Introduction to R – final week

Indrek Seppo

2020-12-15

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

Modelling data in R

Lets start with reading the `piaac` dataset in again:

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

Most of the modelling in R is done in very similar way, so the tools you learn are applicable for most of the modelling.

The formula interface in R has in general the following shape:

```
y ~ a + b
```

where intercept will be added automatically. If you want to model interactions, add them with `*`. If you want to do some additional arithmetics, do them in `I()`:

```
y = a*b + I(a^2)
```

Or you can specify polynomials like this:

```
y = a + poly(b, 2)
```

Lets try to predict wages using education level (**Education**), and numeracy score (**numeracy**) 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 very much preferred – it would describe the data better and would even fit the theory better).

Only leave the people with some income in (this is a crucial step, some of the later steps will not work without this):

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

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

```
model.ols <- lm(Income ~ Education + Numeracy, 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.

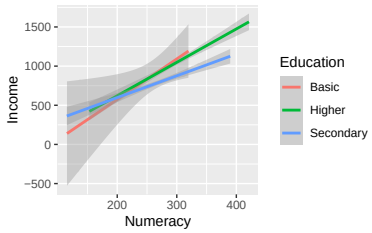
Lets graph it. Let me remind you the basics of ggplot. A typical ggplot graph would look something like this:

```
ggplot(data=mydataset, aes(x=firstvariable, y=secondvariable))+
  geom_something(aes(color=groupingvariable))+
  geom_otherthing(size=4, aes(x=thirdvariable))+
  facet_wrap(~othergroupingvariable)+
  ggtitle("My fantastic graph")+
  theme_minimal()
```

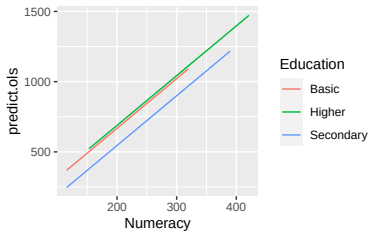
If you want something at the graph to be connected to your data, you will need to put it **into the aes()-call**. If you want to set the property yourself (like the `size=4` in this example), it has to be **outside of the aes()-call**.

Your turn:

- create a graph with `Numeracy` on x axis and `Income` 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 `Education` as grouping variable (differentiate the education levels by color for example).



- create the same graph but now with predicted value mapped to y-axis. I think the predicted value was called `predict.ols` (you can use `geom_line()` or `geom_smooth(method="lm")`, doesn't matter in this case.



If you would want to have the model's confidence intervals (showing for each person what is the uncertainty around expected or average value of similar persons), then you could have added `interval="confidence"` to the `predict()` call. It would also allow you to create prediction intervals with

`interval = "prediction"`, showing where would the model expect for 95% of cases of similar persons to lie in, but be very careful with it – your model has to be very close to reality for it to give you something valuable; always check how well it performs on your testdata before applying in real world.

```
piaac <- bind_cols(piaac, as.data.frame(predict(model.ols, newdata=piaac, interval = "confidence")))
```

You would now see that there are now three variables created into the dataset: `fit`, `lwr` and `upr`[1].

If you look back at the last two graphs you will notice a difference – in our model the education level lines are rising in parallel. This is of course because in the model we said that every point of increase in numeracy should add some cents to the salary (this is what defines how quickly the lines rise). 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 – to better model the data we are describing with the model, we would need to add a cross effect (notice the `*` instead of `+` in the model):

[1] Notice I had to explicitly convert the result from `predict()` to data.frame, before binding it to the piaac dataset.

```
model.ols2 <- lm(Income~Education*Numeracy, data=piaac)
```

This would now give us exactly the same result as the ggplot's internal linear model.

Couple of additional tools into your tool set. There is a great library called `broom`, which helps you to quickly get the most important parameters from the model:

```
library(broom)
tidy(model.ols2)
```

It will also work instead of `predict()`. You can use `augment()`:

```
library(broom)
piaac.modelols <- augment(model.ols, data=piaac)
```

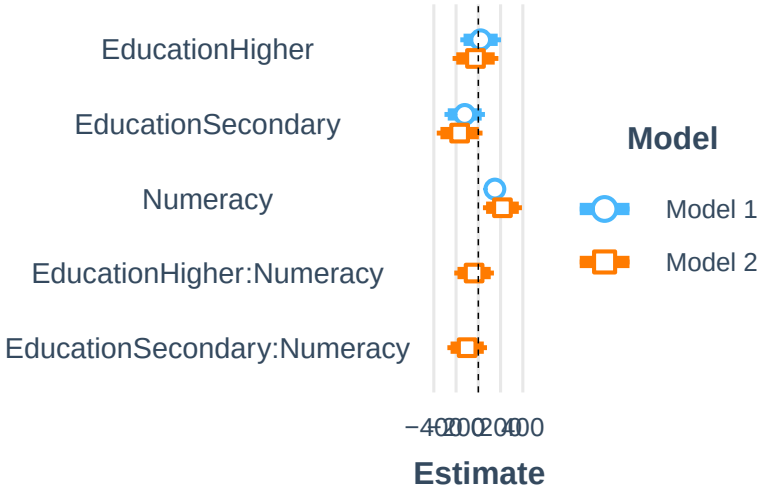
This will create you new values in the dataset like `.fitted` and `.resid` for fitted values and residuals.

Then there was a package called `jtools` (but there are more, like stargazer), which helps you to create some tables of the regressions etc. Let's install `jtools`, `huxtable` and `ggstance`, which jtools needs for some of it's functionality.

```
library(jtools)
library(huxtable)
export_summs(model.ols, model.ols2, scale = TRUE,
             error_format = "[{conf.low}, {conf.high}]", to.file="html")
```

And you can also visualize the coefficients as forestplots:

```
library(jtools)
plot_summs(model.ols, model.ols2, scale = TRUE, inner_ci_level = .9)
```



Sometimes you want to analyze some binary outcomes – like if somebody is employed or not.

You can of course do logits and probits in R (without setting the link="logit" it would do a probit) to analyze binary outcomes:

```
library(dplyr)
piaac.empl <- read.csv("piaacENG.csv")

piaac.empl <- filter(piaac.empl, !is.na(empl_status))
piaac.empl$isemployed <- piaac.empl$empl_status=="Employed"
model.logit <- glm( isemployed~Education + gender + Numeracy, family=binomial(link='logit'), data=piaac
```

```
summary(model.logit)
```

What you usually want to do with logits-probits, is to give margins (a la Stata). This would be achieved by package `margins`:

```
library(margins)
margins.model <- margins(model.logit)
summary(margins.model)
```

```
##              factor    AME     SE      z      p   lower  upper
##     EducationHigher 0.1996 0.0440 4.5353 0.0000  0.1133 0.2859
##  EducationSecondary 0.0679 0.0437 1.5529 0.1204 -0.0178 0.1536
##          genderMale 0.0516 0.0105 4.9089 0.0000  0.0310 0.0723
##            Numeracy 0.0011 0.0001 9.1917 0.0000  0.0009 0.0014
```

It also offers plotting functions, check it out at https://github.com/leeper/margins.

You should also check out a package called `lime`.

Machine learning

Ordinary least squares (OLS) regression 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 will answer this kind of questions – it might not model some of the nonlinearities perfectly, but it gives you an average over the groups.

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 truly 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[2]. You will probably be expected to be able to evaluate the performance of different models proposed to you by your companies data scientist.

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 you can see immediately 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 yourself. 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

[2] 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.

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 `Income` variable) will be represented in both groups approximately similarly[3].

```
library(caret)
# create the indices
set.seed(2017)
rowsToTrain <- createDataPartition(piaac$Income, 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[4]. The model tries to avoid overfitting and will adjust its parameters automatically. 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 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(Income ~ Education + Numeracy, data=trainingset,
                  method="rf", ntree=50, trControl=traincontrol)
```

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

[3] 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 HIV/Does not have HIV, 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.

[4] 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.

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.

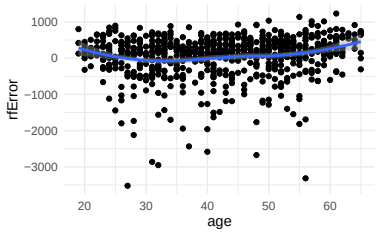
```
testset$rfPrediction <- predict(model.rf, newdata=testset)
```

Your turn

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

I think the answer should be 426.53 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 errors, so:

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

```
## [1] 348957
```

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 `Income ~ Education+Numeracy` 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 might even 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(Numeracy=c(100:101), Education=unique(piaac$Education))
```

```
##   Numeracy Education
## 1      100 Secondary
## 2      101 Secondary
## 3      100    Higher
## 4      101    Higher
## 5      100     Basic
## 6      101     Basic
```

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

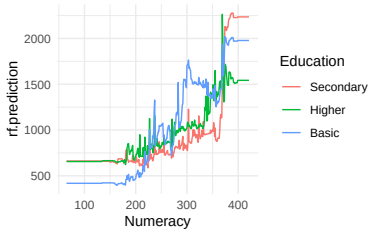
```
demodata <- expand.grid(Education=unique(piaac$Education), Numeracy=c(65:421))
```

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=Numeracy, y=rf.prediction))+
  geom_line(aes(color=Education))+
  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. You will need to predict from `model.ols` using the `demodata` and put it on the graph.