**Classification**

[*David Wagner*](https://en.wikipedia.org/wiki/David_A._Wagner)*is the primary author of this chapter.*

*Machine learning* is a class of techniques for automatically finding patterns in data and using it to draw inferences or make predictions. You have already seen linear regression, which is one kind of machine learning. This chapter introduces a new one: *classification*.

Classification is about learning how to make predictions from past examples. We are given some examples where we have been told what the correct prediction was, and we want to learn from those examples how to make good predictions in the future. Here are a few applications where classification is used in practice:

* For each order Amazon receives, Amazon would like to predict: ***is this order fraudulent?*** They have some information about each order (e.g., its total value, whether the order is being shipped to an address this customer has used before, whether the shipping address is the same as the credit card holder’s billing address). They have lots of data on past orders, and they know which of those past orders were fraudulent and which weren’t. They want to learn patterns that will help them predict, as new orders arrive, whether those new orders are fraudulent.
* Online dating sites would like to predict: ***are these two people compatible?*** Will they hit it off? They have lots of data on which matches they’ve suggested to their customers in the past, and they have some idea which ones were successful. As new customers sign up, they’d like to make predictions about who might be a good match for them.
* Doctors would like to know: ***does this patient have cancer?*** Based on the measurements from some lab test, they’d like to be able to predict whether the particular patient has cancer. They have lots of data on past patients, including their lab measurements and whether they ultimately developed cancer, and from that, they’d like to try to infer what measurements tend to be characteristic of cancer (or non-cancer) so they can diagnose future patients accurately.
* Politicians would like to predict: ***are you going to vote for them?*** This will help them focus fundraising efforts on people who are likely to support them, and focus get-out-the-vote efforts on voters who will vote for them. Public databases and commercial databases have a lot of information about most people: e.g., whether they own a home or rent; whether they live in a rich neighborhood or poor neighborhood; their interests and hobbies; their shopping habits; and so on. And political campaigns have surveyed some voters and found out who they plan to vote for, so they have some examples where the correct answer is known. From this data, the campaigns would like to find patterns that will help them make predictions about all other potential voters.

All of these are classification tasks. Notice that in each of these examples, the prediction is a yes/no question – we call this *binary classification*, because there are only two possible predictions.

In a classification task, each individual or situation where we’d like to make a prediction is called an *observation*. We ordinarily have many observations. Each observation has multiple *attributes*, which are known (for example, the total value of the order on Amazon, or the voter’s annual salary). Also, each observation has a *class*, which is the answer to the question we care about (for example, fraudulent or not, or voting for you or not).

When Amazon is predicting whether orders are fraudulent, each order corresponds to a single observation. Each observation has several attributes: the total value of the order, whether the order is being shipped to an address this customer has used before, and so on. The class of the observation is either 0 or 1, where 0 means that the order is not fraudulent and 1 means that the order is fraudulent. When a customer makes a new order, we do not observe whether it is fraudulent, but we do observe its attributes, and we will try to predict its class using those attributes.

Classification requires data. It involves looking for patterns, and to find patterns, you need data. That’s where the data science comes in. In particular, we’re going to assume that we have access to *training data*: a bunch of observations, where we know the class of each observation. The collection of these pre-classified observations is also called a training set. A classification algorithm is going to analyze the training set, and then come up with a classifier: an algorithm for predicting the class of future observations.

Classifiers do not need to be perfect to be useful. They can be useful even if their accuracy is less than 100%. For instance, if the online dating site occasionally makes a bad recommendation, that’s OK; their customers already expect to have to meet many people before they’ll find someone they hit it off with. Of course, you don’t want the classifier to make too many errors — but it doesn’t have to get the right answer every single time.

**Nearest Neighbors**

In this section we’ll develop the *nearest neighbor* method of classification. Just focus on the ideas for now and don’t worry if some of the code is mysterious. Later in the chapter we’ll see how to organize our ideas into code that performs the classification.

**Chronic kidney disease**

Let’s work through an example. We’re going to work with a data set that was collected to help doctors diagnose chronic kidney disease (CKD). Each row in the data set represents a single patient who was treated in the past and whose diagnosis is known. For each patient, we have a bunch of measurements from a blood test. We’d like to find which measurements are most useful for diagnosing CKD, and develop a way to classify future patients as “has CKD” or “doesn’t have CKD” based on their blood test results.

ckd **=** Table**.**read\_table(path\_data **+** 'ckd.csv')**.**relabeled('Blood Glucose Random', 'Glucose')

ckd

| **Age** | **Blood Pressure** | **Specific Gravity** | **Albumin** | **Sugar** | **Red Blood Cells** | **Pus Cell** | **Pus Cell clumps** | **Bacteria** | **Glucose** | **Blood Urea** | **Serum Creatinine** | **Sodium** | **Potassium** | **Hemoglobin** | **Packed Cell Volume** | **White Blood Cell Count** | **Red Blood Cell Count** | **Hypertension** | **Diabetes Mellitus** | **Coronary Artery Disease** | **Appetite** | **Pedal Edema** | **Anemia** | **Class** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 48 | 70 | 1.005 | 4 | 0 | normal | abnormal | present | notpresent | 117 | 56 | 3.8 | 111 | 2.5 | 11.2 | 32 | 6700 | 3.9 | yes | no | no | poor | yes | yes | 1 |
| 53 | 90 | 1.02 | 2 | 0 | abnormal | abnormal | present | notpresent | 70 | 107 | 7.2 | 114 | 3.7 | 9.5 | 29 | 12100 | 3.7 | yes | yes | no | poor | no | yes | 1 |
| 63 | 70 | 1.01 | 3 | 0 | abnormal | abnormal | present | notpresent | 380 | 60 | 2.7 | 131 | 4.2 | 10.8 | 32 | 4500 | 3.8 | yes | yes | no | poor | yes | no | 1 |
| 68 | 80 | 1.01 | 3 | 2 | normal | abnormal | present | present | 157 | 90 | 4.1 | 130 | 6.4 | 5.6 | 16 | 11000 | 2.6 | yes | yes | yes | poor | yes | no | 1 |
| 61 | 80 | 1.015 | 2 | 0 | abnormal | abnormal | notpresent | notpresent | 173 | 148 | 3.9 | 135 | 5.2 | 7.7 | 24 | 9200 | 3.2 | yes | yes | yes | poor | yes | yes | 1 |
| 48 | 80 | 1.025 | 4 | 0 | normal | abnormal | notpresent | notpresent | 95 | 163 | 7.7 | 136 | 3.8 | 9.8 | 32 | 6900 | 3.4 | yes | no | no | good | no | yes | 1 |
| 69 | 70 | 1.01 | 3 | 4 | normal | abnormal | notpresent | notpresent | 264 | 87 | 2.7 | 130 | 4 | 12.5 | 37 | 9600 | 4.1 | yes | yes | yes | good | yes | no | 1 |
| 73 | 70 | 1.005 | 0 | 0 | normal | normal | notpresent | notpresent | 70 | 32 | 0.9 | 125 | 4 | 10 | 29 | 18900 | 3.5 | yes | yes | no | good | yes | no | 1 |
| 73 | 80 | 1.02 | 2 | 0 | abnormal | abnormal | notpresent | notpresent | 253 | 142 | 4.6 | 138 | 5.8 | 10.5 | 33 | 7200 | 4.3 | yes | yes | yes | good | no | no | 1 |
| 46 | 60 | 1.01 | 1 | 0 | normal | normal | notpresent | notpresent | 163 | 92 | 3.3 | 141 | 4 | 9.8 | 28 | 14600 | 3.2 | yes | yes | no | good | no | no | 1 |

... (148 rows omitted)

Some of the variables are categorical (words like “abnormal”), and some quantitative. The quantitative variables all have different scales. We’re going to want to make comparisons and estimate distances, often by eye, so let’s select just a few of the variables and work in standard units. Then we won’t have to worry about the scale of each of the different variables.

ckd **=** Table()**.**with\_columns(

'Hemoglobin', standard\_units(ckd**.**column('Hemoglobin')),

'Glucose', standard\_units(ckd**.**column('Glucose')),

'White Blood Cell Count', standard\_units(ckd**.**column('White Blood Cell Count')),

'Class', ckd**.**column('Class')

)

ckd

| **Hemoglobin** | **Glucose** | **White Blood Cell Count** | **Class** |
| --- | --- | --- | --- |
| -0.865744 | -0.221549 | -0.569768 | 1 |
| -1.45745 | -0.947597 | 1.16268 | 1 |
| -1.00497 | 3.84123 | -1.27558 | 1 |
| -2.81488 | 0.396364 | 0.809777 | 1 |
| -2.08395 | 0.643529 | 0.232293 | 1 |
| -1.35303 | -0.561402 | -0.505603 | 1 |
| -0.413266 | 2.04928 | 0.360623 | 1 |
| -1.28342 | -0.947597 | 3.34429 | 1 |
| -1.10939 | 1.87936 | -0.409356 | 1 |
| -1.35303 | 0.489051 | 1.96475 | 1 |

... (148 rows omitted)

Let’s look at two columns in particular: the hemoglobin level (in the patient’s blood), and the blood glucose level (at a random time in the day; without fasting specially for the blood test).

We’ll draw a scatter plot to visualize the relation between the two variables. Blue dots are patients with CKD; gold dots are patients without CKD. What kind of medical test results seem to indicate CKD?

color\_table **=** Table()**.**with\_columns(

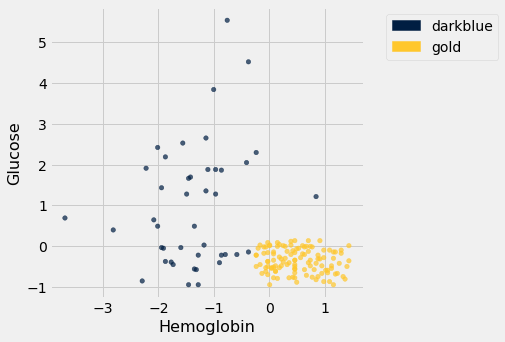
'Class', make\_array(1, 0),

'Color', make\_array('darkblue', 'gold')

)

ckd **=** ckd**.**join('Class', color\_table)

ckd**.**scatter('Hemoglobin', 'Glucose', colors**=**'Color')



Suppose Alice is a new patient who is not in the data set. If I tell you Alice’s hemoglobin level and blood glucose level, could you predict whether she has CKD? It sure looks like it! You can see a very clear pattern here: points in the lower-right tend to represent people who don’t have CKD, and the rest tend to be folks with CKD. To a human, the pattern is obvious. But how can we program a computer to automatically detect patterns such as this one?

**A Nearest Neighbor Classifier**

There are lots of kinds of patterns one might look for, and lots of algorithms for classification. But I’m going to tell you about one that turns out to be surprisingly effective. It is called *nearest neighbor classification*. Here’s the idea. If we have Alice’s hemoglobin and glucose numbers, we can put her somewhere on this scatterplot; the hemoglobin is her x-coordinate, and the glucose is her y-coordinate. Now, to predict whether she has CKD or not, we find the nearest point in the scatterplot and check whether it is blue or gold; we predict that Alice should receive the same diagnosis as that patient.

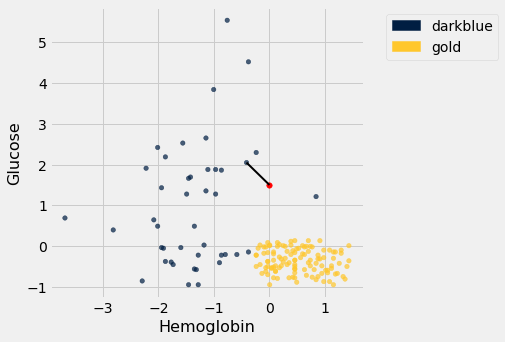
In other words, to classify Alice as CKD or not, we find the patient in the training set who is “nearest” to Alice, and then use that patient’s diagnosis as our prediction for Alice. The intuition is that if two points are near each other in the scatterplot, then the corresponding measurements are pretty similar, so we might expect them to receive the same diagnosis (more likely than not). We don’t know Alice’s diagnosis, but we do know the diagnosis of all the patients in the training set, so we find the patient in the training set who is most similar to Alice, and use that patient’s diagnosis to predict Alice’s diagnosis.

In the graph below, the red dot represents Alice. It is joined with a black line to the point that is nearest to it – its *nearest neighbor* in the training set. The figure is drawn by a function called show\_closest. It takes an array that represents the xx and yy coordinates of Alice’s point. Vary those to see how the closest point changes! Note especially when the closest point is blue and when it is gold.

*# In this example, Alice's Hemoglobin attribute is 0 and her Glucose is 1.5.*

alice **=** make\_array(0, 1.5)

show\_closest(alice)



Thus our *nearest neighbor classifier* works like this:

* Find the point in the training set that is nearest to the new point.
* If that nearest point is a “CKD” point, classify the new point as “CKD”. If the nearest point is a “not CKD” point, classify the new point as “not CKD”.

The scatterplot suggests that this nearest neighbor classifier should be pretty accurate. Points in the lower-right will tend to receive a “no CKD” diagnosis, as their nearest neighbor will be a gold point. The rest of the points will tend to receive a “CKD” diagnosis, as their nearest neighbor will be a blue point. So the nearest neighbor strategy seems to capture our intuition pretty well, for this example.

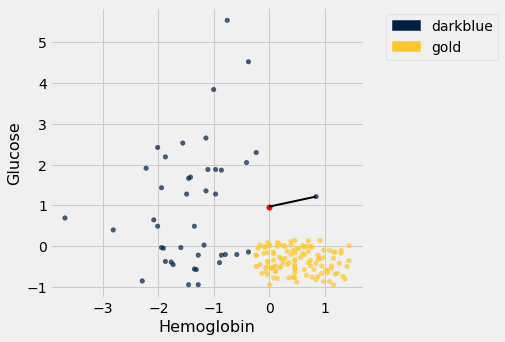
**Decision boundary**

Sometimes a helpful way to visualize a classifier is to map out the kinds of attributes where the classifier would predict ‘CKD’, and the kinds where it would predict ‘not CKD’. We end up with some boundary between the two, where points on one side of the boundary will be classified ‘CKD’ and points on the other side will be classified ‘not CKD’. This boundary is called the *decision boundary*. Each different classifier will have a different decision boundary; the decision boundary is just a way to visualize what criteria the classifier is using to classify points.

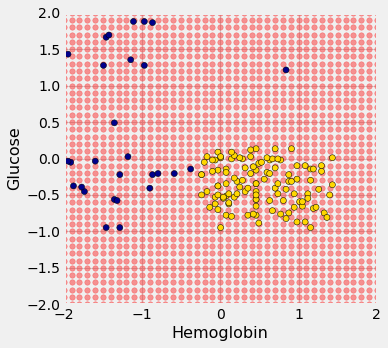
For example, suppose the coordinates of Alice’s point are (0, 1.5). Notice that the nearest neighbor is blue. Now try reducing the height (the yy-coordinate) of the point. You’ll see that at around y=0.95y=0.95 the nearest neighbor turns from blue to gold.

alice **=** make\_array(0, 0.97)

show\_closest(alice)



Here are hundreds of new unclassified points, all in red.

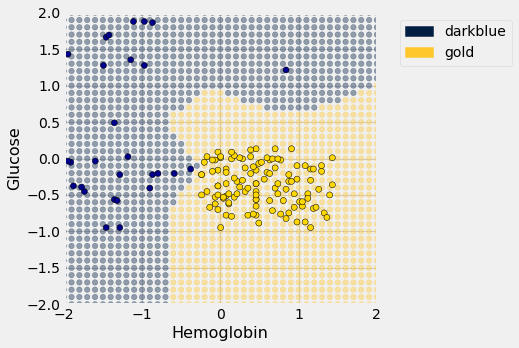


Each of the red points has a nearest neighbor in the training set (the same blue and gold points as before). For some red points you can easily tell whether the nearest neighbor is blue or gold. For others, it’s a little more tricky to make the decision by eye. Those are the points near the decision boundary.

But the computer can easily determine the nearest neighbor of each point. So let’s get it to apply our nearest neighbor classifier to each of the red points:

For each red point, it must find the closest point in the training set; it must then change the color of the red point to become the color of the nearest neighbor.

The resulting graph shows which points will get classified as ‘CKD’ (all the blue ones), and which as ‘not CKD’ (all the gold ones).

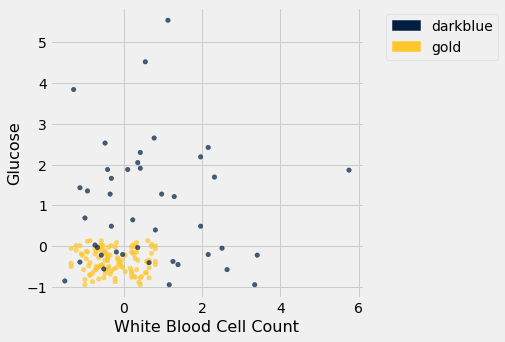


The decision boundary is where the classifier switches from turning the red points blue to turning them gold.

**k-Nearest Neighbors**

However, the separation between the two classes won’t always be quite so clean. For instance, suppose that instead of hemoglobin levels we were to look at white blood cell count. Look at what happens:

ckd**.**scatter('White Blood Cell Count', 'Glucose', colors**=**'Color')



As you can see, non-CKD individuals are all clustered in the lower-left. Most of the patients with CKD are above or to the right of that cluster… but not all. There are some patients with CKD who are in the lower left of the above figure (as indicated by the handful of blue dots scattered among the gold cluster). What this means is that you can’t tell for certain whether someone has CKD from just these two blood test measurements.

If we are given Alice’s glucose level and white blood cell count, can we predict whether she has CKD? Yes, we can make a prediction, but we shouldn’t expect it to be 100% accurate. Intuitively, it seems like there’s a natural strategy for predicting: plot where Alice lands in the scatter plot; if she is in the lower-left, predict that she doesn’t have CKD, otherwise predict she has CKD.

This isn’t perfect – our predictions will sometimes be wrong. (Take a minute and think it through: for which patients will it make a mistake?) As the scatterplot above indicates, sometimes people with CKD have glucose and white blood cell levels that look identical to those of someone without CKD, so any classifier is inevitably going to make the wrong prediction for them.

Can we automate this on a computer? Well, the nearest neighbor classifier would be a reasonable choice here too. Take a minute and think it through: how will its predictions compare to those from the intuitive strategy above? When will they differ?

Its predictions will be pretty similar to our intuitive strategy, but occasionally it will make a different prediction. In particular, if Alice’s blood test results happen to put her right near one of the blue dots in the lower-left, the intuitive strategy would predict ‘not CKD’, whereas the nearest neighbor classifier will predict ‘CKD’.

There is a simple generalization of the nearest neighbor classifier that fixes this anomaly. It is called the *k-nearest neighbor classifier*. To predict Alice’s diagnosis, rather than looking at just the one neighbor closest to her, we can look at the 3 points that are closest to her, and use the diagnosis for each of those 3 points to predict Alice’s diagnosis. In particular, we’ll use the majority value among those 3 diagnoses as our prediction for Alice’s diagnosis. Of course, there’s nothing special about the number 3: we could use 4, or 5, or more. (It’s often convenient to pick an odd number, so that we don’t have to deal with ties.) In general, we pick a number kk, and our predicted diagnosis for Alice is based on the kk patients in the training set who are closest to Alice. Intuitively, these are the kk patients whose blood test results were most similar to Alice, so it seems reasonable to use their diagnoses to predict Alice’s diagnosis.

The kk-nearest neighbor classifier will now behave just like our intuitive strategy above.

**Training and Testing**

How good is our nearest neighbor classifier? To answer this we’ll need to find out how frequently our classifications are correct. If a patient has chronic kidney disease, how likely is our classifier to pick that up?

If the patient is in our training set, we can find out immediately. We already know what class the patient is in. So we can just compare our prediction and the patient’s true class.

But the point of the classifier is to make predictions for *new* patients not in our training set. We don’t know what class these patients are in but we can make a prediction based on our classifier. How to find out whether the prediction is correct?

One way is to wait for further medical tests on the patient and then check whether or not our prediction agrees with the test results. With that approach, by the time we can say how likely our prediction is to be accurate, it is no longer useful for helping the patient.

Instead, we will try our classifier on some patients whose true classes are known. Then, we will compute the proportion of the time our classifier was correct. This proportion will serve as an estimate of the proportion of all new patients whose class our classifier will accurately predict. This is called *testing*.

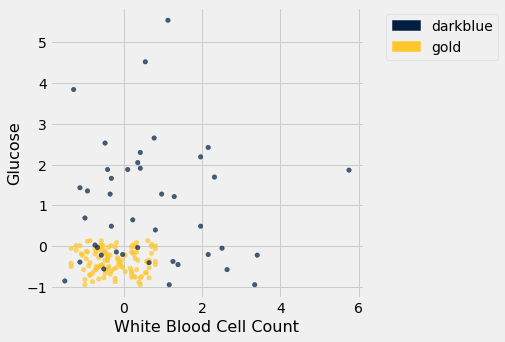
**Overly Optimistic “Testing”**

The training set offers a very tempting set of patients on whom to test out our classifier, because we know the class of each patient in the training set.

But let’s be careful … there will be pitfalls ahead if we take this path. An example will show us why.

Suppose we use a 1-nearest neighbor classifier to predict whether a patient has chronic kidney disease, based on glucose and white blood cell count.

ckd**.**scatter('White Blood Cell Count', 'Glucose', colors**=**'Color')



Earlier, we said that we expect to get some classifications wrong, because there’s some intermingling of blue and gold points in the lower-left.

But what about the points in the training set, that is, the points already on the scatter? Will we ever mis-classify them?

The answer is no. Remember that 1-nearest neighbor classification looks for the point *in the training set* that is nearest to the point being classified. Well, if the point being classified is already in the training set, then its nearest neighbor in the training set is itself! And therefore it will be classified as its own color, which will be correct because each point in the training set is already correctly colored.

In other words, **if we use our training set to “test” our 1-nearest neighbor classifier, the classifier will pass the test 100% of the time.**

Mission accomplished. What a great classifier!

No, not so much. A new point in the lower-left might easily be mis-classified, as we noted earlier. “100% accuracy” was a nice dream while it lasted.

The lesson of this example is *not* to use the training set to test a classifier that is based on it.

**Generating a Test Set**

In earlier chapters, we saw that random sampling could be used to estimate the proportion of individuals in a population that met some criterion. Unfortunately, we have just seen that the training set is not like a random sample from the population of all patients, in one important respect: Our classifier guesses correctly for a higher proportion of individuals in the training set than it does for individuals in the population.

When we computed confidence intervals for numerical parameters, we wanted to have many new random samples from a population, but we only had access to a single sample. We solved that problem by taking bootstrap resamples from our sample.

We will use an analogous idea to test our classifier. We will *create two samples out of the original training set*, use one of the samples as our training set, and *the other one for testing*.

So we will have three groups of individuals:

* a training set on which we can do any amount of exploration to build our classifier;
* a separate testing set on which to try out our classifier and see what fraction of times it classifies correctly;
* the underlying population of individuals for whom we don’t know the true classes; the hope is that our classifier will succeed about as well for these individuals as it did for our testing set.

How to generate the training and testing sets? You’ve guessed it – we’ll select at random.

There are 158 individuals in ckd. Let’s use a random half of them for training and the other half for testing. To do this, we’ll shuffle all the rows, take the first 79 as the training set, and the remaining 79 for testing.

shuffled\_ckd **=** ckd**.**sample(with\_replacement**=**False)

training **=** shuffled\_ckd**.**take(np**.**arange(79))

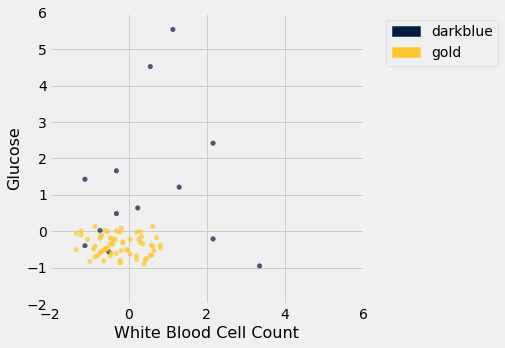
testing **=** shuffled\_ckd**.**take(np**.**arange(79, 158))

Now let’s construct our classifier based on the points in the training sample:

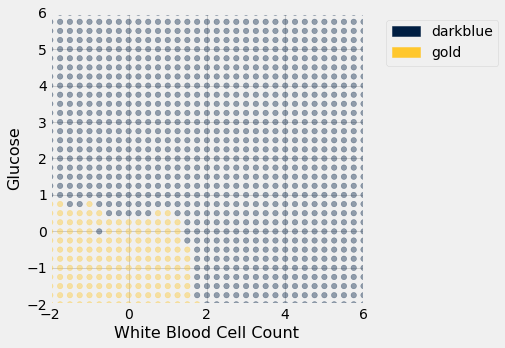
training**.**scatter('White Blood Cell Count', 'Glucose', colors**=**'Color')

plt**.**xlim(**-**2, 6)

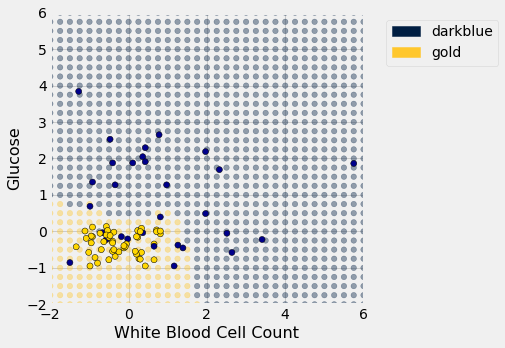
plt**.**ylim(**-**2, 6);



We get the following classification regions and decision boundary:



Place the *test* data on this graph and you can see at once that while the classifier got almost all the points right, there are some mistakes. For example, some blue points of the test set fall in the gold region of the classifier.



Some errors notwithstanding, it looks like the classifier does fairly well on the test set. Assuming that the original sample was drawn randomly from the underlying population, the hope is that the classifier will perform with similar accuracy on the overall population, since the test set was chosen randomly from the original sample.

**Rows of Tables**

Now that we have a qualitative understanding of nearest neighbor classification, it’s time to implement our classifier.

Until this chapter, we have worked mostly with single columns of tables. But now we have to see whether one *individual* is “close” to another. Data for individuals are contained in *rows* of tables.

So let’s start by taking a closer look at rows.

Here is the original table ckd containing data on patients who were tested for chronic kidney disease.

ckd **=** Table**.**read\_table(path\_data **+** 'ckd.csv')**.**relabeled('Blood Glucose Random', 'Glucose')

The data corresponding to the first patient is in row 0 of the table, consistent with Python’s indexing system. The Table method row accesses the row by taking the index of the row as its argument:

ckd**.**row(0)

Row(Age=48, Blood Pressure=70, Specific Gravity=1.005, Albumin=4, Sugar=0, Red Blood Cells='normal', Pus Cell='abnormal', Pus Cell clumps='present', Bacteria='notpresent', Glucose=117, Blood Urea=56, Serum Creatinine=3.8, Sodium=111, Potassium=2.5, Hemoglobin=11.2, Packed Cell Volume=32, White Blood Cell Count=6700, Red Blood Cell Count=3.9, Hypertension='yes', Diabetes Mellitus='no', Coronary Artery Disease='no', Appetite='poor', Pedal Edema='yes', Anemia='yes', Class=1)

Rows have their very own data type: they are *row objects*. Notice how the display shows not only the values in the row but also the labels of the corresponding columns.

Rows are in general **not arrays**, as their elements can be of different types. For example, some of the elements of the row above are strings (like 'abnormal') and some are numerical. So the row can’t be converted into an array.

However, rows share some characteristics with arrays. You can use item to access a particular element of a row. For example, to access the Albumin level of Patient 0, we can look at the labels in the printout of the row above to find that it’s item 3:

ckd**.**row(0)**.**item(3)

4

**Converting Rows to Arrays (When Possible)**

Rows whose elements are all numerical (or all strings) can be converted to arrays. Converting a row to an array gives us access to arithmetic operations and other nice NumPy functions, so it is often useful.

Recall that in the previous section we tried to classify the patients as ‘CKD’ or ‘not CKD’, based on two attributes Hemoglobin and Glucose, both measured in standard units.

ckd **=** Table()**.**with\_columns(

'Hemoglobin', standard\_units(ckd**.**column('Hemoglobin')),

'Glucose', standard\_units(ckd**.**column('Glucose')),

'Class', ckd**.**column('Class')

)

color\_table **=** Table()**.**with\_columns(

'Class', make\_array(1, 0),

'Color', make\_array('darkblue', 'gold')

)

ckd **=** ckd**.**join('Class', color\_table)

ckd

| **Class** | **Hemoglobin** | **Glucose** | **Color** |
| --- | --- | --- | --- |
| 0 | 0.456884 | 0.133751 | gold |
| 0 | 1.153 | -0.947597 | gold |
| 0 | 0.770138 | -0.762223 | gold |
| 0 | 0.596108 | -0.190654 | gold |
| 0 | -0.239236 | -0.49961 | gold |
| 0 | -0.0304002 | -0.159758 | gold |
| 0 | 0.282854 | -0.00527964 | gold |
| 0 | 0.108824 | -0.623193 | gold |
| 0 | 0.0740178 | -0.515058 | gold |
| 0 | 0.83975 | -0.422371 | gold |

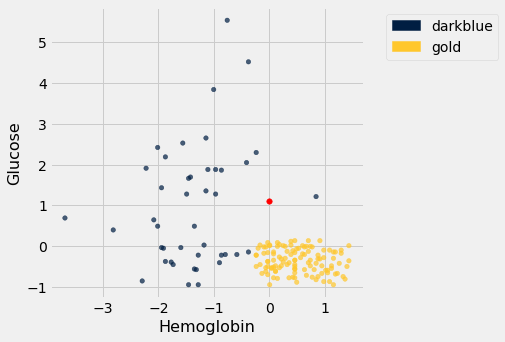
... (148 rows omitted)

Here is a scatter plot of the two attributes, along with a red point corresponding to Alice, a new patient. Her value of hemoglobin is 0 (that is, at the average) and glucose 1.1 (that is, 1.1 SDs above average).

alice **=** make\_array(0, 1.1)

ckd**.**scatter('Hemoglobin', 'Glucose', colors**=**'Color')

plots**.**scatter(alice**.**item(0), alice**.**item(1), color**=**'red', s**=**30);



To find the distance between Alice’s point and any of the other points, we only need the values of the attributes:

ckd\_attributes **=** ckd**.**select('Hemoglobin', 'Glucose')

ckd\_attributes

| **Hemoglobin** | **Glucose** |
| --- | --- |
| 0.456884 | 0.133751 |
| 1.153 | -0.947597 |
| 0.770138 | -0.762223 |
| 0.596108 | -0.190654 |
| -0.239236 | -0.49961 |
| -0.0304002 | -0.159758 |
| 0.282854 | -0.00527964 |
| 0.108824 | -0.623193 |
| 0.0740178 | -0.515058 |
| 0.83975 | -0.422371 |

... (148 rows omitted)

Each row consists of the coordinates of one point in our training sample. **Because the rows now consist only of numerical values**, it is possible to convert them to arrays. For this, we use the function np.array, which converts any kind of sequential object, like a row, to an array. (Our old friend make\_array is for *creating* arrays, not for *converting* other kinds of sequences to arrays.)

ckd\_attributes**.**row(3)

Row(Hemoglobin=0.5961076648232668, Glucose=-0.19065363034327712)

np**.**array(ckd\_attributes**.**row(3))

array([ 0.59610766, -0.19065363])

This is very handy because we can now use array operations on the data in each row.

**Distance Between Points When There are Two Attributes**

The main calculation we need to do is to find the distance between Alice’s point and any other point. For this, the first thing we need is a way to compute the distance between any pair of points.

How do we do this? In 2-dimensional space, it’s pretty easy. If we have a point at coordinates (x0,y0)(x0,y0) and another at (x1,y1)(x1,y1), the distance between them is

D=√(x0−x1)2+(y0−y1)2D=(x0−x1)2+(y0−y1)2

(Where did this come from? It comes from the Pythogorean theorem: we have a right triangle with side lengths x0−x1x0−x1 and y0−y1y0−y1, and we want to find the length of the hypotenuse.)

In the next section we’ll see that this formula has a straightforward extension when there are more than two attributes. For now, let’s use the formula and array operations to find the distance between Alice and the patient in Row 3.

patient3 **=** np**.**array(ckd\_attributes**.**row(3))

alice, patient3

(array([0. , 1.1]), array([ 0.59610766, -0.19065363]))

distance **=** np**.**sqrt(np**.**sum((alice **-** patient3)**\*\***2))

distance

1.421664918881847

We’re going to need the distance between Alice and a bunch of points, so let’s write a function called distance that computes the distance between any pair of points. The function will take two arrays, each containing the (x,y)(x,y) coordinates of a point. (Remember, those are really the Hemoglobin and Glucose levels of a patient.)

**def** **distance**(point1, point2):

"""Returns the Euclidean distance between point1 and point2.

Each argument is an array containing the coordinates of a point."""

**return** np**.**sqrt(np**.**sum((point1 **-** point2)**\*\***2))

distance(alice, patient3)

1.421664918881847

We have begun to build our classifier: the distance function is the first building block. Now let’s work on the next piece.

**Using apply on an Entire Row**

Recall that if you want to apply a function to each element of a column of a table, one way to do that is by the call table\_name.apply(function\_name, column\_label). This evaluates to an array consisting of the values of the function when we call it on each element of the column. So each entry of the array is based on the corresponding row of the table.

If you use apply without specifying a column label, then the entire row is passed to the function. Let’s see how this works on a very small table t containing the information about the first five patients in the training sample.

t **=** ckd\_attributes**.**take(np**.**arange(5))

t

| **Hemoglobin** | **Glucose** |
| --- | --- |
| 0.456884 | 0.133751 |
| 1.153 | -0.947597 |
| 0.770138 | -0.762223 |
| 0.596108 | -0.190654 |
| -0.239236 | -0.49961 |

Just as an example, suppose that for each patient we want to know how unusual their most unusual attribute is. Concretely, if a patient’s hemoglobin level is further from the average than her glucose level, we want to know how far it is from the average. If her glucose level is further from the average than her hemoglobin level, we want to know how far that is from the average instead.

That’s the same as taking the maximum of the absolute values of the two quantities. To do this for a particular row, we can convert the row to an array and use array operations.

**def** **max\_abs**(row):

**return** np**.**max(np**.**abs(np**.**array(row)))

max\_abs(t**.**row(4))

0.4996102825918697

And now we can apply max\_abs to each row of the table t:

t**.**apply(max\_abs)

array([0.4568837 , 1.15300352, 0.77013762, 0.59610766, 0.49961028])

This way of using apply will help us create the next building block of our classifier.

**Alice’s**kk**Nearest Neighbors**

If we want to classify Alice using a k-nearest neighbor classifier, we have to identify her kk nearest neighbors. What are the steps in this process? Suppose k=5k=5. Then the steps are:

* **Step 1.** Find the distance between Alice and each point in the training sample.
* **Step 2.** Sort the data table in increasing order of the distances.
* **Step 3.** Take the top 5 rows of the sorted table.

Steps 2 and 3 seem straightforward, provided we have the distances. So let’s focus on Step 1.

Here’s Alice:

alice

array([0. , 1.1])

What we need is a function that finds the distance between Alice and another point whose coordinates are contained in a row. The function distance returns the distance between any two points whose coordinates are in arrays. We can use that to define distance\_from\_alice, which takes a row as its argument and returns the distance between that row and Alice.

**def** **distance\_from\_alice**(row):

"""Returns distance between Alice and a row of the attributes table"""

**return** distance(alice, np**.**array(row))

distance\_from\_alice(ckd\_attributes**.**row(3))

1.421664918881847

Now we can apply the function distance\_from\_alice to each row of ckd\_attributes, and augment the table ckd with the distances. Step 1 is complete!

distances **=** ckd\_attributes**.**apply(distance\_from\_alice)

ckd\_with\_distances **=** ckd**.**with\_column('Distance from Alice', distances)

ckd\_with\_distances

| **Class** | **Hemoglobin** | **Glucose** | **Color** | **Distance from Alice** |
| --- | --- | --- | --- | --- |
| 0 | 0.456884 | 0.133751 | gold | 1.06882 |
| 0 | 1.153 | -0.947597 | gold | 2.34991 |
| 0 | 0.770138 | -0.762223 | gold | 2.01519 |
| 0 | 0.596108 | -0.190654 | gold | 1.42166 |
| 0 | -0.239236 | -0.49961 | gold | 1.6174 |
| 0 | -0.0304002 | -0.159758 | gold | 1.26012 |
| 0 | 0.282854 | -0.00527964 | gold | 1.1409 |
| 0 | 0.108824 | -0.623193 | gold | 1.72663 |
| 0 | 0.0740178 | -0.515058 | gold | 1.61675 |
| 0 | 0.83975 | -0.422371 | gold | 1.73862 |

... (148 rows omitted)

For Step 2, let’s sort the table in increasing order of distance:

sorted\_by\_distance **=** ckd\_with\_distances**.**sort('Distance from Alice')

sorted\_by\_distance

| **Class** | **Hemoglobin** | **Glucose** | **Color** | **Distance from Alice** |
| --- | --- | --- | --- | --- |
| 1 | 0.83975 | 1.2151 | darkblue | 0.847601 |
| 1 | -0.970162 | 1.27689 | darkblue | 0.986156 |
| 0 | -0.0304002 | 0.0874074 | gold | 1.01305 |
| 0 | 0.14363 | 0.0874074 | gold | 1.02273 |
| 1 | -0.413266 | 2.04928 | darkblue | 1.03534 |
| 0 | 0.387272 | 0.118303 | gold | 1.05532 |
| 0 | 0.456884 | 0.133751 | gold | 1.06882 |
| 0 | 0.178436 | 0.0410639 | gold | 1.07386 |
| 0 | 0.00440582 | 0.025616 | gold | 1.07439 |
| 0 | -0.169624 | 0.025616 | gold | 1.08769 |

... (148 rows omitted)

Step 3: The top 5 rows correspond to Alice’s 5 nearest neighbors; you can replace 5 by any other positive integer.

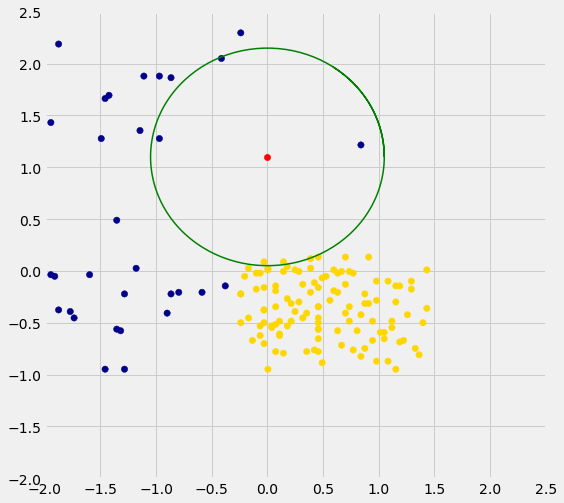
alice\_5\_nearest\_neighbors **=** sorted\_by\_distance**.**take(np**.**arange(5))

alice\_5\_nearest\_neighbors

| **Class** | **Hemoglobin** | **Glucose** | **Color** | **Distance from Alice** |
| --- | --- | --- | --- | --- |
| 1 | 0.83975 | 1.2151 | darkblue | 0.847601 |
| 1 | -0.970162 | 1.27689 | darkblue | 0.986156 |
| 0 | -0.0304002 | 0.0874074 | gold | 1.01305 |
| 0 | 0.14363 | 0.0874074 | gold | 1.02273 |
| 1 | -0.413266 | 2.04928 | darkblue | 1.03534 |

Three of Alice’s five nearest neighbors are blue points and two are gold. So a 5-nearest neighbor classifier would classify Alice as blue: it would predict that Alice has chronic kidney disease.

The graph below zooms in on Alice and her five nearest neighbors. The two gold ones just inside the circle directly below the red point. The classifier says Alice is more like the three blue ones around her.



We are well on our way to implementing our k-nearest neighbor classifier. In the next two sections we will put it together and assess its accuracy.

**Implementing the Classifier**

We are now ready to impelment a kk-nearest neighbor classifier based on multiple attributes. We have used only two attributes so far, for ease of visualization. But usually predictions will be based on many attributes. Here is an example that shows how multiple attributes can be better than pairs.

**Banknote authentication**

This time we’ll look at predicting whether a banknote (e.g., a $20 bill) is counterfeit or legitimate. Researchers have put together a data set for us, based on photographs of many individual banknotes: some counterfeit, some legitimate. They computed a few numbers from each image, using techniques that we won’t worry about for this course. So, for each banknote, we know a few numbers that were computed from a photograph of it as well as its class (whether it is counterfeit or not). Let’s load it into a table and take a look.

banknotes **=** Table**.**read\_table(path\_data **+** 'banknote.csv')

banknotes

| **WaveletVar** | **WaveletSkew** | **WaveletCurt** | **Entropy** | **Class** |
| --- | --- | --- | --- | --- |
| 3.6216 | 8.6661 | -2.8073 | -0.44699 | 0 |
| 4.5459 | 8.1674 | -2.4586 | -1.4621 | 0 |
| 3.866 | -2.6383 | 1.9242 | 0.10645 | 0 |
| 3.4566 | 9.5228 | -4.0112 | -3.5944 | 0 |
| 0.32924 | -4.4552 | 4.5718 | -0.9888 | 0 |
| 4.3684 | 9.6718 | -3.9606 | -3.1625 | 0 |
| 3.5912 | 3.0129 | 0.72888 | 0.56421 | 0 |
| 2.0922 | -6.81 | 8.4636 | -0.60216 | 0 |
| 3.2032 | 5.7588 | -0.75345 | -0.61251 | 0 |
| 1.5356 | 9.1772 | -2.2718 | -0.73535 | 0 |

... (1362 rows omitted)

Let’s look at whether the first two numbers tell us anything about whether the banknote is counterfeit or not. Here’s a scatterplot:

color\_table **=** Table()**.**with\_columns(

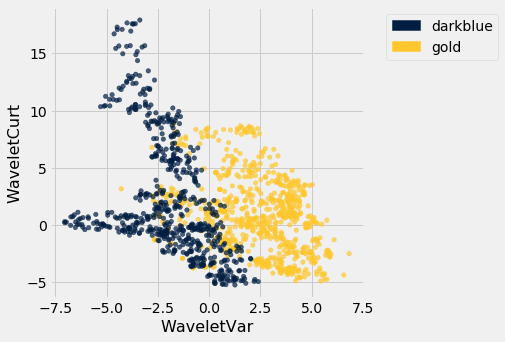
'Class', make\_array(1, 0),

'Color', make\_array('darkblue', 'gold')

)

banknotes **=** banknotes**.**join('Class', color\_table)

banknotes**.**scatter('WaveletVar', 'WaveletCurt', colors**=**'Color')

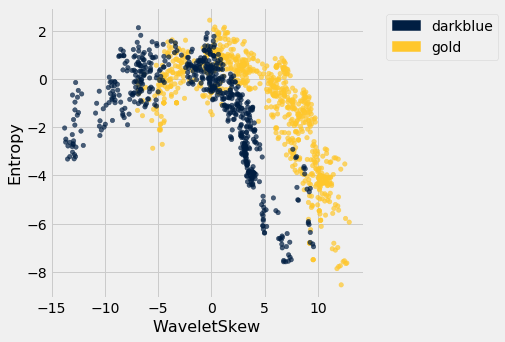


Pretty interesting! Those two measurements do seem helpful for predicting whether the banknote is counterfeit or not. However, in this example you can now see that there is some overlap between the blue cluster and the gold cluster. This indicates that there will be some images where it’s hard to tell whether the banknote is legitimate based on just these two numbers. Still, you could use a kk-nearest neighbor classifier to predict the legitimacy of a banknote.

Take a minute and think it through: Suppose we used k=11k=11 (say). What parts of the plot would the classifier get right, and what parts would it make errors on? What would the decision boundary look like?

The patterns that show up in the data can get pretty wild. For instance, here’s what we’d get if used a different pair of measurements from the images:

banknotes**.**scatter('WaveletSkew', 'Entropy', colors**=**'Color')



There does seem to be a pattern, but it’s a pretty complex one. Nonetheless, the kk-nearest neighbors classifier can still be used and will effectively “discover” patterns out of this. This illustrates how powerful machine learning can be: it can effectively take advantage of even patterns that we would not have anticipated, or that we would have thought to “program into” the computer.

**Multiple attributes**

So far I’ve been assuming that we have exactly 2 attributes that we can use to help us make our prediction. What if we have more than 2? For instance, what if we have 3 attributes?

Here’s the cool part: you can use the same ideas for this case, too. All you have to do is make a 3-dimensional scatterplot, instead of a 2-dimensional plot. You can still use the kk-nearest neighbors classifier, but now computing distances in 3 dimensions instead of just 2. It just works. Very cool!

In fact, there’s nothing special about 2 or 3. If you have 4 attributes, you can use the kk-nearest neighbors classifier in 4 dimensions. 5 attributes? Work in 5-dimensional space. And no need to stop there! This all works for arbitrarily many attributes; you just work in a very high dimensional space. It gets wicked-impossible to visualize, but that’s OK. The computer algorithm generalizes very nicely: all you need is the ability to compute the distance, and that’s not hard. Mind-blowing stuff!

For instance, let’s see what happens if we try to predict whether a banknote is counterfeit or not using 3 of the measurements, instead of just 2. Here’s what you get:

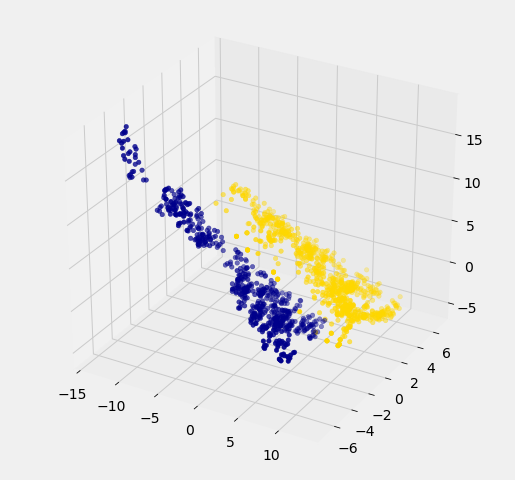
ax **=** plt**.**figure(figsize**=**(8,8))**.**add\_subplot(111, projection**=**'3d')

ax**.**scatter(banknotes**.**column('WaveletSkew'),

banknotes**.**column('WaveletVar'),

banknotes**.**column('WaveletCurt'),

c**=**banknotes**.**column('Color'));



Awesome! With just 2 attributes, there was some overlap between the two clusters (which means that the classifier was bound to make some mistakes for pointers in the overlap). But when we use these 3 attributes, the two clusters have almost no overlap. In other words, a classifier that uses these 3 attributes will be more accurate than one that only uses the 2 attributes.

This is a general phenomenom in classification. Each attribute can potentially give you new information, so more attributes sometimes helps you build a better classifier. Of course, the cost is that now we have to gather more information to measure the value of each attribute, but this cost may be well worth it if it significantly improves the accuracy of our classifier.

To sum up: you now know how to use kk-nearest neighbor classification to predict the answer to a yes/no question, based on the values of some attributes, assuming you have a training set with examples where the correct prediction is known. The general roadmap is this:

1. identify some attributes that you think might help you predict the answer to the question.
2. Gather a training set of examples where you know the values of the attributes as well as the correct prediction.
3. To make predictions in the future, measure the value of the attributes and then use kk-nearest neighbor classification to predict the answer to the question.

**Distance in Multiple Dimensions**

We know how to compute distance in 2-dimensional space. If we have a point at coordinates (x0,y0)(x0,y0) and another at (x1,y1)(x1,y1), the distance between them is

D=√(x0−x1)2+(y0−y1)2.D=(x0−x1)2+(y0−y1)2.

In 3-dimensional space, the points are (x0,y0,z0)(x0,y0,z0) and (x1,y1,z1)(x1,y1,z1), and the formula for the distance between them is

D=√(x0−x1)2+(y0−y1)2+(z0−z1)2D=(x0−x1)2+(y0−y1)2+(z0−z1)2

In nn-dimensional space, things are a bit harder to visualize, but I think you can see how the formula generalized: we sum up the squares of the differences between each individual coordinate, and then take the square root of that.

In the last section, we defined the function distance which returned the distance between two points. We used it in two-dimensions, but the great news is that the function doesn’t care how many dimensions there are! It just subtracts the two arrays of coordinates (no matter how long the arrays are), squares the differences and adds up, and then takes the square root. To work in multiple dimensions, we don’t have to change the code at all.

**def** **distance**(point1, point2):

"""Returns the distance between point1 and point2

where each argument is an array

consisting of the coordinates of the point"""

**return** np**.**sqrt(np**.**sum((point1 **-** point2)**\*\***2))

Let’s use this on a [new dataset](https://archive.ics.uci.edu/ml/datasets/Wine). The table wine contains the chemical composition of 178 different Italian wines. The classes are the grape species, called cultivars. There are three classes but let’s just see whether we can tell Class 1 apart from the other two.

wine **=** Table**.**read\_table(path\_data **+** 'wine.csv')

*# For converting Class to binary*

**def** **is\_one**(x):

**if** x **==** 1:

**return** 1

**else**:

**return** 0

wine **=** wine**.**with\_column('Class', wine**.**apply(is\_one, 0))

wine

| **Class** | **Alcohol** | **Malic Acid** | **Ash** | **Alcalinity of Ash** | **Magnesium** | **Total Phenols** | **Flavanoids** | **Nonflavanoid phenols** | **Proanthocyanins** | **Color Intensity** | **Hue** | **OD280/OD315 of diulted wines** | **Proline** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 14.23 | 1.71 | 2.43 | 15.6 | 127 | 2.8 | 3.06 | 0.28 | 2.29 | 5.64 | 1.04 | 3.92 | 1065 |
| 1 | 13.2 | 1.78 | 2.14 | 11.2 | 100 | 2.65 | 2.76 | 0.26 | 1.28 | 4.38 | 1.05 | 3.4 | 1050 |
| 1 | 13.16 | 2.36 | 2.67 | 18.6 | 101 | 2.8 | 3.24 | 0.3 | 2.81 | 5.68 | 1.03 | 3.17 | 1185 |
| 1 | 14.37 | 1.95 | 2.5 | 16.8 | 113 | 3.85 | 3.49 | 0.24 | 2.18 | 7.8 | 0.86 | 3.45 | 1480 |
| 1 | 13.24 | 2.59 | 2.87 | 21 | 118 | 2.8 | 2.69 | 0.39 | 1.82 | 4.32 | 1.04 | 2.93 | 735 |
| 1 | 14.2 | 1.76 | 2.45 | 15.2 | 112 | 3.27 | 3.39 | 0.34 | 1.97 | 6.75 | 1.05 | 2.85 | 1450 |
| 1 | 14.39 | 1.87 | 2.45 | 14.6 | 96 | 2.5 | 2.52 | 0.3 | 1.98 | 5.25 | 1.02 | 3.58 | 1290 |
| 1 | 14.06 | 2.15 | 2.61 | 17.6 | 121 | 2.6 | 2.51 | 0.31 | 1.25 | 5.05 | 1.06 | 3.58 | 1295 |
| 1 | 14.83 | 1.64 | 2.17 | 14 | 97 | 2.8 | 2.98 | 0.29 | 1.98 | 5.2 | 1.08 | 2.85 | 1045 |
| 1 | 13.86 | 1.35 | 2.27 | 16 | 98 | 2.98 | 3.15 | 0.22 | 1.85 | 7.22 | 1.01 | 3.55 | 1045 |

... (168 rows omitted)

The first two wines are both in Class 1. To find the distance between them, we first need a table of just the attributes:

wine\_attributes **=** wine**.**drop('Class')

distance(np**.**array(wine\_attributes**.**row(0)), np**.**array(wine\_attributes**.**row(1)))

31.265012394048398

The last wine in the table is of Class 0. Its distance from the first wine is:

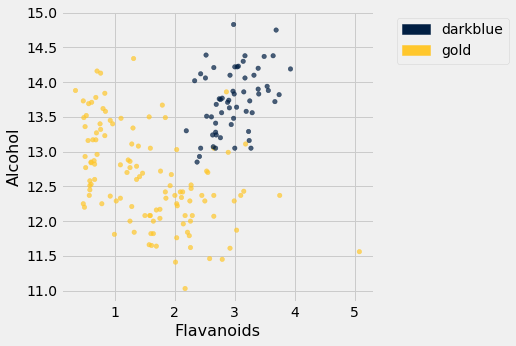
distance(np**.**array(wine\_attributes**.**row(0)), np**.**array(wine\_attributes**.**row(177)))

506.05936766351834

That’s quite a bit bigger! Let’s do some visualization to see if Class 1 really looks different from Class 0.

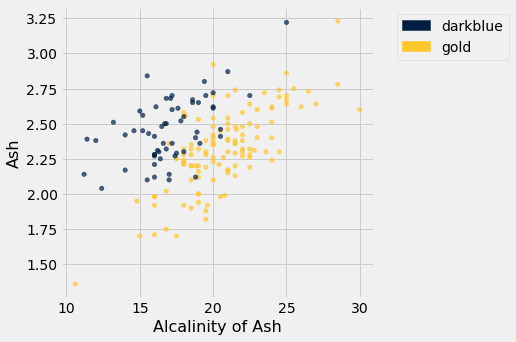
wine\_with\_colors **=** wine**.**join('Class', color\_table)

wine\_with\_colors**.**scatter('Flavanoids', 'Alcohol', colors**=**'Color')



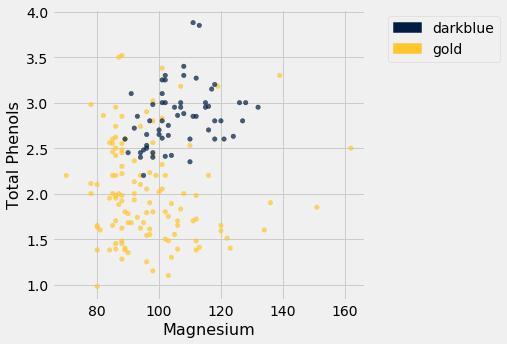
The blue points (Class 1) are almost entirely separate from the gold ones. That is one indication of why the distance between two Class 1 wines would be smaller than the distance between wines of two different classes. We can see a similar phenomenon with a different pair of attributes too:

wine\_with\_colors**.**scatter('Alcalinity of Ash', 'Ash', colors**=**'Color')



But for some pairs the picture is more murky.

wine\_with\_colors**.**scatter('Magnesium', 'Total Phenols', colors**=**'Color')



Let’s see if we can implement a classifier based on all of the attributes. After that, we’ll see how accurate it is.

**A Plan for the Implementation**

It’s time to write some code to implement the classifier. The input is a point that we want to classify. The classifier works by finding the kk nearest neighbors of point from the training set. So, our approach will go like this:

1. Find the closest kk neighbors of point, i.e., the kk wines from the training set that are most similar to point.
2. Look at the classes of those kk neighbors, and take the majority vote to find the most-common class of wine. Use that as our predicted class for point.

So that will guide the structure of our Python code.

**def** **closest**(training, p, k):

**...**

**def** **majority**(topkclasses):

**...**

**def** **classify**(training, p, k):

kclosest **=** closest(training, p, k)

kclosest**.**classes **=** kclosest**.**select('Class')

**return** majority(kclosest)

**Implementation Step 1**

To implement the first step for the kidney disease data, we had to compute the distance from each patient in the training set to point, sort them by distance, and take the kk closest patients in the training set.

That’s what we did in the previous section with the point corresponding to Alice. Let’s generalize that code. We’ll redefine distance here, just for convenience.

**def** **distance**(point1, point2):

"""Returns the distance between point1 and point2

where each argument is an array

consisting of the coordinates of the point"""

**return** np**.**sqrt(np**.**sum((point1 **-** point2)**\*\***2))

**def** **all\_distances**(training, new\_point):

"""Returns an array of distances

between each point in the training set

and the new point (which is a row of attributes)"""

attributes **=** training**.**drop('Class')

**def** **distance\_from\_point**(row):

**return** distance(np**.**array(new\_point), np**.**array(row))

**return** attributes**.**apply(distance\_from\_point)

**def** **table\_with\_distances**(training, new\_point):

"""Augments the training table

with a column of distances from new\_point"""

**return** training**.**with\_column('Distance', all\_distances(training, new\_point))

**def** **closest**(training, new\_point, k):

"""Returns a table of the k rows of the augmented table

corresponding to the k smallest distances"""

with\_dists **=** table\_with\_distances(training, new\_point)

sorted\_by\_distance **=** with\_dists**.**sort('Distance')

topk **=** sorted\_by\_distance**.**take(np**.**arange(k))

**return** topk

Let’s see how this works on our wine data. We’ll just take the first wine and find its five nearest neighbors among all the wines. Remember that since this wine is part of the dataset, it is its own nearest neighbor. So we should expect to see it at the top of the list, followed by four others.

First let’s extract its attributes:

special\_wine **=** wine**.**drop('Class')**.**row(0)

And now let’s find its 5 nearest neighbors.

closest(wine, special\_wine, 5)

| **Class** | **Alcohol** | **Malic Acid** | **Ash** | **Alcalinity of Ash** | **Magnesium** | **Total Phenols** | **Flavanoids** | **Nonflavanoid phenols** | **Proanthocyanins** | **Color Intensity** | **Hue** | **OD280/OD315 of diulted wines** | **Proline** | **Distance** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 14.23 | 1.71 | 2.43 | 15.6 | 127 | 2.8 | 3.06 | 0.28 | 2.29 | 5.64 | 1.04 | 3.92 | 1065 | 0 |
| 1 | 13.74 | 1.67 | 2.25 | 16.4 | 118 | 2.6 | 2.9 | 0.21 | 1.62 | 5.85 | 0.92 | 3.2 | 1060 | 10.3928 |
| 1 | 14.21 | 4.04 | 2.44 | 18.9 | 111 | 2.85 | 2.65 | 0.3 | 1.25 | 5.24 | 0.87 | 3.33 | 1080 | 22.3407 |
| 1 | 14.1 | 2.02 | 2.4 | 18.8 | 103 | 2.75 | 2.92 | 0.32 | 2.38 | 6.2 | 1.07 | 2.75 | 1060 | 24.7602 |
| 1 | 14.38 | 3.59 | 2.28 | 16 | 102 | 3.25 | 3.17 | 0.27 | 2.19 | 4.9 | 1.04 | 3.44 | 1065 | 25.0947 |

Bingo! The first row is the nearest neighbor, which is itself – there’s a 0 in the Distance column as expected. All five nearest neighbors are of Class 1, which is consistent with our earlier observation that Class 1 wines appear to be clumped together in some dimensions.

**Implementation Steps 2 and 3**

Next we need to take a “majority vote” of the nearest neighbors and assign our point the same class as the majority.

**def** **majority**(topkclasses):

ones **=** topkclasses**.**where('Class', are**.**equal\_to(1))**.**num\_rows

zeros **=** topkclasses**.**where('Class', are**.**equal\_to(0))**.**num\_rows

**if** ones **>** zeros:

**return** 1

**else**:

**return** 0

**def** **classify**(training, new\_point, k):

closestk **=** closest(training, new\_point, k)

topkclasses **=** closestk**.**select('Class')

**return** majority(topkclasses)

classify(wine, special\_wine, 5)

1

If we change special\_wine to be the last one in the dataset, is our classifier able to tell that it’s in Class 0?

special\_wine **=** wine**.**drop('Class')**.**row(177)

classify(wine, special\_wine, 5)

0

Yes! The classifier gets this one right too.

But we don’t yet know how it does with all the other wines, and in any case we know that testing on wines that are already part of the training set might be over-optimistic. In the final section of this chapter, we will separate the wines into a training and test set and then measure the accuracy of our classifier on the test set.

**The Accuracy of the Classifier**

To see how well our classifier does, we might put 50% of the data into the training set and the other 50% into the test set. Basically, we are setting aside some data for later use, so we can use it to measure the accuracy of our classifier. We’ve been calling that the *test set*. Sometimes people will call the data that you set aside for testing a *hold-out set*, and they’ll call this strategy for estimating accuracy the *hold-out method*.

Note that this approach requires great discipline. Before you start applying machine learning methods, you have to take some of your data and set it aside for testing. You must avoid using the test set for developing your classifier: you shouldn’t use it to help train your classifier or tweak its settings or for brainstorming ways to improve your classifier. Instead, you should use it only once, at the very end, after you’ve finalized your classifier, when you want an unbiased estimate of its accuracy.

**Measuring the Accuracy of Our Wine Classifier**

OK, so let’s apply the hold-out method to evaluate the effectiveness of the kk-nearest neighbor classifier for identifying wines. The data set has 178 wines, so we’ll randomly permute the data set and put 89 of them in the training set and the remaining 89 in the test set.

shuffled\_wine **=** wine**.**sample(with\_replacement**=**False)

training\_set **=** shuffled\_wine**.**take(np**.**arange(89))

test\_set **=** shuffled\_wine**.**take(np**.**arange(89, 178))

We’ll train the classifier using the 89 wines in the training set, and evaluate how well it performs on the test set. To make our lives easier, we’ll write a function to evaluate a classifier on every wine in the test set:

**def** **count\_zero**(array):

"""Counts the number of 0's in an array"""

**return** len(array) **-** np**.**count\_nonzero(array)

**def** **count\_equal**(array1, array2):

"""Takes two numerical arrays of equal length

and counts the indices where the two are equal"""

**return** count\_zero(array1 **-** array2)

**def** **evaluate\_accuracy**(training, test, k):

test\_attributes **=** test**.**drop('Class')

**def** **classify\_testrow**(row):

**return** classify(training, row, k)

c **=** test\_attributes**.**apply(classify\_testrow)

**return** count\_equal(c, test**.**column('Class')) **/** test**.**num\_rows

Now for the grand reveal – let’s see how we did. We’ll arbitrarily use k=5k=5.

evaluate\_accuracy(training\_set, test\_set, 5)

0.9438202247191011

The accuracy rate isn’t bad at all for a simple classifier.

**Breast Cancer Diagnosis**

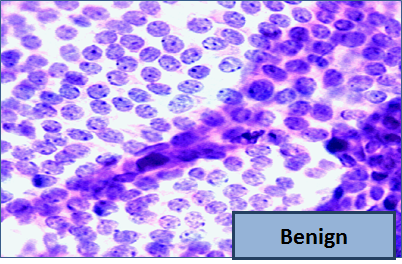
Now I want to do an example based on diagnosing breast cancer. I was inspired by Brittany Wenger, who won the Google national science fair in 2012 a 17-year old high school student. Here’s Brittany:

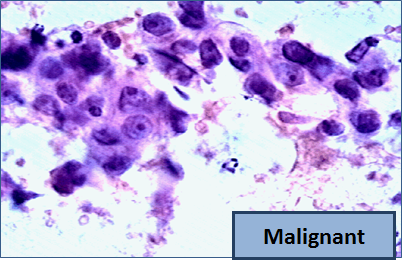


Brittany’s [science fair project](https://sites.google.com/a/googlesciencefair.com/science-fair-2012-project-64a91af142a459cfb486ed5cb05f803b2eb41354-1333130785-87/home) was to build a classification algorithm to diagnose breast cancer. She won grand prize for building an algorithm whose accuracy was almost 99%.

Let’s see how well we can do, with the ideas we’ve learned in this course.

So, let me tell you a little bit about the data set. Basically, if a woman has a lump in her breast, the doctors may want to take a biopsy to see if it is cancerous. There are several different procedures for doing that. Brittany focused on fine needle aspiration (FNA), because it is less invasive than the alternatives. The doctor gets a sample of the mass, puts it under a microscope, takes a picture, and a trained lab tech analyzes the picture to determine whether it is cancer or not. We get a picture like one of the following:





Unfortunately, distinguishing between benign vs malignant can be tricky. So, researchers have studied the use of machine learning to help with this task. The idea is that we’ll ask the lab tech to analyze the image and compute various attributes: things like the typical size of a cell, how much variation there is among the cell sizes, and so on. Then, we’ll try to use this information to predict (classify) whether the sample is malignant or not. We have a training set of past samples from women where the correct diagnosis is known, and we’ll hope that our machine learning algorithm can use those to learn how to predict the diagnosis for future samples.

We end up with the following data set. For the “Class” column, 1 means malignant (cancer); 0 means benign (not cancer).

patients **=** Table**.**read\_table(path\_data **+** 'breast-cancer.csv')**.**drop('ID')

patients

| **Clump Thickness** | **Uniformity of Cell Size** | **Uniformity of Cell Shape** | **Marginal Adhesion** | **Single Epithelial Cell Size** | **Bare Nuclei** | **Bland Chromatin** | **Normal Nucleoli** | **Mitoses** | **Class** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 5 | 1 | 1 | 1 | 2 | 1 | 3 | 1 | 1 | 0 |
| 5 | 4 | 4 | 5 | 7 | 10 | 3 | 2 | 1 | 0 |
| 3 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 1 | 0 |
| 6 | 8 | 8 | 1 | 3 | 4 | 3 | 7 | 1 | 0 |
| 4 | 1 | 1 | 3 | 2 | 1 | 3 | 1 | 1 | 0 |
| 8 | 10 | 10 | 8 | 7 | 10 | 9 | 7 | 1 | 1 |
| 1 | 1 | 1 | 1 | 2 | 10 | 3 | 1 | 1 | 0 |
| 2 | 1 | 2 | 1 | 2 | 1 | 3 | 1 | 1 | 0 |
| 2 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 5 | 0 |
| 4 | 2 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 0 |

... (673 rows omitted)

So we have 9 different attributes. I don’t know how to make a 9-dimensional scatterplot of all of them, so I’m going to pick two and plot them:

color\_table **=** Table()**.**with\_columns(

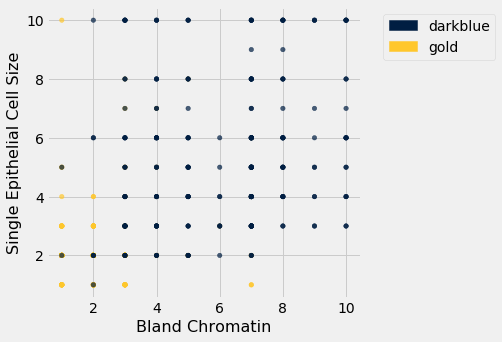
'Class', make\_array(1, 0),

'Color', make\_array('darkblue', 'gold')

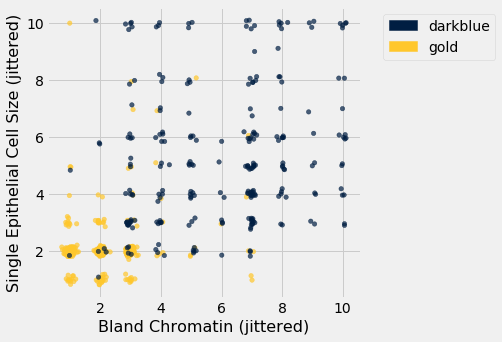
)

patients\_with\_colors **=** patients**.**join('Class', color\_table)

patients\_with\_colors**.**scatter('Bland Chromatin', 'Single Epithelial Cell Size', colors**=**'Color')



Oops. That plot is utterly misleading, because there are a bunch of points that have identical values for both the x- and y-coordinates. To make it easier to see all the data points, I’m going to add a little bit of random jitter to the x- and y-values. Here’s how that looks:



For instance, you can see there are lots of samples with chromatin = 2 and epithelial cell size = 2; all non-cancerous.

Keep in mind that the jittering is just for visualization purposes, to make it easier to get a feeling for the data. We’re ready to work with the data now, and we’ll use the original (unjittered) data.

First we’ll create a training set and a test set. The data set has 683 patients, so we’ll randomly permute the data set and put 342 of them in the training set and the remaining 341 in the test set.

shuffled\_patients **=** patients**.**sample(683, with\_replacement**=**False)

training\_set **=** shuffled\_patients**.**take(np**.**arange(342))

test\_set **=** shuffled\_patients**.**take(np**.**arange(342, 683))

Let’s stick with 5 nearest neighbors, and see how well our classifier does.

evaluate\_accuracy(training\_set, test\_set, 5)

0.9765395894428153

Over 96% accuracy. Not bad! Once again, pretty darn good for such a simple technique.

As a footnote, you might have noticed that Brittany Wenger did even better. What techniques did she use? One key innovation is that she incorporated a confidence score into her results: her algorithm had a way to determine when it was not able to make a confident prediction, and for those patients, it didn’t even try to predict their diagnosis. Her algorithm was 99% accurate on the patients where it made a prediction – so that extension seemed to help quite a bit.

Now that we have explored ways to use multiple attributes to predict a categorical variable, let us return to predicting a quantitative variable. Predicting a numerical quantity is called regression, and a commonly used method to use multiple attributes for regression is called multiple linear regression.

## Home Prices

The following dataset of house prices and attributes was collected over several years for the city of Ames, Iowa. A [description of the dataset appears online](http://ww2.amstat.org/publications/jse/v19n3/decock.pdf). We will focus only a subset of the columns. We will try to predict the sale price column from the other columns.

all\_sales **=** Table**.**read\_table(path\_data **+** 'house.csv')

sales **=** all\_sales**.**where('Bldg Type', '1Fam')**.**where('Sale Condition', 'Normal')**.**select(

'SalePrice', '1st Flr SF', '2nd Flr SF',

'Total Bsmt SF', 'Garage Area',

'Wood Deck SF', 'Open Porch SF', 'Lot Area',

'Year Built', 'Yr Sold')

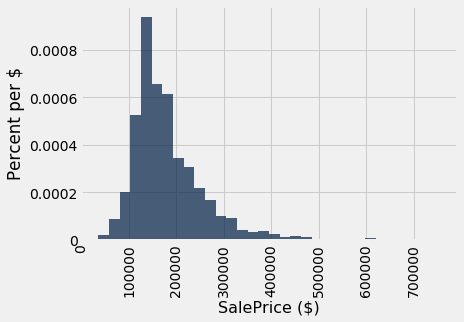
sales**.**sort('SalePrice')

| **SalePrice** | **1st Flr SF** | **2nd Flr SF** | **Total Bsmt SF** | **Garage Area** | **Wood Deck SF** | **Open Porch SF** | **Lot Area** | **Year Built** | **Yr Sold** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 35000 | 498 | 0 | 498 | 216 | 0 | 0 | 8088 | 1922 | 2006 |
| 39300 | 334 | 0 | 0 | 0 | 0 | 0 | 5000 | 1946 | 2007 |
| 40000 | 649 | 668 | 649 | 250 | 0 | 54 | 8500 | 1920 | 2008 |
| 45000 | 612 | 0 | 0 | 308 | 0 | 0 | 5925 | 1940 | 2009 |
| 52000 | 729 | 0 | 270 | 0 | 0 | 0 | 4130 | 1935 | 2008 |
| 52500 | 693 | 0 | 693 | 0 | 0 | 20 | 4118 | 1941 | 2006 |
| 55000 | 723 | 363 | 723 | 400 | 0 | 24 | 11340 | 1920 | 2008 |
| 55000 | 796 | 0 | 796 | 0 | 0 | 0 | 3636 | 1922 | 2008 |
| 57625 | 810 | 0 | 0 | 280 | 119 | 24 | 21780 | 1910 | 2009 |
| 58500 | 864 | 0 | 864 | 200 | 0 | 0 | 8212 | 1914 | 2010 |

... (1992 rows omitted)

A histogram of sale prices shows a large amount of variability and a distribution that is clearly not normal. A long tail to the right contains a few houses that had very high prices. The short left tail does not contain any houses that sold for less than $35,000.

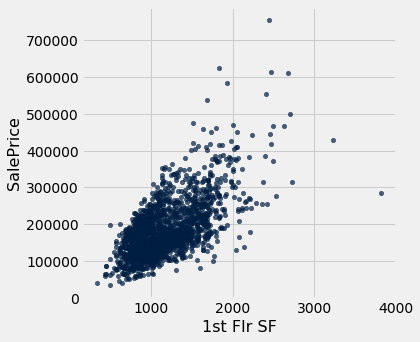
sales**.**hist('SalePrice', bins**=**32, unit**=**'$')



#### Correlation

No single attribute is sufficient to predict the sale price. For example, the area of first floor, measured in square feet, correlates with sale price but only explains some of its variability.

sales**.**scatter('1st Flr SF', 'SalePrice')



correlation(sales, 'SalePrice', '1st Flr SF')

0.6424662541030225

In fact, none of the individual attributes have a correlation with sale price that is above 0.7 (except for the sale price itself).

**for** label **in** sales**.**labels:

**print**('Correlation of', label, 'and SalePrice:\t', correlation(sales, label, 'SalePrice'))

Correlation of SalePrice and SalePrice: 1.0

Correlation of 1st Flr SF and SalePrice: 0.6424662541030225

Correlation of 2nd Flr SF and SalePrice: 0.3575218942800824

Correlation of Total Bsmt SF and SalePrice: 0.652978626757169

Correlation of Garage Area and SalePrice: 0.6385944852520443

Correlation of Wood Deck SF and SalePrice: 0.3526986661950492

Correlation of Open Porch SF and SalePrice: 0.3369094170263733

Correlation of Lot Area and SalePrice: 0.2908234551157694

Correlation of Year Built and SalePrice: 0.5651647537135916

Correlation of Yr Sold and SalePrice: 0.02594857908072111

However, combining attributes can provide higher correlation. In particular, if we sum the first floor and second floor areas, the result has a higher correlation than any single attribute alone.

both\_floors **=** sales**.**column(1) **+** sales**.**column(2)

correlation(sales**.**with\_column('Both Floors', both\_floors), 'SalePrice', 'Both Floors')

0.7821920556134877

This high correlation indicates that we should try to use more than one attribute to predict the sale price. In a dataset with multiple observed attributes and a single numerical value to be predicted (the sale price in this case), multiple linear regression can be an effective technique.

## Multiple Linear Regression

In multiple linear regression, a numerical output is predicted from numerical input attributes by multiplying each attribute value by a different slope, then summing the results. In this example, the slope for the 1st Flr SF would represent the dollars per square foot of area on the first floor of the house that should be used in our prediction.

Before we begin prediction, we split our data randomly into a training and test set of equal size.

train, test **=** sales**.**split(1001)

**print**(train**.**num\_rows, 'training and', test**.**num\_rows, 'test instances.')

1001 training and 1001 test instances.

The slopes in multiple regression is an array that has one slope value for each attribute in an example. Predicting the sale price involves multiplying each attribute by the slope and summing the result.

**def** **predict**(slopes, row):

**return** sum(slopes **\*** np**.**array(row))

example\_row **=** test**.**drop('SalePrice')**.**row(0)

**print**('Predicting sale price for:', example\_row)

example\_slopes **=** np**.**random**.**normal(10, 1, len(example\_row))

**print**('Using slopes:', example\_slopes)

**print**('Result:', predict(example\_slopes, example\_row))

Predicting sale price for: Row(1st Flr SF=707, 2nd Flr SF=707, Total Bsmt SF=707.0, Garage Area=403.0, Wood Deck SF=100, Open Porch SF=35, Lot Area=7750, Year Built=2002, Yr Sold=2008)

Using slopes: [ 9.70697704 8.68451487 9.48574052 11.65887763 9.76283493 7.75180442

10.26963618 12.39555854 9.93561073]

Result: 150011.62264018963

The result is an estimated sale price, which can be compared to the actual sale price to assess whether the slopes provide accurate predictions. Since the example\_slopes above were chosen at random, we should not expect them to provide accurate predictions at all.

**print**('Actual sale price:', test**.**column('SalePrice')**.**item(0))

**print**('Predicted sale price using random slopes:', predict(example\_slopes, example\_row))

Actual sale price: 176000

Predicted sale price using random slopes: 150011.62264018963

#### Least Squares Regression

The next step in performing multiple regression is to define the least squares objective. We perform the prediction for each row in the training set, and then compute the root mean squared error (RMSE) of the predictions from the actual prices.

train\_prices **=** train**.**column(0)

train\_attributes **=** train**.**drop(0)

**def** **rmse**(slopes, attributes, prices):

errors **=** []

**for** i **in** np**.**arange(len(prices)):

predicted **=** predict(slopes, attributes**.**row(i))

actual **=** prices**.**item(i)

errors**.**append((predicted **-** actual) **\*\*** 2)

**return** np**.**mean(errors) **\*\*** 0.5

**def** **rmse\_train**(slopes):

**return** rmse(slopes, train\_attributes, train\_prices)

**print**('RMSE of all training examples using random slopes:', rmse\_train(example\_slopes))

RMSE of all training examples using random slopes: 103585.76518182222

Finally, we use the minimize function to find the slopes with the lowest RMSE. Since the function we want to minimize, rmse\_train, takes an array instead of a number, we must pass the array=True argument to minimize. When this argument is used, minimize also requires an initial guess of the slopes so that it knows the dimension of the input array. Finally, to speed up optimization, we indicate that rmse\_train is a smooth function using the smooth=True attribute. Computation of the best slopes may take several minutes.

best\_slopes **=** minimize(rmse\_train, start**=**example\_slopes, smooth**=**True, array**=**True)

**print**('The best slopes for the training set:')

Table(train\_attributes**.**labels)**.**with\_row(list(best\_slopes))**.**show()

**print**('RMSE of all training examples using the best slopes:', rmse\_train(best\_slopes))

The best slopes for the training set:

| **1st Flr SF** | **2nd Flr SF** | **Total Bsmt SF** | **Garage Area** | **Wood Deck SF** | **Open Porch SF** | **Lot Area** | **Year Built** | **Yr Sold** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 78.7701 | 75.9304 | 49.6108 | 42.9615 | 38.8186 | 13.2336 | 0.328059 | 510.312 | -508.186 |

RMSE of all training examples using the best slopes: 32283.50513136445

#### Interpreting Multiple Regression

Let’s interpret these results. The best slopes give us a method for estimating the price of a house from its attributes. A square foot of area on the first floor is worth about $75 (the first slope), while one on the second floor is worth about $70 (the second slope). The final negative value describes the market: prices in later years were lower on average.

The RMSE of around $30,000 means that our best linear prediction of the sale price based on all of the attributes is off by around $30,000 on the training set, on average. We find a similar error when predicting prices on the test set, which indicates that our prediction method will generalize to other samples from the same population.

test\_prices **=** test**.**column(0)

test\_attributes **=** test**.**drop(0)

**def** **rmse\_test**(slopes):

**return** rmse(slopes, test\_attributes, test\_prices)

rmse\_linear **=** rmse\_test(best\_slopes)

**print**('Test set RMSE for multiple linear regression:', rmse\_linear)

Test set RMSE for multiple linear regression: 29898.407434368237

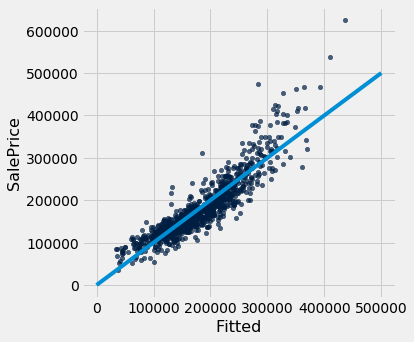
If the predictions were perfect, then a scatter plot of the predicted and actual values would be a straight line with slope 1. We see that most dots fall near that line, but there is some error in the predictions.

**def** **fit**(row):

**return** sum(best\_slopes **\*** np**.**array(row))

test**.**with\_column('Fitted', test**.**drop(0)**.**apply(fit))**.**scatter('Fitted', 0)

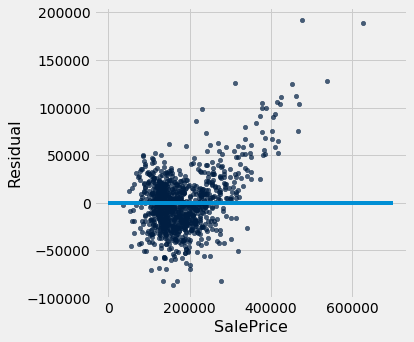
plots**.**plot([0, 5e5], [0, 5e5]);



A residual plot for multiple regression typically compares the errors (residuals) to the actual values of the predicted variable. We see in the residual plot below that we have systematically underestimated the value of expensive houses, shown by the many positive residual values on the right side of the graph.

test**.**with\_column('Residual', test\_prices**-**test**.**drop(0)**.**apply(fit))**.**scatter(0, 'Residual')

plots**.**plot([0, 7e5], [0, 0]);



As with simple linear regression, interpreting the result of a predictor is at least as important as making predictions. There are many lessons about interpreting multiple regression that are not included in this textbook. A natural next step after completing this text would be to study linear modeling and regression in further depth.

## Nearest Neighbors for Regression

Another approach to predicting the sale price of a house is to use the price of similar houses. This nearest neighbor approach is very similar to our classifier. To speed up computation, we will only use the attributes that had the highest correlation with the sale price in our original analysis.

train\_nn **=** train**.**select(0, 1, 2, 3, 4, 8)

test\_nn **=** test**.**select(0, 1, 2, 3, 4, 8)

train\_nn**.**show(3)

| **SalePrice** | **1st Flr SF** | **2nd Flr SF** | **Total Bsmt SF** | **Garage Area** | **Year Built** |
| --- | --- | --- | --- | --- | --- |
| 67500 | 1012 | 0 | 816 | 429 | 1920 |
| 116000 | 734 | 384 | 648 | 440 | 1920 |
| 228500 | 1689 | 0 | 1680 | 432 | 1991 |

... (998 rows omitted)

The computation of closest neighbors is identical to a nearest-neighbor classifier. In this case, we will exclude the 'SalePrice' rather than the 'Class' column from the distance computation. The five nearest neighbors of the first test row are shown below.

**def** **distance**(pt1, pt2):

"""The distance between two points, represented as arrays."""

**return** np**.**sqrt(sum((pt1 **-** pt2) **\*\*** 2))

**def** **row\_distance**(row1, row2):

"""The distance between two rows of a table."""

**return** distance(np**.**array(row1), np**.**array(row2))

**def** **distances**(training, example, output):

"""Compute the distance from example for each row in training."""

dists **=** []

attributes **=** training**.**drop(output)

**for** row **in** attributes**.**rows:

dists**.**append(row\_distance(row, example))

**return** training**.**with\_column('Distance', dists)

**def** **closest**(training, example, k, output):

"""Return a table of the k closest neighbors to example."""

**return** distances(training, example, output)**.**sort('Distance')**.**take(np**.**arange(k))

example\_nn\_row **=** test\_nn**.**drop(0)**.**row(0)

closest(train\_nn, example\_nn\_row, 5, 'SalePrice')

| **SalePrice** | **1st Flr SF** | **2nd Flr SF** | **Total Bsmt SF** | **Garage Area** | **Year Built** | **Distance** |
| --- | --- | --- | --- | --- | --- | --- |
| 175000 | 729 | 717 | 729 | 406 | 1996 | 33.3617 |
| 176000 | 728 | 728 | 728 | 400 | 2005 | 36.6197 |
| 189000 | 728 | 728 | 728 | 410 | 2005 | 37.1618 |
| 159500 | 698 | 728 | 690 | 440 | 1977 | 52.9623 |
| 174000 | 742 | 742 | 742 | 390 | 2005 | 62.0725 |

One simple method for predicting the price is to average the prices of the nearest neighbors.

**def** **predict\_nn**(example):

"""Return the majority class among the k nearest neighbors."""

**return** np**.**average(closest(train\_nn, example, 5, 'SalePrice')**.**column('SalePrice'))

predict\_nn(example\_nn\_row)

174700.0

Finally, we can inspect whether our prediction is close to the true sale price for our one test example. Looks reasonable!

**print**('Actual sale price:', test\_nn**.**column('SalePrice')**.**item(0))

**print**('Predicted sale price using nearest neighbors:', predict\_nn(example\_nn\_row))

Actual sale price: 176000

Predicted sale price using nearest neighbors: 174700.0

#### Evaluation

To evaluate the performance of this approach for the whole test set, we apply predict\_nn to each test example, then compute the root mean squared error of the predictions. Computation of the predictions may take several minutes.

nn\_test\_predictions **=** test\_nn**.**drop('SalePrice')**.**apply(predict\_nn)

rmse\_nn **=** np**.**mean((test\_prices **-** nn\_test\_predictions) **\*\*** 2) **\*\*** 0.5

**print**('Test set RMSE for multiple linear regression: ', rmse\_linear)

**print**('Test set RMSE for nearest neighbor regression:', rmse\_nn)

Test set RMSE for multiple linear regression: 29898.407434368237

Test set RMSE for nearest neighbor regression: 33424.833033298106

For these data, the errors of the two techniques are quite similar! For different data sets, one technique might outperform another. By computing the RMSE of both techniques on the same data, we can compare methods fairly. One note of caution: the difference in performance might not be due to the technique at all; it might be due to the random variation due to sampling the training and test sets in the first place.

Finally, we can draw a residual plot for these predictions. We still underestimate the prices of the most expensive houses, but the bias does not appear to be as systematic. However, fewer residuals are very close to zero, indicating that fewer prices were predicted with very high accuracy.

test**.**with\_column('Residual', test\_prices**-**nn\_test\_predictions)**.**scatter(0, 'Residual')

plots**.**plot([0, 7e5], [0, 0]);

