Chapter 2

Instance-based Learning

To begin this course, we will focus on supervised learning, which represents the most popular and well-studied variant of machine learning. The goal of supervised learning is to infer the label y_i of an example i, given some features $\mathbf{x}_i \in \mathbb{R}^m$. Here, we will assume that the label is a binary value $y_i \in \{0, 1\}$ and the feature is a real-valued vector $\mathbf{x}_i \in \mathbb{R}^m$. This is the basic *binary classification* setting. As discussed in the previous chapter, we will assume that we have a training set $\mathcal{D}_{\text{trn}}$, which we use to develop our model, and a testing set $\mathcal{D}_{\text{tst}}$, which we will use to evaluate our model.

2.1 Predictions and Decision Boundaries

There are many different ways to motivate the basic principles of supervised learning, including probabilistic perspectives, motivations from the optimization literature, and ideas from information theory. We will survey these various perspectives in the course, but to begin, in this chapter, we will start with a somewhat utilitarian perspective based on the notion of *decision boundaries*.

Our goal in this perspective is to *partition* the input (i.e., feature) space into positive and negative sets, which we will denote $\mathcal{X}_1 \subset \mathbb{R}^m$ and $\mathcal{X}_0 \subset \mathbb{R}^m$, respectively. Formally, we will define our binary classification model based on the following prediction function:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \mathcal{X}_1 \\ 0 & \text{if } \mathbf{x} \in \mathcal{X}_0 \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{2.1}$$

In this approach all points in $\mathbf{x} \in \mathcal{X}_1$ are labeled as positive while points $\mathbf{x} \in \mathcal{X}_0$ are labeled as negative. Note, however, that we leave open the possibility for points to be labeled as *undefined*, which essentially amounts to the model making no prediction for that input.

2.2 Nearest Neighbors: A Simple Partitioning

There are innumerably many ways to define decision boundaries, and, in fact, *all* the supervised learning methods we will discuss in this course will specify some kind of partitioning of the input space (either explicitly or implicitly). But what is the *simplest* thing we could do? This is the key question that we consider in this chapter, as well as the next.

One very simple idea is to just let our training data define the partitioning over the input space. We know that all the positive training points are positive, and we know that all the negative training points are negative; so, let's just partition the space so that the region around each positive point is positive and the region around each negative point is negative. To formalize this idea, we introduce the nearest neighbor binary classification function:

$$f_{\text{NN}}(\mathbf{x}) = \texttt{MAJ}\left(\{y_i : (\mathbf{x}_i, y_i) \in \mathcal{D}_{\text{trn}} \wedge \forall (y_j, \mathbf{x}_j) \in \mathcal{D}_{\text{trn}} : d(\mathbf{x}, \mathbf{x}_i) \leq d(\mathbf{x}, \mathbf{x}_j)\}\right), \tag{2.2}$$

where we use `MAJ` to denote the function that returns the majority value of the multiset[1] of labels or returns "undefined" if the set has no clear majority. The term $d(\mathbf{x}, \mathbf{x}_i)$ in this equation measures the distance between the two points. In intuitive terms, $f_{\text{NN}}(\mathbf{x})$ finds the nearest neighbor(s) of the input $\mathbf{x}$ and returns the (majority) label associated with the neighbor(s).

2.3 Defining the Distance

A key element of Equation (2.2) is the distance function $d : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^+$, which we use to measure the distance between two points. In principle, we can define this distance function however we like, but, in practice, we tend to use standard distance functions that satisfy axioms of a formal distance metric:

1. $d(\mathbf{x}, \mathbf{y}) = 0 \Leftrightarrow \mathbf{x} = \mathbf{y}$ (**Identity of Indiscernibles**)
2. $d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x})$ (**Symmetry**)
3. $d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z}) + d(\mathbf{z}, \mathbf{y})$ (**Triangle Inequality**)

Assuming that our input space is the standard Euclidean space $\mathbb{R}^m$, the most popular distance function that satisfies these axioms is the Euclidean distance (which is also known as the L2 distance):

$$d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\| = \sqrt{(\mathbf{x}[0] - \mathbf{y}[0])^2 + ... + (\mathbf{x}[m-1] - \mathbf{y}[m-1])^2}. \tag{2.3}$$

Another popular distance metric is the absolute distance (which is also known as the L1 distance or the Manhattan distance):

$$d(\mathbf{x}, \mathbf{y}) = |\mathbf{x} - \mathbf{y}| = |\mathbf{x}[0] - \mathbf{y}[0]| + ... + |\mathbf{x}[m-1] - \mathbf{y}[m-1]|. \tag{2.4}$$

In general, we must choose a distance metric that is appropriate for our task.

[1] Technically, it is a multiset since it can contain repeated values.

Normalizing features It is also important to ensure that our features have a similar scale; otherwise, one feature can dominate the computation of the distance metric. For this reason, it is important to *normalize* the input data before running the nearest neighbors approach. One popular approach to normalization is to compute the mean

$$\mu_j = \frac{\sum_{(\mathbf{x},y)\in\mathcal{D}_{\text{trn}}} \mathbf{x}[j]}{|\mathcal{D}_{\text{trn}}|} \tag{2.5}$$

and standard deviation

$$\sigma_j = \sqrt{\frac{\sum_{(\mathbf{x},y)\in\mathcal{D}_{\text{trn}}} (\mathbf{x}[j]-\mu_j)^2}{|\mathcal{D}_{\text{trn}}|-1}} \tag{2.6}$$

of each feature and then to normalize the features by subtracting the mean and dividing by the standard deviation

$$\tilde{\mathbf{x}}[j] = \frac{\mathbf{x}[j]-\mu_j}{\sigma_j}, \tag{2.7}$$

where we use $\tilde{\mathbf{x}}[j]$ to denote the normalized feature value.

2.3.1 The Geometry of Nearest Neighbors

In terms of a partition of the input space, the nearest neighbor prediction function would correspond to the following partitions

$$\mathcal{X}_0 = \{\mathbf{x}\in\mathbb{R}^m : f_{\text{NN}}(\mathbf{x}) = 0\}$$
$$\mathcal{X}_1 = \{\mathbf{x}\in\mathbb{R}^m : f_{\text{NN}}(\mathbf{x}) = 1\}.$$

Note that $\mathcal{X}_0 \cup \mathcal{X}_1 \neq \mathbb{R}^m$, which means that there are points that belong to neither set and thus lie on the decision boundary itself. In this case, these points are equidistant from both the same number of positive and negative training points, meaning that we cannot confidently assign them a label.

Visually, the partitioning induced by the nearest-neighbor approach can be understood through the notion of a Voronoi diagram (Figure 2.1). The idea is that we partition the space into regions around each training example, such that all points in each partition are closer to that training example than any other point. Equivalently, the lines in a Voronoi diagram are all equidistant from at least two points. The decision boundary of this model corresponds to lines that separate positive (i.e., blue) and negative (i.e., red) regions, and the predictions for points that lie directly on these boundary lines are undefined.

As we can see in Figure 2.1, one important aspect of the nearest neighbor approach is that the decision boundary is not smooth or continuous. For exam-

[2]Image credit: http://scott.fortmann-roe.com/docs/BiasVariance.html

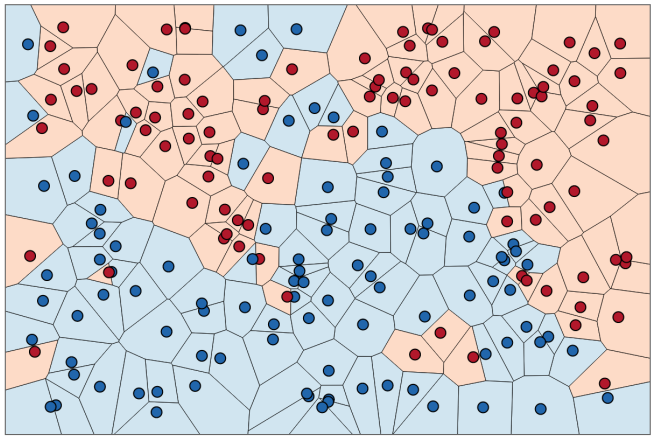

Figure 2.1: The nearest neighbor approach partitions the space according to a Voronoi diagram, where the lines around each point delineate the region of space that is closer to that point than any other point in the dataset.[2]

ple, there are regions of negative (i.e., red) surrounded by regions of blue, and the shape of the decision boundary involves many sharp and complex edges.

2.4 From Nearest Neighbors to k-NNs

As discussed above, a major issue with the nearest neighbor approach is that the decision boundary can be very complicated and non-smooth. This is problematic because it can lead to unstable predictions (i.e., different predictions for points that are close together) and it can make the model sensitive to outliers (i.e., a single training example can have a large impact on the model predictions).

One way to alleviate these issues is to generalize the nearest-neighbor approach is to the k-nearest-neighbour (k-NN) algorithm. Instead of finding just the nearest training example for an input point, we instead find the k-nearest training examples and make a classification based on the majority label of this set. Formally, the prediction function for the k-NN algorithm is given by

$$f_{\text{k-NN}}(\mathbf{x}) = \texttt{MAJ}\left(\{y_i : (\mathbf{x}_i, y_i) \in \mathcal{D}_{\text{trn}} \wedge \exists_{<k}(y_j, \mathbf{x}_j) \in \mathcal{D}_{\text{trn}} : d(\mathbf{x}, \mathbf{x}_i) > d(\mathbf{x}, \mathbf{x}_j)\}\right), \tag{2.8}$$

where we use $\exists_{<k}$ to denote that there exists less than k elements of the set satisfying the condition. (Note that $\exists_{<1}$ is thus equivalent to $\neg\exists$, i.e., there being no point satisfying the condition.) As with a 1-NN classifier, here we can have undefined predictions in the case where there is an equal number of positive and negative nodes in the k-NN set. Note also that a k-NN classifier might actually use labels from more than k neighbors in cases where there are training examples that are equidistant from the input point.

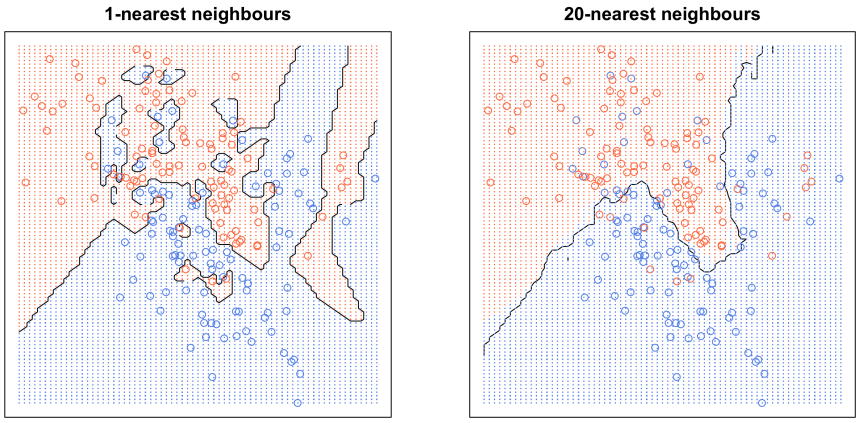

Figure 2.2: Comparison of the decision boundaries for a 1-NN and 20-NN model.

Figure 2.2 illustrates the decision boundaries learned by a 1-NN model and a 20-NN model. Notice that the decision boundary for the 20-NN is far more smooth and continuous, compared to the 1-NN model.

2.5 Training and Inference with k-NNs

In the previous sections, we discussed the prediction functions and decision boundaries for k-NN models. However, we did not formally discuss how these models are trained and evaluated using the training and test sets. Generally, in a machine learning model, we distinguish between the *training phase*, where the model is trained using the data in $\mathcal{D}_{\text{trn}}$, and the *testing phase*, where the model is evaluated using data in $\mathcal{D}_{\text{tst}}$. Put in another way, we use the training examples $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{D}_{\text{trn}}$ to determine (or learn) the prediction model, and then we used this trained model to *infer* the labels test examples $(\mathbf{x}_*, y_*) \in \mathcal{D}_{\text{tst}}$ given only their features $\mathbf{x}_*$.

k-NNs are one of the simplest machine learning models in that there is no "learning" to do during the training phase. We simply store all the training points to define the set $\mathcal{D}_{\text{trn}}$ in the k-NN prediction function (i.e., Equation 2.8). Then, when we want to do *inference* on a point $\mathbf{x}_* \in \mathbb{R}^m$, we simply apply the k-NN prediction function with the training set fixed. Note that this implies the following:

- The *training time complexity* of a k-NN is $\mathcal{O}(1)$.
- the *space complexity* of a k-NN is $\mathcal{O}(mn)$.
- the *inference complexity* of a k-NN is $\mathcal{O}(mn)$, assuming that it takes $\mathcal{O}(m)$ time to compute the distance between two points with m-dimensional features.

Thus, k-NNs are very efficient at training, since we just need to store the data, but they can be expensive at inference time.

Terminology check: learning, training, inference, and predictions In this course, we use the term *learning* to refer to the process of specifying, defining, or optimizing a model given some input data. In contrast, we use the term *inference* to refer to the process of making a prediction on an input point given a fully specified (i.e., trained) model. However, it is worth noting that this terminology can be mixed and confusing in the literature. We stick by a particular convention in this course—which has theoretical motivations—but you will likely see authors refer to the learning process as *inference* in other texts. In general, there are many competing terminologies in machine learning, largely driven by connections to different subfields of mathematics (e.g., statistics and optimization).

2.6 Choosing k with a Validation Set

So far in this chapter we discussed the idea of a k-NN classifier. However, how do we choose the right k for this model in practice? We saw in Section 2.4 that increasing k tends to produce a more smooth decision boundary, but how do we know if this is what we want? In general, there is no *a priori* way to specify the right k for a k-NN classifier. Indeed, the value of k is known as a *hyperparameter* for this model; it is a parameter that impacts the model's performance, but it is not something that we learn from the training set.

Many machine learning models have hyperparameters, and we will see this term often throughout the course. The standard practice for specifying hyperparameter is to use a *validation set*. The idea is that we split our training set $\mathcal{D}_{\text{trn}}$ into two partitions: the first partition is still used as our training data, and we will still call it $\mathcal{D}_{\text{trn}}$ for simplicity; the second component is what we call the validation set, or $\mathcal{D}_{\text{val}}$, and we use it to set the model hyperparameters. The basic idea is the following:

Step 1 We train N variants of our model using different hyperparameter settings on $\mathcal{D}_{\text{trn}}$. For example, in the k-NN setting, we might train models with $k = 1, 2, 5, 10$.

Step 2 Next, we evaluate the *validation performance* of each of the N model variants on $\mathcal{D}_{\text{val}}$.

Step 3 Finally, we choose the best performance variant from Step 2 and we retrain this model using the full original training set before running on the test set $\mathcal{D}_{\text{tst}}$.

The key idea is that we use a subset of the training data to evaluate performance so that we can choose between different model variants *before running*

on the test set. Indeed, it is crucial that we do not use the test set for hyperparameter selection, since this would amount to a form of cheating or *training on the test set*, which is serious error in machine learning model development.

Validation sets and cross-validation

Sometimes researchers use a more advanced form of validation known as *cross-validation*. In this approach, rather than using a single validation set, we evaluate different model variants using K different validation sets. The basic idea is the following:

Step 1 Split $\mathcal{D}_{\text{trn}}$ into K different partitions $\mathcal{D}^1_{\text{trn}}, ..., \mathcal{D}^K_{\text{trn}}$.

Step 2 For each split $i = 1, ..., K$ train all the N model variants using $\cup_{j \in [k], j \neq i} \mathcal{D}^j_{\text{trn}}$ and evaluate using $\mathcal{D}^i_{\text{trn}}$.

Step 3 Compute the average validation performance for each model across the K different splits in Step 2.

Step 4 Finally, we choose the best performance variant from Step 3 in terms of average or median performance and retrain this model using the full original training set before running on the test set $\mathcal{D}_{\text{tst}}$.

The intuition behind this approach is that we validate our model K different times using a different subset of the training data. This approach is more expensive computationally than simple validation with a single validation set, but it also provides a more accurate estimate of model performance.