# COS 424: Interacting with Data

Lecturer: Robert Schapire — Lecture #3
Scribe: Umar Javed — February 13, 2007

## 1 Introduction

Classifying objects from a data set based on a certain characteristic is a problem that comes up in many contexts. Our focus in this lecture is on binary classification, i.e. given an input, the classification scheme labels the input either positive or negative. In general, there might be more than two classes; we focus just on the two-class case for simplicity.

A classification learning algorithm takes as input a data set. Each member of the data set is classified as either positive or negative. The algorithm outputs another program, called the classifier or hypothesis, that has the ability to predict the label of input examples that are unclassified. The reasons that it is more desirable to produce a classifier program in this way than to write a program directly are that classifier programs produced in this way are more general purpose and therefore cheaper in terms of time and effort in the long term, and that an automated learning algorithm has a greater capacity than humans to express the differences in data objects in quantitative terms. For example, a human may know the difference between a lowercase and an uppercase letter, but is not able to express it in quantitative terms. On the other hand, a learning algorithm running on a computer might be able to learn the difference from examples. Figure 1 shows the general idea of a classification learning algorithm.

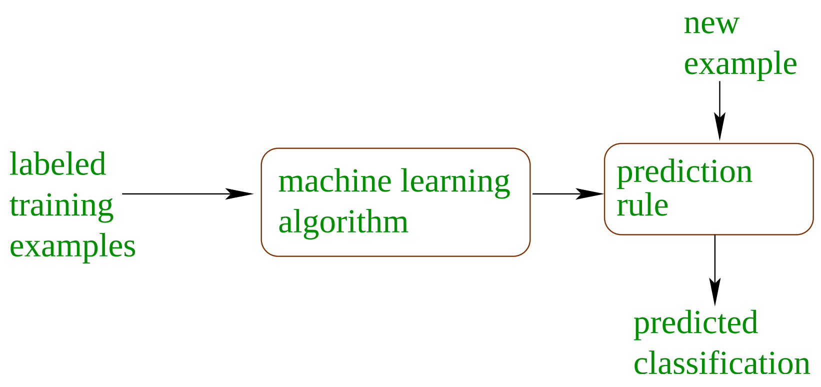

Figure 1: Flow diagram for a classification learning algorithm

**Example 1**

| **Integer** | **label** |
|---:|---|
| *train* | |
| 228 | − |
| 67 | + |
| 138 | + |
| 209 | − |
| 156 | + |
| 46 | − |
| 197 | − |
| 6 | − |
| 173 | + |
| *test* | |
| 111 | |
| 23 | |
| 55 | |

Table 1: A simple training data set example

The above table shows an input data set to a classification learning program and asks the classifier program to predict the labels for 111, 23 and 55. Based on the input values and the labels, the program might come up with the classification criterion that any integer greater than 196 or less than 47 will be labeled negative, and positive otherwise; or that any integer greater than 173 or less than 67 will be labeled negative, and positive otherwise. Below is another example.

**Example 2**

| **Integer** | **label** |
|---:|---|
| *train* | |
| 197 | + |
| 128 | − |
| 30 | − |
| 72 | − |
| 133 | − |
| 109 | + |
| 213 | + |
| 84 | + |
| 3 | − |
| *test* | |
| 200 | |
| 68 | |

Table 2: A simple training data set example

It is not obvious how to find a simple classification rule for this dataset. However, if we change the representation of the numbers by writing them in binary, the problem becomes simpler:

**Example 3**

| **Decimal** | **Binary** | | | | | | | | **label** |
|---|---|---|---|---|---|---|---|---|---|
| | *train* | | | | | | | | |
| 197 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | + |
| 128 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | − |
| 30 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | − |
| 72 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | − |
| 133 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | − |
| 109 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | + |
| 213 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | + |
| 84 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | + |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | − |
| | *test* | | | | | | | | |
| 200 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | |
| 68 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | |

Table 3: A simple training data set example

One possible classification scheme for this example is: a training example, which is a decimal integer, is positive if the second and sixth most significant bits in its binary representation are set; it's negative otherwise.

In general, for a classification learning algorithm to have a higher rate of success, the data set should be large enough, the classification rule should be as consistent to the training examples as possible (i.e. the training error should be as low as possible), and the classification/prediction rule should be as simple as possible (Occam's razor). The last two principles are often in confict with one another, so choosing the right balance between them is critical to the performance of the learning algorithm.

## 2 The Nearest Neighbor Algorithm

The Nearest neighbor algorithm in $n$ dimensions is an example of a learning algorithm.
**Training.** There are $m$ training examples. Each training example is of the form $(x_i, y_i)$, where $x_i \in \mathbf{R}^n$ and $y_i \in \{-,+\}$. Store all the training examples.
**Testing.** Given a test point $x$, predict $y_i$ where $x_i$ is the closest training example to $x$.

## 3 The $k$-Nearest Neighbor Algorithm

**Training.** This is identical to the Nearest Neighbor Algorithm.
**Testing.** Given a test point $x$, predict $y_i$ where $y_i$ is the majority vote of $\{y_1, y_2, \dots y_k\}$ and where $\{x_1, x_2, \dots x_k\}$ are the $k$ closest training points to $x$.

The performance of the nearest neighbor and the $k$-Nearest Neighbor algorithms depend upon the assumptions that we have a clear definition of distance between two points; that all the data is contained in $\mathbf{R}^n$; the training set of examples provide an adequate coverage of the $n$-dimensional space; and that nearby points are likely to have the same label.

As $m$, i.e. the number of training examples, approaches infinity, the nearest neighbor algorithm is guaranteed to yield an error rate no worse than twice the Bayes error rate (the minimum achievable error rate given the distribution of the data). $k$-nearest neighbor is guaranteed to approach the Bayes error rate, for some value of $k$ (where $k$ increases as a function of the number of data points). However, these convergence results are of little value in practice since $m$ is finite. This is why it is not rare to come across large error rates for the the NN and the $k$-NN algorithms in practice.

## 4 The curse of dimensionality

As the number of dimensions in the feature space grows, the error rate of some learning algorithms, including NN and $k$-NN, can be quite poor. This problem is called the "curse of dimensionality." It arises because the number of training examples are too few to cover the feature space adequately. To see this, consider the classification of points on an $n$-dimensional Boolean space. The training data consists of $m$ bit vectors, each of length $n$. Let $m = 1000$ and $n = 20$. If the first bit of the vector is 1, then the vector is labeled positive; negative otherwise. There are a total of $2^{20} \approx 10^6$ possible values on the 20-dimensional boolean cube, and only 1000 have been covered by the training data (assuming each of the $m$ bit vectors is different). This makes it extremely unlikely for a random $n$-dimensional test bit vector to fall in the vicinity of a training example, making the nearest neighbor and the $k$-nearest neighbor algorithms extremely error-prone.