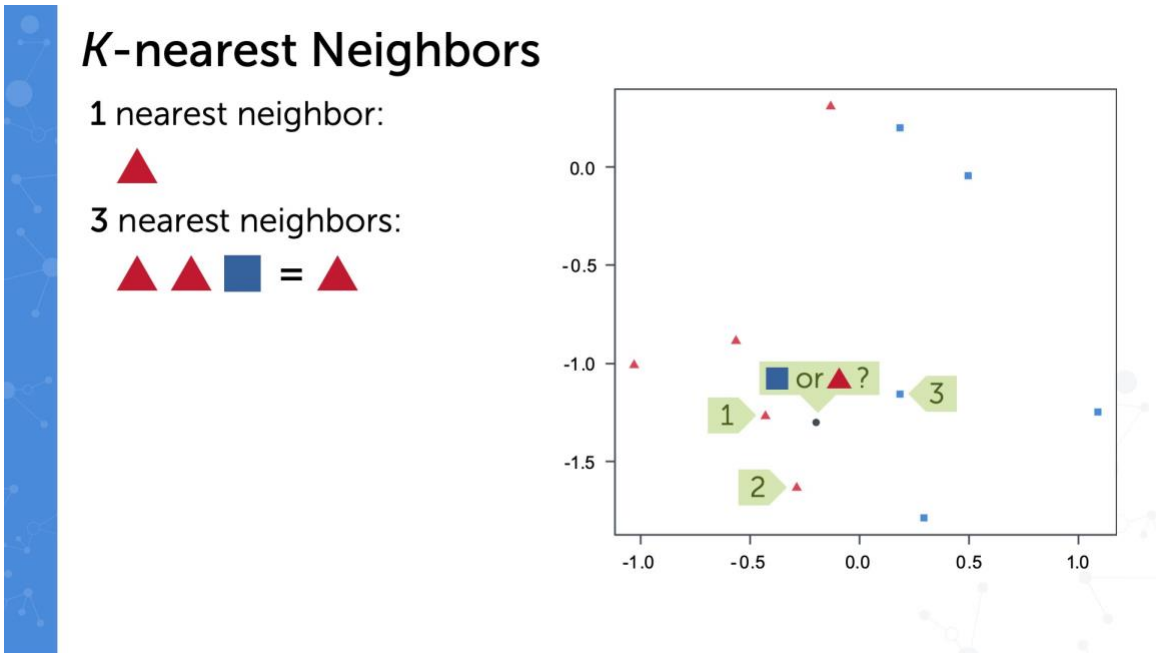
UNIVERSITY OF WISCONSIN
DATA SCIENCE

Important note: Transcripts are **not** substitutes for textbook assignments.

Learning Objectives

By the end of this lesson, you will be able to:

- Explain how *K*-nearest neighbors works for classification problems.
- Implement *K*-nearest neighbors in R.
- Choose a value for *K* based on the error rate on a validation set.
- Interpret *K*-nearest neighbors models.



K nearest-neighbors is a simple but effective method of supervised learning that can be used for classification or regression problems. It's non-parametric, which means it doesn't assume a particular shape of the model for the relationship between the predictor and response variables. Here's how it works.

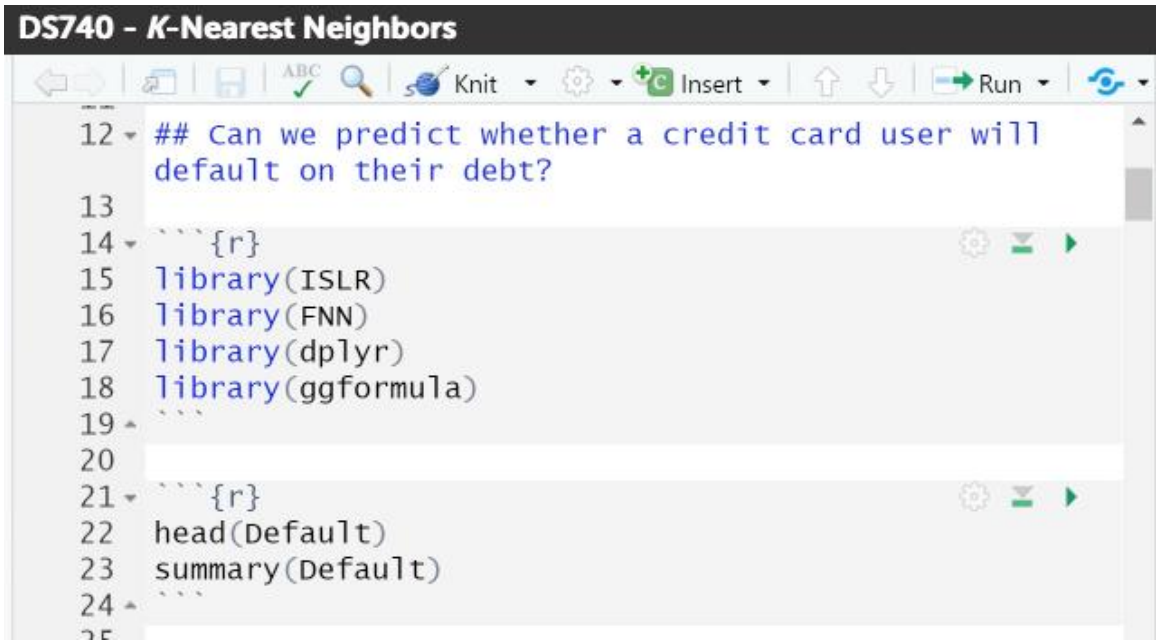
Suppose we have two categories of data, blue squares and red triangles, and we'd like to predict which category the new data point, the black circle, falls into. If we're using 1 nearest-neighbors, then we would just look at the one neighbor, the data point that's closest to that black circle. It's in the red triangle category, so that's the category that we would predict for the black circle.

For 3 nearest-neighbors, we would simply look out a bit farther and look at the three data points that are closest to the black circle. Of these, two are red triangles and only one is a blue square. So we would still assign the black circle to the red triangle category. When you have two categories, it's generally a good idea to choose K to be some odd number so that you can avoid having ties.

Sample Problem

- `Default` data set in ISLR library
- Can we predict whether a credit card user will default on their debt?

```
> install.packages("ISLR")
> Library(ISLR)
> Head(Default)
  default student   balance    income
1      No      No  729.5265 44361.625
2      No     Yes  817.1804 12106.135
3      No      No 1073.5492 31767.139
4      No      No  529.2506 35704.494
5      No      No  785.6559 38463.496
6      no     yes  919.5885  7491.559
> dim(Default)
[1] 10000     4
```



This slide represents a video/screencast in the lecture. The transcript does not substitute video content.

Let's use k nearest neighbors to predict whether people will default on their credit card debt. I've already installed the ISLR package, which contains the data set we'll be using, as well as the FNN package, which contains the function for k nearest neighbors. So I'm going to start by loading those libraries. And I'll also load dplyr and ggformula, which we'll use to manipulate the data and make some graphs later on.

Next, let's look at the data to get a sense of what we're working with. The head function will show us the first six rows of data. And we can see we have four columns. It looks like default with a lowercase D and student are categorical variables, and balance and income are quantitative variables.

The summary function will give us some basic summaries of the data. So for the categorical variables, we can see how many nos and yeses there are, and for the quantitative variables, we can see their minimum, their maximum, and the median, mean, and first and third quartiles. Looking at this information shows us that there are no NAs, so no missing data. And the quantitative values look like-- don't have any unusual values, such as people with a negative income, or anything like that. So we don't have any outliers to deal with. So let's proceed with preparing our data set for the analysis.

We want to start by making sure that all of our predictor variables are quantitative because that's what KNN expects. So we need to convert the categorical variable student into a dummy variable or indicator variable, where a 0 will indicate that no, a

person is not a student, and a 1 will indicate that, yes, they are a student. There are many different ways to do this. I'm doing it using the mutate function from dplyr and the ifelse function.

So the way to interpret this is we first to check if the value student equals no. Remember to use a double equals sign to check for equality. If that's true, then we put the value 0 in. And if it's false, then we put the value 1 in. And in doing this, I'm creating a copy of the data frame called Default2, just so that it's easy to go back to the original version of the data in case I don't like my work. So we can look at the first six lines of this new version of the data set, and we see that student is indeed changed into zeros and ones.

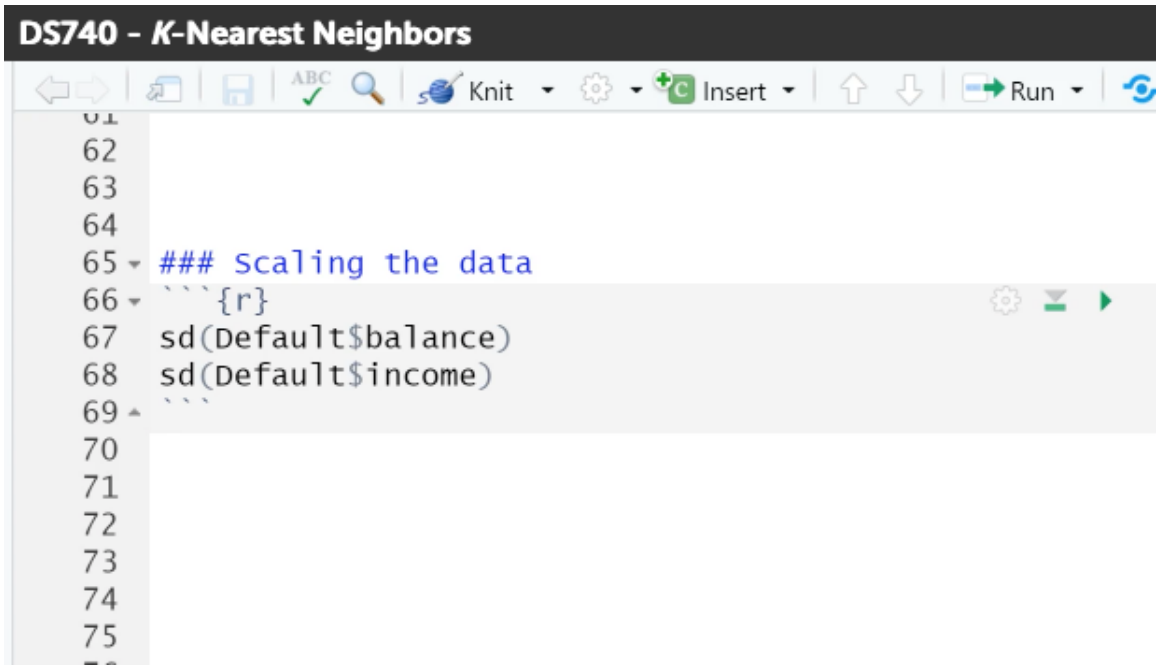
The next thing to do is to split the data into a training set and a test set. We'll use the training set to build or create the model, and then we'll use the test set to make predictions and check the model's accuracy. In this way, we're not checking the model's accuracy on the exact same data that we used to create the model. So we get a more honest sense of how well the model would do on new data.

In order to do this, I'm starting by setting a random seed using the set.seed function. And then you can put any positive integer in as an argument. I just used 123. Then I'm creating a groups vector, where I'll use the number 1 to indicate that a data point is in the training set and a number 2 to indicate that it's in the test set. It's pretty common to use about 2/3 of your data for the training set. And our data set here had 10,000 rows, which we could have seen by using the dim function, D-I-M.

So here I'm repeating the number 1 6,666 times, and I'm repeating the number 2 3,334 times. So this is great, but now the vector groups has all of the first 2/3 of the data in the training set, and all of the last 1/3 in the test set. We probably don't want to do that. It's better to mix things up and have a random sample in the training set. So now we can mix up our groups variable to create the variable random_groups using the sample function.

So here we're sampling 10,000 elements, or all of them, from the vector groups. And by default, the sample function does this without replacement. So this is simply creating a random permutation or rearrangement of the groups vector. Then we'll let in_train be equal to random groups double equal to 1. So this is going to be equal to true for all of the elements where random groups was equal to 1. So it's equal to true for all of the data points that should be in the training set. And it'll be equal to false for all the data points in the test set.

So looking at the first six elements of in_train, we see that the first two elements or first two rows are in the training set, and then the third row is in the test set, and so on.



This slide represents a video/screencast in the lecture. The transcript does not substitute video content.

Intuitively, a difference of, say, 1,000 means a different thing for each of our two quantitative variables. For example, two people who have incomes that are $1,000 apart have pretty similar incomes. But if one person has a credit card balance of $500 and another person has a credit card balance of $1,500, that's a pretty big difference. The second person's credit card balance is three times the credit card balance of the first person.

So we have this sense that we shouldn't treat $1,000 as being the same for each of these two variables. And we can verify that by looking at the standard deviations of these two variables. We see that the standard deviation of balance is just under $500. So a difference of $1,000 would be a little over two standard deviations. And the standard deviation of income is over $13,000. So a difference of $1,000 in income is less than one standard deviation-- not a very big difference.

However, k nearest neighbors uses Euclidean distance to define how similar two points are. So it would use a difference of 1,000 as being the same, regardless of which variable it came from. We don't want that. We instead want to rescale these variables so that our k nearest neighbors will work in terms of standard deviations, not in terms of raw dollar amounts.

We can do that by scaling or standardizing the data. In other words, we're subtracting the mean of each variable and dividing by the standard deviation. So each variable will now have a mean of 0 and a standard deviation of 1. Here I'm using the scale function

on columns 3 and 4, so that's balance and income, of the training set of Default2. And I'm going to store that in a new variable, quant_train_std. So in other words, the standardized version of the quantitative variables in the training set.

We want to standardize the training set by itself based on its mean and standard deviation, because the most honest way of assessing our model is not to let the training set know anything about the test set before we run the model. So in other words, we're not even being affected by the values from the test set in determining the mean and standard deviation we use to scale the training set.

But we do want a mean of 0 to represent the same thing in both the training set and test set. So we want to scale the test set, which I'm denoting by !in_train to get the values where in_train is not true. In other words, it's false. I'm scaling that based on the center equal to the center attribute of the training set, and the scale or standard deviation based on the scale attribute of the training set.

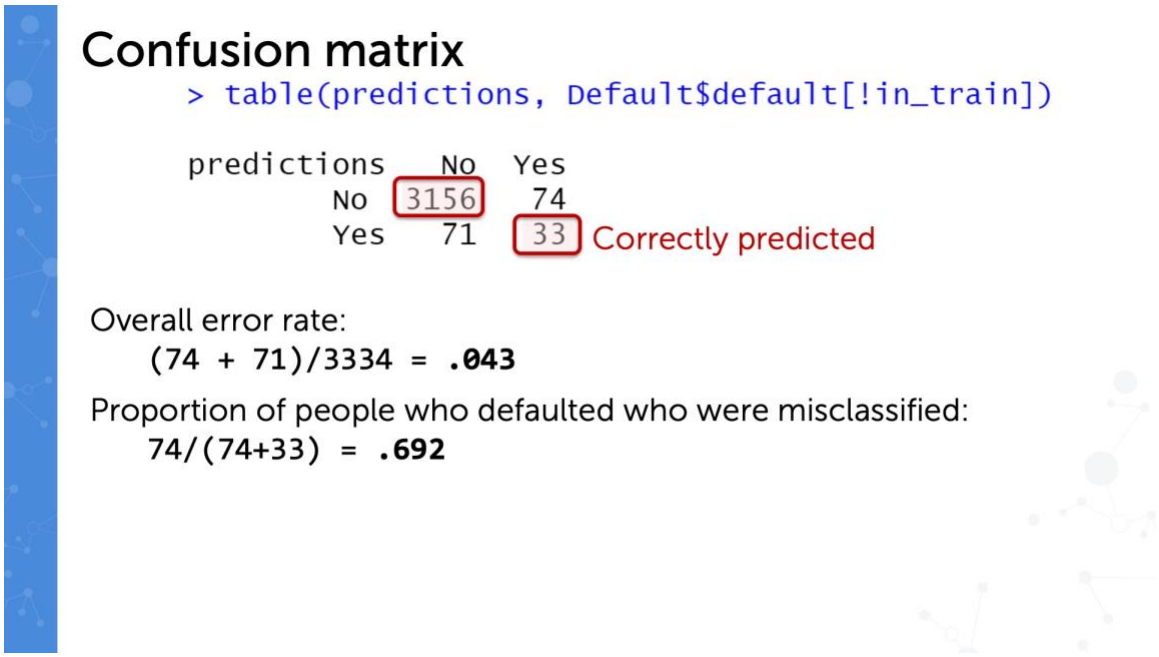
So this whole second line, or lines 3 through 4, is simply using the original mean and standard deviation from the training set from before we did the scaling in order to do the scaling of the test set. That way a value of 0 in the training set after it's standardized will mean the same thing as a value of 0 in the test set after it's standardized.

All right. Now we're ready to build the model. I've started by creating a variable called x_train by simply combining the value of student in the training set with the standardized quantitative variables from the training set using cbind to call and bind those into a single matrix. And similarly, x_test contains the value of student and the standardized quantitative variables for the test set.

Then we can use the knn function from the fnn package with the arguments train equal to the x values or predictor values of the training set, and test equal to the predictor values of the test set. cl is the classifications, or response values of the training set. So that's the first column of the Default2 data frame was whether the person defaulted on their credit card balance.

And then the final argument is k, the number of nearest neighbors we want to consider in order to classify each point from the test set. And here we'll just use k equal to 1, so one nearest neighbor. I'm storing those results in predictions, and then we'll look at the first few entries in that vector.

So here we see that predictions is a vector with levels no and yes. And it looks like the first six people in that vector all had a prediction of, no, they will not default on their credit card debt.



We can compute the error rate of our model on the validation or test set by using the table function in R to compute the confusion matrix. Here I've used predictions as the first argument in the table function. So the predicted response values will be represented in the rows of the confusion matrix. The actual observed response values are represented in the columns.

In the confusion matrix, the elements on the main diagonal are the number of observations that were correctly predicted. The elements on the off diagonal, the 74 and 71, are the elements that were misclassified. We said the prediction was no, but the truth was yes, or we said, yes, and the truth was no. That means that the overall error rate is the sum of these two values, 74 plus 71, divided by the total number of observations in the validation set, which is also the total number of observations in the confusion matrix. In this case, we get 0.043, so an overall error rate of about 4%, which is a pretty good overall error rate.

We can also compute the error rate on subsets of the data. For example, in this case, the Yes column represents people who really did default on their credit card debt. So we can compute the proportion of people who actually defaulted who were misclassified-- that is incorrectly predicted as not defaulting-- by taking the 74 divided by the total in the Yes column. That gets us 0.692, or about 69%, which is a much worse error rate.

Question 1

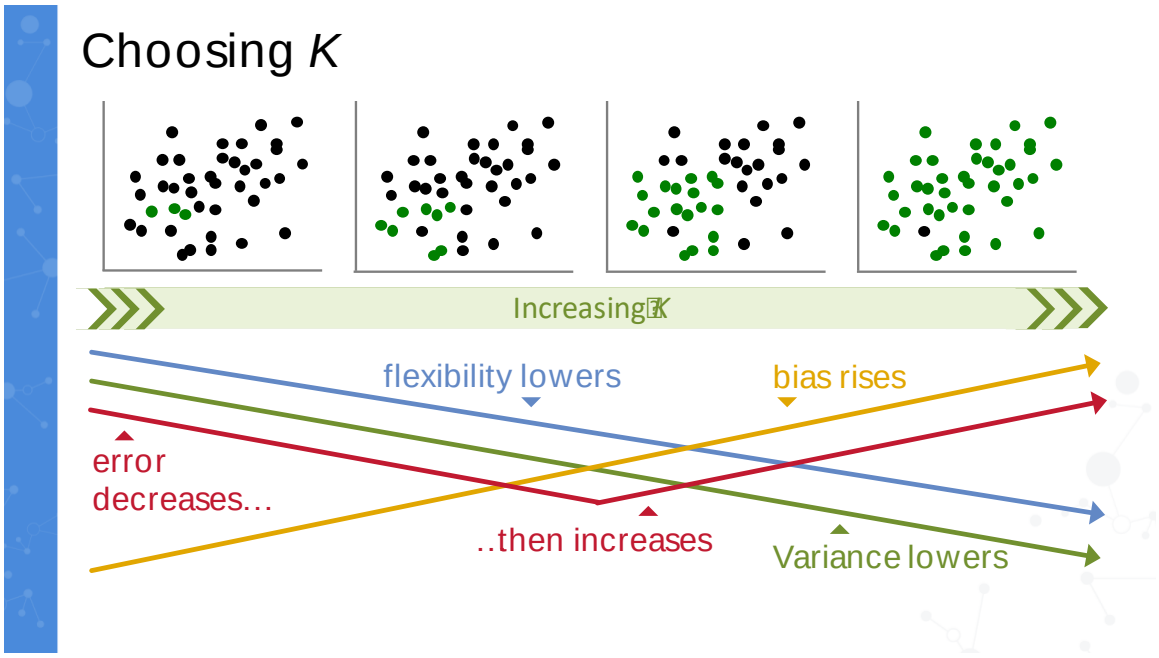
Self Assessment

In this example, what proportion of people who did not default on their credit cards were misclassified?

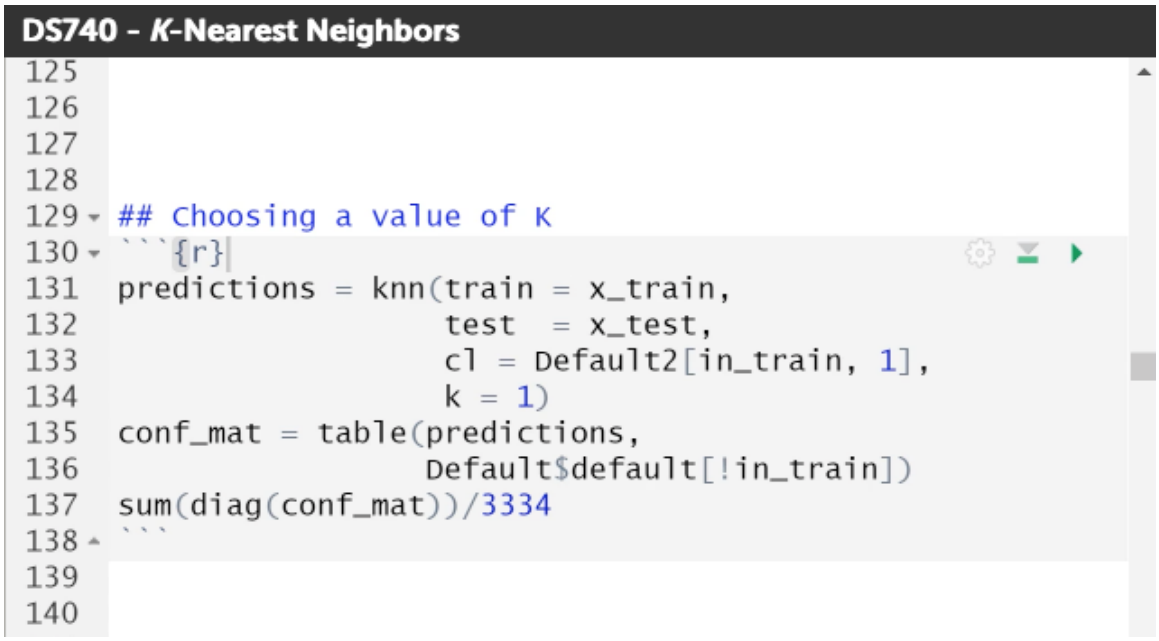
```
> table(predictions, Default$default[-train])

Predictions   No  Yes
        No  3156   71
        Yes   74   33
```

- ○ 74/3230
- ○ 71/3334
- ○ 72/3227

Submit



An important question when using K nearest-neighbors is what value of K to choose. As you increase K-- the number of neighbors used to classify each point-- the flexibility of the model decreases. That means that the variance decreases and the bias increases until, in the extreme case, when K equals the entire size of the training data set, every point is classified as belonging to the same category, whichever category is most common in the training data. Because of the trade-off between bias and variance, we expect the error to decrease and then increase as we increase K. So we expect there to be some middle point between K equals 1 and K equals the entire training data set when the overall error rate is at a minimum.



This slide represents a video/screencast in the lecture. The transcript does not substitute video content.

Let's test a variety of different values of k to find which one gives us the best model for this data. Here we have the same code as we had before running k nearest neighbors with k equal to 1. And we're computing the confusion matrix for our results. The only difference is that we're storing the confusion matrix in a variable, which I've called conf_mat.

Then on the next line, I'm taking the diagonal elements of the confusion matrix and finding their sum. So this will tell me the total number of correctly classified observations. And I'm dividing that by the total number of observations in the validation set. So overall, this line of code is giving me the accuracy on the validation set, which is 1 minus the overall error rate.

Now we want to take this whole chunk of code and iterate it over many different values of k. Let's say values of k from 1 up to 150. And because our response variable is binary-- that is the response values of default can be either yes or no, two possible values-- we'd like to avoid tie votes between the number of nearest neighbors voting for yes versus the number voting for no.

So let's go ahead and test all the odd numbers of neighbors from 1 up to 149. We can do that using the seq function to create a sequence of numbers from 1 up to 150 by increments of two. And we're storing that in a vector of k_vals. I've also created a vector here to store the accuracy values for each value of k. And that's just an empty vector with a numeric type. And its length is going to be equal to the length of the k_vals vector.

Then we want to iterate our code over all the values in k_vals. We can do that using a for loop, for ii in 1 colon the length of k_vals. So for every element in k_vals, we'll do the code once. And here I'm using a double letter, ii, for my index of which iteration I'm on. I've learned that using double letters, say ii, instead of just the letter i, for my iteration indices makes it much easier to find where in my code I've done iterations.

All right. So now I'm going to take the code that we wrote previously and Control-C, Control-V to copy it into my for loop, and I can indent it to make it more obvious that it's inside the for loop. And there are two things we want to change here. First, we want to change the value of k from 1 to the ii-th entry in k_vals. Then we want to be able to store the accuracy from each iteration. So we'll store that in the ii-th position of accuracy.

All right. And now we'll run this code. This will take a little bit of time because we do need to compute the nearest neighbors model for each value of k from 1 up to 149. All right. And now we can scroll down and plot the results. Here I'm using the gf_line function from the ggformula package.

And I want to plot the accuracy on the y-axis as a function of the value of k. And this makes sense to have accuracy on the y-axis and k on the x-axis, because it makes more sense to think about k influencing the accuracy, rather than the other way around. And then I'm just using the argument lwd equal to 2 to set the line width equal to 2 to make it a little easier to read.

So here we have our graph of the accuracy. It looks like by increasing k from 1 up to something a little bit bigger than 1, we get a big increase in accuracy. We then have a peak in accuracy around 25 for k, and again around 40. So it looks like any value of k between about 13 and about 50 would be reasonable.

If we want to find the exact maximum on this validation set, we can use + function max of accuracy to find that the maximum accuracy was 97.4%. And then we can use which.max(accuracy) to tell us which position in the k values, which value of ii, gave the maximum. And then k_vals square bracket which.max of accuracy tells us the value of k that corresponds to that.

So in this case, we had k equal to 41 nearest neighbors gave us the best accuracy. But remember that this might change somewhat if we repeated this analysis with a different random seed that gave us a different validation set.

Interpreting the best model

- KNN is easy to understand, but doesn't produce a list of coefficients to interpret
- Make predictions for a set of "example points"
- Not all of these combinations are necessarily realistic

```
balance_to_check = seq(0, 2600, by = 100)
income_to_check = seq(500, 73500, by = 1000)
student_to_check = c(0, 1)

example_data = expand.grid(student_to_check,
                           balance_to_check,
                           income_to_check)
```

Var1 <dbl>	Var2 <dbl>	Var3 <dbl>
0	0	500
1	0	500
0	100	500
1	100	500
0	200	500
1	200	500

Now that we know which model is the best, it's time to interpret it. K nearest neighbors is a modeling technique that's easy to understand. It simply looks at the K nearest neighbors of a point, and lets them vote on the predicted response value of that point. But it doesn't produce a list of coefficients that we can use to interpret the effect of each of the predictor variables in the way that, say, linear regression does.

So we need to be a little bit creative in order to understand how each of the predictor variables contributes to the prediction. One strategy I really like for this is to make predictions for a set of example points. For example, here I'm creating vectors of values of the balance, income, and student, ranging from a nice round number close to the minimum up to a nice round number close to the maximum for each of balance and income.

Then I'm using the function expand.grid to create a grid of all combinations of balance, income, and student within these vectors. So here's what the first few rows of my example data data frame looks like. We have all the possible combinations of student and balance being combined with the possible values of income.

One caveat to this strategy is that not all of these combinations are necessarily realistic. It may be that there simply aren't any students in the data set who have incomes equal to 73,500. And so we need to take our interpretations based on this with something of a grain of salt.

Notes:

```
balance_to_check = seq(0, 2600, by = 100)
income_to_check = seq(500, 73500, by = 1000)
student_to_check = c(0, 1)


example_data = expand.grid(student_to_check,
                           balance_to_check,
                           income_to_check)
```

Make the predictions

```
example_std = scale(example_data[ , 2:3],
    center = attr(quant_train_std, "scaled:center"),
    scale = attr(quant_train_std, "scaled:scale"))

# Be sure the columns are in the same
# order as the training data
x_example = cbind(example_data[ ,1],
                  example_std)
```

```
set.seed(123)
predictions = knn(train = x_train,
                  test  = x_example,
                  cl = Default2[in_train, 1],
                  k = 41)
```

Next, we need to make the predictions for our example data. So here I'm scaling the income and balance of the example data using the mean and standard deviation from the training set. So this is exactly the same thing as we did for the test set. And then I'm appending the column of the student values to make the data frame x_example.

So we'll treat this like our test set to make the predictions. So here we're making the predictions using k equal to 41, our best model. So the only thing that has changed compared to the analysis we did before is that now we have test equals x_example, our made up data frame of example points.

Notes:

```
example_std = scale(example_data[ , 2:3],
center = attr(quant_train_std, "scaled:center"),
scale = attr(quant_train_std, "scaled:scale"))

# Be sure the columns are in the same
# order as the training data
x_example = cbind(example_data[ ,1],
       example_std)

set.seed(123)
predictions = knn(train = x_train,
            test  = x_example,
            cl = Default2[in_train, 1],
            k = 41)
```

Graph the predictions

```
example_data <- example_data %>%
  mutate(pred = predictions) %>%
  rename(student = Var1,
         balance = Var2,
         income = Var3)
```

```
example_data %>%
  filter(student == 0) %>%
  gf_point(balance ~ income, color =~ pred) %>%
  gf_labs(title = "Non-students")
```

Next, we need to graph the predictions. So here I'm using the mutate function from the dplyr package to append the predictions from the k nearest neighbors model to the example data data frame. And then I'm renaming the columns to make them a little bit easier to read in the graph.

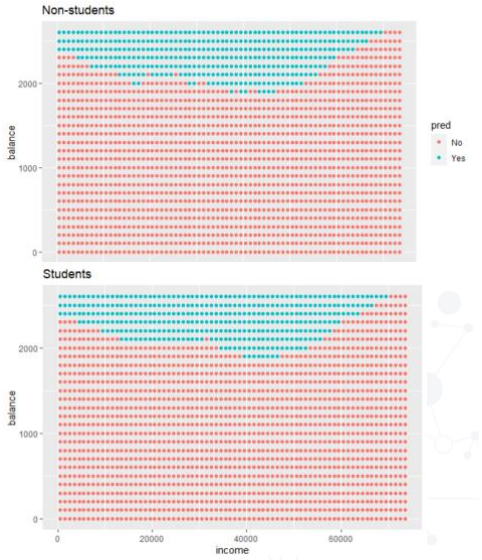
Here I'm filtering to just the rows where student is equal to 0, so just the non-students. And using the gf_point function to create a scatter plot of balance versus income and coloring the points based on the predictions. And remember that when you're using ggformula and adding a color that varies based on the value of a variable, you use the equals squiggle.

Notes:

```
example_data <- example_data %>%
  mutate(pred = predictions) %>%
  rename(student = Var1,
         balance = Var2,
         income = Var3)

example_data %>%
  filter(student == 0) %>%
  gf_point(balance ~ income, color =~ pred) %>%
  gf_labs(title = "Non-students")
```

Results in context—Balance and students

- High balance → predicted to default
  - Makes sense, because a higher balance is harder to pay off
- Very little difference between students and non-students
  - Not an important variable in this model

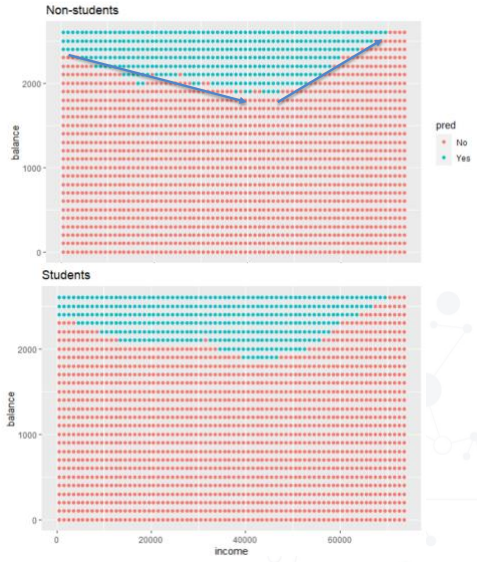


And here is what that graph looks like for non-students. And here's what it looks like when we filter to just the students. So right away looking at these graphs, we see that higher up on the graph when balance is higher, we have more blue dots, indicating that the prediction is, yes, the person will default. This makes sense because a higher balance is harder to pay off. So this agrees with our intuitive understanding of the real world context of the data.

We also see that there's very little difference between the graph for students and the graph for non-students. So it's looking like whether a person is a student or not is not a very important variable in this model.

Results in context—Income

- Income has a weak, curved relationship
  - As income increases, the balance required for a prediction of default decreases, then increases
  - Suggests gathering data on social structures, reasons for incurring debt, …

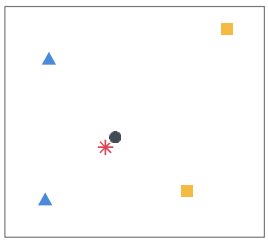


We also see that income has a weak curved relationship with the balance required for a prediction of default. As income increases-- that is as we move from left to right on the graph-- the balance that's required for a prediction of default first decreases, and then increases. This might be somewhat surprising because you might expect that as people's income increases, it should be easier to pay off a given amount of debt. So you might expect that the relationship would just be increasing.

So this raises additional questions for future models to be built or future data that we could gather. Perhaps you want to gather information on people's social structures. Perhaps people with lower incomes are more likely to have family who can help them pay off debt, or their reasons for incurring debt. Perhaps people with low incomes who incur high amounts of debt are only doing so under very specialized circumstances that make them easier to pay off.

Tie votes

FNN::knn( ) will choose a winner alphabetically
You can see the threshold for decision-making:



```
> result = knn(train.x, test.point,
+              cl = train.y, k = 5, prob = TRUE)
> result[1]
[1] blue
Levels: blue
> attr(result, "prob")
[1] 0.4
```

When you're using k nearest neighbors for classification with two categories, it's a good idea to choose k to be an odd number to avoid having tie votes. However, when you're doing classification with more than two categories in your response variable, it can be hard to avoid having tie votes. In this case, the version of knn that's in the fnn package in R will choose a winner alphabetically.

For example, if we're using k nearest neighbors with k equal to 5 in the example shown here to classify the black circle, then we have a tie between blue triangles and yellow squares. The knn function would choose blue triangles because that's the one that comes first alphabetically.

You can see the threshold that knn is using for its decision making by adding the argument prob=true. This will let you see that the probability attribute of the result was 0.4, or two out of five of the nearest neighbors voted for blue.

Tie votes are not a problem for knn for regression problems. That is predicting a quantitative response, because in that case, we're not having a tie between multiple quantitative values. Instead, all of the quantitative values from the nearest neighbors get averaged.

Notes:
train.x = data.frame(x1 = c(1,3,1,3,2), x2 = c(1,1,5,5,2))
train.y = c("blue", "yellow","blue","yellow","red")
test.point = data.frame(x1 = c(2.1), x2 = c(2.1))

```
result = knn(train.x, test.point,
   cl = train.y, k = 5, prob = TRUE)
result[1]
attr(result, "prob")
```

Number of predictor variables

More variables ➔ Fewer data points look similar

One issue with using K nearest-neighbors for classification or regression is that it tends not to perform very well when the number of predictor variables is large relative to the size of your data set. This is common among nonparametric methods. Because they don't assume a particular shape for your model, they tend to require more information in the form of more data points in order to make good predictions.

For K nearest-neighbors the issue is that the more variables there are, the fewer data points there are that look similar to or nearby each point for which we want to make a prediction. If you think about each variable as being a dimension of the data set, then three variables would correspond to three dimensional space, with each data point being a star. There can be a lot of empty space between two different stars.

Predictor variables: how many is too many?

More variables ➔ Fewer data points look similar

No simple solution, but:

- Do exploratory analysis or use cross-validation to choose predictors
- Consider another method

There is no one answer to how many predictor variables is too many for K nearest-neighbors. However, if you are interested in performing K nearest-neighbors on a data set with lots of predictor variables, it might be a good idea to do an exploratory analysis or use cross-validation to help you choose a subset of the predictor variables, or consider another method, such as logistic or linear regression.

Notes:
Here's an example that uses linear regression as an exploratory analysis to choose predictor variables for KNN:
https://www3.nd.edu/~steve/computing_with_data/17_Refining_kNN/refining_knn.html

Summary

- *KNN* is a simple but effective method for classification and regression.
- For classification problems: `knn` in `FNN` library in R.
- Use a `for()` loop to choose the value of *K* that minimizes the overall error rate on new data points.
- Graph the predicted response for a set of example points to aid interpretation.

Answer to Question 1

Self Assessment Feedback

✔ Correct!

In this example, what proportion of people who did not default on their credit cards were misclassified?

```
> table(predictions, Default$default[-train])

Predictions   No  Yes
        No  3156   71
        Yes   74   33
```

Your answer:

72/3227

Correct answer:

72/3227

Feedback:

Correct! The number of people who did not default is 71 + 3156 = 3227. Of these, 71 were misclassified, so the proportion is 71/3227 = .0220.