7.1 Nearest neighbor classification

7.1.1 Digit recognition

Countless pieces of mail pass through the postal service daily. A key step in handling them efficiently is automatically scanning and parsing their destination zipcodes. Each zipcode can be segmented into five (or sometimes nine) digits. Here is a smattering of them:

How can each such image be mapped to the corresponding digit? One approach is to use hand-coded rules. This would involve two steps:

- Write subroutine to identify key features (such as loops) of an image.
- Write a bunch of rules, like “if there’s a loop then it’s not a 1, 3, 5 or 7”.

Both steps are problematic. As you can see from the examples above, real handwritten digits are rife with deviations from ideal script, such as almost-loops. These variations can easily trip up a set of rigid rules. A much better idea is to learn a classifier automatically from data.

7.1.2 The input space and the label space

We want to create a classifier that takes an image $x$ and outputs a label $y$. What are the spaces $\mathcal{X}$ and $\mathcal{Y}$ from which these images and labels are drawn?

The Post Office has made available a training set of 60,000 digit-images. Each of these is a $28 \times 28$ greyscale image of a single digit. We can represent an image $x$ by a vector of 784 coordinates, one per pixel. The input space is then $\mathcal{X} = \mathbb{R}^{784}$. The label space is, naturally, $\mathcal{Y} = \{0, 1, \ldots, 9\}$. Thus the classifier we seek is a function $f : \mathcal{X} \to \mathcal{Y}$.

The training set can be written as $(x_1, y_1), \ldots, (x_n, y_n)$ where $n = 60,000$ and each $x_i \in \mathcal{X}$, $y_i \in \mathcal{Y}$. How can we use this data to find a good classifier $f$?
7.1.3 A nearest neighbor classifier

Here’s a simple classifier: for any image $x$, the label $f(x)$ is given by the following procedure.

- Find the $x_i$ that is closest to $x$ (out of $x_1, \ldots, x_n$).
- Return $y_i$.

What is meant by “closest to”? Well, we can use any notion of distance. One natural option is just Euclidean distance. For two-dimensional vectors $a = (a_1, a_2)$ and $b = (b_1, b_2)$, the Euclidean distance is given by the familiar formula

$$\|a - b\| = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}.$$ 

A similar formula applies in higher dimensions. For $a, b \in \mathbb{R}^d$, we have

$$\|a - b\| = \sqrt{\sum_{i=1}^{d} (a_i - b_i)^2}.$$ 

How good is this classifier? Is it always correct? Well, it is certainly has zero error on the training set: that is, $f(x_i) = y_i$ for training points $(x_i, y_i)$. But $f$ might not be correct for other images $x$. How can we assess its accuracy?

To this end, the Post Office has also provided a separate test set of different images and their labels. Any classifier can be tried out on this test set to see what fraction of images it gets correct. The performance on this test set is then a good indication of the performance of the classifier in practice (using the standard theory of sampling). It turns out that the classifier we have just constructed has an error rate of 23% on the test set.

Randomly guessing a label would have an error rate of 90%, so 23% isn’t too shabby, but it is certainly not good enough for the Post Office’s purposes. How can we do better?

7.1.4 Two improvements

Euclidean distance is not really an ideal distance measure between images. Consider two images that are identical, except that one is shifted slightly to the right, or is rotated slightly, or is slightly thicker. The Euclidean distance between these images will be substantial. It would be a lot more sensible to compute the distance between two images $x$ and $x'$ as follows:

- First maximally “align” the two images by translating and rotating them.
- Then compute Euclidean distance.

A further improvement is obtained by looking not just at the nearest neighbor of $x$, but at the $k$ nearest neighbors (for some small value of $k$ like 7), and returning the most common label amongst these neighbors.

When these two changes are made, the error of the classifier on the test set drops below 1%.

7.1.5 The computational complexity of finding the nearest neighbor

Suppose points $x_1, \ldots, x_n$ lie in $\mathbb{R}^d$. How does one find the nearest neighbor of a new point $x$? Here’s the brute-force method:

- Compute all distances $\|x_i - x\|$.
- Pick the $x_i$ for which this is smallest.
This algorithm takes time $O(n)$, which is prohibitive when $n$ is large. We don’t want to have to look through 60,000 images just to classify one new image!

There are two ways around this.

1. There are various data structures which enable efficient nearest neighbor search. The most popular of these—spatial partition trees and hashing—rely heavily on randomization, and can bring the search time down to $O(\log n)$.

2. Instead of using the entire training set, we can just pick a few representative examples of each digit: these are called prototypes. However, the question of how to choose prototypes has still not been resolved satisfactorily.

### 7.2 Decision trees

#### 7.2.1 Credit card fraud detection

Credit card fraud is a massive problem. How can it be reduced, given that before every transaction, the credit company has a brief moment in which to review the details of the purchase and decline it if it is suspicious? Can a computer pick out transactions that are likely to be fraudulent?

One approach is to ask a set of “experts” to hand-code some criteria, such as:

- Is the purchase amount more than twice the usual purchase price for this customer?
- Is the purchase outside the customer’s home area?
- Has the customer bought other items of the same type over the past year?

This approach has many problems: there are far too many rules needed, and it is not clear how to set the constants in each rule (for instance, the ratio “twice” in the first rule above), or how to weight the relative importance of the different rules. A more promising strategy is to learn rules automatically from data.

#### 7.2.2 The input space and the label space

Each input $x$ is the description of a credit card transaction. We can code it as a vector with a large number of features (coordinates), for instance:

- **Customer data**
  - Information about customer: sex, age, city of residence, etc.
  - Purchase history: for each category of purchases, typical dollar amount per purchase, number of purchases per year, number of purchases outside home area, etc.
- **Details of current purchase**: type of item, dollar amount, location, does it fall into a standard “dubious” category (firearms, alcohol, ...), etc.
- **Relation of current purchase to prior purchase history**: price of item divided by average price of similar items purchased over past year, etc.

Say these form a $d$-dimensional vector. Then $\mathcal{X} = \mathbb{R}^d$. The label space is $\mathcal{Y} = \{+,-\}$ where “+” means the transaction is legitimate while “−” means it is fraudulent.

As always, we need a training set $(x_1, y_1), \ldots, (x_n, y_n)$, and to evaluate our classifier, we will also need a (typically smaller) test set.
7.2.3 Classification by decision tree

A decision tree is a binary tree where each internal node checks a specific coordinate of the input $x$, to see whether it lies in a specified range. Each leaf of the tree is a label, + or −. To classify an input $x$, you answer the question at the root, then move to either the left or right child, depending on the answer; and continue this way until you reach a leaf.

Here’s a toy example of a decision tree.

(We assume the vectorial representation is sufficiently rich that each question can be answered by looking at a single coordinate of $x$.)

How can such a tree be learned from data? The answer is, by building it top-down, adding nodes greedily to reduce uncertainty.

7.2.4 Learning a decision tree

Suppose the training set has 10000 points, with

- 6000 legitimate (+)
- 4000 fraudulent (−)

If we were allowed just one node, it would be a leaf with label “+”:

This has an error rate of 40% on the training data.

Now suppose we were allowed just one question. For instance, we might ask whether the location of the purchase is in the customer’s home area. The answer to this question, yes or no, splits the training set into two subsets. Let’s say the breakdown is as follows.
home area?  

\[
\begin{array}{c|cc}
\text{location} & \text{home area?} \\
\hline
\text{yes} & + & - \\
\text{no} & 1000 & 5000 \\
\text{no} & 1500 & 2500 \\
\end{array}
\]

Then on the left leaf we’d predict “−” while on the right we’d predict “+”, yielding:

\[
\begin{array}{c|cc}
\text{location} & \text{home area?} \\
\hline
\text{yes} & + & - \\
\text{no} & - & + \\
\end{array}
\]

A quarter of the training points end up in the left leaf, and amongst them the error rate is 2/5. Three-quarters of the points end up in the right leaf, and amongst them the error rate is 1/3. Thus the overall error rate is now

\[
\frac{1}{4} \cdot \frac{2}{5} + \frac{3}{4} \cdot \frac{1}{3} = \frac{7}{20},
\]

or 35%, less than before!

So asking this particular question reduces the error rate. We should pick the question that most reduces the error, and then recurse on the leaves. Here’s the procedure:

- Start with a single leaf node for all the training points.
- Repeat:
  - Pick a leaf that has significant error and contains quite a lot of training points. (If there is no such leaf, halt.)
  - Split it by asking a question that maximally reduces error within the leaf.

### 7.3 Linear classifiers

#### 7.3.1 Document classification

The internet brings with it a host of important document classification tasks. For instance,

- Is an email message spam or not? Input: email, label: + (legitimate) or − (spam).
- Sentiment detection. Is an article (such as a review) positive/favorable about its subject or negative/unfavorable? Input: article, label: + (favorable) or − (unfavorable).
• Is the text on a webpage pornographic (in which case Google wouldn’t want to return it) or not? Input: text on webpage, label: + (suitable for general audiences) or − (pornographic).

As always, one approach towards solving these problems is to hand-code rules. For instance, given the large volume of spam that seems to involve getting money out of Nigerian bank accounts, a possible rule for spam might be to flag emails containing the words “Nigeria” and “bank”. But vast numbers of such rules are needed, and they are constantly changing. It is more convenient and reliable to learn rules automatically from data.

7.3.2 The input space \( \mathcal{X} \) and label space \( \mathcal{Y} \)

A document is a sequence of words, and different documents have different lengths. How can they be represented as vectors of fixed dimension? The standard way to do so is the bag of words model.

Start by picking a fixed list of words, for instance, 50,000 of the most common words in English. Now represent each document as a vector \( x \) with 50,000 coordinates, each associated with a particular word. That coordinate records how many times the word occurs in the document. For example, the really short piece of text

\[
a \text{rose is a rose}
\]

would correspond to a vector in which the coordinate for “a” has value 2, the coordinate for “is” has value 1, the coordinate for “rose” has value 2, and the remaining 49,997 coordinates are zero.

Thus \( \mathcal{X} = \mathbb{R}^{50000} \) and \( \mathcal{Y} = \{+, −\} \). As always, we will need a training set \((x_1, y_1), \ldots, (x_n, y_n)\), to guide our choice of classifier, as well as a test set on which to evaluate our final classifier.

7.3.3 Linear classifiers

Suppose the data lie in \( \mathbb{R}^2 \) instead of \( \mathbb{R}^{50000} \). Then we can plot each training point \( x_i \) and annotate it with its label \( y_i \).

A linear classifier \( f : \mathbb{R}^2 \to \{+, −\} \) is simply a line with + on one side and − on the other side. Future points can be classified by which side of the line they lie on.
This also works in higher dimension—that is, when $X = \mathbb{R}^d$—but instead of a line we have a $(d - 1)$-dimensional hyperplane. For instance, when $d = 3$ the boundary between positive and negative is a plane.

7.3.4 **Learning a linear classifier**

Given a training set $(x_1, y_1), \ldots, (x_n, y_n)$, we’d like to find a linear function $f$ that correctly classifies all the points, that is, $f(x_i) = y_i$ for all $i$. There are two complications, however.

First, in the example above there are infinitely many solutions: infinitely many ways to draw a line between the positive and negative points. Which one should be used? A popular choice is to pick the line that is most squarely in the middle (according to a precise criterion), something like:

![Diagram](image)

A second problem is that sometimes there is no linear classifier that gets all the points correct:

![Diagram](image)

In such cases, we’d like to pick a linear function that makes the fewest mistakes possible (or some approximation thereof). Both these problems are handled by the widely-used *support vector machine*. We won’t get into the details here, but there are plenty of software packages that will take as input a data set and produce a linear classifier from it.

In fact, there are further extensions that allow the boundary between the two classes to be nonlinear (quadratic, or cubic, or even pretty arbitrary), and that allow non-vector data such as DNA sequences or trees!