Introduction to OLS regression and summary statistics in R

Lex Comber and Paul Harris

15 May 2017

Contents

Overview

In this session, you will explore basic regression as a precursor to developing Geographically Weighted Regression (GWR) analyses (Brunsdon et al, 1996; Fotheringham et al, 2002) in the next session. The data are introduced along with some techniques for generating summary statistics and simple visualizations. This session will:

- Describe OLS regression;
- Introduce the Liudaogou data;
- Develop some exploratory visualizations of the data using histograms, boxplots, conditional boxplots;
- Apply OLS using the `lm` function in R;

- Refine the OLS models to identify the model with the best *fit*;
- Generate some spatial data and map the residuals in the model.

The data used in these workshops will be the sub-catchment soils data for the Liudaogou watershed, as described in Wang et al (2009). Along the way, you will be exposed to some of the graphics functions available in the `ggplot2` package in R.

We will take as an initial example the objective of developing a model of Total Phosphorus percentage.

Ordinary Least Squares Regression

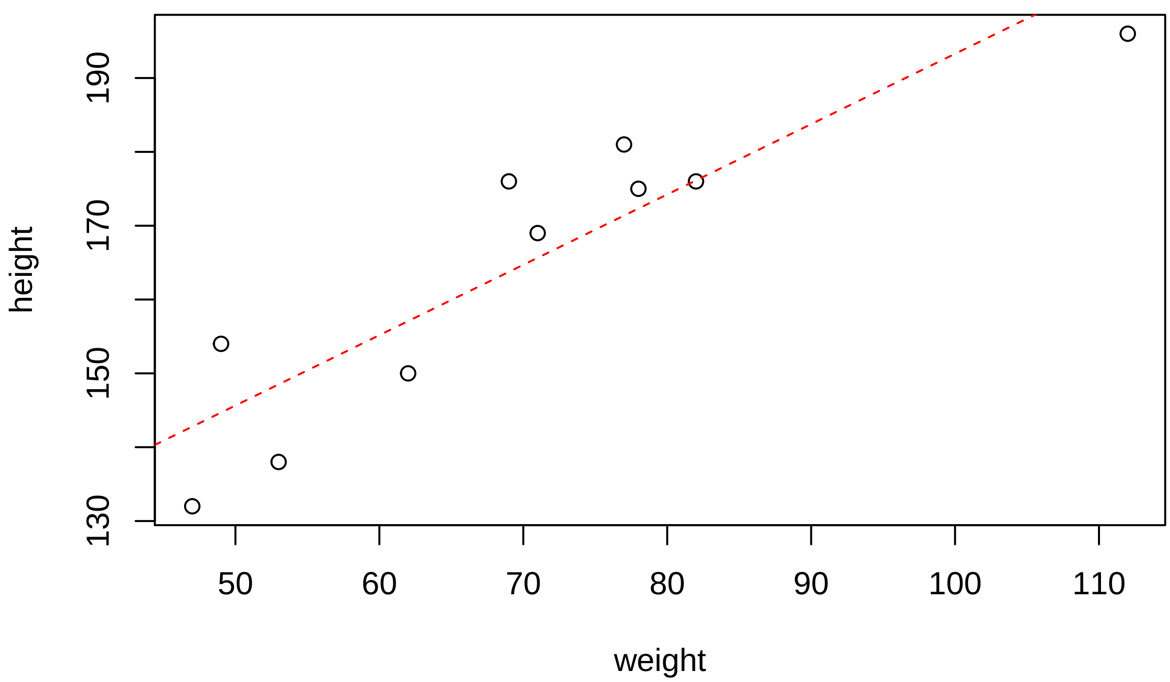
Ordinary Least Squares (OLS) or linear regression has the basic form of:

$$Y = \beta_0 + \beta_n X_n$$

where Y is the value that you are trying to model, fit or predict and X_n are the n variables or *covariates* that you are trying to predict Y from. Note that Y is referred to as the *dependent variable* and X as the independent variables. The intercept term is given by β_0 and the values of β_n describe the coefficient estimates, indicate the degree to which the changes in X are associated with changes in Y.

So consider the hypothetical example of the relationship between height and weight below. These can be plotted and a regression fitted that models the relationship between the 2 variables using the `lm` function:

```r
height <- c(176, 154, 138, 196, 132, 176, 181, 169, 150, 175)
weight <- c(82, 49, 53, 112, 47, 69, 77, 71, 62, 78)
plot(weight, height)
model1 <- lm(height~weight)
abline(model1, lty = 2, col = "red")
```



```r
summary(model1)
```

```
##
## Call:
## lm(formula = height ~ weight)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -10.786  -8.307   1.272   7.818  12.253
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  98.0054    11.7053   8.373 3.14e-05 ***
## weight        0.9528     0.1618   5.889 0.000366 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 9.358 on 8 degrees of freedom
## Multiple R-squared:  0.8126, Adjusted R-squared:  0.7891
## F-statistic: 34.68 on 1 and 8 DF,  p-value: 0.0003662
```

What the model describes is the relationship between height and weight. In this case each additional 1cm of height above a baseline of 98cm is associated with an additional weight of 0.95kg. We will develop these ideas in more detail in the sections below.

The Liudaogou Data

We can explore the application of `lm` to generate linear models using some example data. You will need to run the code below to download the Liudaogou watershed data from Lex's Github repository. However, as you may have gathered from your earlier introductions to R, much functionality is contained in R packages. To get the data from GitHub you will need to install the `repmis` package only before the **first** that you load it, and then load the package:

```r
install.packages("repmis", dep = T)
```

Then once it is loaded to your computer, it can be loaded into R / RStudio using the `library` function:

```r
library(repmis)
source_data("https://github.com/lexcomber/CAS_GW_Training/blob/master/Liudaogou.RData?raw=True")
```

Here the `source_data` function reads the `.RData` file from the GitHub repository. You should check what has been loaded:

```r
ls()
```

```
## [1] "data"   "height" "model1" "weight"
```

And then you should explore the data using some of the commands below:

```r
dim(data)
class(data)
data[1:5, ]
## you can veiw the full dataset
data
## or specific elements / columns
data$TPPC
data$TPPC[1:100]
data[1:10,6]
```

Reminder: our initial objective is to develop a model of Total Phosphorus percentage. This is the `TPPC` variable in `data`.

Exploratory data analysis

It possible to explore the data in a number of ways using simple plots, correlations and boxplots. Specifically, we are interested in correlations between numeric, continuous variables and conditional boxplot for ordinal variables. First examine the data to determine the nature of the different variables using the `names` and `summary` functions:

```r
names(data)
head(data)
summary (data)
```

Numeric, continuous data

We can examine the correlations using the `cor` function:

```r
round(cor(data[, -c(1:3, 17:19)]), 3)
```

```
##               TNPC   TPPC SOCgkg ClayPC SiltPC SandPC NO3Ngkg NH4Ngkg
## TNPC         1.000  0.206  0.323  0.197  0.265 -0.263   0.163   0.125
## TPPC         0.206  1.000  0.373  0.315  0.507 -0.492   0.153   0.226
## SOCgkg       0.323  0.373  1.000  0.253  0.298 -0.315   0.226   0.271
## ClayPC       0.197  0.315  0.253  1.000  0.735 -0.817   0.173   0.159
## SiltPC       0.265  0.507  0.298  0.735  1.000 -0.987   0.198   0.210
## SandPC      -0.263 -0.492 -0.315 -0.817 -0.987  1.000  -0.203  -0.215
## NO3Ngkg      0.163  0.153  0.226  0.173  0.198 -0.203   1.000   0.304
## NH4Ngkg      0.125  0.226  0.271  0.159  0.210 -0.215   0.304   1.000
## N2P          0.929 -0.016  0.261  0.174  0.205 -0.207   0.132   0.077
## CoveragePC   0.001 -0.035 -0.048 -0.026  0.038 -0.023   0.041  -0.077
## Slope       -0.014  0.054 -0.077 -0.029 -0.036  0.043  -0.079   0.008
## Aspect       0.080  0.116 -0.025  0.157  0.201 -0.193  -0.070   0.022
## Altitude_m   0.022 -0.136 -0.246  0.101  0.153 -0.135  -0.006  -0.079
##                N2P CoveragePC  Slope Aspect Altitude_m
## TNPC         0.929      0.001 -0.014  0.080      0.022
## TPPC        -0.016     -0.035  0.054  0.116     -0.136
## SOCgkg       0.261     -0.048 -0.077 -0.025     -0.246
## ClayPC       0.174     -0.026 -0.029  0.157      0.101
## SiltPC       0.205      0.038 -0.036  0.201      0.153
```

```
## SandPC      -0.207      -0.023  0.043 -0.193      -0.135
## NO3Ngkg      0.132       0.041 -0.079 -0.070      -0.006
## NH4Ngkg      0.077      -0.077  0.008  0.022      -0.079
## N2P          1.000       0.010 -0.026  0.069       0.067
## CoveragePC   0.010       1.000 -0.098  0.006       0.164
## Slope       -0.026      -0.098  1.000  0.336      -0.074
## Aspect       0.069       0.006  0.336  1.000       0.091
## Altitude_m   0.067       0.164 -0.074  0.091       1.000
```

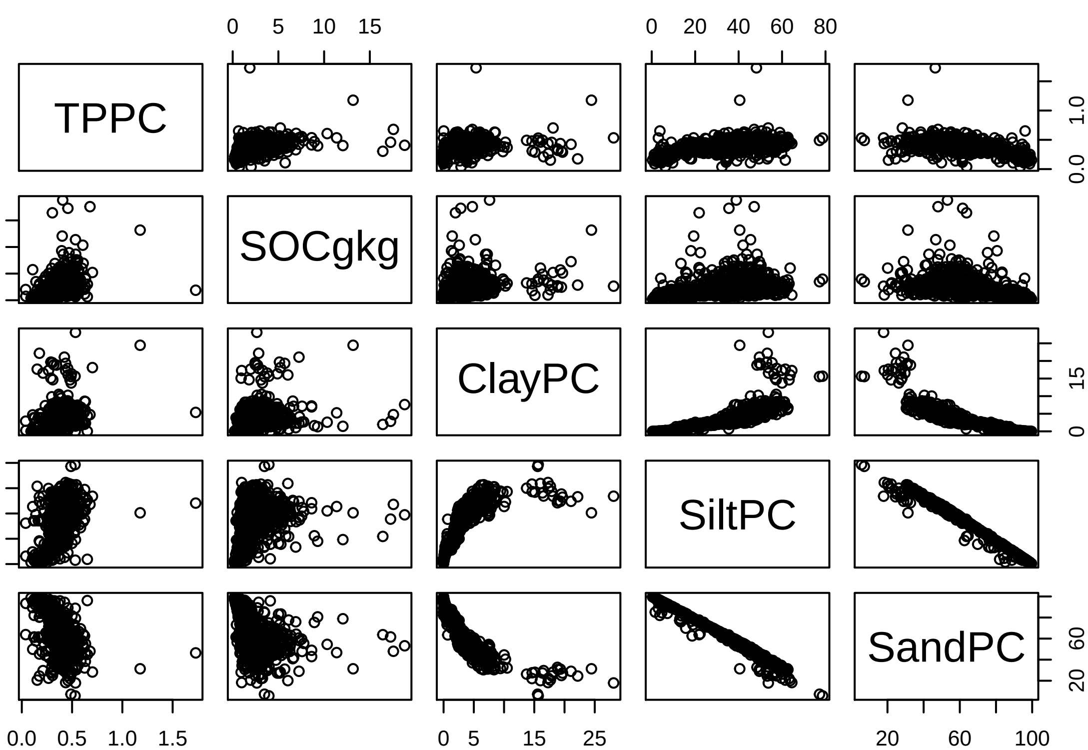
Notice that the `-c(1:3, 17:19)` was used to omit some data columns because they are not numeric. The `round` function was used to limit the number of decimal places of the output. The `names(data)` command can help decide which variables to consider. And this can be refined to consider just correlations with the TPPC variable

```
cor(data[, -c(1:3, 17:19)])[2,-(1:2)]
```

```
##      SOCgkg      ClayPC      SiltPC      SandPC     NO3Ngkg     NH4Ngkg
##  0.37295038  0.31478474  0.50747990 -0.49175723  0.15334713  0.22569414
##         N2P  CoveragePC       Slope      Aspect  Altitude_m
## -0.01569446 -0.03488437  0.05360494  0.11623834 -0.13623987
```

It is evident that the variables for `SOCgkg`, `ClayPC`, `SiltPC` and `SandPC` are all positively correlated with `TPPC`. We can examine these in further detail:

```
plot(data[,5:9])
```




```
cor(data[, 5:9])[2,-(1:2)]
```

```
##     ClayPC     SiltPC     SandPC
##  0.2532472  0.2978149 -0.3148977
```

The `plot` functions plots all of the variables against each other and is useful for displaying how all variables correlate. Can you work out how the last command of `cor(data[, 5:9])[2,-(1:2)]` is working and critically what it is showing? The `5:9` was used to select just the 5th to 9th data columns.

In fact, `ClayPC`, `SiltPC` and `SandPC` all sum to 100 which can cause problems. For this reason Clay was removed from further analysis.

We can also generate some nicer plots using the `ggplot` function included as part of the `tidyverse` package. You should install and load this now if you have not already installed this:

```r
install.packages("tidyverse", dep = T)
```

We will also need some other packages. These will be used in later sessions as well.

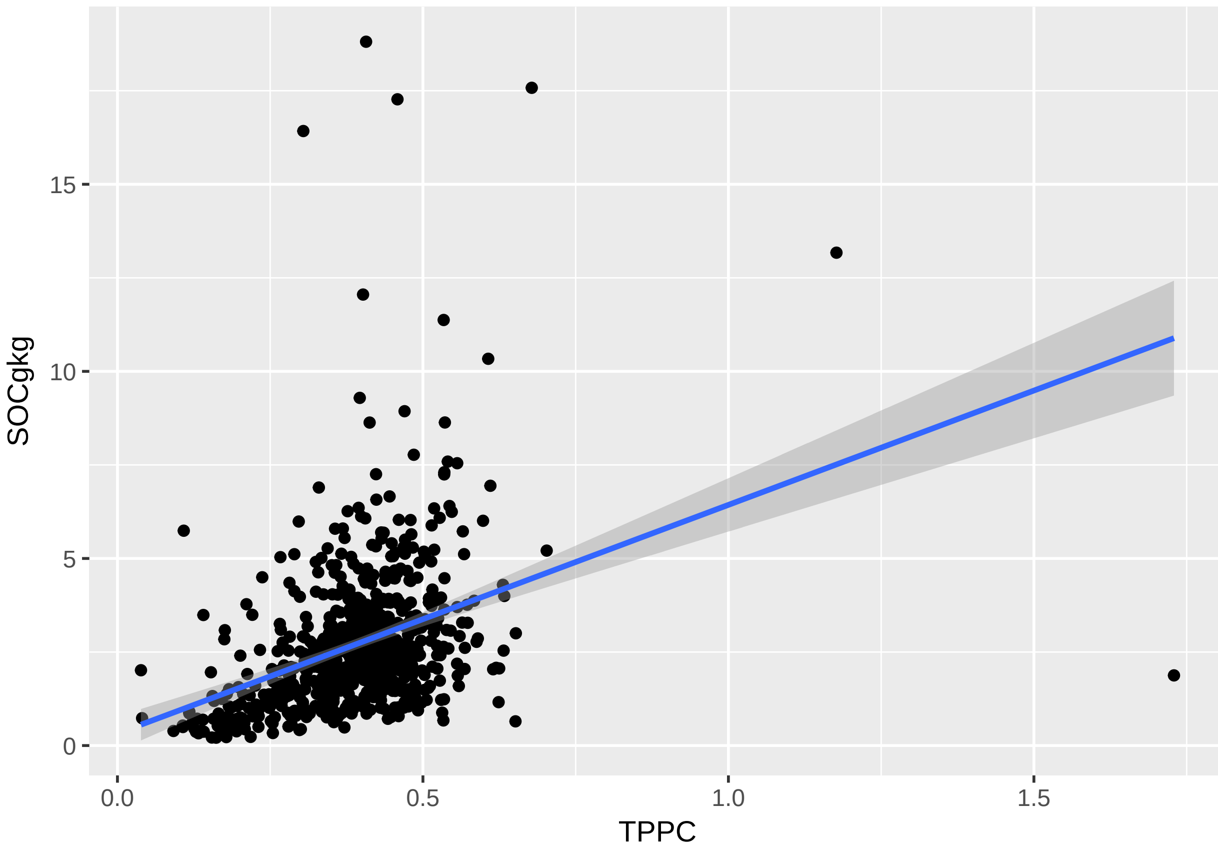
```r
install.packages("GISTools", dep = T)
install.packages("plyr", dep = T)
```

Using `ggplot` involves a bit of a learning curve at first but there is plenty of help and advice on the internet. For example the following sites may be useful:

- http://ggplot2.tidyverse.org/reference/
- http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/
- https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf
- http://tutorials.iq.harvard.edu/R/Rgraphics/Rgraphics.html … We should all strive to be part of the `tidyverse`!: http://tidyverse.org

We can use the extensive `ggplot` parameters to plot the data in different ways. The code below plots the data as points, and fits a regression line through the 'TPPC`and`SOCgkg' variables:

```r
ggplot(data,aes(TPPC,SOCgkg))+geom_point() + geom_smooth(method='lm')
```



TASK: can you modify the code above to produce plots for the `TPPC` against `SiltPC` and `SandPC`?

Boxplots and conditional boxplots;

Having loaded `tidyverse` we can use some of the functions it contains including the `tibble` data structure:

```
as_tibble(data)
```

```
## # A tibble: 689 × 19
##        ID Latitude Longitude       TNPC      TPPC    SOCgkg   ClayPC
##    <fctr>    <dbl>     <dbl>      <dbl>     <dbl>     <dbl>    <dbl>
## 1  WYQ001  4300830  705431.9 0.10938171 0.3157755  2.779208 0.816840
## 2  WYQ002  4300752  705462.6 0.09225275 0.2768932  1.613443 0.110693
## 3  WYQ003  4300652  705451.5 0.25516038 0.3966375  9.291738 1.310971
## 4  WYQ004  4300546  705439.6 0.15497834 0.2896334  4.126008 1.313521
## 5  WYQ005  4300465  705444.1 0.20280727 0.3561499  5.795655 2.151067
## 6  WYQ006  4300370  705484.8 0.10995866 0.2529443  2.043729 0.335551
## 7  WYQ007  4300259  705463.4 0.20154837 0.3296626  6.895536 1.046108
## 8  WYQ008  4300168  705468.8 0.35927889 0.3042410 16.423025 1.957946
## 9  WYQ009  4300045  705500.1 0.18379566 0.3249357  4.111666 0.021057
## 10 WYQ010  4300060  705555.3 0.77681312 0.4583591 17.270962 2.840421
## # ... with 679 more rows, and 12 more variables: SiltPC <dbl>,
## #   SandPC <dbl>, NO3Ngkg <dbl>, NH4Ngkg <dbl>, N2P <dbl>,
## #   CoveragePC <int>, Slope <int>, Aspect <int>, Altitude_m <dbl>,
## #   SoilType <chr>, LandUse <chr>, Position <fctr>
```

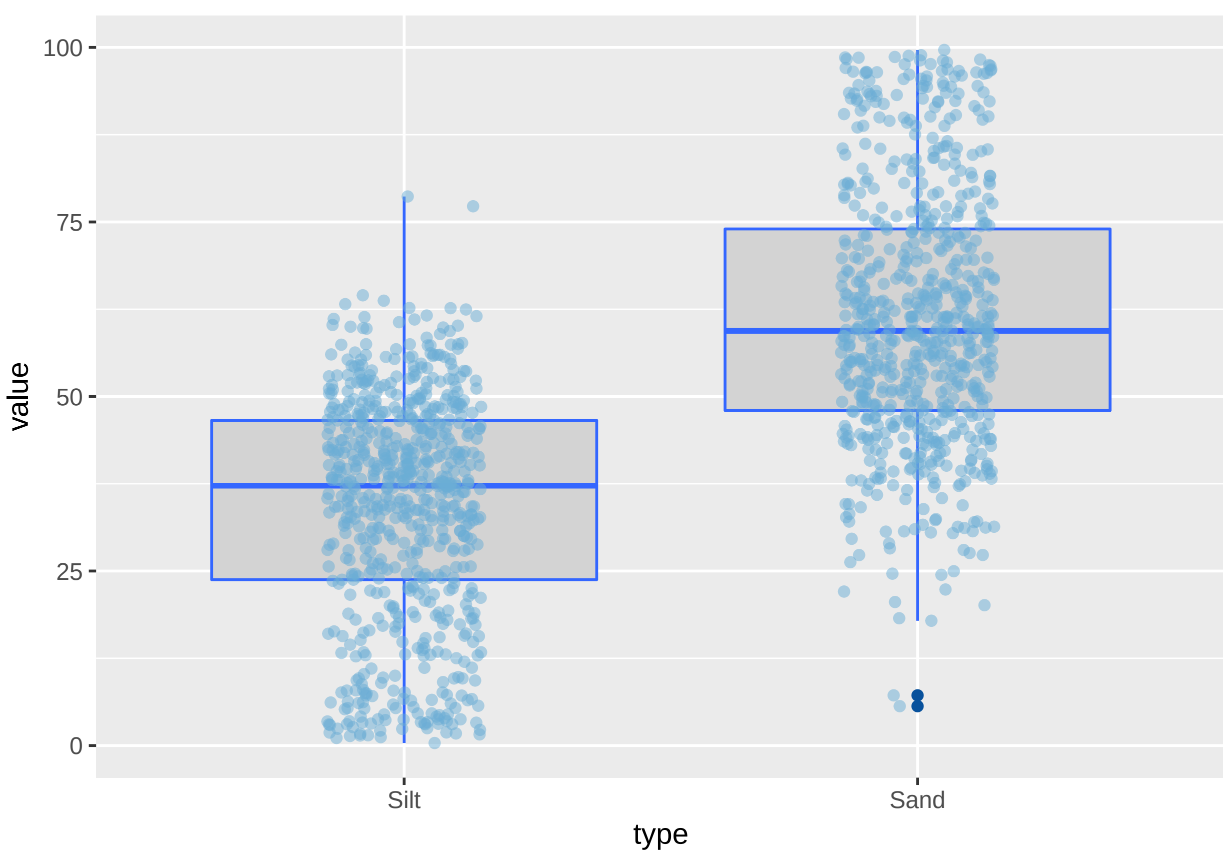
What is interesting here is that when `tibble` data structures are called, they automatically display the first 10 rows of data and however many columns will fit. Interestingly they describe the type of each variable in the `tibble` format of the `data.frame`. So we can see that we have a number of non-numeric character and factor variables (`chr` and `fctr`) including `SoilType`, `LandUse` and `Position`. It would be interesting to explore how these variables are related to `TPPC`. Conditional boxplots can help do this.

First lets start with a standard set of boxplots to display the distributions of the **numeric** data we are interested in. A standard function for a boxplot is below, but this is not very beautiful - try the code yourself:

```
boxplot(data[,8:9], outline = F)
```

Boxplots can be created using `ggplot` with a small amount of data manipulation:

```
# create a data frame
a = data.frame(type = "Silt", value = data[,8])
b = data.frame(type = "Sand", value = data[,9])
df <- rbind(a,b)
# plot with ggplot and geom_boxplot()
ggplot(df, aes(type, value)) +
    geom_boxplot(fill = "lightgrey", colour = "#3366FF", outlier.colour = "#08519C") +
    geom_jitter(width = 0.15, colour = "#6BAED680")
```

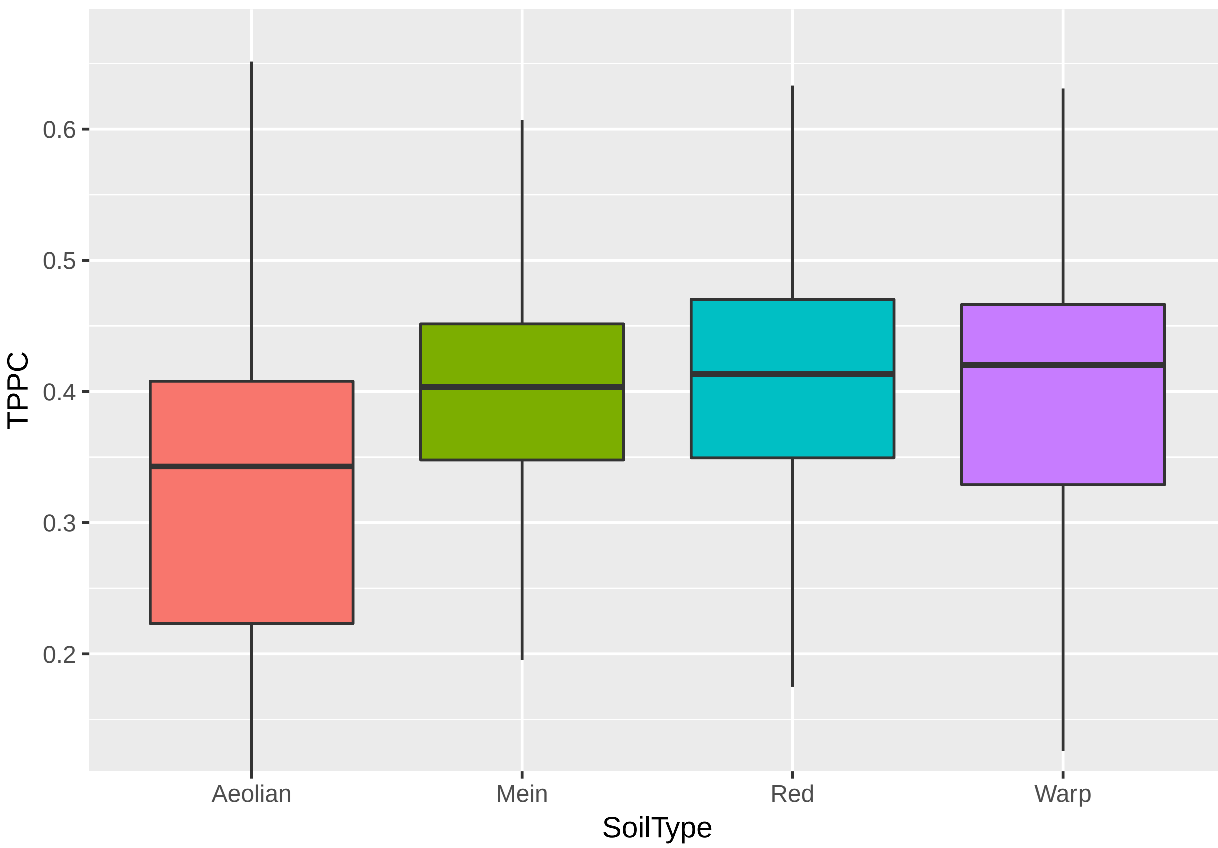



```
# see http://ggplot2.tidyverse.org/reference/geom_boxplot.html
# note the transparent blue colour was identified from
# add.alpha(brewer.pal(5, "Blues"), 0.5)
```

This shows the distributions of the sand and silt percentages, which are are on the same scale (0-100).

Next we can use *Conditional* boxplots to examine the distribution of TPPC against the categorical variables such as `SoilType`.

```
## identify the outliers to set Y limits
ylim1 = boxplot.stats(data$TPPC)$stats[c(1, 5)]
## then plot
ggplot(data, aes(SoilType, TPPC, fill = SoilType)) +
    geom_boxplot(outlier.shape = NA) +
    coord_cartesian(ylim = ylim1*1.05) +
    theme(legend.position = "none")
```




```
## the conventional box plot code is below
# boxplot(data$TPPC~data[,19], las=1,
#       outline = F, ylab = "", xlab = "",
#       col = brewer.pal(length(unique(data[,19])), "Spectral"))
```

TASK: you should use boxplots to determine whether there are potentially important differences in Phosphorus percentages and distributions on different soils, land uses and positions. You may find is useful to examine the help for boxplots in `ggplot` see http://ggplot2.tidyverse.org/reference/geom_boxplot.html.

Regression with `lm`

Here we will start with a simple regression, with the objective of constructing a model of Phosphorus percentage, `TPPC` from `SOCgkg`, `ClayPC`, `SiltPC` and `SandPC` (soil organic carbon in g/kg, percentages of clay, silt and sand):

```
m1 <- lm(TPPC ~ SOCgkg + SiltPC + SandPC, data = data)
```

So in the linear models fitted above, the response is TPPC and the predictors are soil organic carbon, clay, silt and sand, plus an intercept term, which is included by default. But essentially this seeks to construct a model of TPPC from the predictor variables.

The model can be examined using the `summary` function:

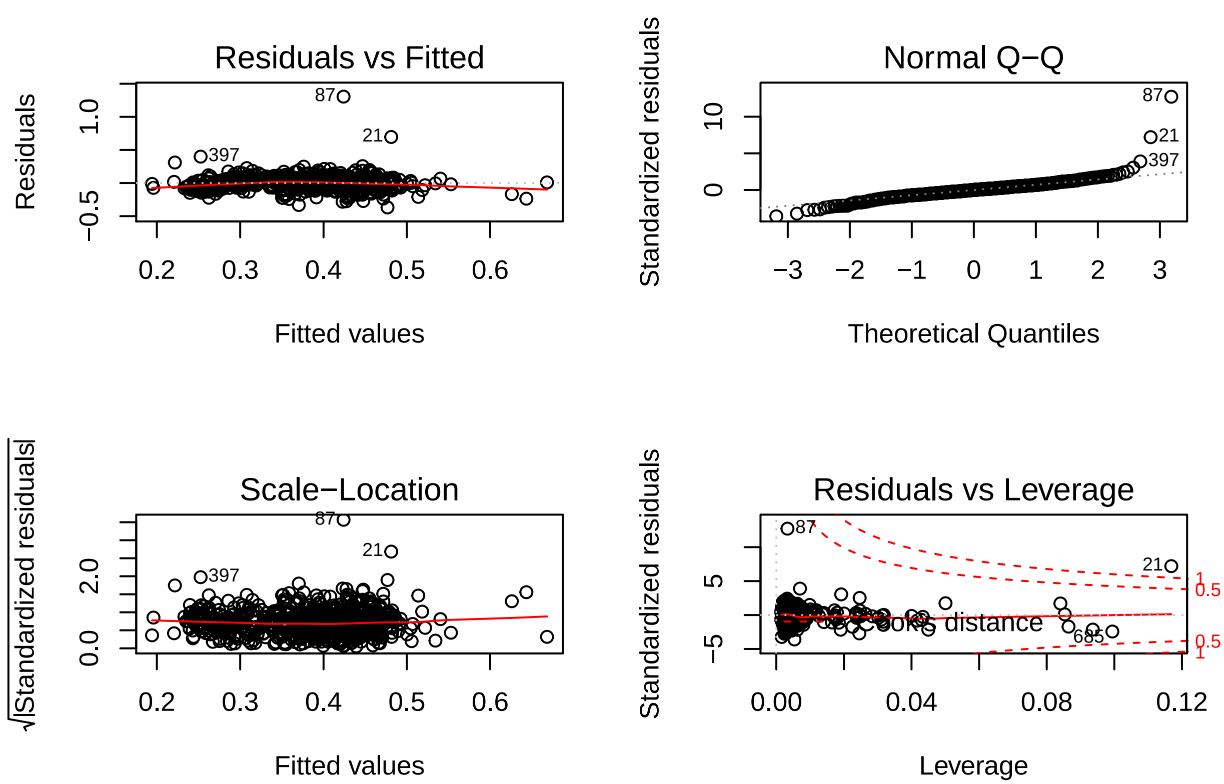
```
summary(m1)
```

This shows a number of things: - the Residuals indicates the empirical quantiles of the residuals. - Estimates for the coefficients i.e. components of β - the standard errors of the estimates - the values of the t-statistic - the probability that a random variable $T \sim t_{n-p}$ is such that $|T|$ exceeds the absolute value of the t-statistic. These latter probabilities are thus p-values for testing the null hypotheses $\beta_{j=0}$ against the alternatives $\beta_{j\neq 0}$ in the normal linear model.

In this case, we can say something about the coefficient estimates that were found to significant (i.e. have associations with the response variable that are unlikely to have occurred by chance): - an increase in 1 of SOC (i.e. an increase of 1 gramme per kilogramme of Carbon) is significantly associated with a 0.015 increase in TPPC; - an increase of 1 of ClayPC is significantly associated with a decrease of 0.0037 of TPPC.

The`plot` function can be used to generate a number of useful diagnostic plots (see `plot.lm` for further details). The `par` function is used to set plotting parameters. Here we set up the display so that four plots are produced on the same screen, saving the old parameters in `old_par`, reinstating the old parameters after the plot:

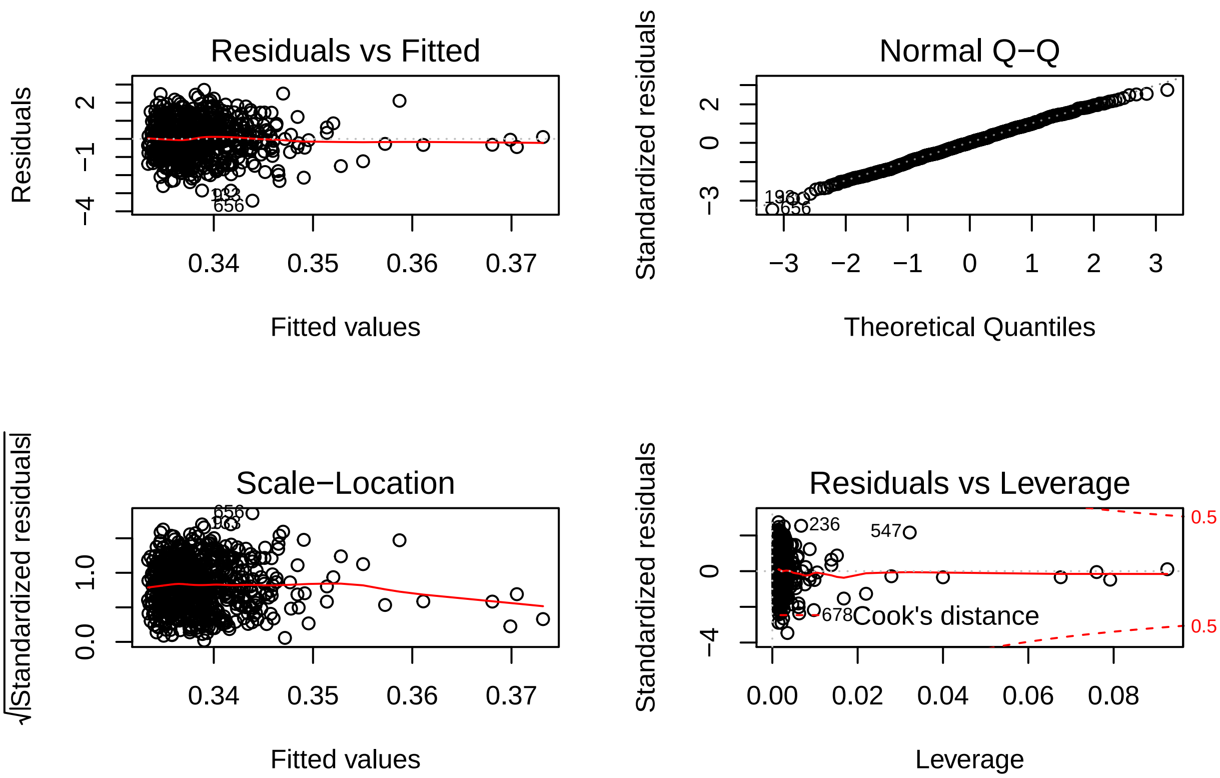
```
old_par <- par(mfrow = c(2, 2))
plot(m1)
```



```r
par(old_par)
```

What should we expect these plots to look like if the all the assumptions for the normal linear model held? One thing we can do is the following to examine the assumptions for covariates: generates a random distribution as a slight error term, combines this with the fitted values from the regression model and compares these against the actual input data values of the covariate.

```r
old_par <- par(mfrow = c(2, 2))
plot(lm(rnorm(length(data$SOCgkg)) + fitted.values(m1) ~ data$SOCgkg))
```




```r
par(old_par)
```

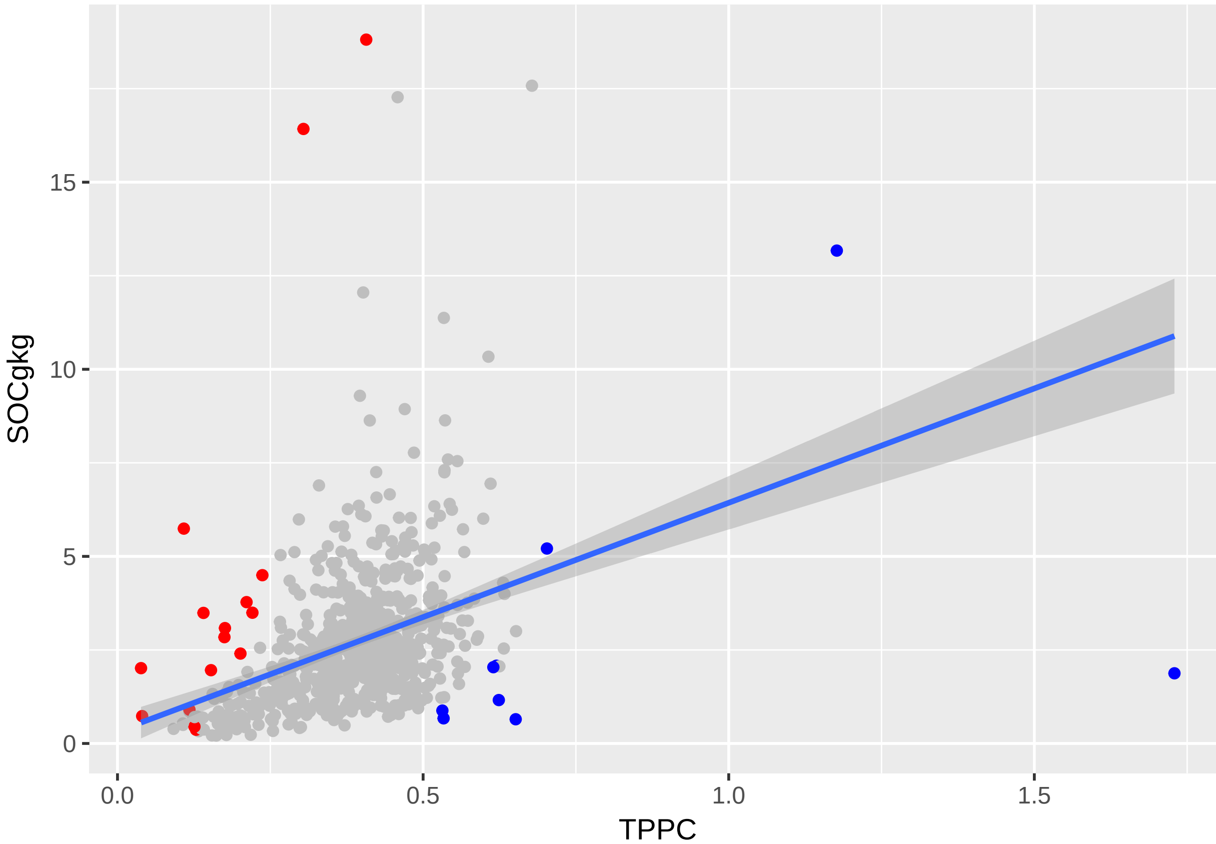
The plots have almost exactly the same *distributions* as those from `m1` if the normal linear model is correct. The only slight difference is that the scale on the y-axis of the first plot will be different.

Residuals 1

We can also plot the residuals. The code below specifies that plot will have three class intervals: below -2, between -2 and 2, and above 2. These are useful intervals given the residuals should be Normally distributed, and these values are the approximate two-tailed 5% points of this distribution. Residuals within these points will be shaded grey, large negative residuals will be red, and large positive ones will be blue. describing the relationship between `TPCC` and `SOCgkg`:

```r
s.resids = rstandard(m1)
cols <- rep("grey", length(s.resids))
cols[s.resids < -2] <- "red"
```

```
cols[s.resids > 2] <- "blue"
ggplot(data,aes(TPPC,SOCgkg)) +
    geom_point(colour = cols) +
    geom_smooth(method='lm')
```



Refine the model: finding the best fit

In the example above, a small set of variables was passed to the `lm` function. It is possible to find the model of TPPC that best fits the data using the `stepAIC` function. In this we start with a model of all variables and then seek to identify the variables the best describe the variation in `TPPC`.

First we need to code the full model (i.e. with all covariates):

```
terms <- names(data)[c(5,6,8:19)]
regmod <- paste(terms[2], "~")
for ( i in 2:14 ) {
    if ( i !=  2) regmod <- paste(regmod, "+", terms[i])
    if ( i == 2) regmod <- paste(regmod, terms[i])
}
```

This creates a character variable called `regmod`. You should examine it:

```
regmod
```

This can be converted into a formula using the `as.formula` function for input into a regression:

```r
regmod <- as.formula(regmod)
```

Finally the `stepAIC` function is used to identify the best fitting model - that is the most parsimonious model. Notice how in the code below the `lm` function is embedded:

And the resulting formula can be exported out for subsequent use:

```r
as.formula(step.i$call)
```

```
## SOCgkg ~ SiltPC + SandPC + NO3Ngkg + NH4Ngkg + N2P + Altitude_m +
##     LandUse + Position
```

```r
new.reg.mod <- as.formula(step.i$call)
```

The `lm` function be used again to develop a regression model of TPPC

```r
m2 <- lm(new.reg.mod, data = data)
summary(m2)
```

```
##
## Call:
## lm(formula = new.reg.mod, data = data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -8.4805 -0.8759 -0.2338  0.5912 14.1776
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)          22.635761   3.018520   7.499 2.03e-13 ***
## SiltPC               -0.041866   0.026199  -1.598  0.11051
## SandPC               -0.061237   0.022635  -2.705  0.00699 **
## NO3Ngkg               0.013562   0.005664   2.394  0.01693 *
## NH4Ngkg               0.033150   0.008014   4.137 3.97e-05 ***
## N2P                   0.416191   0.071725   5.803 1.00e-08 ***
## Altitude_m           -0.013926   0.001865  -7.465 2.57e-13 ***
## LandUseCropland       0.494492   0.289244   1.710  0.08780 .
## LandUseGrassland      0.276422   0.255246   1.083  0.27921
## LandUseShrubland     -0.186347   0.277204  -0.672  0.50166
## PositionGully         0.246590   0.211337   1.167  0.24370
## PositionMiddle slope -0.254692   0.216909  -1.174  0.24073
## PositionTop           0.397650   0.221285   1.797  0.07278 .
## PositionUp slope     -0.280141   0.268851  -1.042  0.29779
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.723 on 675 degrees of freedom
## Multiple R-squared:  0.2978, Adjusted R-squared:  0.2843
## F-statistic: 22.03 on 13 and 675 DF,  p-value: < 2.2e-16
```

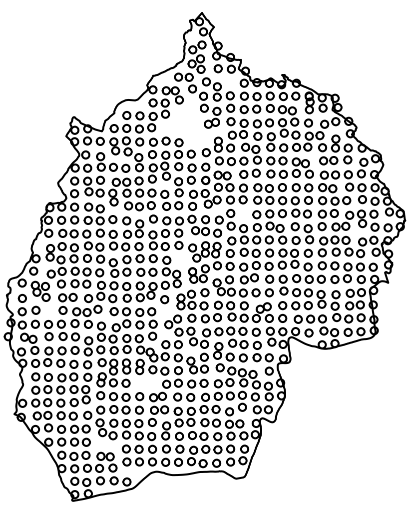
Mapping the data

You will have noticed that the Liudaogou data contains locational attributes, `Latitude` and `Longitude`. We can use these to create a spatial point dataset, similar to a point shapefile. The functionality for this is provided by the `sp` package loaded as part of `GISTools` above.

```
## define a projection
proj. <- CRS("+proj=tmerc +lat_0=0 +lon_0=108 +k=1 +x_0=500000
             +y_0=0 +ellps=krass +units=m +no_defs ")
## define coordinates
coords <- data[,3:2]
## create a SpatialPointsDataFrame
data.sp <- SpatialPointsDataFrame(coords,
    data = data.frame(data),
    proj4string = proj.)
```

We can map the data, load a boundary dataset and add this to the map:

```
## have a quick look!
plot(data.sp, pch = 1, cex = 0.5)
library(repmis)
source_data("https://github.com/lexcomber/CAS_GW_Training/blob/master/boundary.RData?raw=True")
plot(boundary, add = T)
```



Residuals 2

We can of course now plot the residuals from `m2` but this time spatially.

First determine the residuals:

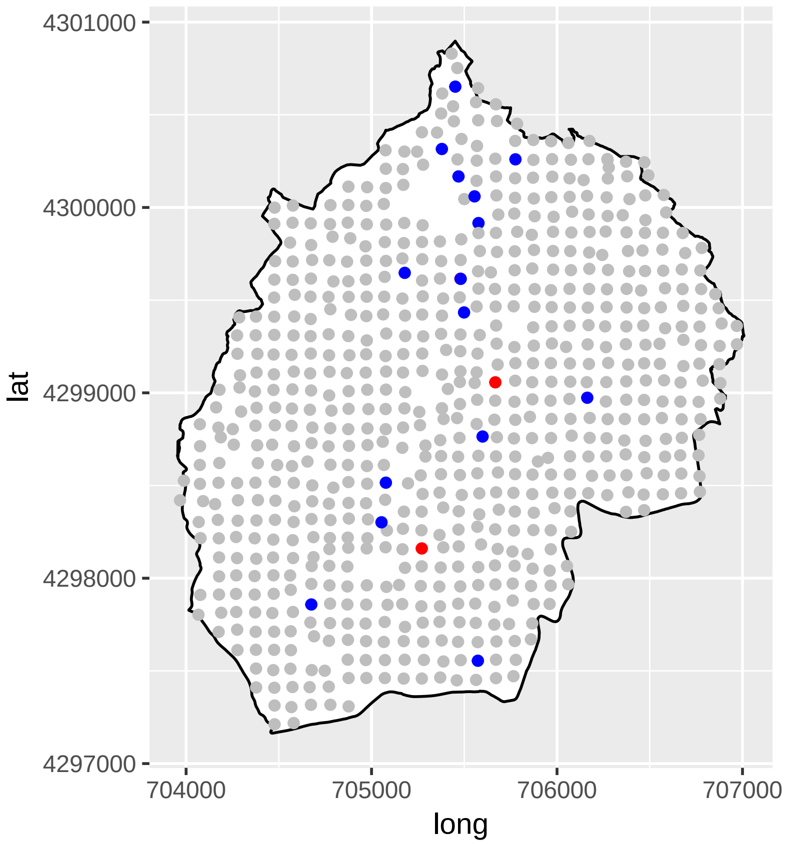
```
s.resids = rstandard(m2)
cols <- rep("grey", length(s.resids))
cols[s.resids < -2] <- "red"
cols[s.resids > 2] <- "blue"
```

Then set up the boundary layer as a background for plotting using `ggplot`:

```
boundary@data$id = rownames(boundary@data)
boundary.points = fortify(boundary, region="id")
boundary.df = join(boundary.points, boundary@data, by="id")
```

Then plot using `ggplot`:

```
ggplot(boundary.df) +
  geom_polygon(aes(x=long, y=lat), colour="black", fill="white") +
  coord_equal() +
  theme() +
  geom_point(data = data, aes(x = Longitude, y = Latitude), colour = cols)
```



Saving your work

You can save your work to an `.RData` file in your working directory using the `save.image` function:

```
save.image(file = "part1.RData")
```

Code

The practical, all of the analyses and mappings were undertaken in R, the free open source statistical software. The RMarkdown script used to produce this practical that includes all the code used in the analysis and to produce all of the tables, figures maps etc, can be found at https://github.com/lexcomber/CAS_GW_Training.

References

Brunsdon, C.F., Fotheringham, A.S. and Charlton M. (1996). Geographically Weighted Regression - A Method for Exploring Spatial Non-Stationarity, *Geographic Analysis*, 28: 281-298.

Fotheringham, A. S., C. Brunsdon, and M. Charlton. (2002). *Geographically Weighted Regression: The Analysis of Spatially Varying Relationships.* Chichester: Wiley

Wang, Y., Zhang, X. and Huang, C., 2009. Spatial variability of soil total nitrogen and soil total phosphorus under different land uses in a small watershed on the Loess Plateau, China. *Geoderma*, 150(1), pp.141-149.