Lecture 5: Linear Regression: Regression in R

```
knitr::opts_chunk$set(
    message = FALSE,
    warning = FALSE,
    include = FALSE
)
library(fivethirtyeight)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(ggplot2)
library(printr)
library(GGally)
```

```
##
## Attaching package: 'GGally'
```

```
## The following object is masked from 'package:dplyr':
##
##     nasa
```

```
library(magrittr)
```

R Basics

Just some basic stuff about R.

Loading Packages

The basic unit of adding functionality in R is the `package`. These are installed using the `install.packages` command or using the GUI of RStudio. These will generally come from a central package repository for R packages called CRAN.

An installed package can be loaded into the present R session (or in an R script) with the `library()` command:

After a package is loaded, all of the functions included to it are available to use without any qualification. To see the list of functions in a package, type in, e.g.

`MASS::`

and hit `Tab`. In RStudio, this will autocomplete to all the functions.

Getting Help

Most functions in R have help pages associated to them. You can access this by prefacing a function with `?`, e.g.

```
?lm
```

Using `??` will search through all of the help pages that exist in your R installation, including packages that are not loaded.

In addition, many packages have *vignettes* built in, which are HTML or PDF documents that provide some sort of instruction about how to use some component of the package. Use these! You can use `browseVignettes` to look at them.

Data Types

The most basic data types in R are:

Vectors

Matrices

Data Frames

A data frame is the basic unit of data storage that you'll encounter in R (and a lot of practical ML in general, given the existence of `pandas` in Python). This is an abstract representation of a spreadsheet, essentially, and consists of a collection of rows of data that have features associated to them, represented by columns.

This is stored internally as a named list, where the names correspond to the names of the columns and the values in the list are vectors that represent the columns.

There are some useful methods that can be applied to data frames:

Reading Data

Data in R can come from several places. Sometimes it is built in (or in a package), in which case you can source it using `data`, e.g.

Sometimes it comes in a tabular form, like a CSV, in which case you can load it in using the `readr` package (there is a built-in csv reader for R, but `readr` does a better job of it):

```
df <- readr::read_csv('path/to/my.csv')
```

`readr` also supports any other delimited format like

RStudio has a nice button that makes importing data easy for you, it's the `Import Dataset` button, which also lets you import Excel, Stata, etc.

For other types of data sources, there are generally R packages that support loading them. Google or ask around for help if you encounter them.

Pipes

Pipes are an incredibly useful piece of syntactic sugar that comes from the R package magrittr. This introduces an operator `%>%` that does partial function composition:

- `x %>% f is equivalent to f(x)`
- `%>% f(y) is equivalent to f(x, y)`
- `%>% f %>% g %>% h is equivalent to h(g(f(x)))`

If the function you want to apply doesn't accept the thing you want to insert as the primary argument, then you can use `.` as a placeholder:

- `x %>% f(y, .) is equivalent to f(y, x)`
- `x %>% f(y, z = .) is equivalent to f(y, z = x)`

You can also write λ functions using braces, which is very nice for not having to explicitly define functions that you won't use repeatedly:

Finally, you can access names of the LHS of an expression using the `%$%` which is primarily useful for extracting the columns of a data frame as vectors:

Read the `magrittr` vignette for more details `vignette("magrittr")`, but you should try and use pipes to clean up your code! Don't go overboard, however – see: Bob Rudis.

Fitting Linear Models in R

Producing suitable input data for machine learning algorithms is of crucial importance to doing machine learning. Typically, we have a collection of data that is represented by the rows of a data frame. Below is the classic iris data set that is built in to R. This consists of 150 samples of measurements from subspecies of the **Iris* family, with measurements of the sepal and petal length and width (the sepal is the part below the petals):

Looking at the pairplots with the useful package GGAlly:

We see that it is definitely reasonable to try and predict the petal length from the width (of course these things are related!), but the data looks pretty linear, without even a dependence on the species. So, let's try and fit a linear hypothesis, with our target values being the petal length and the feature space being the 1-dimensional petal width space.

We now need to coerce our data into a training set, now, and to put it in a form that R's linear model function `lm` can understand. The standard way that these things are presented are as *design (or model) matrices*, which have columns that represent the dimensions of the feature space X. We provide this design matrix and a vector that stores the target values in the training set, $\{y_i\}$, and the machine learning algorithm should know what to do.

In this case, we'd want a matrix that looks like

With a target vector:

We can then stick this into the `lm.fit` function (which is in base R and fits linear models):

`lm.fit` returns a list of useful things (see `?lm.fit`), including the coefficients:

Which tells us that we expect that the petal length increases by about 2.8 cm for every increase of 1cm in petal length. We can access the predicted values and residuals:

Let's compare to our original values:

So, the average value of our loss function (for each data point) is .56, which doesn't seem bad. We can also compute the absolute error:

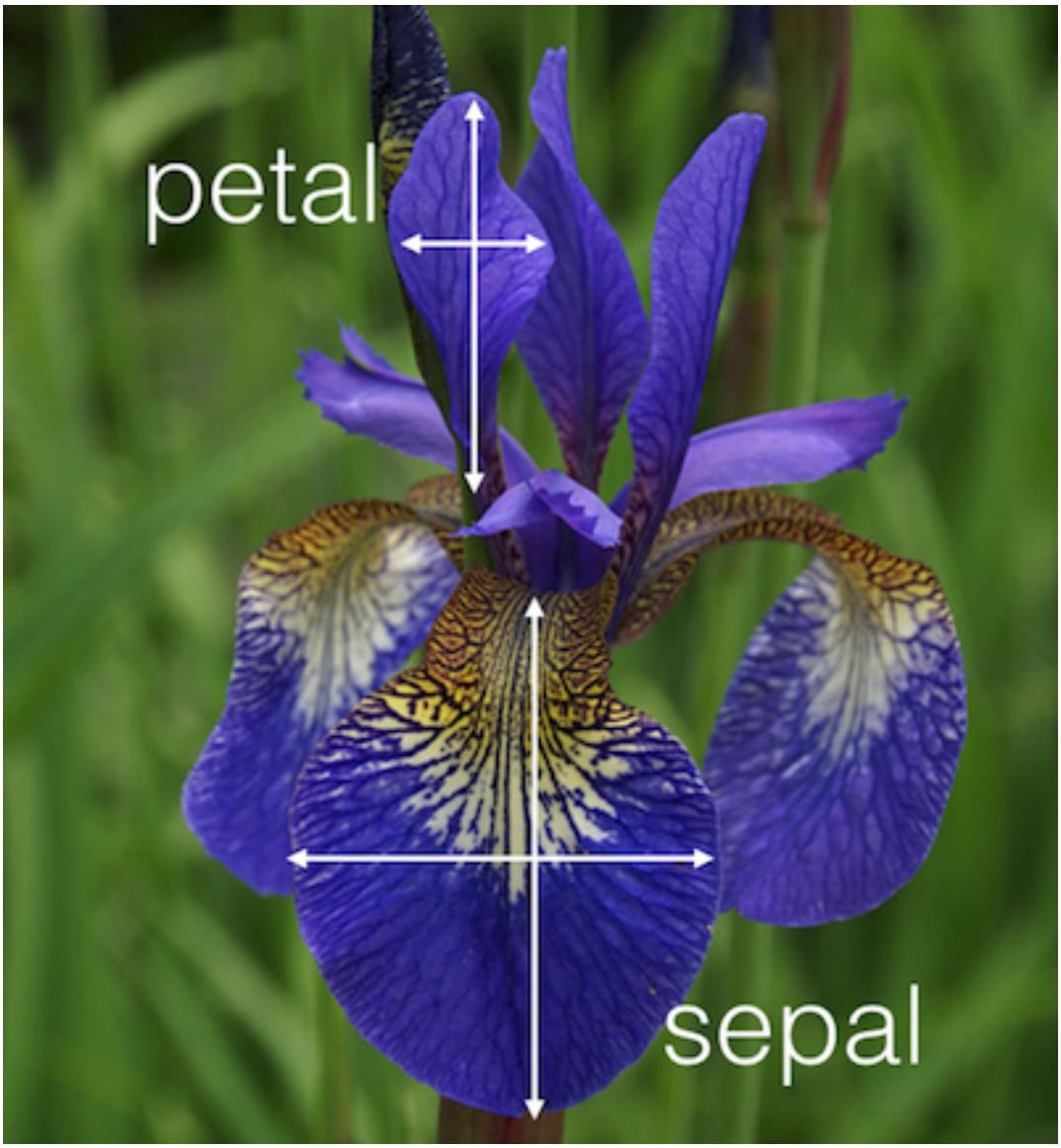

Figure 1: petal

Which doesn't seem too bad either. Let's plot:

This looks like it could definitely work better if we included the intercept! There was none here. We need to stick that in our old model matrix:

Let's try again:

Much lower!

The Formula Interface

As we just saw, it's a bit obnoxious to manually construct the matrices that go into `lm.fit` – it'd be much better if we could tell R: "construct a design matrix for me from the `iris` dataframe that uses the variable `Petal.Width` to predict `Petal.Length`, including an intercept term".

This sounds like wishful thinking, but this is exactly what R allows us to do, which makes specifying design matrices much simpler! Here's an example!

`model_frame` is a data frame that has some extra information about the formula that we specified. These are stored in the attributes of the data frame:

From this, we can use the `model.matrix` function to actually generate the matrix:

We could, now, pass this to the `lm.fit` function, but actually, it's even easier than this. The `lm` function will deal with formulas like this for us: