UNIVERSITY OF WISCONSIN
DATA SCIENCE

DS 740

Data Mining

K-Nearest Neighbors for Regression and Working With Categorical Predictors

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

Learning Objectives

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

- Use *K*-nearest neighbors for regression in R.
- Use one-hot encoding for categorical predictors.
- Decide when to treat a discrete, quantitative predictor as categorical.
- Explain why we set the random seed in R.

K-nearest Neighbors for Regression

```
> install.packages("FNN")
```

Take average value of response variable from *k* nearest points

k does ***not*** need to be odd

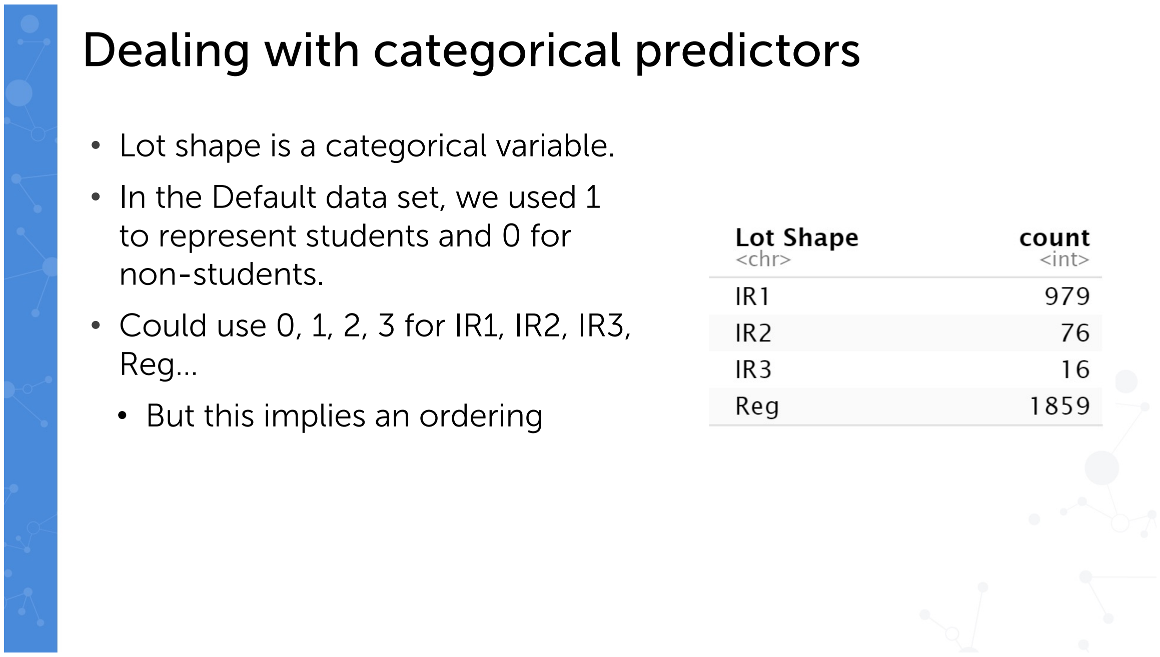
Sample problem

Predict the sale price of homes based on their lot shape and the year they were built.

Data set: `AmesHousing.csv`

Notes:

Dataset: “Ames, IA Real Estate Data," submitted by Dean De Cock, Truman State University. Dataset obtained from the Journal of Statistics Education (http://www.amstat.org/publications/jse). Accessed 26 May 2016. Used by permission of author.



Dealing with categorical predictors

- Lot shape is a categorical variable.
- In the Default data set, we used 1 to represent students and 0 for non-students.
- Could use 0, 1, 2, 3 for IR1, IR2, IR3, Reg…
  - But this implies an ordering

Lot Shape <chr>	count <int>
IR1	979
IR2	76
IR3	16
Reg	1859

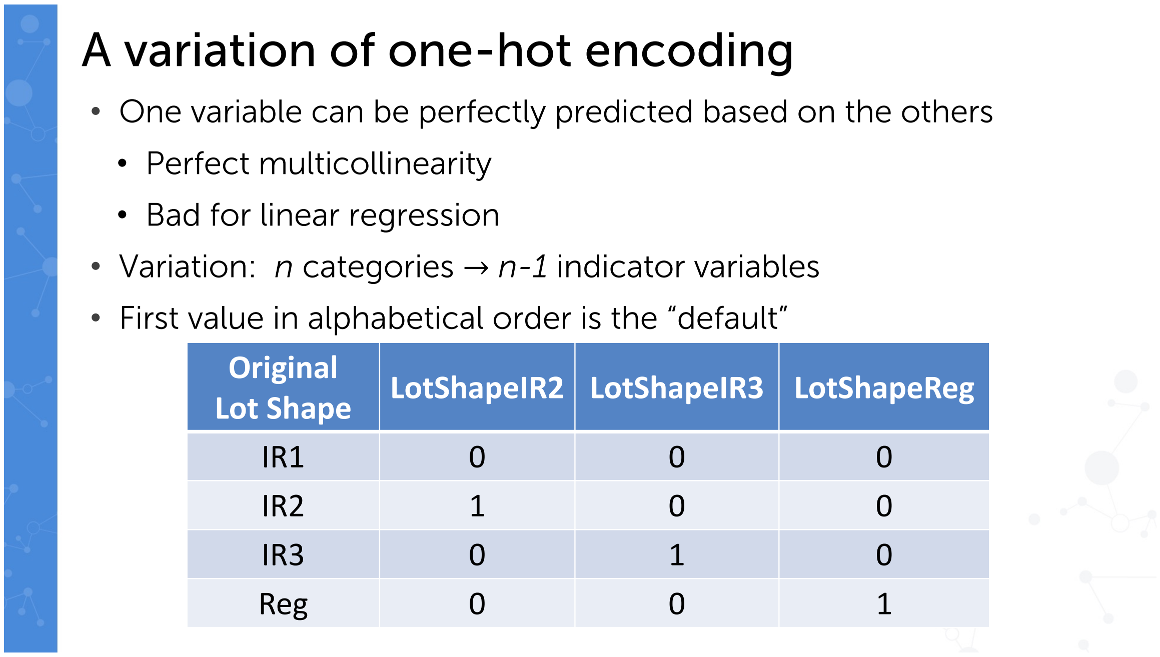
In the Ames Housing dataset, lot shape is a categorical variable. We need to converge it into numbers in order to use K-nearest neighbors. Previously, in the default dataset, we used 1 to represent students and 0 to represent non-students. So we could do something similar here, using 0 to represent IR1, 1 to represent IR2, 2 to represent IR3, and 3 to represent Reg. But that implies an ordering that IR1 is closer to IR2 than it is to IR3 or to Reg. And we may not want to assume that ordering.

One-hot encoding

- Each level of the categorical variable becomes an indicator variable
- Good for KNN, because distance between two points with different lot shapes is the same

Original Lot Shape	is_IR1	is_IR2	is_IR3	is_Reg
IR1	1	0	0	0
IR2	0	1	0	0
IR3	0	0	1	0
Reg	0	0	0	1

An alternative is one-hot encoding, in which each level of the categorical variable becomes its own 0-1 indicator variable. In this case, we have four possible values of lot shape. So we would create four indicator variables, where a 1 for, say, is IR1 would indicate that the lot shape of that house is IR1 and a 0 would indicate that it's not. This is a good strategy for K-nearest neighbors because the distance between two points with different lot shapes is the same no matter which two lot shapes we're talking about.



A variation of one-hot encoding

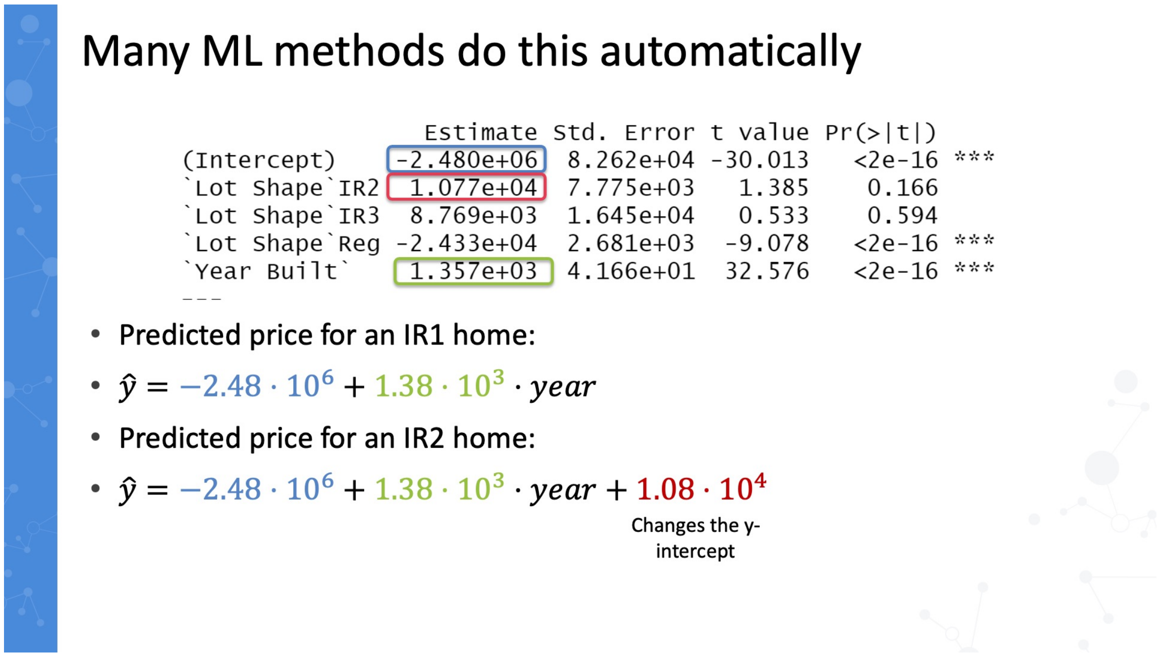
- One variable can be perfectly predicted based on the others
  - Perfect multicollinearity
  - Bad for linear regression
- Variation: *n* categories → *n-1* indicator variables
- First value in alphabetical order is the “default”

Original Lot Shape	LotShapeIR2	LotShapeIR3	LotShapeReg
IR1	0	0	0
IR2	1	0	0
IR3	0	1	0
Reg	0	0	1

A potential disadvantage of one-hot encoding is that any one predictor variable that we create in this way can be perfectly predicted based on the others. If a particular house has zeros for IR1, IR2, and IR3, then we know it must have a one for is Reg. This means that the indicator variables have perfect multicollinearity. And this creates problems for methods such as linear regression, which rely on matrices to compute the models.

So if we're using one of those machine learning methods, we instead use a variation of one-hot encoding in which a categorical predictor variable with n levels is turned into n-1 indicator variables. In this case, we have four possible values of lot shape. So we would turn that into three indicator variables. The first value of the categorical variable in alphabetical order is the default value.

So in this case, IR1 is represented by zeros in all three of the indicator variables, saying that a house with a lot shape of IR1 is neither lot shape IR2, nor lot shape IR3, nor lot shape Reg.



Many ML methods do this automatically

```
                     Estimate Std. Error t value Pr(>|t|)    
(Intercept)        -2.480e+06  8.262e+04 -30.013   <2e-16 ***
`Lot Shape`IR2      1.077e+04  7.775e+03   1.385    0.166    
`Lot Shape`IR3      8.769e+03  1.645e+04   0.533    0.594    
`Lot Shape`Reg     -2.433e+04  2.681e+03  -9.078   <2e-16 ***
`Year Built`        1.357e+03  4.166e+01  32.576   <2e-16 ***
---
```

- Predicted price for an IR1 home:
- $\hat{y} = -2.48 \cdot 10^6 + 1.38 \cdot 10^3 \cdot year$
- Predicted price for an IR2 home:
- $\hat{y} = -2.48 \cdot 10^6 + 1.38 \cdot 10^3 \cdot year + 1.08 \cdot 10^4$

Changes the y-intercept

Many machine learning methods will do this variation of one-hot encoding for us automatically. For example, if we do linear regression to predict a house's sale price based on its year built and its lot shape, R will produce output that looks like this. This tells us that if we want to predict the price for a home on an IR1 lot, we would use the formula shown here, taking the y-intercept and the slope times the year it was built.

But if we want to predict the price for a home on an IR2 lot, we would have an additional term of the effect size of the IR2 lot shape. This has the effect of changing the y-intercept of the model for IR2 homes.

One-hot encoding Lot Shape

- For KNN, use 4 indicator variables for 4 categories
  - 3 indicator variables:

Point 1	Point 2	Distance
Reg, 2000	IR1, 2000	1
Reg, 2000	IR2, 2000	$\sqrt{2}$

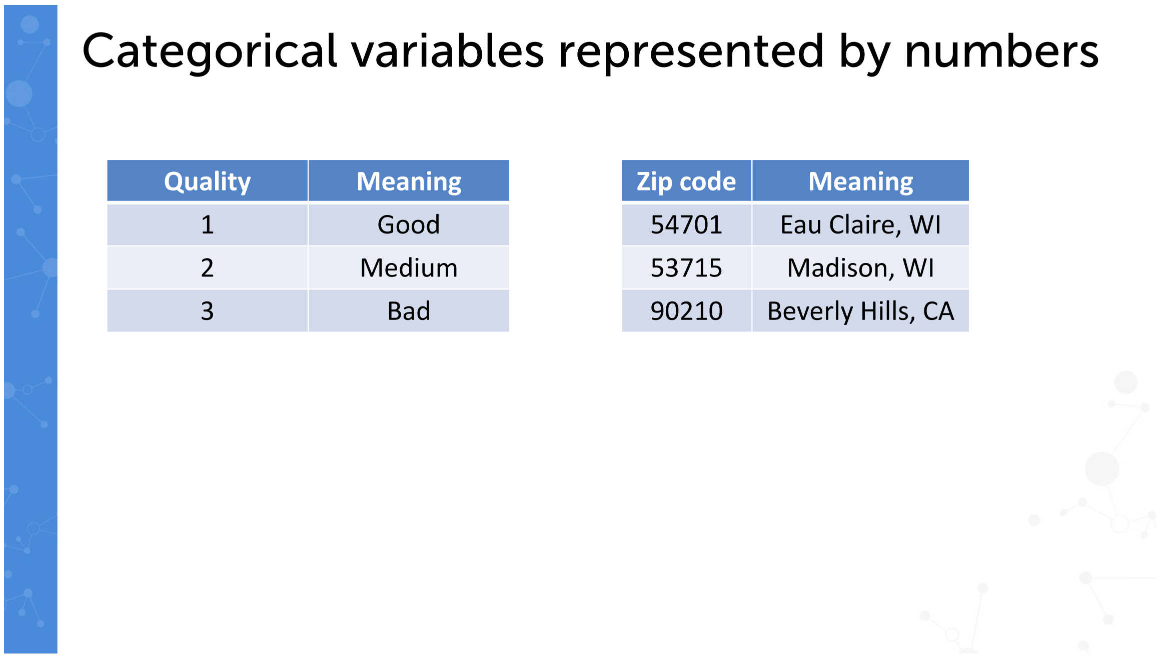
```r
ames <- ames %>%
  mutate(is_IR1 = ifelse(`Lot Shape` == "IR1", 1, 0),
         is_IR2 = ifelse(`Lot Shape` == "IR2", 1, 0),
         is_IR3 = ifelse(`Lot Shape` == "IR3", 1, 0),
         is_Reg = ifelse(`Lot Shape` == "Reg", 1, 0))
```

In this case, because we're doing KNN, we want to use four indicator variables to represent the four categories of lot shape. If we only had three indicator variables, then we would end up with different distances between pairs of points with different shapes simply depending on whether those lot shapes included the default value of IR1 or not. So to do our one-hot encoding, we can do this in a variety of ways.

I chose to use mutate from the dplyr package along with the ifelse function.

Notes:

```
ames <- ames %>%
 mutate(is_IR1 = ifelse(`Lot Shape` == "IR1", 1, 0),
    is_IR2 = ifelse(`Lot Shape` == "IR2", 1, 0),
    is_IR3 = ifelse(`Lot Shape` == "IR3", 1, 0),
    is_Reg = ifelse(`Lot Shape` == "Reg", 1, 0))
```



Categorical variables represented by numbers

Quality	Meaning
1	Good
2	Medium
3	Bad

Zip code	Meaning
54701	Eau Claire, WI
53715	Madison, WI
90210	Beverly Hills, CA

Another thing to watch out for as you're preparing your dataset for analysis is categorical variables represented by numbers. For example, maybe your dataset has a variable called quality, which ranges from 1, representing good, to 3, representing bad. Even though this variable looks like a number, it's actually categorical because numerical comparisons don't make sense here.

It doesn't make sense to say that medium is twice as much as good or the distance between good and bad is twice as much as the distance between good and medium. This is what's known as an ordinal categorical variable. So we do have an ordering to the categories. In this case, you might want to leave this represented by numbers for the purposes of K-nearest neighbors. But if you wanted to remove that sense of ordering and simply treat the qualities as being three different possibilities, then you might want to one-hot encode this. You could try it both ways and see which way gave you the better predictions.

Another example would be a data set which includes people's zip codes as one of the variables. Again, even though zip code is something that looks numeric, it's actually categorical because it doesn't make any sense to say that the distance between my zip code and your zip code is 200. This is a nominal categorical variable, one where ordering doesn't really make any sense. So we would definitely want to one-hot encode people's zip codes.

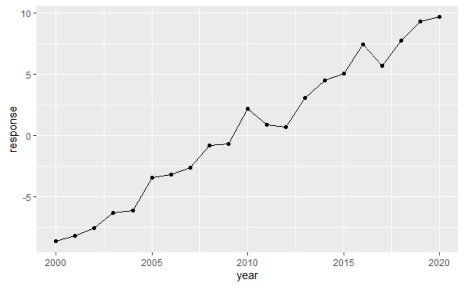
Should Year be categorical?

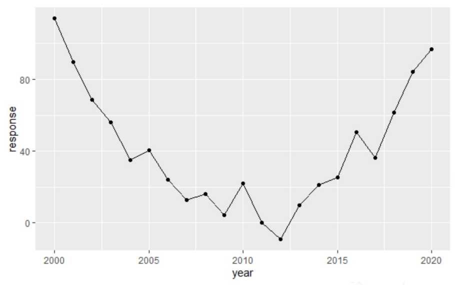
- Discrete, quantitative

Can’t have a year 2019.5317

Numerical comparisons make sense

- Treating as quantitative will give better predictions if 2020 is more similar to 2015 than it is to 2000



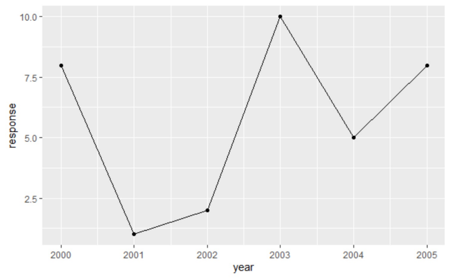


A common question that people have is, should year be treated as categorical? Year is a discrete, quantitative variable. It's quantitative because numerical comparisons make sense. It makes sense to say that the year 2020 is twice as far from the year 2000 as it was from the year 2010. But it's discrete because it only comes in certain specified values-- in this case, whole numbers.

You can't have a year 2019.5317. When you have a predictor variable that's discrete and quantitative, treating it as quantitative will tend to give you better predictions if the numerical value of the variable is informative about the similarity of the response variable-- for example, if 2020 is more similar to 2015 in terms of the sale prices of homes than it is to the year 2000. So if you have some sort of gentle trend in the sale prices of homes as year progresses, then you want to treat year as quantitative.

Treating Year as categorical

- Treating as categorical will give better predictions if the years are all equally dissimilar

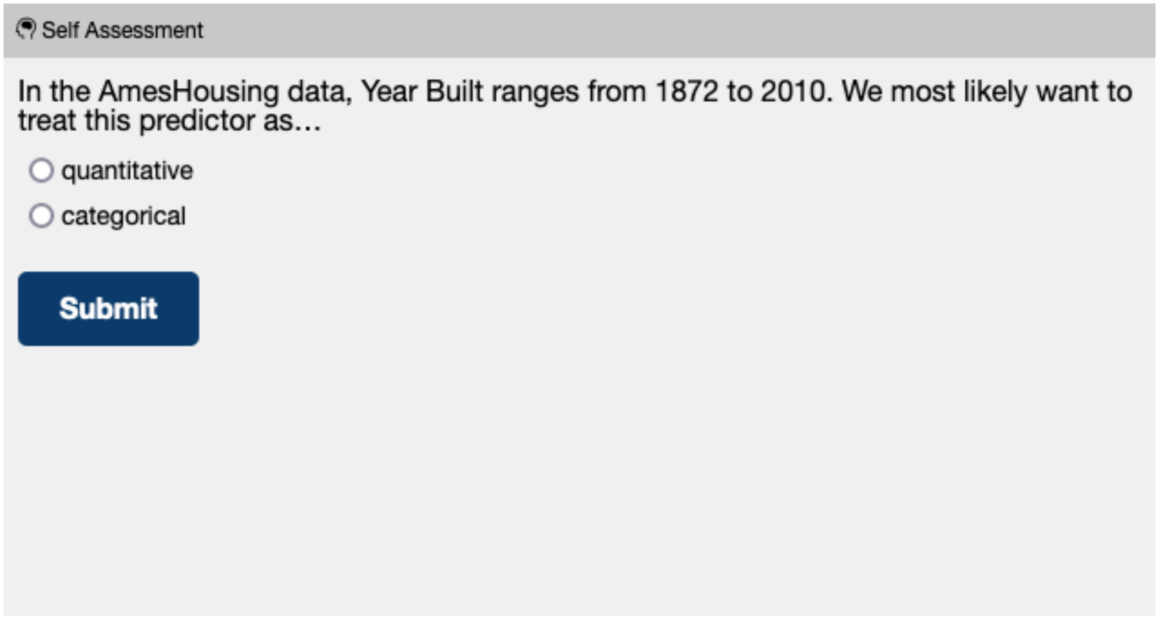


On the other hand, treating year as categorical will tend to give better predictions if the years are all equally dissimilar. In other words, if the fact that 2020 and 2019 are close together and 2000 is far different from both of them-- doesn't give us any information that we can use to predict house prices. If we could just as easily replace the numbers of the years by the shapes, year circle, year square, and year triangle, then treating year as categorical makes sense.

If you plan to do this, the number of distinct values of your predictor variable should be small compared to the number of data points. That's because with one-hot encoding you'll be creating a new indicator variable for every level of that predictor variable. And you don't want your total number of variables to get too large compared to the number of data points in your dataset. Another thing to keep in mind before you treat a discrete quantitative variable as categorical is that for some machine learning methods, it can be impossible to make predictions for unseen values of a categorical variable.

This isn't a problem with K-nearest neighbors. But it is a problem with linear regression. So if you're doing linear regression to predict house prices and using year as categorical, if your training dataset only contains data from the year 2000 to 2020, then you're not going to be able to use it to predict the sale price of a house in 2021 or 2025 because you won't have been able to use your training set to estimate the effect size of that new year.

Self-assessment Question 1

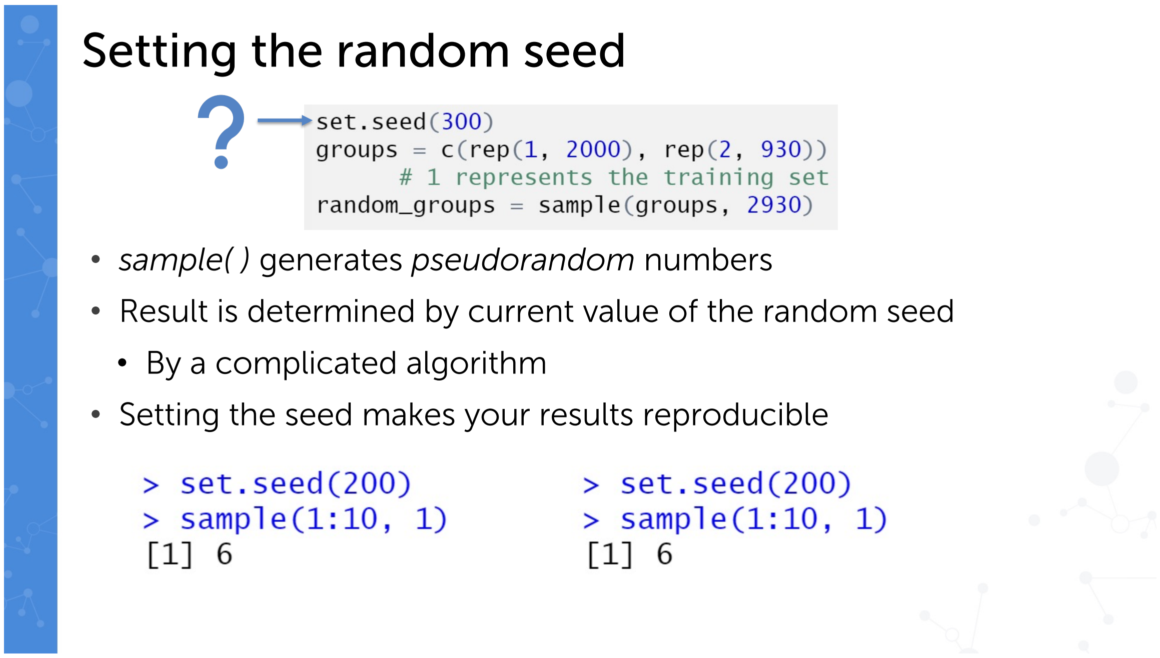

Self Assessment

In the AmesHousing data, Year Built ranges from 1872 to 2010. We most likely want to treat this predictor as…

○ quantitative

○ categorical

Answer is at the end of the transcript.



Setting the random seed

?

```
set.seed(300)
groups = c(rep(1, 2000), rep(2, 930))
       # 1 represents the training set
random_groups = sample(groups, 2930)
```

- *sample( )* generates *pseudorandom* numbers
- Result is determined by current value of the random seed
  - By a complicated algorithm
- Setting the seed makes your results reproducible

```
> set.seed(200)          > set.seed(200)
> sample(1:10, 1)        > sample(1:10, 1)
[1] 6                    [1] 6
```

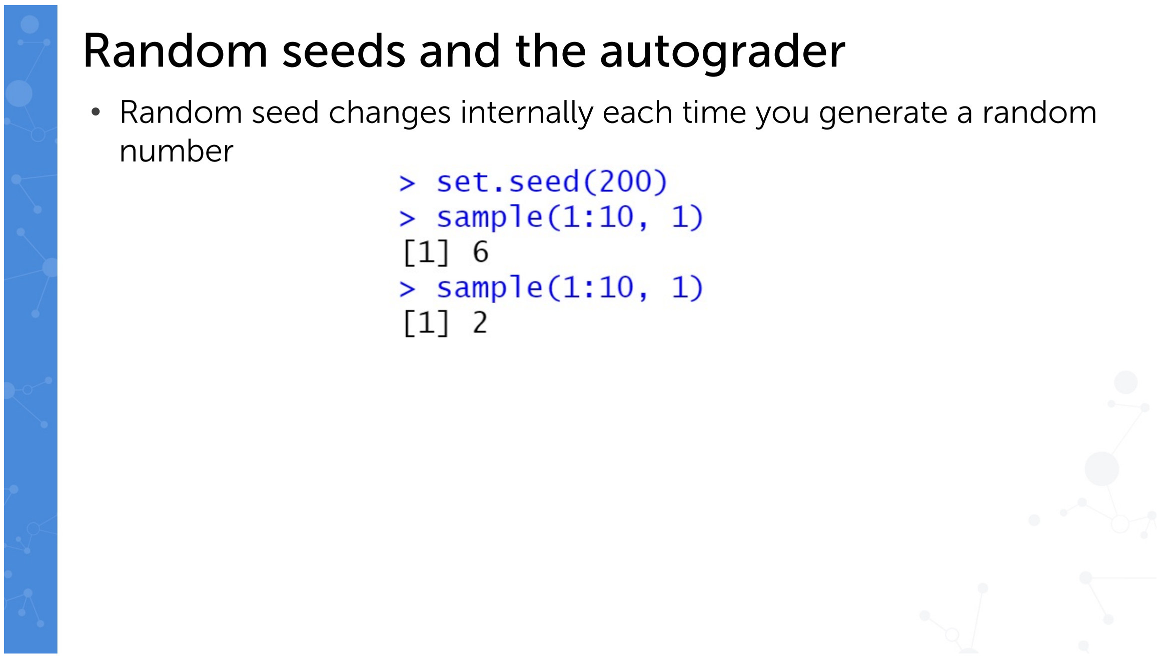
Now that we've one-hot encoded our categorical variable, it's time to randomly divide the dataset into a training set and validation set. But why do we use the function set.seed before we do this. We do this because the sample function isn't really dividing up our dataset randomly. It's not rolling a die or flipping a coin behind the scenes. Instead, the sample function generates pseudo-random numbers. These are numbers that are determined by a complicated algorithm so they look random to us.

But they're actually determined by the current value of the random seed. This is great because it means that we can make our results reproducible by setting the seed. For example, if you set the seed equal to 200 and then do an analysis today, you can come back tomorrow and say, gee, I'd really like to take another look at that analysis with the same training and validation sets and maybe make some new graphs. You can do that by setting the seed equal to 200 and repeating the same code that you did before. You'll get the same set of random numbers in the same order.

Notes:
set.seed(300)
groups = c(rep(1, 2000), rep(2, 930))
    # 1 represents the training set
random_groups = sample(groups, 2930)

set.seed(200)
sample(1:10, 1)

```
set.seed(200)
sample(1:10, 1)
```



Random seeds and the autograder

- Random seed changes internally each time you generate a random number

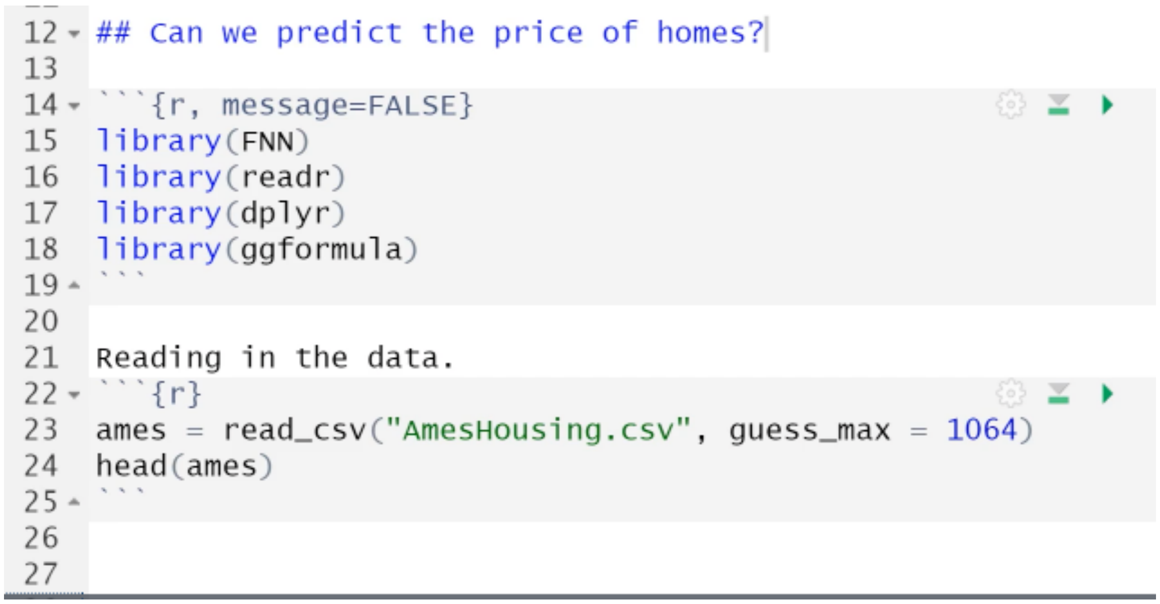
```
> set.seed(200)
> sample(1:10, 1)
[1] 6
> sample(1:10, 1)
[1] 2
```

In addition to changing when you use the function set.seed, the random seed also changes internally each time you generate a random number. This means that you can write code like this, where you set the seed and then call the sample function twice to get two different random numbers. In this course, we'll often tell you to set the random seed to a specific value so that you get the same answers on your homework as we got.

But this feature of the random seed changing when you generate random numbers means that your answers might get off of what the autograder is expecting if you reset the seed-- back to 200-- at a place when we're not expecting you to or if you generate extra random numbers, for example, by running a code cell multiple times in the process of debugging it. So for this class, you should set the seed where the homework or web work tells you to and no other places.

When you have your code working the way you want it, it's a good idea to clear your R environment and then run your code from top to bottom or knit the RMD file. This make sure that you didn't run any code sections twice to generate additional random numbers and throw off your random seed.

Screencast



~~~
## Can we predict the price of homes?

```{r, message=FALSE}
library(FNN)
library(readr)
library(dplyr)
library(ggformula)
~~~

Reading in the data.
```{r}
ames = read_csv("AmesHousing.csv", guess_max = 1064)
head(ames)
```
```

Let's use k nearest neighbors to predict the price of homes in Ames, Iowa, using their lot shape and the year they were built. I've already loaded in my packages, the FNN package, which contains the function knn.reg, as well as the readr package to read in the data, dplyr for some data manipulation, and ggformula for some graphing for interpretation later.

So next I want to read in the data. I'll do that using the read_csv function in the readr package. And my data set is already saved in the same folder where this RMD file is saved. So I don't need to include the whole path to the file, just the name of the file. I'm also including an additional argument, guess_max equals 1,064. I did this because when I tried using just read_csv CSV of AmesHousing, I got some warning messages saying that r wasn't able to interpret several of the rows in the data set, starting with row number 1,064.

This happened because it turned out that none of the first thousand rows of the data had pools. So they all had NA for the quality of the pool variable. So that meant that read_csv was guessing the incorrect data type for that variable. So by using guess_max equals 1,064, I'm telling read_csv to wait until it reads that row of the data to guess the data type for each of the variables.

All right, we can check what's in the Lot Shape variable and verify that it has four possible values as a categorical variable. The IR2 and IR3 values are fairly infrequent. So we expect to get better predictions for houses with Lot Shape equal to Reg or IR1.
```

So now we can One-hot encode the lot shape using four indicator variables. And then create the training and validation sets. Using the dim function, we see that this data set contains 2,930 rows. And we typically want to have the training set be about 2/3 of the data. So I'll create a training set with 2,000 rows and let the other 930 rows be in my validation set. So this code is exactly the same as what we did for the default data set, just with a different number of data points in the training and validation sets.

Next, we'll create the x_train and x_test data frames containing the predictor variables, the x variables, for each of the training and validation sets. And I'm using the filter function to select the rows that are in the training set and exclamation point representing not, so not in the training set for the test set.

Now I want to scale the Year Built variable. In this case, we only have one quantitative variable. So I did consider simply not scaling it. But I did want a value of 1, difference in year built, to be similar to the difference in 1 versus a 0 for each of the last shapes. So I am going to scale this so that a value of 1 in the year built represents 1 standard deviation, rather than one year. And just like we did for the default data set, we're scaling the test set according to the mean and standard deviation of the training set from before we did the scaling.

Next, we'll build the model. So this looks very similar to what we did for knn for classification. The differences are that we're using the function knn.reg instead of just knn. And the response values, instead of being cl for the classifications are called y. Another small change is that the predictions output contains some additional information, not just the predictions. So to get the actual numerical predictions of the sale prices, we want to look at predictions, whatever variable we call that, $pred, so the pred component of the object that we got out of knn.reg.

All right, to look at the accuracy of our model, we can compute the mean squared error. So the mean squared error is computed in exactly the way it sounds. We first compute the error of each data point by taking the predictions and subtracting the observed values of the sale prices from the test set, so remember the !in_train. So that's our error. Then we square it, and finally, we take the mean. One thing to be aware of is make sure that you're doing the squaring inside of taking the mean.

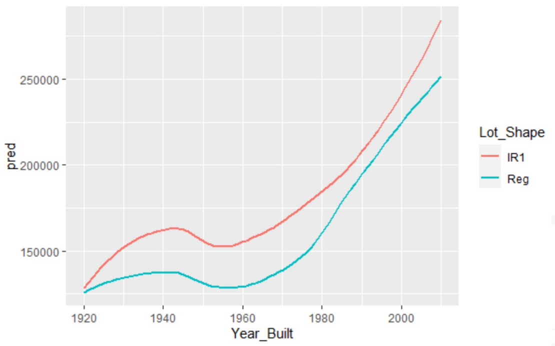
So here we get quite a large mean squared error. This isn't too surprising because sale price has a large standard deviation. So it's not surprising that it's difficult to predict. And we're only using two predictor variables, the lot shape and the year it was built. So it's not surprising that it would be difficult to predict. However, the mean squared error is difficult to put into context because the units of this are dollars squared. So what really counts as a large value of dollars squared isn't something that's really intuitive.

Two things that we can do to make this easier to interpret. Our first, if we were testing out multiple models, for example, if we had a for loop to iterate over different values of k, then we would be interested in the model that had the lowest mean squared error. So that could help put it into context. The other thing we could do is, instead of just computing the mean squared error, we can compute the mean absolute error or MAE. So this is very similar to the mean squared error. We start out by computing the error, the difference between the predictions, and the observed values of the response variable. But then instead of squaring them, we take the absolute value, so simply making them all positive. And then we take the mean.

So this result now has units of dollars, which is much easier to interpret. So we can see that, on average, our predictions are off by plus or minus about $56,000. We can further put this into context by comparing our error to the values in the data. So here we had a median sale price of $160,000 and a maximum sale price of $755,000. So we can compare the MAE to the median or the maximum and see that our error is about 35% of the median home price or about 7% of the maximum home price. So we can see that this is still a fairly large error. That we could probably do better by testing out different models, either by including more of the predictor variables or by changing our value of K.

## Interpreting the model

- Make predictions for a grid of example points
- 1 quantitative + 1 categorical predictor → use a line graph with colors
- Newer homes cost more, but only since about 1960
- Interestingly, irregular lots cost more



To interpret our model, we can make predictions for a grid of example points like we did for the default data set. In this case, we only have one quantitative and one categorical predictor. So we can use a line graph to plot the predicted house prices as a function of the year built with different colors to represent the lot shape. In this case, the line graph turned out to be pretty noisy. So I'm using a loess smoothing spline instead.

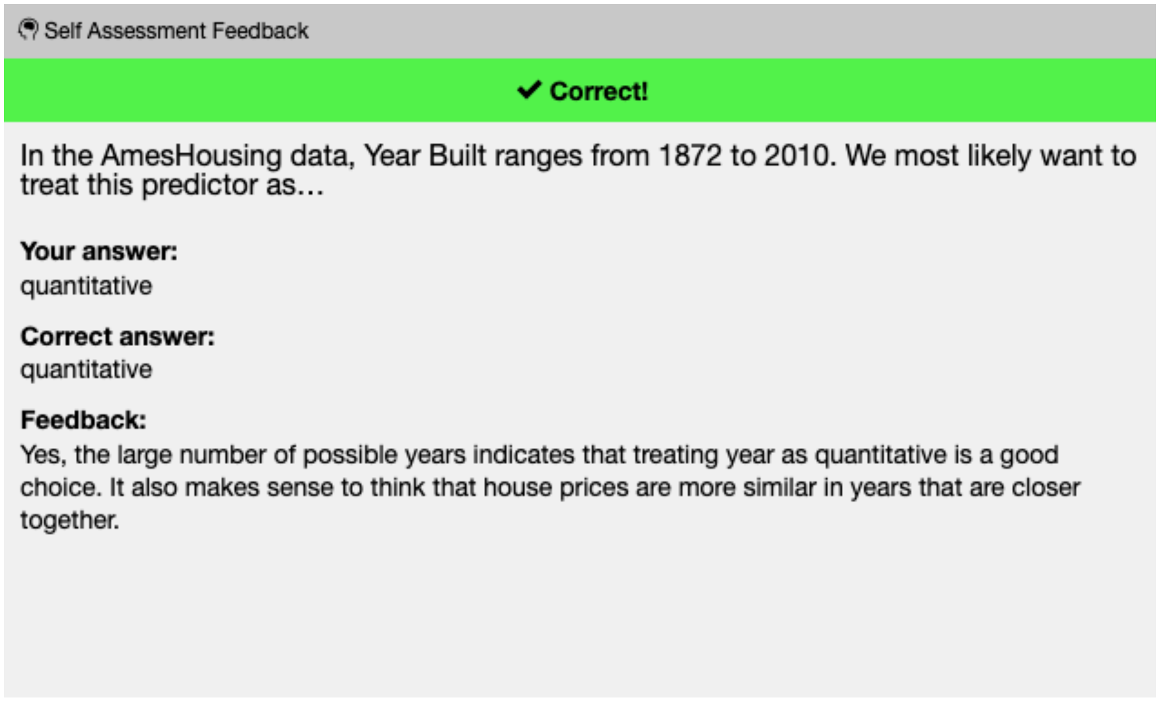
From this graph, we can see that newer homes do tend to cost more, which agrees with my intuitive expectation. But this is only true since about 1960. Before that, it seems like the house prices were relatively flat, perhaps even with a peak right around World War II. So this could mean that once we get past 1960, people simply think of the house as old and don't put any more of a discount on how much it should cost.

Interestingly, we see that irregular shaped lots of lot shape IR1 tend to cost more than the regular shaped lots. This is worth investigating. Perhaps the irregular shaped lots tend to be larger so they end up being more desirable.

## Summary

- *KNN* for regression problems: `knn.reg` in `FNN` library in R
- Standardize predictor variables, same as for classification problems.
- Use MSE or MAE to measure error, unlike for classification problems.
- Treating a discrete, quantitative predictor (like Year) as categorical can improve predictions if different values of the predictor are all equally dissimilar.
  - For some ML methods, can be impossible to make predictions for unseen values.

Answer to Self-Assessment 1



In the AmesHousing data, Year Built ranges from 1872 to 2010. We most likely want to treat this predictor as…

**Your answer:**
quantitative

**Correct answer:**
quantitative

**Feedback:**
Yes, the large number of possible years indicates that treating year as quantitative is a good choice. It also makes sense to think that house prices are more similar in years that are closer together.