CHAPTER 12

k-Nearest Neighbors

If you want to annoy your neighbors, tell the truth about them.
—Pietro Aretino

Imagine that you're trying to predict how I'm going to vote in the next presidential election. If you know nothing else about me (and if you have the data), one sensible approach is to look at how my *neighbors* are planning to vote. Living in Seattle, as I do, my neighbors are invariably planning to vote for the Democratic candidate, which suggests that "Democratic candidate" is a good guess for me as well.

Now imagine you know more about me than just geography—perhaps you know my age, my income, how many kids I have, and so on. To the extent my behavior is influenced (or characterized) by those things, looking just at my neighbors who are close to me among all those dimensions seems likely to be an even better predictor than looking at all my neighbors. This is the idea behind *nearest neighbors classification*.

The Model

Nearest neighbors is one of the simplest predictive models there is. It makes no mathematical assumptions, and it doesn't require any sort of heavy machinery. The only things it requires are:

- Some notion of distance
- An assumption that points that are close to one another are similar

Most of the techniques we'll see in this book look at the dataset as a whole in order to learn patterns in the data. Nearest neighbors, on the other hand, quite consciously neglects a lot of information, since the prediction for each new point depends only on the handful of points closest to it.

What's more, nearest neighbors is probably not going to help you understand the drivers of whatever phenomenon you're looking at. Predicting my votes based on my neighbors' votes doesn't tell you much about what causes me to vote the way I do, whereas some alternative model that predicted my vote based on (say) my income and marital status very well might.

In the general situation, we have some data points and we have a corresponding set of labels. The labels could be `True` and `False`, indicating whether each input satisfies some condition like "is spam?" or "is poisonous?" or "would be enjoyable to watch?" Or they could be categories, like movie ratings (G, PG, PG-13, R, NC-17). Or they could be the names of presidential candidates. Or they could be favorite programming languages.

In our case, the data points will be vectors, which means that we can use the `distance` function from Chapter 4.

Let's say we've picked a number *k* like 3 or 5. Then, when we want to classify some new data point, we find the *k* nearest labeled points and let them vote on the new output.

To do this, we'll need a function that counts votes. One possibility is:

```
from typing import List
from collections import Counter

def raw_majority_vote(labels: List[str]) -> str:
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner

assert raw_majority_vote(['a', 'b', 'c', 'b']) == 'b'
```

But this doesn't do anything intelligent with ties. For example, imagine we're rating movies and the five nearest movies are rated G, G, PG, PG, and R. Then G has two votes and PG also has two votes. In that case, we have several options:

- Pick one of the winners at random.
- Weight the votes by distance and pick the weighted winner.
- Reduce *k* until we find a unique winner.

We'll implement the third:

```
def majority_vote(labels: List[str]) -> str:
    """Assumes that labels are ordered from nearest to farthest."""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])
```

```
        if num_winners == 1:
            return winner                     # unique winner, so return it
        else:
            return majority_vote(labels[:-1]) # try again without the farthest

    # Tie, so look at first 4, then 'b'
    assert majority_vote(['a', 'b', 'c', 'b', 'a']) == 'b'
```

This approach is sure to work eventually, since in the worst case we go all the way down to just one label, at which point that one label wins.

With this function it's easy to create a classifier:

```
from typing import NamedTuple
from scratch.linear_algebra import Vector, distance

class LabeledPoint(NamedTuple):
    point: Vector
    label: str

def knn_classify(k: int,
                 labeled_points: List[LabeledPoint],
                 new_point: Vector) -> str:

    # Order the labeled points from nearest to farthest.
    by_distance = sorted(labeled_points,
                         key=lambda lp: distance(lp.point, new_point))

    # Find the labels for the k closest
    k_nearest_labels = [lp.label for lp in by_distance[:k]]

    # and let them vote.
    return majority_vote(k_nearest_labels)
```

Let's take a look at how this works.

Example: The Iris Dataset

The *Iris* dataset is a staple of machine learning. It contains a bunch of measurements for 150 flowers representing three species of iris. For each flower we have its petal length, petal width, sepal length, and sepal width, as well as its species. You can download it from *https://archive.ics.uci.edu/ml/datasets/iris*:

```
import requests

data = requests.get(
  "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
)

with open('iris.dat', 'w') as f:
    f.write(data.text)
```

The data is comma-separated, with fields:

```
sepal_length, sepal_width, petal_length, petal_width, class
```

For example, the first row looks like:

```
5.1,3.5,1.4,0.2,Iris-setosa
```

In this section we'll try to build a model that can predict the class (that is, the species) from the first four measurements.

To start with, let's load and explore the data. Our nearest neighbors function expects a `LabeledPoint`, so let's represent our data that way:

```python
from typing import Dict
import csv
from collections import defaultdict

def parse_iris_row(row: List[str]) -> LabeledPoint:
    """
    sepal_length, sepal_width, petal_length, petal_width, class
    """
    measurements = [float(value) for value in row[:-1]]
    # class is e.g. "Iris-virginica"; we just want "virginica"
    label = row[-1].split("-")[-1]

    return LabeledPoint(measurements, label)

with open('iris.data') as f:
    reader = csv.reader(f)
    iris_data = [parse_iris_row(row) for row in reader]

# We'll also group just the points by species/label so we can plot them
points_by_species: Dict[str, List[Vector]] = defaultdict(list)
for iris in iris_data:
    points_by_species[iris.label].append(iris.point)
```

We'd like to plot the measurements so we can see how they vary by species. Unfortunately, they are four-dimensional, which makes them tricky to plot. One thing we can do is look at the scatterplots for each of the six pairs of measurements (Figure 12-1). I won't explain all the details, but it's a nice illustration of more complicated things you can do with matplotlib, so it's worth studying:

```python
from matplotlib import pyplot as plt
metrics = ['sepal length', 'sepal width', 'petal length', 'petal width']
pairs = [(i, j) for i in range(4) for j in range(4) if i < j]
marks = ['+', '.', 'x']  # we have 3 classes, so 3 markers

fig, ax = plt.subplots(2, 3)

for row in range(2):
    for col in range(3):
        i, j = pairs[3 * row + col]
```

```
            ax[row][col].set_title(f"{metrics[i]} vs {metrics[j]}", fontsize=8)
            ax[row][col].set_xticks([])
            ax[row][col].set_yticks([])

            for mark, (species, points) in zip(marks, points_by_species.items()):
                xs = [point[i] for point in points]
                ys = [point[j] for point in points]
                ax[row][col].scatter(xs, ys, marker=mark, label=species)

    ax[-1][-1].legend(loc='lower right', prop={'size': 6})
    plt.show()
```

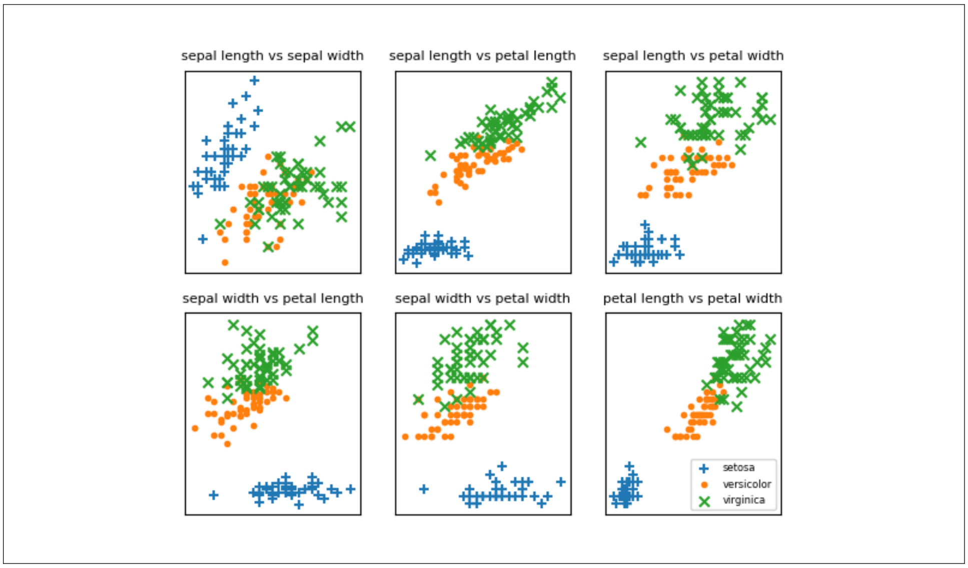

Figure 12-1. Iris scatterplots

If you look at those plots, it seems like the measurements really do cluster by species. For example, looking at sepal length and sepal width alone, you probably couldn't distinguish between *versicolor* and *virginica*. But once you add petal length and width into the mix, it seems like you should be able to predict the species based on the nearest neighbors.

To start with, let's split the data into a test set and a training set:

```
import random
from scratch.machine_learning import split_data

random.seed(12)
iris_train, iris_test = split_data(iris_data, 0.70)
assert len(iris_train) == 0.7 * 150
assert len(iris_test) == 0.3 * 150
```

The training set will be the "neighbors" that we'll use to classify the points in the test set. We just need to choose a value for *k*, the number of neighbors who get to vote. Too small (think *k* = 1), and we let outliers have too much influence; too large (think *k* = 105), and we just predict the most common class in the dataset.

In a real application (and with more data), we might create a separate validation set and use it to choose *k*. Here we'll just use *k* = 5:

```
from typing import Tuple

# track how many times we see (predicted, actual)
confusion_matrix: Dict[Tuple[str, str], int] = defaultdict(int)
num_correct = 0

for iris in iris_test:
    predicted = knn_classify(5, iris_train, iris.point)
    actual = iris.label

    if predicted == actual:
        num_correct += 1

    confusion_matrix[(predicted, actual)] += 1

pct_correct = num_correct / len(iris_test)
print(pct_correct, confusion_matrix)
```

On this simple dataset, the model predicts almost perfectly. There's one *versicolor* for which it predicts *virginica*, but otherwise it gets things exactly right.

The Curse of Dimensionality

The *k*-nearest neighbors algorithm runs into trouble in higher dimensions thanks to the "curse of dimensionality," which boils down to the fact that high-dimensional spaces are *vast*. Points in high-dimensional spaces tend not to be close to one another at all. One way to see this is by randomly generating pairs of points in the *d*-dimensional "unit cube" in a variety of dimensions, and calculating the distances between them.

Generating random points should be second nature by now:

```
def random_point(dim: int) -> Vector:
    return [random.random() for _ in range(dim)]
```

as is writing a function to generate the distances:

```
def random_distances(dim: int, num_pairs: int) -> List[float]:
    return [distance(random_point(dim), random_point(dim))
            for _ in range(num_pairs)]
```

For every dimension from 1 to 100, we'll compute 10,000 distances and use those to compute the average distance between points and the minimum distance between points in each dimension (Figure 12-2):

```
import tqdm
dimensions = range(1, 101)

avg_distances = []
min_distances = []

random.seed(0)
for dim in tqdm.tqdm(dimensions, desc="Curse of Dimensionality"):
    distances = random_distances(dim, 10000)      # 10,000 random pairs
    avg_distances.append(sum(distances) / 10000)  # track the average
    min_distances.append(min(distances))          # track the minimum
```

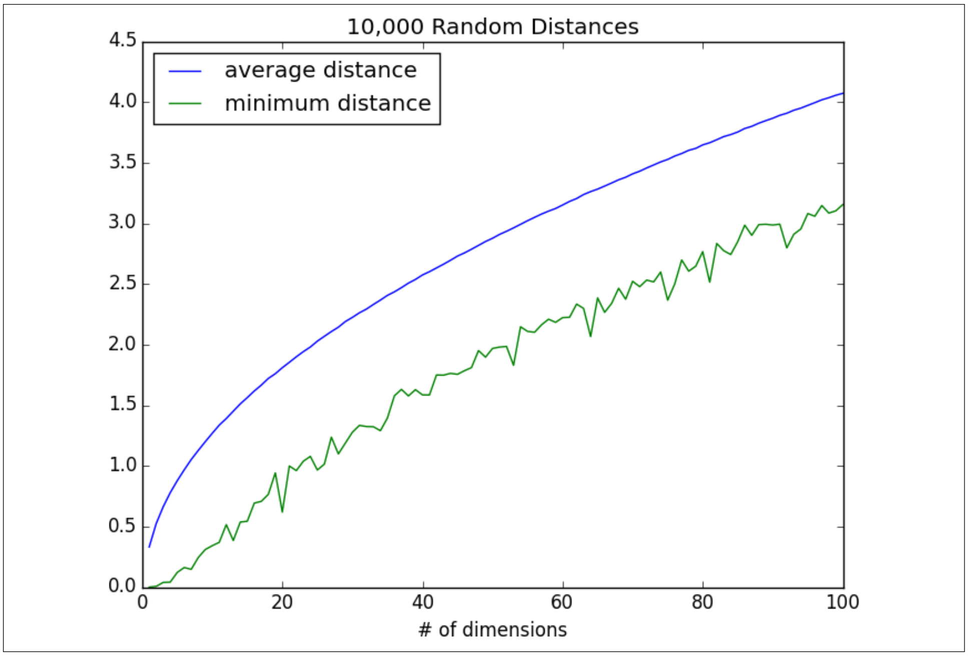

Figure 12-2. The curse of dimensionality

As the number of dimensions increases, the average distance between points increases. But what's more problematic is the ratio between the closest distance and the average distance (Figure 12-3):

```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances, avg_distances)]
```

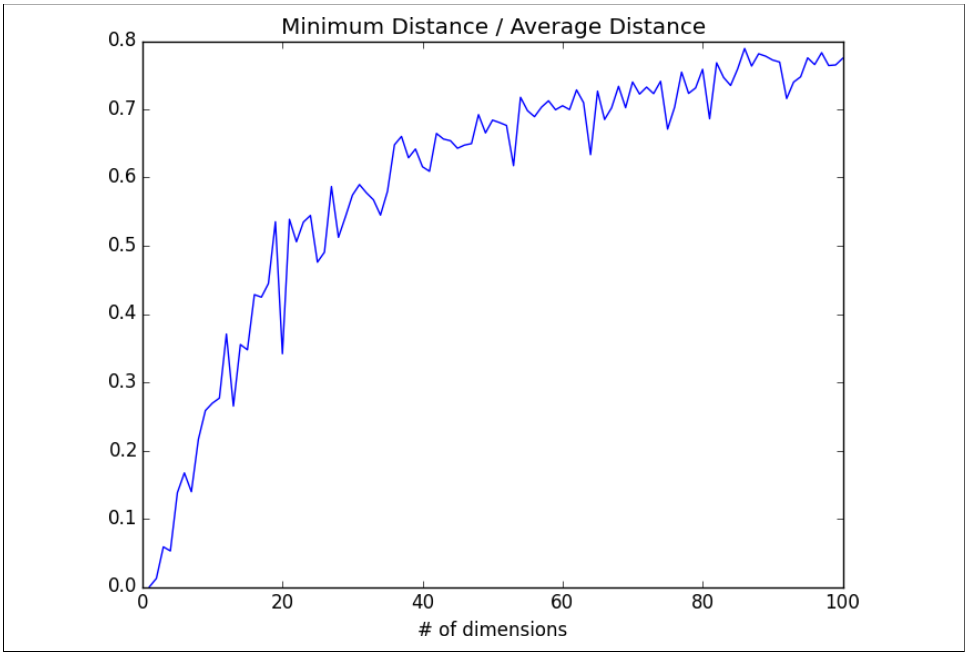

Figure 12-3. The curse of dimensionality again

In low-dimensional datasets, the closest points tend to be much closer than average. But two points are close only if they're close in every dimension, and every extra dimension—even if just noise—is another opportunity for each point to be farther away from every other point. When you have a lot of dimensions, it's likely that the closest points aren't much closer than average, so two points being close doesn't mean very much (unless there's a lot of structure in your data that makes it behave as if it were much lower-dimensional).

A different way of thinking about the problem involves the sparsity of higher-dimensional spaces.

If you pick 50 random numbers between 0 and 1, you'll probably get a pretty good sample of the unit interval (Figure 12-4).

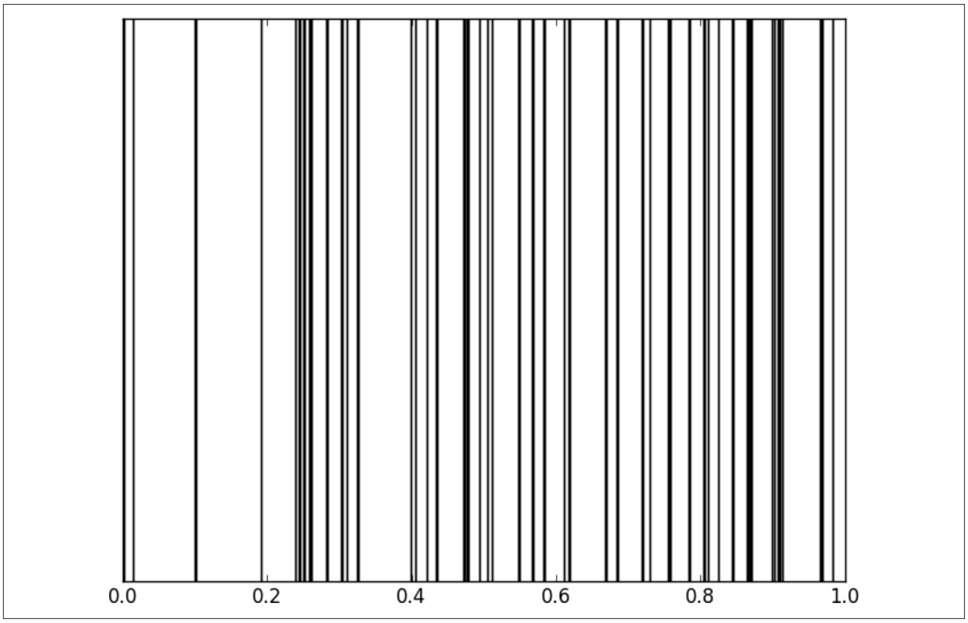

Figure 12-4. Fifty random points in one dimension

If you pick 50 random points in the unit square, you'll get less coverage (Figure 12-5).

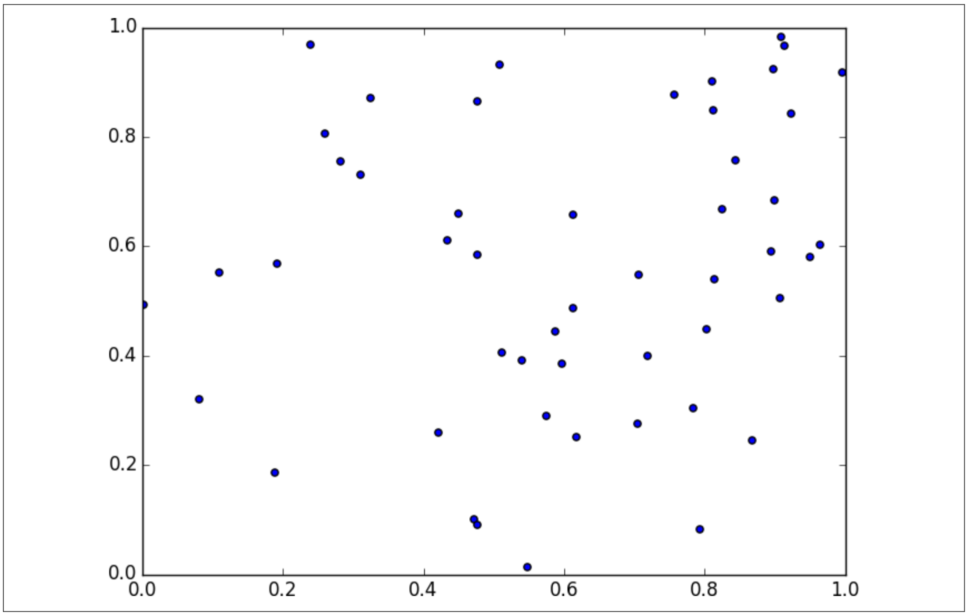

Figure 12-5. Fifty random points in two dimensions

And in three dimensions, less still (Figure 12-6).

matplotlib doesn't graph four dimensions well, so that's as far as we'll go, but you can see already that there are starting to be large empty spaces with no points near them. In more dimensions—unless you get exponentially more data—those large empty spaces represent regions far from all the points you want to use in your predictions.

So if you're trying to use nearest neighbors in higher dimensions, it's probably a good idea to do some kind of dimensionality reduction first.

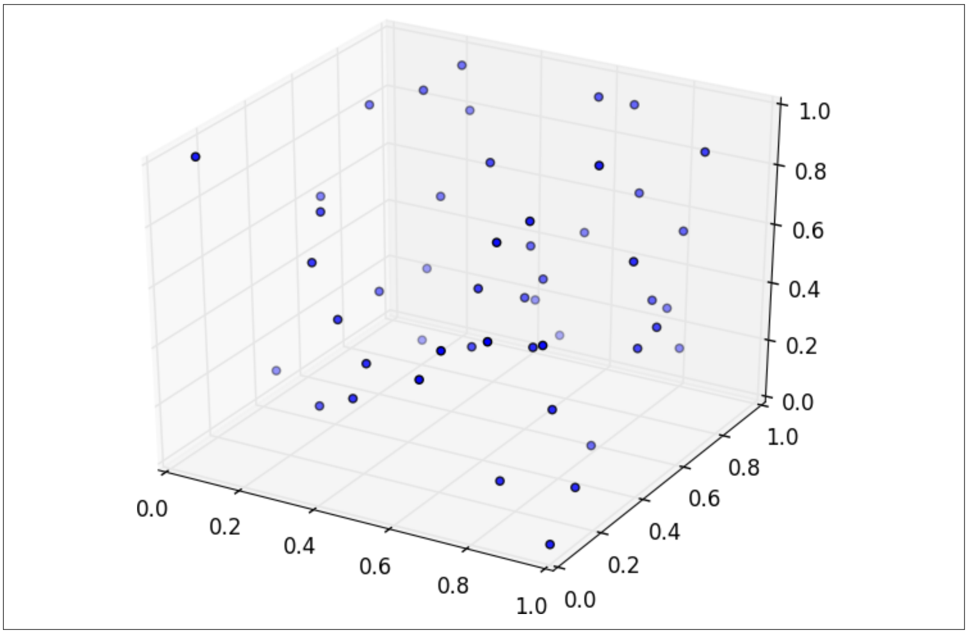

Figure 12-6. Fifty random points in three dimensions

For Further Exploration

scikit-learn has many nearest neighbor models.