Module 4 - Supervised Learning: Regression Techniques.

4.3 Machine learning methods for regression

- \textit{KNN} for regression
- \textit{M5}-regression model (Cubist)

Authors:

By María José Gacto

Senior Lecturer at the University of Granada.

Department of Software Engineering, ETSIIT.

by Augusto Anguita-Ruiz

Postdoctoral Research Fellow at Barcelona Institute for Global Health- ISGlobal.
Reminder: Introduction to NoteBook.

In this *NoteBook* you will be guided, step-by-step, through loading a dataset to the descriptive analysis of its contents. The Jupyter NoteBook (*Python*) is an approach that combines text blocks (like this one) and code blocks or cells. The great advantage of this system is its interactivity because cells can be executed to directly check the results they contain.

**Very important**: the order of the instructions is fundamental and so each cell in this NoteBook must be executed sequentially. If any are omitted, the program may throw an error and so if there is any doubt, you will have to start from the beginning again.

First, it is very important to select "*Open in draft mode*" (draft mode) at the top left at the beginning. Otherwise, for security reasons, you will not be allowed to execute any code blocks. When the first of the blocks is executed, the following message will appear: “*Warning: This NoteBook was not created by Google*”. Don’t worry, you can trust the contents of the NoteBook and click on “*Run anyway*”.

Let’s start!

Click on the “*play*” button on the left side of each code cell. Remember that lines beginning with a hashtag (#) are comments and do not affect the execution of the script. You can also click on each cell and press “*Ctrl+enter*” (“*Cmd+Enter*” on Mac). Each time you execute a block, you will see the output just below it. The information is usually always the last statement, along with any *print()* commands present in the code.

**INDEX**

In this *notebook*:

1. We learn the general concepts of the k nearest-neighbors technique for regression.
2. We will discover the general concepts of the M5 regression-model learning technique.
3. We apply both these techniques to our childhood obesity dataset as more automated alternatives to regression analyses.

Contents:

1. k-nearest neighbors (KNN) for regression.
2. The M5 regression model
3. R installation, libraries and reading of childhood obesity data
4. Application of KNN to the problem
5. Application of M5 to the problem
6. Cross-validation for both techniques: comparison and model choice
7. Bibliography
1. $K$-NEAREST NEIGHBORS FOR REGRESSION (KNN)

Basic idea: "If it quacks like a duck, walks like a duck, and behaves like a duck, then it's probably a duck!"

It is based on the same ideas as the well-known classification version. It needs the following three components: Having the training dataset with the possibility of being able to store it, a metric to calculate the distance between any pair of data, and a value of $k$ (number of nearest neighbours to consider during estimation). Once the above three components are in place, without any learning algorithms and model fitting, the main idea of the nearest neighbour algorithm ($k = 1$) is to:

- Store all training instances/data $< x^i, f(x^i) >$.
- Given a new query instance $x^q$, first find the nearest training instance $x^n$, and then directly estimate that $f(x^q) = f(x^n)$.

It is quite obvious that for a $k$-nearest-neighbours algorithm ($k = 1$) one will always obtain zero error with the training set itself. While this may be impressive for those with little experience, it should be remembered that the ultimate goal of any machine learning algorithm is to generalise correctly, i.e. to infer accurately when presented with new cases that have not been seen before. The fact that this technique fits so closely to the training data tends to lead to high overlearning and, therefore, a rather poor generalisation capacity with very high errors in the test data.

That is why, except in very specific situations, the use of $k = 1$ is usually avoided. In the case of a $k$-nearest-neighbour algorithm ($k > 1$), given a new query instance $x^q$, we would replace the direct estimation by:

- Ranking (categorical variable) - Choosing the vote among its $k$ nearest neighbours as an estimate of $f(x^q)$. Odd values of $k$ are often used as a way of trying to avoid ties in the vote.
- Regression (continuous variable) - Take the average of the $f$ values of its $k$ nearest neighbours as the estimate, $f(x^q) = \frac{1}{k} \sum_{i=1}^{k} f(x^i)$. In this case it is indifferent to have odd or even $k$ values (see Figure 1 for an example with $k = 3$, where the horizontal lines come from the average of the three nearest examples).
1.1 Measurement of distance

The most commonly used measure of distance is the Euclidean or Minkowski distance, although it is not the only one. Other distance metrics such as absolute distance, Manhattan distance, are also used. Algorithms often implement the Minkowski distance, as it allows the distance metric to be varied by simply changing the value of a parameter:

\[ d(x^i, x^j) = \left( \sum_{v=1}^{p} |x^i_v - x^j_v|^q \right)^{\frac{1}{q}}, \text{q} = 1 (\text{absolute or Manhattan}); \text{q} = 2 (\text{Euclidean}); \text{etc.} \]

In any case it is necessary to normalise the data beforehand so that all variables are affected equally (although in some implementations this is already done internally). To understand the importance of normalisation we propose the following example: the average working hours per day of a person has to be within the range of 24 hours (and in theory should not be more than 8), while his salary could vary from thousands to hundreds of thousands. If the values of the variables are not normalised, small changes in salary (50,000 to 51,000 euros/year) would weigh a lot more than large changes in the number of hours (4 to 8 hours). Specifically, for these values we would have \( |x^i_{\text{salary}} - x^j_{\text{salary}}|^2 = 1.000.000 \) versus \( |x^i_{\text{hours}} - x^j_{\text{hours}}|^2 = 16 \), so the working hours would therefore have practically no influence on the calculation of the distances.

1.2 Choosing the value of \( k \)

We should bear in mind that the choice of the parameter \( k \) can negatively affect the estimates. We can be guided by the following two assumptions:

- If \( k \) is too small, sensitive to noise points.
- If \( k \) is too large, the neighbourhood may include unrepresentative points.

One way to mitigate these problems is to calculate an average weighted by the distance to the \( k \) points. The most extreme case of distance-weighted KNN directly uses all the data in the training set and is known as Shepard’s method. There are also what are called Kernel functions. A Kernel function serves to regulate how the distance to the \( k \) points affects the calculation of the weighted mean to obtain the final output. Examples are: The rectangular kernel, triangular kernel, Epanechnikov kernel, Gaussian kernel, etc.
2. REGRESSION MODEL M5

As already indicated in capsule 1 of this module, **M5 could be considered almost a paradigm in the area of regression** (see the 2019 study among 164 algorithms by Gacto et. al. where a simple tree is shown to be competitive against the best ensembles of 500 trees - Bibliography section). The M5 regression model (**model tree**) is based on a regression tree structure (decision trees with numerical constants in their leaves instead of classes). The main difference between regression models and regression trees is that they include multi-variate linear functions in the leaves instead of a constant. In short, the **M5 is a set of piecewise linear functions**, which makes it perfect for fitting all types of data, both linear and non-linear. **Figure 1 shows an example of a model obtained with M5, which includes 7 different linear functions to be applied according to the conditions learned for the different variables in the tree.** Unlike the regression methods we have seen so far, M5 **does allow and takes advantage of non-ordinal categorical variables.** Thus, in the example shown in the figure we have that: If \( pgain \leq 3.5 \) and \( motor = B \) or \( A \) the LM3 linear model would be applied (taking advantage of the categorical variable \( motor \) as a decision variable in a node); and that however if \( pgain \in (3.5, 4.5] \) and \( vgain > 2.5 \) the LM5 linear model would be applied (only taking into account numerical variables in this case).

![Diagram of M5 model](image)

Figure 2. Example of a model obtained with M5

When properly implemented, M5 is one of the most efficient methods available and can **handle data sets with relatively high dimensionality (up to hundreds of attributes) while remaining a fairly fast method.** This ability differentiates M5 from other regression techniques such as multivariate adaptive regression splines (MARS), whose computational cost (execution time) grows very quickly as the number of input variables increases. **Another advantage of M5 over CART and other regression trees is that the models obtained are generally much smaller, simpler, and accurate.**
NOTE: From here onwards we will present the operation of the M5 learning method. In other words, how regression models such as the example shown in the figure are obtained from the training data for a given problem. While the regression techniques described so far might have required some effort for our students to understand, they should not have been very difficult to follow. However, due to the greater technical complexity of M5, the content will now become more technical and, depending on your prior knowledge, may become more difficult to understand.

The fact you are taking this MOOC demonstrates your interest in getting started in bioinformatics. Even though a much more technical and perhaps even 'boring' explanation is required for the M5 algorithm, we decided to include a very brief and simplified description of the method for those of you with a healthy curiosity about how these models work. Understanding how these models are taught will make them much easier to understand later. However, if you already understand what we mean when we talk about regression models obtained with M5 (a tree with conditions from its root to its leaves which determines the local application of small linear models) you can skip this explanation and continue directly with the capsule explanations starting from Section 3 where we will see how to run KNN and M5 on our childhood obesity dataset.

The M5 method was proposed by J.R. Quinlan in 1992, and although it is a classic method, it is still recognized today as one of the most powerful ones available. M5 employs the same approach as the CART regression tree to try to minimize the Mean Squared Error (MSE) with its impurity or gain function which is used to decide upon the possible splits of a node. As mentioned, M5 does not assign a constant to the leaf node but rather, fits a multi-variate linear regression model. In the following we will look at the three main components of its learning algorithm that will allow us to describe how it works.
2.1 Tree construction

The M5 method follows the recursive node splitting strategy of decision trees. Suppose that at a given step of the algorithm (a node of the tree to be generated) we have a set of instances or training data \( T \). Either \( T \) will be associated with a leaf, or one of the possible divisions will be chosen according to the impurity or gain criterion, dividing \( T \) into two subgroups. The same process is applied recursively to all subsets. This process often produces over-learning structures that must be subsequently pruned.

To minimize the MSE when attempting to apply a split, the information gain that the split would cause is measured by the reduction of the standard deviation (sd) of the data output variables before and after the split. To do this, first we calculate the sd

\[
\text{sd}(T) = \sqrt{\frac{1}{|T|} \sum_{i=1}^{|T|} (T_i - \text{mean}(T))^2}
\]

To minimize the MSE when attempting to apply a split, the information gain that the split would cause is measured by the reduction of the standard deviation (sd) of the data output variables before and after the split. To do this, first we calculate the sd of the output values of the instances in \( T \). In case \( T \) contains very few instances (e.g. less than 4) or \( \text{sd}(T) \) is already less than a few percent of the initial \( \text{sd} \), the node will be converted directly into a leaf node. Otherwise, all possible divisions of \( T \) will be calculated for each possible cut-off point on each input variable. Let us call \( T_i \) the subset of \( i-th \) instances corresponding to a specific split. If the deviation \( \text{sd}(T_i) \) of the output variable values of the instances in \( T_i \) is used as an error measure, the expected error reduction can be defined as follows:

\[
\Delta \text{error} = \text{sd}(T) - \sum_{i} \frac{|T_i|}{|T|} \text{sd}(T_i)
\]

Thus, the split that maximises the expected error reduction will be applied. Although this formulation tries to be as general as possible, in the case of M5 (as in most tree-based algorithms), the maximum number of subsets that each split will generate is two. That is, a split would only give rise to the subsets \( T_1 \) and \( T_2 \). Therefore, for each split node only two child nodes are generated (binary tree).

2.2 Pruning the tree

Pruning is performed from the leaves to the root node. At each internal node, M5 compares the estimated error of that node (as if it had not been split and was a leaf node) and the estimated error of the sub-tree hanging from it. Then, the sub-tree is pruned if it does not produce a sufficient improvement in tree performance.

The key factor in this method is how it estimates that error so that it does not over-learn too much and generalises well to new, unseen instances. M5 calculates it by first averaging the absolute difference between the values of the output variable of the training data and the values of its corresponding predictions. This will generally underestimate the error in unseen instances, so M5 multiplies it by \((n + v) = (n - v)\), where \( n \) is the number of instances associated with the node and \( v \) is the number of parameters in the linear model that would remain if it were ultimately a leaf. The intention is to increase the estimated error of models with many parameters built from a small number of instances. The estimated error of a sub-tree is calculated as the weighted sum of the estimated error of its left and right branches, multiplied by the proportion of samples descending to the left and right tree, respectively.
2.3 Linear model output

A multi-variate linear model is fitted to the training data at each node of the tree using standard regression techniques. Before pruning is performed, taking advantage of the backward loop of recursion used to generate the tree, a linear regression model is fitted for each node on the training instances associated with it.

It therefore starts with the leaf nodes considering only the variables used in the splits from the root of the tree to the leaf node (it looks upwards). However, in the case of the internal nodes (those that are not leaf/terminal), M5 does not use the variables of the splits looking upwards, but restricts itself to the variables that are referenced by the splits or by the linear models in the subtrees hanging from that node (looking downwards). These internal linear models are only used to be able to apply the pruning criteria. As M5 will compare the accuracy of such a linear model with the accuracy of the subtrees hanging from that node, this ensures a level playing field where both types of models use the same information.

In addition, after building any linear model M5 simplifies it by removing the coefficients one by one using a voracious algorithm (in the style of the top-down approach we used in capsule 2 of this module but automated). In general, this might result in an increase in the average error, however they also reduce the previous multiplicative factors, so the estimated error (for new instances not yet seen) can decrease.

3. R INSTALLATION, LIBRARIES, AND READING THE CHILDHOOD OBESITY DATA

As explained in the first capsule of this module, we need to run the following 3 cells before starting with the KNN and M5 algorithms.
In [1]:

# Estimated execution time: approx. 20 seconds.

### Installing R on Google Colab notebooks ###

!apt-get update
!apt-get install r-base
!pip install rpy2==3.5.1
%load_ext rpy2.ipython

print ("R installation on Google Colab completed")
Get:1 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Get:2 https://cloud.r-project.org/bin/linux/ubuntu bionic-cran40/ InRelease [3,626 B]
Get:3 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu bionic InRelease [15.9 kB]
Hit:4 http://archive.ubuntu.com/ubuntu bionic InRelease
Get:5 https://archive.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Hit:6 http://ppa.launchpad.net/cran/libgit2/ubuntu bionic InRelease
Get:7 http://security.ubuntu.com/ubuntu bionic-security/restricted amd64 Packages [806 kB]
Get:8 http://archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
Hit:9 http://ppa.launchpad.net/deadsnakes/ppa/ubuntu bionic InRelease
Get:10 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages [1,474 kB]
Get:11 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages [2,596 kB]
Get:13 http://ppa.launchpad.net/graphics-drivers/ppa/ubuntu bionic InRelease [21.3 kB]
Get:14 https://cloud.r-project.org/bin/linux/ubuntu bionic-cran40/ Packages [76.8 kB]
Get:17 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu bionic/main Sources [1,827 kB]
Get:21 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu bionic/main amd64 Packages [936 kB]
Get:22 http://archive.ubuntu.com/ubuntu bionic-updates/restricted amd64 Packages [840 kB]
Get:24 http://ppa.launchpad.net/graphics-drivers/ppa/ubuntu bionic/main amd64 Packages [42.8 kB]
Fetched 15.1 MB in 5s (3,297 kB/s)
Reading package lists... Done
Reading package lists... Done
Building dependency tree
Reading state information... Done
r-base is already the newest version (4.1.2-1.1804.0).
The following package was automatically installed and is no longer required:
  libnvidia-common-470
Use 'apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 69 not upgraded.
Requirement already satisfied: rpy2 in /usr/local/lib/python3.7/dist-packages (3.4.5)
Requirement already satisfied: tzlocal in /usr/local/lib/python3.7/dist-packages (from rpy2) (1.5.1)
R installation on Google Colab completed
In [2]:

# Estimated running time: 2:15 minutes approx.
# Libraries needed:
# ISLR for multivariate linear regression
# kknn for k-nearest neighbours regression
# Cubist for regression models based on M5

```R
### Installation of required libraries
#install.packages(c("ISLR", "kknn", "Cubist"))
install.packages(c("kknn", "Cubist")) #ISLR was used in the previous capsule
print ("Installation of R libraries for this module completed")

### Importing the required libraries ###
#require(ISLR)
require(kknn)
require(Cubist)
print ("Import of R libraries for this module completed")
```
R[write to console]: Installing packages into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)

R[write to console]: also installing the dependencies ‘plyr’, ‘igraph’, ‘reshape2’

R[write to console]: trying URL ‘https://cran.rstudio.com/src/contrib/plyr_1.8.6.tar.gz’

R[write to console]: Content type 'application/x-gzip'
R[write to console]: length 401191 bytes (391 KB)
R[write to console]: =
R[write to console]: =
R[write to console]: =
R[write to console]: =

R[write to console]: downloaded 391 KB

R[write to console]: trying URL 'https://cran.rstudio.com/src/contrib/igraph_1.2.11.tar.gz'
R[write to console]: Content type 'application/x-gzip'
R[write to console]: length 2398028 bytes (2.3 MB)
```
R[write to console]: =
R[write to console]: =
R[write to console]: =
R[write to console]: =
R[write to console]:
R[write to console]: downloaded 2.3 MB

R[write to console]: trying URL 'https://cran.rstudio.com/src/contrib/reshape2_1.4.4.tar.gz'
R[write to console]: Content type 'application/x-gzip'
R[write to console]:  length 37307 bytes (36 KB)
```

R[write to console]: =
R[write to console]: =
R[write to console]: =
R[write to console]: =
R[write to console]: =

R[write to console]: downloaded 36 KB

R[write to console]: trying URL 'https://cran.rstudio.com/src/contrib/kknn_1.3.1.tar.gz'
R[write to console]: Content type 'application/x-gzip'
R[write to console]: length 388410 bytes (379 KB)
R[write to console]: =
R[write to console]: =
R[write to console]: =
R[write to console]: =
R[write to console]: =
R[write to console]: =
R[write to console]: downloaded 379 KB

R[write to console]: trying URL 'https://cran.rstudio.com/src/contrib/Cubist_0.4.0.tar.gz'
R[write to console]: Content type 'application/x-gzip'
R[write to console]: length 1523565 bytes (1.5 MB)
R[write to console]: =
R[write to console]: =
R[write to console]: =
R[write to console]:

R[write to console]: downloaded 1.5 MB

R[write to console]:

R[write to console]:
R[write to console]: The downloaded source packages are in
   '/tmp/RtmpIR5xJS/downloaded_packages'
R[write to console]:
R[write to console]:

[1] "Installation of R libraries for this module completed"

R[write to console]: Loading required package: kknn

R[write to console]: Loading required package: Cubist

R[write to console]: Loading required package: lattice

[1] "Import of R libraries for this module completed"
In [3]:

# Estimated execution time: 2 seconds approx.

```r
### Reading
data <- read.csv(url("https://drive.google.com/uc?id=1GO2NBxYw54K6HkN-YgXbNadrLo5O6-0u"))

### Visualisation of a small part of the data
head(data)

<table>
<thead>
<tr>
<th>Sex</th>
<th>Age</th>
<th>Tanner</th>
<th>Height</th>
<th>BMI</th>
<th>TAGmgDL</th>
<th>HDLCmgDL</th>
<th>LDLCmgDL</th>
<th>SBP</th>
<th>DBP</th>
<th>Sede</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.5</td>
<td>0</td>
<td>1.55</td>
<td>11.34</td>
<td>55</td>
<td>51</td>
<td>93</td>
<td>97</td>
<td>60</td>
<td>41</td>
</tr>
<tr>
<td>2</td>
<td>8.0</td>
<td>0</td>
<td>1.15</td>
<td>12.40</td>
<td>51</td>
<td>70</td>
<td>59</td>
<td>90</td>
<td>55</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>10.5</td>
<td>0</td>
<td>1.42</td>
<td>12.99</td>
<td>65</td>
<td>60</td>
<td>96</td>
<td>96</td>
<td>54</td>
<td>48</td>
</tr>
<tr>
<td>4</td>
<td>8.1</td>
<td>0</td>
<td>1.27</td>
<td>13.43</td>
<td>41</td>
<td>78</td>
<td>100</td>
<td>108</td>
<td>46</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>10.4</td>
<td>0</td>
<td>1.32</td>
<td>13.72</td>
<td>39</td>
<td>100</td>
<td>120</td>
<td>107</td>
<td>69</td>
<td>51</td>
</tr>
<tr>
<td>6</td>
<td>10.4</td>
<td>0</td>
<td>1.29</td>
<td>14.02</td>
<td>57</td>
<td>76</td>
<td>73</td>
<td>87</td>
<td>59</td>
<td>45</td>
</tr>
</tbody>
</table>

Light Moderate Vigorous HOMA
1 321.5804 22.13393 3.982143 1.98
2 316.9762 48.05952 14.273810 0.87
3 337.7857 33.30952 7.988095 1.46
4 241.9762 39.67857 11.821429 1.07
5 216.0357 9.75000 2.410714 0.80
6 257.6429 36.40179 9.767857 1.35
4. APPLICATION OF KNN TO THE PROBLEM

Once we have imported the libraries and read the data we are ready to apply \textit{KNN} to our problem. The implementation of \textit{k}nn that we are going to use is included as part of the R library \textit{kknn}, which we have already installed in our environment. The function is also known as \textit{kknn}(.) and follows the following syntax:

\textit{kknn(formula = formula(train), train, test, k = 7, distance = 2, kernel ="optimal", scale :}

(Appplies k-NN to a test set given a training set -> Reminder: No learning algorithm)

\textit{formula} - variables used in \textit{lm} style; \textit{train and test} - training and test data sets; \textit{k} - number of neighbours (if not stated it is 7); \textit{distance} - Minkowski distance, i.e. with 1 Manhattan and with 2 Euclidean (if not stated it is 2); \textit{scale} - scaling of variables to have equal standard deviation (if not given, TRUE); \textit{kernel} - type of kernel used for the use of distance weights (if not given, "optimal"). The "rectangular" is \textit{k}-NN standard without weights. Existing types: "rectangular", "triangular", "epanechnikov", "biweight", "tri-weight", "cos", "inv", "gaussian", "rank" and "optimal".

In our case, for now we will leave the default parameters, see how the algorithm runs and calculate its errors, etc., by using the complete set for training and testing. This will also allow us to see how some variables affect the outcome by varying the formula and even by testing it with the same formula we obtained for the linear regression in the previous capsule. Later on, we will also demonstrate how to apply cross-validation.

In [4]:

\begin{verbatim}
# Estimated execution time: approx. 2 seconds

%%R

fitKNN <- kknn(HOMA ~ ., data, data)
yprime = fitKNN$fitted.values
cat('\n(formula with all variables) RMSE:', sqrt(sum((data$HOMA-yprime)^2)/length(yprime)), '
')

fitKNN_lm <- kknn(HOMA ~ BMI+Height+TAGmgDL+Sex+WC+LDLCmgDL+I(BMI^2), data, data)
yprime = fitKNN_lm$fitted.values
cat('(formula obtained with lm) RMSE:', sqrt(sum((data$HOMA-yprime)^2)/length(yprime)), "\n")
\end{verbatim}

(formula with all variables) RMSE: 0.4974704
(formula obtained with lm) RMSE: 0.5062813
Based on the values obtained, it is clear that the formula obtained for one technique does not necessarily behave in the same way with the KNN method. For example, significant improvements were achieved with linear regression, while these results are now worse with KNN.

This behavior is expected because KNN is based on distances and if improvements are achieved, it is usually because some variables that might be hindering or counteracting the distances calculated for the rest of the variables have been eliminated. In the following code you can see the final result of trying to eliminate some of the variables. When we removed all the variables one at a time, the Sex variable produced the greatest reduction in error. In a second iteration, the SBP variable could also be removed. However, removing more variables thereafter only made the model worse.

In [5]:

```R
# Estimated execution time: 2 seconds approx.

fitKNN <- kknn(HOMA ~ .-Sex-SBP, data, data)
yprime = fitKNN$fitted.values
cat('\n(formula with all variables) RMSE:', sqrt(sum((data$HOMA-yprime)^2)/length(yprime)), "\n")
```

(formula with all variables) RMSE: 0.4824407

Similarly, one can also try to include some non-linear term to improve the calculation of the distances. In our case we knew that $BMI$ showed a quadratic behaviour. Let's try to include it.

In [6]:

```R
# Estimated execution time: 2 seconds approx.

fitKNN <- kknn(HOMA ~ .-Sex-SBP+I(BMI^2), data, data)
yprime = fitKNN$fitted.values
cat('\n(formula with all variables) RMSE:', sqrt(sum((data$HOMA-yprime)^2)/length(yprime)), "\n")
```

(formula with all variables) RMSE: 0.4590836

Visualization:
In [7]:

```r
# Estimated execution time: 2 seconds approx.
# yprime = fitKNN$fitted.values

plot(data$HOMA~data$BMI)
points(data$BMI,yprime,col="red",pch=20)
```

But haven't we forgotten something? !!!

What about the NORMALISATION of the data? !!!

Indeed, this is a crucial step that we must never forget. If we fail to consider it, we will have to discard all our work so that we can normalize the data and then repeat the entire process. However, we have taken it into account, haven’t we? The algorithm we used already performs this normalization internally. If we remember the parameters that the `knn( )` function accepted as input, we had `scale = TRUE`, which makes normalization active by default. This parameter is used to tell it whether or not we want it to normalise the data before applying the algorithm (TRUE or FALSE).

Not all implementations of k-nearest-neighbours allow normalisation to be performed as part of the algorithm itself. Therefore, it is essential to pay close attention to this aspect and perform the normalisation by hand if the version we use does not normalise by itself.
5. APPLICATION OF M5 TO THE PROBLEM

In this section we will see how to apply M5 to our insulin resistance estimation problem. The implementation of M5 that we are going to use is included as part of R's Cubist library, which we have already installed in our environment. Cubist is actually an extension of M5 that includes a series of M5 iterations based on boosting to combine several trees. The combination of trees or ensembles represents more complex or advanced techniques that are not the subject of this introductory MOOC on bioinformatics. Moreover, it is also not at all clear that Cubist performs significantly better than its base algorithm, and it is definitely not better than M5 bagging (an alternative technique to boosting for also combining multiple trees). See the study by Gacto et al. 2019 where it is seen that M5 is competitive with Cubist and a bagging of M5 is the best scoring technique - References section.

We will therefore focus on M5 by telling the cubist( ) function by parameter that the number of iterations of the boosting is equal to 1. In this way, we have just the behaviour of the original M5. The function is known as cubist( ) and follows the following syntax:

\[
cubist(x, y, committees = 1)
\]

(Apply Cubist to training data with separate X's and Y's, and get a single tree for inter-model consensus -> M5)

\(x\) - a data matrix with the values of the input variables. Missing values are allowed but only numerical. Categorical values are also allowed; \(y\) - a numeric vector with the corresponding output variable values; \(committees\) - an integer, number of models for consensus (if not given it is 1 -> M5).

As with \(k\)-nn we will see how to run it, calculate errors, and in this case visualise the tree, by using the full set separating X's and Y. However, M5 already selects the most relevant variables during the learning process itself, so in this case we will not have to do it manually. In fact, we can see how the cubist( ) function does not include any way to specify the formula. Likewise, we will show how to apply cross-validation later together with \(k\)-nn.
In [8]:

# Estimated execution time: 2 seconds approx.

```r
	tam <- length(names(data))
	fitM5 <- cubist(x = data[, -tam], y = data$HOMA) #committees = 1
	yprime = predict(fitM5, data)

cat('\n(M5) RMSE:', sqrt(sum((data$HOMA-yprime)^2)/length(yprime)), "\n")
	summary(fitM5)
```
```
Call:
cubist.default(x = data[, -tam], y = data$HOMA)

---------------------------------
Target attribute `outcome'

Read 292 cases (16 attributes) from undefined.data

Model:

Rule 1: [120 cases, mean 1.309, range 0.26 to 3.27, est err 0.428]

if
    Age <= 9.5
    BMI <= 30.65
then
    outcome = -2.244 + 0.069 BMI + 0.154 Age + 0.39 Sex + 0.66 Height
              - 0.0047 WC + 0.0017 TAGmgDL + 0.0027 HDLCmgDL
              - 0.0012 LDLCmgDL

Rule 2: [118 cases, mean 1.310, range 0.12 to 2.92, est err 0.419]

if
    Age > 9.5
    BMI <= 27.3
then
    outcome = -1.301 + 0.038 BMI + 0.0013 Sedentary + 0.67 Height
              + 0.17 Tanner + 0.0053 HDLCmgDL - 0.0024 LDLCmgDL
              + 0.0018 TAGmgDL - 0.0007 WC

Rule 3: [35 cases, mean 2.217, range 0.76 to 3.58, est err 0.532]

if
    BMI > 27.3
    BMI <= 30.65
then
    outcome = -0.678 + 0.044 BMI + 0.79 Height + 0.0062 HDLCmgDL
              - 0.0028 LDLCmgDL + 0.0022 TAGmgDL + 0.0006 Sedentary

Rule 4: [18 cases, mean 3.376, range 1.38 to 5.41, est err 0.733]

if
    BMI > 30.65
    BMI <= 32.1
then
    outcome = -12.599 + 0.439 BMI - 0.0154 WC + 1.45 Height + 0.0081 SBP
              + 0.0032 TAGmgDL + 0.18 Sex + 0.0046 HDLCmgDL - 0.002 LDLCmgDL

Rule 5: [9 cases, mean 4.458, range 2.55 to 8.99, est err 0.622]

if
    BMI > 32.1
then
    outcome = -58.467 + 1.901 BMI
```
In this case we can see how a model with a RMSE of 0.5392671 is obtained that includes different lines (or linear models) for the different cut-off points for $BMI$ and $Age$. Since we started with the insulin resistance problem, we have seen that these are probably the two most decisive variables in this problem. Therefore, it makes perfect sense that M5 would select them to make the decisions comprising its tree.

Visualization:
In [9]:
#yprime = predict(fitM5, data)
%%R
plot(data$HOMA ~ data$BMI)
points(data$BMI, yprime, col="green", pch=20)
Now we have reached the point where we have executed all the methods being studied and have obtained their different RMSE error values. However, we must not rush to choose which one is the winning method. As we well know, we must first estimate the real error of our problem by performing cross-validation and comparing the results with the average errors obtained for the test sets. We already saw how to calculate this for the multivariate linear regression, where we obtained a \( RMSE \) of 0.6956061. In this last section we will now demonstrate how to apply cross-validation with KNN and M5.
In [10]:

# Estimated execution time: approx. 2 seconds.
### KNN ###

```r
set.seed(123456)  # We set the same seed used with linear regression.
                 # to get the same partitions
k <- 5
data$kfold <- sample(1:k, nrow(data), replace = T)

performances <- c()
# One iteration per fold
for (fold in 1:k){
    # The training set for the iteration is created.
    training_set <- data[data$kfold != fold,]
    nombres <- names(training_set)
    tam <- length(nombres)-1
    training_set <- training_set[,nombres[1: tam]]

    # The test set for the iteration is created.
    testing_set <- data[data$kfold == fold,]
    nombres <- names(testing_set)
    tam <- length(nombres)-1
    testing_set <- testing_set[,nombres[1: tam]]

    # Training the model for iteration
    model <- kknn(HOMA ~ .-Sex-SBP+I(BMI^2), training_set, testing_set)

    # Calculating the test error
    yprime = model$fitted.values  # yprime <- predict(model, testing_set)
    # changed because predict doesn't work with kknn,
    # since kknn itself is the predict
    RMSE <- sqrt(sum((testing_set$HOMA-yprime)^2)/length(yprime))

    # RMSE is added to the list of errors
    performances[fold] <- RMSE
}

# remove the artificial column added for kfold
# (so that it doesn't accumulate columns if it is executed several times)

#names <- names(data)
tam <- length(nombres)-1
data <- data[,nombres[1: tam]]

cat("Average RMSE in test for 5-fcv in K-nn: ", mean(performances))
```

Average RMSE in test for 5-fcv in K-nn: 0.7929245
In [11]:

# Estimated execution time: 2 seconds approx.
### M5 ###

```r
set.seed(123456) # set the same seed used with linear regression
    # to get the same partitions
k <- 5
data$kfold <- sample(1:k, nrow(data), replace = T)

performances <- c()
# one iteration per fold
for (fold in 1:k){
    # Training set is created for the iteration
    training_set <- data[data$kfold != fold,]
    nn <- names(training_set)
    tam <- length(nn)-1
    training_set <- training_set[,nn[1: tam]]

    # The test set for the iteration is created.
    testing_set <- data[data$kfold == fold,]
    nn <- names(testing_set)
    tam <- length(nn)-1
    testing_set <- testing_set[,nn[1: tam]]

    ## Training the model for iteration
    tam <- length(names(training_set))
    model <- cubist(x = training_set[, -tam], y = training_set$HOMA) #committees = 1

    ## Calculating test error
    yprime <- predict(model, testing_set)
    RMSE <- sqrt(sum((testing_set$HOMA-yprime)^2)/length(yprime))

    # RMSE is added to the error list
    performances[fold] <- RMSE
}

# Remove the artificial column added for kfold
# (so that it doesn't accumulate columns if run multiple times)
nn <- names(data)
tam <- length(nn)-1
data <- data[,nn[1: tam]]

cat("mean RMSE in test for 5-fcv on M5:", mean(performances))

mean RMSE in test for 5-fcv on M5: 0.7807851
Below are the results we obtained when we were selecting the variables to fix the formula or seeing how M5 runs, i.e. without cross-validation:

- Linear regression without validation - RMSE: 0.6507716.
- KNN without validation - RMSE: 0.4590836
- M5 without cross-validation - RMSE: 0.6507716
- M5 without validation - RMSE: 0.5392671

In this case we would have chosen k-nn as the technique to apply to our problem (if we think only about accuracy and not interpretation).

However, let's check what happens when we perform the cross-validation:

- Linear regression with validation - RMSE: 0.6956061
- KNN with validation - RMSE: 0.7929245
- M5 with validation - RMSE: 0.7807851

You can draw your own conclusions based on these findings.

Therefore, the last step when running different techniques to solve a given problem should be to compare the mean test errors obtained by cross-validation of these models. This will allow you to select the best one(s) based on these values and/or interpretability criteria. Once the technique is fixed, you can reapply it with the whole data set to obtain a final candidate model.
REFERENCES

KNN


M5

- Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani. An Introduction to Statistical Learning with Applications in R Springer, 2013 (Chapter 08)
- J. R. Quinlan, Learning With Continuous Classes, In Proc. of the 5th Australian Joint Conf. on Artificial Intelligence, pp. 343-348, 1992

Libraries

- Using read.csv from R: https://www.rdocumentation.org/packages/utils/versions/3.6.2/topics/read.table
- Using kknn in R: https://cran.r-project.org/web/packages/kknn/index.html
- Using Cubist in R: https://cran.r-project.org/web/packages/Cubist/index.html

ADDITIONAL REFERENCES


MOOC Machine Learning y Big Data para la Bioinformática (1ª edición) http://abierta.ugr.es