What we did last time

For the Perceptron algorithm we specified a prediction function and a loss function

In order to solve the optimization problem of finding a set of parameters that minimized the loss on the training data, we reduced the problem to an iterative search

We used the gradient of the loss function to figure out how to update our best guess for the parameters at each iteration

This gave us a straightforward set of update equations that we could use to train the Perceptron with our training data
Predicting on new data

After training, we have an $a$ and $b$ which we can use to predict the class labels of new data

For this we just use our prediction function with our actual learned values of $a$ and $b$

$\text{sign}(a \circ x + b)$
The Naive Bayes Classifier
A probabilistic classifier

If we suppose there is a probabilistic relationship between our class labels and feature vectors, and we have determined what this relationship is, then the following is a valid classifier

For a feature vector \( x \), evaluate \( p(y \mid x) \) and output the class \( y \) that maximizes this function

Some classifiers attempt to learn or model \( p(y \mid x) \) directly using max likelihood estimation (homework problem 9.17 actually did this for a classifier known as logistic regression). With the naive Bayes classifier we will take a slightly different approach
The naive Bayes model

Instead of modeling \( p(y \mid x) \) directly, we will use Bayes’ theorem to observe that

\[
p(y \mid x) = \frac{p(x \mid y)p(y)}{p(x)}
\]

And come up with a model for \( p(x \mid y) \) and \( p(y) \)

The naive part about naive Bayes is that it makes an assumption that isn’t usually true but we nevertheless might be able to get away with. We will assume that the features of \( x \) are conditionally independent given the class label, i.e. that

\[
p(x \mid y) = \prod_{j=1}^{d} p(x^{(j)} \mid y)
\]
Naive Bayes prediction

Making the assumption that

\[ p(x|y) = \prod_{j=1}^{d} p(x^{(j)}|y) \]

And using Bayes theorem means that

\[
\arg \max_y p(y|x) = \arg \max_y \frac{p(x|y)p(y)}{p(x)} \\
= \arg \max_y p(x|y)p(y) \\
= \arg \max_y \prod_{j=1}^{d} p(x^{(j)}|y)p(y)
\]
Training and modeling

\[
\arg \max_y p(y|x) = \arg \max_y \prod_{j=1}^{d} p(x^{(j)}|y)p(y)
\]

In order to make a prediction, then, we are going to need a model of \( p(y) \) and \( p(x^{(j)}|y) \) for every dimension \( j \) and for each value of \( y \).

If feature (dimension) \( j \) is numerical, perhaps we model \( p(x^{(j)}|y) \) as a normal random variable. If it is a count perhaps we model \( p(x^{(j)}|y) \) as a Poisson random variable, etc.

The prior probability \( p(y) \) is probably best modeled as a Bernoulli random variable, in the binary classification case, where max likelihood would tell us that \( p(y=1) = \) fraction of training examples where \( y=1 \).
Example

Suppose we had the following training data

<table>
<thead>
<tr>
<th></th>
<th>x1</th>
<th>x2</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.4</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1.1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>10</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>-2.1</td>
<td>13</td>
<td>-1</td>
</tr>
</tbody>
</table>

And we choose to suppose that \( p(x_1 | y) \) is normally distributed, \( p(x_2 | y) \) is Poisson, and \( p(y) \) is Bernoulli

Then \( p(x_1 | y=1) \) has an MLE mean of 2.25 and standard deviation of 1.323
\( p(x_1 | y=-1) \) has an MLE mean of -1.05 and standard deviation of 1.1025
\( p(x_2 | y=1) \) has an MLE lambda of 9.5
\( p(x_2 | y=-1) \) has an MLE lambda of 11.5
and \( p(y=1) = 0.5 \) by taking the MLE of a Bernoulli random variable

If we saw some new point \((-1, 12)\), we would predict the class label by evaluating

\[
\arg \max_y \prod_{j=1}^d p(x^{(j)} | y)p(y)
\]
One caveat

If we have a large or even modest number of dimensions, we can run into numerical issues evaluating

$$\arg \max_y \prod_{j=1}^d p(x^{(j)}|y)p(y)$$

Each $p(x^{(j)}|y)$ is some small number between 0 and 1, if we multiply many of these probabilities together, we are very likely to get a numerical underflow. To stop this from happening we often use logarithms since

$$\arg \max_y \prod_{j=1}^d p(x^{(j)}|y)p(y) = \arg \max_y \sum_{j=1}^d \log(p(x^{(j)}|y)) + \log(p(y))$$
By making a simplistic assumption we are able to get a probabilistic classifier that reduces to basic MLE calculations for training.

We can do this because even if we poorly model \( p(y \mid x) \) or even \( p(x \mid y) \), we can be okay as long as for any \( x \) the score for the right class is higher than the score for the wrong class and this can be the case even with poor assumptions.

The x-axis is the value of some feature \( x \) and the y-axis is the proportion of the training data that has a value around \( x \), i.e. it’s a class conditional histogram.

Here we assume that \( p(x \mid y) \) is normal for some feature \( x \), when the distribution clearly isn’t actually normal, but we would still get the right predictions.
Takeaway

By making a simplistic assumption we are able to get a probabilistic classifier that reduces to basic MLE calculations for training.

We can do this because even if we poorly model \( p(y \mid x) \) or even \( p(x \mid y) \), we can be okay as long as for any \( x \) the score for the right class is higher than the score for the wrong class and this can be the case even with poor assumptions.

The x-axis is the value of some feature \( x \) and the y-axis is the proportion of the training data that has a value around \( x \), i.e. it’s a class conditional histogram.

Here we assume that \( p(x \mid y) \) is normal for some feature \( x \), when the distribution clearly isn’t actually normal, but we would still get the right predictions.
The support vector machine
The support vector machine or SVM is a classifier that is similar to the one we have just constructed. Our prediction function is still

\[ \text{sign}(a \circ x + b) \]

So our goal is to find an \( a \) and \( b \) that produces a decision boundary that separates the data well.

However, we will make some modifications to the loss function to account for the fact that having a loss function that just looks at the training data might not be wise.
The loss function

We will choose an $a$ and $b$ by choosing values that minimize a cost function. Our cost function will have the form

$$\text{training error cost} + \text{penalty term}$$

First let’s focus just on how we can quantify the training error
The training error

For data item $i$, we will write the prediction that our function makes as

$$
\gamma_i = \mathbf{a}^T \mathbf{x}_i + b
$$

For training data, we have the true labels of the data, so we compare our prediction with the true label. If we write a function that compares our prediction with the true label

$$
C(\gamma_i, y_i)
$$

Then our training error cost will be of the form

$$
\frac{1}{N} \sum_{i=1}^{N} C(\gamma_i, y_i)
$$
Specifying C

What properties should C have?

We have a correct prediction if our $\gamma_i$ is negative when the true label is -1 or positive when the true label is +1. So we are happy if $\gamma_i$ and $\chi_i$ have the same sign.

So C should be large when $\gamma_i$ and $\chi_i$ have different signs. Furthermore, if the signs are different and $|\gamma_i|$ has a large magnitude C should be even larger.

The reason is that $ax+b$ gets larger in magnitude as $x$ gets further from the decision boundary.
Specifying C

If $y_i$ and $Y_i$ have the same sign but $y_i$ has a small magnitude, then that means $x_i$ is close to the decision boundary, so while $x_i$ was correctly predicted, points close to it in the test set may not be. So in order to give ourselves some wiggle room, we will have

$$C(y_i, Y_i) = \max(0, 1 - y_i y_i)$$

We see that if the predicted and the actual value have different sign—a wrong prediction—then $C > 1$. If the prediction and the actual value have the same sign and the magnitude of the prediction is large, i.e. $y_i y_i > 1$ the cost is 0. And if the prediction is correct but very close to the boundary, i.e. $0 \leq y_i y_i \leq 1$ the cost is between 0 and 1

The kind of loss function is called the **hinge loss**
Hinge loss

Hinge loss for a single example with $y=1$
Loss functions

Our loss function for the Perceptron only depended on the training data.

It is not uncommon in practice to include other terms in a loss function that have nothing to do with the training data.

Sometimes we might want to express other preferences via the loss function.
The penalty term

For test data that we cannot yet observe, imagine we classify an example wrongly. If $||a||$ is small, then this example is at least close to the decision boundary and so nearby examples might be classified correctly. So all else being equal, we prefer a small $||a||$

So if we try to minimize

$$S(a, b; \lambda) = \left(\frac{1}{N}\right) \sum_{i=1}^{N} \max(0, 1 - y_i (a \cdot x_i + b)) + \frac{\lambda}{2} a^T a$$

We are taking into account both our desire for small $||a||$ and our desire to do well on the training set. The value of $\lambda$ controls the degree to which we emphasize these two aspects of the loss function. It is sometimes referred to as a regularization parameter.
Finding good parameters

\[
S(a, b; \lambda) = \left(\frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i (a \cdot x_i + b))\right) + \frac{\lambda}{2} a^T a
\]

In practice what we will do, is assume we have chosen a \( \lambda \) and then find an \( a \) and \( b \) to minimize \( S \). And then separately we will try to estimate a good value of \( \lambda \).

We will use stochastic gradient descent yet again which means we need to compute the gradient of the loss function with respect to a single training example.
Our loss function is different this time, so our update rules will be as well, for our step direction, we choose a random $k$ from among our training examples and have

\[
S(a, b; \lambda) = \left(\frac{1}{N}\sum_{i=1}^{N} \max(0, 1 - y_i (a \cdot x_i + b))\right) + \frac{\lambda}{2} a^T a
\]

\[
p_k = \begin{cases} 
\begin{bmatrix} \lambda a \\ 0 \end{bmatrix} & \text{if } y_k (a \cdot x_k + b) \geq 1 \\
\begin{bmatrix} \lambda a - y_k x \\ -y_k \end{bmatrix} & \text{otherwise}
\end{cases}
\]

Giving update rules of

\[
a^{(n+1)} = a^{(n)} - \eta \begin{cases} 
\lambda a \\ \lambda a - y_k x \end{cases} \text{ if } y_k (a \cdot x_k + b) \geq 1
\]

and

\[
b^{(n+1)} = b^{(n)} - \eta \begin{cases} 
0 \\ -y_k \end{cases} \text{ if } y_k (a \cdot x_k + b) \geq 1
\]
The regularization parameter

Our training for the SVM so far was only with respect to the parameters $a$ and $b$. Our prediction function is still just

$$\text{sign}(a \circ x + b)$$

This doesn’t depend on $\lambda$. Nevertheless, our update equations for training did use this parameter, so we can’t really find $a$ and $b$ without some value for this parameter.

For a parameter like this that isn’t part of the prediction function, we will just try different values of it and see which one works best rather than derive an optimal value.

We do this by holding out some portion of our training data as a validation set. We run the training procedure on the training data for a few different values of the parameter and evaluate which setting of the parameter gives the best accuracy on the held out data.
Classifier evaluation

The normal flow for training and evaluating a classifier is to take the data we have available, keep most of it for training and some for evaluation.

You run the training algorithm on the classifier with the training data and then evaluate it against the validation set on which it was not trained.

If there is something like the regularization parameter which the SVM has, the process can be repeated above for multiple different values of $\lambda$.

We will keep the classifier (characterized by $a$ and $b$) which performs best on the validation data and we will report its accuracy: percentage of items in the validation set whose labels were correctly predicted.
After we’ve trained a classifier we have a nice machine that can look at new data and hopefully accurately apply class labels. There are many other design considerations besides accuracy that we may want to take into account.

How long does a classifier take to train?

How long does it take to make a prediction?

Is the classifier interpretable? Does looking at the resulting classifier tell us anything interesting about our data?

Among others
Multi-class classification

We have considered the Perceptron and SVM classifiers which are binary classifiers.

However, we can use binary classifiers to handle multiple classes in a couple of different ways.

In the all vs all approach, we would train a classifier for each pair of potential class labels. For instance, if we were trying to train a classifier that analyzed radiology images and made a diagnosis of healthy, benign tumor, malignant tumor, in the all vs all approach we would train a classifier for healthy vs benign, healthy vs malignant, and benign vs malignant. We then run all of the classifiers and see which class label wins the most 1vs1 competitions.
Multi-class classification

Another approach for using binary classifiers to handle multiple classes is the one-vs-all approach.

In this case we train a classifier for each label separately. For example we would train a healthy vs non-healthy, benign vs non-benign, and malignant vs non-malignant classifier. To classify a new example, we show it to each of the classifiers and record the score. We output the class label which had the highest score.

All vs all requires training $O(N^2)$ classifiers, whereas one vs all requires $O(N)$. In practice both methods work fairly well.
# Class confusion matrices

A tool for evaluating multi-class (including binary) classifiers

<table>
<thead>
<tr>
<th></th>
<th>Predict 0</th>
<th>Predict 1</th>
<th>Predict 2</th>
<th>Predict 3</th>
<th>Predict 4</th>
<th>Class error</th>
</tr>
</thead>
<tbody>
<tr>
<td>True 0</td>
<td>151</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>7.9%</td>
</tr>
<tr>
<td>True 1</td>
<td>32</td>
<td>5</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>91%</td>
</tr>
<tr>
<td>True 2</td>
<td>10</td>
<td>9</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>81%</td>
</tr>
<tr>
<td>True 3</td>
<td>6</td>
<td>13</td>
<td>9</td>
<td>5</td>
<td>2</td>
<td>86%</td>
</tr>
<tr>
<td>True 4</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>
Normalization

For the classifiers we have learned so far, there is one caveat worth commenting on.

It is usually a good idea to normalize your data—make sure each dimension has 0 mean and variance of 1—before feeding it into an SVM or perceptron.

Due to the loss function, a feature with a much larger variance than the others will have a larger effect on the resulting classifier than the other features.
Decision trees
Examples

(0, 0)
(2, 1)
A decision tree is a type of classifier that operates as follows

Each internal node corresponds to a single dimension of the data and a binary **split** for that dimension, i.e. a set of values for the dimension which points to the left child and the remainder of values which point to the right child.

To classify a new data item, you start at the root node. Being an internal (non-leaf) node, the root will refer to one of the dimensions of the data item. Depending on the split, the data item will next visit the left or right child.

Continue to traverse the tree according to dimensions and splits until reaching a leaf. Each leaf in the tree gives a class label.
The decision surface
How to make a decision tree from training data

We will recursively build our tree using the training data. We will need some principles in order to

Decide which dimension to split on

Decide how to split the dimension

Decide how deep to make the tree or equivalently when to end recursion

Decide which labels to put on the leaves
Recursively splitting the data

\[(x_1, +), (x_2, -), (x_3, +), (x_4, +), (x_5, -), (x_6, +), (x_7, -), (x_8, -)\]
What kind of splits

We want to split the data in some informative way

Suppose this was our entire training data set

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>8.5</td>
<td>-1</td>
</tr>
<tr>
<td>0.5</td>
<td>3.25</td>
<td>-1</td>
</tr>
<tr>
<td>1.2</td>
<td>-1.5</td>
<td>-1</td>
</tr>
<tr>
<td>2.5</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>3.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>4.5</td>
<td>4.5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3.25</td>
<td>1</td>
</tr>
<tr>
<td>5.25</td>
<td>5.5</td>
<td>1</td>
</tr>
</tbody>
</table>

Why is X1 > 3 a great split? Why is X2 > 3 not as good?
What kind of split

In general we would like it if whatever the “class balance” at the node is (say it starts out 50-50 class 1 and class -1) is more imbalanced in the children.
It turns out that a good way to measure this “class imbalance” feature of a dataset comes to us from communication theory or information theory.

Suppose we had four classes: A, B, C, and D.

One way to encode those classes, for communication over a network for example, would be A=00, B=01, C=10, D=11. If we did this, it would take on average, indeed in every case, 2 bits to communicate the class label.

The idea with entropy is that if we have some class imbalance, say \( P(A) = \frac{1}{2}, P(B) = \frac{1}{4}, P(C) = \frac{1}{8}, P(D) = \frac{1}{8} \), maybe there is a better encoding we can use.
Entropy

If we had classes with frequencies given by $P(A) = 1/2$, $P(B) = 1/4$, $P(C) = 1/8$, $P(D) = 1/8$

Then if we used the following encoding scheme

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>110</td>
</tr>
<tr>
<td>D</td>
<td>111</td>
</tr>
</tbody>
</table>

The expected number of bits to communicate a class label is given by

$$(1/2)(1) + (1/4)(2) + (1/8)(3) + (1/8)(3)$$

Which is 1.75 bits
Entropy

The entropy of a probability distribution is the smallest number of bits on average you would need to identify an item sampled from a distribution. It can be thought of as measuring the degree of uncertainty of the distribution. We can use this to evaluate the splits we are considering for our data.

This quantity is large when the probability distribution is close to uniform and small when one class is very likely while the others are unlikely. The entropy is denoted with an $H$ and is given by the following formula:

$$- \sum_i p(i) \log_2 p(i)$$

Where the $p(i)$ are the probability of class $i$ which in this case is just the relative frequency of the class.
Example

Calculate the entropy for this dataset

\[-\sum_i p(i) \log_2 p(i)\]

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>8.5</td>
<td>-1</td>
</tr>
<tr>
<td>0.5</td>
<td>3.25</td>
<td>-1</td>
</tr>
<tr>
<td>1.2</td>
<td>-1.5</td>
<td>-1</td>
</tr>
<tr>
<td>2.5</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>3.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>4.5</td>
<td>4.5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3.25</td>
<td>1</td>
</tr>
<tr>
<td>5.25</td>
<td>5.5</td>
<td>1</td>
</tr>
</tbody>
</table>

We get

\[-5/10 \log(5/10) - 5/10 \log(5/10) = 1\]
Split evaluation

Let $H(P)$ be the entropy of all the training data. In order to evaluate a potential split at the root node of the tree, we want to compute the difference between this entropy and the average entropy remaining in the left and right children nodes, which we will call $H(P_l)$ and $H(P_r)$ respectively.

The average entropy after the split between the left and right subtrees will be

$$\frac{N(P_l)}{N(P)} H(P_l) + \frac{N(P_r)}{N(P)} H(P_r)$$

Where $N(P_l)$ is, for instance, the number of items that wind up in the left subtree
Information gain

This difference between the entropy or average number of bits needed before and after a split is called the **information gain** of the split. We would like to choose the split with the largest information gain.

\[
I(P_l, P_r; P) = H(P) - \left( \frac{N(P_l)}{N(P)} H(P_l) + \frac{N(P_r)}{N(P)} H(P_r) \right)
\]
Example

For the training set below, evaluate the information gain of the split $x_2 > 1$

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>8.5</td>
<td>-1</td>
</tr>
<tr>
<td>0.5</td>
<td>3.25</td>
<td>-1</td>
</tr>
<tr>
<td>1.2</td>
<td>-1.5</td>
<td>-1</td>
</tr>
<tr>
<td>2.5</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>3.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>4.5</td>
<td>4.5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3.25</td>
<td>1</td>
</tr>
<tr>
<td>5.25</td>
<td>5.5</td>
<td>1</td>
</tr>
</tbody>
</table>

From our prior example we know that $H(P) = 1$

Let’s say that data items \{2, 3, 5, 7, 8, 9, 10\} go to the right subtree and items \{1, 4, 6\} go to the left
Example

Let’s say that data items \{2, 3, 5, 7, 8, 9, 10\} go to the right and items \{1, 4, 6\} go to the left

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>8.5</td>
<td>-1</td>
</tr>
<tr>
<td>0.5</td>
<td>3.25</td>
<td>-1</td>
</tr>
<tr>
<td>1.2</td>
<td>-1.5</td>
<td>-1</td>
</tr>
<tr>
<td>2.5</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>3.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>4.5</td>
<td>4.5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3.25</td>
<td>1</td>
</tr>
<tr>
<td>5.25</td>
<td>5.5</td>
<td>1</td>
</tr>
</tbody>
</table>

What is $H(P_r)$?

$$-3/7 \log(3/7) - 4/7 \log(4/7) = 0.985$$
Example

Let’s say that data items \{2, 3, 5, 7, 8, 9, 10\} go to the right and items \{1, 4, 6\} go to the left

<table>
<thead>
<tr>
<th>X1</th>
<th>X2</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>8.5</td>
<td>-1</td>
</tr>
<tr>
<td>0.5</td>
<td>3.25</td>
<td>-1</td>
</tr>
<tr>
<td>1.2</td>
<td>-1.5</td>
<td>-1</td>
</tr>
<tr>
<td>2.5</td>
<td>5</td>
<td>-1</td>
</tr>
<tr>
<td>3.75</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>4.5</td>
<td>4.5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3.25</td>
<td>1</td>
</tr>
<tr>
<td>5.25</td>
<td>5.5</td>
<td>1</td>
</tr>
</tbody>
</table>

What is H(P_l)?

\[-\frac{2}{3} \log(\frac{2}{3}) - \frac{1}{3} \log(\frac{1}{3}) = 0.918\]
Example

\[ I(P_l, P_r; P) = H(P) - \left( \frac{N(P_l)}{N(P)} H(P_l) + \frac{N(P_r)}{N(P)} H(P_r) \right) \]

So we had \( H(P) = 1, \ H(P_l) = 0.918, \ H(P_r) = 0.985, \ N(P_l) = 3, \ N(P_r) = 7, \ N(P) = 10 \)

Giving an information gain of 0.0351
Choosing a split in practice

In practice the way we will choose which feature to split on will be by looking at a random subset of the features. For each node in the tree, we will choose \( m \) of the dimensions of the data at random and then evaluate each as a candidate for the split for that node.

We will choose \( m = \text{square root of } d \)

Then for each of our \( m \) features we will compute the best split for that feature. And we will choose to split on the feature that gives the greatest information gain.
Evaluating an ordinal feature

If the feature we are trying to evaluate is continuous or otherwise ordinal we can sort the data according to that feature and N-1 potential splits by evaluating the information gain of splitting on the halfway point between each consecutive pair of points, i.e.

\[
\frac{x_i + x_{i+1}}{2}
\]

For example if our data in a given dimension \( x^{(j)} \) was \{1, 3, 6, 8\} we would evaluate the information gain for \( x^{(j)} > 2 \), \( x^{(j)} > 4.5 \), and \( x^{(j)} > 7 \).
Evaluating a categorical feature

If we have a categorical feature, one trick we can do is look at each potential value for the feature and just flip a coin to say whether data with that value goes left or right. In order to give the feature a good evaluation, we will want to repeat this some number of times and keep the split that gives the best information gain.

For example if we had some dimension that was categorical with possible values \{rain, sunny, windy, cloudy, snow\}. We would flip coins for each value and maybe say that data items with \{rain, snow\} go left and items with \{sunny, windy, cloudy\} go right.

We would then evaluate the information gain if we did that split and repeat the process a few times in order to find a good split for the categorical variable in question.
So when do we stop splitting?

If all of the data in a node has the same class label, there is no need to continue splitting. So we could adopt the strategy of continuing to split until each item in the training set is unambiguously classified by the tree.

Alternatively, we can specify a maximum depth for the tree ahead of time. We can use a validation set in order to test out various depths to see which one seems to work best.
Prediction

If we do happen to stop creating nodes after a depth threshold we will have a situation in which the leaves of the tree may not be unanimous.

If this is the case we will do the obvious thing of outputting the class label that has a majority in the leaf, flipping a coin in the case of ties.
Forests

The approach to constructing a tree we have just outlined won’t find the “best possible” tree for the data. At each node we have only looked at a random subset of the features. If we looked at every feature we would very often get a decision tree that predicts the training data with higher accuracy.

It turns out that decision trees can fit training data really well (even data that’s not linearly separable). So well in fact that overfitting can become a problem and generalization to unseen test data can suffer.

It also turns out that a very effective strategy in both theory and practice is to train a number of slightly different decent classifiers and combine them together to make one really good classifier.
Forests

A decision forest is when we train many distinct decision trees on the same training dataset, using randomization to get different trees.

To make a prediction with a decision forest, we present the data item to all of the trees and predict the class label that gets the most votes.

An alternative and even more effective way to predict is to give each tree not just one vote. In particular, if a tree reaches a leaf with $p$ training data instances that vote for its majority class, then that tree will cast $p$ votes for that class instead of just one.
Nearest neighbors
Closeness and labels

We’ve made the argument a few times so far that if two points are close to one another, we expect them to have the same label much of the time.

Of course, we could have two points that are close and on opposite sides of a decision boundary, but we expect that for most pairs of points that are near one another, their labels will be the same.

Our prediction function for this classifier will find the closest point in the training set to the query point and predict the label associated with that item of training data.
The decision surface
k-nearest neighbors, etc.

Rather than just use one neighbor for the classification, we might use $k$. This is the most common implementation of nearest neighbors and is called k-nearest neighbors.

Furthermore, we might only output a label if at least $l$ of these $k$ can agree on a common label. This is maybe less common and is called a $(k,l)$-nearest neighbor classifier.
Training and other practical issues

“Training” then for this classifier is just any pre-processing we do that makes looking up near neighbors easier.

Of course, one of the things we are going to need to make this work at all is a notion of “distance” between points. There are many potential choices we could use depending on the nature of our data and our assumptions about it.

If our dimensions have wildly different scales, this might be problematic. We may wish to standardize our data in each dimension prior to inserting it into the search data structure.