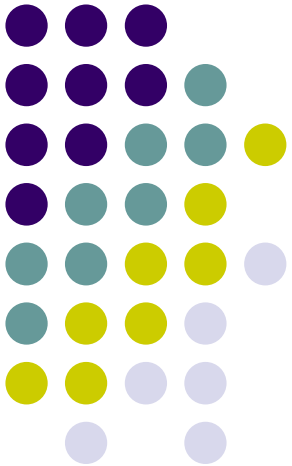
Applied Scaling & Classification Techniques in Political Science

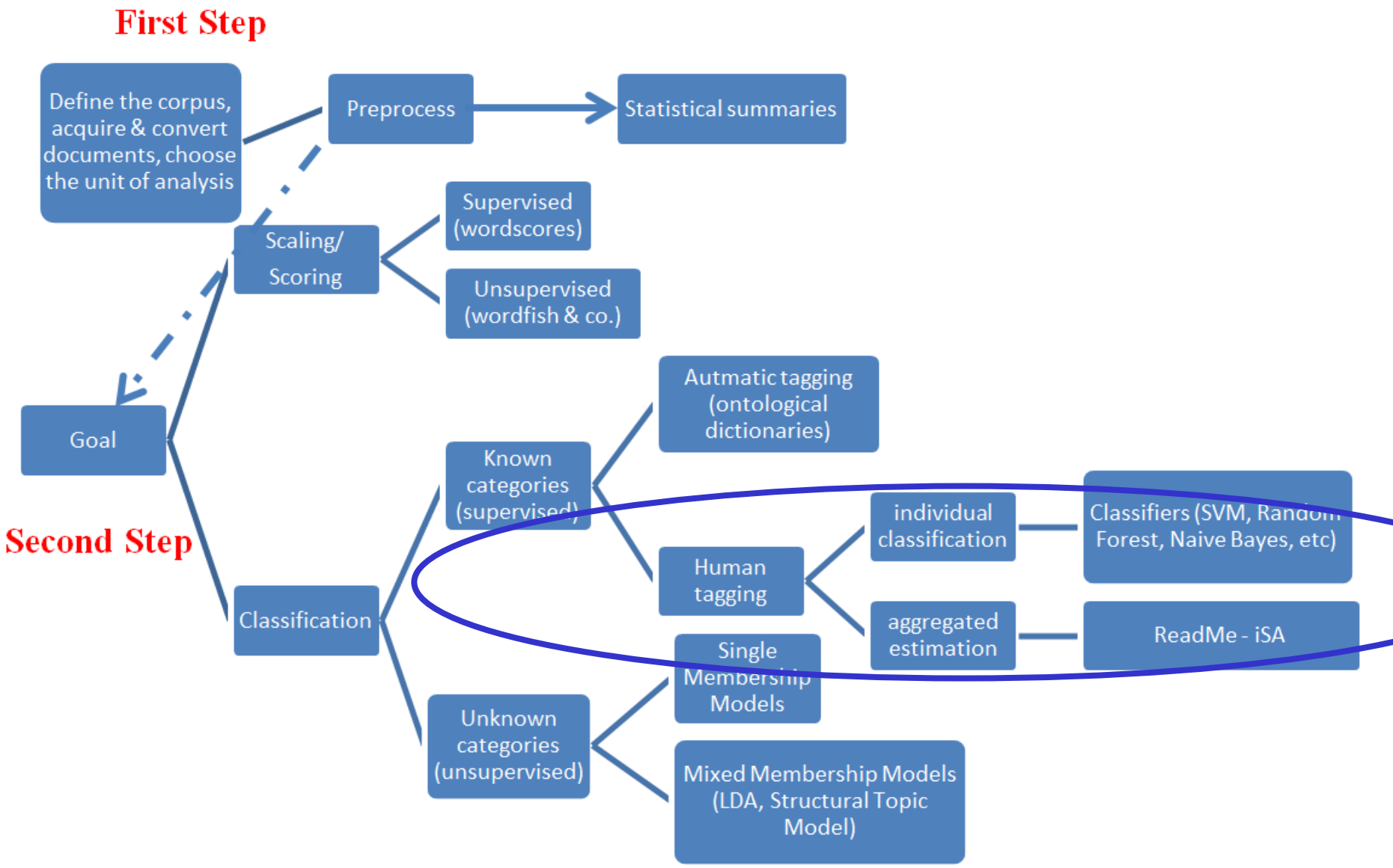
Lecture 7 – part 2

Supervised classification methods:

A review of (two) ML Algorithms



Our Course Map



References

- ✓ Olivella, Santiago, and Shoub Kelsey (2020). Machine Learning in Political Science: Supervised Learning Models. In Luigi Curini and Robert Franzese (eds.), *SAGE Handbook of Research Methods is Political Science & International Relations*, London, Sage, chapter 56

ML alghoritms

Several different possible machine learning algorithms are available out there

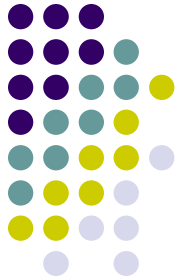
We will offer an intuitive introduction to the following two algorithms (pretty standard in text analytics):

- Naïve Bayes classifier
- Random Forest

Other possible ML algorithms out there: Supper Vector Machine; Regularized regression; Gradient Boosting; Neural Networks (among the others…)

Unfortunately we have no time to discuss about. However, the general logic remains the same…

Naïve Bayes classifier



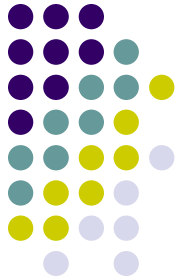
Naïve Bayes classifier: the algorithm allows us to predict a class, given a set of features using (Bayes) probability theorem

But what do we mean by Bayes probability theorem?

Bayesian probability incorporates the concept of *conditional probability*, the probability of event *A* given that event *B* has occurred, i.e., p(A|B)

Within a text-analytics framework, the goal (as already discussed!) is to infer the probability that document *i* belongs to category *k* given word profile $\mathbf{W}_i$ (i.e., the probability of a text belonging to category *k* given that its predictors – features – values are $x_1,x_2,\ldots,x_p$. This can be written as $p(C_k|x_1, x_2,\ldots,x_p)$)

Naïve Bayes classifier



More formally, the Bayesian formula for calculating this probability is:

$$\mathrm{p}(C_k|\boldsymbol{W}_i) = \frac{p(C_k)*p(\boldsymbol{W}_i|C_k)}{p(\boldsymbol{W}_i)}$$

In plain English:

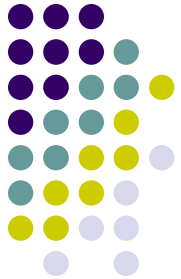
$p(\boldsymbol{W}_i|C_k)$= conditional probability or likelihood

$p(C_k)$=prior probability of the outcome (i.e., the average probabily of that outcome in the training-set)

$p(C_k|\boldsymbol{W}_i)$=posterior probability: by combining our observed information, we are updating our *a priori* information on probabilities to compute a posterior probability that an observation has class C_k

$p(\boldsymbol{W}_i)$=evidence (the word profile we observe)

Naïve Bayes classifier



In other words, the Bayesian formula is simply:

$$\text{Posterior} = \frac{prior * likelihood}{evidence}$$

From above, we can drop $p(\boldsymbol{W}_i)$ - i.e., the evidence - from the denominator since it is a constant across the different categories for each document

Naïve Bayes classifier

An example: let's say we have a training-set on 1000 pieces of fruit. The fruit being a Banana, Orange or some Other fruit. We know **3 variables of each fruit**, whether it's long or not, sweet or not and yellow or not:

Fruit	Long	Sweet	Yellow	Total
Banana	400	350	450	500
Orange	0	150	300	300
Other	100	150	50	200
Total	**500**	**650**	**800**	**1000**

Naïve Bayes classifier

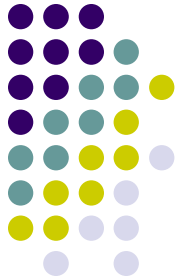
Fruit	Long	Sweet	Yellow	Total
Banana	400	350	450	500
Orange	0	150	300	300
Other	100	150	50	200
Total	**500**	**650**	**800**	**1000**

From the table we know that, in our training-set: 50% of the fruits are bananas; 30% are oranges; 20% are other fruits (our *priors*!)

Based on our table, we can *also* say the following:

Out of 500 bananas 400 (0.8) are Long, 350 (0.7) are Sweet and 450 (0.9) are Yellow; Out of 300 oranges 0 are Long (0.0), 150 (0.5) are Sweet and 300 (1.0) are Yellow; From the remaining 200 fruits, 100 (0.5) are Long, 150 (0.75) are Sweet and 50 (0.25) are Yellow. All these values refer to c*onditional probabilities* or *likelihood*!

Naïve Bayes classifier

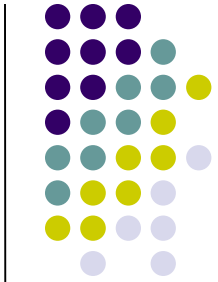


Now let's say we're given the variables of a piece of fruit and we need to predict the class (*out-of-sample prediction*!)

If we're told that the **additional fruit is Long, Sweet and Yellow**, we can classify it using the Bayes formula and subbing in the values for each outcome, whether it's a Banana, an Orange or Other Fruit

The one with the highest probability (score) being the **winner class**!

Naïve Bayes classifier



p(Banana|Long, Sweet, Yellow)=p(Banana=0.5)*p(Long|Banana=0.8)*p(Sweet|Banana=0.7)*p(Yellow|Banana=0.9)=**0.252**

p(Orange|Long, Sweet, Yellow)=p(Orange=0.3)*p(Long|Orange=0)*p(Sweet|Orange=0.5)*p(Yellow|Orange=1)=**0**

p(Other|Long, Sweet, Yellow)=p(Other=0.2)*p(Long|Other=0.5)*p(Sweet|Other=0.75)*p(Yellow|Other=0.25)=**0.01875**

In this case, based on the **highest score**, we can classify this Long, Sweet and Yellow fruit as a Banana

Naïve Bayes classifier

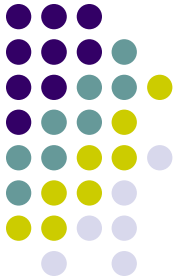
In the case of text-classification, instead of *Banana, Orange and Other* you will have some specific categories (say, *Positive, Negative and Neutral* if we are applying a sentiment analysis) and instead of *Long, Sweet and Yellow* we will have the features (aka: words) included in our DfM

The logic however followed in predicting the category of each text included in the test-set remains exactly the same as the one we just discussed!

Given the features included in that (unread) text, we will estimate its posterior probability to belong to each given category…

…then we will assign that (unread) text to the category with the largest posterior probability among the one we estimated!

Naïve Bayes classifier

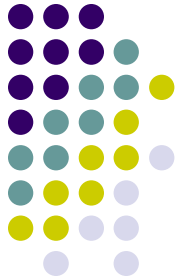


Note a possible problem of the logic just explained: given that naïve Bayes uses the product of variable probabilities conditioned on each class, we run into a serious problem when new data includes a variable value that **never occurs** for one or more levels of a response class

This is what happens with $p(Long|Orange)$ =0 . This “0” will ripple through the entire multiplication of all variable and will always force the posterior probability to be zero for that class

This is clear a HUGE PROBLEM when dealing with a sparse DfM (as it is usually the case)!

Naïve Bayes classifier

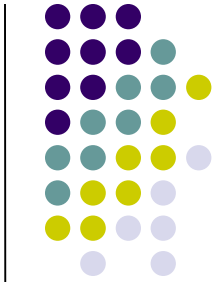


A solution to this problem involves using the ***Laplace smoother***

The Laplace smoother adds a small number to each of the counts in the frequencies for each feature, which ensures that each feature has a nonzero probability of occurring for each class

Typically, a value of 1 for the Laplace smoother is employed, but this is a *tuning parameter* to incorporate and optimize with cross validation (more on this later on!)

Naïve Bayes classifier



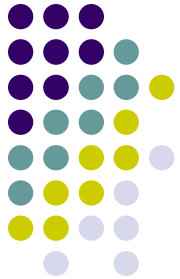
Why Naïve?

Cause it assumes that every feature being classified is **independent** of the value of any other variable given the response variable

In the previous example each of the three variables (Long, Sweet, Yellow) are considered to *contribute independently* to the probability that the fruit is a Banana, *regardless of any correlations* between features

By making this assumption we can simplify our calculation such that the posterior probability is simply the product of the probability distribution for each individual variable conditioned on the response category

Naïve Bayes classifier

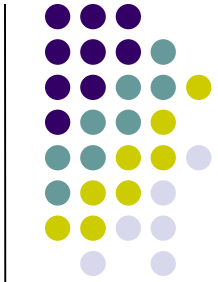


Variables, however, aren't always independent!

Although the model is *clearly wrong* – quite often features are not conditionally independent - it has proven to be a useful classifier for a diverse set of tasks

The same is true for text analysis: assuming that features (i.e., words) are generated indipendently is wrong given that the use of words is highly correlated in any data set. However, Naïve Bayes classifier can still be useful (remember the First Principle of text-analysis!)

Naïve Bayes classifier



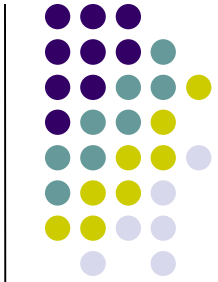
Naïve Bayes classifier algorithm has a simple, but powerful (and fast!), approach to learning the relationship between words and categories

Moreover, it has been shown to perform surprisingly well with very small amounts of training data that most other classifiers, would find significantly insufficient

As a result, if you find yourself with a small amount of training data Naïve Bayes would be a good bet!

Likewise, Naïve Bayes' simplicity prevents it from fitting its training data too closely and therefore does not tend toward **overfitting** especially on smaller datasets like other approaches do

Naïve Bayes classifier



However, Naïve Bayes classifier also behaves differently on the other end of the spectrum, when provided with large amounts of training data

As it is fed increasing quantities of training data, the performance of the Naïve Bayes classifier plateaus above a certain threshold

Its simplicity prevents it from benefiting incrementally from training data past a certain point

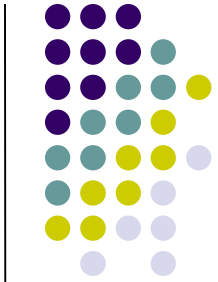
Random Forest classifier

Jordan Goldmeier
@Option_Explicit

Where do data scientists go camping?

In random forests

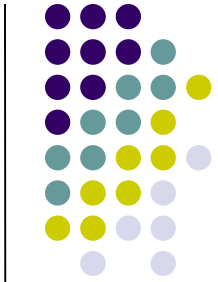
Random Forest classifier



To understand random forest, let's start with what we mean by a *Decision tree model*

A **decision tree** is basically a **set of rules** used to classify data into categories. In particular it is the set of rules **which best partitions the data**

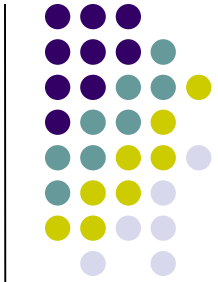
Random Forest classifier



More in details…

…For a single-tree model, the goal is to partition the space of predictor features (i.e. the set of all unique combinations of predictor values) into *B* non-overlapping and exhaustive regions, R_1, R_2, . . . , R_B, that are **relatively homogeneous** with respect to the outcome *y*, thus improving overall predictive accuracy by sorting observations into their respective bins

Random Forest classifier



An example: Given only the gender and weight of a person, can we predict whether they are Japanese or American (our 2 classes/categories)?

Let's train a decision-tree algorithm by using the following training set:

Weight (lbs.)/Sex/Nationality

195 M American
190 M American
160 F American
165 F American
165 M Japanese
160 M Japanese
130 F Japanese
140 F Japanese

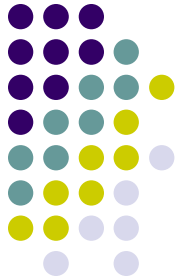
Random Forest classifier

Key idea: the procedure to create decision trees is **recursive**. For a set (**S**) of observations, the following algorithm is applied:

Step 1: If every observation in **S** is the **same class** or if **S** is very small, the tree becomes an **endpoint**, labeled with the *most frequent class*

Clearly the initial group with all our 8 observations does not satisfy *Step 1*! Therefore, we need to move to *Step 2*

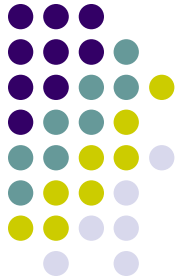
Random Forest classifier



Step 2: If **S** is too large and it contains more than one class (as in our case!), find the **best rule** based **on one feature** to split it into subsets, one for each class (different rules can be used in this respect

- ✓ The aim is always the same: the "best" branching rule is the one that results in the **most information gain**)

Random Forest classifier



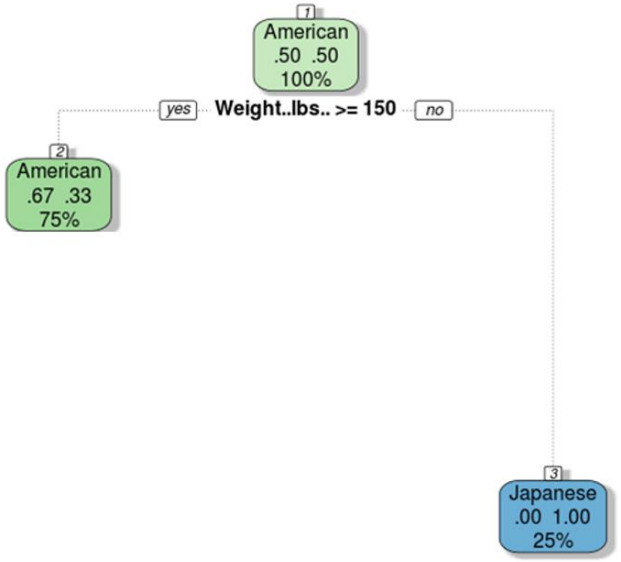
So given our initial **S**, on which feature should we focus?

- ✓ Clearly not the *gender* feature! If we divide our initial S according to gender (Male/Female) we would in fact remain with two sub-groups that will have the same problems of the initial S (50% of the two groups). No information gain at all!
- ✓ Therefore, let's focus on the *weight* feature. But weight is a continuous feature! Therefore which rule should we apply with respect to *weight*? For example we could apply the following one: *if weight is larger or lower than 150*

Random Forest classifier

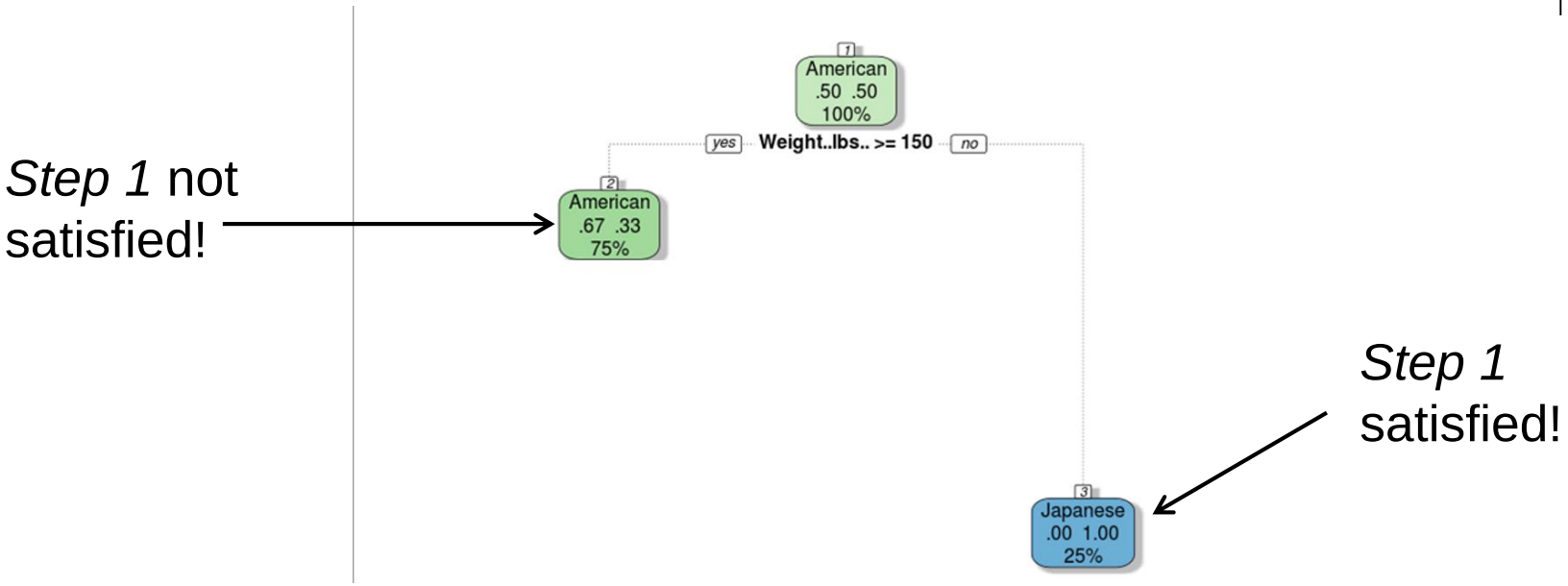
The decision tree: first feature selection rule

What's the result of this first rule?



Random Forest classifier

If you had to go to *step 2*, apply *step 1* to each new subset

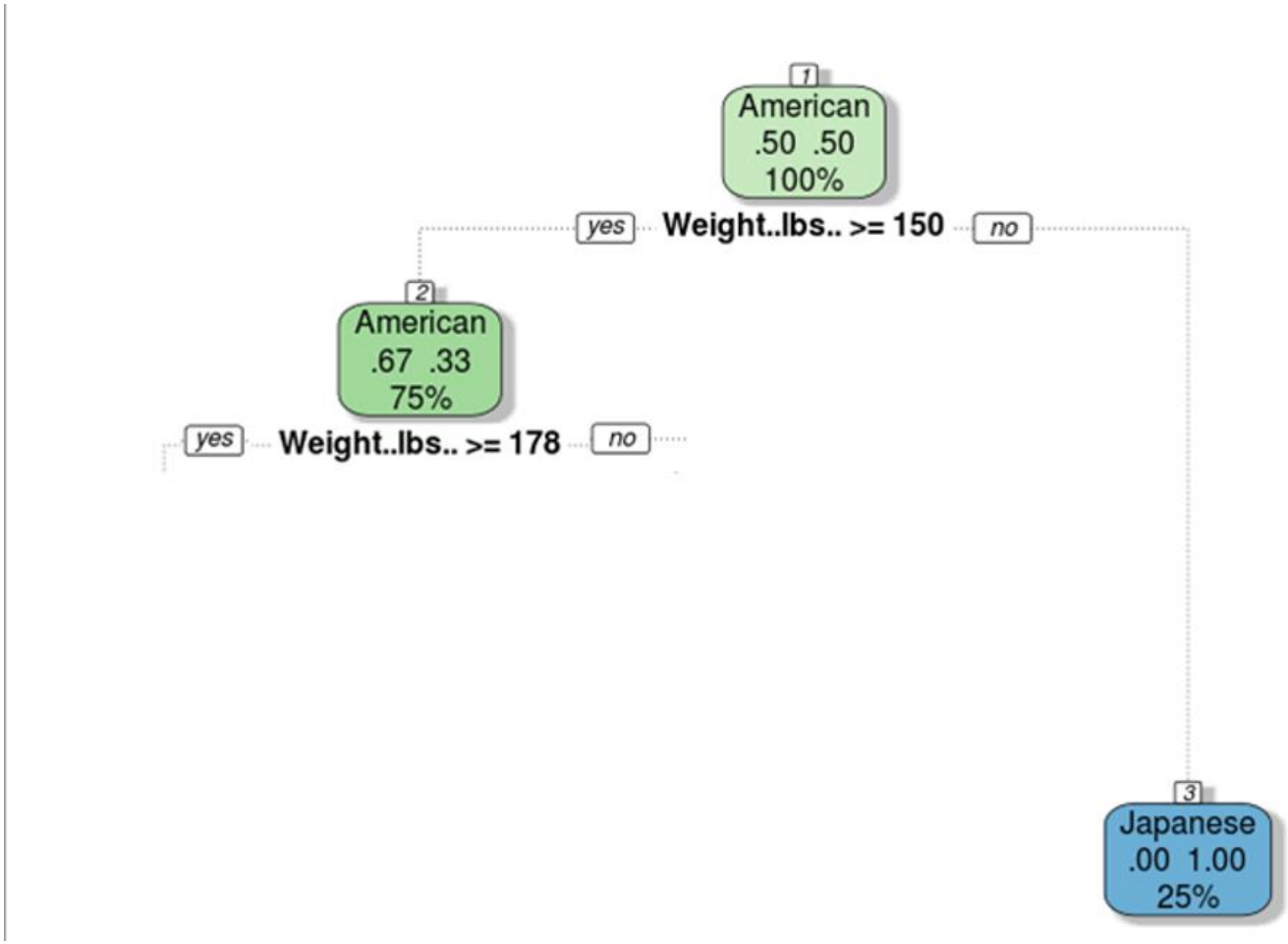


If your subsets need to go to *step 2*, apply *step 1* to the sub-subsets, etc.

When everything is split up appropriately (into buckets that are very small or entirely one class), you have a set of rules that look like a tree!

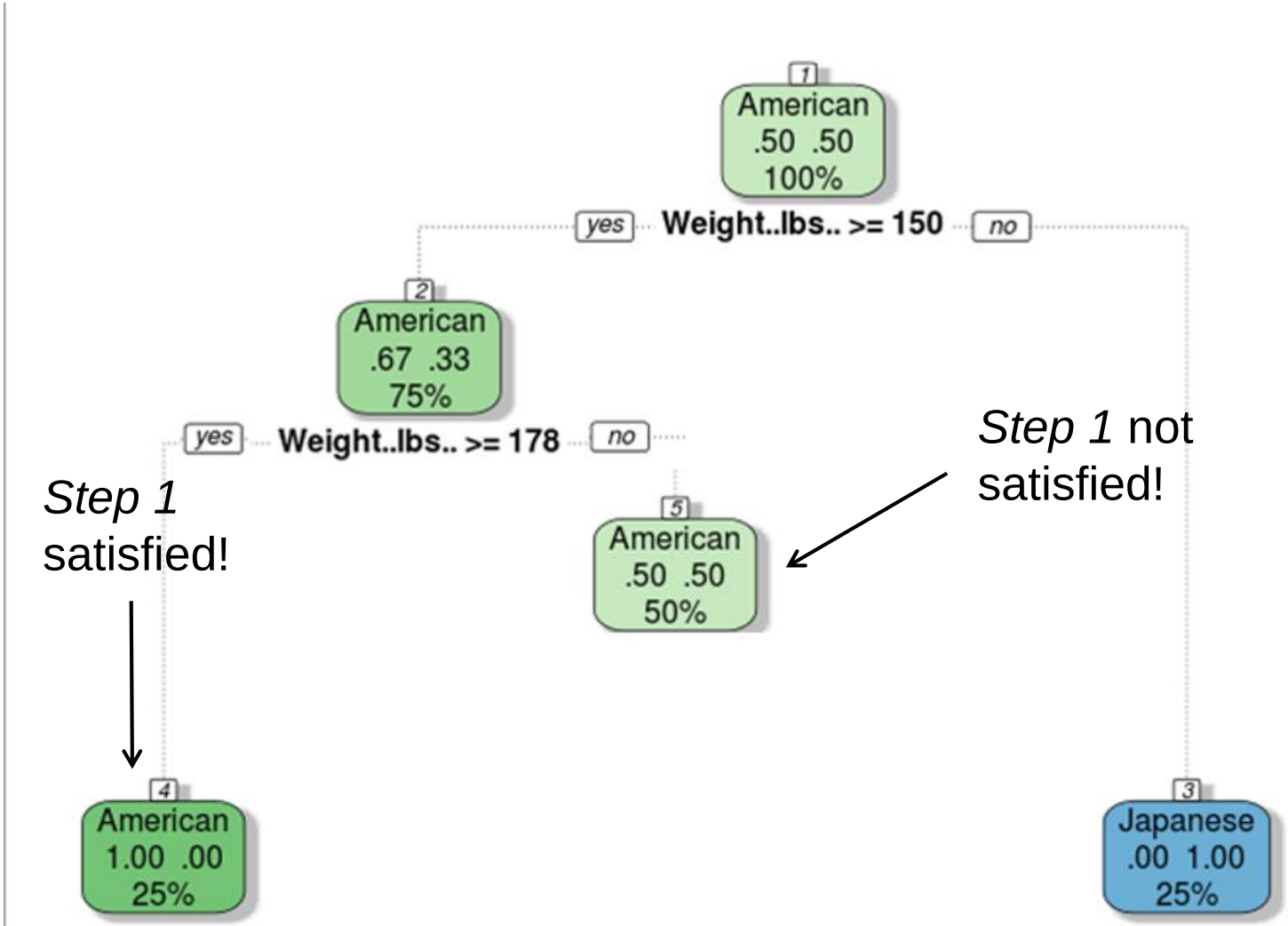
Random Forest classifier

The decision tree: second feature selection rule



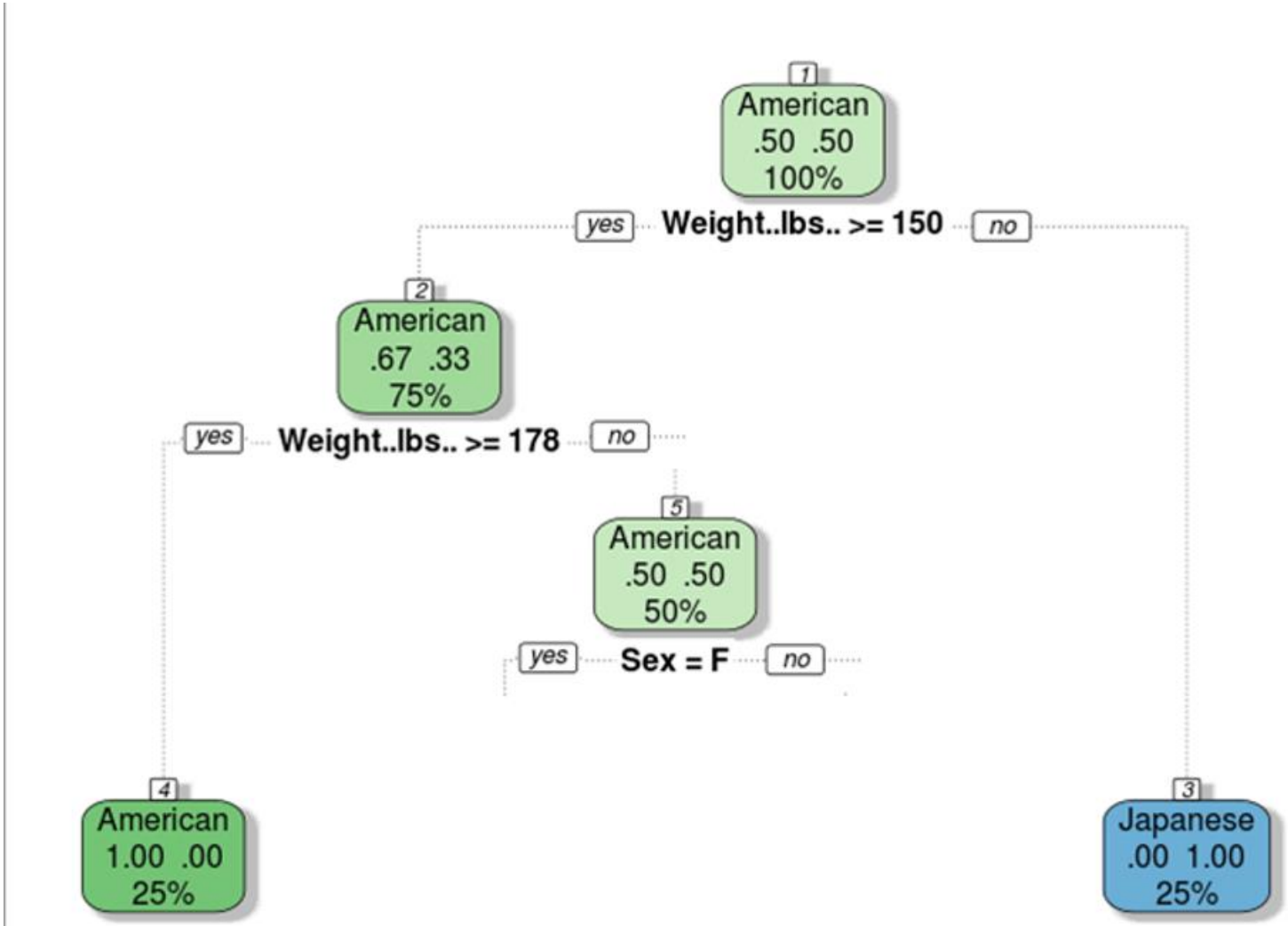
Random Forest classifier

The decision tree: second feature selection rule



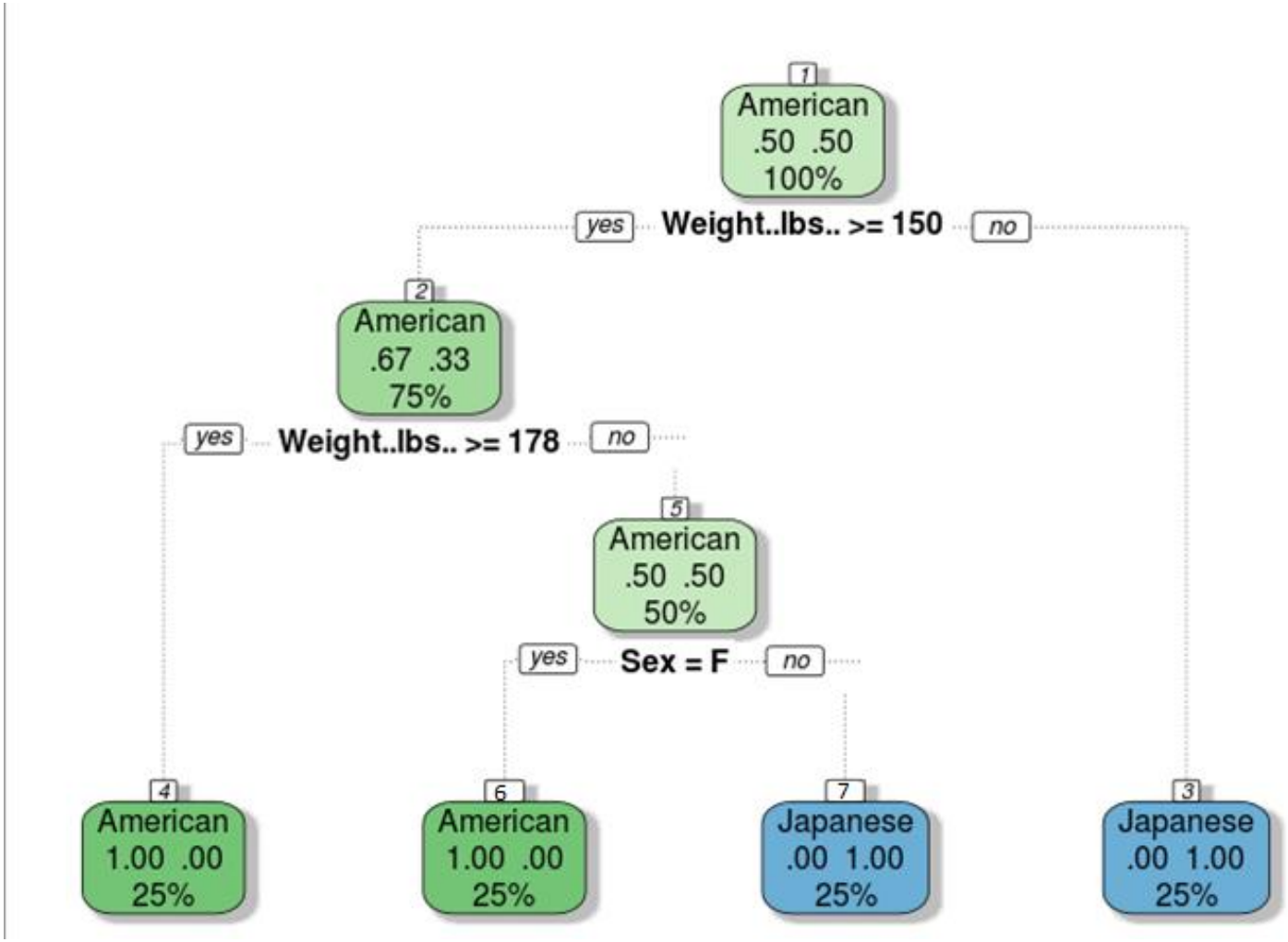
Random Forest classifier

The decision tree: third feature selection rule



Random Forest classifier

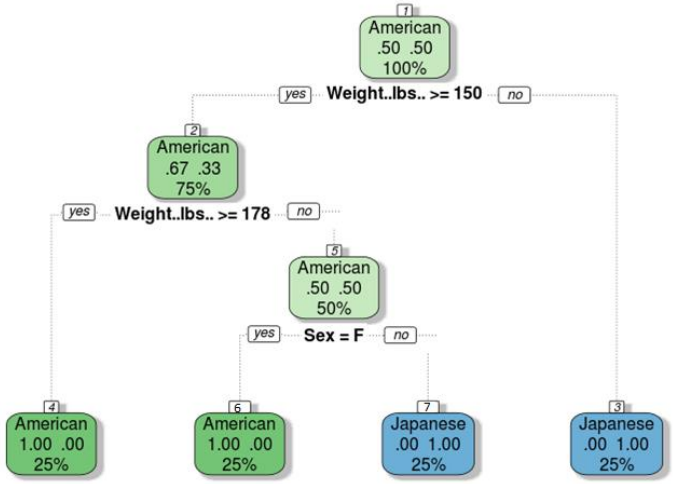
The final decision tree: all subgroups satisfy *Step 1*!



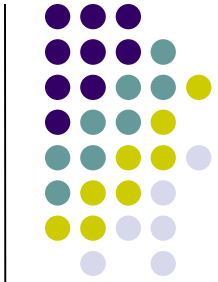
Random Forest classifier

Wanna replicate it? Two lines of command!

```
library(rpart)

library(rattle)

nation <- read.csv("Nationality.csv", stringsAsFactors=FALSE)

fit <- rpart(Nationality ~ Sex + Weight, method="class", data=nation,
    minsplit=2, minbucket=1)

fancyRpartPlot(fit, palettes = c("Greens", "Blues"), sub = "")
```



Random Forest classifier

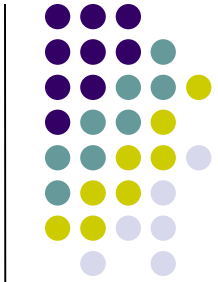


We can now use the decision tree (and its rules) just obtained to estimate the nationality of any individual not original included in our training-set

For example, if I am telling you that we have a female that weights more than 178 pounds…

…by applying the decision tree just fitted, we would predict that individual as being an American one!

Random Forest classifier



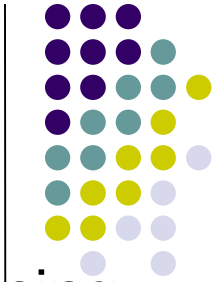
In this example the tree can **perfectly explain** the data

This is a **serious limitations**! In the real world, there is overlap: there are fat (not many, still…) Japanese people and skinny (not many, still…) Americans

In other words, growing a single, deep tree using binary recursive splitting can result in a grossly overt model. In turn, this high level of in-sample predictive accuracy usually comes at the expense of high estimator variance, as single trees grown recursively can often times yield wildly different predictions as a result of small changes in the training set…**overfitting**!!!

So, what to do?

Random Forest classifier



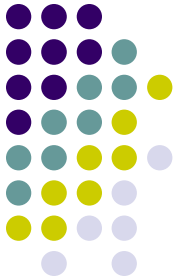
Trees are usually "pruned" to avoid **overfitting**. The pruning algorithm removes final nodes so that the model is a little more general and will tend to generalize better to new, unseen data

“Pruning” however is not always an advisable solution to the problem of overfitting!

The sequential nature of the recursive splitting algorithm means in fact that the structure of the tree is often highly sensitive to small changes in the observations included in the training-set

So what to do to avoid overfitting (part 2)?

Random Forest classifier



***B**ootstrap **agg**regat**ing** (**bagging**)!*

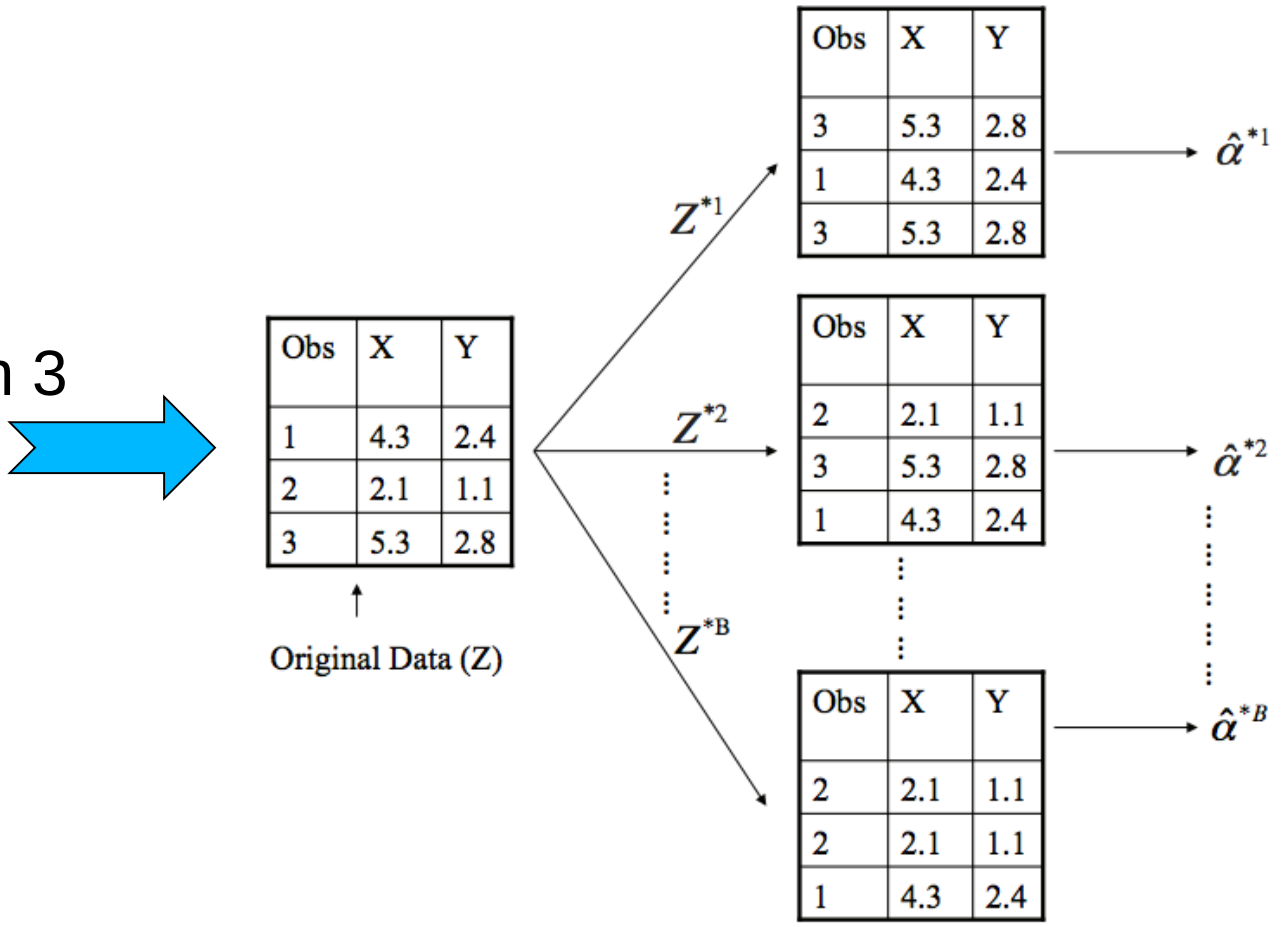
But before moving to bagging, what do you mean by **bootstrapping**?

In essence bootstrapping **repeatedly draws independent samples** from our data set to create bootstrap data sets. This sample is usually performed with *replacement*, which means that the same observation can be sampled more than once

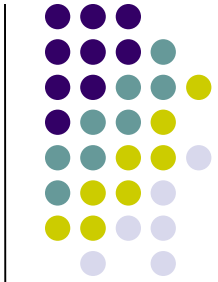
Each bootstrap is the used to compute the estimated statistic we are interested in (i.e., a mean or anything else)

Random Forest classifier

An example with 3 resamples



Random Forest classifier

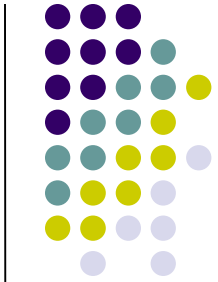


Bootstrapping is an extremely powerful statistical tool that can be used to quantify the uncertainty associated with a given estimator or statistical learning method

We can in fact use all the bootstrapped data sets to compute the standard error of the desired statistics, or their 95% confidence intervals, etc.

Moreover, and crucially given the problem discussed with respect to decision-trees, this computation is robust to (i.e., less affected from) sample specific characteristics

Random Forest classifier



B*ootstrap* ***agg****regat****ing*** (***bagging***)!

Now back to Bagging!

Bagging follows three simple steps:

1. Create *m bootstrap samples* from the training data. Bootstrapped samples allow us to create many slightly different data sets but with the same distribution as the overall training set. Note: the bootstrap samples *must not (and usually do not!)* necessary have the same size of the original training-set. Why? Give me a minute!
2. For each bootstrap sample train a single, unpruned classification tree
3. Average individual predictions from each tree to create an overall average predicted value

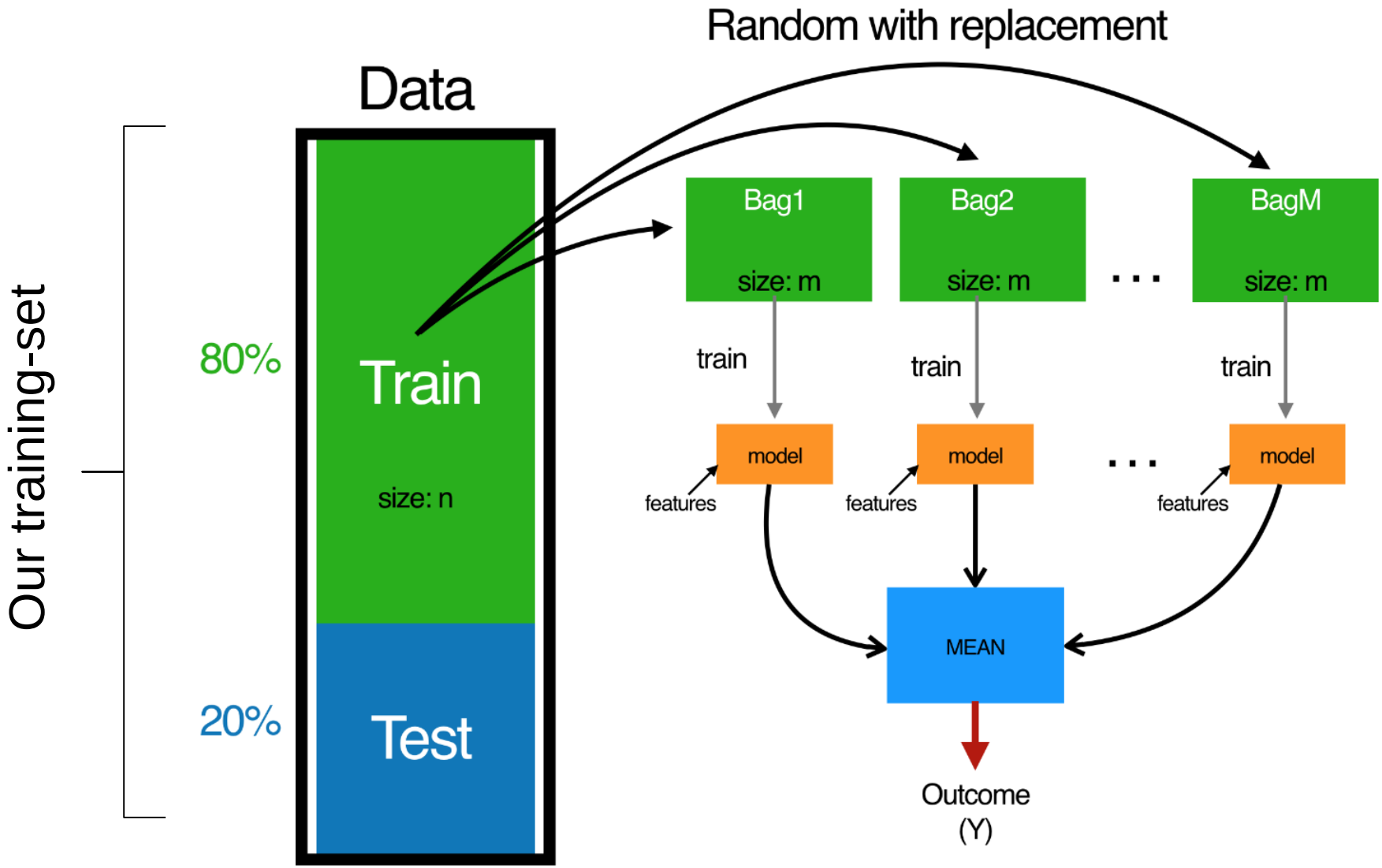
Random Forest classifier

***B**ootstrap **agg**regat**ing** (**bagging**)!*

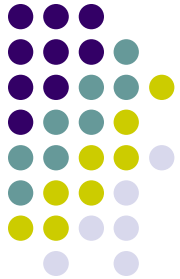
Bagging combines and averages therefore multiple models

- Averaging across multiple trees reduces the variability of any one tree and reduces overfitting, which improves predictive performance

Random Forest classifier



Random Forest classifier



As written earlier, the bootstrap samples have not usually the same size of the original training-set

On average, a bootstrap sample will contain as a default 63% of the training data (a parameter you can change!)

This leaves therefore about 37% of the data out of the bootstrapped sample. This is the *out-of-bag (OOB) sample*. What's the advantage of that?

Random Forest classifier

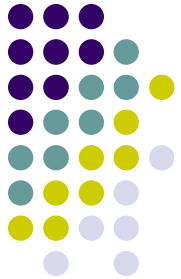
Imagine that we have 500 documents in the training-set, and imagine that our bootstrapped sample is based on 400 documents

We can then estimate the ML algorithm on such 400 documents (i.e., $\boldsymbol{Y}_{boot\ train} = \hat{f}(\boldsymbol{W}_{boot\ train})$), and using such model to classify the 100 documents in the OOB sample (i.e., $\widehat{\boldsymbol{Y}_{OOB}} = \hat{f}(\boldsymbol{W}_{OOB})$)

Then we can contrast the prediction of our model with the "true" value of the OOB sample (that we know about, given that such sample is included in the training-set after all!) to evaluate the accuracy of our prediction! (i.e., $\widehat{\boldsymbol{Y}_{OOB}} = \boldsymbol{Y}_{OOB}$)

We can use the **OOB observations** to produce therefore a natural cross-validation (?!?) process: more on this later on!

Random Forest classifier



Still, bagging for itself cannot be enough…

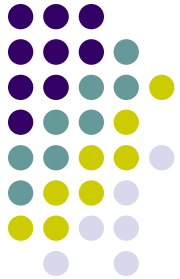
Bagging trees introduces a random component in to the tree building process that reduces the variance of a single tree's prediction and improves predictive performance

However, the trees in bagging are not completely independent of each other since **all the original predictors are considered at every split of every tree**

Therefore, trees from different bootstrap samples typically have similar structure to each other (especially at the top of the tree) due to underlying relationships

And so? How to reduce the correlation among trees?

Random Forest classifier



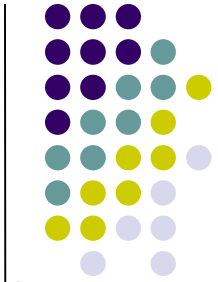
The Random Forest (RF) idea!

Let's inject more randomness into the tree-growing process. RF achieve this in two ways:

Bootstrap: similar to bagging, each tree is grown to a bootstrap resampled data set

Split-variable randomization (this is new!): each time a split is to be performed, the search for the split variable is limited to a random subset of *m* of the *p* variables (features). This is a tuning parameter. When m=p, the randomization amounts to using only step 1 and is the same as *bagging*

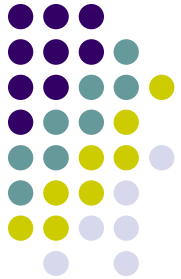
Random Forest classifier



The basic algorithm for a random forest can be therefore generalized to the following:

1. Given training data set
2. Select the number of trees to build (ntrees)
3. for i = 1 to ntrees do
4. | Generate a bootstrap sample of the original data
5. | Grow a tree to the bootstrapped data
6. | for each split do
7. | | Select m variables at random from all p variables
8. | | Pick the best variable/split-point among the m
9. | | Split the node into two child nodes
10. | end
11. | Use typical tree model stopping criteria to determine when a tree is complete (but do not prune)
12. end

Random Forest classifier



By fitting a tree (with no pruning) to each bootstrapped sample **and** by restricting the choice of each splitting variable to a random subset of predictors, we are sure that each boostrapped tree provides a truly different “perspective" on the prediction problem. All this, of course, minimizes the risk of overfitting!

The final prediction is going to be a function of each prediction in each random sample, for example it can be **the average of each prediction**

Furthermore, measures of uncertainty can be readily produced out of the bootstrapped samples!

R packages to install

install.packages("randomForest", repos='http://cran.us.r-project.org')

install.packages('caret', repos='http://cran.us.r-project.org', dependencies = TRUE)

install.packages("naivebayes", repos='http://cran.us.r-project.org', dependencies = TRUE)

install.packages("car", repos='http://cran.us.r-project.org')