Summary of STK2100

Håkon Kvernmoen

January 2021

1 Regression

1.1 Basic Overview

To preform a regression fit, data is needed, as well as some "target" or desired output from this data. The goal is to try to predict the target given the data. First, we record one type or multiple types of data. If we have recorded p different types of data, we store these in a row vector $\mathbf{x}_i = \{x_1, ..., x_p\}$. This is a data point, located in the p-dimensional space of all the different things we think can predict the desired target. We also need record the target (the thing we want to model, output). This is a number y_i that corresponded to the data $\mathbf{x_i}$.

One data-point is useless, so we have to record multiple different data-points as well as targets. If we have collected n data-points we store these as rows in a $n \times (p+1)$ data-matrix, $\mathbf{X} = (1, \mathbf{x_1}, ..., \mathbf{x_n})^T$. The first column is filled with ones to allow for a constant term in the model. We also need targets to each of these data-points, stored as a $n \times 1$ column vector $\mathbf{y} = (y_1, ..., y_n)^T$.

1.2 Linear regression

We assume a linear relationship between the p different types of data with the target. To do this we make a $(p+1) \times 1$ parameter column vector $\boldsymbol{\beta} = (\beta_0, ..., \beta_p)^T$.

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

The $\boldsymbol{\epsilon}$ here represents an error term, since our assumption of a linear relationship is probably not correct and there might be noise or systematic errors in the data. Due to this error term we cannot know $\boldsymbol{\beta}$ exactly, so we approximate it!

$$\hat{\mathbf{y}} = \mathbf{X}\hat{\beta}$$

When minimizing the distance between the $\mathbf{y}$ and $\hat{\mathbf{y}}$ points, we obtain the expression for the $\hat{\beta}$ values corresponding to the lowest error (still assuming a linear relationship). This is called the *deviance* (D), as it quantifies the discrepancy between the observed and fitted values. This is the *least squares estimate.*

$$D(\boldsymbol{\beta}) = ||\mathbf{y} - \hat{\mathbf{y}}||^2 \Rightarrow \hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

1.3 Regularization to Linear regression

Linear regression is a 1-to-1 fit, and the only real constraints/modifications we can apply is removing explanatory variables or variable transformations. Two of the more common methods to en-pose stricter assumptions is *Ridge* and *Lasso* regression. To use these models, the explanatory variables has to be standardises. We could normalize the explanatory variables (map to [0,1]), but we want to keep model interpretability. Note that the standard is unitless.

$$x_{stanard} = \frac{x - \mu}{\sigma}$$

1.3.1 Ridge

In ridge regression we preform ordinary linear regression with an addition to the deviance (D).

$$D(\boldsymbol{\beta}, \lambda) = ||\mathbf{y} - \hat{\mathbf{y}}||^2 + \lambda||\boldsymbol{\beta}||^2$$

With $\lambda \geq 0$, a tuning parameter where a good value is essential. The extra term here is called a shrinkage penalty and is small when the coefficients $\beta_1, ..., \beta_p$ are small. $\lambda = 0$ gives no shrinkage and we have least squares, $\lambda \to \infty$ makes all coefficients (not intercept) go towards zero. The shrinkage penalty does NOT include the intercept β_0 since we want the intercept to represent the mean of the response when $x_{i1} = x_{i2} = ... = x_{ip} = 0$.

The benefit of this approach is that tough the coefficients decrease as a whole with λ, some might stay the same and some might even increase. Thus this shrinkage penalty "pulls out" the most significant coefficients while suppressing the non-significant. Bias increases with λ, but variance can decrease.

1.3.2 Lasso

Lasso is the counter part to Ridge. One problem with Ridge is that coefficients will not directly be set to 0, but only approach it for large λ. It is thus not fit for variable selection. This is not necessarily bad for predictions (as un-significant variables will have little impact on the result), but is bad for model interpretability (especially if p is large). The deviance for Lasso is

$$D(\boldsymbol{\beta}, \lambda) = ||\mathbf{y} - \hat{\mathbf{y}}||^2 + ||\boldsymbol{\beta}||_1$$

Again with $\lambda \geq 0$ and a good value is again essential. Where $||\cdot||_i$ means the sum of the absolute value, raised to the i'th power of all its components ($||\cdot||$ implies $\sqrt{(||\cdot||_2)^2}$).

1.3.3 Comparing Ridge and Lasso

Lasso has a clear advantage in that it can preform variable selection, setting some coefficients to exactly zero. This of course have limitations as, i.e. if two or more features are highly co-linear Lasso will select on of them randomly with the rest set to zero (Bad for model interpretability). In (very general) terms Ridge is best if you have a small amount of explanatory variables and/or a lot of co-linearity. Lasso shines when there are more explanatory variables and/or where a simpler model is desired.

2 Classification

2.1 Bayes classifier

The Bayes classifier is a classification rule that minimizes misclassification. The most common approach to estimate a model $\hat{f}$ is to minimize the *error rate* given by

$$L(y_i, \hat{y}_i) = \frac{1}{n} \sum_{i=1}^{n} I(y_i \neq \hat{y}_i)$$

If all the fitted values matches the actual value (on this training data), the training *error rate* is zero. Often one is most concerned by minimizing the testing *error rate*, that is the previous equation evaluated on data the model have not seen before (not used for training).
It can be shown that this very problem has a very simple answer, the Bayes classifier. It simply assigns each observation to the most likely class, given the data. Given a new data point x_0, it should be assigned to the class j with the largest:

$$P(y = j | X = x_0)$$

In the binary case ($j = 0, 1$) this correspond to assigning x_0 to class 1 if $P(y = 1|X = x_0) > 0.5$. For the binary case this draws a decision boundary at $P(y = 1|X = x_0) = 0.5$, effectively splitting the two classes by a line (2D) or hyper plane (nD). The error rate at $X = x_0$ is $1-\text{Max}_j P(y = j|X = x_0)$ and thus the overall error rate is given by:

$$1 - \mathbb{E}[\text{Max}_j P(y = j|X = x_0)]] \tag{1}$$

2.2 Logistic regression

Logistic regression is kind of a classification method **and** a regression method. Pro die hard statistician's want to call it a regression, but I put it here since the goal of the method is to *classify* a target. However the process of achieving the classification model, requires a *regression.* If we want to preform a logistic regression on two data variables $\mathbf{x} = (x_1, x_2)$ with a probability p of outcome A. We define $\beta(x_1, x_2) = \beta_0 + \beta_1 x_1 + \beta_2 x_2$ (can of course be generalized to more)

$$logit(p) = log\left(\frac{p}{1-p}\right) = \beta(x_1, x_2)$$

where *logit* maps the probability $p \in (0, 1)$ to $logit(p) \in (-\infty, \infty)$. Solving this expression for p gives.

$$p(x_1, x_2) = \frac{e^{\beta(x_1,x_2)}}{e^{\beta(x_1,x_2)} + 1} = \frac{1}{1 + e^{-\beta(x_1,x_2)}}$$

Now $\beta(x_1, x_2)$ can be fitted on $(-\infty, \infty)$ while the probability p for outcome A is still $(0, 1)$. To evaluate a threshold $\mathcal{T}$ is given, typically chosen to be 0.5 (Bayes classifier). If the probability p is bigger than the threshold, classify as A, else not A. In our example.

$$\text{if } p(x_1, x_2) > \mathcal{T} \to A, \text{ else} \to \text{not } A$$

2.3 K-Nearest Neighbor

K-Nearest Neighbor (KNN) can not calculate the conditional distribution of Y given X and can thus not use the Bayes classifier. It deals with classifying a new point x_0 to the same class as its K nearest neighbours (in feature space). The choice of K is thus important, as it considers how many points should be considered when calculating the probability.

$$P(y = j|X = x_0) = \frac{1}{K} \sum_{i \in \mathcal{N}_0} I(y_i = j)$$

With a small K (say 5), the model only considers a handful of neighbours and the model has a low bias but high variance. In the extreme case $K = 1$ only one point is considered. The train error in this case would always be zero as it classifies each point to itself. Larger K's has a higher bias, but often a lower variance. The test error rate, plotted as a function of K, often has a characteristic U-shape, where it preforms bad for low K due to the high variance, finding some sort of minimum at a bias variance trade-off, before increasing again due to the high bias. The train error on the other hand will most likely grow with K (with a guaranteed minimum at $K = 1$).

2.4 Discriminant Analysis

Using Bayes theorem, we can construct classification with more $K \geq 2$. When given the data, the goal is to somehow split the dataset with a line and project the datapoints on this line, with some criteria on how the line is drawn.

2.4.1 Linear (LDA)

Assuming that we have on explanatory variable ($p = 1$) it can be shown from Bayesian statistics that we should assign x to the class where $\delta_k(x)$ is largest (assuming x is draw from a Gaussian distribution). The *linear* comes from $\delta_k(x)$ being linear w.r.t. x.

$$\delta_k(x) = x \cdot \frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2} + log(\pi_k)$$

Where μ_k is the mean of the k'th class, the variance σ^2 is assumed to be equal for all the classes and π_k is the probability of class k. The LDA estimates for these are.

$$\hat{\mu}_k = \frac{1}{n_k} \sum_{i:y_i=k} x_i \quad , \quad \hat{\sigma}^2 = \frac{1}{n-K} \sum_{k=1}^{K} \sum_{i:y_i=k} (x_i - \hat{\mu}_k)^2 \quad , \quad \hat{\pi}_k = \frac{n_k}{n}$$

Where $i : y_i = k$ means sum over all points belonging to class k

This can be expanded to handle more explanatory variables ($p > 1$), but dependence between variables may pose a problem. Therefore instead of p Gaussian distributions, one for each variable, we rather consider a *multivariate Gaussian* distribution. With a shared covariance matrix between all the classes $\mathbf{\Sigma}$ (a $p \times p$ matrix) and a mean for each variable $\mathbb{E}[\mathbf{X}] = \boldsymbol{\mu}$ (a vector of length p holding all the means for each variable) the Bayesian classifier now assigns an observation $\mathbf{X} = \mathbf{x}$ to the class where to following is highest.

$$\delta_k(\mathbf{x}) = \mathbf{x}^T\mathbf{\Sigma}^{-1}\boldsymbol{\mu}_k - \frac{1}{2}\boldsymbol{\mu}_k^T\mathbf{\Sigma}^{-1}\boldsymbol{\mu}_k + log(\pi_k)$$

With $\boldsymbol{\mu}_k$ being the mean vector of x for a specific class. If $p = 2$ we do not get one decision boundary as for $p = 1$, but rather 3 separating the 3 classes. These boundaries are the set of $\mathbf{x}$'s where $\delta_k = \delta_l, l \neq k$ (equal propability for both classes).

2.4.2 Quadratic (QDA)

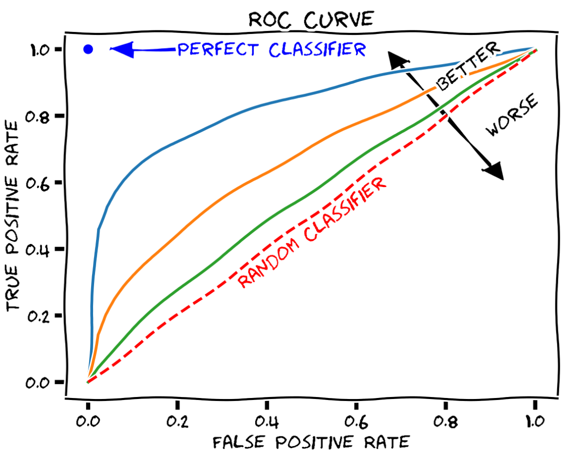
Instead of assuming that all the explanatory variables share the same covariance matrix, independent of class, we will now assume that each class has a specific covraiance matrix $\mathbf{\Sigma}_k$. With some data point, the Bayesian classifier assigns to the class which maximizes:

$$\delta_k(\mathbf{x}) = -\frac{1}{2}\mathbf{x}^T\mathbf{\Sigma}_k^{-1}\boldsymbol{x} + \boldsymbol{x}^T\mathbf{\Sigma}_k^{-1}\boldsymbol{\mu}_k - \frac{1}{2}\boldsymbol{\mu}_k^T\mathbf{\Sigma}^{-1}\boldsymbol{\mu}_k + log(\pi_k)$$

This is *quadratic* since one of the terms include $\mathbf{x}$ twice.

2.5 ROC curves

ROC (receiver operating characteristic) curves is a graphical way to visualize how a binary (true/false) classification preforms. The y-axis contains the *true positive rate* (TPR) and the x-axis the *false positive rate* (FPR). These are defined as $TPR = TP/(TP + FN)$ and $FPR = TN/(TN + FP)$. The thing that varies (drawing the curve) is different thresholds $\mathcal{T}$.



An example of a ROC curve. If we were to guess 50/50 (i.e. no prediction model) our classifier would simply draw a straight line from (0,0) to (1,1). If the threshold was extremely high (very close to 1) we would have no FP (since it always guesses *false*) but no TP by the same reason. Decreasing the threshold to extremely low (very close to 0) has the other effect, since it always guesses *true*. If the model has any meaning full predictions the ROC curve should lay over the random classifier line. A perfect classifier would get all TP and no FP and is thus located at (0,1).

Figure 1: ROC curve

Another measure for this is the AUC (area under curve). If $AUC = 0.5$ we have a random classifier, and $AUC = 1$ is a perfect classifier. Anything in between is expected if the model preforms meaning full predictions.

3 Basis functions and "Non-linear" models

Many of these methods uses basis functions, a tool that makes classifying and distinguishing between different models easier. In general they take the form with k explanatory variables.

$$y_i = \beta_0 + \beta_1 b_1(x_i) + ... + \beta_k b(x_k) + \epsilon_i$$

This is just linear regression and can easily be computed using a LS fit. The $b(\cdot)$ functions serves as some transformation of the data. For instance $b_j(x_i) = x_i^j$ is a basis function for polynomial regression.

3.1 Step Functions

Step functions divides X into k bins with endpoints $c_1, ..., c_k$. This makes $k+1$ "categories" to place our data in, given by the $k+1$ indicator functions $C_0(X) = I(X < C_1), ..., C_j(X) = I(c_j < X < c_{j+1}), ..., C_k(X) = I(c_k \leq X)$. Essentially a data point is given a fixed value when it falls between a fixed interval. Note that $C_0(X) + ... + C_k(X) = 1$ since one of the indicator functions has to equal 1. This method is good if there are natural splits in the data, but looses a lot of data-trends. The corresponding basis function is

$$b_j(x_i) = I(c_j < x_i < c_{j+1})$$

3.2 Splines

Splines depend on dividing X into k cuts and fitting a polynomial of degree d between each cut, demanding that the $d-1$ derivative is continuous. Splines are preferred to polynomial regression since one can achieve high flexibility without high degrees.

3.2.1 Knot placement

The natural approach to knot placement is to place more knots where the target is changing rapidly (to increase flexibility). In practice however, placing them uniformly often works best and is almost always done.

Deciding the number of knots can be tricky, as too many will give too much flexibility (low bias, high variance) and too few will be too strict (high bias, low variance). Cross validation is a good option to test different values, with lets say 5-fold CV being preformed for a given amount of knots.

3.2.2 Cubic splines

The most common degree is $d = 3$, appropriately named *cubic splines.*

If there were no constraints, one cut would give 8 degrees of freedom for a cubic spline. Due to the constraints demanding the second derivative to be continuous we end up with 5 degrees of freedom (continuous 0'th, 1st and 2nd derivative). Thus a know only adds 1 degree of freedom and in general with k cuts we have $4+k$ degrees of freedom. We often write the basis function for cubic splines as. In general for basic splines use from `gam`, `bs()`.

$$h(x_i, \xi_j) = (x_i - \xi_j)_+^3 = \begin{cases} (x_i - \xi_j)^3 & \text{if } x_i > \xi_j \\ 0 & \text{otherwise} \end{cases}$$

Where ξ_j is a knot. To fit a cubic spline with k knots. To fit this model, one considers the constant term and (global) X, X^2, X^3. In addition for each knot $j = 1, ..., k$ fit $h(X, \xi_j)$, in total $4+k$ degrees of freedom.

3.2.3 Natural splines

Natural splines is identical to normal splines, but we demand that the second derivative is zero at the end-points. This results in the model not going crazy for the extrapolated values at the same time not disrupting its smoothness. R comand: From `gam`, `ns()`

3.2.4 Smoothing splines

Smoothing splines takes a little different approach than normal splines. The goal is to minimize RSS as much as possible. The problem arises when considering the best fit is simply an interpolation of all the training data $x_1, ..., x_n$. This is of course highly overfitted, so there needs to be some sort of penalty for this high variability. One approach is find the g that minimizes:

$$\sum_{i=1}^{n}(y_i - g(x_i))^2 + \lambda \int_{x_1}^{x_n} g''(t)^2 dt$$

This is simply normal RSS, with an additional integral over the second derivative of g (over the whole data set), with λ being a tuning parameter. This takes the form of a "Loss+Penalty" formulation. The second derivative represents the *roughness* of the function. The function g is squared to not discriminate against negative concave functions and the integral sums over the whole data-range. It can be shown that this represents a natural cubic spline with knots at $x_1, ..., x_n$ with an additional penalty demanding that the function should be smooth.

With $\lambda = 0$ there will be no roughness penalty and the points $x_1, ..., x_n$ will be exactly interpolated. With $\lambda \to \infty$ the fit will be perfectly smooth and just be a straight line (equal to the normal LS fit). Thus the *effective degrees of freedom* decreases from $[n, 2)$ with $\lambda \in [0, \infty)$. We do not need to choose the location of the knots as with normal splines or natural splines, but a good λ is needed. This can be done with CV, but there is a closed formed solution for LOOCV.

$$RSS_{LOOCV}(\lambda) = \sum_{i=1}^{n}\left(y_i - \hat{g}_\lambda^{(-i)}(x_i)\right)^2 = \sum_{i=1}^{n}\left(\frac{y_i - \hat{g}_\lambda(x_i)}{1 - [\mathbf{S}_\lambda]_{ii}}\right)^2$$

$\hat{g}_\lambda^{(-i)}$ is the fit of g using all the data except i, then evaluated at i. Where $\mathbf{S}_\lambda$ is the solution to $\hat{\mathbf{g}}_\lambda = \mathbf{S}_\lambda \mathbf{y}$. $\mathbf{S}_\lambda$ is statistically difficult, but there is a formula for this (INSERT R COMMAND HERE). In addition the effective degrees of freedom are $df_\lambda = Tr(\mathbf{S}_\lambda)$. R command from `gam`, `s()`

3.3 Generalized Additive Models (GAMs)

GAMs is a natural extension of linear regression. To allow for non-linear relationships between the explanatory variables, each component is modeled by a smooth non-linear functions $f(\mathbf{x}_j)$. It is important that these function still are *additive* such that the response $\mathbf{y}$ can be expressed as a sum of these non-linear functions. The Model then becomes.

$$y_i = \beta_0 + \sum_{i=1}^{p} f_j(x_{ij}) + \epsilon_i$$

- Pros
  1. Since f_j is non-linear, we do not need to manually try different transformations of X_j as we need in a linear regression setting
  2. Since we still have a additive model, investigating the effect of variable X_j on the response Y is easy, no loss of interpretability.
  3. The smoothness of f_j can be summarized via degrees of freedom
- Cons
  1. Since the model is restricted to be additive, interacting terms of the form $X_j \times X_k$ are still missed if they are not explicitly included.

In the `gam` library the `gam` function allows for GAM fits. `s(variable, degree)` fits smoothing splines

4 Tree-based Methods

These methods are based on segmenting the predictor space into a number of simpler regions. The rules for the segmentation are summarized in a decision tree giving the name tree-based methods. They can be applied to both regression and classification. Each region is called a *terminal node.* Where the split happens is called a *internal node*, opening up two new *branches.*

4.1 Regression

When building a regression tree there are two main steps.

1. Use all the features $\mathbf{x}_1, ..., \mathbf{x}_p$ to divide the data into J non-overlapping regions $R_1, ..., R_J$.
2. When predicting using a new data point $\mathbf{x}_k$ use the decision tree to find the right regions R_j. The predicted value will be the mean of all the training data points belonging to region R_j

Step 1 can be done multiple different ways, but a common approach is to divide the regions using hyper-cubes (rectangles for two explanatory variables, cubes for three) that minimizes the RSS.

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_i)^2$$

A common top-down (starting from the top), greedy (best split is done at a particular step, not looking forward for better splits) approach called *recursive binary splitting.* This is very computationally expensive. Starting at the top, for each explanatory variables x_j propose an *internal node* with the cut point s. It then makes a split into to regions corresponding the.

$$R_1(j, s) = \{x | x_j < s\} \text{ and } R_2(j, s) = \{x | x_j \geq s\}$$

And choose the j, s combination that minimizes RSS. That is minimizes ($\hat{y}_{R_1}, \hat{y}_{R_2}$ are the means of the traning observations in $R_1(j,s), R_2(j,s)$). Repeat this at each new *branch* and continue until we have a pre-set number of J *terminal nodes* (regions) or another stopping criteria such as maximum n number of observations for each region. Surprisingly this can be done quite quickly is p is not too large and J is not stupidly large.

$$\sum_{i:x_i \in R_1} (y_i - \hat{y}_{R_1})^2 + \sum_{i:x_i \in R_2} (y_i - \hat{y}_{R_2})^2$$

4.2 Classification

Classification trees are very similar to regression trees, however instead of predicting a value based one the mean in each region, we instead classify to the class containing most instances in each region. In addition we can not use RSS, as it does not have any significant meaning in classification. A natural choice would be the *classification error rate*, the fraction of training observations in each region that to not belong to the most common class. This is often not sufficiently sensitive, and to other measures of region "purity" is often prefred. The *Gini index* and *cross-entropy.*

$$\textit{Gini index: } G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$
$$\textit{Cross-entropy: } D = -\sum_{k}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

Where $\hat{p}_{mk}$ is the proportion of the observations from the mth region that are from the kth class. The gini index is small if $\hat{p}_{mk} \approx 1$ or $\hat{p}_{mk} \approx 0$. Thus if a region contains a lot of instances of the same class ($\hat{p}_{mk} \approx 1$) or very few instances of the same class ($\hat{p}_{mk} \approx 0$) the Gini index will report a small error. The same follows for the cross-entropy ($log(1) = 0$) and the minus sign is there since $0 \leq \hat{p}_{mk} \leq 1 \rightarrow \log(\hat{p}_{mk}) \leq 0$. Both these measures takes on small values if the region is *pure.*

4.3 Tree Pruning

If the amount of regions are large, this is prone to over fitting. One can of course decrease the number of regions J or set a RSS-decrease threshold for each split. Even tough this will result in smaller trees, one can fall into the trap where one misses a seemingly useless split early on that turns out to reduce RSS by a lot later down the tree. A better candidate is *cost complexity pruning/weakest link pruning* (a rose by any other name would smell as sweet). We begin by growing a very large tree T_0. We want to check different MSE of sub-trees of T_0, but of course the amount of combinations are ridiculously large. By considering a sub-tree with T *terminal nodes* and a non-negative tuning parameter α. For different alpha's we train.

$$\sum_{m=1}^{T} \sum_{x_i \in R:m} (y_i - \hat{y}_{R_m})^2 + \alpha T$$

$\alpha = 0$ leaves us (pun intended) with the un-pruned tree T_0 and increasing α penalises the tree for having a lot of branches. Now that we have trees for different amount of leaves one can use k-fold CV to plot the MSE against the number of *external nodes.*

4.4 Bagging

In general tree based methods suffer from high variance. This means that if we train 100 different trees of 100 different subsets of a data set, the models might look very different. To reduce the high variance, a technique called *bagging* is often applied.
The idea of bagging is to pick B number of subset from the original data. For each B, we train a deep tree model $\hat{f}_b(x)$. Finally we average over all $\hat{f}_b(x)$ to achieve our final model.

$$\hat{f}_{bag}(x) = \sum_{b=1}^{B} \hat{f}_b(x)$$

Since each tree is grown deep, the individual trees have a high variance but low bias. Averaging over many trees aims to reduce the variance. For regression one simply averages over the predicted target $\hat{y}$ to find the bag target. Classification on the other, might use something like *majority vote*, the target that has the most trees classify to its class is chosen. The choice of large B's often does not lead to over fitting. One can chose a high enough B such that the train error stabilizes.

4.4.1 Out-of-Bag error estimation (OOB error)

In stead of using CV or validation sets to evaluate the test error of a bagged tree, one can use the OOB error. For n data points pick out n randomly (allowing for picking the same point more than once). On average this chooses $1 - e^{-1} \approx 2/3$ of the data points. For each tree one can use the remaining $1/3$ points to produce the test error. For the ith OOB observation one can average over all the predictions (regression) or majority vote (classification). This is a good choice when cross-validation is too computationally expensive.

4.4.2 Variable Importance Measures

Since we have a B trees, we loose a lot of interpretability. One can iterate over all B trees and for each variable split, record the *decrease* in RSS or Gini index. The total amount can then be displayed for each variable giving some hints to what variables are important for the collection of trees.

- Pros
  1. Bagging typically results in improved accuracy over predictions using a single tree
  2. Can evaluate test error using OOB error estimation. No need for CV
  3. Using variable importance one can "regain" some of the lost interpretability.
- Cons
  1. Since there are a lot of trees that average/vote for (regression/classification) a prediction, we loose a lot of the interpretability inherent in the tree model.

4.5 Boosting

Boosting is a way to let new grown trees learn from its predecessors mistakes. One sets the number of trees to train B and the number of splits d (and thus $d + 1$ terminal nodes). Each tree is fitted on the data X, but the target is the residuals from the previous tree. This can be implemented for regression and classification, but we will only deal with regression. The algorithm can be outlined as.

1. We set $r_i = y_i$ for all i. This can be seen as the residuals of the empty tree (represented by $\hat{f}(x) = 0$ if you will).
2. For b = 1, ... , B
   (a) Fit a tree $\hat{f}_b$ (with the d splits) with the features X and target r.
   (b) Record tree $\hat{f}_b$
   (c) Update residuals $r_i \leftarrow r_i - \lambda \hat{f}_b(x_i)$
3. The final model is then $\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}_b(x)$

4.6 Random Forest

In bagging the B trees are somewhat correlated. Since at every split all trees considers all the p predictors to decrease RSS/Gini index the trees might look very similar. Random forest tries to *decorrelate* the trees, by limiting the amount of possible variables to split. At each split a random sample of $m > p$ possible predictors are chosen. Commonly $m = \sqrt{p}$ is chosen, but can be varied to optimize performance, $m = p$ reduces to bagging.
This is (counter-intuitively) often a good choice since a strong predictor at the top of the tree will produce similar trees if all p predictors are considered. Limiting the choice to m predictors for each split motivates more diverse trees. As in bagging large Bs does not overfit and can be chosen large enough for test error to stabilize.

5 Support Vector Machines

Support vector machines is a loose term containing three models. All the methods are used for *classification*, but can under some circumstances to modified to preform *regression*. All these methods need *feature scaling*

1. Maximal margin classifier
2. Support vector classifier
3. Support vector machine

5.1 Maximal margin classifier

For a binary classification (2 targets, true/false situation) we try to split the features space using a *hyperplane*. A hyperplane is a flat $p - 1$ dimensional affine (it dose not need to pass the origin) subspace of a p dimensional feature space. $p = 2$ gives a line, $p = 3$ gives a plane and so on. Given a data point (column vector) $x_i = (x_{i1}, x_{i2}, ..., x_{ip})$ the hyperplane can be drawn by the line.

$$f(x_i) = \beta_0 + x_i^T \beta \quad \text{plane at } f(x_i) = 0$$

For a new data-point x_j it can be classified to two different classes $\hat{y}_j = \{-1, 1\}$ by.

$$f(x_j) = \begin{cases} f(x_j) > 0 & \hat{y}_j = 1 \\ f(x_j) < 0 & \hat{y}_j = -1 \end{cases}$$

The magnitude of $|f(x_j)|$ can be used to decide how "sure" the classifier is about the class. Thus $|f(x_j)| >> 0$ means we are very sure and $|f(x_j)| \approx 0$ means we are not. There is actually an infinite amount of possible planes, as very tiny rotations/translations of $f(x)$ can still divide the two classes. To pick the "best" plane one finds the plane that has the largest margin. With the margin as M one tries:

$$\underset{\beta}{\text{Max}}\ M \qquad \text{with} \qquad \sum_{j=1}^{p} \beta_i^2 = 1 \qquad \text{such that} \qquad f(x_i, \beta) \geq M$$

To read this one can go backwards. The last function $f(x_i, \beta) \geq M$ means that if x_i belongs to class 1 the point will be on the right side of the plane $f(x_i, \beta) > 0$ and have at least the distance M ($f(x_i, \beta) \geq M$) from the plane, with negative sign for class -1. This should hold for all $i = 1, ..., n$. The constraint on the β's ensures the $f(x_i, \beta)$ is the distance from the hyperplane to the observation. Lastly we find this margin M with varying the hyperplane trough the β's.

5.2 Support Vector Classifier

To draw a hyperplane splitting the two classes is of course not always possible. In the cases that we can draw a hyperplane, we often don't want to draw a margin that exactly splits the training data. Such a margin is very sensitive to points close to the margin, thus the decision boundary can by extremely influenced by one point (resulting in overfitting). We now allow some points to be inside the margin (*soft-margin classifier*) and even on the wrong side of the hyperplane.

This is done by adding a *slack variable* $\epsilon = \epsilon_1, ..., \epsilon_n$ to each of the training data-points. These are also tweaked to find the optimal M such that we now have $n + p$ parameters to tweak. The optimization is identical to the maximal margin classifier but with the modification:

$$f(x_i, \beta) \geq M(1 - \epsilon_i) \qquad \text{with every } \epsilon_i \geq 0 \qquad \text{such that} \qquad \sum_{i=1}^{n} \epsilon_i \leq C$$

Where C is a positive tuning parameter. Classification is equal to the maximal margin classifier, assign class according to which side of the hyperplane x_j is located. C is a budget. Larger C means more freedom to violate the margin (or even hyperplane). If $\epsilon_i > 0$ the margin is violated and if $\epsilon_i > 1$ then x_i is on the wrong side of the hyperplane. $C > 0$ means that a maximum of C observations are located on the wrong side of the hyperplane. The margin M tends to grow with the value of C. In the optimization in turns out that only the observations directly on or in violation of the margin (*support-vectors*) influences the classifier. With a high C and thus a high margin, many observations violate the margin and thus we have a low variance but potentially high bias. The opposite is true for a small C.

5.3 Support Vector Machine

Support vector machines (SVM) tries to deal with non-linear boundaries. There are many different ways to do this. The most obvious solution is to make new observations by interaction terms, higher order polynomials or special function with the old observation. The problem with this is that the solution space will be enlarged and can result in very costly computations.

Instead of increasing the observation space, SVM obtains non-linear boundaries by the use of *kernals*. A kernal defines a new inner product. It turns out that the solution to the support vector classifier only involes inner products between the support vectors. Let $\mathcal{S}$ hold the indicies on the support vectors. Then any solution can be written as the function:

$$f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \alpha_i K(x, x_i)$$

Where K is the kernal, defining the inner product. It can take many forms, a few of them are:

- Linear: $K(x_i, x_j) = \sum_{k=1}^{p} x_{ik}x_{jk}$. This is a support vector classifier
- Polynomial: $K(x_i, x_j) \left(1 + \sum_{k=1}^{p} x_{ik}x_{jk}\right)^d, d > 1$
- Radial: $K(x_i, x_j) = exp(-\gamma \sum_{k=1}^{p}(x_{ik} - x_{jk})^2), \gamma > 0$

6 Unsupervised Learning

Unsupervised learning differs from supervised learning in one import note. The target (that we are trying to predict) is not tagged by a human. This means that instead of predicting y from X, y is unknown and thus fundamentally changes the core problem. As we do not know the target y, prediction is uninteresting. Rather, the goal is to discover "interesting things" about the data X. We are here concerned with two methods of finding "interesting things", namely *Principal Component Analysis* (PCA) and *clustering*.

Both these methods need feature scaling. The natural choice is to *standardize* them (subtract mean and divide by standard deviation for each variable).

6.1 Principal Component Analysis

The idea of principal component analysis is to make linear combinations of the original variables to maximize the variance. There are two views of this problem, both making use of the fact that the features are standardized. With $\mathbf{X}$ as a $n \times p$ matrix, we construct new points using:

$$z_{i1} = \sum_{j=1}^{p} \phi_{j1} x_{ij}$$

The coefficients ϕ_{i1} are called the loading's of the first principal component. The second would be ϕ_{i2}, the third ϕ_{i3} and so on. As x_{ij} is standardized, this is equal to (if not standardized) $x_{ij} - \mathbb{E}[x_j]$. Then if we square $z_i 1$ and find $\mathbb{E}[x_{i1}]$ we obtain a sort of weighted variance over all p features. We can now maximize the variance by tweaking the loading's $\phi_{11}, ..., \phi_{p1}$.

$$\underset{\phi_{11},\dots,\phi_{p1}}{\text{Max}} \left(\frac{1}{n} \sum_{i=1}^{n} z_{i1}^2 \right) \text{ subject to } \sum_{j=1}^{p} \phi_{j1}^2 = 1$$

The constraints on ϕ_{j1} is there to make sure the variance can not be arbitrarily large. Now to find the second PC one preforms the same optimization problem, with the added constraint that the second PC should be uncorrelated with the first. The second view is to note that $\Sigma = \mathbf{X}^T\mathbf{X}$ now forms a co-variance matrix (since the data has been standardized). The eigenvectors of this matrix correspond to the PC's, with the first having the largest eigenvalue, the second the second largest and so on. Since Σ forms a symmetric matrix, we know that all the eigenvectors should be orthogonal! (Assuming the eigenvalues are not equal). For a data set with p features we have a maximum of $p-1$ PC's.

Often we wish to use quite a low number of PC's. The goal of PCA is to reduce the dimensionality of the feature space, while still explaining a good fraction of the variance.

6.2 Clustering

Clustering deals with partitioning the feature space into subgroups, or *clusters*. This course deals with 2 types of clustering methods, *K-means* and *hierarchical*.

6.2.1 K-means

K-means clustering divides the data into K non-overlapping clusters. The value of K is chosen before the algorithm is preformed. We write the indices of each data point as a member of one and only one cluster $C_1, ..., C_k$. Since every data point should belong to some cluster we must have $C_1 \cup .... \cup C_k = \{1, ..., n\}$. This means that the union of every set must contain every data index (no one is left out). In addition $C_i \cap C_j = \emptyset, i \neq j$, meaning the overlap of any two sets must be an empty set. To best split the clusters, we need to define a measure that captures the amount of which the data points within a cluster differs from one another. There are many ways to define this measure, but the most common uses the squared Euclidean distance.

$$W(C_k) = \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2$$

$|C_k|$ is the number of observations inside the kth cluster. The sum goes over all p-features (finding the Euclidean distance). Then one finds the distance between the ith and i'th data point inside the cluster. We want to select the clusters such that $W(C_k)$ is as low as possible over all clusters. Thus

$$\underset{C_1,\dots,C_k}{\text{Min}} \left(\sum_{k=1}^{K} W(C_k) \right)$$

Finding the K clusters such that the sum of $W(C_k)$ is minimized is very difficult, since there are extremely many ways to partition the clusters. Luckily one can use randomness in algorithms to produce a *pretty good* solution (Locally the best pick, dependent on the random initialization).

1. Each observation gets randomly selected to one of the K classes. This is the random part that can produce different results
2. Iterate until assignments to new clusters stop
   (a) Each cluster calculates its *centroid*. This is the mean distance (in p-dimensional feature space) from all the data points in each cluster.

(b) Go over all data points and calculate the distance to each of the K centroids. Assign to the cluster where the distance from the data point to the centroid is the smallest.

The algorithm ensures to find a solution, but is dependent on how the data points are randomly assigned. Thus, one should run the alogrithm multiple times and choose the best clustering based on the minimization over $C_1, ..., C_k$.

6.2.2 Hierarchical

Hierarchical clustering algorithms (HCA) is an alternative to K-means that does not require to choose K **before** the algorithm is preformed. After the algorithm is preformed, one uses a *dendrogram* to choose K (how many clusters you want) based on the shape/properties of the dendrogram.

To preform HCA we have to specify a dissimilarity measure. In the K-means section we use the Euclidean distance, but other measures can be used as well. In addition we need to select a linkage. This expands the idea of dissimilarity between observation to dissimilarity between clusters. In the K-means section, we used the *centroid* of the clusters, but other measures exists (see list after dendrogram explanation). The recipe for HCA can either bottom-up (agglomerative) or top-down (divisive). Agglomerative puts each observation in it's on cluster, and merge one cluster at each iteration until there is only one cluster left. Divisive starts with all observations in one cluster, and splits the clusters at each iteration until each observation has its own cluster. The most common approach is agglomerative and is the one used in this course:

1. Each observation gets it's own cluster. Calculate the dissimilarity between all $\binom{n}{2}$ clusters.
2. For i = n, ... , 2
   (a) For each cluster, search the pair-wise dissimilarity between every other cluster. Merge the clusters with the lowest dissimilarity (most similar) to form a new cluster. We have now reduced from i clusters to $i-1$ clusters. This is a new "fuse" on the dendrogram and the height corresponds to the dissimilarity between the clusters.
   (b) Use your selected linkage to find the dissimilarity between each of the now $i-1$ clusters. Repeat until $i = 2$, so that we end up with $i - 1 = 1$ clusters.

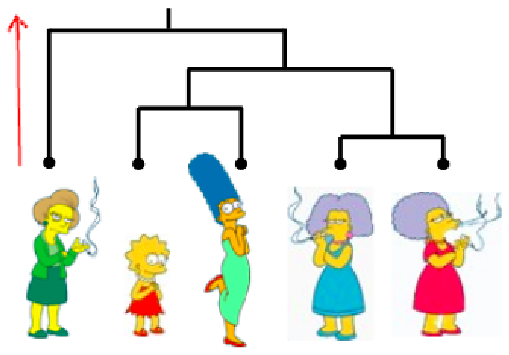


Figure 2: Illustration of a dendrogram.

An example of a dendrogram. The red vertical line represents dissimilarity. The higher up the red arrow the split happens, the more dissimilar the characters are. Patty and Selma are twins, so they are definitely very similar (and thus low on the dissimilarity arrow). Lisa is Marge's daughter, thus they are also similar, but less so than the twins (thus the split happens higher up the dissimilarity arrow). As Marge, Patty and Selma all are sisters, they are similar and gets connected higher up the tree. As Lisa is already connected with Marge, they are both connected with Patty and Selma. Finally Edna is Lisa's teacher, but have no direct connection to any of the other four. Thus they are connected at the very end (at the top of the dissimilarity arrow).

Different linkage methods. Here we use linkage to compute dissimilarity between clusters. We always merge the clusters with the *least* dissimilarity (most similar), but the linkage only decides how this distance is measured.

- *Single*: $D(c_1, C_2) = \text{Min}_{x_i \in c_1, x_j \in c_2} D(x_i, x_j)$ between cluster 1 and cluster 2. This means that we find (out of every point in c_1 and c_2) the minimum distance between two points. Can result in trailing where clusters merge in single observations one at a time.
- *Complete*: $D(c_1, C_2) = \text{Max}_{x_i \in c_1, x_j \in c_2} D(x_i, x_j)$ between cluster 1 and 2. We find the maximum distance between two points in two clusters. Forces spherical clusters with almost constant diameter.
- *Average*: $D(c_1, c_2) = (|c_1||c_2|)^{-1} \sum_{x_i \in c_1} \sum_{x_j \in c_2} D(x_i, x_j)$. between cluster 1 and 2. Here we record the distance between every pair of points between the two clusters and uses the mean of these values. A middle ground between single and complete, less affected by outliers.
- *Centroid*: $D(c_1, c_2) = D(|c_1|^{-1} \sum_{x_i \in c_1} x_i, |c_2|^{-1} \sum_{x_i \in c_2} x_i)$.This is the one used in K-means clustering.

7 Very brief Neural Networks

For neural nets we use *layers.* Each layer has a set amount of *nodes.* The first layer has p nodes, corresponding to one node for each dimension in the data set (input). The last layer has q nodes, one for each class of the response variable (output). What makes neural nets different are the layer(s) in between the input and output layer, these are called hidden or latent layers. For each layer, there is a connection between the layer before and after, where every node in the middle is connected to every node in front and back.

7.1 One hidden layer

For only hidden one layer, we have r nodes connected to all the p input nodes and q output nodes. We first need to connect the input layer with the hidden layer, and then the hidden layer with the output layer. We write the input nodes as x_k, hidden layer nodes as z_l and output layer nodes as y_m. We want every node to contribute a constant value as the function argument. Also setting $x_0 = z_0 = 1$ we allow for constant terms. Thus the node values z_l and y_k can expressed as:

$$z_l = f_0 \left(\sum_{i=0}^{n} \alpha_{il} x_i \right) \qquad y_k = f_1 \left(\sum_{i=0}^{r} \beta_{im} z_i \right)$$

One of the functions f_0, f_1 has to be non linear. If not, the problem would reduce to matrix multiplication. A popular choice is the logistic function.

$$f(u) = \frac{1}{1 + e^{-u}}$$

To estimate the α and β coefficients we can use the usual RSS.

$$D_0 = \sum_{i=1}^{n} ||y_i - f(x_i)||^2$$

Where λ is a tuning parameter. Since all the features are effected by the same tuning parameter, all the features should be scaled before fitting. One can also add a penalty term to avoid potential overfitting problems, as these models are very flexible.

$$D = D_0 + \lambda J(\alpha, \beta)$$

- Pros
  1. As we can choose the linking functions $f(\cdot)$, the different methods are very flexible.
  2. Sequential upgrading, as in α and β can be updated after the training is done
- Cons
  1. Hard to choose r and λ. Sort of arbitrary.
  2. The huge number of α and β values make is hard to find the lowest D value. There can exist a lot of local minimum points, making methods like cross validation hard.
  3. Black box, very difficult to interpret.

8 Potential pitfalls

8.1 Overfitting

Considering the MSE $= \mathbb{E}[\hat{\mathbf{y}} - f(\mathbf{x})]$ where $f(\mathbf{x})$ represents the target (real, but unknown) function. This quantity can be interpreted as.

$$MSE = \mathbb{E}[\hat{\mathbf{y}} - f(\mathbf{x})] = \text{bias}^2 + \text{variance} + \text{irreducible}$$

Type	High	low
Bias	Strict model, a lot of assumptions	Flexible model, few assumptions
Variance	New data preforms bad, overfitting	New data preforms well
Irreducible	Lots of noise from data	Little noise from data

Table 1: Caption

9 Data-set splitting

Fitting all the data to a model is in general a bad approach, as the model can be *overfitted* for flexible models and/or makes testing the accuracy of the model infeasible (the model is bias to the sample).

9.1 Test/train split

This approach simply splits the data set into a train and test set. The exact split is dependent on many factors, but normally something like 2/3 train and 1/3 test is used. The model is fitted on the training data, and selection of which model to use/evaluation of the accuracy is chosen trough predictions using the test set. To make the train/test split as un-biased as possible, random selection is preferred (the data can somehow be structured).

- Pros
    1. Very computationally cheap, pretty much the "simplest" thing to do
    2. Provides more precise results than fitting with all the data despite not being computationally expensive
- Cons
    1. If many models are tested and the test set is used to pick the best model, the model can be biased towards the test set. Sometimes a third split is made (validation set) to reduce this bias.
    2. The number of samples to preform the fit are reduced, can in turn increase error.
    3. Not stable results, can depend on the randomly chosen train/test split.

9.2 Cross Validation

Cross validation (CV) is based on splitting the data into different *folds* (parts) and training the model one some folds, while testing on others, repeated many times to make averages of RSS, MSE, ...

- Pros
    1. More stable than Train/Test
    2. Reflects the population better than train/test
- Cons
    1. More computationally expensive than train/test (tough often doable is if $k \approx 5 - 10$)
    2. Can produce some instability, folds are selected randomly

9.3 Leave-one-out CV (LOOCV)

Pick on data point. Train the model on every other data point, and evaluate on that chosen point. Reapeat for all points

- Pros
    1. Very low bias, model is trained on the whole data-set
    2. Do not overestimate test error
    3. Zero randomness, stable results
- Cons
    1. VERY computationally expensive. Especially for large n and EXTREMELY if model takes a long time to train.

LOOCV is pretty much never feasible to do, but for ordinary LS (No Ridge/Lasso) there is a closed form solution.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{1-h_i}\right)^2 \qquad h_i = \frac{1}{n} + \frac{(x_i - \mathbb{E}[x])^2}{\sum_{i'=1}^{n}(x_{i'} - \mathbb{E}[x])^2}$$

10 Model Selection

All selection models can use different measures to classify the "best" model or "smallest" error. Some examples:

- Complexity-indipendent: MSE, RSS, R^2
- Complexity-dependent: AIC, BIC, adjsted R^2

10.1 Best Subset Selection

With p features try all 2^p model and use the one with the smallest error. Very good but stupidly expensive, especially if model takes a long time to train. Large p makes this infeasible. Often models are trained on the training data, and the $p+1$ lowest RSS/R^2 models are chosen. Then some CV scheme is used to choose which of these models to use with some complexity penalising measure (to reduce overfitting). All in all conceptually appealing but computationally infeasible.

10.2 Stepwise Selection

10.2.1 Forward and backward

Start with constant term. For $k = 0, ..., p-1$ try all $p-k$ models and save the best model for each k. Then select a single model from these using some complexity-dependent measure. Results in a total number of models:

$$1 + \sum_{k=0}^{p-1}(p-k) = 1 + \frac{p(p+1)}{2}$$

More computationally feasible than Best Subset. Works well in practice, but not guaranteed to find the best possible model of the 2^p possibilities. Similarly one can use backward selection. Totally identical but start with p predictors and remove one at a time. Forward selection works for (Assuming LS) $p > n$ as a maximum of n features can be chosen, but backward selection can NOT do this.

11 Special functions and (some) needed statistics

11.1 Some error analysis

There are a few assumptions about the errors that we (almost always) make to preform error analysis.

1. Independent variable(s) measured without uncertainty (the $\mathbf{X}$ data). Almost always false but always assumed.
2. Errors are normally distributed $N \sim (0, \sigma^2)$
3. Constant σ^2
4. Errors are independent, e_i does not effect e_j

Quantifying this error, we define the *residual* of the ith data point $e_i = y_i - \hat{y}_i$. We are mostly interested in the global error, so we define the residual sum of squares (RSS) as:

$$RSS = \sum_{i=1}^{n} e_i^2$$

The *root square error* (RSE). Provides an absolute measure of lack of fit of the model to the data.

$$RSE = \sqrt{\frac{RSS}{n-p-1}}$$

And the *total sum of squares* (TSS)

$$TSS = \sum_{i=0}^{n} (y_i - \bar{y})^2$$

We also define R^2 as a mesure of how good a linear fit our model is. Note that R^2 will **always** increase (or stay the same) when adding more paramters p as we fit the data set better, but this might not reflect the true population and might preform worse on new test data. Adjusted R^2 tries to compensate for this.

$$R^2 = 1 - \frac{RSS}{TSS}$$

An alternative to pure R^2 is the adjusted R^2, that tries to compensate for removing predictors as well as sample size.

$$R^2_{adj} = 1 - \frac{RSS/(n-p-1)}{TSS/(n-1)} \tag{2}$$

When comparing models the response should ideally be on the original scale of the data. You can compeer models with the same target transformation, but not when they differ.

11.2 Likelihood function

Often just called likelihood describes how "good" a fit is as a function of parameters $\boldsymbol{\theta}$. It is for by assuming that both the input variables $\mathbf{X}$ *and* output variables $\mathbf{Y}$ comes from a join probability distribution. It is viewed as a function of $\boldsymbol{\theta}$ only, so the recorded data ($\mathbf{x}$ and $\mathbf{y}$) are viewed as constant. This is often written as

$$L(\mathbf{y}|\boldsymbol{\theta}) = p_\theta(\mathbf{y}) = P_\theta(\mathbf{Y} = \mathbf{y})$$

Meaning the likelihood of the outcome $\mathbf{y}$ (sampled from the random variable $\mathbf{Y}$), as a function of $\boldsymbol{\theta}$. We say that the $\hat{\theta}$ that maximizes L, is the maximum likelihood estimate (MLE).

The data is assumed to be collected from a pdf/pmf q. If we have n data points $x_1, ..., x_n$ these are sampled with a given parameters θ. That is, they are sampled from $q(x|\theta)$.

11.2.1 Indicator Variables

If we have a qualitative (non-continuous, often not numerical) variable we have to apply indicator functions. If we add k indicator functions, we expand our data-matrix $\mathbf{X}$ with k new columns, and $\hat{\beta}$ we k new entries respectively. This is normally represented by the function I_i corresponding to one of the k qualitative variables. If the qualitative variable is "male" or "female", the form of I_M might look like.

$$I_M = \begin{cases} 1 & \text{if "male"} \\ 0 & \text{if "female"} \end{cases}$$

We now have to make model for "female", such that $I_M = 0$ gives expected results. The parameter β_i corresponding to "male" should be the average difference between "male" and "female" in our model, such that in case the data point is "male" our model will compensate for this.

11.2.2 Variable Transformations

The data has to be linear with respect to the parameter, but we can still use non-linear variables as out data. We can apply how as many transformations as we want, i.e. $X = (1, x_1, x_2, x_2/x_1, e^{(x_2^2 - x_1)})$. A common variant is the polynomial model $X = (1, x_1, x_2^2, ..., x_n^p)$. We can also transform the target.