Table des matières

Statistics

Missing Data

Vidéos YouTube

- ritvikmath: Missing Data Mechanisms
- ritvikmath: Dealing With Missing Data Part I
- ritvikmath: Dealing With Missing Data - Multiple Imputation

Notes sur les vidéos YouTube

Video 1: ritvikmath: Missing Data Mechanisms

- MCAR
  - Librarians forget to enter the data completely randomly ;
- MAR
  - Women are 90% likely to respond to a survey on the number of overdue books while men are 70% likely to respond ;
- MNAR : **The missingness of a certain value depends on the true value itself**
  - For example :
    - If I have 0 books overdue, I'm 90% likely to respond to a question asking how many overdue books I have ;
    - If I have 1 books overdue, I'm 80% likely to respond to a question asking how many overdue books I have ;
    - If I have 2 books overdue, I'm 70% likely to respond to a question asking how many overdue books I have ;
    - ...
  - So, the more books I have that are overdue, the less likely I am to respond to a question asking how many books I have that are overdue because I may be embarassed or feel shame ;
  - Therefore MNAR is kind of a **chicken and egg** scenario ;
    - If try to figure if a column is MNAR, need to figure out if those missing values are based on the actual values ;
    - But, I *don't know* the **actual** values of that column because they're missing in the first place !
    - So it's really hard to figure out if something is MNAR ;
  - In comparision, MCAR and MAR are easier to figure out if something is one or the other ;
  - Can slice a dataset by values of a column

* For example, by sex or by age group, . . . ;
* If missing value rate is about the same for all different slices then likely to be Missing Completely At Random ;
* If, however, it's different for each slice, then it's likely MAR ;

Video 2: ritvikmath: Dealing With Missing Data Part I

› Row deletion ;
- Most common and easiest ;
- Omit any row in dataset with a missing value—pretend it does not exist ;
- Seems too good to be true because it usually is ;
- Can only do this if the data is Missing Completely At Random—biased otherwise ;

  Makes sense if you reason that any other way you'd obviously be creating a bias in your data ;

  Each column would have missing values completely at random and without respect to, for example, the gender and the estimations would be *unbiased* ;
- Thus, have to be very careful it's really what we want to do because likely cause bias if there's any sort of relationship between the missing variables and other columns ;

Pros	Cons
simple	**biased**

› Mean/Median imputation ;
- A little more « clever » ;
- **Seems** intuitive and is pretty simple ;
- Mean
  * Fill in, for example, 1.8 as the average of a few discrete values ;
  * BUT, will **artificially** *reduce variability* of the data ;
  * *Seem* like several values have the exact same value ;
- Median is the same idea but will overepresent one fixed value ;

Pros	Cons
simple	**lower variability**

› Hot Deck methods ;
  – Most clever so far ;
  – Any family of methods where :
    * Compute a missing value based on the value of examples that are *similar* to it ;
  – For example
    * Fill in the missing value of a female by the average of only other females ;
  – Better because imputing more information (whether someone is female or male) ;
    * Imagine if there's a bunch of columns (income, family members, where they live, etc.) ;
    * Then, we can input missing values based on a few people that are really similar to the missing person ;
    * Logical as we would expect that person's missing value to be similar to other similar people's ;
  – **May not be true**, but it's a very ***educated*** *guess* ;

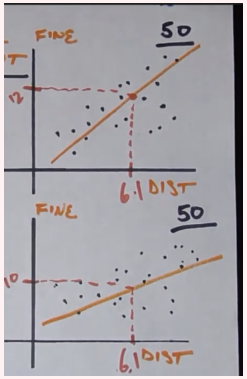
Pros	Cons
more educated	more (computationally) expensive

Video 3: ritvikmath: Dealing With Missing Data - Multiple Imputation

› *single* imputation implique qu'on se ramasse avec une seule valeur (peu importe ce que c'est) ;
  – Régression, moyenne, médiane, etc. sont tous une seule valeur.
› *multiple* imputation even more clever than hot-deck methods ;
› For example, regression of library fees in function of kilometer distance from the library
  – Sample 50 data points from thousands, and estimate fees for a given distance ;
  – Repeat with different samples ;
    Generally the more repetitions, the less biased the estimations but 5 is a good rule of thumb ;
  – Treat each predicted value as a complete observation ;
  – Then with this "complete" data set, we do what we want ;
  – Take some kind of aggregate of all the values we wanted from each of 5-ish set ;
  – Analyze how far from each other the aggregated values are ;

If a lot of variability, bad ;

If few variability, good-ish because means aggregated values are closer.



› Cons : complicated.

Regularisation

Vidéos YouTube

› Regularization Part 3: Elastic Net Regression

Video 4: Regularization Part 3: Elastic Net Regression

› Lasso est optimal lorsque le modèle contient beaucoup de variables *inutiles* puisqu'ils les éliminent. Il permet de créer un modèle plus simple qui sera plus facilement interprétable.

› En contraste, la régression Ridge est optimale lorsque le modèle contient beaucoup de variables qui sont *majoritairement utiles*. Il permet de réduire l'importance des variables en diminuant leurs coefficients, mais n'ira pas si loin que les éliminer.

› Mais quoi faire lorsqu'il y a encore plus de variables ? ?

› For example, with big data there are millions of variables—far too many to know everything about them.

› So, when there are many variables you almost certainly need some sort of regularization to estimate parameters.

› In this scenario, you don't have to choose between Ridge and Lasso instead you just use **Elastic-Net Regression**.

- Start with least squares (like Lasso and Ridge).
- Combine the Lasso λ_1 and Ridge λ_2 penalties.

$$SS(\text{residuals}) + \lambda_1 |\text{parameter}_1| + \cdots + |\text{parameter}_X| + \lambda_2 \text{parameter}_1^2 + \cdots + \text{parameter}_Y^2$$

› The regression penalty coefficients λ_1 and λ_2, use CV.

Lasso parameter	Ridge parameter	Regression
$\lambda_1 = 0$	$\lambda_2 = 0$	Ordinary Least Squares
$\lambda_1 > 0$	$\lambda_2 = 0$	Lasso
$\lambda_1 = 0$	$\lambda_2 > 0$	Ridge
$\lambda_1 > 0$	$\lambda_2 > 0$	Elastic Net

› Elastic Net is particularly good at dealing with situations where there are correlations between parameters.

- Lasso tends to pick just one of the correlated terms and eliminate the others.
- Ridge tends to shrink all of the correlated variables parameters together.
- Elastic Net groups and shrinks the correlated variables' parameters and either leaves them or removes all of them.

Trees

Vidéos YouTube

- StatQuest: Random Forests Part 1 - Building, Using and Evaluating
- StatQuest: Random Forests Part 2: Missing data and clustering
- StatQuest: Random Forests in R

Video 5: StatQuest: Random Forests Part 1 - Building, Using and Evaluating

- The only caveat of CARTs are that they are very inaccurate for new data.
- Thus, random forests permit us to combine the simplicity of decision trees with flexibility for much better accuracy.
- To create a random forest :
  - Create a bootstrapped dataset.
  - Create a decision tree with the bootstrapped dataset only considering a random subset of the variables at each step.
  - Repeat with a new bootstrapped dataset (usually hundreds of times).
  - To get a prediction, we run our observation through the decision trees and the result with the most votes is the prediction :

- This provides a much wider variety of decision trees which is why random forests are much more flexible.

  Note : **B**ootstrapping the data and using the **agg**regate to make a decision is called **bagg**ing.
  - Typically about 1/3 of the original data does not end up in the bootstrapped dataset—this is the **Out-Of-Bag** dataset.
  - A better terminology, however, would be the **Out-Of-Boot** dataset as it's the data which wasn't in the bootstrapped dataset.
  - It becomes our testing dataset with which we calculate the **Out-Of-Bag Error**.
- We do this with a different number of variables being used per step to find the optimal number of variables to use.
- Typically, we start with the square of the number of variables and try a few settings above and below that value.

Video 6: StatQuest: Random Forests Part 2: Missing data and clustering

› Random forests consider 2 types of missing data :

1. Missing data in the original data set used to create the RF.
   - The idea is to make an initial guess that could be bad, then gradually refine the guess until it is (hopefully) a good guess.
   - The initial bad guess is to input the most common value for a categorical variable, and the median for a numeric one.
   - To refine the guess, we find observations which are *similar* to the one with missing data.
   - In the case of a RF, samples which end up in the same terminal node are *similar*.
   - We can keep track of similar samples with a proximity matrix.
   - After each iteration of the RF, we add one to the samples which end up together and then divide the total by the number of trees.

	1	2	3	4
1		0.2	0.1	0.1
2	0.2		0.1	0.1
3	0.1	0.1		0.8
4	0.1	0.1	0.8	

   - Therefore we can make better guesses about the missing data using proximity values to weigh the frequency of different values.
   - weight for the sample 4 yes $= \frac{\text{proximity where yes}}{all proximities for sample 4}$.
   - Can find the distance matrix which is $1 - proximity$ and them do heat maps or MDS / PCoA.
2. Missing data in a new sample we want to categorize.
   - Create two copies of the observation—Yes and No.
   - Then, use the same method to input the missing data.
   - Then, take the option which classifies the best as either Yes or No respectively.

Video 7: StatQuest: Random Forests in R

› package `randomForest`.

› Default number of splits is the square root of the number of variables.

- Plot the error rates to determine the number of trees for optimal classification.

Boosting

Vidéos YouTube

- AdaBoost, Clearly Explained
- Gradient Boost Part 1: Regression Main Ideas
- Gradient Descent, Step-by-Step
- Stochastic Gradient Descent, Clearly Explained!!!
- Gradient Boost Part 2: Regression Details
- Gradient Boost Part 3: Classification
- Gradient Boost Part 4: Classification Details

Video 8: AdaBoost, Clearly Explained

Differences between Random Forests and Adaboost

Random Forest

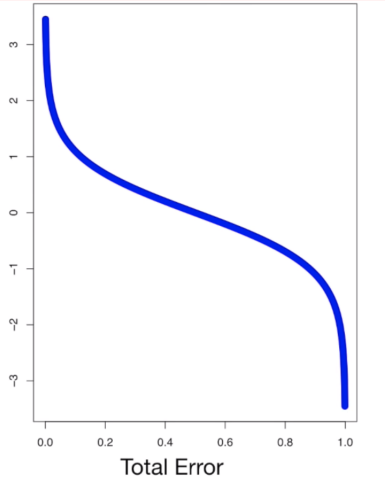
- Each time you make a tree, you make a full sized tree.
  - Some trees may be bigger than others, but there's **no predetermined maximum depth**.
- Each tree has an equal vote on the final classification.
- Each tree is made independently of the others.

Adaboost Random Forest

- Trees are usually *just one node* and *two leaves*—a **stump**.
  - So Adaboost is really a random "*Forest of Stumps*".
  - Stumps are not great at making classifications as they only have one variable to make a decision—they are "*weak learners*".
- Some stumps get more say than others in the final classification.
- Trees are mare sequentially thus order has an impact.
  - The first stump's errors influences how the second stump is made.

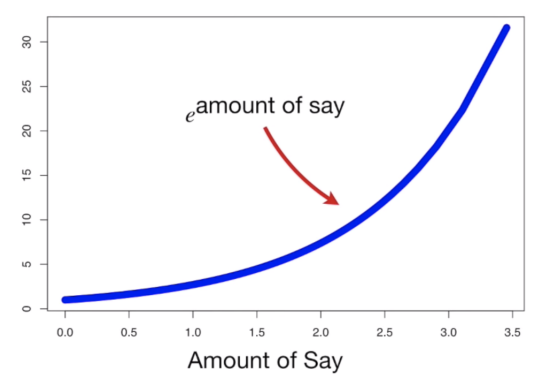
Procedure :

1. We give each observation, or *sample*, a **"sample" weight** to indicate how important it is that it be correctly classified.
   - Initially, all samples get the same weight $\frac{1}{\text{total number of samples}}$.
   - However, after making the first stump, these weights will change in order to guide how the next stump is created.
2. To make the first stump, we find the variable which does the best job classifying the sample.
   - Given all the weights are the same right now, we can ignore them.
   - We could use the Gini index to compare the variables.
3. We determine how much say the first stump will have in the final classification.
   - The **total error** for a stump is the sum of the "sample" weights associated with the incorrectly classified samples.
     - For example, incorrectly classifying 2 samples with sample weights of 1/8 leads to a total error of 2/8.
     - Given the sample weights add up to 1, the total error will always be between 0 (a perfect stump) and 1 (a terrible stump).
   - The **amount of say** $= \frac{1}{2}\log\left(\frac{1-\text{Total Error}}{\text{Total Error}}\right)$.
     Its graph looks like this :



     - When a stump is efficient and has a relatively small total error, its amount of say will be relatively large.
     - If a stump is really bad, the negative value will turn a "Yes" into a "No".
   - In brief, the sample weights from ***incorrectly*** classified samples determine the *amount of say* each stump gets.

4. Modify the weights so that the next stump will account for the previous stump's errors.
   - › Originally all the sample weights were identical.
     Therefore, there was no emphasis on the *importance of correctly classifying* any one particular sample.
     However, we really want the samples (observations) which were incorrectly classified before to be correctly classified in the next stump.
     So, we give the incorrectly classified samples more weight and reduce the weight of the samples that were correctly classified.
   - › When we want to ***increase*** the weights, the **new sample weight** = sample weight $\times$ $e^{\text{amount of say}}$.
     Its graph looks like this :



     - – When a stump is efficient and has a relatively high amount of say, then the new sample weight will be much larger than the previous.
     - – If a stump is really bad, then the new sample weight will not change very much.
   - › When we want to ***decrease*** the weights, the **new sample weight** = sample weight $\times$ $e^{-\text{amount of say}}$.
   - › We then normalize them so they're between 0 and 1 with $\frac{\text{sample weight}}{\text{sum of the new sample weights}}$.
   - › Theoretically, these weights could be used to calculate the weighted Gini index.
   - › Instead, we make a new sample of the same size picking observations according to their weight using the inversion method.
     - – Thus, some observations could (should) be repeated in the new sample set.
     - – Then, we give equal weights to this new sample.
     - – Because some observations will be repeated, they will have a higher weight as the will be treated like a "block" with a large penalty for misclassification.
5. Predict
   - › We sum the amount of say from each root tree and the class having the highest weight is the predicted value.

Three main ideas :

1. **Adaboost** combines a lot of *weak learners* to make classifications.
   - The weak learners are almost always **stumps**.
2. Some **stumps** get more say in the classification than others.
3. Each **stump** is made by taking the previous **stump's** mistakes into account.
   - If we have a *weighted Gini function*, then we use it with the sample weights.
   - Otherwise, we use the sample weights to make a new dataset reflect them (generate a number between 0 and 1, choose observation according to the cumulative proportion).

Video 9: Gradient Boost Part 1: Regression Main Ideas

Comparison / general details Gradient boost

1. Make a single leaf, instead of a tree or a stump.
   - The leaf represents an initial guess for the weights of all the samples.
   - When trying to predict a continuous value, the first guess is the average value.
2. Build a tree
   - Like Adaboost, usually based on the previous tree's errors.
   - Unlike Adaboost, the tree is usually larger than a stump.
   - Gradient boost does however restrict the size of the tree.
     - Often the maximum number of leaves is between 8 and 32.
   - It scales all trees by the same amount.

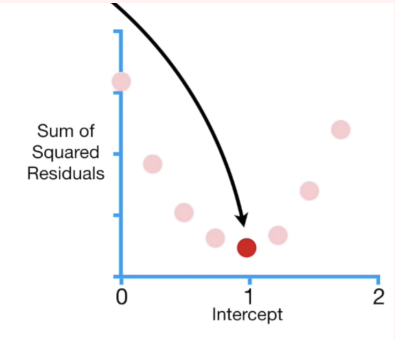
Procedure

1. Calculate the average weight = 71.2kg.
   - First "attempt" at predicting everyone's weight ; i.e., if we stopped right now, we'd predict that everyone weighs 71.2 kg.
2. Build a tree based on the errors the previous tree made.
   - The error, or **pseudo-residual**, is the difference between the observed x and predicted $\bar{x}$ weights $x - \bar{x}$.
     - "Residual" is based on linear regression's residuals between the curve and real data points.

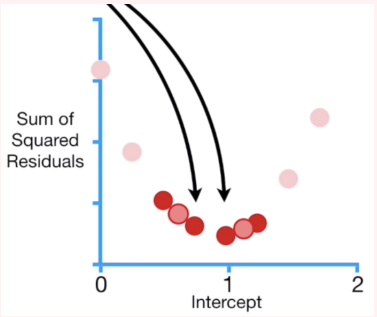
3. We use the variables to **predict the residuals**.
   - The tree will predict the average of the residuals from the observations in the leaf.
     - So, with an initial prediction of 71.2 ($\bar{x}$), the predicted weight is (for example) $71.2 + 16.8 = 88$.
     - In this case, the prediction is bad because it is overfit—low bias and high variance.
   - We address this with a learning rate λ which scales the value.
     - For example, the predicted weight would be $71.2 + 0.10 \times 16.8 = 72.9$.
     - The prediction is a **little worse** than before but a **little better** than the initial guess of 71.2.
   - So, we take **many little steps** in the right direction for better predictions and see this by having slightly smaller residuals.
4. We repeat with the new tree.
   - For example, we'd predict $71.2 + 0.1 \times 16.8 + 0.1 \times 15.1 = 74.4$.

Video 10: Gradient Descent, Step-by-Step

- Examples of times we optimize :
  - Linear regression : we optimize the intercept and slope when fitting a line.
  - Logistics regression : we optimize a squiggle.
- **Gradient descent** can optimize *all* of this and more.
- To optimize the intercept when fitting a line with gradient descent :
  1. Pick a random value for the intercept.
     - This initial guess gives gradient descent something to improve upon.
  2. Evaluate how well the line fits the data with the SSR.
     - In machine learning, we refer to the SSR as a type of **Loss Function**.
  3. Repeat this with several possible values for the intercept and plot the SSR in function of the intercept :



- The long way to then find the optimal intercept and find the best value would be to test the values between the lowest points to see if another number is better.
- Gradient descent does a few calculations far from the optimal solution and increases the number as it gets closer to the optimal one :



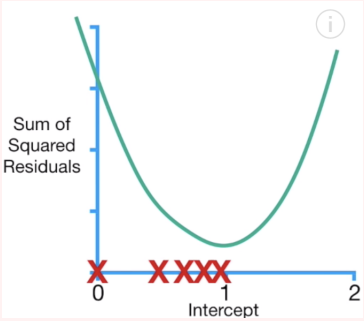
i.e., it takes big steps when far away and baby steps when it is close.

› To optimize the intercept, we develop SSR as a function of the intercept :

$$\begin{aligned}\text{Sum of squared residuals} = (\mathbf{1.4} - (\text{intercept} + 0.64 \times \mathbf{0.5}))^2 \\ + (\mathbf{1.9} - (\text{intercept} + 0.64 \times \mathbf{2.3}))^2 \\ + (\mathbf{3.2} - (\text{intercept} + 0.64 \times \mathbf{2.9}))^2\end{aligned}$$

› Then, we take it's derivative so that **Gradient Descent** can use it to find the point where the slope is 0.

› In linear regression, find where the slope is 0 but gradient descent tries values sequentially :



- Step size = slope × learning rate and new intercept = old intercept - step size.
- It stops taking steps when the step size is very close to 0, in practice it is often 0.001 or less, or when it reaches a maximum number of steps.
- To optimize find the slope **AND** intercept, we take the derivative with respect to both.
- When you have multiple derivatives of the same function, they're referred to as the "**gradient**" of a function.
- We use the **gradient** to **descend** to the lowest point of the **loss function**.
- Gradient descent is very sensitive to the learning rate—in practice, we start large and try lower and lower values until we converge.

In brief

1. Take the derivative of the **Loss Function** for each parameter in it.
2. Pick random values for the parameters.
3. Plug the parameter values into the derivatives (the gradient).
4. Calculate the step sizes : **step size** = **slope** × **learning rate**.
5. Calculate the new parameters : **new parameter** = **old parameter** - **step size**
6. Repeat from step 3 until the step size is very small, or you reach the maximum number of steps.

With many data points, it can be very long to calculate the SSR at each step. **Stochastic Gradient Descent** uses a random subset of the data at every step instead of the full dataset.

Video 11: Stochastic Gradient Descent, Clearly Explained!!!

- The main advantage of stochastic gradient descent is to apply gradient descent to a large amount of data.
- For example, with only 10'000 data points, we'd already have to calculate 20'000 residuals *at every step*.
- Stochastic Gradient Descent is also very sensitive to the learning rate and we also start large and get progressively smaller. The way the rate changes is called the **schedule**.
- It is especially useful when there are redundancies in the data.
- The **strict** definition is to use 1 observation per step.
  However, it is more common to use a small subset, or **mini-batch**, of data for each step.
- It can lead to more stable estimates of the parameters in fewer steps.
- An advantage of Stochastic Gradient Descent is we can use it to adjust our guess instead of restarting the whole process.

Video 12: Gradient Boost Part 2: Regression Details

The formula looks complicated because it's designed to be applicable in many situations.

-

Video 13: Gradient Boost Part 3: Classification

-

Video 14: Gradient Boost Part 4: Classification Details

-