# Data Wrangling and ML 1 (Advanced Regression)

## JSC 370: Data Science II

### February 5, 2024

# Today's goals

We will learn how to wrangle and manipulate large data with `dtplyr` - in particular,

- Selecting variables.
- Filtering data.
- Creating variables.
- Summarize data.

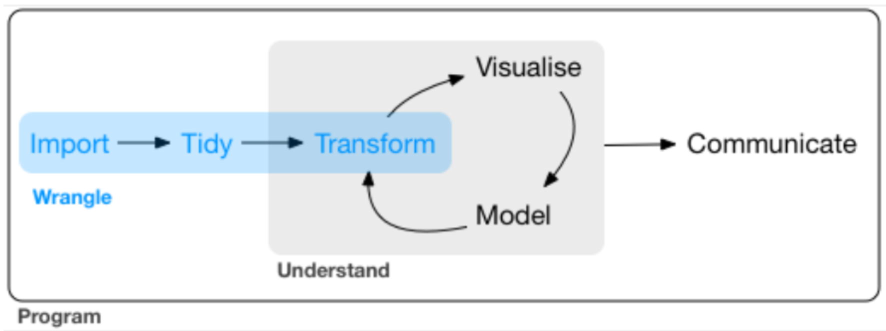
Throughout the session we will see examples using:

- **data.table** in R,
- **dtplyr** in R, and
- **pydatatable**

All with the MET dataset.

We will also take a look at advanced regression, for which you will need the `mgcv()` package.

# Data wrangling in R




Data wrangling describes the processes designed to import, clean up, and transform raw datasets from their messy and complex "raw" forms into high-quality data. You can use your wrangled data to produce valuable insights.

# Data wrangling in R

Overall, you will find the following approaches:

- **base R**: Use only base R functions.
- **dplyr**: Using "verbs".
- **data.table**: High-performing (ideal for large data)
- **dplyr + data.table = dtplyr**: High-performing + dplyr verbs.

Other methods involve, for example, using external tools such as Spark, sparkly.

We will be focusing on data.table because of this

Take a look at this very neat cheat sheet by Erik Petrovski here.

# Selecting variables: Load the packages

```
library(data.table)
library(dtplyr)
library(dplyr)
library(ggplot2)
library(mgcv)
library(lubridate)
```

The `dtplyr` R package translates `dplyr` (`tidyverse`) syntax to `data.table`, so that we can still use **the dplyr verbs** while at the same time leveraging the performance of `data.table`.

The `mgcv` package enables advanced regression models with basis splines.

# Loading the data

We will use the MET dataset, which we can download (and load) directly in our session using the following commands:

```r
# Where are we getting the data from
met_url <- "https://raw.githubusercontent.com/JSC370/JSC370-2024/main/data/met.gz"

# Downloading the data to a tempfile (so it is destroyed afterwards)
# you can replace this with, for example, your own data:
# tmp <- tempfile(fileext = ".gz")
tmp <- "met.gz"

# We sould be downloading this, ONLY IF this was not downloaded already.
# otherwise is just a waste of time.
if (!file.exists(tmp)) {
  download.file(
    url      = met_url,
    destfile = tmp,
    # method   = "libcurl", timeout = 1000 (you may need this option)
  )
}
```

Now we can load the data using the `fread()` function.

# Reading in the data

In R, fread, do a quick wrangle to remove outliers (discovered earlier), and print head

```
met_dt <- fread(tmp)
met_dt <- met_dt[temp > -10][order(temp)]
head(met_dt)
```

In Python, import with `datatable`, read, and print first 5 rows

```
import datatable as dt
met_dt_py = dt.fread("met.gz")
met_dt_py.head(5)
```

Before we continue, let's learn a bit more on `data.table` and `dtplyr`

# `data.table` and `dtplyr`: Data Table's Syntax

- As you have seen in previous lectures, in `data.table` all happens within the square brackets. Here is common way to imagine DT:

```
DT[i , j, by]
```

Rows or logical rule
to subset the rows

Some function
of the data

To which groups
apply the function

Take **`DT`**
Subset rows using **`i`**
then calculate **`j`**
grouped by **`by`**

- Any time that you see **:=** in **j** that is "Assignment by reference." Using **=** within **j** only works in some specific cases.

# `data.table` and `dtplyr`: Data Table's Syntax

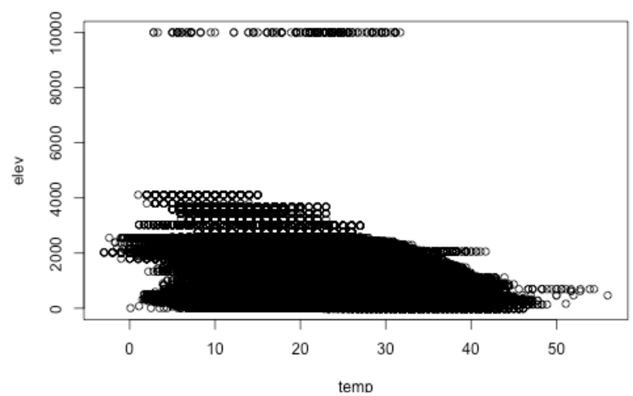
Operations applied in **j** are evaluated *within* the data, meaning that names work as symbols, e.g.,

```
# This returns an error (met is not referencing the data.table)
met[, elev]

# This works fine
met_dt[, elev]
```

# `data.table` and `dtplyr`: Data Table's Syntax

Furthermore, we can do things like this:

```
met_dt[, plot(temp, elev)]
```



## NULL

# Lazy loading, queries

- From Wikipedia "Lazy Loading" (also known as asynchronous loading) is a design pattern commonly used in computer programming and mostly in web design and development to defer initialization of an object until the point at which it is needed. It can contribute to efficiency in the program's operation if properly and appropriately used.
- Lazy loading means that the code for a particular function doesn't actually get loaded into memory until the last minute – when it's actually being used.
- When you create a "lazy" query, you're creating a pointer to a set of conditions on the database, but the query isn't actually run and the data isn't actually loaded until you call "next" or some similar method to actually fetch the data and load it into an object.

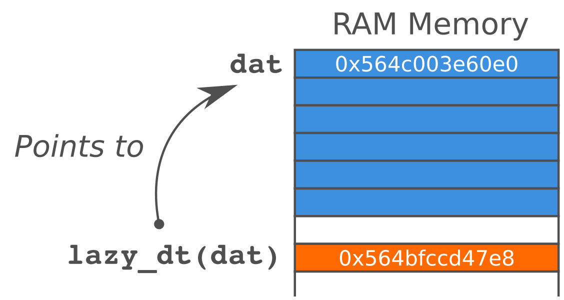
# `data.table` and `dtplyr`: Lazy table

- The `dtplyr` package provides a way to translate `dplyr` verbs to `data.table` syntax.
- The key lies on the function `lazy_dt` from `dtplyr` (see `?dtplyr::lazy_dt`).
- This function creates a wrapper that "points" to a `data.table` object

# `data.table` and `dtplyr`: Lazy table (cont.)

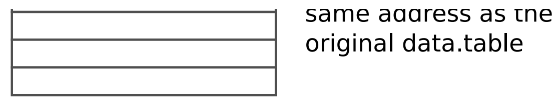
```r
# Creating a lazy table object
met_ldt <- lazy_dt(met_dt, immutable = FALSE)

# We can use the address() function from data.table
address(met_ldt)
address(met_ldt$parent)
```

```
## [1] "0x7fcbff8c9788"
## [1] "0x7fcbe5255c00"
```




The lazy table only points to the actual data, avoiding duplicating memory.

Lazy tables have their own address, but the underlying object has the




# `data.table` selecting columns

How can we select the columns `USAFID`, `lat`, and `lon`, using `data.table` where the `j` argument accepts the column names:

```
met_dt[, list(USAFID, lat, lon, temp, elev)]
# met_dt[, .(USAFID, lat, lon, temp, elev)]  # Alternative 1 (. is an alias to list)
# met_dt[, c("USAFID", "lat", "lon", "temp", "elev")] # Alternative 2
```

```
##          USAFID    lat      lon temp elev
##       1: 726764 44.683 -111.116 -3.0 2025
##       2: 726764 44.683 -111.116 -3.0 2025
##       3: 726764 44.683 -111.116 -3.0 2025
##       4: 726764 44.683 -111.116 -3.0 2025
##       5: 720411 36.422 -105.290 -2.4 2554
##      ---
## 2317200: 690150 34.300 -116.166 52.8  696
## 2317201: 690150 34.296 -116.162 52.8  625
## 2317202: 690150 34.300 -116.166 53.9  696
## 2317203: 690150 34.300 -116.166 54.4  696
## 2317204: 720267 38.955 -121.081 56.0  467
```

What happens if instead of `list()` you used `c()`?

# Selecting columns (cont. 1)

Using the **dplyr::select** verb:

```
met_dt |>
  select(USAFID, lat, lon, temp, elev)
```

```
##          USAFID    lat      lon temp elev
##       1: 726764 44.683 -111.116 -3.0 2025
##       2: 726764 44.683 -111.116 -3.0 2025
##       3: 726764 44.683 -111.116 -3.0 2025
##       4: 726764 44.683 -111.116 -3.0 2025
##       5: 720411 36.422 -105.290 -2.4 2554
##      ---
## 2317200: 690150 34.300 -116.166 52.8  696
## 2317201: 690150 34.296 -116.162 52.8  625
## 2317202: 690150 34.300 -116.166 53.9  696
## 2317203: 690150 34.300 -116.166 54.4  696
## 2317204: 720267 38.955 -121.081 56.0  467
```

# Selecting columns (cont. 2)

In the case of `pydatatable`

```
met_dt_py[:,["USAFID", "lat", "lon", "temp","elev"]]
```

What happens if instead of `["USAFID", "lat", "lon", "temp", "elev"]` you used `{"USAFID", "lat", "lon", "temp", "elev"}` (vector vs set).

# Selecting columns (cont. 3)

For the rest of the session we will be using these variables: USAFID, WBAN, year, month, day, hour, min, lat, lon, elev, wind.sp, temp, and atm.press.

```
# Data.table
met_dt <- met_dt[,
  .(USAFID, WBAN, year, month, day,
    hour, min, lat, lon, elev,
    wind.sp, temp, atm.press)
  ]

# Need to redo the lazy table
met_ldt <- lazy_dt(met_dt)
```

# Data filtering: Logical conditions

- Based on logical operations, e.g. `condition 1 [and|or condition2 [and|or ...]]`
- Need to be aware of ordering and grouping of `and` and `or` operators.
- Fundamental **logical** operators:

| x | y | Negate<br>!x | And<br>x & y | Or<br>x \| y | Xor<br>xor(x, y) |
|---|---|---|---|---|---|
| true | true | false | true | true | false |
| false | true | true | false | true | true |
| true | false | false | false | true | true |
| false | false | true | false | false | false |

- Fundamental **relational** operators, in R: `<`, `>`, `<=`, `>=`, `==`, `!=`.

# XOR operations

- The XOR logical operation, exclusive or, takes two Boolean operands and returns true if, and only if, the operands are different. Conversely, it returns false if the two operands have the same value.
- So, for example, the XOR operator can be used when we have to check for two conditions that can't be true at the same time.

# How many ways can you write an XOR operator?

Write a function that takes two arguments `(x,y)` and applies the XOR operator element wise. Here you have a template:

```
myxor <- function(x, y) {
  res <- logical(length(x))
  for (i in 1:length(x)) {
    res[i] <- # do something with x[i] and y[i]
  }
  return(res)
}
```

Or if vectorized (this would be better)

```
myxor <- function(x, y) {
  # INSERT YOUR CODE HERE
}
```

Hint 1: Remember that negating `(x & y)` equals `(!x | !y)`.

Hint 2: Logical operators are a distributive, meaning a `a * (b + c) = (a * b) + (a + c)`, where * and + are & or `|`.

In R

```
myxor1 <- function(x,y) {(x & !y) | (!x & y)}
myxor2 <- function(x,y) {!((!x | y) & (x | !y))}
myxor3 <- function(x,y) {(x | y) & (!x | !y)}
myxor4 <- function(x,y) {!((!x & !y) | (x & y))}
cbind(
  ifelse(xor(test[,1], test[,2]), "true", "false"),
  ifelse(myxor1(test[,1], test[,2]), "true", "false"),
  ifelse(myxor2(test[,1], test[,2]), "true", "false"),
  ifelse(myxor3(test[,1], test[,2]), "true", "false"),
  ifelse(myxor4(test[,1], test[,2]), "true", "false")
)
```

```
##      [,1]    [,2]    [,3]    [,4]    [,5]
## [1,] "false" "false" "false" "false" "false"
## [2,] "true"  "true"  "true"  "true"  "true"
## [3,] "true"  "true"  "true"  "true"  "true"
## [4,] "false" "false" "false" "false" "false"
```

Or in Python

```python
# Loading the libraries
import numpy as np
import pandas as pa

# Defining the data
x = [True, True, False, False]
y = [False, True, True, False]
ans = {
    'x'   : x,
    'y'   : y,
    'and' : np.logical_and(x, y),
    'or'  : np.logical_or(x, y),
    'xor' : np.logical_xor(x, y)
    }
pa.DataFrame(ans)
```

Or in Python (bis)

```
def myxor(x,y):
    return np.logical_or(
        np.logical_and(x, np.logical_not(y)),
        np.logical_and(np.logical_not(x), y)
    )

ans['myxor'] = myxor(x,y)
pa.DataFrame(ans)
```

We will now see applications using the `met` dataset

# Filtering (subsetting) the data

Need to select records according to some criteria. For example:

- First day of the month, and
- Above latitude 40, and
- Elevation outside the range 500 and 1,000.

The logical expressions would be

- `(day == 1)`
- `(lat > 40)`
- `((elev < 500) | (elev > 1000))`

Respectively.

In R with `data.table`:

```
met_dt[(day == 1) & (lat > 40) & ((elev < 500) | (elev > 1000))] |>
  nrow()
```

```
## [1] 27049
```

In R with **dplyr::filter()**:

```r
met_ldt |>
  filter(day == 1, lat > 40, (elev < 500) | (elev > 1000)) |>
  collect() |> # Notice this line!
  nrow()
```

```
## [1] 27049
```

With lazy tables, R delays doing any work until the last possible moment: it collects together everything you want to do and then sends it to the database in one step.

In Python

```
met_dt_py[(dt.f.day == 1) & (dt.f.lat > 40) & ((dt.f.elev < 500) | (dt.f.elev > 1000)), :].nrows
# met_dt_py[dt.f.day == 1,:][dt.f.lat > 40,:][(dt.f.elev < 500) | (dt.f.elev > 1000),:].nrows
```

In the case of pydatatable we use `dt.f.` to refer to a column. `df.` is what we use to refer to datatable's namespace.

The `f.` is a symbol that allows accessing column names in a datatable's `Frame`.

# More wrangling questions

1. How many records have a temperature within 18 and 25 C?
2. Some records have missings. Count how many records have `temp` as `NA`?
3. Following the previous question, plot a sample of 1,000 of `(lat, lon)` of the stations with temp as NA and those with data.

# Solutions

```r
# Question 1
message("Question 1: ", nrow(met_dt[(temp < 25) & (temp > 18)]))
```

```
## Question 1: 908047
```

```r
# met_dt[temp %between% c(18, 25), .N]

# met_ldt |>
#   filter(between(temp, 18, 25)) |>
#   collect() |>
#   nrow()

# Question 2
message("Question 2: ", met_dt[is.na(temp), .N])
```

```
## Question 2: 60089
```

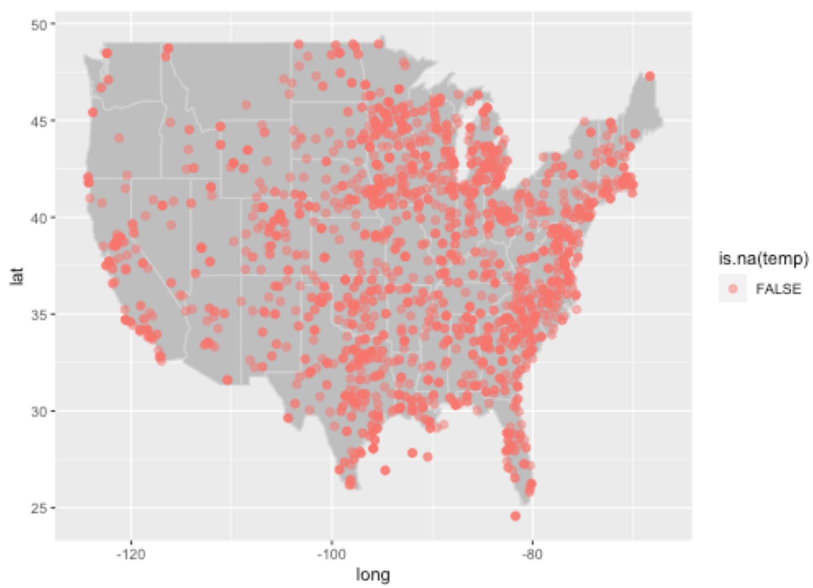
- Note the special symbol `.N` in `j`

- `.N` can be used in `j`, which is particularly useful to get the number of rows after subsetting

# Solutions (con't)

```
# Question 3
set.seed(123)
message("Question 3")

# Drawing a sample
idx <- met_dt[, list(x = sample.int(.N, 2000, replace = FALSE)), by = is.na(temp)]$x

# Visualizing the data
ggplot(map_data("state"), aes(x = long, y = lat)) +
  geom_map(aes(map_id = region), map = map_data("state"), col = "lightgrey", fill = "gray") +
  geom_jitter(
    data    = met_dt[idx],
    mapping = aes(x = lon, y = lat, col = is.na(temp)),
    inherit.aes = FALSE, alpha = .5, cex = 2
    )
```

# Solutions (con't)



# Creating variables: Data types

- **logical**: Bool true/false type, e.g. dead/alive, sick/healthy, good/bad, yes/no, etc.
- **strings**: string of characters (letters/symbols), e.g. names, text, etc.
- **integer**: Numeric variable with no decimal (discrete), e.g. age, days, counts, etc.
- **double**: Numeric variable with decimals (continuous), e.g. distance, expression level, time.

In C (and other languages), strings, integers, and doubles may be specified with size, e.g. in `python` integers can be of 9, 16, and 32 bits. This is relevant when managing large datasets, where saving space can be fundamental (more info).

# Creating variables: Special data types

Most programming languages have special types which are built using basic types. A few examples:

- **time**: Could be date, date + time, or a combination of both. Usually it has a reference number defined as date 0. In R, the `Date` class has as reference 1970-01-01, in other words, "days since January 1st, 1970".
- **categorical**: Commonly used to represent strata/levels of variables, e.g. a variable "country" could be represented as a factor, where the data is stored as numbers but has a label.
- **ordinal**: Similar to factor, but it has ordering, e.g. "satisfaction level: 5 very satisfied, ..., 1 very unsatisfied".

Other special data types could be ways to represent missings (usually described as `na` or `NA`), or special numeric types, e.g. `+-Inf` and Undefined (`NaN`).

When storing/sharing datasets, it is a good practice to do it along a dictionary describing each column data type/format.

# Questions 3: What's the best way to represent the following

- 0, 1, 1, 0, 0, 1
- Diabetes type 1, Diabetes type 2, Diabetes type 1, Diabetes type 2
- on, off, off, on, on, on
- 5, 10, 1, 15, 0, 0, 1
- 1.0, 2.0, 10.0, 6.0
- high, low, medium, medium, high
- -1, 1, -1, -1, 1,
- .2, 1.5, .8, $\pi$
- $\pi$, $\exp 1$, $\pi$, $\pi$

# Variable creation

If we wanted to create two variables, `elev^2` and the scaled version of `wind.sp` by it's standard deviation, we could do the following

With `data.table`

```
met_dt[, elev2         := elev^2]
met_dt[, windsp_scaled := wind.sp/sd(wind.sp, na.rm = TRUE)]

# Alternatively:
# met_dt[, c("elev2", "windsp_scaled") := .(elev^2, wind.sp/sd(wind.sp,na.rm=TRUE)) ]
```

# Variable creation (cont. 1)

With the verb **dplyr::mutate()**:

```
met_dt[, c("elev2", "windsp_scaled") := NULL] # This to delete these variables
met_ldt |>
  mutate(
    elev2         = elev ^ 2,
    windsp_scaled = wind.sp/sd(wind.sp,na.rm=TRUE)
  ) |>
  collect()
```

```
## # A tibble: 2,317,204 × 15
##    USAFID  WBAN  year month   day  hour   min   lat   lon  elev wind.sp  temp
##     <int> <int> <int> <int> <int> <int> <int> <dbl> <dbl> <int>   <dbl> <dbl>
##  1 726764 94163  2019     8    27    11    50  44.7 -111.  2025       0    -3
##  2 726764 94163  2019     8    27    12    10  44.7 -111.  2025       0    -3
##  3 726764 94163  2019     8    27    12    30  44.7 -111.  2025       0    -3
##  4 726764 94163  2019     8    27    12    50  44.7 -111.  2025       0    -3
##  5 720411   137  2019     8    18    12    35  36.4 -105.  2554       0    -2.4
##  6 726764 94163  2019     8    26    12    30  44.7 -111.  2025       0    -2
```

```
## 7 726764 94163 2019     8    26    12    50  44.7 -111.  2025     0    -2
## 8 726764 94163 2019     8    26    13    10  44.7 -111.  2025     0    -2
## 9 726764 94163 2019     8    27    10    30  44.7 -111.  2025     0    -2
## 10 726764 94163 2019    8    27    10    50  44.7 -111.  2025     1.5  -2
```

# Variable creation (cont. 2)

Imagine that we needed to generate all those calculations (scale by sd) on many more variables. We could then use the **.SD** symbol:

```r
# Listing the names
in_names  <- c("wind.sp", "temp", "atm.press")
out_names <- paste0(in_names, "_scaled")
met_dt[,
    c(out_names) := lapply(.SD, function(x) x/sd(x, na.rm = TRUE)),
    .SDcols = in_names
    ]

# Looking at the first 4
head(met_dt[, .SD, .SDcols = out_names], n = 4)
```

```
##    wind.sp_scaled temp_scaled atm.press_scaled
## 1:              0  -0.4955951               NA
## 2:              0  -0.4955951               NA
## 3:              0  -0.4955951               NA
## 4:              0  -0.4955951               NA
```

- Key things to notice here: **c(out_names)**, **.SD**, and **.SDCols**.
- More on `.SD`

# Variable creation (cont. 3)

In the case of dplyr, we could use the following

```
as_tibble(met_ldt) |>
  mutate(
    across(
      all_of(in_names),
      function(x) x/sd(x, na.rm = TRUE),
      .names = "{col}_scaled2"
      )
  ) |>
  # Just to print the last columns
  select(ends_with("_scaled2")) |>
  head(n = 4)
```

```
## # A tibble: 4 × 3
##   wind.sp_scaled2 temp_scaled2 atm.press_scaled2
##             <dbl>        <dbl>             <dbl>
## 1               0       -0.496                NA
## 2               0       -0.496                NA
```

```
## 3                 0      -0.496                 NA
## 4                 0      -0.496                 NA
```

Key thing here: This approach has no direct translation to `data.table`, which is why we used **as_tibble()**.

# Merging data

- While building the MET dataset, we dropped the State data.
- We can use the original Stations dataset and *merge* it to the MET dataset.
- But we cannot do it right away. We need to process the data somewhat first.

# Merging data (cont. 1)

```r
stations <- fread("ftp://ftp.ncdc.noaa.gov/pub/data/noaa/isd-history.csv")
stations[, USAF := as.integer(USAF)]

# Dealing with NAs and 999999
stations[, USAF   := fifelse(USAF == 999999, NA_integer_, USAF)]
stations[, CTRY   := fifelse(CTRY == "", NA_character_, CTRY)]
stations[, STATE  := fifelse(STATE == "", NA_character_, STATE)]

# Selecting the three relevant columns, and keeping unique records
stations <- unique(stations[, list(USAF, CTRY, STATE)])

# Dropping NAs
stations <- stations[!is.na(USAF)]

head(stations, n = 4)
```

```
##    USAF CTRY STATE
## 1: 7018 <NA>  <NA>
## 2: 7026   AF  <NA>
## 3: 7070   AF  <NA>
```

```
## 4: 8260 <NA>  <NA>
```

Notice the function `fifelse()`. Now, let's try to merge the data!

# Merging data (cont. 2)

```
merge(
  # Data
  x     = met_dt,
  y     = stations,
  # List of variables to match
  by.x  = "USAFID",
  by.y  = "USAF",
  # Which obs to keep?
  all.x = TRUE,
  all.y = FALSE
  ) |> nrow()
```

```
## [1] 2385443
```

This is more rows! The original dataset, `met_dt`, has 2317204. This means that the `stations` dataset has duplicated IDs. We can fix this:

```
stations[, n := 1:.N, by = .(USAF)]
stations <- stations[n == 1,][, n := NULL]
```

# Merging data (cont. 3)

We now can use the function `merge()` to add the extra data

```
met_dt <- merge(
  # Data
  x     = met_dt,
  y     = stations,
  # List of variables to match
  by.x  = "USAFID",
  by.y  = "USAF",
  # Which obs to keep?
  all.x = TRUE,
  all.y = FALSE
  )
```

```
head(met_dt[, list(USAFID, WBAN, STATE)], n = 4)
```

```
##    USAFID  WBAN STATE
## 1: 690150 93121    CA
## 2: 690150 93121    CA
```

```
## 3: 690150 93121    CA
## 4: 690150 93121    CA
```

What happens when you change the options `all.x` and `all.y`?

# Aggregating data: Adding grouped variables

- Many times we need to either impute some data, or generate variables by strata.
- If we, for example, wanted to impute missing temperature with the daily state average, we could use **by** together with the **data.table::fcoalesce()** function:

```
met_dt[, temp_imp := fcoalesce(temp, mean(temp, na.rm = TRUE)),
  by = .(STATE, year, month, day)]
```

- In the case of dplyr, we can do the following using **dplyr::group_by** together with **dplyr::coalesce**():

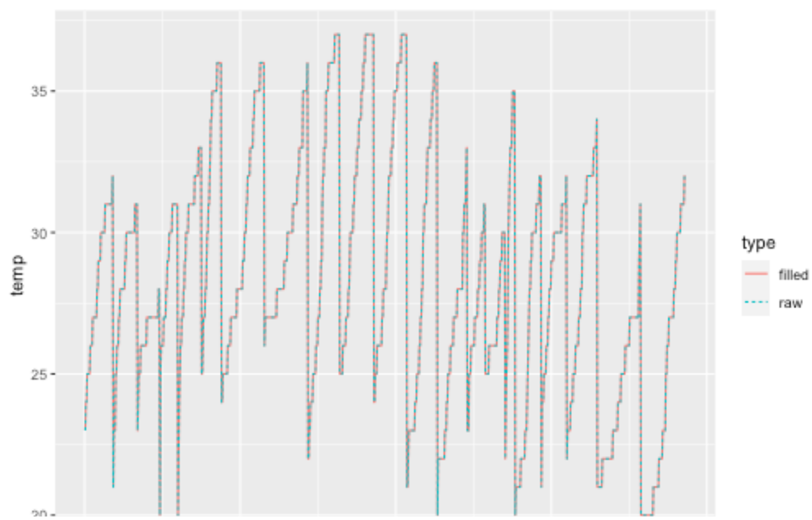
```
# We need to create the lazy table again, since we replaced it in the merge
met_ldt <- lazy_dt(met_dt, immutable = FALSE)

met_ldt |>
  group_by(STATE, year, month, day) |>
  mutate(
    temp_imp2 = coalesce(temp, mean(temp, na.rm = TRUE))
    ) |> collect()
```

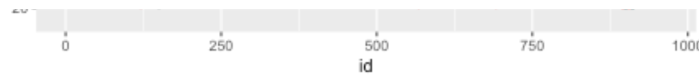
# Aggregating data: Adding grouped variables (cont.)

Let's see how it looks like

```
# Preparing for ggplot2
plotdata <-met_dt[USAFID == 720172][order(year, month, day)]
plotdata <- rbind(
  plotdata[, .(temp = temp, type = "raw")],
  plotdata[USAFID == 720172][, .(temp = temp_imp, type = "filled")]
)

# Generating an 'x' variable for time
plotdata[, id := 1:.N, by = type]

plotdata |>
  ggplot(aes(x = id, y = temp, col = type, lty = type)) +
  geom_line()
```

# Aggregating data: Adding grouped variables (cont.)






# Aggregating data: Summary table

- Using by also allow us creating summaries of our data.
- For example, if we wanted to compute the average temperature, wind-speed, and atmospheric preassure by state, we could do the following

```
met_dt[, .(
  temp_avg      = mean(temp, na.rm=TRUE),
  wind.sp_avg   = mean(wind.sp, na.rm=TRUE),
  atm.press_avg = mean(atm.press, na.rm = TRUE)
  ),
  by = STATE
  ][order(STATE)] |> head(n = 4)
```

```
##    STATE temp_avg wind.sp_avg atm.press_avg
## 1:    AL 26.19799    1.563645      1016.148
## 2:    AR 26.20697    1.872876      1014.551
## 3:    AZ 28.80596    2.983999      1010.771
## 4:    CA 22.36199    2.614711      1012.637
```

# Aggregating data: Summary table (cont. 1)

When dealing with too many variables, we can use the `.SD` special symbol in `data.table`:

```r
# Listing the names
in_names  <- c("wind.sp", "temp", "atm.press")
out_names <- paste0(in_names, "_avg")

met_dt[,
  setNames(lapply(.SD, mean, na.rm = TRUE), out_names),
  .SDcols = in_names, keyby   = STATE
  ] |> head(n = 4)
```

```
##    STATE wind.sp_avg temp_avg atm.press_avg
## 1:    AL    1.563645 26.19799      1016.148
## 2:    AR    1.872876 26.20697      1014.551
## 3:    AZ    2.983999 28.80596      1010.771
## 4:    CA    2.614711 22.36199      1012.637
```

Notice the **keyby** option here: "Group by STATE and order by STATE".

# Aggregating data: Summary table (cont. 2)

- Using **dplyr** verbs

```
met_ldt |>
  group_by(STATE) |>
  summarise(
    temp_avg      = mean(temp, na.rm=TRUE),
    wind.sp_avg   = mean(wind.sp, na.rm=TRUE),
    atm.press_avg = mean(atm.press, na.rm = TRUE)
  ) |> arrange(STATE) |> head(n = 4)
```

```
## Source: local data table [4 x 4]
## Call:   head(setorder(`_DT3`[, .(temp_avg = mean(temp, na.rm = TRUE),
##     wind.sp_avg = mean(wind.sp, na.rm = TRUE), atm.press_avg = mean(atm.press,
##         na.rm = TRUE)), keyby = .(STATE)], STATE, na.last = TRUE),
##     n = 4)
##
##   STATE temp_avg wind.sp_avg atm.press_avg
##   <chr>    <dbl>       <dbl>         <dbl>
## 1 AL        26.2        1.56         1016.
```

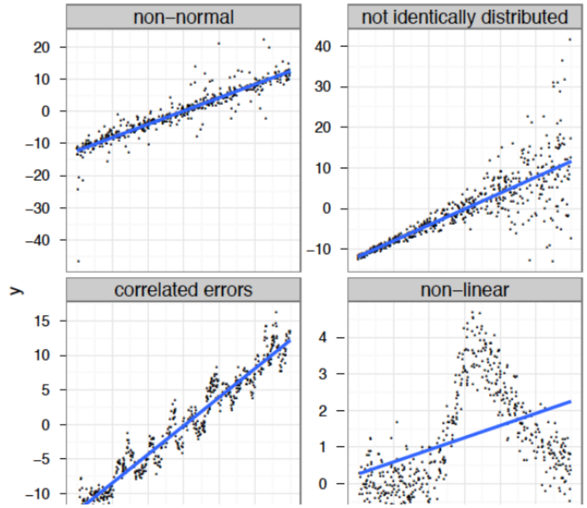
```
## 2 AR        26.2        1.87         1015.
## 3 AZ        28.8        2.98         1011.
## 4 CA        22.4        2.61         1013.
##
```

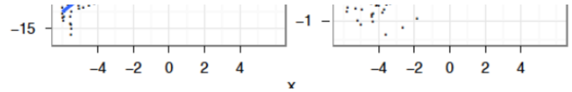
# Other data.table goodies

- `shift()` Fast lead/lag for vectors and lists.
- `fifelse()` Fast if-else, similar to base R's `ifelse()`.
- `fcoalesce()` Fast coalescing of missing values.
- `%between%` A short form of `(x < lb) & (x > up)`
- `%inrange%` A short form of `x %in% lb:up`
- `%chin%` Fast match of character vectors, equivalent to `x %in% X`, where both `x` and `X` are character vectors.
- `nafill()` Fill missing values using a constant, last observed value, or the next observed value.

# Machine Learning 1: Advanced Regression

- Linear regression is useful, but there are so many ways in which it can fail






# Machine Learning 1: Advanced Regression

- A linear model tries to fit the best straight line that passes through the data, so it doesn't work well for all datasets.
- In general, $Y(s) = f(s) + \epsilon$ where in regular linear regression $f(s)$ is a linear combination of variables $X\beta$.
- If we want to represent the regression more generally, we can define $f(s)$ as a "smooth" function described by a basis function consisting of 'non-linear' terms.

# Basis Function

Basics of Basis Functions

- We will start with a 1-dimensional, univariate case. For example this could be seen in time series, where we are modeling time (x) with basis functions.
- Polynomial bases are a good way to illustrate what is going on. Consider the regression model:

$$y_i = f(x_i) + \epsilon_i$$

and let's expand it out by a polynomial

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3 + \beta_4 x_i^4 + \epsilon_i$$

# Basis Function

Here

$$f(x_i) = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3 + \beta_4 x_i^4$$

is a 4th order polynomial. So, $f(x)$ is a function represented by **five** basis functions

$$f(x_i) = \sum_{j=1}^{5} x^j \beta_j = \sum_{j=1}^{5} b_j(x)\beta_j$$

that are defined by:

$$b_1(x) = 1, b_2(x) = x, b_3(x) = x^2, b_4(x) = x^3, b_5(x) = x^4$$

# Basis Functions

- In general, a basis is a set of functions that can be added together in a weighted fashion to form a more complicated function
- Here our weights are the regression coefficients $\beta_j$
- In general, a basis function is represented by

$$f_i = \sum b_j(x_i)\beta_j$$

$$\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{pmatrix} = \begin{bmatrix} 1 & b_1(x_1) & b_2(x_1) & b_3(x_1) & b_4(x_1) & b_5(x_1) \\ 1 & b_1(x_2) & b_2(x_2) & b_3(x_2) & b_4(x_2) & b_5(x_2) \\ 1 & b_1(x_3) & b_2(x_3) & b_3(x_3) & b_4(x_3) & b_5(x_3) \\ 1 & b_1(x_4) & b_2(x_4) & b_3(x_4) & b_4(x_4) & b_5(x_4) \\ 1 & b_1(x_5) & b_2(x_5) & b_3(x_5) & b_4(x_5) & b_5(x_5) \end{bmatrix} \times \begin{pmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \\ \beta_5 \end{pmatrix}$$

# Polynomial Basis

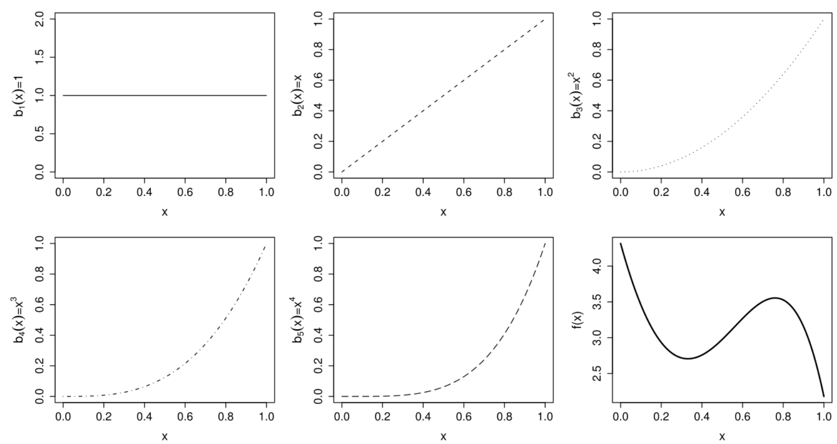
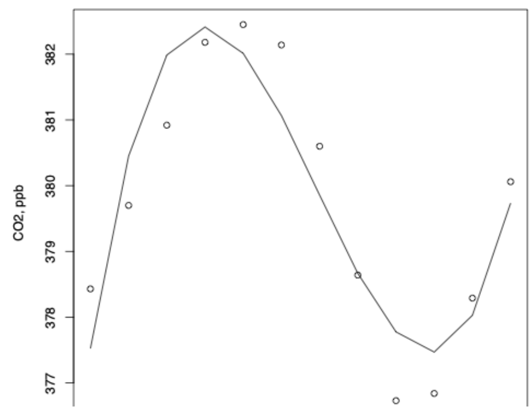


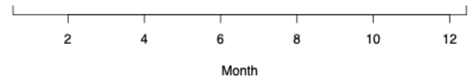

Figure 3.1 *Illustration of the idea of representing a function in terms of basis functions, using a polynomial basis. The first 5 panels (starting from top left), illustrate the 5 basis functions, $b_j(x)$, for a 4th order polynomial basis. The basis functions are each multiplied by a real*

*$b_j(x)$, for a 4th order polynomial basis. The basis functions are each multiplied by a real valued parameter, $\beta_j$, and are then summed to give the final curve $f(x)$, an example of which is shown in the bottom right panel. By varying the $\beta_j$, we can vary the form of $f(x)$, to produce any polynomial function of order 4 or lower. See also figure 3.2*

# Polynomial Basis

- The basis functions are each multiplied by $\beta_j$ and then summed to give the final curve $f(x)$. In the previous slide, this is shown in the bottom left figure.
- Below, we show this concept in terms of an example of $CO_2$ concentrations over a year (monthly data).





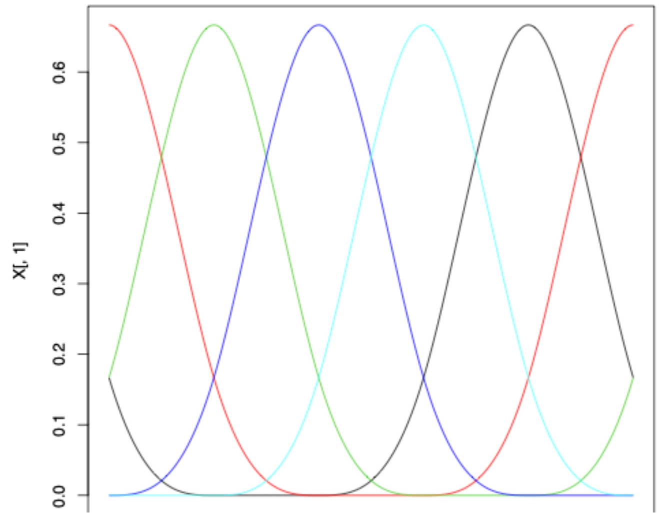

# Splines

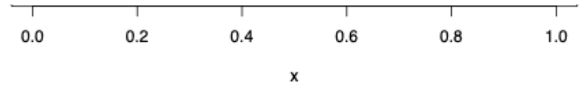
- In general, splines are curves that are formed by combining pieces of a polynomial.
- There are several types of splines including natural, cubic, and b-splines (the b stands for basis).

$$f(t_i) = \sum_{j=1}^{4} t^j \beta_j$$

- B-spline curves are made up of polynomial pieces and are defined by a set of knots.
- Choosing the number of knots defines how smooth (few) or wiggly (many) your functions.

# Splines






# Splines

- Smoothing splines with penalty allows us to estimate where to put the knots by penalizing the wiggliness of the function
- Minimize the function

$$\sum_i (y_i - f(t_i))^2 + \lambda \int f''(t)^2 \mathrm{d}t$$

.

- Here, $\lambda$ is a penalty parameter that controls how much to penalize wiggly functions (roughness penalty).
- Trade-off between the goodness of fit (the sum of squares) and the wiggliness of the function (the integral).
- Start by putting a knot at every data point, then penalize.
- It is an optimization problem m where we minimize:

$$\sum_i (y_i - B_i^T \beta)^2 + \lambda \beta^T S \beta$$

- the matrix S is constructed by using the spline basis we chose, B is the basis matrix

# Splines

- This function

$$\sum_i (y_i - f(t_i))^2 + \lambda \int f''(t)^2 \mathrm{d}t$$

  represents the loss + penalty.
- We see similar functions in lasso and ridge regression.
- The second derivative $f''(t)^2$ corresponds to how much the slope is changing (whereas the first derivative $f'(t)$ measures the slope of the function at $t$).
- The integral $\int f''(t)^2 \mathrm{d}t$ is a measure of the total change in the function $f'(t)$, over its entire range. If $f$ is very smooth then $f'(t)$ will be close to constant and the integral will take on a small value.
- A large $\lambda$ will will make the function $f$ smoother, but $\lambda = 0$ means the penalty has no effect and the function will be very wiggly.
- As $\lambda \to \infty$, $f$ will be perfectly smooth, a straight line that passes through the points.

# 1-D Splines

Types of 1-D splines include:

- cubic splines (basically piecewise cubic polynomials)
- cyclic splines (cubic splines with connected ends)
- basis splines (B-spline) with other polynomial orders
- cardinal splines (where knot placement is always a certain distance)
- wavelets (often cardinal wavelet splines)

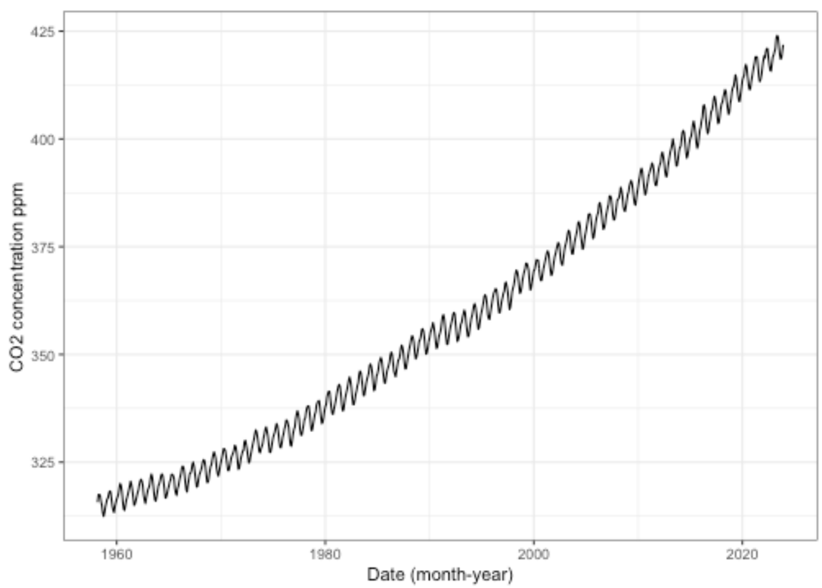
# Fitting Spline Regression Models

We will use CO$_2$ data from the Mauna Loa observatory in Hawaii: https://gml.noaa.gov/ccgg/trends/data.html

- important variables are: average (monthly CO$_2$ concentrations), year, month, and decimal.date
- we will make a month-year variable

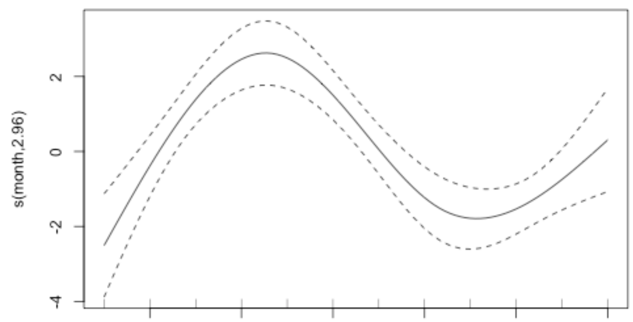
```
co2 <- read.csv("co2_mm_mlo.csv", skip=40)

co2 <- co2 |>
  mutate(month_year = make_date(co2$year, co2$month)) |>
  rename(co2 = average)

co2 |>
  ggplot(aes(y=co2,x=month_year)) +
  geom_line() +
  labs(x='Date (month-year)', y='CO2 concentration ppm')+
  theme_bw()
```

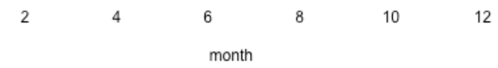
# Fitting Spline Regression Models



# Fitting Spline Regression Models

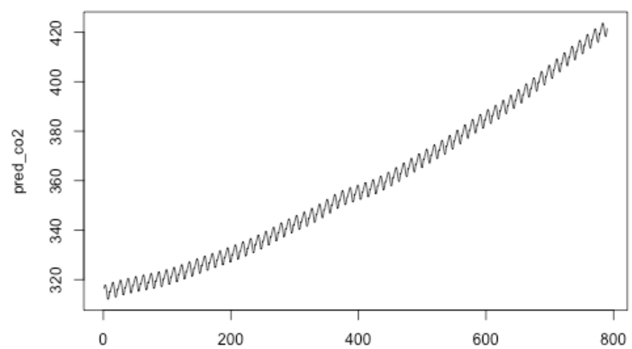
```
library(mgcv)
# Using cubic regression spline bases with 4 knots to show trends in one year
co2_2023 <- co2[co2$year==2023,]
gam_co2 <- gam(co2~s(month,bs="cr", k=4),data=co2_2023)
plot(gam_co2)
```






# Fitting Spline Regression Models

```
# try fitting to all data and smoothing date (overall trends) and month (to get within year tren
gam_co2_all <- gam(co2~s(decimal.date,bs="cr",k=20)+s(month,bs="cc"),data=co2)
# predict on data
pred_co2 <- predict.gam(gam_co2_all,co2)
plot(pred_co2,type='l')
```



# 2-D Splines

- Thin plate splines are smoothing splines in 2-d
- Extend the 1-d case to:

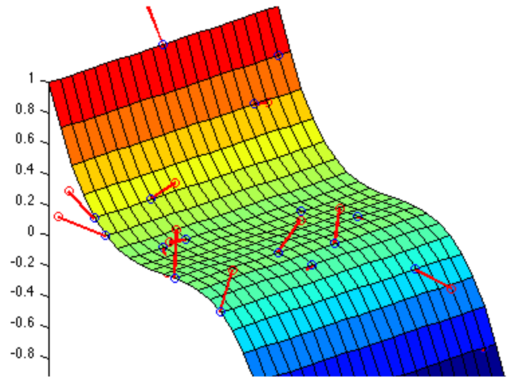
$$\sum_i (z_i - g(s_1, s_2))^2 + \lambda \iint g''(s_1, s_2)^2 \mathrm{d}s_1 \mathrm{d}s_2$$

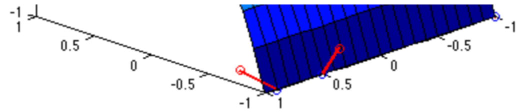
- where the penalty breaks down to the sum of the partial second derivatives
- $\lambda$ controls the "wiggliness" as in the 1-D spline (roughness penalty)

# Thin Plate Splines

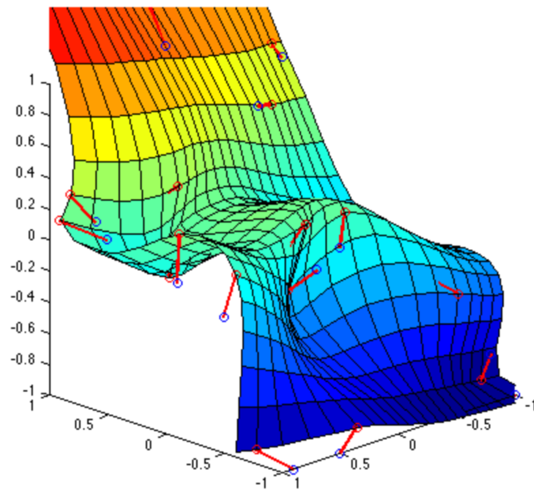
The idea behind a thin plate spline is:

- Basically we put a bendable plane through over the space and the points in the space pull the plane (by way of knots)
- Where there are more points grouped, we expect the plane to be pulled more significantly
- If there is a very bumpy surface, there will be more knots used and a more wiggly surface



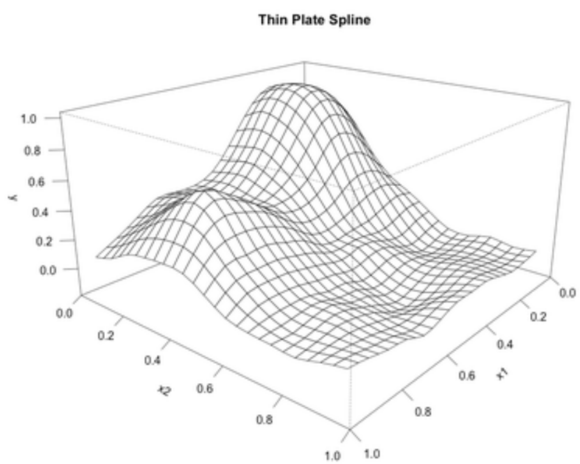



# Thin Plate Splines

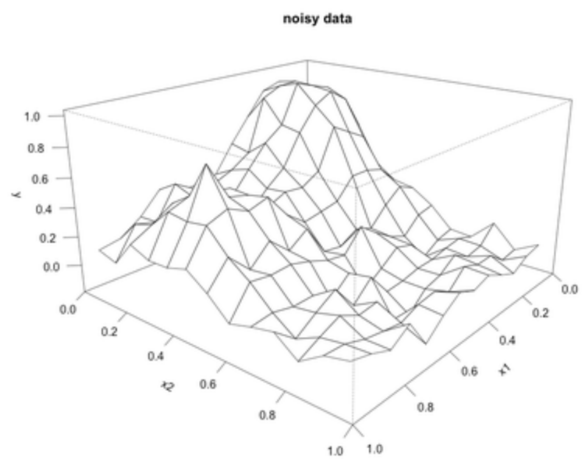


# Thin Plate Spline Regression

The height of where the surface is pulled is going to depend on the magnitude of what we are modeling, Y(s)

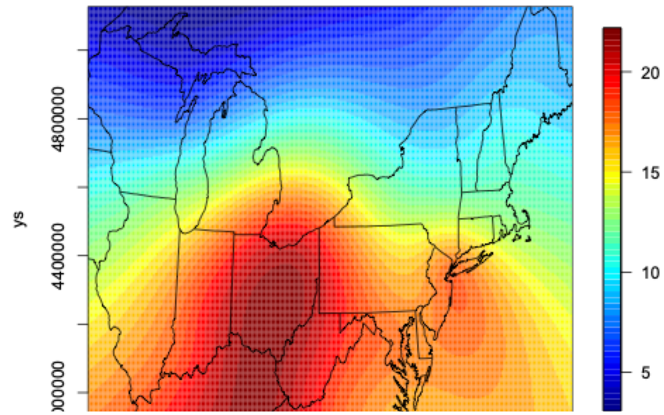


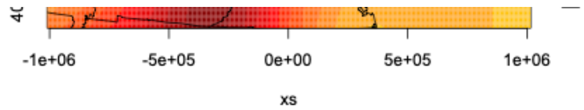
# Thin Plate Spline Regression



# Fitting Spline Regression Models

```
gam_temp <- gam(temp~s(x,y,bs="ts",k=60, fx=TRUE),data=idx)
plot(gam_temp)
summary(gam_temp)
```






# More on Advanced Regression

For more information and examples about regression that includes basis functions, see Ch 7 of [An Introduction to Statistical Learning with applications in R]