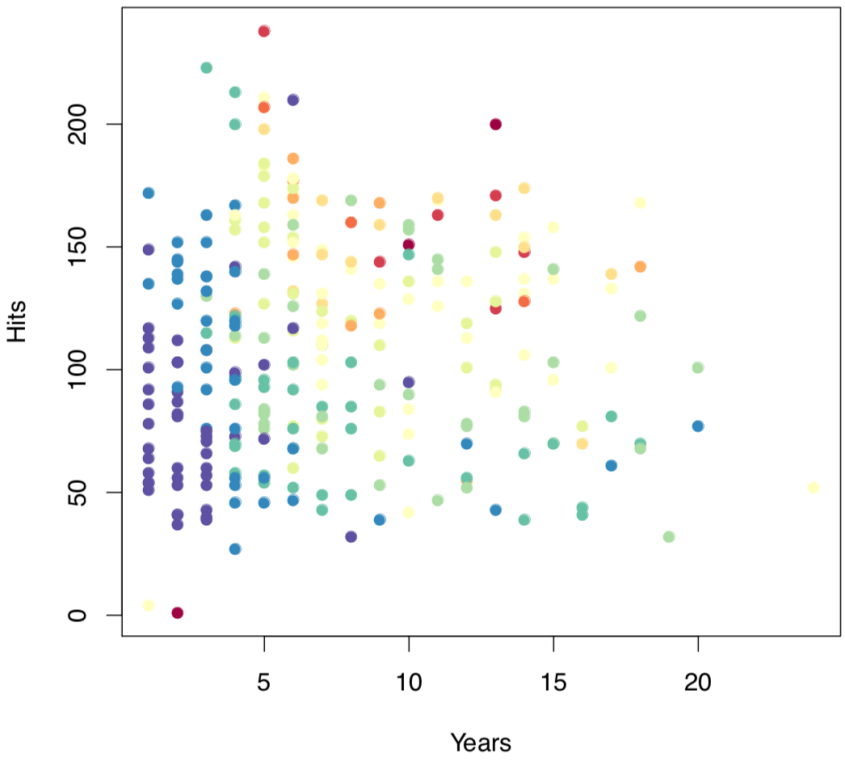
Task	Feature set, **x**	Class label, y
Spam filtering	Features extracted from email message header and content	spam or non-spam
Tumor identification	Features extracted from MRI scans	malignant or benign
Bridge warning	Features extracted from river velocity and depth	danger or safe

This chapter presents concepts, examples, minimum theories and specific techniques for some of the important classifiers. Performance measurements for class imbalance problems will also be discussed.

2.1 Decision Tree

This section introduces a simple classification technique known as the decision tree classifier. These involve stratifying (分開) or segmenting the predictor space into a number of simple regions. Since the set of splitting rules used to segment the predictor space can be summarized in a tree, these types of approaches are known as **decision tree methods**.

Example (Predicting Baseball Players' Salaries Using Trees): Let us use a dataset to predict a baseball player's **Salary** based on **Years** (the number of years that he has played in the major leagues) and **Hits** (the number of hits that he made in the previous year). Salary is measured in thousands of dollars and is color-coded from low (blue, green) to high (yellow, red).



Ans:

Decision tree methods are **simple and useful for interpretation**. However, they typically are not competitive with the best supervised learning approaches in terms of prediction accuracy. Hence in this Chapter we will also introduce random forests. The approach involves producing multiple trees which are then combined to yield a single consensus prediction. We will see that combining a large number of trees can often result in **dramatic improvements in prediction accuracy**, at the expense of some loss in interpretation. Random forests are an example of **an ensemble method**, a method that relies on aggregating the results of an ensemble of simpler estimators.

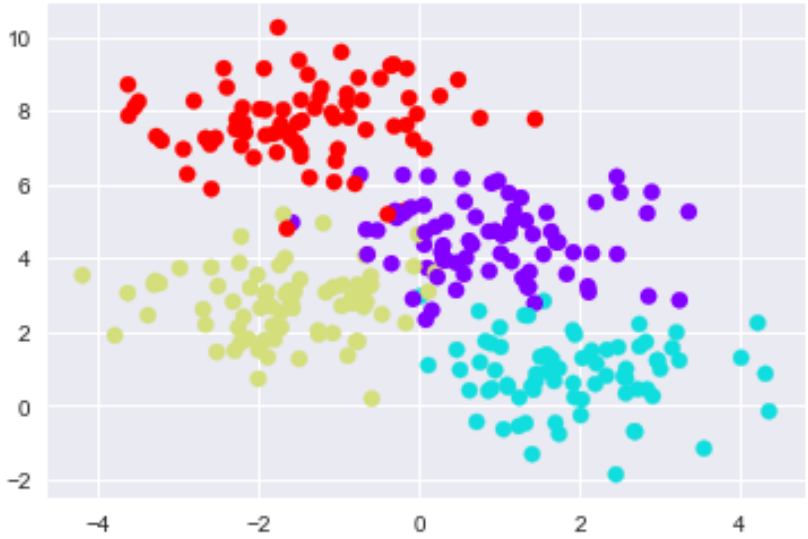
2.1.1 Creating a Decision Tree

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. Let's now take a look at an example. Consider the following two-dimensional data, which has one of four class labels:

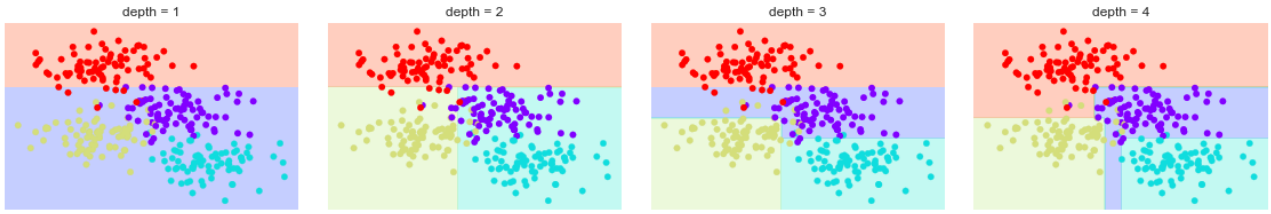
```
%matplotlib inline
```

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=300, centers=4,
                  random_state=0, cluster_std=1.0)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```



A simple decision tree built on this data will iteratively split the data along one or the other axis according to some quantitative criterion. Let us visualize the first four levels of a decision tree classifier for this data:



Below are the python codes to accomplish the task (including a utility function `visualize_classifier` to help us visualize the output of the classifier:):

```
def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):
    ax = ax or plt.gca()

    # Plot the training points
    ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,
               clim=(y.min(), y.max()), zorder=3)
    ax.axis('tight')
    ax.axis('off')
    xlim = ax.get_xlim()
```

```
    ylim = ax.get_ylim()

    # fit the estimator
    model.fit(X, y)
    xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                         np.linspace(*ylim, num=200))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
    # Create a color plot with the results
    n_classes = len(np.unique(y))
    contours = ax.contourf(xx, yy, Z, alpha=0.3,
                           levels=np.arange(n_classes + 1) - 0.5,
                           cmap=cmap, clim=(y.min(), y.max()),
                           zorder=1)
    ax.set(xlim=xlim, ylim=ylim)

from sklearn.tree import DecisionTreeClassifier

fig, ax = plt.subplots(1, 4, figsize=(16, 3))
fig.subplots_adjust(left=0.02, right=0.98, wspace=0.1)

for axi, depth in zip(ax, range(1, 5)):
    model = DecisionTreeClassifier(max_depth=depth)
    visualize_classifier(model, X, y, ax=axi)
    axi.set_title('depth = {0}'.format(depth))
```

Conceptually, a decision tree is grown by first splitting all data points into two groups, such that similar data points are grouped together, and then further repeating this binary splitting process within each group.

As a result, each subsequent leaf node would **have fewer but more homogeneous data points**. The basis of decision trees is that data points following the same path are likely to be similar to each other.

The process of repeatedly splitting data to obtain homogeneous groups is called recursive partitioning. It involves just two steps:

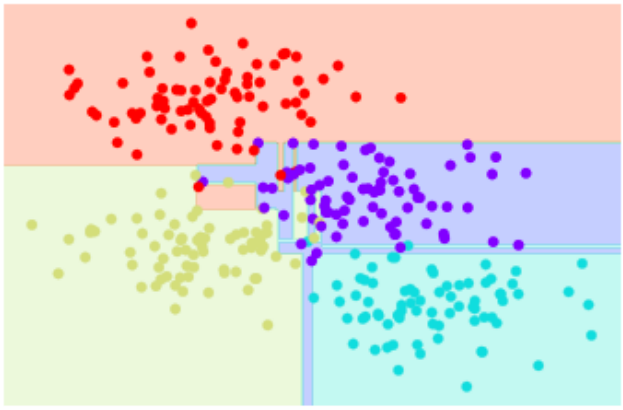
- **Step 1**: Identify the binary question that best splits data points into two groups that are most homogeneous.
- **Step 2**: Repeat Step 1 for each leaf node, until a stopping criterion is reached.

Step 1 involves best split of two groups and we will visit the best splitting criteria later. **Step 2** involves recursive stopping criteria. There are various possibilities for stopping criteria, which can be selected via cross-validation. These include:

- Stop when data points at each leaf are all of the same predicted category or value.
- Stop when the leaf contains less than five data points.
- Stop when further branching does not improve homogeneity beyond a minimum threshold.

This process of fitting a decision tree to our data can be done in `Scikit-Learn` with the `DecisionTreeClassifier` estimator and we can again visualize the final results of classification using the `visualize_classifier` utility function:

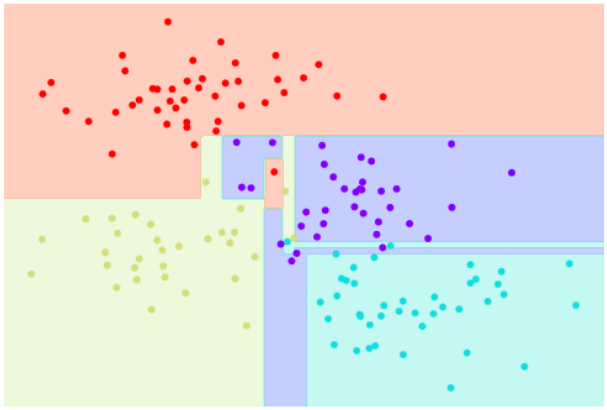
```
model = DecisionTreeClassifier()
visualize_classifier(model, X, y)
```

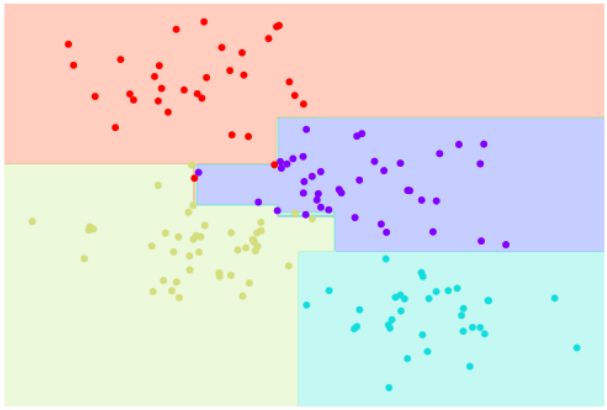


Notice that as the depth increases, we tend to get very strangely shaped classification regions; for example, at a depth of five, there is a tall and skinny purple region between the yellow and blue regions. It's clear that this is less a result of the true, intrinsic data distribution, and more a result 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.

2.1.2 Decision Tree and Over-fitting

The 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:





Q: What have you observed from these two figures?
A:

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**.

You can download the above Python codes `Ch4_decision_tree-tutorial.ipynb` from the course website.

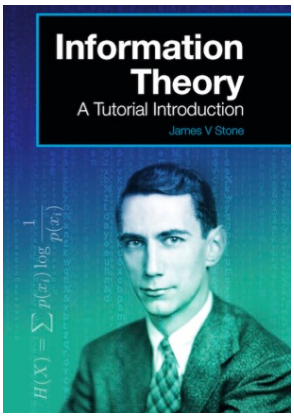
2.1.3 (Optional) Theoretical Minimum: How a Decision Tree Picks Its Split

A key step in decision tree is to identify the binary question that best splits data points into two groups that are most homogeneous.

The obvious question becomes "What is best?"

The **solution** comes from "Information Theory" by Claude Shannon[1]

[1] In 1948, Claude Shannon published a paper called *A Mathematical Theory of Communication*. This paper heralded a transformation in our understanding of information. Before Shannon's paper, information had been viewed as a kind of poorly defined miasmic fluid. But after Shannon's paper, it became apparent that information is a well-defined and, above all, measurable quantity.



The most commonly used solution is either the "Gini" criteria, or the "Entropy" criteria. The next few pages give examples of both equations which are similar, but a little different. However, at the end of the day, it usually makes very little difference which one you use, as they tend to give results that are only a few percent different.

In `Scikit-Learn`, the default is the "Gini" criteria, so we'll start with that.

Gini Criteria

The equation for the **Gini impurity** is:

$$Gini = 1 - \sum_j p_j^2$$

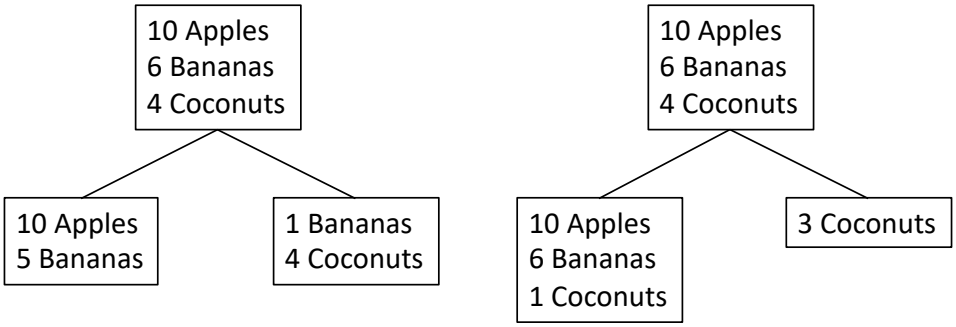
where p is the probability of having a given data class in your dataset. The lower the Gini impurity, the better.

For instance, let's say that you have a dataset of 10 Apples, 6 Bananas, and 4 Coconuts. The probability for each class is $\frac{10}{20}$ (Apple), $\frac{6}{20}$ (Banana), $\frac{4}{20}$ (Coconut).

Q: What is the Gini impurity for this case?
A:
$Gini = 1 - \sum_j p_j^2 = (1 - 0.5^2 - 0.3^2 - 0.2^2) = 0.62$

Remark: The best value that we could have is an impurity of 0. That would occur if we had a branch that is 100% one class, since the equation would become 1 – 1.

Now let's say that the decision tree has two possibilities to split the dataset into two branches:



Q: Which one is better?

A:

First Possible Split:

For the first possible split, we calculate the Gini Impurity of Branch 1 to be .444 and the Gini Impurity of Branch 2 to be .32

First Alternative – Branch 1			
Class	Count	Percentage	Square of Percentage
Apples	10	0.667	0.444
Bananas	5	0.333	0.111
Total	15	Total	0.556
		Gini Impurity – This Branch	0.444
First Alternative – Branch 2			
Class	Count	Percentage	Square of Percentage
Bananas	1	0.2	0.04
Coconuts	4	0.8	0.64
Total	5	Total	0.68
		Gini Impurity – This Branch	0.32
		Weighted Gini Impurity	0.413

There are 15 items in branch 1, and there are 5 items in branch 2. So we can calculate the combined Gini impurity of both branches by taking the weighted average of the two branches = (15*.444+5*.32)/20 which gives a total value of **.413**.

You can do the same thing for the Second Possible Split and obtain a total value of **.447**. So with these

two alternatives a decision tree would be generated with the first choice.

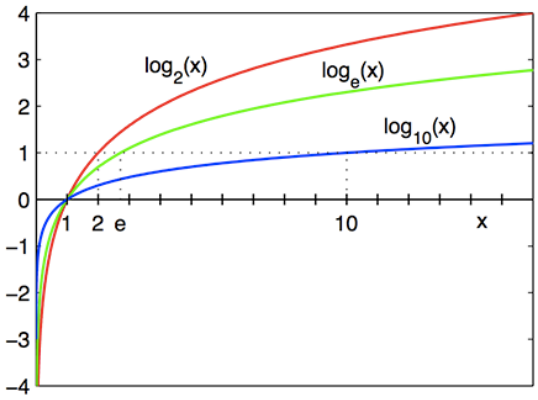
Entropy Criteria

The equation for entropy is different than the Gini equation, but other than that, the process is pretty much the same. The equation for entropy is:

$$Entropy = \sum_j -p_j \times \log_2(p_j)$$

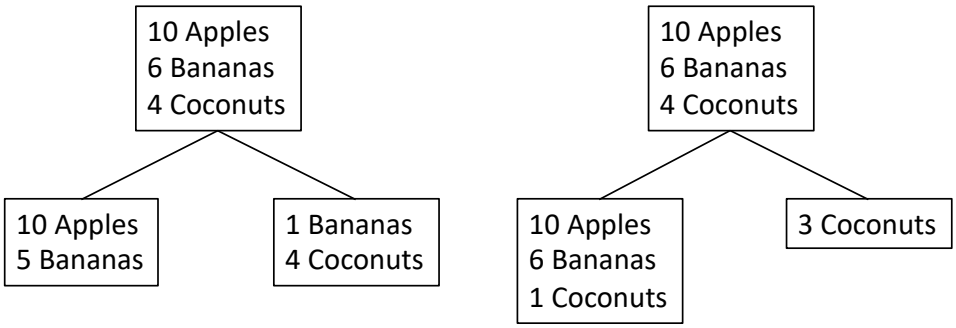
where p is the probability of having a given data class in your dataset. For entropy, just like the Gini criteria, the lower the number the better, with the best being an Entropy of zero.

Remark: For this equation, for each probability, we are multiplying that probability by the base 2 logarithm of that probability. Since each of these probabilities are a decimal between 0 and 1, the base 2 logarithm will always be negative (or zero), which when multiplied by the negative sign in the equation will give a positive number for the total entropy summation.



If we go back to the scenario where we have a dataset of 10 Apples, 6 Bananas, and 4 Coconuts. The probability for each class is $\frac{10}{20}$ (Apple), $\frac{6}{20}$ (Banana), $\frac{4}{20}$ (Coconut). We can calculate the total entropy to be 1.485 as show below:

Class	Count	Percentage	-Percent*Log2(Percent)
Apples	10	0.5	0.500
Bananas	6	0.3	0.521
Coconuts	4	0.2	0.464
Total	20	Entropy Sum	1.485



For first possibility of split, we can calculate the weighted entropy to be **.869** (see below) and for second possibility, the weighted entropy is **1.038**.

First Alternative – Branch 1			
Class	Count	Percentage	-Percent*Log2(Percent)
Apples	10	0.667	0.390
Bananas	5	0.333	0.528
Total	15	Entropy Sum	0.918
First Alternative – Branch 2			
Class	Count	Percentage	-Percent*Log2(Percent)
Bananas	1	0.2	0.464
Coconuts	4	0.8	0.258
Total	5	Entropy Sum	0.722
		Weighted Entropy	0.869

So for this example, just like for the Gini criteria, we see that the first possible split has a lower entropy than the second possible split, so the first possibility would be the branching that was generated.

The total **information gain** for the first split would be the entropy before splitting minus the entropy after splitting. Which is 1.485 - .869 = .616

2.1.4 Decision Tree Classifier: Summary

- A decision tree makes predictions by asking a sequence of binary questions.

- The data sample is split repeatedly to obtain homogeneous groups in a process called recursive partitioning, until a stopping criterion is reached. The most commonly used solution to determine the best split is either the "Gini" criteria, or the "Entropy" criteria from Information Theory.
- **While easy to use and understand**, decision trees are prone to overfitting, which leads to inconsistent results. To minimize this, we could use an ensemble of randomized decision trees such as random forests.

2.2 Naïve Bayes Classifier

Many classification problems involve **uncertainty**. In the presence of uncertainty, there is a need to not only make predictions of class labels but also provide a measure of confidence associated with every prediction. Probability theory offers a systematic way for quantifying and manipulating uncertainty in data.

Classification models that uses the probability theory to represent the relationships between features and class labels are known as **probabilistic classification model**. In this section, we present the naïve Bayes classifier, which is one of the simplest and most widely-used probabilistic classification model.

Naive Bayes models are a group of extremely fast and simple classification algorithms that are often suitable for very **high-dimensional datasets**. Because they are so fast and have so few tunable parameters, they end up being very useful as a quick-and-dirty baseline for a classification problem.

2.2.1 Theoretical Minimum: Bayes' Theorem

Naive Bayes classifiers are built on Bayesian classification methods. These methods rely on **Bayes' theorem** which describes how to **update the probabilities of hypotheses when given evidence (data)**. Bayes' theorem follows simply from the axioms of conditional probability, but can be used to powerfully reason about a wide range of problems involving belief updates.

Given a hypothesis H and data D, Bayes' theorem states that the relationship between the probability of the hypothesis **before** getting the data $P(H)$ and the probability of the hypothesis **after** getting the data $P(H|D)$ is:

$$P(H|D) = \frac{P(D|H)}{P(D)} P(H)$$

in which the vertical bar $|$ stands for ***given that***. $P(H|D)$ is the probability of hypothesis H being true *given that* data D is true. $P(D|H)$ is probability of data D being true *given that* hypothesis H is

true.

Bayes' theorem relates the probability of the hypothesis **before** getting the data $P(H)$, to the probability of the hypothesis after getting the data, $P(H|D)$. For this reason, $P(H)$ is called the **prior probability**, while $P(H|D)$ is called the **posterior probability**.

Example: If a single card is drawn from a standard deck of cards, what is the probability that the card is a king?

A:

If observed data is provided (for instance, someone looks at the card) that the single card is a face card (Jack, Queen, King), what is the *posterior* probability $P(\text{King}|\text{Face})$?

A:

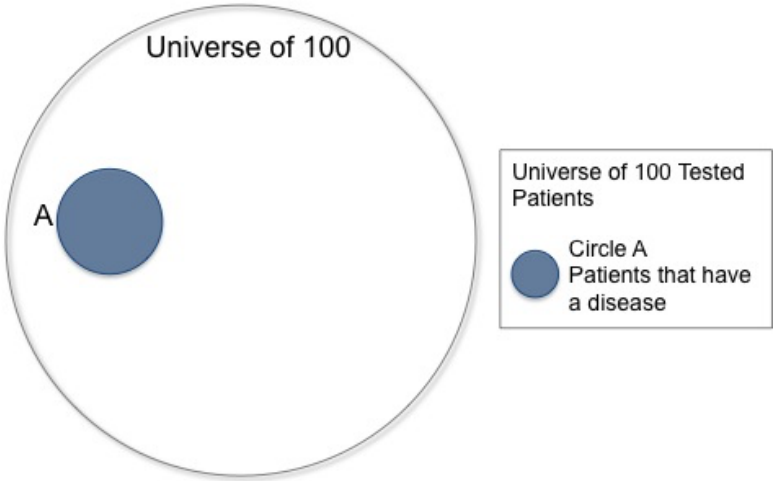
Example: Disease Diagnostic Test: Visualizing Bayes' Theorem

Venn diagrams are particularly useful for visualizing Bayes' theorem, since both the diagrams and the theorem are about looking at the intersections of different spaces of events.

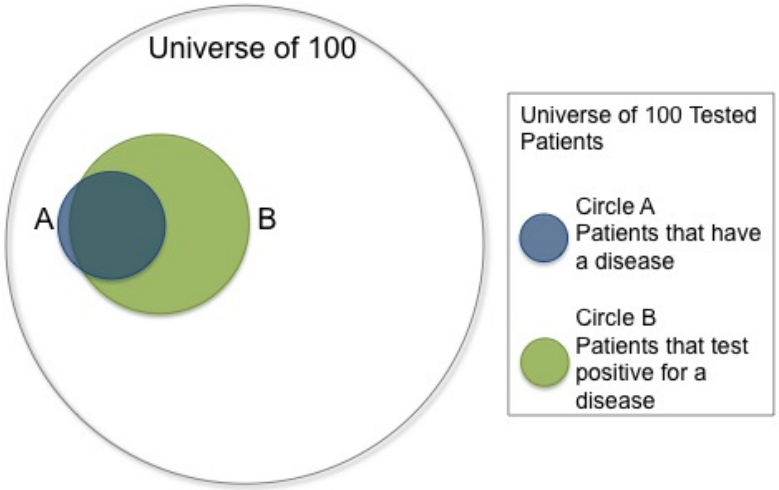
A disease is present in 5 out of 100 people, and a test that is 90% accurate (meaning that the test produces the correct result in 90% of cases) is administered to 100 people. If one person in the group tests positive, what is the probability that this one person has the disease?

The intuitive answer is that the one person is 90% likely to have the disease. But we can visualize this to show that **it's not accurate**!

First, draw the total population and the 5 people who have the disease:



Next, overlay a circle to represent the people who get a positive result on the test. We know that 90% of those with the disease will get a positive result, so need to cover 90% of circle A, but we also know that 10% of the population who does not have the disease will get a positive result, so we need to cover 10% of the non-disease carrying population (the total universe of 100 less circle A).



Circle B is covering a substantial portion of the total population. It actually covers more area than the total portion of the population with the disease. This is because 14 out of the total population of 100 (90% of the 5 people with the disease + 10% of the 95 people without the disease) will receive a positive result. Even though this is a test with 90% accuracy, this visualization shows that any one patient who tests positive (Circle B) for the disease only has a 32.14% (2.5 in 14) chance of actually having the disease.

Example: Disease Diagnostic Test: Calculation with Bayes' Theorem

Let + be the event that the result of a diagnostic test is positive. Let H and H^c be the event that the people have or do not have the disease respectively.

Q: How to read $P(H|+)$, $P(+|H)$ and $P(+|H^c)$?

A:

Mathematically, we want $P(H|+)$ given $P(+|H)$:

$$P(H|+) = \frac{P(+|H)}{P(+)}P(H)$$

in which $P(+|H) = \frac{90}{100}$ and $P(H) = \frac{5}{100}$. $P(+)$ is the probability of people who have and do not have the disease will get a positive result. Mathematically, it is:

$$P(+) = P(+|H)P(H) + P(+|H^c)P(H^c) = \frac{90}{100}\frac{5}{100} + \frac{10}{100}\frac{95}{100}$$

Substituting these probability values into the Bayes' theorem, we have

$$P(H|+) = \frac{\frac{90}{100}}{\left(\frac{90}{100}\frac{5}{100} + \frac{10}{100}\frac{95}{100}\right)}\frac{5}{100} = \frac{2.5}{14} = 32.14\%$$

Using Bayes' Theorem for Classification

For the purpose of classification, we are interested in computing the probability of a class label *y* for a data instance given its features **x**. Using Bayes' theorem, we can express the *posterior* probability as:

$$P(y|\mathbf{x}) = \frac{P(\mathbf{x}|y)}{P(\mathbf{x})}P(y)$$

Remarks:

1. $P(\mathbf{x}|y)$ is known as the **class-conditional probability** of the features given that class label. $P(\mathbf{x}|y)$ measures the likelihood of observing **x** from the distribution of instance belonging to *y*. If **x** indeed belongs to *y*, then we should expect $P(\mathbf{x}|y)$ to be high. From this point of view, the use of the **class-conditional probability** attempts to capture the process from which the data instances were generated. Because of this interpretation, probabilistic classification models that involve computing class-conditional probabilities are known as **generative classification models**, that attempt to provide insights about the underlying mechanism behind the generation of feature values.
2. The naïve Bayes classifier **assumes** that the **class-conditional probability** of all features **x** can be factored as a product of class-conditional probabilities of every feature x_i:

$$P(\mathbf{x}|y) = \prod_{i=1}^{d} P(x_i|y)$$

The means that we can consider the features to be independent of each other.

2.2.2 Gaussian Naïve Bayes

For every data instance **x** consists of *d* features $\{x_1 \quad x_2 \quad \cdots \quad x_d\}$, the naïve Bayes classifier assumes that the feature values x_i are **conditionally independent** of each other, given the class label y. This is where the "naive" in "naive Bayes" comes in: if we make very naive assumptions about the generative model for each label, we can find a rough approximation of the generative model for each class, and then proceed with the Bayesian classification. Different types of naive Bayes classifiers rest on different naive assumptions about the data, and we will examine a few of these in the following sections.

The easiest naive Bayes classifier to understand is Gaussian naive Bayes. In this classifier, the assumption is that continuous **data from each label is drawn from a simple Gaussian distribution**. Recall the Gaussian distribution is characterized by two parameters, the mean μ and the variance σ^2. For each class y_i, the class-conditional probability for feature X_i is:

$$P(X_i = x_i|Y = y_i) = \frac{1}{\sqrt{2\pi}\sigma_{ij}} \exp\left[-\frac{(x_i - \mu_{ij})^2}{2\sigma_{ij}^2}\right]$$

The parameters μ_{ij} and σ_{ij}^2 can be estimated using the sample mean of $X_i(\bar{x})$ and sample variance for all the training instances that belong to y_i.

Example: Consider a dataset in the figure below. Find the conditional probability of 120K income with respect to the class `No` using Gaussian distribution.

	binary	categorical	continuous	class
Tid	Home Owner	Marital Status	Annual Income	Defaulted borrower
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

Remarks

1. The reason that naive Bayes models are so efficient is that they learn parameters by looking at each feature individually and collect simple per-class statistics from each feature.
2. There are three kinds of naive Bayes classifiers implemented in `scikit-learn`: `GaussianNB`, `BernoulliNB`, and `MultinomialNB`. `GaussianNB` can be applied to any continuous data, while `BernoulliNB` assumes binary data and `MultinomialNB` assumes count data (that is, that each feature represents an integer count of something, like how often a word appears in a sentence). `BernoulliNB` and `MultinomialNB` are mostly used in text data classification.

Python Example

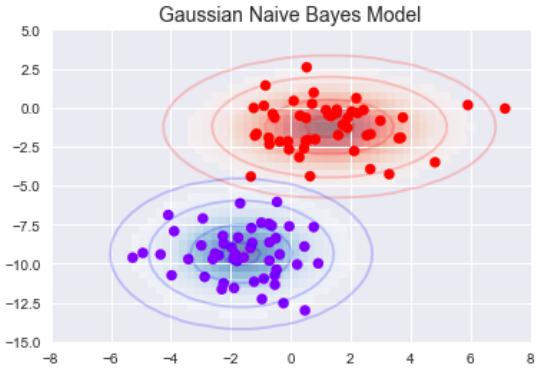
Imagine that you have the following data:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()

from sklearn.datasets import make_blobs
```

```
X, y = make_blobs(100, 2, centers=2, random_state=2, cluster_std=1.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```

One extremely fast way to create a simple model is to assume that the data is described by a Gaussian distribution with no covariance between dimensions. This model can be fit by simply finding the mean and standard deviation of the points within each label, which is all you need to define such a distribution. The result of this naive Gaussian assumption is shown in the following figure (see Python code on the course website on how to plot the figure):



The ellipses here represent the **Gaussian generative model** for each label, with larger probability toward the center of the ellipses. With this generative model in place for each class, we have a simple recipe to compute $P(\mathbf{x}|y) = \prod_{i=1}^{d} P(x_i|y)$ for any data point, and thus we can quickly compute the posterior probability and determine which label is the most probable for a given point. This procedure is implemented in `Scikit-Learn`'s `sklearn.naive_bayes.GaussianNB` estimator:

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X, y);
```

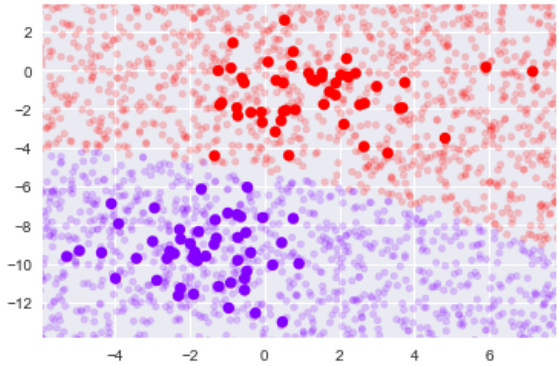
Now let's generate some new data and predict the label:

```
rng = np.random.RandomState(0)
Xnew = [-6, -14] + [14, 18] * rng.rand(2000, 2) # Create
ynew = model.predict(Xnew)
```

Programming Tip: `rng` is an object from the class `np.random.RandomState` and `rng.rand(2000, 2)` creates an array of 2000 by 2 and populate it with random samples from a uniform distribution over [0, 1).

Now we can plot this new data to get an idea of where the decision boundary is:

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow')
lim = plt.axis()
plt.scatter(Xnew[:, 0], Xnew[:, 1], c=ynew, s=20, cmap='rainbow',
alpha=0.2)
plt.axis(lim);
```



We see a slightly curved boundary in the classifications—in general, the boundary in Gaussian naive Bayes is quadratic.

A nice piece of this Bayesian formalism is that it naturally allows for probabilistic classification, which we can compute using the `predict_proba` method:

```
yprob = model.predict_proba(Xnew)
yprob[-8:].round(2)
```

```
Out[7]: array([[ 0.89,  0.11],
               [ 1.  ,  0.  ],
               [ 1.  ,  0.  ],
               [ 1.  ,  0.  ],
               [ 1.  ,  0.  ],
               [ 1.  ,  0.  ],
               [ 0.  ,  1.  ],
               [ 0.15,  0.85]])
```

The columns give the **posterior probabilities** of the first and second label, respectively. If you are looking for estimates of uncertainty in your classification, Bayesian approaches like this can be a useful approach.

You can download the above Python code `Ch4_GaussianNBBlob.ipynb` from the course website.

2.2.3 Multinomial Naïve Bayes

The Gaussian assumption just described is by no means the only simple assumption that could be used to specify the generative distribution for each label. Another useful example is multinomial naïve Bayes, where the features are assumed to be generated from **a simple multinomial distribution**. The multinomial distribution describes the probability of observing counts among a number of categories[2], and thus multinomial naïve Bayes is most appropriate for features that represent **counts** or count rates.

The idea is precisely the same as before, except that instead of modeling the data distribution with the best-fit Gaussian, we model the data distribution with a best-fit multinomial distribution.

One place where multinomial naive Bayes is often used is in text classification, where the features are related to **word counts** or frequencies within the documents to be classified. Before discussing the classification problem, let us go through how to extract features from text.

Text Features

A common need in feature engineering is to **convert text to a set of representative numerical values**. For example, most automatic mining of social media data relies on some form of encoding the text as numbers. One of the simplest methods of encoding data is by **word counts**: you take each snippet of text, count the occurrences of each word within it, and put the results in a table.

For example, consider the following set of three phrases:

```
sample = ['problem of evil',
          'evil queen',
          'horizon problem']
```

For a vectorization of this data based on word count, we could construct a column representing the word "problem," the word "evil," the word "horizon," and so on. While doing this by hand would be possible, the tedium can be avoided by using Scikit-Learn's `CountVectorizer`:

```
from sklearn.feature_extraction.text import CountVectorizer

vec = CountVectorizer()
X = vec.fit_transform(sample)
X
```

[2] See https://www.youtube.com/watch?v=5eLZtikDR4c for explanation of multinomial distribution.

The result is a sparse matrix recording the number of times each word appears; it is easier to inspect if we convert this to a `DataFrame` with labeled columns:

```
import pandas as pd
pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
```

There are some issues with this approach, however: the raw word counts lead to features which put too much weight on words that appear very frequently, and this can be sub-optimal in some classification algorithms. One approach to fix this is known as term frequency-inverse document frequency (TF–IDF) which weights the word counts by a measure of how often they appear in the documents[3]. The syntax for computing these features is similar to the previous example:

```
from sklearn.feature_extraction.text import TfidfVectorizer
vec = TfidfVectorizer()
X = vec.fit_transform(sample)
pd.DataFrame(X.toarray(), columns=vec.get_feature_names())
```

Example: Classifying Text

One place where multinomial naive Bayes is often used is in text classification, where the features are related to word counts or frequencies within the documents to be classified. We will use the sparse word count features from the 20 Newsgroups corpus to show how we might classify these short documents into categories.

Let's download the data and take a look at the target names:

```
from sklearn.datasets import fetch_20newsgroups

data = fetch_20newsgroups()
data.target_names
```

[3] tf-idf 是一種用於資訊檢索與文字挖掘的常用加權技術。tf-idf 是一種統計方法，用以評估一字詞對於一個檔案集或一個語料庫中的其中一份檔案的重要程度。字詞的重要性隨著它在檔案中出現的次數成正比增加，但同時會隨著它在語料庫中出現的頻率成反比下降。See https://zh.wikipedia.org/wiki/Tf-idf.

```
Out[1]: ['alt.atheism',
         'comp.graphics',
         'comp.os.ms-windows.misc',
         'comp.sys.ibm.pc.hardware',
         'comp.sys.mac.hardware',
         'comp.windows.x',
         'misc.forsale',
         'rec.autos',
         'rec.motorcycles',
         'rec.sport.baseball',
         'rec.sport.hockey',
         'sci.crypt',
         'sci.electronics',
         'sci.med',
         'sci.space',
         'soc.religion.christian',
         'talk.politics.guns',
         'talk.politics.mideast',
         'talk.politics.misc',
         'talk.religion.misc']
```

For simplicity here, we will select just a few of these categories, and download the training and testing set:

```
categories = ['talk.religion.misc', 'soc.religion.christian',
              'sci.space', 'comp.graphics']
train = fetch_20newsgroups(subset='train', categories=categories)
test = fetch_20newsgroups(subset='test', categories=categories)
```

Here is a representative entry from the data:

```
print(train.data[5])
```

```
From: dmcgee@uluhe.soest.hawaii.edu (Don McGee)
Subject: Federal Hearing
Originator: dmcgee@uluhe
Organization: School of Ocean and Earth Science and Technology
Distribution: usa
Lines: 10


Fact or rumor....?  Madalyn Murray O'Hare an atheist who eliminated the
use of the bible reading and prayer in public schools 15 years ago is now
going to appear before the FCC with a petition to stop the reading of the
Gospel on the airways of America.  And she is also campaigning to remove
Christmas programs, songs, etc from the public schools.  If it is true
then mail to Federal Communications Commission 1919 H Street Washington DC
20054 expressing your opposition to her request.  Reference Petition number

2493.
```

In order to use this data for machine learning, we need to be able to convert the content of each string into a vector of numbers. For this we will use the TF-IDF vectorizer, and create a pipeline that attaches it to a multinomial naive Bayes classifier:

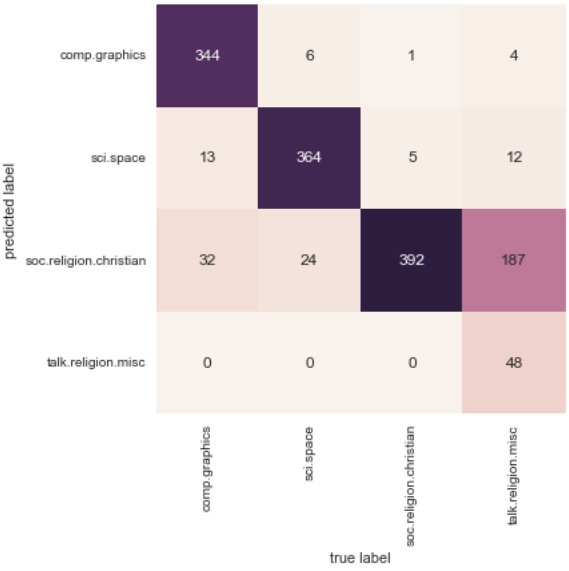
```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline

model = make_pipeline(TfidfVectorizer(), MultinomialNB())
```

With this pipeline, we can apply the model to the training data, and predict labels for the test data:

```
model.fit(train.data, train.target)
labels = model.predict(test.data)
```

Now that we have predicted the labels for the test data, we can evaluate them to learn about the performance of the estimator. For example, here is the confusion matrix between the true and predicted labels for the test data:



Remark: Evidently, even this very simple classifier can successfully separate space talk from computer talk, but it gets confused between talk about religion and talk about Christianity. This is perhaps an expected area of confusion!

The very cool thing here is that we now have the tools to determine the category for any string, using the `predict()` method of this pipeline. Here's a quick utility function that will return the prediction for a single string:

```
def predict_category(s, train=train, model=model):
    pred = model.predict([s])
    return train.target_names[pred[0]] #[pred[0] is the label number
```

Let's try it out:

```
predict_category('sending a payload to the ISS')
```

?

A: `'sci.space'`

```
predict_category(' discussing islam vs atheism')
```

```
Out[10]: 'soc.religion.christian'
```

```
predict_category('determining the screen resolution')
```

?

A: `'comp.graphics'`

You can download the above Python code `Ch4_TextClassification.ipynb` from the course website.

2.2.4 Summary: When to Use Naive Bayes

Because naive Bayesian classifiers make such stringent assumptions about data, they will generally not perform as well as a more complicated model. That said, they have several advantages:

- They are extremely fast for both training and prediction
- They provide straightforward probabilistic prediction
- They are often very easily interpretable
- They have very few (if any) tunable parameters

These advantages mean a naive Bayesian classifier is often a good choice as an initial baseline classification. If it performs suitably, then congratulations: you have a very fast, very interpretable classifier for your problem. If it does not perform well, then you can begin exploring more sophisticated models, with some baseline knowledge of how well they should perform.

Naive Bayes classifiers tend to perform especially well in one of the following situations:

- When the naive assumptions actually match the data (very rare in practice)
- For very well-separated categories, when model complexity is less important
- For very high-dimensional data, when model complexity is less important

The last two points seem distinct, but they actually are related: as the dimension of a dataset grows, it is much less likely for any two points to be found close together (after all, they must be close in every single dimension to be close overall). This means that clusters in high dimensions tend to be more separated, on average, than clusters in low dimensions, assuming the new dimensions actually add

information. For this reason, simplistic classifiers like naive Bayes tend to work as well or better than more complicated classifiers as the dimensionality grows: once you have enough data, even a simple model can be very powerful.

2.3 Support Vector Machine (SVM)

In this section, we will discuss the support vector machine (SVM), an approach for classification that was developed in the computer science community in the 1990s and that has grown in popularity since then. Support vector machines are a particularly powerful and flexible class of supervised algorithms for **both classification and regression**.

Support vector machines are used in a variety of classification scenarios, such as **image recognition** and **hand-writing pattern recognition**. Being able to classify thousands or millions of images is becoming more and more important with the use of smartphones and applications like FaceBook. SVMs can also do text classification on normal text or web documents, for instance.

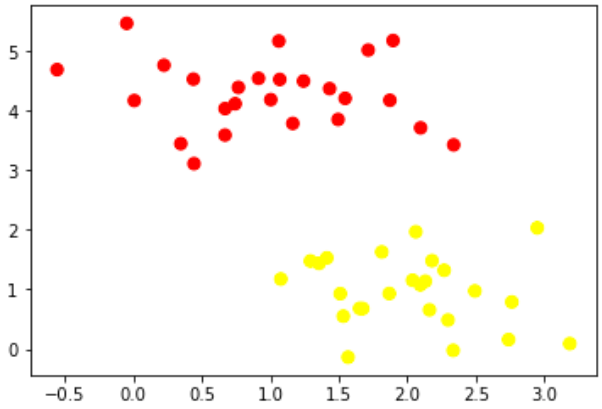
Medical and bioinformatics science has long used support vector machines for **protein classification**. The National Institute of Health has even developed a support vector machine protein software library. It's a web-based tool that classifies a protein into its functional family.

2.3.1 Motivation

Let us consider the simple case of a classification task, in which the two classes of points are well separated:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets.samples_generator import make_blobs

X, y = make_blobs(n_samples=50, centers=2,
                  random_state=0, cluster_std=0.60)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn');
```



Python Programming Tip: (1) you can color-code the lines by specifying a colormap with the `cmap` argument and (2) `make_blobs` creates multiclass datasets by allocating each class one or more normally-distributed clusters of points. It returns `X`: the generated samples with an array of shape [`n_samples`, `n_features` (default =2)] and `y`: the integer labels for cluster membership of each sample with an array of shape [`n_samples`]

Q: We would like to perform classification for the two sets of data. How can we do?
A:

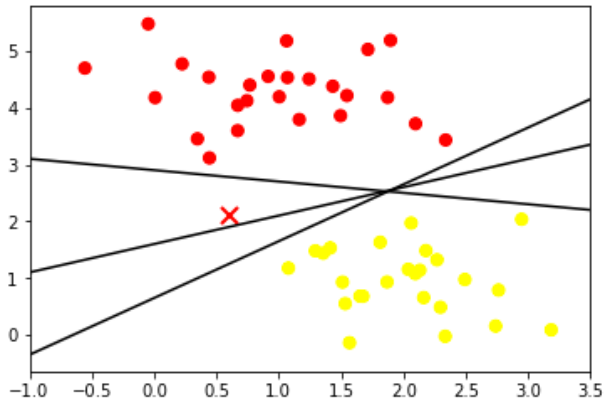
For two-dimensional data like that shown here, this is a task we could do by hand. But immediately we see a problem: there is more than one possible dividing line that can perfectly discriminate between the two classes!

We can draw them as follows:

```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plt.plot([0.6], [2.1], 'x', color='red', markeredgewidth=2,
markersize=10)

for m, b in [(1, 0.65), (0.5, 1.6), (-0.2, 2.9)]:
   plt.plot(xfit, m * xfit + b, '-k')

plt.xlim(-1, 3.5);
```



Q: Do these three lines separate the two sets of data? What is the problem?
A:

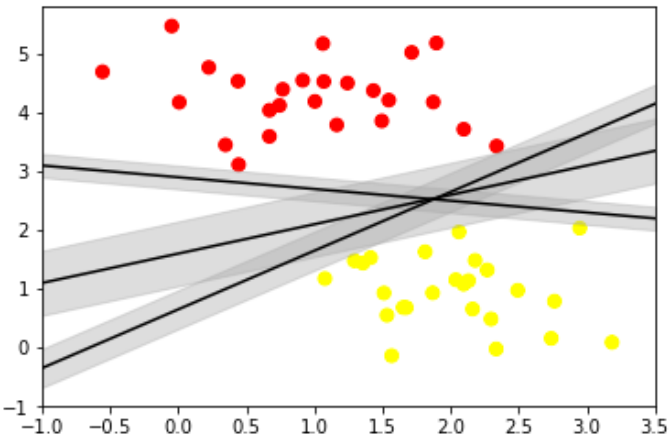
2.3.2 Maximizing the Margin

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:

```
xfit = np.linspace(-1, 3.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')

for m, b, d in [(1, 0.65, 0.33), (0.5, 1.6, 0.55), (-0.2, 2.9, 0.2)]:
    yfit = m * xfit + b
    plt.plot(xfit, yfit, '-k')
    plt.fill_between(xfit, yfit - d, yfit + d, edgecolor='none',
                     color='#AAAAAA', alpha=0.4)

plt.xlim(-1, 3.5);
```



In support vector machines, **the line that maximizes this margin is the one we will choose as the optimal model**. Support vector machines are an example of such a maximum margin estimator.

Remarks:

1. The maximal margin is a natural choice. It is an optimal separating hyperplane (a line in 2D, a plane in 3D, a hyperplane in n-D) that is farthest from the training observations. We hope that a classifier that has a large maximal margin classifier margin on the training data will also have a large margin on the test data, and hence will classify the test observations correctly.
2. The search for an optimal separating hyperplane naturally leads to an optimization problem. We shall discuss the theoretical minimum of SVM later.

Fitting a support vector machine

Let's see the result of an actual fit to this data: we will use `Scikit-Learn`'s support vector classifier to train an SVM model on this data. For the time being, we will use a linear kernel and set the C parameter to a very large number (we'll discuss the meaning of kernel and C parameter in more depth momentarily).

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

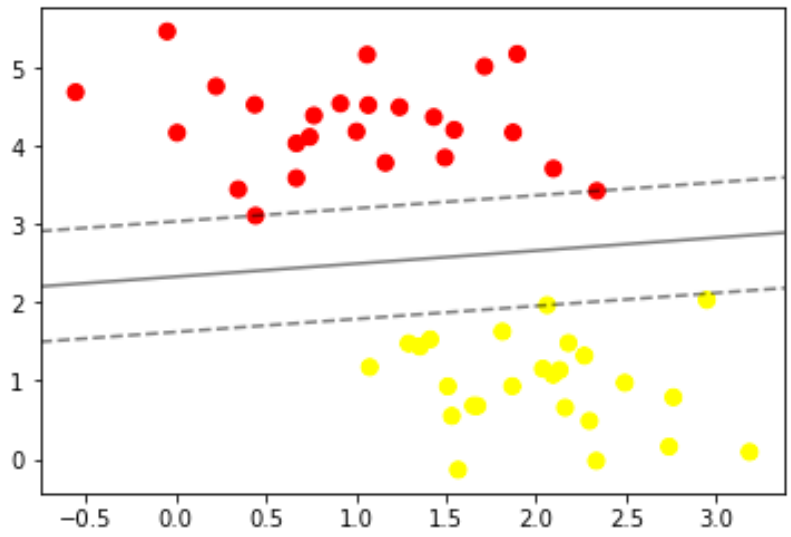
To better visualize what's happening here, let's create a quick convenience function `plot_svc_decision_function` that will plot SVM decision boundaries for us:

```
def plot_svc_decision_function(model, ax=None, plot_support=True):
    """Plot the decision function for a 2D SVC"""
    if ax is None:
        ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # create grid to evaluate model
    x = np.linspace(xlim[0], xlim[1], 30)
    y = np.linspace(ylim[0], ylim[1], 30)
    Y, X = np.meshgrid(y, x)
    xy = np.vstack([X.ravel(), Y.ravel()]).T
    P = model.decision_function(xy).reshape(X.shape)

    # plot decision boundary and margins
    ax.contour(X, Y, P, colors='k',
               levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])

    # plot support vectors
    if plot_support:
        ax.scatter(model.support_vectors_[:, 0],
                   model.support_vectors_[:, 1],
                   s=300, linewidth=1, facecolors='none');
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
```



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. **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 is 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]])
```

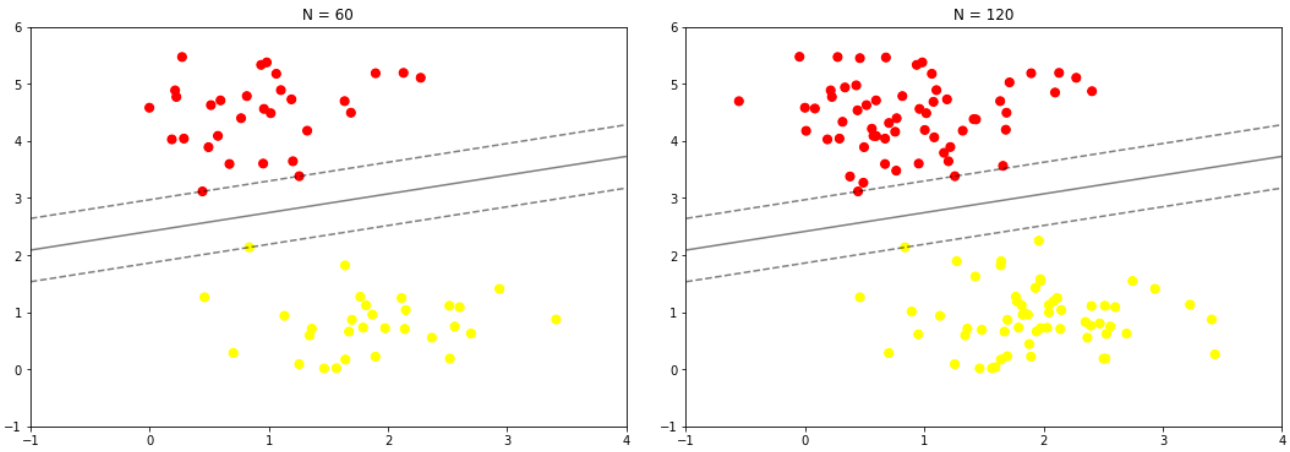
A key to this classifier's success is that for the fit, **only the position of the support vectors matter**; any points further from the margin which are on the correct side do not modify the fit! Technically, this is because these points **do not contribute to the loss function used to fit the model**,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

We can see this, for example, if we plot the model learned from the first 60 points and first 120 points of this dataset:

```
def plot_svm(N=10, ax=None):
    X, y = make_blobs(n_samples=200, centers=2,
                      random_state=0, cluster_std=0.60)
    X = X[:N]
    y = y[:N]
    model = SVC(kernel='linear', C=1E10)
    model.fit(X, y)

    ax = ax or plt.gca()
    ax.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
    ax.set_xlim(-1, 4)
    ax.set_ylim(-1, 6)
    plot_svc_decision_function(model, ax)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)
for axi, N in zip(ax, [60, 120]):
    plot_svm(N, axi)
    axi.set_title('N = {0}'.format(N))
```



In the left panel, we see the model and the support vectors for 60 training points. In the right panel, we

have doubled the number of training points, 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**.

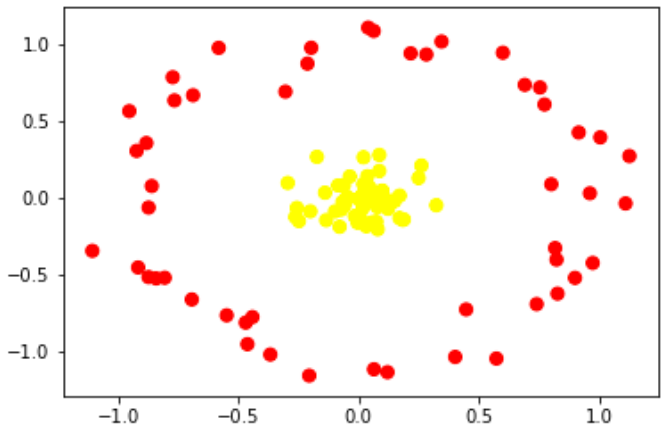
2.3.3 Beyond linear boundaries: Kernel SVM

Where SVM becomes extremely powerful is when it is combined with nonlinear **kernels**. To motivate the need for kernels, let's look at some data that is not linearly separable:

```
from sklearn.datasets.samples_generator import make_circles
X, y = make_circles(100, factor=.1, noise=.1)

clf = SVC(kernel='linear').fit(X, y)

plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
```



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

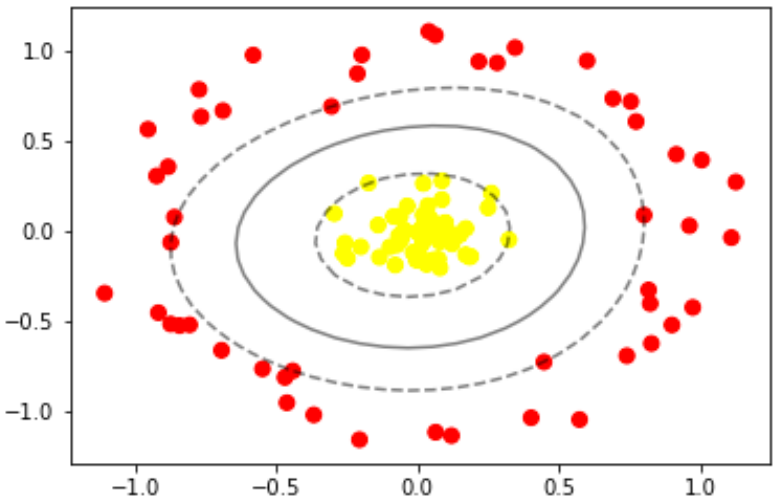
Q: Any potential basis function we can use to project the data so a linear separator would be sufficient?
A:

In `Scikit-Learn`, we can apply kernelized SVM simply by changing our linear kernel to an RBF (radial basis function) kernel:

```
clf = SVC(kernel='rbf', C=1E6)
clf.fit(X, y)
```

```
SVC(C=1000000.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

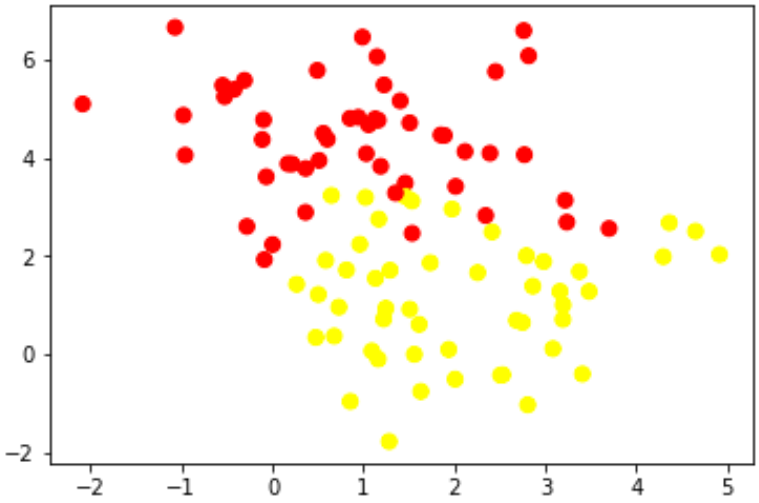
```
plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
plot_svc_decision_function(clf)
```



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.

2.3.4 Tuning the SVM: Softening Margins

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?* For example, you may have data like this:



To handle this case, the SVM implementation has a bit of a *fudge-factor*[4] 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.

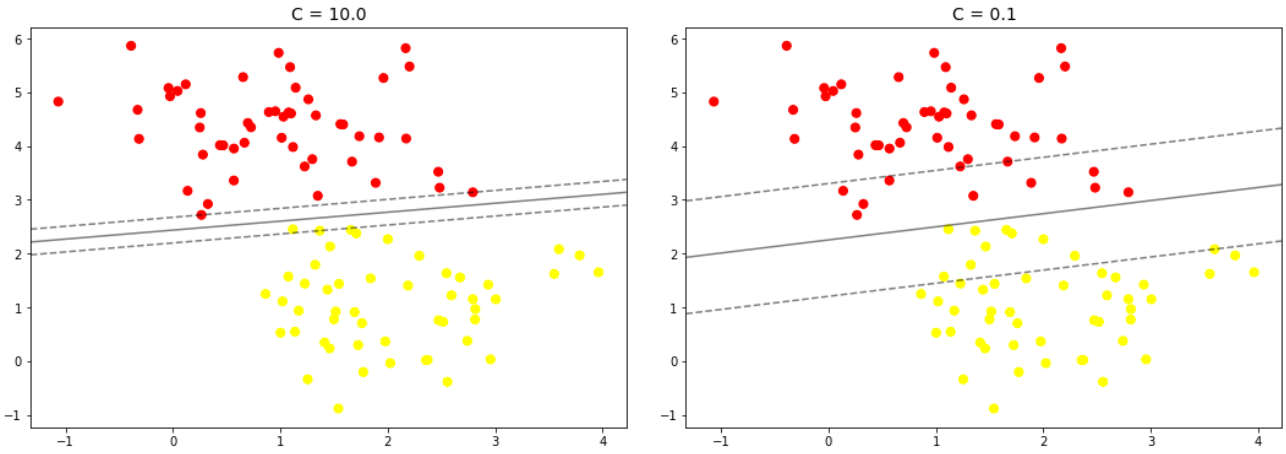
The plot shown below gives a visual picture of how a changing C parameter affects the final fit, via the softening of the margin:

```
X, y = make_blobs(n_samples=100, centers=2,
                  random_state=0, cluster_std=0.8)

fig, ax = plt.subplots(1, 2, figsize=(16, 6))
fig.subplots_adjust(left=0.0625, right=0.95, wspace=0.1)

for axi, C in zip(ax, [10.0, 0.1]):
    model = SVC(kernel='linear', C=C).fit(X, y)
    axi.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='autumn')
    plot_svc_decision_function(model, axi)
    axi.scatter(model.support_vectors_[:, 0],
                model.support_vectors_[:, 1],
                s=300, lw=1, facecolors='none');
    axi.set_title('C = {0:.1f}'.format(C), size=14)
```

[4] A *fudge-factor* is an ad hoc quantity or element introduced into a calculation, formula or model in order to make it fit observations or expectations. Examples include Einstein's Cosmological Constant, dark energy, the initial proposals of dark matter and inflation.



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

You can download the above Python codes `Ch4_SVM_Tutorial.ipynb` from the course website.

2.3.5 Example: Face Recognition with SVC

We will now apply model validation and hyperparameter optimization for support vector machines. Let's take a look at the facial recognition problem. We will use a dataset consisting of 1,348 collated photos of various public figures, each image with 62×47 pixels. The dataset is built into `Scikit-Learn` (you will need to install `pillow` into your Python environment to show up the images):

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_lfw_people

faces = fetch_lfw_people(min_faces_per_person=60)
print(faces.target_names)
print(faces.images.shape)
```

```
['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush'
 'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']
(1348, 62, 47)
```

Let's plot a few of these faces to see what we're working with:

```
fig, ax = plt.subplots(3, 4)
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i], cmap='bone')
    axi.set(xticks=[], yticks=[],
            xlabel=faces.target_names[faces.target[i]])
```

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; here we will use a principal component analysis (PCA) to extract 150 fundamental components to feed into our support vector machine classifier.

Remark: [Dimensionality Reduction] In practice, when we have thousands or millions of variables, we often select the first few principal components for further analysis. Using PCA for dimensionality reduction involves zeroing out some small principal components, resulting in a lower-dimensional projection of the data that **preserves the maximal data variance so we can best differentiate our data**.

We can do this most straightforwardly by packaging the preprocessor and the classifier into a single pipeline:

```
from sklearn.svm import SVC
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline

pca = PCA(n_components=150,whiten=True)
svc = SVC(kernel='rbf',class_weight='balanced')
model = make_pipeline(pca, svc)
```

We will split the data into a training and testing set:

```
from sklearn.model_selection import train_test_split
Xtrain, Xtest, ytrain, ytest = train_test_split(faces.data,
                                faces.target,random_state=42,
                                test_size = 0.25)
```

Finally, we can use a grid search cross-validation to explore combinations of parameters. Here we will

adjust `C` (which controls the margin hardness) and `gamma` (which controls the size of the radial basis function kernel), and determine the best model:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'svc__C': [0.1, 1, 5, 10, 50],
              'svc__gamma': [0.0001, 0.0005, 0.001, 0.005, 0.01]}
grid = GridSearchCV(model, param_grid) #the default 3-fold cross
validation

%time grid.fit(Xtrain, ytrain)
print(grid.best_params_)
```

```
CPU times: user 1min 39s, sys: 568 ms, total: 1min 39s
Wall time: 25.1 s
{'svc__C': 10, 'svc__gamma': 0.001}
```

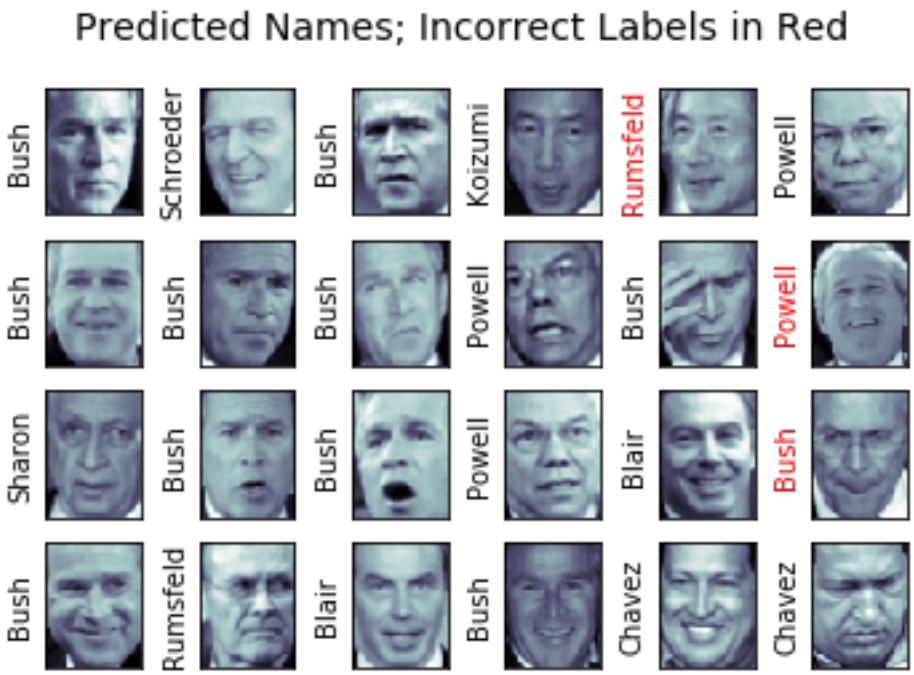
The optimal values fall toward the middle of our grid; if they fell at the edges, we would want to expand the grid to make sure we have found the true optimum.

Now with this cross-validated model, we can predict the labels for the test data, which the model has not yet seen:

```
model = grid.best_estimator_
yfit = model.predict(Xtest)
```

Let's take a look at a few of the test images along with their predicted values:

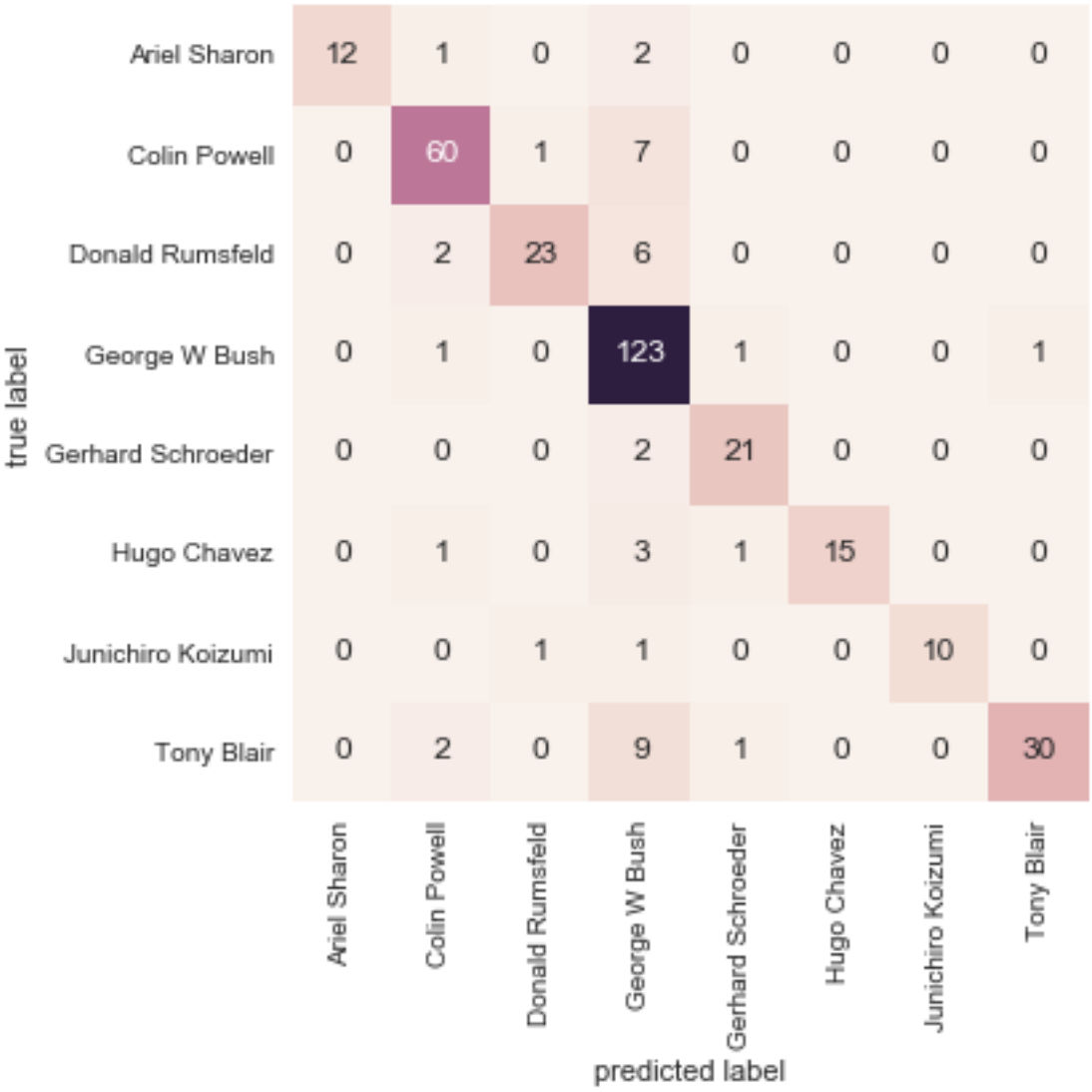
```
fig, ax = plt.subplots(4, 6)
for i, axi in enumerate(ax.flat):
    axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')
    axi.set(xticks=[], yticks=[])
    axi.set_ylabel(faces.target_names[yfit[i]].split()[-1],
                   color='black' if yfit[i] == ytest[i] else 'red')
fig.suptitle('Predicted Names; Incorrect Labels in Red', size=14);
```



Out of this small sample, our optimal estimator mislabeled only three faces. We can get a better sense of our estimator's performance by displaying the *confusion matrix* which we can compute with `Scikit-Learn` and plot with `seaborn`:

```
from sklearn.metrics import confusion_matrix
# use seaborn plotting defaults
import seaborn as sns; sns.set()

mat = confusion_matrix(yfit, ytest)
sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False,
            xticklabels=faces.target_names,
            yticklabels=faces.target_names)
plt.xlabel('predicted label')
plt.ylabel('true label');
```



This shows us where the mislabeled faces tend to be: for example, 7 images of Colin Powell are misclassified as George W. Bush.

Q: How many images of Tony Blair are misclassified as George W. Bush?
A:

We can also use the classification report, which lists recovery statistics label by label:

```
from sklearn.metrics import classification_report
print(classification_report(ytest, yfit,
      target_names=faces.target_names))
```

	precision	recall	f1-score	support
Ariel Sharon	1.00	0.80	0.89	15
Colin Powell	0.90	0.88	0.89	68
Donald Rumsfeld	0.92	0.74	0.82	31
George W Bush	0.80	0.98	0.88	126
Gerhard Schroeder	0.88	0.91	0.89	23
Hugo Chavez	1.00	0.75	0.86	20
Junichiro Koizumi	1.00	0.83	0.91	12
Tony Blair	0.97	0.71	0.82	42
avg / total	0.89	0.87	0.87	337

Precision, Recall and F1-score are some of the mostly used measures in evaluating how good the system works. Reading from the first column of the confusion matrix, there is one cell with entry 12 and the rest are zeros. This means you have 12 images of Ariel Sharon predicted and all 12 images are predicted as Ariel Sharon. This is precision 12/12 = 1.0. Now look only at the first row in the confusion matrix, among 15 images of Ariel Sharon, you have 12 images correctly. That's your recall. 12/15 = 0.8. F1-score Measure is harmonic mean of Precision and Recall. We will further discuss the importance of these performance measurements in the later Section on Class Imbalance Problem.

You can download the above Python codes `Ch4_FaceRecog.ipynb` from the course website.

2.3.6 (Optional) Theoretical Minimum: Support Vector Machine[5]

A SVM is a classifier that searches for an **optimal separating hyperplane** with the largest margin. In this section, we define a hyperplane and introduce the concept of a separating hyperplane, margin and an optimal separating hyperplane.

What Is Hyperplane?

In a d-dimensional space, a hyperplane is a flat subspace of dimension $d - 1$. For instance, in two dimensions, a hyperplane is a flat one-dimensional subspace—in other words, a line. In three dimensions, a hyperplane is a flat two-dimensional subspace—that is, a plane. In $d > 3$ dimensions, it can be hard to visualize a hyperplane, but the notion of a $(d - 1)$-dimensional flat subspace still applies.

The mathematical definition of a hyperplane is quite simple. In two dimensions, a hyperplane is

[5] Major contents are adapted from dynamic e-chapters of Abu-Mostafa, Y S, Magdon-Ismail, M., Lin, H-T (2012) *Learning from Data*, AMLbook.com.

defined by the equation:

(2.1) $w_1 x_1 + w_2 x_2 + b = 0$

for parameters w_1, w_2, and b. When we say the equation "defines" the hyperplane, we mean that any point $\mathbf{x} = [x_1, x_2]^T$ for which the equation holds is a point on the hyperplane.

Q: Equation (2.1) is simply the equation of a line, since indeed in two dimensions a hyperplane is a line. Convert $\mathbf{x}$ into 2D Cartesian coordinate and convince yourself it is the equation of a line.
A:

We can easily extend the hyperplane to the *d*-dimensional setting:

$$w_1 x_1 + w_2 x_2 + \cdots w_d x_d + b = 0$$

Or

(2.2) $\mathbf{w}^T \mathbf{x} + b = 0$

in which $\mathbf{w} = [w_1, w_2, \ldots, w_d]^T$, $\mathbf{x} = [x_1, x_2, \ldots, x_d]^T$; $\mathbf{w}, \mathbf{x} \in \mathbb{R}^d$ where $\mathbb{R}^d$ is the *d*-dimensional Euclidean space and $b \in \mathbb{R}$.

Separating Hyperplane

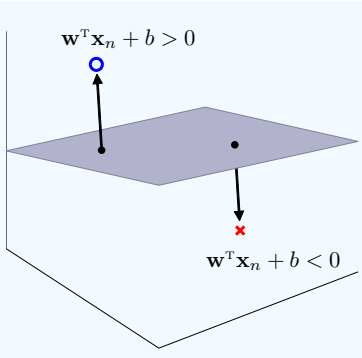
Consider a binary classification problem in the *d*-dimensional Euclidean space. The problem consists of N training examples: $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_N, y_N)$ where $y_n = f(\mathbf{x}_n)$ for $n = 1, \cdots, N$. Again let $\mathbf{x} \in \mathbb{R}^d$ where $\mathbb{R}^d$ is the *d*-dimensional Euclidean space and by convention let $y \in \{-1, 1\}$ denote its class label.

We are interested in finding a hyperplane that places instances of both classes on **opposite** sides of the hyperplane, thus resulting in a **separation** of the two classes. This means that:

$\mathbf{w}^T \mathbf{x}_n + b > 0$ if $y_n = 1$
$\mathbf{w}^T \mathbf{x}_n + b < 0$ if $y_n = -1$

In other words, the hyperplane h, defined by $(b, \mathbf{w})$, separates the training examples if and only if for $n = 1, \cdots, N$,

(2.3) $y_n(\mathbf{w}^T\mathbf{x}_n + b) > 0$ separating hyperplane



The signal $y_n(\mathbf{w}^T\mathbf{x}_n + b)$ is positive for each data point. However, the magnitude of the signal is not meaningful since we can make it arbitrarily small or large for the same hyperplane by rescaling $\mathbf{w}$ and b.

Q: Why?
A:

By rescaling the weights, we can control the magnitude of the signal for our data points. Let us pick a particular value of ρ,

$$\rho = \min_{n=1,\cdots,N} y_n(\mathbf{w}^T\mathbf{x}_n + b)$$

which is positive because of Equation (2.2). Now, rescale the weights to obtain the same hyperplane $(b/\rho, \mathbf{w}/\rho)$. **For these rescaled weights**,

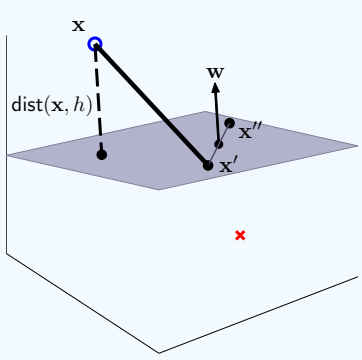
$$\min_{n=1,\cdots,N} y_n\left(\frac{\mathbf{w}^T}{\rho}\mathbf{x}_n + \frac{b}{\rho}\right) = \frac{1}{\rho}\min_{n=1,\cdots,N} y_n(\mathbf{w}^T\mathbf{x}_n + b) = \frac{\rho}{\rho} = 1$$

Thus, for any separating hyperplane, it is always **possible** to choose weights so that all the signals $y_n(\mathbf{w}^T\mathbf{x}_n + b)$ are of magnitude greater than or equal to 1, with equality satisfied by at least one $(\mathbf{x}_n, y_n)$. This motivates our new definition of a separating hyperplane, equivalent to Equation (2.3):

(2.4) $\min_{n=1,\cdots,N} y_n(\mathbf{w}^T\mathbf{x}_n + b) = 1$ canonical representation of separating hyperplane

Margin of Separating Hyperplane

To compute the margin of a separating hyperplane, we need to compute the distance from the hyperplane to the nearest data point. As a start, let us compute the distance from an arbitrary point $\mathbf{x}$ to a separating hyperplane $h = (b/\rho, \mathbf{w}/\rho)$ that satisfies Equation (2.4). Denote this distance by $\text{dist}(\mathbf{x}, h)$. Referring to the figure below, $\text{dist}(\mathbf{x}, h)$ is the length of the perpendicular line from $\mathbf{x}$ to h.



Q: Let $\mathbf{x}'$ be any point on the hyperplane, which means $\mathbf{w}^T\mathbf{x}' + b = 0$ (from Equation (2.1)). Let $\mathbf{u}$ be a unit vector that is normal to the hyperplane h. What is $\text{dist}(\mathbf{x}, h)$ in terms of $\mathbf{x}$, $\mathbf{x}'$, and $\mathbf{u}$?
A:

From the figure above, we realize any vector lying on the hyperplane can be expressed by $(\mathbf{x}'' - \mathbf{x}')$ for some points $\mathbf{x}'$ and $\mathbf{x}''$ **on the hyperplane**. This leads to $\mathbf{w}^T\mathbf{x}' + b = 0$ and $\mathbf{w}^T\mathbf{x}'' + b = 0$ or:

(2.5) $\mathbf{w}^T(\mathbf{x}'' - \mathbf{x}') = 0$

Q: From Equation (2.5), what a remarkable geometric property we have for $\mathbf{w}$?
A:

Setting $\mathbf{u} = \mathbf{w}/\|\mathbf{w}\|$, the distance from $\mathbf{x}$ to h becomes:

$$\text{dist}(\mathbf{x}, h) = |\mathbf{u}^T(\mathbf{x} - \mathbf{x}')| = \frac{|\mathbf{w}^T\mathbf{x} - \mathbf{w}^T\mathbf{x}'|}{\|\mathbf{w}\|} = \frac{|\mathbf{w}^T\mathbf{x} + b|}{\|\mathbf{w}\|}$$

We thus lead to an important conclusion: distance of any data points $(\mathbf{x}_n, y_n)$ in our training examples to the hyperplane is:

$$\text{dist}(\mathbf{x}_n, h) = \frac{|\mathbf{w}^T\mathbf{x}_n + b|}{\|\mathbf{w}\|} = \frac{y_n(\mathbf{w}^T\mathbf{x}_n + b)}{\|\mathbf{w}\|}$$

Q: What is the distance of the data point nearest to the hyperplane (**hint**: take a look of Equation (2.4))?
A:

Remark: This simple expression for the distance of the nearest data point to the hyperplane is the entire reason why we chose to normalize $(b, \mathbf{w})$ to $(b/\rho, \mathbf{w}/\rho)$ as we did in (2.4). For any separating hyperplane satisfying (2.4), the margin is $1/\|\mathbf{w}\|$. If you hold on a little longer, you are about to reap the full benefit, namely a simple algorithm for finding the optimal hyperplane.

Maximum Margin of Separating Hyperplane

The maximum-margin separating hyperplane $(b^*, \mathbf{w}^*)$ is the one that satisfies the separation condition (2.4) with minimum weight-norm (since the margin is the inverse of the weight-norm). Instead of minimizing the weight-norm, we can equivalently minimize $\frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$, which is analytically more friendly. Therefore, to find this optimal hyperplane, we need to solve the following optimization problem.

(2.6) $\underset{b,\mathbf{w}}{\text{minimize:}} \quad \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$

$\text{subject to:} \quad \min_{n=1,\cdots,N} y_n(\mathbf{w}^T\mathbf{x}_n + b) = 1$

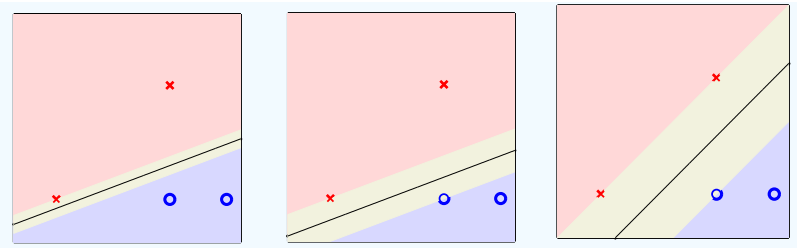
The constraint ensures that the hyperplane separates the data as per (2.4). Observe that the bias b does not appear in the quantity being minimized, but it is involved in the constraint. To make the optimization problem easier to solve, we can replace the single constraint $\min_{n=1,\cdots,N} y_n(\mathbf{w}^T\mathbf{x}_n + b) = 1$ with N 'looser' constraints $y_n(\mathbf{w}^T\mathbf{x}_n + b) \geq 1$ for $n = 1, \cdots, N$ and solve the optimization problem:

(2.7) $\underset{b,\mathbf{w}}{\text{minimize:}} \quad \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{w}$

subject to: $y_n(\mathbf{w}^T\mathbf{x}_n + b) \geq 1 \qquad (n = 1, \cdots, N)$

The constraint in (2.6) implies the constraints in (2.7), which means that the constraints in (2.7) are looser. Fortunately, at the optimal solution, the constraints in (2.7) become equivalent to the constraint in (6).

Example: We will use a toy example to solve (2.7). In two dimensions, a hyperplane is specified by the parameters (b, w_1, w_2). Let us consider a data set that was the basis for the figure below. There are many possibilities of hyperplanes to separate the data points. Our task is to use (2.7) to find the **optimal separating hyperplane** with the largest margin.



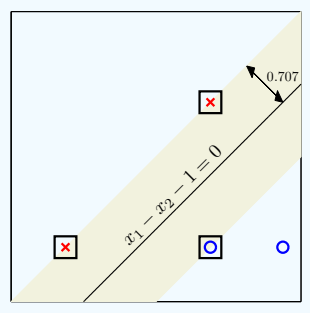
The data matrix and target values, together with the separability constraints from (2.7) are summarized below. The inequality on a particular row is the separability constraint for the corresponding data point in that row.

$$\mathrm{X} = \begin{bmatrix} 0 & 0 \\ 2 & 2 \\ 2 & 0 \\ 3 & 0 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} -1 \\ -1 \\ +1 \\ +1 \end{bmatrix} \qquad \begin{aligned} -b &\geq 1 \quad \text{(i)} \\ -(2w_1 + 2w_2 + b) &\geq 1 \quad \text{(ii)} \\ 2w_1 + b &\geq 1 \quad \text{(iii)} \\ 3w_1 + b &\geq 1 \quad \text{(iv)} \end{aligned}$$

Q: What happens when we combine (i) and (iii)?
A:

Q: What happens when we combine (ii) and (iii)?

This means that $\frac{1}{2}(w_1^2 + w_2^2) \geq 1$ with the equality $\frac{1}{2}(w_1^2 + w_2^2) = 1$ when $w_1 = 1$ and $w_2 = -1$. We can then substitute these values into (i) – (iv) and verify that $(b^* = -1, w_1^* = 1, w_2^* = -1)$ satisfies all four constraints, minimizes $\frac{1}{2}(w_1^2 + w_2^2)$, and therefore gives the optimal hyperplane. The optimal hyperplane is shown in the following figure.



Q: What is our final hypothesis $g(\mathbf{x})$?
A:

Q: How to solve the maximum margin?
A:

Remark: Data points (i), (ii) and (iii) are boxed because their separation constraints are exactly met: $y_n(\mathbf{w}^T\mathbf{x}_n + b) = 1$.

Quadratic Programming (QP): For bigger data sets, manually solving the optimization problem in (2.7) as we did in Example is no longer feasible. Fortunately, Equation (2.7) belongs to a well-studied family of optimization problems known as **quadratic programming (QP)**. Whenever you minimize a (convex) quadratic function, subject to linear inequality constraints, you can use quadratic programming. Quadratic programming is such a well-studied area that excellent, publicly available solvers exist for many numerical computing platforms.

2.3.7 Summary: Support Vector Machine

We have seen a brief introduction to the principles 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:

- The scaling with the number of samples N is $O(N^3)$ at worst, or $O(N^2)$ for efficient implementations. For large number 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, we generally only turn to SVMs once other simpler, faster, and less tuning-intensive methods have been shown to be insufficient for our needs. Nevertheless, 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.