Advanced Research Methods - E7004

Day 5 - Machine Learning

Dr. Fabio Veronesi

23 March 2018

Summary

- Introduction
  - Bias-Variance Trade-Off
  - Cross-Validation
- Regression trees
- Random forest
- Boosted regression trees

Packages

For this exercise we will need the following packages:

```r
install.packages("mvtnorm")
install.packages("caret")
install.packages("mlbench")
install.packages("MLmetrics")
install.packages("rpart")
install.packages("randomForest")
install.packages("dismo")
install.packages("gbm")

library(mvtnorm)
library(caret)
library(mlbench)
library(MLmetrics)
library(rpart)
library(randomForest)
library(dismo)
library(gbm)
```

Introduction

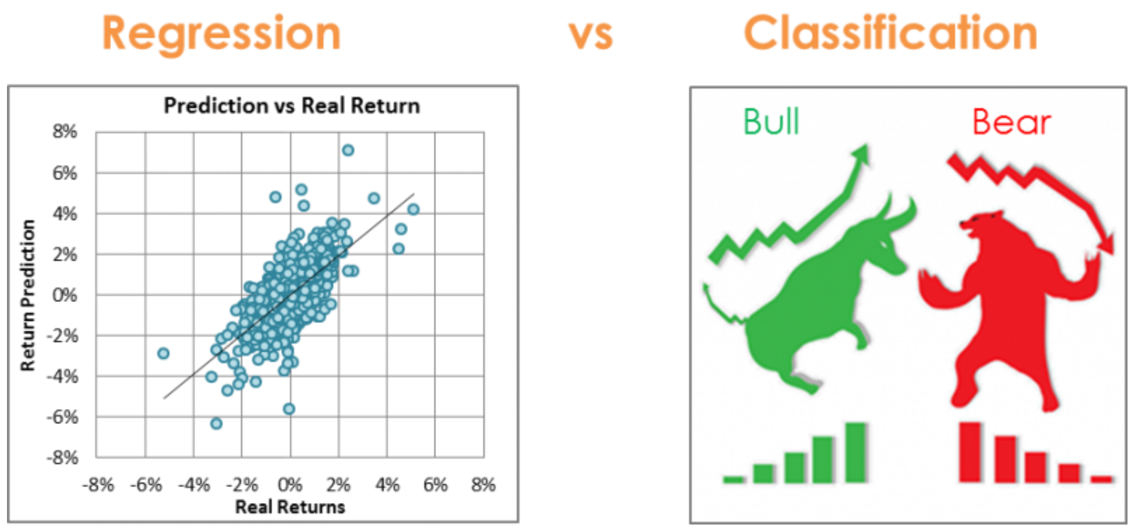
So far we looked at descriptive statistics, which aims at describing a dataset or a series of datasets: centrality, spread, plotting. We also looked at inferential statistics, which aims at using data from relatively small samples to draw conclusions that can be extrapolate to a wider population: ANOVA, linear regression, GLM. In this lecture we will look at predictive modelling, meaning techniques to build statistical models that can help explain the variance in our dependent variable using a set of predictors. Moreover, as the word suggests predictive modelling is also very useful to predict (or estimate) the dependent variable for new values of the predictors. This type of statistical tools are very powerful and can be used for any type of problem, being spatial modelling, temporal forecasting or any other type of modelling. For example, predictive modelling is used by banks to predict the probability of a mortgage default for new clients. It is also used by Netflix to suggest you potential movies you might like, and Microsoft to recognize your body movements on the kinetic device for the XBox.

Predictive modelling can take many forms, but the most popular is supervised machine learning. This is a class of algorithms, which require a training dataset of observations and a set of predictors. The aim of machine learning is finding a function that can model the variance of the dependent variable based on the predictors:

$$Y = f(X) + \epsilon$$

where Y is the dependent (or target) variable we aimed at modelling and predicting, and $f(X)$ is a function of the set of predictors (X). Finally, ϵ is the random error component and depends on several factors (e.g. measurement error) and it is the irreducible part of the variance (i.e. the variance in the data that the model cannot explain).

The term machine learning is generally used to define complex algorithms, like the convolutional neural network employed by driverless cars to recognise their surroundings. This is however very misleading because supervised learning can take many forms. For example, linear regression can be used to model and predict the relation between target variable and predictors. Logistic regression, which we used to model presence/absence data, can also be employed as a predictive model to classify binary outcomes. In fact, machine learning is generally divided into two classes: regression and classification:



Regression and Classification

The difference between the two is that regression aims at predicting continuous variables (e.g. soil moisture, house prices ect.), while classification aims at predicting categorical variables (e.g. good/bad, soil classes).

In this lecture we will focus on regression, and explore some popular machine learning algorithms and we will see how to select the best among the ones we will test.

Bias-Variance Trade-Off

As mentioned, machine learning tries to model the target variable as a function of the predictors. The error of the function that models the variable is the sum of two quantities: bias and variance. Bias refers to the approximation error created by using strict functions. A linear model serves as an example: no matter how many observations are in the dataset and their general pattern, linear regression will always model them using a line. This creates an error that is intrinsic to the fact that the general shape of the function does not change. Thus linear regression is a biased method.

On the contrary, variance measures the amount of change that the function experiences with changes in the training set. An example of a method with high variance can be a cubic spline. If applied to a dataset, since it fits a local polynomial of third order, it will probably fit most of the observations very closely. However, substantially changing the training data will also drastically modify the shape of the curve, since it will again try to fit all observations. Thus, this method has high variance and low bias.

For a more comprehensive explanation please refer to free Springer book by James et al. (2013).

The bias-variance trade-off is extremely important in machine learning and has very real implications for the model selection. You may think that selecting a complex model will always produce better predictive results. This is however not true, everything depends on the dataset we are dealing with and the new data we wish to predict. We can look at the following example to better understand this point.

Example - Wind Speed Mapping

Let's assume you want to use machine learning to develop a wind speed atlas to use for identifying sites where to build new wind farms. Your research framework involves collecting wind speed data, for example from sensors mounted at airports and airfields. Wind speed is your target variable, so now you need to consider predictors. In general, for spatial mapping predictors are environmental variables: such as terrain elevation, slope, air temperature, humidity, solar irradiation, land cover. These values are available for all the locations in our training set, i.e. airports and airfields, but also for all the locations we may be interested in predicting, i.e. test set. If the predictors do not cover the test area there is no point of applying machine learning.

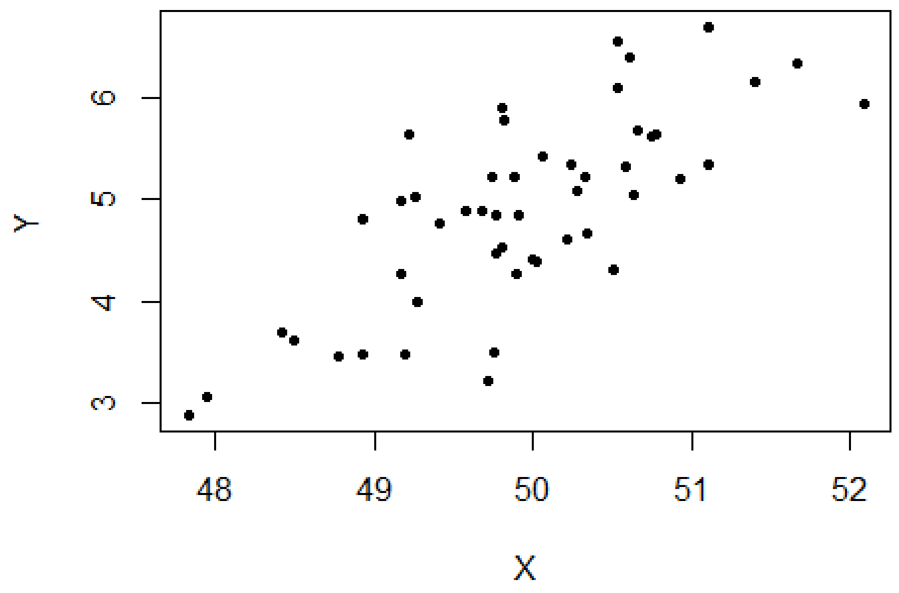
Generally, airports and airfields are built in low-lying areas, therefore the predictors will all be an expression of environmental variables in these areas. However, it is important to get wind speed estimates maybe along ridges or high elevation areas, where wind speed should be high and building wind turbine would make more sense. So in this case the range of predictors in the training set is different compared to the range of predictors in the test set. This could be an issue, particularly for complex models.

To better understand this point we can run a simple simulation:

```r
sigma <- matrix(c(1, 0.8,
                  0.8, 1 ), ncol=2, byrow=T)

Data = rmvnorm(n=50, mean=c(50, 5), sigma=sigma)

str(Data)

##  num [1:50, 1:2] 49.3 50.8 49.9 49.6 49.3 ...
```

The function `rmvnorm` is included in the package `mvtnorm` and allows the simulation of normally distributed data with particular correlations. In this example we want to simulate two vectors with a 0.8 correlation coefficient (i.e. very linearly correlated). The function requires us to specify `n`, sample size for each vector; `mean`, mean values; and `sigma`, which is the covariance matrix. In this case we are specifying a 0.8 correlation coefficient. The function `rmvnorm` returns a `data.frame` with two columns, one of each of the two vectors. To visually check their correlation we can create a scatterplot (please be aware that because these are simulated data your results may be different):

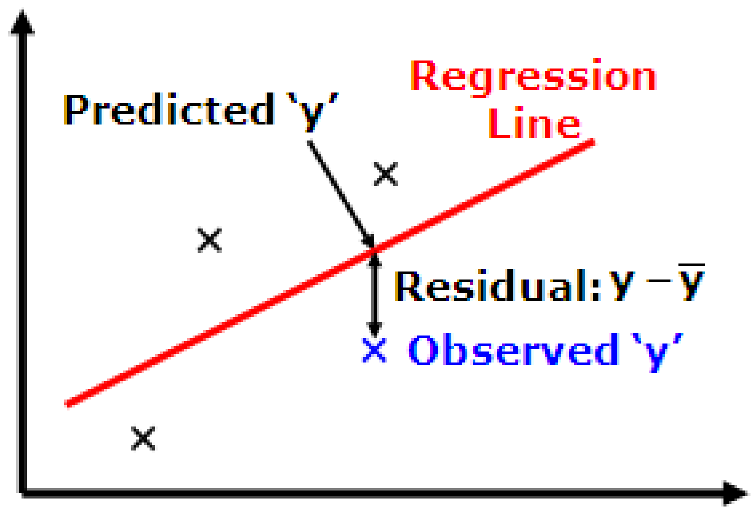
```r
X = Data[,1]
Y = Data[,2]

plot(Y ~ X, xlab="X", ylab="Y", pch=20)
```



Since we simulated these data we know that they are very linearly correlated and therefore the best model is a linear regression. However, let's assume we do not know that and we want to find the best model between linear model, quadratic and fourth order polynomials. We first fit the three models:

```r
Linear = lm(Y ~ X)
Quadratic = lm(Y ~ poly(X, 2))
Fourth = lm(Y ~ poly(X, 4))
```

Please notice the function `poly` to easily fit polynomials in R. At this point we could check their goodness of fit by comparing their estimates with our observations of Y. In other words checking the residuals of each model. Residuals are both positive and negative:



Residuals

For this reason, simply computing their mean value is pointless, since it will be very close to zero. A popular index to evaluate accuracy of machine learning models is the mean absolute error (MAE), which is simply the average of the absolute residuals:

$$MAE = \frac{\sum_{i=1}^{n} | y_i - \hat{y} |}{n}$$

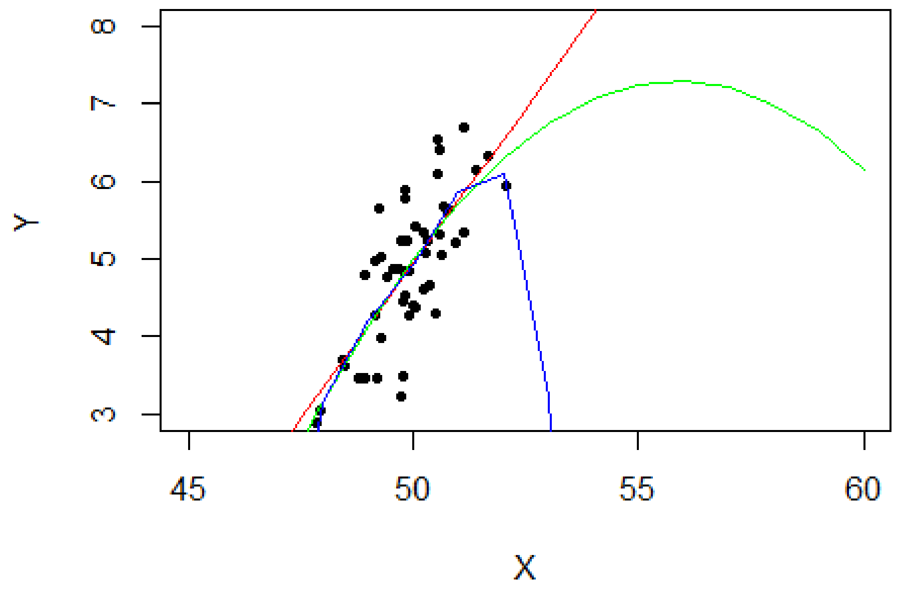
The code to compute MAE using the entire training set, known as training error, is simply:

```
mean(abs(residuals(Linear)))

## [1] 0.4895765

mean(abs(residuals(Quadratic)))

## [1] 0.4801885

mean(abs(residuals(Fourth)))

## [1] 0.4713434
```

The best model is the one that minimises MAE, so it seems the best model is the fourth order polynomial (please be aware that because it is a simulation your results may be different). However, let's look at how each model performs outside the range of predictors, for example with values of X between 45 and 60. We first use the function `predict` to obtain estimates for the whole range, then create a plot:

```
P.LIN = predict(Linear, newdata=data.frame(X=45:60))
P.QUA = predict(Quadratic, newdata=data.frame(X=45:60))
P.FOU = predict(Fourth, newdata=data.frame(X=45:60))

plot(Y ~ X, xlab="X", ylab="Y", pch=20, xlim=c(45, 60), ylim=c(3, 8))
```

```r
lines(x = 45:60, y = P.LIN, type = "l", col="red")
lines(x = 45:60, y = P.QUA, type = "l", col="green")
lines(x = 45:60, y = P.FOU, type = "l", col="blue")
```

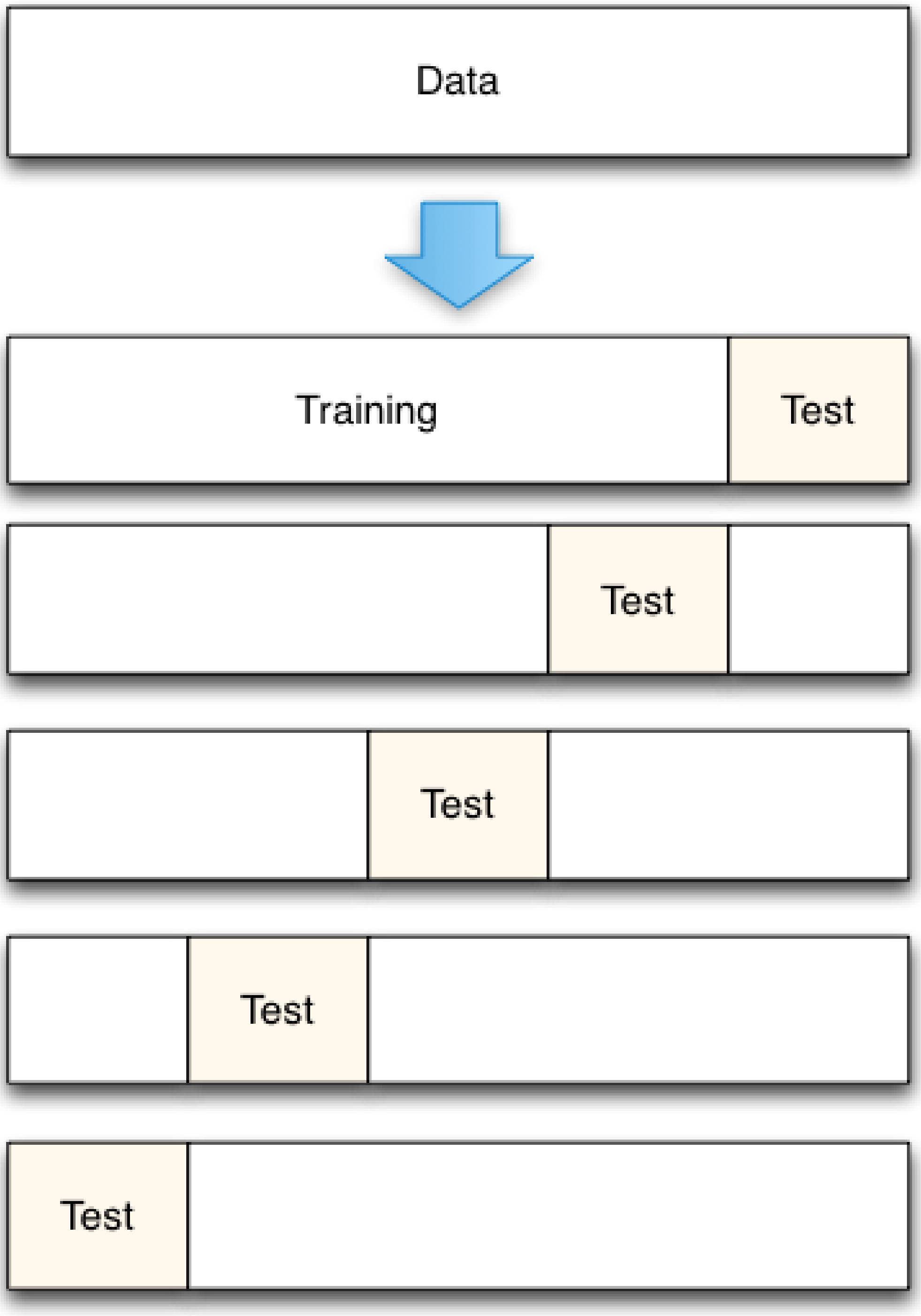


It seems clearer now that the fourth order polynomial is not the right model for this dataset. This is an issue known as overfitting: simply put, complex algorithms tend to fit the training set too well, and thus they tend to fit the random noise (which may be caused by factors that predictors cannot control). This decreases their predictive power, i.e. accuracy when predicting new data.

Cross-Validation

The problem is that minimizing the training error is not sufficient for finding the best model. We need to determine the accuracy of the model when predicting new data, i.e. the test error. However, we want to use machine learning to estimate data which are not part of our training set, so by definition we do not have a test set to use for assessing the accuracy of our models.

The solution is to use the training set to estimate the test error, by using a procedure called cross-validation. This technique splits the training set into several (usually 5) random subsets, or folds. The algorithm is then trained using only four folds, and tested on the one that was excluded. This procedure is then repeated until all folds are used once for testing. A graphical representation of cross-validation is below:



5-Folds Cross-Validation

We will look at the code to perform cross-validation below.

Regression trees

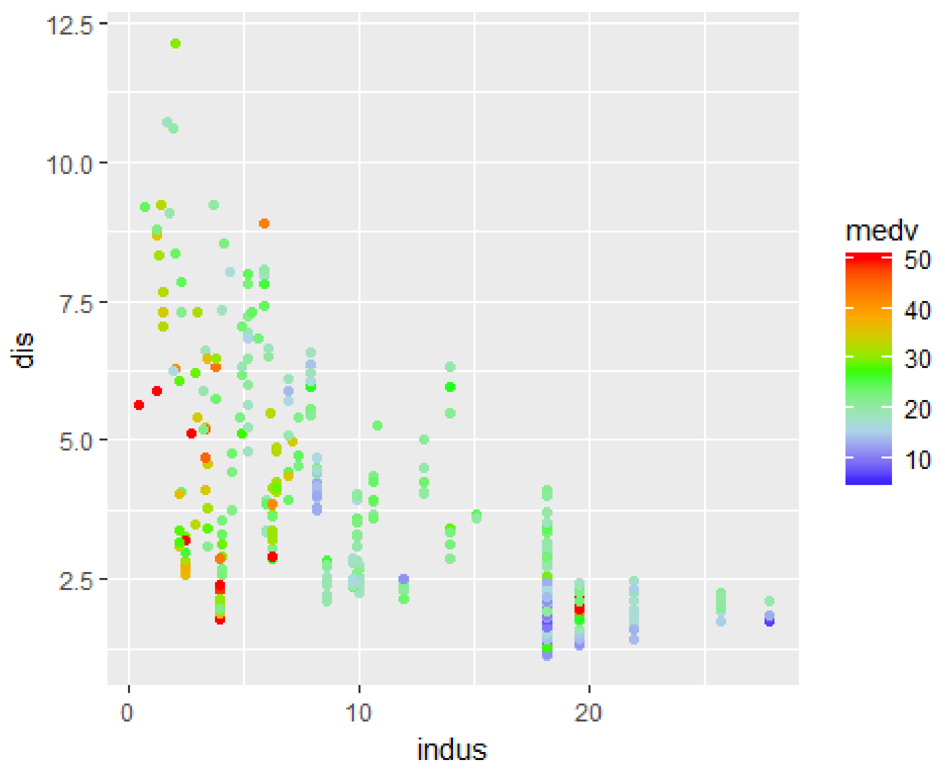
Regression trees partition the predictor space creating a set of “if-then” rules that are used to estimate classes of probabilities. To better understand this let’s load a sample dataset:

```
data(BostonHousing)
head(BostonHousing)
```

```
##      crim zn indus chas   nox    rm  age    dis rad tax ptratio      b
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12
##   lstat medv
## 1  4.98 24.0
## 2  9.14 21.6
## 3  4.03 34.7
## 4  2.94 33.4
## 5  5.33 36.2
## 6  5.21 28.7
```

This dataset contains housing data for 506 census tracts of Boston from the 1970 census. The target variable is `medv`, which stands for median value of owner-occupied homes in USD 1000’s. Let’s say we want to fit a model that predicts `medv` using `indus`, proportion of non-retail business acres per town, and `dis`, weighted distances to five Boston employment centres. Let’s plot these variables:

```
library(ggplot2)

ggplot(data=BostonHousing, aes(x=indus, y=dis, color=medv)) +
  geom_point() +
  scale_color_gradientn(colours=c("blue","light
blue","green","orange","red"))
```



This code creates a plot where `indus` is on the X axis and `dis` is on the Y axis. Dots are coloured by `medv`. Unfortunately the course is not focused on data visualization, so we cannot describe in details the code I used to create this plot. However, if you are interestied in knowing more about this topic please complete my course on Data Visualization, available on OneDrive.

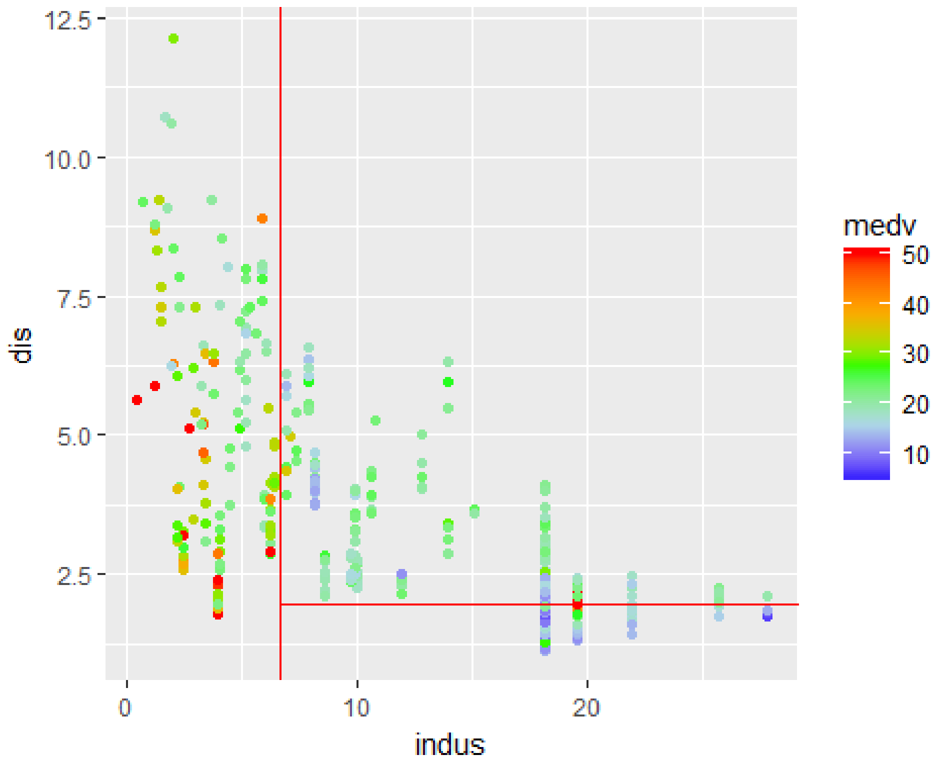
From this image it seems clear that on the left side there is a higher concentration of relatively high values of `medv`. In other words, for values of `indus` below 6 or 7 we have a higher probability of having high value properties. This could be our first split, since values seems to be more similar within these two subgroups. This guarantees that if we need to estimate new data we can use this information to predict a possible property value, which would be the average of observations on the two sides of the division line. Let's see how `CART`, which is the most popular algorithm for regression trees, partitions this dataset:

```
rpart(medv~indus+dis, data=BostonHousing)

## n= 506
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 506 42716.30000 22.53281
##    2) indus>=6.66 320 18197.95000 18.96469
##      4) dis< 1.96955 102  9478.71800 15.91078
##        8) dis>=1.35735 89  3166.89200 14.06067
##         16) indus< 18.84 58  1592.74400 12.34655 *
##         17) indus>=18.84 31  1084.88800 17.26774 *
##        9) dis< 1.35735 13  3921.58300 28.57692 *
##      5) dis>=1.96955 218  7322.85100 20.39358  
##       10) indus< 18.84 194  3598.93700 19.79588  
##         20) dis< 2.37495 36   451.36970 15.84722 *
##         21) dis>=2.37495 158  2458.36700 20.69557 *
##       11) indus>=18.84 24  3094.38500 25.22500  
##         22) indus>=20.735 13    69.74769 19.33077 *
##         23) indus< 20.735 11  2039.22900 32.19091 *
##    3) indus< 6.66 186 13435.12000 28.67151  
##      6) indus>=3.985 94  4620.25400 25.53511  
##       12) indus< 6.145 71  1419.39700 23.05211 *
##       13) indus>=6.145 23  1411.86000 33.20000 *
##      7) indus< 3.985 92  6945.40700 31.87609  
##       14) dis>=6.54565 28   859.14110 26.61786 *
##       15) dis< 6.54565 64  4973.39500 34.17656 *
```

Clearly, the algorithm creates a lot more partitions, and they are here numbered according to their importance. If we look at number 2 (number 1 is the full dataset), we would see that the first probability creates a split at `indus` above or below 6.66. This is exactly what we noticed visually. The last value in each row is the prediction for that class, which is the average of all observations in the partition. So if we look at particion 2 and 3 we can see that below 6.66 the median value of a property is 28.67 (USD 1000s), while above is just 18.96 (USD 1000s).

We can plot the first two splits with the code below:

```
ggplot(data=BostonHousing, aes(x=indus, y=dis, color=medv)) +
  geom_point() +
  scale_color_gradientn(colours=c("blue","light
blue","green","orange","red")) +
  geom_vline(xintercept = 6.66, color="red") +
  geom_segment(x=6.66, y=1.96955, xend=30, yend=1.96955, col="red")
```



A simple visual example allowed us to understand how regression tree work. This is the great advantage of regression tree. These algorithms, even though not in their simplest form (i.e. CART) are really powerful and accurate. However, the way they work is also relatively easy to understand and communicate for example to clients.

We can now try CART in a cross-validation framework to compute its accuracy. To create the random folds we can use a simple function from the package caret:

```
k_folds <- createFolds(y=1:nrow(BostonHousing),k=5)

k_folds

## $Fold1
##   [1]   6  12  16  17  22  23  34  35  40  41  49  55  68  75  81  83  88
##  [18]  93  95 100 103 104 107 110 122 123 131 134 137 141 142 143 148 158
##  [35] 162 168 173 184 192 198 200 208 210 216 223 229 237 241 243 246 250
##  [52] 258 260 261 263 265 270 277 279 282 290 292 302 303 315 318 320 321
##  [69] 332 345 349 354 367 372 375 377 386 390 393 400 401 402 411 417 424
##  [86] 431 437 438 444 456 457 458 459 468 474 475 479 488 490 503 504 505
##
## $Fold2
##   [1]   2   3   8  18  20  25  31  32  36  42  43  44  52  58  66  69  78
##  [18]  82  85  86  87  91 118 125 126 138 140 144 145 154 157 163 172 174
##  [35] 175 178 199 201 202 207 217 220 222 224 225 226 231 235 238 249 254
##  [52] 259 264 272 287 288 298 299 311 312 314 326 327 330 337 338 339 340
##  [69] 343 344 356 358 370 373 379 381 387 391 392 394 396 398 405 423 440
##  [86] 441 443 450 462 464 465 467 469 483 484 485 494 495 497 500
```

```
##
## $Fold3
##   [1]  10  11  15  26  27  45  46  47  53  57  63  64  77  79  84  90  92
##  [18]  94  98 101 105 112 117 120 124 132 135 136 139 146 156 159 160 161
##  [35] 170 179 180 186 189 191 193 195 203 204 213 233 236 242 245 253 262
##  [52] 269 276 278 286 300 301 304 305 306 313 317 323 324 325 328 333 334
##  [69] 335 342 348 352 355 357 359 366 380 388 395 399 406 407 413 421 426
##  [86] 430 432 434 435 445 447 452 454 455 461 466 476 477 478 482 502
##
## $Fold4
##   [1]   1   7  13  14  28  30  39  59  60  61  62  67  72  73  74  80  89
##  [18]  99 102 106 108 111 115 116 119 127 128 129 130 133 153 164 166 167
##  [35] 169 176 183 185 187 190 206 209 214 218 219 221 227 230 239 244 247
##  [52] 251 255 256 267 271 275 283 285 293 296 307 308 309 310 316 319 329
##  [69] 331 336 346 347 353 363 369 371 374 382 384 385 404 410 412 416 420
##  [86] 422 428 439 442 446 448 453 471 472 480 487 489 491 492 493 496 498
## [103] 501
##
## $Fold5
##   [1]   4   5   9  19  21  24  29  33  37  38  48  50  51  54  56  65  70
##  [18]  71  76  96  97 109 113 114 121 147 149 150 151 152 155 165 171 177
##  [35] 181 182 188 194 196 197 205 211 212 215 228 232 234 240 248 252 257
##  [52] 266 268 273 274 280 281 284 289 291 294 295 297 322 341 350 351 360
##  [69] 361 362 364 365 368 376 378 383 389 397 403 408 409 414 415 418 419
##  [86] 425 427 429 433 436 449 451 460 463 470 473 481 486 499 506
```

This function takes a vector of row indexes (from 1 to the number of rows of `BostonHousing`) and splits into five non-overlapping random subsets. Now we need to create a `for` loop that iterate through the folds:

```
ERROR <- c()

for(i in 1:5){
  training <- BostonHousing[-k_folds[[i]],]
  test     <- BostonHousing[k_folds[[i]],]

  CART.model <- rpart(medv~.,data=training)

  CART.pred <- predict(CART.model, test)

  ERROR[i] <- MAE(CART.pred, test$medv)
}

mean(ERROR)

## [1] 3.1306
```

First of all, an empty vector named `ERROR` is created, here we will store the mean absolute error for each fold. Then we start the `for` loop, where we iterate from 1 to 5. Inside the loop we create a training and a test set, the first by including all folds except the one which index is equal to `i`, while the other including only the fold with index equal to `i`. This creates a test set that includes one folds, while the training set includes the other four.

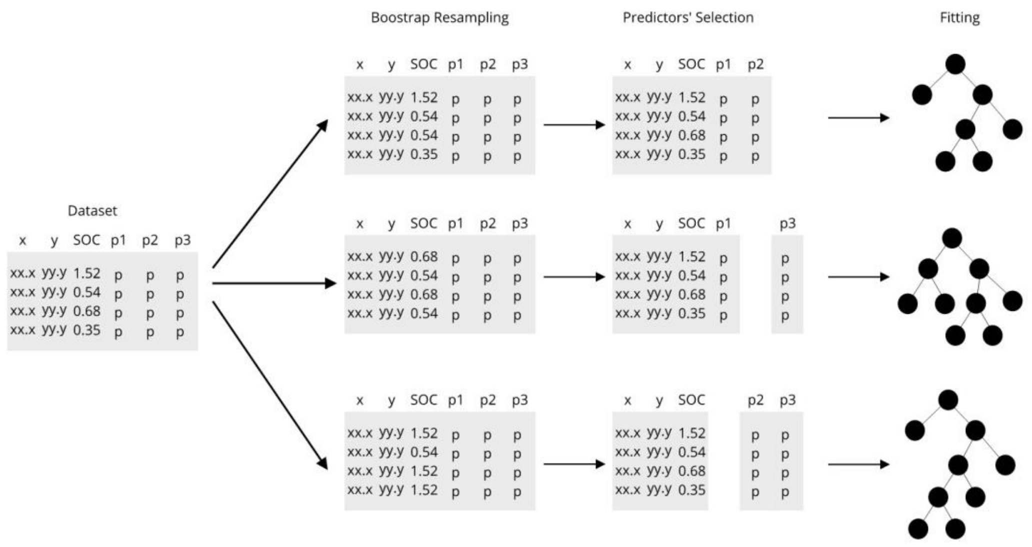
At this point we fit `CART` not to the whole dataset, but only to the training set. Please notice the formula `medv~.`. The dot after the tilde means that we want to include in the formula all the other variables in the `data.frame`.

Finally, the function `predict` is used to estimate values of `medv` in the test set. The final part of the script computes the MAE, and stores it into the vector `ERROR`. This is repeated for each fold.

The numerical value that is returned is the average MAE of the cross-validation. In this case `CART` estimates with an error of around $3200 (remember that `medv` is expressed in USD 1000's).

Random forest

Random forest is another algorithm based on regression trees, and it is probably the most popular to date of this class. It is an ensemble method, meaning that instead of fitting a single tree, it fits multiple trees, thus creating a "forest" of regression trees. It does that by using a technique called bagging, which employs bootstrapping, i.e. resampling with repetitions, and random selection of predictors. In essence, random forest performs a simulation at each run. The simulation creates a series of boostrap replicates of the training set, each slightly different from the original but with equal number of rows (some are repeated). In each simulation random forest also includes in the training only a certain percentage of predictors (usually a third), selected randomly. This creates trees that are not correlated with each other, and this procedure can greatly increase the accuracy as compared to the classic `CART`. Below is a schematic representation of random forest:



Random Forest - Schematic Representation

The beauty of R and its consistent syntax, is that once we know how to fit one model, fitting another is just a matter of changing a couple of functions. For example, let's look at the code to perform a five folds cross-validation on the `BostonHousing` dataset with random forest:

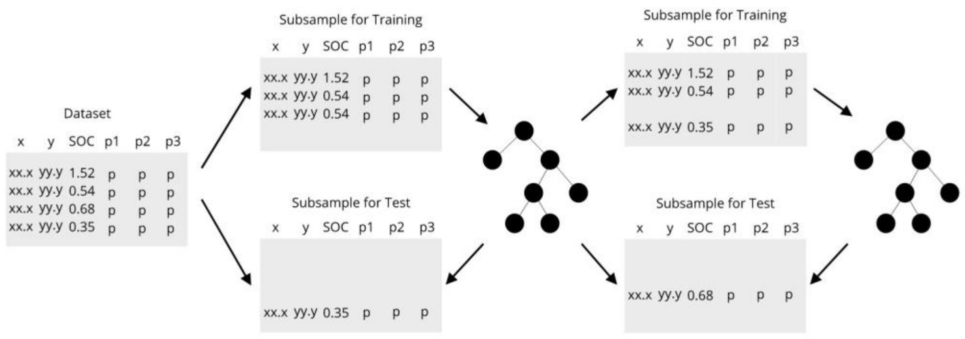
```
ERROR <- c()

for(i in 1:5){
  training <- BostonHousing[-k_folds[[i]],]
  test     <- BostonHousing[k_folds[[i]],]

  RF.model <- randomForest(medv~.,data=training, ntree=1000)

  RF.pred <- predict(RF.model, test)

  ERROR[i] <- MAE(RF.pred, test$medv)
}

mean(ERROR)

## [1] 2.181042
```

We only need to change a few lines to make it work. Clearly we need to include the function `randomForest`, which again works on a formula. This function also takes an option for the number of trees (or simulations) we want to fit. Other than this everything else remains the same.

As you can see random forest can substantially reduce the error of the model, bringing it to around $2200

Boosted regression trees

This algorithm is another powerful example of regression trees. It is again an ensemble method, but it works in a fundamentally different way compared to random forest. Boosting is initialized by fitting a single regression tree to a subset of the entire training dataset and testing its performance on the remaining data. The next iteration fits another tree, but this time focusing on trying to decrease the error from the previous step. This process continues until adding more trees does not provide any improvement in accuracy. Below is a schematic representation of boosted regression trees:



Boosted Regression Trees - Schematic Representation

Let's see the code to do a cross-validation for boosted regression trees:

```
ERROR <- c()

for(i in 1:5){
  training <- BostonHousing[-k_folds[[i]],]
  test     <- BostonHousing[k_folds[[i]],]

  GBM.mod = gbm.step(data=training, gbm.x=1:13, gbm.y=14, tree.complexity =
5, family="gaussian", silent = TRUE, plot.main = FALSE)

  PRED_GBM = predict.gbm(GBM.mod, newdata=test,
n.trees=GBM.mod$gbm.call$best.trees, type="response")

  ERROR[i] <- MAE(PRED_GBM, test$medv)
}

mean(ERROR)

## [1] 2.089127
```

Again the script is very similar to what we used in previous examples. The differences are in the functions `gbm.step`, which trains the algorithm, and `predict.gbm`, which estimates new values. As you can see the function `gbm.step` is a bit different to what we used before. In particular, it does not allow the formula syntax. So we need to specify the indexes in the `data.frame` for target variable and predictors. We can visually check the `BostonHousing` `data.frame` to obtain these values. The variable `medv` is in column number 14, so `gbm.y` takes the value 14; since we want to include all variables `gbm.x` takes values from 1 to 13. In this simple dataset this new syntax is easy to follow; however, for datasets with a lot more predictors you can imagine that knowing the indexes of all predictors could be challenging.

The more complex the algorithm, the more options we need to input. In this case we are only adding the option `tree.complexity`, which controls the maximum number of branches allowed, larger trees may overfit the training set. There are other options, also known as hyperparameters, but it is not the purpose of this lecture to provide details on this aspect.

It seems that boosted regression trees are the most accurate, even though the most complex, with an average error of $2100.

Conclusions

In this lecture we explored the basic concepts behind machine learning regression. Then we looked at the code to fit some of the most popular machine learning algorithms based on regression trees.

Homework

- Test machine learning algorithms on the Crimes and Diet datasets we used in previous lectures.