Data Cleaning

Lecture 13

Connor Dowd

May 11th, 2021

Today's Class

1. Review
   - Ensembles
   - Boosting
2. Data Cleaning Basics
3. Dates
4. Wide vs long data
5. Merges/Joins
6. `NA`s, `NaN`, etc.
7. Predictions 2
   - CI vs PI
   - Sundays
   - My forecast
   - Ensemble PIs

Review

Weighted – Example

Suppose I have two models: - M_1 always predicts $\hat{y}_1 = M_1(X) = \bar{y}$ - M_2 uses p variables and linear regression to predict $\hat{y}_2 = M_2(X) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + ... + \hat{\beta}_p x_p$

These are not equivalently good. M_2 is *likely* better (not always – see overfitting again).

We could plug them both into our naive average, and get a prediction.

Inverse Variance Weights

The basic idea is that a single measure of the prediction errors of each model can help us generate the best weights.

The models with the smallest average errors should get the most weight, and the models with the largest average errors should get the least weight.

If you are minimizing MSE, this will look like weights which are proportional to the “precision" (aka the inverse of the variance).

Procedure

1. Estimate K models $M_1, ..., M_K$
2. Use cross-validation (k-fold) to estimate OOS errors for each model.
3. Calculate average OOS error MSE_k for each model.
4. Find weights $\mathbf{w} \propto MSE_k$ such that $\sum w_k = 1$
5. Make your predictions using those weights

This will be reasonably quick. For K models and m folds in the cross validation, you'll only need to fit Km models. The weights pop out analytically.

Boosting

Boosting – Ensembles 2

Boosting changes things up slightly. It says “we want to do a better job of predicting when we are wrong”.

Thus far, we have been giving each tree slightly different data, and averaging across each tree.

But each tree still was optimizing the same thing – mean prediction error[1] :

$$\frac{1}{n}\sum_{i=1}^{n} l(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n} 1(\hat{y} \neq y)$$

[1] This could be MSE, etc depending on the model.

Boosting Insight 1

Some observations are harder to predict.

We want to build a model that predicts everything well.

We can give *more weight* to observations that are hard to predict.

$$\frac{1}{n}\sum_{i=1}^{n} 1(\hat{y} \neq y) = \sum_{i=1}^{n} \frac{1}{n} 1(\hat{y} \neq y)$$

The weights ideally sum to 1, but we need not have each weight be $\frac{1}{n}$. Feel familiar?

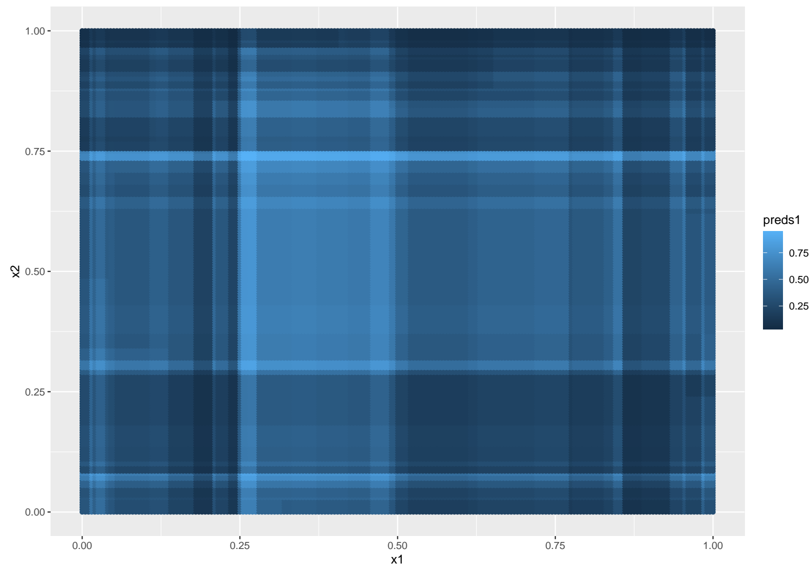
Boosting Algorithm

1. Start with a class of models F (e.g. tree, glm, etc). And weights $w_i = \frac{1}{n}$
2. Fit a model M_k in that class using those weights.
3. Find the prediction error for each observation from that model.
4. Increase the weights for observations where predictions were most wrong, decrease the weights for observations where predictions were most correct.
5. Repeat steps 2-4 until you've built K models.

Model M_K will be using weights based on how poorly each previous model did at predicting each observation.

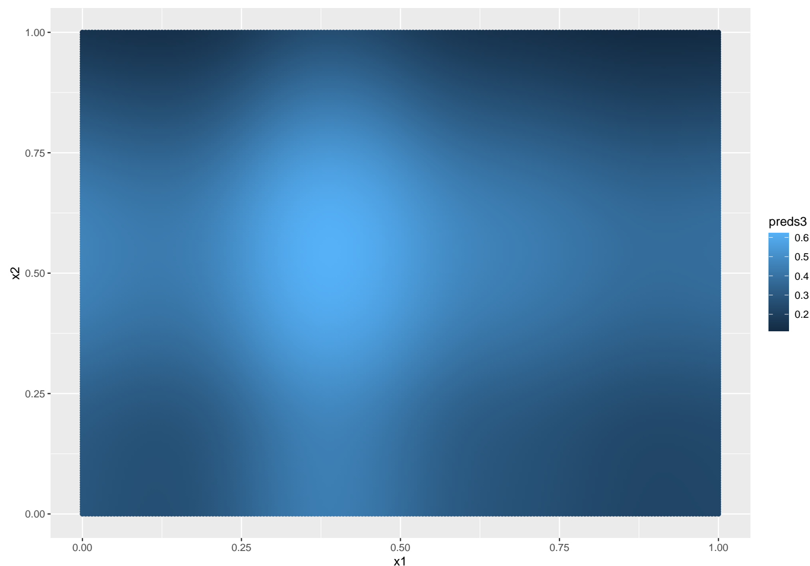
Boosting Trees

```r
ggplot(grid,aes(x=x1,y=x2,col=preds1))+geom_point()
```



Boosting GAM

```
ggplot(grid,aes(x=x1,y=x2,col=preds3))+geom_point()
```



Data Cleaning

Data Cleaning 101

Rules:

1. Keep looking at the data

Data Cleaning 101

Rules:

1. Keep looking at the data
2. Make small changes.

Data Cleaning 101

Rules:

1. Keep looking at the data
2. Make small changes.
3. Test your changes before you overwrite variables.

Data Cleaning 101

Rules:

1. Keep looking at the data
2. Make small changes.
3. Test your changes before you overwrite variables.
4. Don't overwrite actual files unless you're certain.

Data Cleaning 101

Rules:

1. Keep looking at the data
2. Make small changes.
3. Test your changes before you overwrite variables.
4. Don't overwrite actual files unless you're certain.
5. Don't throw away potentially useful data.

Basics

What is data cleaning?

(A) We get raw data – which consists of all kinds of nonsense and structure

To get from $A \rightarrow B$$, we may need to remove some structure, build our own structure, deal with missing values, get rid of irrelevant data, add in relevant variables, deal with sampling design, and more. This is cleaning.

Basics

What is data cleaning?

(A) We get raw data – which consists of all kinds of nonsense and structure

(B) We want to have a matrix x to model an outcome y with.

To get from $A \rightarrow B$$, we may need to remove some structure, build our own structure, deal with missing values, get rid of irrelevant data, add in relevant variables, deal with sampling design, and more. This is cleaning.

Formats

CSVs aren't everything

Data you want comes in many forms.E.g. See SCF

- flat: CSV, arrow, parquet, etc
- compressed: .zip, .RData, etc
- proprietary: .xlsx, STATA, SAS, etc
- Internet-standards (scraped): html, json, etc

Converting between these is non-trivial and critical.

- But there are mostly packages for this – so a google search will typically solve your problems, and I'll assume you can get to a data.frame or two.

Other Formats

Most other non-trivial data formats are an entire courses worth of material. E.g.How to deal with:

```
- paragraphs of text
- Images
```

At a high level. My advice for those situations *at the moment* is to train a model (or borrow a pre-trained one) that can deal with those things.

Images

For example, Images are really just a matrix with a color at each point. Detecting a face or something in that matrix is well explored at this point, and well beyond what we can cover here.

- We can use other people's plug-and-play models for face detection as inputs.
  - E.g. run that model to determine "is there a face", then use "face-presence" as a variable.

Text

Text models are similar. Well explored, mostly beyond the time we have in this course. But you can take someone else's sentiment analysis, or subject analysis as an input to your models very easily.

Models are Legos

I want you to see models as building blocks. We have some goal, we can combine 10 models to build inputs that we throw into an 11th model to help us with our goals.

1. This is a multi-step thing.
2. Most of the time, good use of data involves doing a good job constructing other pieces from it.

Dates

Seconds since 1970

Most dates are stored as an integer value representing the number of seconds since 12:00:00 AM on Jan 1, 1970.

```
time = Sys.time()
time
```

```
## [1] "2021-05-11 12:39:32 CDT"
```

```
as.integer(time)
```

```
## [1] 1620754772
```

Days since 1970

```r
date = Sys.Date()
date
```

```
## [1] "2021-05-11"
```

```r
as.integer(date)
```

```
## [1] 18758
```

This is the number of days since Jan 1, 1970.

Advantages

The great thing about setting up variables like this, is that we can easily determine the gap between two dates.

```
newtime = Sys.time()
newtime-time
```

```
## Time difference of 0.009299994 secs
```

That is how long it took to run the code in the middle there.

Beyond this – it is also a specific, definitely identified time.

Problems

I don't care about the number of seconds or days since 1970.

For basically this reason, dates are notorious to work with. They are usally stored in a format which is focused on being good at things that we don't care about.

Subsecond accuracy also going to require other things...

Other Formats

You may be used to this format:

```
date = "2021-05-11 8:05:32 AM"
is.character(date)
```

```
## [1] TRUE
```

This is a character. I can understand what it is saying, but it can't be added and subtracted.

Moreover, it is underidentified.

- Is this date in November or May? Need more info.
- Is this time during Business hours in NYC?
    - Need timezone info.
    - Including Daylight Savings

Dates in R

Tidyverse has the only good package for working with dates I've encountered.

```
library(lubridate)
```

Package comes with a host of useful functions for dealing with dates. E.g. I can convert characters to "Dates".

```
date = "2021-05-11"
ymd(date) #This doesn't look different, but it is.
```

```
## [1] "2021-05-11"
```

Example

```r
as.integer(ymd(date))
```

```
## [1] 18758
```

```r
as.integer(date)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA
```

Example: CDC Data

Posted Data from my prediction

```
cases = read_csv("https://codowd.com/bigdata/predictions/co
```

```
##
## -- Column specification ----------------------------------
## cols(
##   State = col_character(),
##   Date = col_character(),
##   `New Cases` = col_double(),
##   `7-Day Moving Avg` = col_double()
## )
```

```
head(cases$Date)
```

```
## [1] "May 6 2021" "May 5 2021" "May 4 2021" "May 3 2021"
## [6] "May 1 2021"
```

Example

```r
cases = cases %>%
  mutate(Date=mdy(Date))
head(cases$Date)
```

```
## [1] "2021-05-06" "2021-05-05" "2021-05-04" "2021-05-03"
## [6] "2021-05-01"
```

```r
head(as.integer(cases$Date))
```

```
## [1] 18753 18752 18751 18750 18749 18748
```

Example

I care about the day of week – going to focus on Sundays

```
cases = cases %>%
  mutate(dow = wday(Date))
head(cases$dow)
```

```
## [1] 5 4 3 2 1 7
```

These are “Thurs”,“Weds”,“Tues”,“Mond”,“Sun”,“Sat” respectively.

Example Etc

Other functions for year, month, etc.

```
cases = cases %>% mutate(months = month(Date))
head(cases$months)
```

```
## [1] 5 5 5 5 5 5
```

Fixed Effects

We haven't discussed Fixed Effects.

But if you wanted Month and Year fixed effects, using the month and year functions to create columns, converting those columns to factors and using those factors as inputs would do it.

```
cases = cases %>% mutate(years = year(Date))
head(cases$years)
```

```
## [1] 2021 2021 2021 2021 2021 2021
```

Fixed Effects

But if you wanted Month by Year (AKA Month x Year) fixed effects, you need to combine both.

```
cases = cases %>% mutate(m_y = paste0(months,"-",years))
sample(cases$m_y,5)
```

```
## [1] "7-2020"  "10-2020" "8-2020"  "3-2020"  "2-2021"
```

That will give us a character again. Which can become a factor, etc.

Wide vs Long

Panel Data

A reasonably common data structure is known as 'panel data'.

In this setting, we have a number of units (e.g. States) which we observe. We also have a number of times at which we observe those units. There are some number of variables we observe.

There are many ways you could store this information.

Wide

Wide makes each unit a column and each time a row. Then it stores our variable of interest in the cells.

```
cigs = read_csv("cigarettes.csv")
cigs
```

```
## # A tibble: 31 x 40
##     year    AL    AR    CA    CO    CT    DE    GA    IA
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1  1970  89.8  100.   123   125.  120   155   110.  108.
##  2  1971  95.4  104.   121   126.  118.  161.  116.  108.
##  3  1972 101.   104.   124.  134.  111.  156.  117   109.
##  4  1973 103.   108    124.  138.  109.  155.  120.  111.
##  5  1974 108.   110.   127.  133.  112.  151.  124.  116.
##  6  1975 112.   115.   127.  131   110.  148.  123.  120.
##  7  1976 116.   119.   128   134.  113.  153   126.  124.
##  8  1977 117.   123.   126.  132   117.  153.  128.  126.
##  9  1978 123    127.   126.  129.  118.  156.  131.  127.
```

Long

Long has many rows. In particular, if there are n units, T times, and 1 variable we observe, Long has nT rows and 3 columns.

```
cigs_long
```

```
## # A tibble: 1,209 x 3
##     year State purchases
##    <dbl> <chr>     <dbl>
##  1  1970 AL         89.8
##  2  1970 AR        100.
##  3  1970 CA        123
##  4  1970 CO        125.
##  5  1970 CT        120
##  6  1970 DE        155
##  7  1970 GA        110.
##  8  1970 IA        108.
##  9  1970 ID        102.
## 10  1970 IL        125.
## # ... with 1,199 more rows
```

Converting Wide-to-Long and back

There are *many* different ways to pivot. Tidvyverse has a number of builtin tools. You could write something to do it by hand. Etc.

```
cigs_long =cigs %>%
      pivot_longer(cols = !year,
                   names_to="State",
                   values_to="purchases")
```

```
cigs_wide = cigs_long %>%
    pivot_wider(names_from = State,
                values_from = purchases)
```

Complications

That looked nice and smooth.

As soon as you have complications, like missing values, or other nonsense, it gets worse.

Merges/Joins

Multiple Sources

Sometimes you have two different sources of information.

E.g. Zillow home data – we have a pile of characteristics for each home. We also have sale prices and sale dates for each home.

Why are these separate?

Looking at Zillow Data

```
sales
```

```
## # A tibble: 90,275 x 3
##    parcelid logerror transactiondate
##       <dbl>    <dbl> <date>
##  1 11016594   0.0276 2016-01-01
##  2 14366692  -0.168  2016-01-01
##  3 12098116  -0.004  2016-01-01
##  4 12643413   0.0218 2016-01-02
##  5 14432541  -0.005  2016-01-02
##  6 11509835  -0.270  2016-01-02
##  7 12286022   0.044  2016-01-02
##  8 17177301   0.164  2016-01-02
##  9 14739064  -0.003  2016-01-02
## 10 14677559   0.0843 2016-01-03
## # ... with 90,265 more rows
```

Each place could be sold multiple times.

Looking at Zillow Data

```
prop
```

```
## # A tibble: 2,985,217 x 58
##    parcelid airconditioning~ architecturalst~ basementsc
##       <dbl>            <dbl> <lgl>                   <db
##  1 12544196               NA NA
##  2 12864620               NA NA
##  3 13113622               NA NA
##  4 11031288               NA NA
##  5 14223664               NA NA
##  6 11626844                1 NA
##  7 17101366               NA NA                        2
##  8 11512387               NA NA
##  9 13883603               NA NA
## 10 13133189               NA NA
## # ... with 2,985,207 more rows, and 53 more variables: b
## #   buildingclasstypeid <dbl>, buildingqualitytypeid <db
## #   calculatedbathnbr <dbl>, decktypeid <dbl>, finishedf
```

Merges

But we want to use the characteristics to predict sales qualities. So we need all the data combined.

This is a merge.

Merge Problems: Non-perfect Match

```r
nrow(prop)
```

```
## [1] 2985217
```

```r
nrow(sales)
```

```
## [1] 90275
```

We do not have a sale for every property. In many situations, we may not have property characteristics for every sale (not a problem here).

Merge Problems: Non-perfect Match

In this case, we don't care about non-matched properties. We want a training data set. We will subset to the data in the sales (our outcome).

Sales data without characteristics would pose a larger concern. But only accepting "matched" values helps.

Merge Problems: Duplicates

```r
sum(duplicated(sales$parcelid))
```

```
## [1] 125
```

And we have some duplicates.

The easiest thing to do with duplicates is to throw them away.

STOP

Merge Problems: Duplicates

Throwing away duplicates could cause problems. What kind of problems?

If homes that sell frequently are fundamentally different from those that don't, and are important to our target questions, then throwing them out may bias our whole procedure.

What then? Keep all sales *if it won't break your model*.

Joins

There are a few basic Joins between Data A and B:

- Left Join: Keep all data in A, add in data from B when available.
- Right Join: Keep all data in B, add in data from A when available
- Inner Join: Keep all observations that are in both A and B
- Outer Join: Keep all observations. Set `NA` for missing columns for data that isn't in both datasets.

We can't build a model using characteristics to predict sales errors without data in A and B, so we want an inner join.

Joining

```
data = inner_join(sales,prop,by=c(parcelid="parcelid"))
data
```

```
## # A tibble: 90,275 x 60
##    parcelid logerror transactiondate airconditioning~ ar
##       <dbl>    <dbl> <date>                     <dbl> <l
##  1 11016594   0.0276 2016-01-01                     1 NA
##  2 14366692  -0.168  2016-01-01                    NA NA
##  3 12098116  -0.004  2016-01-01                     1 NA
##  4 12643413   0.0218 2016-01-02                     1 NA
##  5 14432541  -0.005  2016-01-02                    NA NA
##  6 11509835  -0.270  2016-01-02                     1 NA
##  7 12286022   0.044  2016-01-02                    NA NA
##  8 17177301   0.164  2016-01-02                    NA NA
##  9 14739064  -0.003  2016-01-02                    NA NA
## 10 14677559   0.0843 2016-01-03                    NA NA
## # ... with 90,265 more rows, and 55 more variables: base
## #   bathroomcnt <dbl>, bedroomcnt <dbl>, buildingclassty
```

IDs

How did we do this?

We had a unique ID for each property. The `parcelid`. This ID was present in both datasets. So if we saw the same ID in each dataset, we knew the property was the same and we could match it.

Sometimes this happens. And when it does, it becomes very easy to combine information across different sources.

- IDFA: unique ID for each apple device
- SSN: unique ID for most US taxpayers
- URL: unique ID for most websites
- IP Address: unique ID for most web-capable devices

IDs pt 2

But often we don't have this information for some of the data we are using.

Methods:

1. Try to reconstruct an Identifier.
   - SSNs historically were relatively easy to identify if you knew birthdate and place of birth.
2. Match on other dimensions.
   - Maybe you don't observe SSN in any of your data. Or place of birth.
   - But you see a billing address, a name, and a phone number.
   - We can match on those dimensions too.

IDs

Matching on other characteristics becomes tricky. The ideal is to say you have a match when multiple characteristics are all identical.

In practice, even this is unlikely.

- Is "Connor Dowd" the same as "Connor J Dowd", or "Connor M Dowd" or "connor dowd"?
- Billing addresses may leave off apartment numbers, etc.
- One of the datasets may be missing one or more details. Does "1(800)333-2283" = NA?

IDs

But these things add up. Even if the address is an apartment building with 5k people in it, you've narrowed things down from 7 billion to 5k. Now how many are named "Connor" or "Dowd"?

Matching on these things rapidly becomes more art than science. There are many resources I can point you to if you have more Qs.

Privacy In two minutes

Privacy in data is mostly out of bounds for this class.

- If you are publishing data, you need to be careful about what you include that people could use to match with.
  - "Anonymising" by dropping e.g. names, is not enough
- The recommended procedure is to add some noise to your data before publishing if you're worried
  - E.g. Randomly change 2 digits in each phone number, randomly adjust numbers by small amounts, randomly drop some data and add in a few fake data points

NAs

What is NA?

Usually `NA` denotes missingness. We don't know whether or not home 1235532 has a fireplace. So we don't put `TRUE` and we don't put `FALSE`, we put `NA`.

Once again, much like duplicates, the standard advice is to drop observations with some `NA` values.

Dropping NAs

Dropping `NA` observations tends to be justified by the following assumption:

- The data is missing *at random*

That is – the observations where we don't know are more like a clerical error than a selection issue. It is not that missing observations are fundamentally different, it is that someone forgot to check a box.

This is frequently implausible.

Dropping NAs Nevertheless

Even under that assumption, dropping those observations can be devastating to your sample size.

```
data %>% drop_na()
```

```
## # A tibble: 0 x 60
## # ... with 60 variables: parcelid <dbl>, logerror <dbl>,
## #   transactiondate <date>, airconditioningtypeid <dbl>,
## #   architecturalstyletypeid <lgl>, basementsqft <dbl>,
## #   bedroomcnt <dbl>, buildingclasstypeid <dbl>, buildin
## #   calculatedbathnbr <dbl>, decktypeid <dbl>, finishedf
## #   calculatedfinishedsquarefeet <dbl>, finishedsquarefe
## #   finishedsquarefeet13 <dbl>, finishedsquarefeet15 <db
## #   finishedsquarefeet50 <dbl>, finishedsquarefeet6 <dbl
## #   fireplacecnt <dbl>, fullbathcnt <dbl>, garagecarcnt
## #   garagetotalsqft <dbl>, hashottuborspa <lgl>, heating
## #   latitude <dbl>, longitude <dbl>, lotsizesquarefeet <
## #   poolsizesum <dbl>, pooltypeid10 <lgl>, pooltypeid2 <
## #   pooltypeid7 <dbl>, propertycountylandusecode <chr>,
```

Dropping NAs

```
colMeans(apply(data,2,is.na))[1:24]
```

```
##                      parcelid                         logerro
##                   0.000000000                       0.0000000
##               transactiondate           airconditioningtype
##                   0.000000000                       0.6811852
##      architecturalstyletypeid                      basementsq
##                   1.000000000                       0.9995236
##                   bathroomcnt                       bedroomc
##                   0.000000000                       0.0000000
##           buildingclasstypeid           buildingqualitytype
##                   0.999822764                       0.3645638
##              calculatedbathnbr                       decktype
##                   0.013093326                       0.9927111
##      finishedfloor1squarefeet  calculatedfinishedsquarefe
##                   0.924054279                       0.0073220
##           finishedsquarefeet12              finishedsquarefeet
##                   0.051830518                       0.9996344
```

NAs

Some of those columns *were entirely missing observations*. We can probably ditch those columns.

But some of them just had a lot of missing observations. Like `fireplacecnt`, which is missing in 89% of homes.

It seems likely that it is not missing at random.

```
table(data$fireplacecnt,useNA="ifany")
```

```
##
##     1     2     3     4     5  <NA>
##  8165  1106   312    21     3 80668
```

It seems likely that people without fireplaces didn't enter a number for fireplace count. Dropping those observations would be a mistake. Most of them are just 0s.

NAs

Instead, what you should do is try to also model the `NA`s.

For some model types and variables this is difficult. We could probably just make `NA` a 0 for `fireplacecnt`. But what about for `airconditioningtype` which is just a factor anyhow? Or for `Square footage`, where it definitely exists?

NAs

For factor variables, this is straightforward. Just make one of your levels "NA". But what about Square footage?

A good rule of thumb *can* be to replace NA with a value that is impossible. E.g. replace `NA` with `-1` for square footage.

1. This will be easily spotted by others using the data, so you don't screw them up.
2. For flexible model types, the model can now just try to make predictions when square footage is negative.
   - E.g. for tree/forest model or KNN, if the model wants it can easily partition those observations away from everything else.

NAs

```
na_helper = function(x) {
  if (is.factor(x)) {
    levels(x) = c("-1",levels(x))
    x[is.na(x)] = -1
  }
  if (is.logical(x)) {
    x = x*1
    x[is.na(x)] = -1
  }
  if (is.character(x)) {
    x[is.na(x)] = "-1"
  }
  if (is.numeric(x)) {
    x[is.na(x)] = -1
  }
  x
}
```

NAs

```
prop2na = prop2 %>% mutate(fips = as.numeric(fips))
prop2na = prop2na %>% mutate(across(where(is.character),as.
prop2na = prop2na %>% mutate(across(everything(),na_helper)
```

This will not work as well for a linear model – where its predictions for square footage −1 affect its predictions for square footage of 2000.

Predictions 2

My forecast

My forecast was very straightforward.

```
small = cases %>% filter(dow == 1 | dow == 5) #Pull out Thu
small %>% filter(Date > "2021-04-25") %>% select(`New Cases
```

```
## # A tibble: 3 x 2
##   `New Cases` Date
##         <dbl> <date>
## 1       44766 2021-05-06
## 2       37885 2021-05-02
## 3       60196 2021-04-29
```

```
37885*(44766/60196)
```

```
## [1] 28173.96
```

Log Diff-in-Diff

Diff in Diff says that the change between April 29th and May 6th is going to be the same as the change between May 2 and May 9. This is a presumption that the trends hold.

Log Diff-in-Diff says that the percent change from April 29th to May 6th will be the same as the percent change from May 2 to May 9.

```
logdiff = log(44766)-log(60196)
logsun = log(37885)
logpred =  logsun + logdiff
pred = exp(logpred)
pred
```

```
## [1] 28173.96
```

Picking That

I also checked to see if the log diff-in-diff model using Thursday's 7-day averages was better. It wasn't.

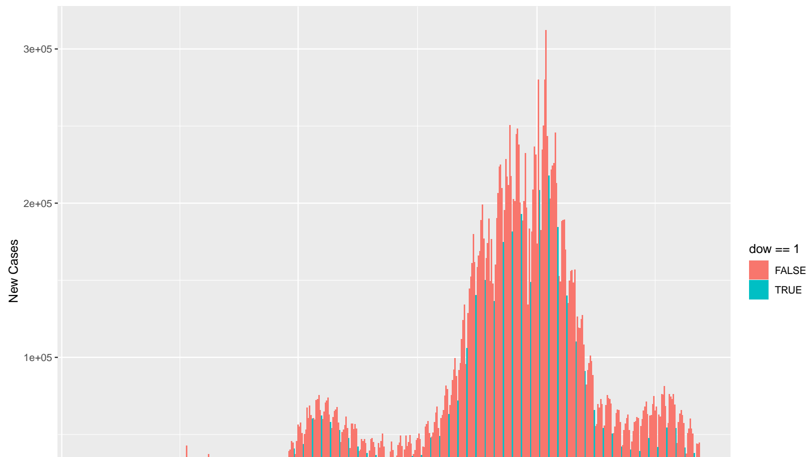
This was *extremely* easy to do out of sample *because* there was nothing estimated. Each week, I purely take 3 numbers from the prior 8 days to predict sunday.

Additionally, this was automatically adaptive to sundays. Its a week-on-week change times the last sunday. Any "sunday fixed effect" gets incorporated automatically.

Sundays are Important

A lot of models looked tuned to predict an average next day (or 3rd day away), rather than a Sunday.

```
ggplot(cases,aes(x=Date,y=`New Cases`,fill=dow==1))+
          geom_col(size=0.1)
```



Price-Is-Right

There was no real wrong answer here. I decided that I would use the 20th percentile for my prediction. A lot of people seemed to have similarly ad hoc procedures.

- *predicting is different in competitions*

But a big difference was confidence interval percentiles vs predictive interval percentiles.

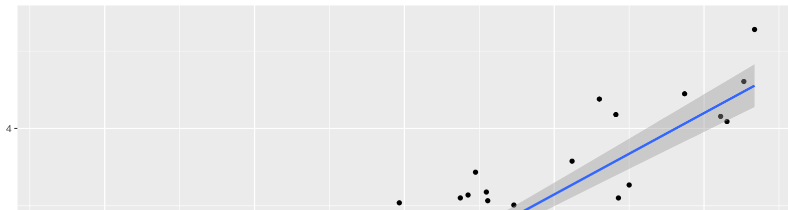
CI vs PI

Let me illustrate the difference with some made up data.

```
n = 100
x = rnorm(n)
y = rnorm(n)+2*x
df = data.frame(x=x,y=y)
```

```
ggplot(df,aes(x=x,y=y))+
  geom_point()+
  geom_smooth(method="lm",se=T)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Probabilities

When you have an unbiased distribution of errors
(i.e. out-of-sample distribution) for a model, getting event probabilities becomes somewhat straightforwards.

Take your prediction. Add those errors. See how often thing happens.

Almost all of you did something like this. Well done.

Ensemble Probabilities

What do you do if you have 10 models though? Find probability *under* each model – then combine.

1. Build each model.
2. Get honest error distributions for each model.
3. Predict P[event] given error distributions for each model
4. Combine $P[event|M_1], ..., P[event|M_{10}]$ in a "sensible way"

Sensible?

If you're averaging these models for your predictions, you can average these probabilities.

What I'm trying to avoid is the following procedure:

1. Build each model.
2. Average to get ensemble prediction
3. Look at distribution of predictions around average to determine uncertainty

And other similar mistakes. We are uncertain about the best model, and each model has beliefs about $P[event]$. We need to combine them well.

Model Uncertainty

What if we only built one model?

- How much do you trust it?

If you had a model telling you that tomorrow the stock price for Apple was guaranteed to go up 50%, does that tell you about Apple or about the model?

If a guy at a casino tells you he has a system, and you should bet it all on red for *certain* winnings, are you now certain of winnings if you bet it all on red?

Model Uncertainty

More broadly, we want to know about the ways in which our models might fail that are relevant.

So an important question is "how often does this model fail in ways relevant to this questions?"

For models that are reasonably certain about an event – this will dominate our uncertainty. E.g. we don't know the guy at the casino, so our difficulty trusting him dominates the chance that we don't win on red.

Model Uncertainty

Why does this matter?

Models in big data settings can become reasonably certain about events not happening, or happening. They have a lot of data, they have a model, the two combine for a lot of certainty.

But the model being certain does not mean *you* need to be certain.
The data wants to trick you

And in competitive settings, like a casino, or the stock market, "the data wants to trick you" is less a metaphor than you might think.

Wrap up

Things to do

Homework 5 is due tomorrow.

See you Thursday.

Bye!