Intro to R days 7 and 8

Indrek Seppo and Nicolas Reigl

March 9-10, 2018

HITSA Preparation of this course was supported by HITSA – Estonian Information Technology Foundation for Education.

From the previous sessions

dplyr and piping

A typical workflow would be something like this:

```
newdataset <- initialdata %>%
  filter(variable>4, !is.na(othervariable))%>%
  group_by(variable1, variable2)%>%
  summarize(meanofvar=mean(variable1, na.rm=TRUE), othervalue=myfunction(variable4, variable5))
```

The main verbs of dplyr's grammar of data manipulation were:

1) **select()** for selecting variables
2) **filter()** for subsetting the dataset. You can separate different conditions by commas, for example `sex=="Male", age > 29`. A useful way to select all the values in a set is to use `%in%` operator: `education %in% c("high", "medium")`.
3) There was a powerful combination of:

- **group_by()** – creates groups in the data by any number of variables
- **summarize()** – computes some summarizing statistics of the variable (by groups if **group_by()** was used previously).

4) There was also **mutate** – to add variables to the dataframe.

Lets read the original piaac data in again

```
piaac <- read.csv("http://www.ut.ee/~iseppo/piaacest.csv")
```

ggplot2

A typical plot in the ggplot language looks something like this:

```
ggplot(data=mydataset, aes(x=firstvariable, y=secondvariable))+
  geom_something(aes(color=groupingvariable))+
  geom_otherthing(size=4)+
  facet_wrap(~othergroupingvariable)
```

If you want something at the graph to be connected to your data, you will need to put it into the **aes()**-call. If you want to tell it yourself (like the `size=4` here), it has to be outside of the `aes()`-call.

Factors

There were three most important functions in the `forcats` package:

- `fct_recode()` to manually change the names of the factor levels
- `fct_relevel()` to manually change the order of factor levels and
- `fct_reorder()` to change the order of factor levels by some other variable

Joining data

We looked at how to join data: `left_join()`, `right_join()`, `inner_join()`, `full_join()` `anti_join()` and `semi_join()`. There are also `bind_rows()` and `bind_cols()` for just adding two datasets on top of each other or next to each other.

Data in long and wide format

We looked at two functions in `tidyr`: `gather()` and `spread()`. While they might seem obscure at first, they are in fact very convenient and often used. `gather()` takes a number of variables and converts them into key-value pairs. `spread()` does the opposite, it creates new columns, setting its names from the key variable and populates the matrix with the value variable.

```
animals <- read.csv("http://www.ut.ee/~iseppo/animals.csv")
```

```
animals
```

```
##   animal length width age
## 1   bear     10     7   3
## 2    cat      4     3   5
```

```
newdata <- gather(animals, key="new var name", value= "value var name", length, width, age)
head(newdata, n=4)
```

```
##   animal new var name value var name
## 1   bear       length             10
## 2    cat       length              4
## 3   bear        width              7
## 4    cat        width              3
```

```
spread(newdata, key="new var name", value="value var name")
```

```
##   animal age length width
## 1   bear   3     10     7
## 2    cat   5      4     3
```

Your turn: YOUR TURN

- use `read_csv()` from package `readr` to download the tax data form `http://www.ut.ee/~iseppo/taxdata.gz`. Write it to some data object. Take a look at the variables there are.
- Use `dplyr`'s `group_by()` and `summarize()` to find nr of companies present in the dataset by quarter and county (you can use `n()` for just getting the nr of rows, as each row correspondends to one company, or you can use `length(unique(code))` to get the nr of unique id-codes in the data slice. You will get a dataset in long format.
- Convert it to a wide format using `spread()` from `tidyr` so that the quarters would now be columns (remember, you need to tell it from which variable the new columns will come by setting key="variable name", you also need to tell it from which variable will the value come by setting value="variable name" – in our case we only have one data column here, but there could be multiple ones.

  The end-result should look something like this:

```
## # A tibble: 2 x 4
##   county       q1.2017 q2.2017 q3.2017
##   <chr>          <int>   <int>   <int>
## 1 Harju county   67374   66225   66495
## 2 Hiiu county      749     729     744
```

An overly quick intro to regressions in R

First lets reread the piaac dataset without the modifications we have made into it:

```
piaac <- read.csv("http://www.ut.ee/~iseppo/piaacest.csv")
```

So you want to run some regressions. How to do it in R? We will use a regular least squares regression here, but everything else would be using more or less the same syntax.

First specify a regression model:

```
piaac$logincome <- log(piaac$earnmth)
model1 <- lm(logincome ~ edlevel3 + gender, weights=weights, data=piaac)
model1
```

```
##
## Call:
## lm(formula = logincome ~ edlevel3 + gender, data = piaac, weights = weights)
```

```
##
## Coefficients:
##    (Intercept)     edlevel3Low  edlevel3Medium      genderMale
##         6.5561         -0.5063         -0.3571          0.5209
```

The `model1` is not terribly informative, showing only the model and the point estimates. Lets try `summary()`:

```
summary(model1)
```

This is slightly better – you get a traditional regression table with standard errors and p-values etc. As we were using logincome as the dependent variable we can interpret the coefficients roughly as percentages[1].

This results dataobject that we just wrote the regression results into, is again a list, and we can get everything out of this list (take a look at `str(model1)`). You will find coefficients there, residuals, fitted values etc. So to access some coefficients (e.g. for automatic inclusion in our report) we can call:

```
model1$coefficients["edlevel3Medium"]
```

```
## edlevel3Medium
##     -0.3571447
```

There is actually a package called `broom` in the tidyverse that makes this much easier creating a regular dataframe of the coefficients:

```
library(broom)
model1overview <- tidy(model1)
model1overview
```

```
##             term   estimate  std.error statistic       p.value
## 1    (Intercept)  6.5560627 0.01635338 400.89967  0.000000e+00
## 2    edlevel3Low -0.5062603 0.03337649 -15.16817  1.369427e-50
## 3 edlevel3Medium -0.3571447 0.02086588 -17.11620  1.810446e-63
## 4     genderMale  0.5209098 0.01987817  26.20512 5.694088e-140
```

Another function `glance()` gives you a dataframe of some parameters helping you to understand how precise the model is with this data.

```
glance(model1)
```

```
##   r.squared adj.r.squared    sigma statistic       p.value df    logLik
## 1 0.1860817     0.1854798 6.764605  309.1765 9.008321e-181  4 -3863.257
##        AIC      BIC deviance df.residual
## 1 7736.515 7768.061 185647.9        4057
```

From the model object you can access fitted values, but residuals are usually more interesting. Lets add the residuals to the original dataframe. To do this we can use `augment()` function from broom[2]:

[1] Economists always tend to interpret them as rough percentages, but be aware if there are anyone from mathematics close to you. For small values of these coefficients they are indeed very close to percentages (so if a coefficient 0.1 for genderMale would have meant that men earn ~10% more). The precise nr would be `exp(0.1)-1=0.1051`, which is around 10%. But `exp(0.3)-1`is already 0.35 and `exp(0.5)-1` is `0.65` (so you should say 65% instead of 50% here). Handle this with some care when the coefficient is higher than 0.2 or so, or someone will remark it at your presentation.

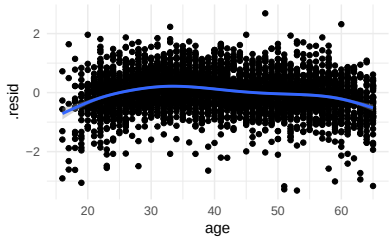
[2] There is an alternative – `residuals()` in base R would provide us the vector of residuals, augment() will work on many types of models but not all, the `residuals()` is usually implemented everywhere.

```
piaacRes <- augment(model1, data=piaac)
```

Note that the resulting piaacRes dataset is smaller than piaac – all the values which had NA-s in one of the regression variables are removed from here (you can add `na.action=na.exclude` as a parameter to the regression call leave the NA-s in).

Why is this important? You can now do some quick analysis of residuals – is there heteroscedasticity in the data? Just plot the residuals against other variables. Would adding some additional variables help? Lets try it and plot the residuals against age:

```
library(ggplot2)
ggplot(piaacRes, aes(x=age, y=.resid))+
  geom_point()+
  geom_smooth()+
  theme_minimal()
```



There is clearly a relationship, but it does not seem to be a linear one. Lets add a polynomial of age to the model:

```
model2 <- lm(logincome ~ edlevel3 + gender + poly(age,2), weights=weights, data=piaac)
```

We could have added `+age +age^2` as well – this would not have changed any other coefficients, but would mess up prediction intervals (why? Because R would treat `age` and `age^2` as separate independent variables while they are clearly not). You wouldn't usually care about prediction intervals, so many people are just adding the `age` and `age^2` to the model and so can you.

If you want to add cross-effects, use `*`:

```
model3 <- lm(logincome ~ edlevel3*gender + poly(age,2), weights=weights, data=piaac)
```

And to change the base-value of a nominal variable, make the value you want to use as a base to be the first in the factor:

```
library(forcats)
piaac$edlevel3<-fct_relevel(piaac$edlevel3, "Medium")
model4 <- lm(logincome ~ edlevel3*gender + poly(age,2), weights=weights, data=piaac)
```

Predictions at specified values

You can easily plot the predicted values of a regression by using a `predict()` function. You need to give the dataframe newdata to it, specifying for which values you want to get the predictions. You can add the initial dataset (but you would also get the predicted results for the initial dataset with a function fitted(modelname)). To create a dataset of all possible values, use `expand.grid()`:

```
mydata <- expand.grid(edlevel3=levels(piaac$edlevel3), gender=levels(piaac$gender), age=c(20:65))
mydata$prediction <- predict(model3, newdata=mydata)
ggplot(mydata, aes(x=age, y=prediction))+
  geom_point()+
  facet_wrap(~edlevel3)+
  theme_minimal()
```

If you want to futher visualize your regressions, take a look at package `visreg` http://pbreheny.github.io/visreg/. If you wan to have nicely outputed regression tables, package `stargazer` should help you out: https://www.jakeruss.com/cheatsheets/stargazer/.

Javelin throwers example

Lets do a regression another way around. Let us create a world with known parameters and then lets try to get these parameters back by using a regression analysis.

Whenever I try to understand how some statistical method works, I simulate the data and run it to see it myself. Even if you are able to follow the math, this will give you a much better intuitive understanding. You can easily see how would the method perform under different assumptions about the data, how wrong if will be on average etc.

Let there be 2000 persons, 600 of them males, and 1400 of them females. I will create a new dataframe simdata, and into this data frame a variable called gender, which is "Male" for the first 600 persons and "Female" for the next 1400 perosns. I will use a function `rep()` here, this just repeats something as many times as we want:

```
simdata <- data.frame(gender=c(rep("Male", 600), rep("Female", 1400)))
```

Next I would like to let every person have an age, a nr of months trained, and how long do they throw their javelin. For this we need to know how to generate

random data in R.

A bit about generating data in R

R has many kinds of distributions built in. So imagine you want to flip a coin, but you do not have a coin available. This would have previously ruined your day, but luckily you now know R. You just have to know that this is a binomial distribution – a coin can either have a value of 1 or 0, and you are done:

```
rbinom(n=1, size=1, prob=0.5)
```

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

Lets look at the help file:

```
?rbinom
```

`r` in front of `binom` in `rbinom()` means that you are asking R to generate a single random value from a binomial distribution, size=1 means that you only toss the coin once, and prob=0.5 means that there is 50% of chance for 0 and 50% of chance for 1. So if you want 10 coin flips:

```
rbinom(n=10, size=1,prob=0.5)
```

```
##  [1] 0 0 0 1 1 0 1 0 1 1
```

Run it again:

```
rbinom(n=10, size=1, prob=0.5)
```

```
##  [1] 1 1 1 0 0 1 0 1 1 0
```

You most probably had different values than the last time. This is because the values are randomly generated. Now, R is all about reproducible research. So what if you want random numbers, but so that they will stay the same next time? Yoy have to set a seed, this will always generate the same values:

```
set.seed(1)
rbinom(n=10, size=1, prob=0.5)
```

```
##  [1] 0 0 1 1 0 1 1 1 1 0
```

There are other distributions. If you want to take a random pick from a normal distribution with mean=100 and standard deviation of 10, you could write:

```
rnorm(n=1, mean=100, sd=10)
```

```
## [1] 91.79532
```

Lets assume every person has a height that is distributed normally. Lets assume the mean of our male javelin throwers is 180cm, with the standard deviation being 8cm, and of female javelin throwers 173cm with standard deviation of 8cm[3].

Let me first create a new variable height, that is by default NA:

```
simdata$height <- NA
```

I know that I have 600 males in the dataset and 1400 females. To generate 600 random values from a normal distribution with mean 180 and standard deviation 8 I would say:

```
rnorm(n=600, mean=180, sd=8)
```

I will now use one way to access a data frame to write these values to the positions of the length variable, where the gender is "Male":

```
set.seed(1)
simdata$height[simdata$gender=="Male"]<- rnorm(n=600, mean=180, sd=8)
```

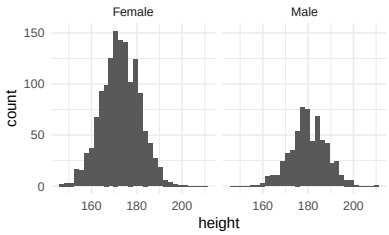
And similarly for women:

```
simdata$height[simdata$gender=="Female"]<- rnorm(n=1400, mean=173, sd=8)
```

Let us take a look if everything went alright. Lets create a graph showing us the distribution of the heights for our javelin throwers by gender.

Your turn:

- See if you manage to create the graph on the right. geom_histogram() is used as a geom, it only requires one x as aesthetic. Remember to use the aes() call to connect variables to aesthetics. facet_wrap() is used for creating the facets. Dont forget to load the ggplot2 library first.



- Actually we chose our javelin throwers randomly from a population with respective mean (180 for men, 173 for women) and standard deviations. Our sample mean does not have to be exactly this. Can you find what is the mean height for our men and women? I think it should be:

```
## # A tibble: 2 x 2
##   gender  mean
##   <fct>  <dbl>
## 1 Female  173.
## 2 Male    180.
```

[3] Remember what this meant – as this is a normal distribution, 66% of the males will be inside 1SD, so 180+/-8cm and 95% of males will be no further than 2 standard deviations, so 180+/-16cm.

Now lets assume that the taller a person is, the better he or she throws the javelin. For example lets connect the length and the result in the following way:

$$result = 50 + 0.2 * length + \varepsilon$$

So every cm of additional length would on average add 0.2m for your javelin result (and a person without any length at all would throw a whopping 50m[4]). And there is some random residual – a lot of other things that we cannot measure but is assumed to distribute normally (meaning there are a lot of small, independent deviations added together – remember how we got to the normal distribution).

Lets create this in our data as potential.

```
simdata$potential <- 50 + 0.2*simdata$height
```

Now assume that there is more to the ability to throw than just height of a person. Lets name this personal talent, and let this talent be normally distributed with standard deviation of 10:

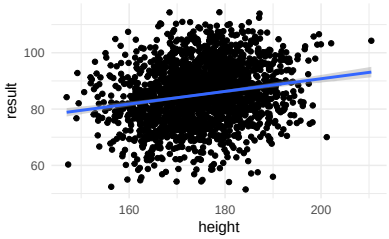
```
simdata$talent <- rnorm(2000, 0, 10)
```

The actual result will be sum of potential + talent (in fact it would probably be a product of those, but lets keep it simple):

```
simdata$result <- simdata$potential + simdata$talent
```

Look at the data visually:

```
ggplot(simdata, aes(x=height, y=result))+
  geom_point()+
  geom_smooth(method="lm")+
  theme_minimal()
```



Lets see how well would a regression recover this parameter:

```
regr1 <- lm(result ~ height, data=simdata)

summary(regr1)
```

You see that the height parameter is 0.2247 with a standard error of 0.0259, so the real value is nicely inside our confidence area. But notice that our point estimate still differs from the real value that we know (we know because we created the world ourselves to have this value). Why? Because this random talent randomly messes with our estimate.

Lets now add one other factor – stength. Lets assume that men a stronger than women, and to keep it simple assume that if you are a man you can throw 5m more than if you are a woman with everything else being equal.

So the new result will now be:

[4] It is quite normal for statistical models to not work "out of the sample". This is usually not a problem, but pay attention to only predict something in the approximate range you have trained the model with.

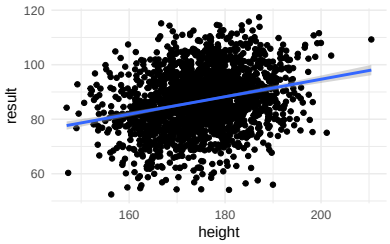
```
simdata$result <- 50 + 0.2*simdata$height + simdata$talent + 5*(simdata$gender=="Male")
```

Lets now run the first regression again. Remember – we have created the world in a way that the height has exactly the same effect as it had before.

```
regr1 <- lm(result ~ height, data=simdata)

summary(regr1)
```

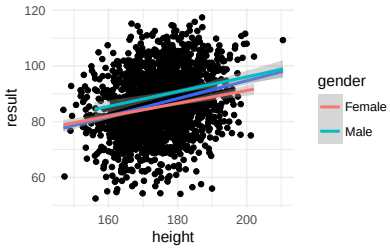
But the parameter for height is now much higher. Why? Because we have a missing variable problem. If we only know the height, this is still the best predictive model. But it will be a bad model for analysing the effect of gaining 10 cm of height. Graphically:

```
ggplot(simdata, aes(x=height, y=result))+
  geom_point()+
  geom_smooth(method="lm")+
  theme_minimal()
```



But remember, men were stonger, and they were also taller. The regression line if divided by sex would be flatter:

```
ggplot(simdata, aes(x=height, y=result))+
  geom_point()+
  geom_smooth(method="lm")+
  geom_smooth(method="lm", aes(color=gender))+
  theme_minimal()
```



And if we add this gender variable to the regression, we would get an estimate that is again nicely in our confidence bands.

```
regr2 <- lm(result ~ height+gender, data=simdata)
```

Time series

We will now take a look at working with time series data in R. We can manipulate data and time objects in objects of the `data.frame`.

Let us install first a new package, *tidyquant*, which brings financial analysis to the 'tidyverse'.

```
library(tidyquant)
library(tidyverse)
```

Let us download some stock prices

```
goog_prices  <- tq_get("GOOG", get = "stock.prices", from = " 1990-01-01") # from yahoo
aapl_prices  <- tq_get("AAPL", get = "stock.prices", from = " 1990-01-01")
```

date	open	high	low	close	volume	adjusted
1990-01-02	1.258929	1.339286	1.250000	1.330357	45799600	0.123853
1990-01-03	1.357143	1.357143	1.339286	1.339286	51998800	0.124684

```
head(aapl_prices)
```

date	open	high	low	close	volume	adjusted
2004-08-19	49.67690	51.69378	47.66995	49.84580	44994500	49.84580
2004-08-20	50.17863	54.18756	49.92529	53.80505	23005800	53.80505

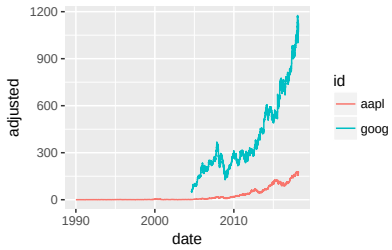
```
head(goog_prices, n=2)
```

We have already seen how to join two dataset

```
stockprices <-bind_rows("aapl"=aapl_prices, "goog"=goog_prices, .id = "id")
head(stockprices)
```

id	date	open	high	low	close	volume	adjusted
aapl	1990-01-02	1.258929	1.339286	1.250000	1.330357	45799600	0.123853
aapl	1990-01-03	1.357143	1.357143	1.339286	1.339286	51998800	0.124684

```
stockprices<- stockprices %>%
  mutate(date = ymd(date))

ggplot(stockprices, aes(x = date, y=adjusted))+
  geom_line(aes(color=id))
```

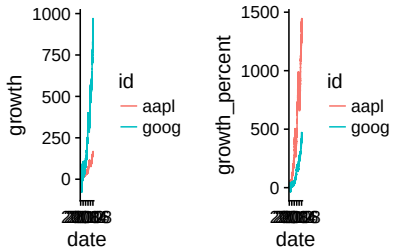


We want the first value to be the beginning of the time period. use `first()` function to get the first 'adjusted' stock price for each company and subtract it from each stock price. We can calculate the percent change since the beginning based on this absolute change values like below.

```
stockprices <-stockprices %>%
  filter(date >= today() - years(10)) %>%
  group_by(id) %>%
  arrange(date) %>%
  mutate(growth = adjusted - first(adjusted),
         growth_percent = (adjusted - first(adjusted))/first(adjusted)*100) %>%
  ungroup()
head(stockprices,n=2)

## # A tibble: 2 x 10
```

```
##   id    date        open  high   low close     volume adjusted growth
##   <chr> <date>     <dbl> <dbl> <dbl> <dbl>      <dbl>    <dbl>  <dbl>
## 1 aapl  2008-03-10  17.4  17.6  17.1  17.1 249897200.     11.6     0.
## 2 goog  2008-03-10 213.  214.  205.  205.   16079000.    205.      0.
## # ... with 1 more variable: growth_percent <dbl>
```

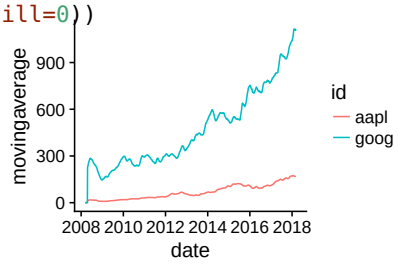
```
library(cowplot)
p1<-ggplot(stockprices, aes(x = date, y=growth))+
  geom_line(aes(color=id))
p2<-ggplot(stockprices, aes(x = date, y=growth_percent))+
  geom_line(aes(color=id))
plot_grid(p1,p2)
```



We can also caculate the moving/ rolling averages. For that install and load the package `RcppRoll`.

```
# install.packages("RcppRoll")
library(RcppRoll)
```

```
stockprices<-stockprices %>%
   group_by(id) %>%
  mutate(movingaverage = roll_mean(adjusted, 30, align="right", fill=0))
```




```
ggplot(stockprices, aes(x = date, y= movingaverage))+
  geom_line(aes(color=id))
```

Even though the `data.frame` object is one of the core objects to hold data in R, you'll find that it's not always efficient or possible to use `data.frames` when you're working with time series data.

We have seen in the last lectures already `data.frame` objects and `lists` objects.

R has many time series classes for example `ts`, `xts` or `zoo` to name just the most important ones.

Let us now do go through some basic R time series principles and how to apply them.

```
library(zoo)
library(xts)
```

The *base R* time series class is `ts`

- Create a univariate monthly time

```
ts1 <- ts(cumsum(1 + round(rnorm(750), 2)),
          start = c(1954, 1), frequency = 12)

str(ts1);
start(ts1);
```

```
end(ts1);
frequency(ts1)
time(ts1)

xts1<-as.xts(ts1)
head(coredata(xts1), n=2) # core data of the time series object

##      [,1]
## [1,] 1.57
## [2,] 1.22

str(xts1);
start(xts1);
end(xts1);
frequency(xts1)
time(xts1)
head(xts1)
tail(xts1)
```

Index

```
index(xts1)

indexClass(xts1)

## [1] "yearmon"
```

Replacing index class

```
indexClass(convertIndex(xts1,'POSIXct'))

## [1] "POSIXct" "POSIXt"
```

Get index class

```
indexTZ(xts1)

## [1] "UTC"
```

Change format of time display

```
indexFormat(xts1) <- "%Y-%m-%d"
```

Estimate frequency of observations

```
periodicity(xts1)

## Monthly periodicity from jaan  1954 to juuni 2016
```

Convert xts1 to yearly OHLC

```
to.yearly(xts1)
```

Convert xts1 to monthly OHLC

```
to.monthly(xts1)
```

Convert xts1 to quarterly OHLC

```
# to.quarterly(xts1) # this works but the time index is not "nice"
to.period(xts1,period="quarters")
```

Count the months in xts1

```
nmonths(xts1)
```

```
## [1] 750
```

Count the quarters in xts1

```
nquarters(xts1)
```

```
## [1] 250
```

Count the years in xts5

```
nyears(xts1)
```

```
## [1] 63
```

Selecting, Subsetting & Indexing

Select

```
jan2010 <- xts1["2010-01"] #Get value for March 1955
jan2010
```

```
##              [,1]
## 2010-01-01 666.96
```

```
xts1_2010 <- xts1["2010"]
xts1_2010
```

Extract data from Jan to April '10

```
xts1_janapril <- xts1["2010/2010-04"]
xts1_janapril
```

Get all data until March '10

```
xts1_janmarch <- xts1["/2010-03"]
xts1_janmarch
```

Lag, Lead and Diff operator

Lag and diff operator for data.frames

```
set.seed(123)
vec_1 <- runif(5) # random uniform
df_1<-cbind(lead = dplyr::lead(vec_1, 1), vec_1, lag = dplyr::lag(vec_1, 1))
df_1
```

```
##           lead     vec_1       lag
## [1,] 0.7883051 0.2875775        NA
## [2,] 0.4089769 0.7883051 0.2875775
## [3,] 0.8830174 0.4089769 0.7883051
## [4,] 0.9404673 0.8830174 0.4089769
## [5,]        NA 0.9404673 0.8830174
```

```
df_2<-cbind(lead = dplyr::lead(vec_1, 2), vec_1, lag = dplyr::lag(vec_1,2))
df_2
```

```
##           lead     vec_1       lag
## [1,] 0.4089769 0.2875775        NA
## [2,] 0.8830174 0.7883051        NA
## [3,] 0.9404673 0.4089769 0.2875775
## [4,]        NA 0.8830174 0.7883051
## [5,]        NA 0.9404673 0.4089769
```

Lag `xts`

```
xts2<-xts1["2016-01/"]
```

```
xts2
dplyr::lag(xts2)
stats::lag(xts2)
```

For a positive k, the series will shift the last value in time one period forward.

```
xts2_lag1<-lag(xts2,1)
xts2_lag2<-lag(xts2,2)
xts2_leadlag1<-merge(merge(xts2, xts2_lag1), xts2_lag2)
xts2_lead1<-stats::lag(xts2,-1)
xts2_lead2<-stats::lag(xts2,-2)
xts2_leadlag2<-merge(xts2_lead1, xts2_lead2)
xts2_leadlag <- merge(xts2_leadlag1, xts2_leadlag2)
xts2_leadlag
```

Lag ts and zoo

Zoo and ts implement the lag behaviour in a different, and for most economists confusing way. One is the direction of the lag for a given k. zoo uses a convention for the sign of k in which negative values indicate lags and positive values indicate leads. That is, in zoo lag(x, k = 1) will shift future values one step back in time. The second one is how *missing* values are handled, namely that no NA are shown.

```
ts2 <- as.ts(xts2)
ts2
```

```
##      Jan    Feb    Mar    Apr    May    Jun
## 1 749.87 749.23 749.49 750.34 753.20 753.60
```

```
ts2_lag1<-stats::lag(ts2,1)
ts2_lead1<-stats::lag(ts2,-1)
```

```
ts2_lag1
```

The “k=1” lag series ts2_lag1 starts one month earlier and there is no NA shown.

```
ts2_leadlag<-cbind(ts2,ts2_lag1)
ts2_leadlag<-cbind(ts2_lead1, ts2_leadlag)
```

```
ts2_leadlag
```

```
library(zoo)
zoo2<-as.zoo(xts2)
```

```
zoo2
```

```
str(zoo2)
```

```
## 'zoo' series from jaan  2016 to juuni 2016
##   Data: num [1:6, 1] 750 749 749 750 753 ...
##   Index: Class 'yearmon'  num [1:6] 2016 2016 2016 2016 2016 ...
```

```
zoo2_lag1<-stats::lag(zoo2,1) # LEAD!
zoo2_lead1<-stats::lag(zoo2,-1) # LAG!
zoo2_lag2<-stats::lag(zoo2,2) # LEAD!
zoo2_lead2<-stats::lag(zoo2,-2) # LAG!
```

```
zoo2_leadlag<-merge(merge(zoo2, zoo2_lag1), zoo2_lead1)
zoo2_leadlag
```

```
zoo2_leadlag2<-merge(merge(zoo2_leadlag, zoo2_lag2), zoo2_lead2)
zoo2_leadlag2
```

Diffs

Calculate a difference of a series using diff(). A single (or "first order") difference is to see it as $x_t - x_{t-k}$ where k is the number of lags to go back.

```
diff(xts2)
```

These are the same

```
diff(xts2, differences = 2)
diff(diff(xts2))
```

```
diff(xts2, lag = 4)
```

Linear regression

For fitting a linear model to the time series data we can use the function `dynlm()` from the `dynlm` package.

```
#install.packages("forecast")
#install.packages("wooldrige")
```

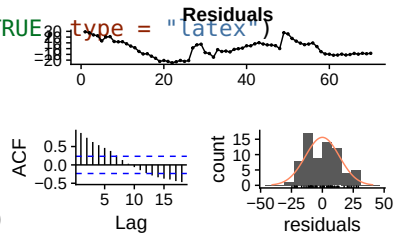
```
library(forecast)
library(wooldridge)
library(dynlm)
library(stargazer)
library(fpp)
```

```
tsdata<-ts(fertil3, start = 1913)
```

```
reg_dynlm <- dynlm(gfr ~ pe + L(pe) + L(pe, 2) +  ww2 + pill, data = tsdata)
reg_lm <- lm(gfr ~ pe + (pe_1) + (pe_2) +  ww2 + pill, data = tsdata)
```

```
stargazer(reg_dynlm, reg_lm,  title="Regression Results", align=TRUE, type = "latex")
```

```
checkresidual
```



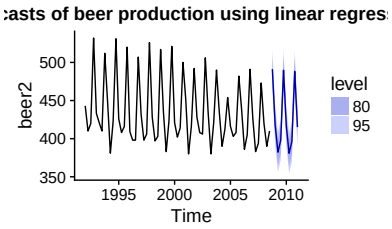

```
##
##  Breusch-Godfrey test for serial correlation of order up to 10
##
## data:  Residuals
## LM test = 55.091, df = 10, p-value = 3.036e-08
```

However, note that the `predict()` function of the `dynlm` package is **broken** with new data if lagged variables are used.

Forecasting with regression

One way to fit with regression is to use the `tslm()` function from the `forecast` package. We take the example directly from Rob Hyndmans time series book. The package `fpp` accompaning his book includes a data set on total quarterly beer production in Australia (in megalitres) from 1956:Q1 to 2008:Q3.

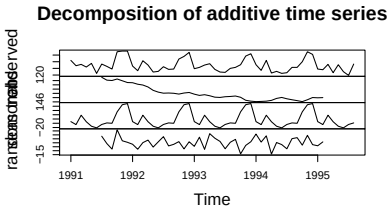
```
data(ausbeer)
beer2 <- window(ausbeer, start=1992)
fit.beer <- tslm(beer2 ~ trend + season) #  trend and seasonal dummies
fcast <- forecast(fit.beer)
autoplot(fcast) +
  ggtitle("Forecasts of beer production using linear regression")
```



Classical decomposition

The function `decompose()` can be used to decompose a time series into seasonal, trend and irregular components using moving averages. Deals with additive or multiplicative seasonal component.

```
decompose(beer)

plot(decompose(beer))
```



Seasonal (SEATS) decomposition

For using the X13 “SEATS“ algorithm, first install the `seasonal` package.

```
library(seasonal)

seas_airpass <-seas(AirPassengers)
```

The final function returns the adjusted series, the plot method shows a plot with the unadjusted and the adjusted series. The summary method allows you to display an overview of the model.

```
final(seas_airpass)
plot(seas_airpass)
summary(seas_airpass)
```

ARIMA models

Stationarity and differencing

```
series<- c("GDPCTPI") # estimate of chain link nominal gdp
```

```
# download data via FRED
gdpchain<-tq_get(series,                          # get selected symbols
            get="economic.data",                  # use FRED
            from="1954-01-01")                    # go from 1954 forward

str(gdpchain)
```

Aggregate monthly to quarterly and get lag

```
gdpchain %>%
  mutate(yq = as.yearqtr(date))  %>%
  rename(gdp = price)  %>%
  group_by(yq) %>%
  summarise(gdp = mean(gdp)) %>%
  mutate(loggdp = log(gdp)) -> gdpchain
```

Unit root tests

```
adf.test(gdpchain$gdp, alternative = "stationary")
adf.test(gdpchain$loggdp, alternative = "stationary")
adf.test(diff(gdpchain$loggdp, 4), alternative = "stationary")
```

ACF/PACF

acf

```
acf(gdpchain$gdp)

acf(diff(log(gdpchain$gdp),4))
```

pacf

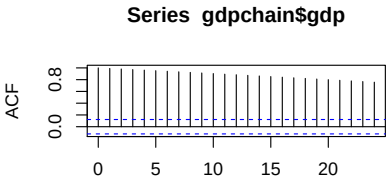
```
pacf(gdpchain$gdp)

pacf(diff(log(gdpchain$gdp),4))
```

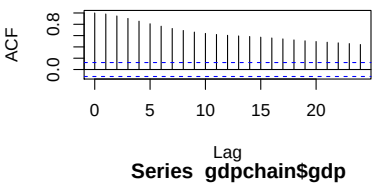
- Cross correlation between 2 timeseries.

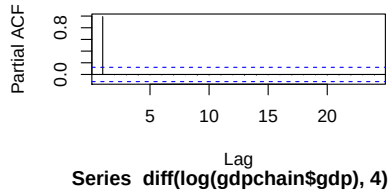
```
ccfRes <- ccf(mdeaths, fdeaths, ylab = "cross-correlation")
```

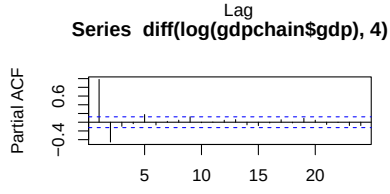
Autoregressive models

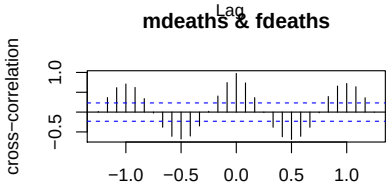
```
ar1<-arima.sim(list(order=c(1,0,0), ar=.9), n=100)
ar2<-arima.sim(list(order=c(1,0,0), ar=.5), n=100)

autoplot(ar1)

autoplot(ar2)
```

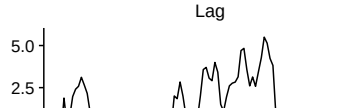












Recalling the regression

We took a quick look at the ordinary least squared (OLS) regression last time. Ordinary least squares (OLS) regression, which we used, assumes that the relationship between variables can be described as being approximately linear. You can transform the variables – take a logarithm or square them etc to model a bit more complex relationships. This simplest method works surprisingly well in many cases, and it has a benefit of being interpretable – you can really look at the regression model and understand what is going on. Moreover, sometimes this is the most important part – you want to see how much on the average the people with a degree in some field earn more than people with a degree in another field, controlling by some other variables. Regression finds you this answer.

Lets try to predict wages using **age**, education level (**edlevel3**), and numeracy score (**pvnum1**) from the `piaac` dataset. We are now predicting normal wages, not the logarithmed ones just for keeping it as simple to understand as possible (for modelling perspective logarithming would be much preferred – it would describe the data better).

First the data:

```
piaac <- read.csv("http://www.ut.ee/~iseppo/piaacest.csv")
```

Only leave the people with some income in:

```
library(dplyr)
piaac <- piaac %>%
  filter(!is.na(earnmth), !is.na(pvnum1))
```

We'll create the simplest model without any cross-effects

```
model.ols <- lm(earnmth~edlevel3+pvnum1, data=piaac)

summary(model.ols)
```

So we have our model. Lets add the predicted values back to the dataset using a function called `predict()`. `predict()` needs to know the model it will be using and the data for which to predict the stuff:

```
piaac$predict.ols <- predict(model.ols, newdata=piaac)
```

You now have a new variable in the dataset piaac – `predict.ols` which shows what did the model predict. You can compare it to the actual value and see yourself how good the model is, where does it miss it mark the most. Usually you find the difference between the actual and predicted value and then try to see, if any other variable could explain this difference.

Lets graph it.

Your turn:

- create a graph with `pvnum1` on x axis and `earnmth` mapped to y-axis. Let ggplot add the geom_smooth to it, but tell it to use `method="lm"`, so it will draw a linear approximation. Use `edlevel3` as grouping variable (differentiate the education levels by color for example).
- create the same graph but now with predicted value mapped to y-axis.

Notice the difference – in our model the education level lines are rising parallelly. This is of course because in the model we said that every point of increase in numeracy should add some cents to the salary. But it was the same amount of euros for every education level, this is what defines the rise of this line (there is nothing in the model saying that for different education levels it can be different). In fact it seems to depend on it – for people with higher education the numeracy seems to add more to the wage. To correct the model, we would need to add a cross effect (notice the $*$ instead of + in the model):

```
model.ols2 <- lm(earnmth~edlevel3*pvnum1, data=piaac)
```

This would now give us exactly the same result as the ggplots internal linear model.

Machine learning

Sometimes you only care about the ability to predict. You have a number of parameters of a javelin thrower and you want to know how far he could throw. You have a lot of information about your consumer, you want to know if he is going to react to your marketing campaign or if he is going to leave soon etc.

In this case regression has some shortcomings:

a) it will not work for truely non-linear stuff
b) you need to specify the expected model beforehand by yourself (you have to explicitly say if you expect there to be some cross-effects or if you expect some variable to enter in quadratic form etc)

Thanks to the advances in computing power we now have powerful machine learning algorithms. It is way beyond this course to understand the may families of them, but luckily it is not in fact required anymore for a lay-user. In most of the cases we can treat them as black boxes and just evaluate them and choose between them considering their actual performance[5]. One thing you will need is the ability to evaluate the performance of different models.

Running a machine learning model on the data is as easy as running a regression. Or easier. You have to be a bit more careful. In case of regression

[5] I am not saying that knowing them is a bad thing – it might help you to understand the limitations etc or to be able to choose an approach which has the best chance of success, but they are getting really good nowadays. If you have enough data it is perfectly possible to use them without knowing what is going on inside the black box.

you can see immidiately if the results look off for some reason. Some parameter has different value than you would expect – you will see it looking at the regression coefficients and start searching for what is wrong. In regression you have some idea how the world works and try to find the correct parameters from the data. You specify the model youself. In case of black box models you let the computer choose the model that seems to fit to the data. Sometimes the rules the computer comes up with are absolutely uninterpretable. So it is extremely important to have a separate data set (called test set) to test the performance of your model after you have trained it on what is usually called the training set.

The machine learning methods usually need a lot of data. You can run regressions on rather small datasets – 50, 100 cases. The results will probably have very high uncertainty levels but you will get something. Machine learning usually needs a lot more – at least thousands of cases. This is not a problem, as every company tends to generate and gather a lot of data nowadays. On the other hand it is important to understand that predictive models are not some kind of magic devices. Just having a lot of data does not mean that there is information in this data. I can have as detailed data as possible about the weather in Sweden but it is highly unlikely that it will help me in predicting who will win the Wimbledon next year. The problem is that if you are not testing your model well enough it might very very easily seem to you that you have developed a model which does just this. You will be able to train a model which gives you a correct answer of every single Wimbeldon winner in the history using just the detailed weather data in Sweden. And still I would guarantee that it will not help you next year.

You'll now need to install two packages for: `caret` and `randomForest`. `caret` is a package that provides a standardized interface for a number of other packages dealing with predictive modelling.

First lets create a training and test set. For this we will divide the dataset first with `createDataPartition()`. This creates two balanced sets so that the outcome (in our case the `earnmth` variable) will be represented in both groups approximately similarly[6].

```
library(caret)
# create the indices
set.seed(2017)
rowsToTrain <- createDataPartition(piaac$earnmth, p=0.80, list=FALSE)
# divide the piaac dataset to the test and training set
trainingset <- piaac[rowsToTrain,]
testset <- piaac[-rowsToTrain,]
```

And lets run the model[7]. The model tries to avoid overfitting and will adjust its parameters automathically. We need to tell it which method to use – I am using here a 10-fold cross-validation. It randomly distributes the data to ten parts, trains on 9 and tests on one. Then takes another 9, trains with them and

[6] Otherwise we might end up with a dataset with very few high wages in the training set or vice versa – very few high ones in the test set. This is usually not a big problem for continuous data, but it would be a problem if the outcome would be for example Has AIDS/Does not have AIDS, where one of the groups is very small and can thus be easily overrepresented in one of the sets. For continuous variable it divides the data by default to five quintiles (you can change this) and then samples randomly from these quintiles.

[7] I have set the parameters to something which should be computable before the end of the lecture. In reality you would probably want to have a higher `ntree` value (by default it should be 500), and also much more variables included – do not worry it only uses the ones it finds to be helpful.

tests on the remaining one, until it finds parameters that seem good enough.

```
set.seed(2017)
traincontrol=trainControl(method="cv", number=10)
model.rf <- train(earnmth ~ edlevel3 + pvnum1, data=trainingset,
                  method="rf", ntree=50, trControl=traincontrol)
```

```
## note: only 2 unique complexity parameters in default grid. Truncating the grid to 2 .
```

Now we should check how well the model functions. Predict the values for testset. Exactly as we would have done for normal regression – using `predict()` with a `newdata=testset` parameter.

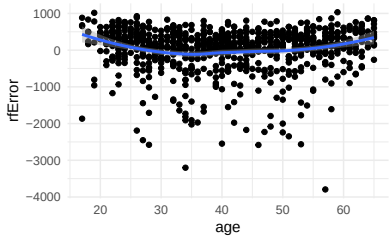
```
testset$rfPrediction <- predict(model.rf, newdata=testset)
```

Your turn

- Find the average absolute error in the test set – how different is the predicted value from the real earnmth value on average?

I think the answer should be 426.09 euros. So if you predict someone wage only from their mathematical abilities and level of education you would expect to be wrong by this much.

- Plot the errors against age variable. I get something like is in the margin. What do you think – would adding age make this model better?



We were actually minimising squares of error, so:

```
mean(testset$rfError^2)
```

```
## [1] 372826.8
```

This is now what tells you how good or bad the model is on a data that has not been used for training the model. In a good day this shows you what to expect from the future data (but be a bit sceptical here as well – this data has been collected at one specific time point etc, maybe the guy collecting the data was drunk, things may change in the future or be different in other datasets.).

How do you know if this is a good or bad model? First of all – if you face a real problem, then you probably know how much can you afford to be wrong. You might not want to target average squared errors, you might want to look at how often would you be wrong by more than 300€ or smth like this. Usually you would have a previous model to compare it to.

Your turn

- We have the regression model. Test it: run it first on the training set using `earnmth ~ edlevel3+pvnum1` as formula (we were previously running it on full data).
- then predict the values of the test set and find the average absolute error.

I think it will be better than our fancy model. The reason being that relationships in this data are rather nicely linear. And this happens. But if there would have been some nonlinearities in it random forest would have definitely outperformed.

Lets now take a look at the model random forest proposed us. One way is to predict the model on every point we have in the data and then graph it (as we did at the beginning for the regression model). But there is a more elegant solution (that also works in cases we would have some ranges missing in the initial dataset). To create newdata so, that we would have all the important parts covered.

We would thus need a new data frame, which consists of ages, education levels and numeracy levels and all the realistic combinations. For this we can use a nice command named `expand.grid()`. It creates exactly the combination of variables we need.

Lets try it:

```
expand.grid(pvnum1=c(100:101), edlevel3=levels(piaac$edlevel3))
```

```
##   pvnum1 edlevel3
## 1    100     High
## 2    101     High
## 3    100      Low
## 4    101      Low
## 5    100   Medium
## 6    101   Medium
```

Lets create a new dataset which have pvnum1 from 65 to 421 (this is the range in the data: check it with `range(piaac$pvnum1)`), and all the education levels.

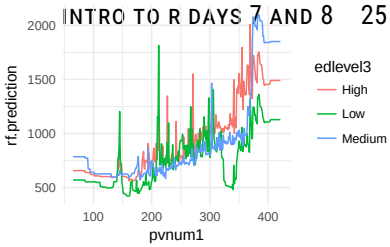
```
demodata <- expand.grid(edlevel3=levels(piaac$edlevel3), pvnum1=c(65:421))
```

And now lets use this as input for our model to graph it:

```
demodata$rf.prediction <- predict(model.rf, newdata=demodata)
```

And lets graph it:

```
ggplot(demodata, aes(x=pvnum1, y=rf.prediction))+
  geom_line(aes(color=edlevel3))+
  theme_minimal()
```



Cant get more nonlinear than this :), lets just say we now understand why it underperformed the regular OLS. It is obviously overfitting.

Your turn

- Create the same graph for our OLS model with the same demo data.