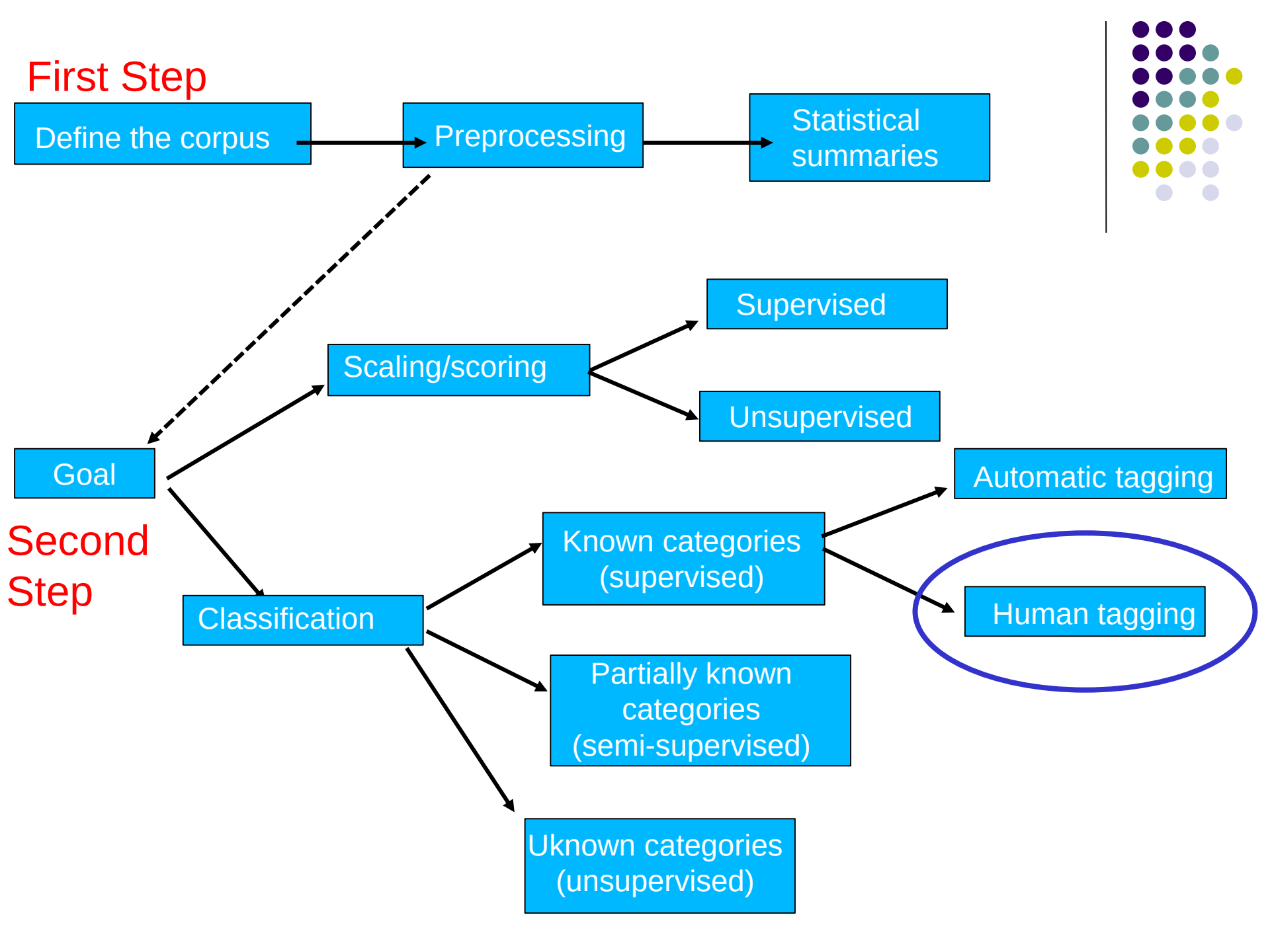
Big Data Analytics

Lecture 4/A

Supervised classification methods (part 2)

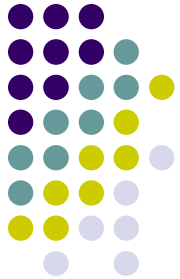
UNIVERSITÄT LUZERN


First Step
Define the corpus
Preprocessing
Statistical summaries
Supervised
Scaling/scoring
Unsupervised
Goal
Automatic tagging
Second Step
Known categories (supervised)
Classification
Human tagging
Partially known categories (semi-supervised)
Uknown categories (unsupervised)

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

Support Vector Machine (SVM)



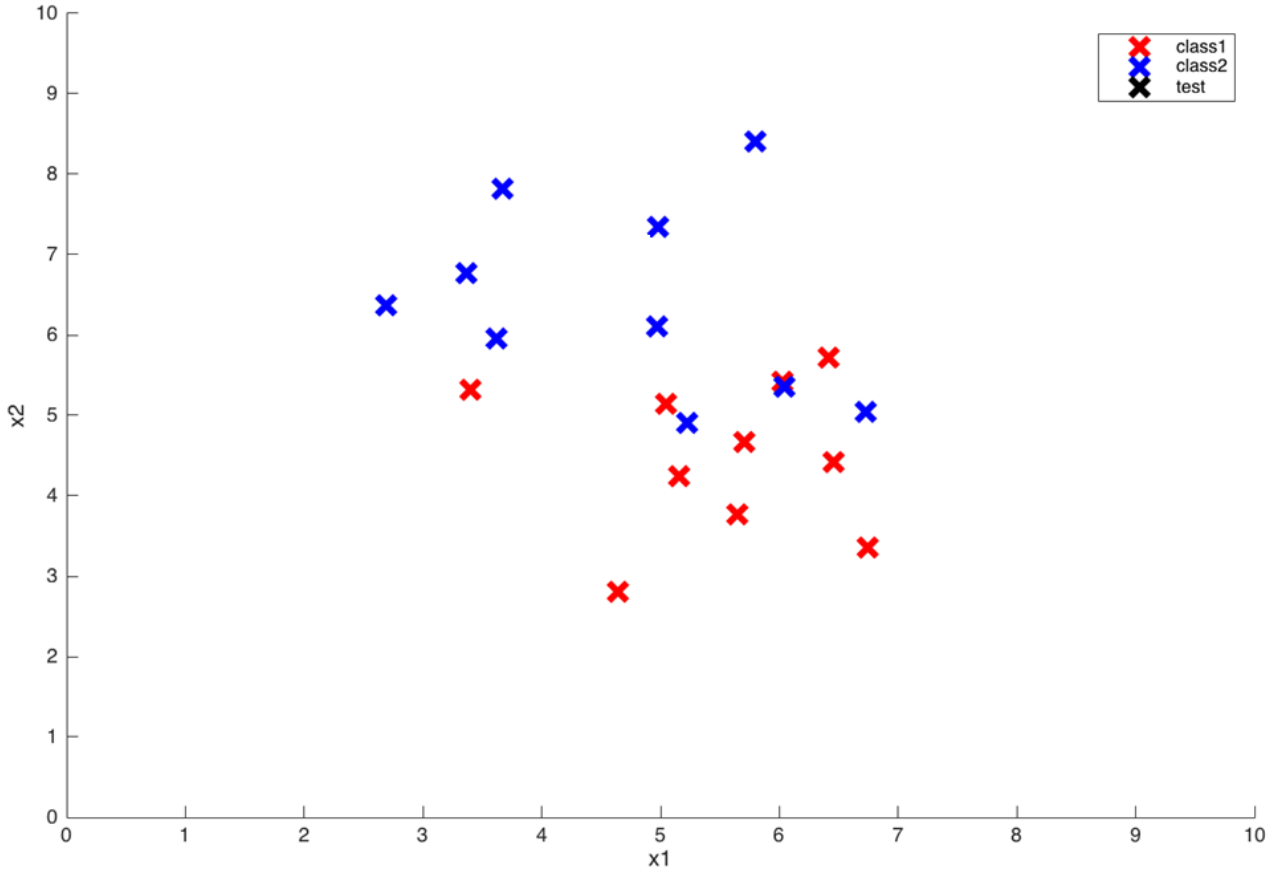
SVM is a generalization of **Nearest Neighbor** (NN) algorithm

NN is a rather simple algorithm. You are given a training data consisting of *m* training documents $\{(\boldsymbol{x}^1, y^1), (\boldsymbol{x}^2, y^2), \ldots (\boldsymbol{x}^m, y^m)\}$, where x is a vector of possible variables and y^i is the class label (the category) of i^{th}

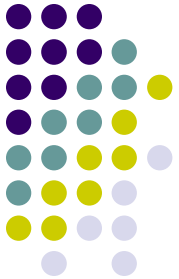
Support Vector Machine (SVM)

For example, in the figure below, we have two variables

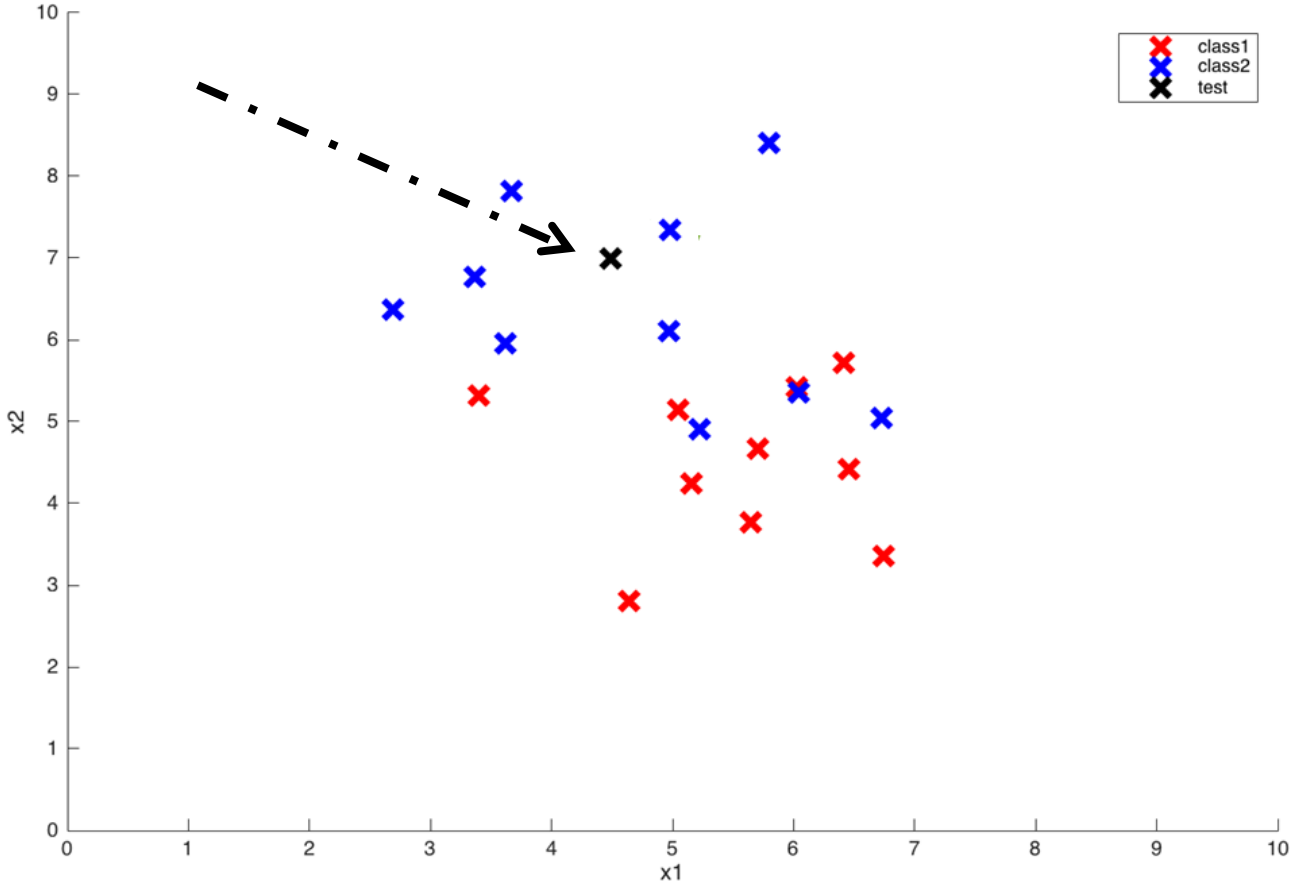
Xs: each document can be either label 1 (blue points), or label −1 (red points)



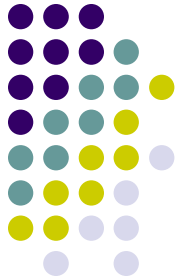
Support Vector Machine (SVM)



Now you are **given a test point** (the black x below), and you have to predict its class, whether it belongs to red class or blue class



Support Vector Machine (SVM)

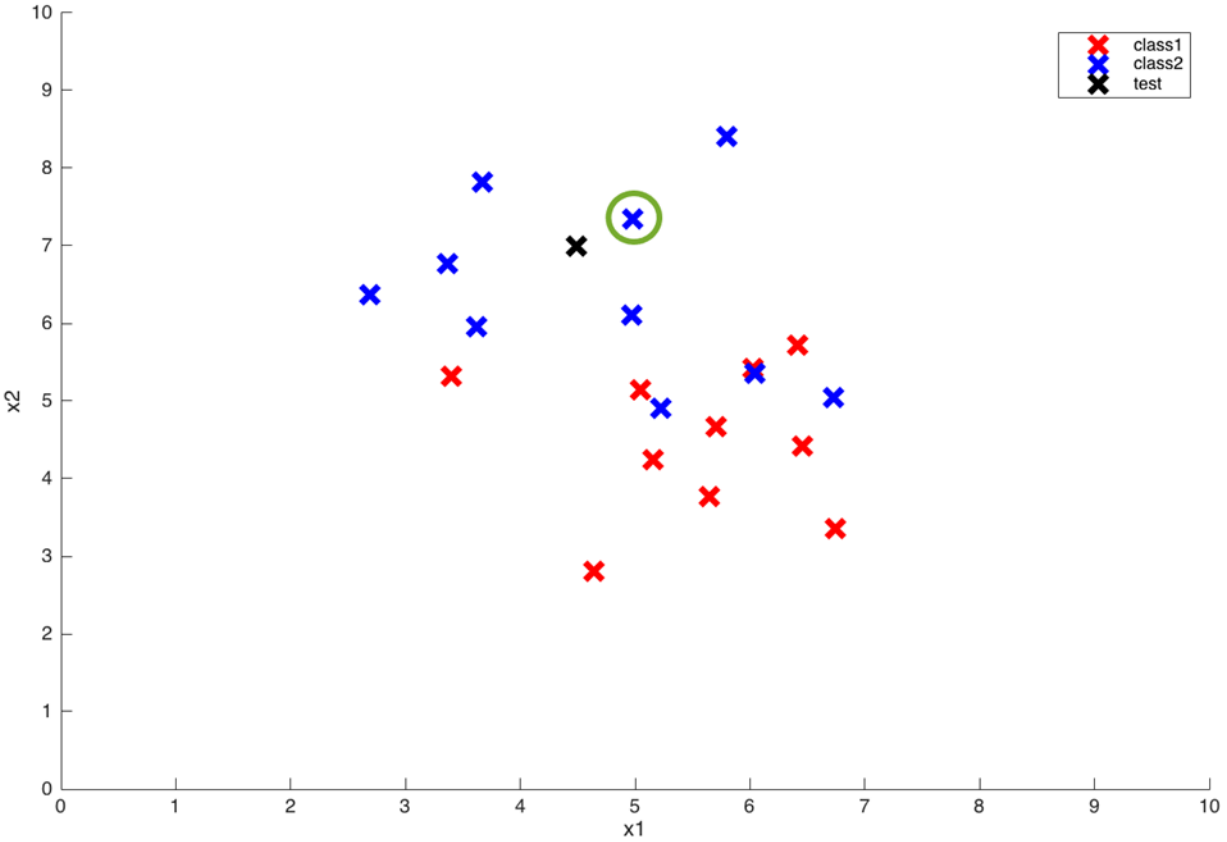


The NN algorithm finds the nearest training point to this test point (by measuring the distance of test point from **every** training point), and the class predicted of test point is the same as of nearest training point

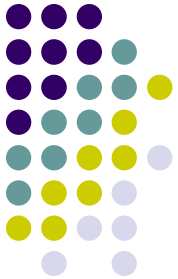
The lower the distance, the higher the **similarity** between two points

Support Vector Machine (SVM)

For example, the circled point is closest to test point, and hence class of test point is blue



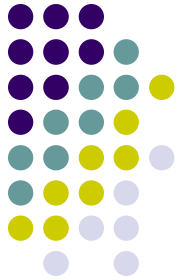
Support Vector Machine (SVM)



Two observations about NN algorithm:

- ✓ We don't do any computation alone with training points. Only when a test point comes, we compute similarity from **every** training point. This is a big disadvantage of NN algorithm
  - Consider having millions of training points, and for every test point, we have to calculate millions of similarities from test point. We calculate similarities from the training points which are very far from test points, which are not really required
- ✓ We don't give any importance to other training points **except** the nearest one

Support Vector Machine (SVM)



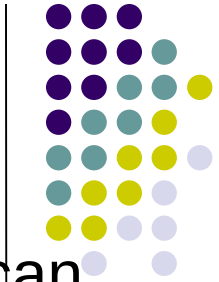
SVM remove each of these two problems

Instead of finding similarity from every training point of any test point, we calculate similarity from only a **subset** of training points (or documents, when dealing with text classification), which we compute in the **training phase**

These selected training points are called **support vectors,** since only these points will support our decision of selecting the class of a test point

Our hope is that our training phase finds as few as support vectors so that we have to compute fewer number of similarities

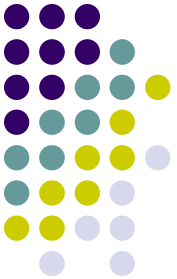
Support Vector Machine (SVM)



Moreover, once we have selected support vectors, we can assign a **weight** to each support vector, which basically tells how much importance we want to give to that support vector while making our decision

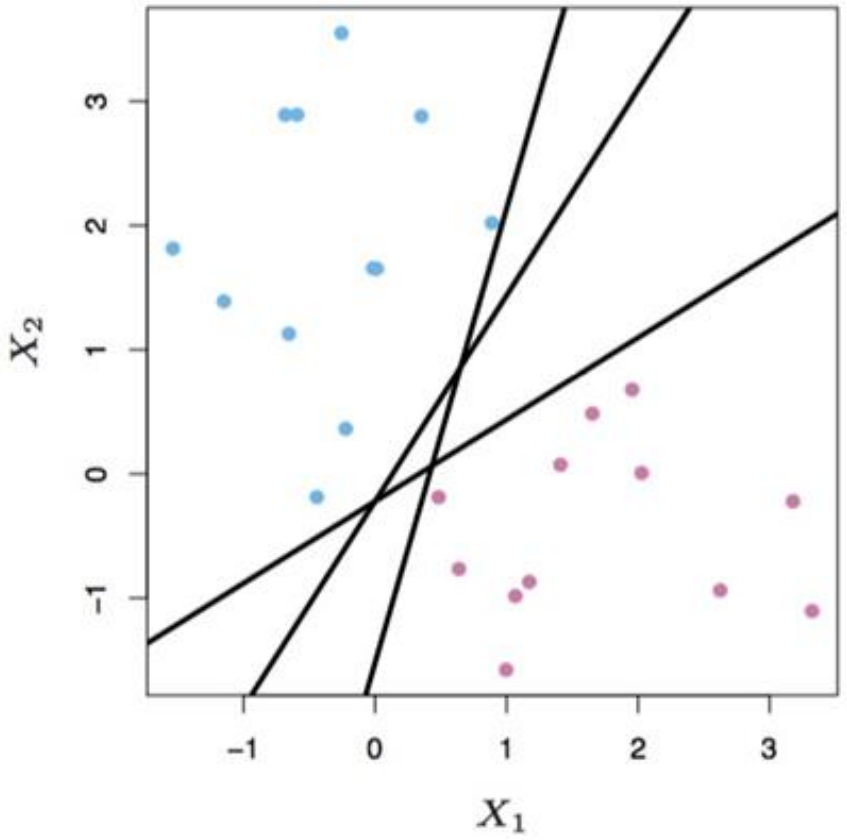
So unlike NN, we don't give importance to only a single training point (i.e., *document*, given that we are dealing with text classification), instead we give each support vector a separate importance

Support Vector Machine (SVM)

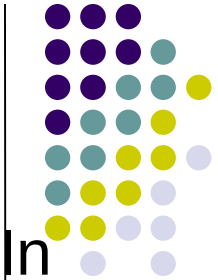


But how to find a **support vector**?

Intuition: first, we need to find the best line that separates observations belonging to different classes!



Support Vector Machine (SVM)

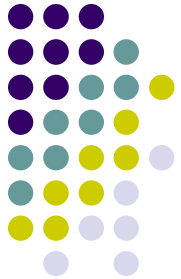


This is harder to visualize in more than two dimensions. In this case you need an hyperplane…a what?!?

An hyperplane is n-1 dimensional subspace of an n-dimensional space (a line in 2D, a plane in 3D and an hyperplane in higher dimensions)

But not only that…

Support Vector Machine (SVM)



We want to find an hyperplane that **best separates two classes of points with the maximum margin** (i.e., we try to find that separating hyperplane from which distance of closest training points is maximum) thus producing the “cleanest" possible sorting of observations

That is, our goal is to identify the hyperplane that *maximizes* the total distance between the line and the closest point in each class

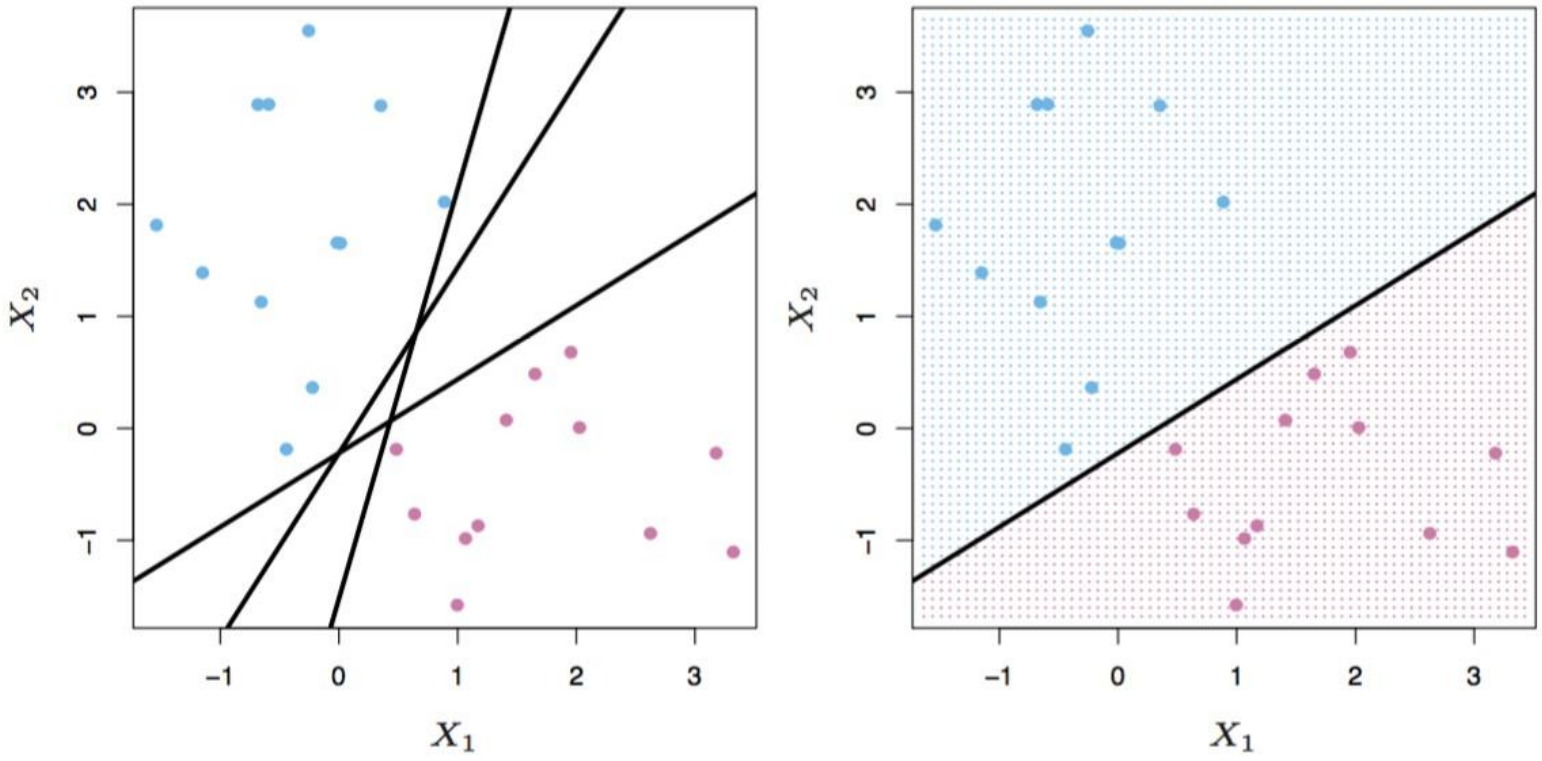
Support Vector Machine (SVM)

Essentially, this is a **constrained optimization problem** where the **margin is maximized** subject to the constraint that it **perfectly classifies the data**

Intuitively, the "maximum-margin" line allows for noise and is most tolerant to mistakes on either side. So that a good thing to look for!

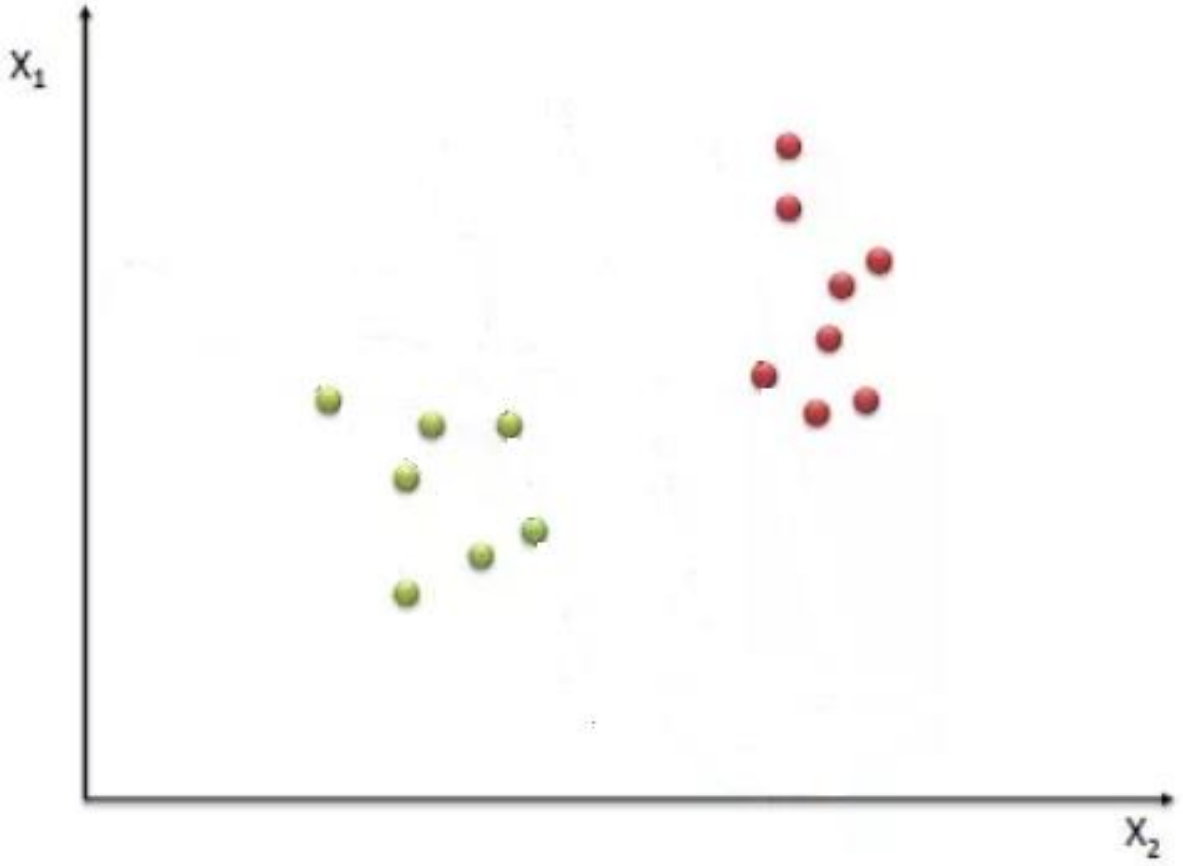
Support Vector Machine (SVM)

In the previous example, there are an infinite number of lines that will accomplish this task, but only **one** "maximum-margin" line

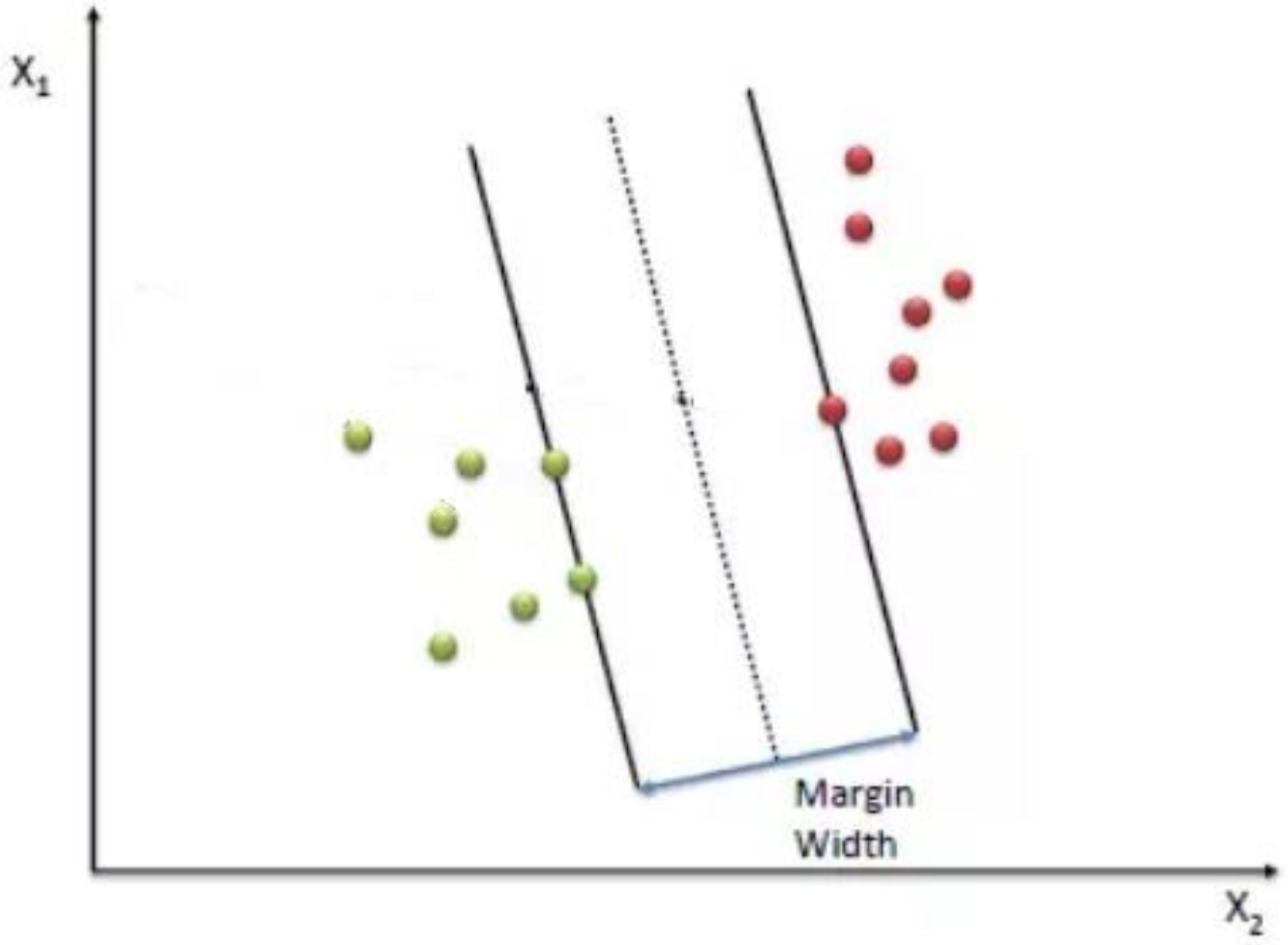


Support Vector Machine (SVM)

Another example: suppose that we want to split the below red circles from the green ones by drawing a line



Support Vector Machine (SVM)

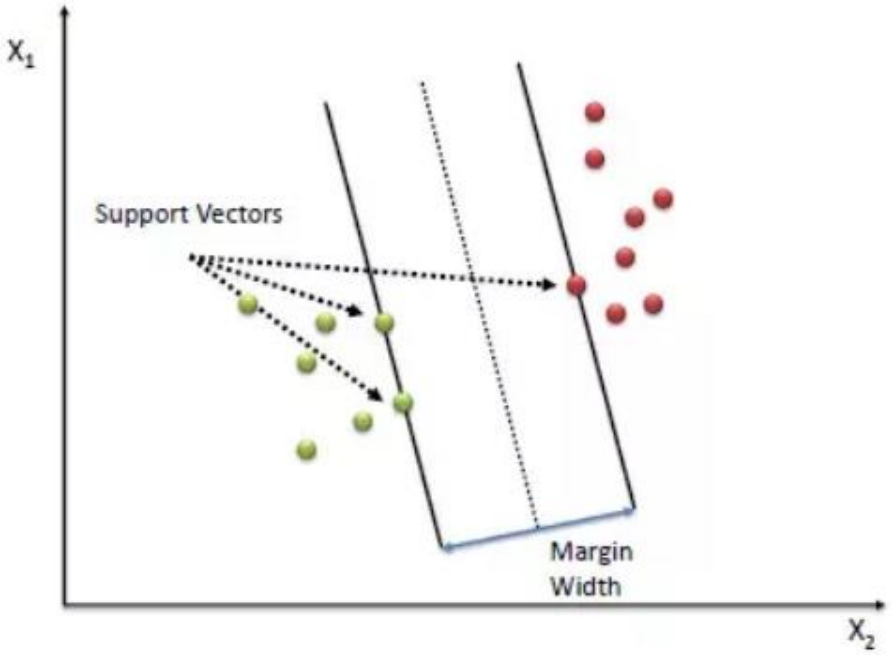


Support Vector Machine (SVM)

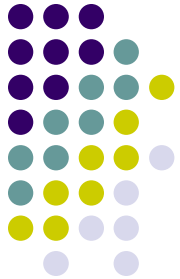
The data points that kind of "support" this hyperplane on either sides (i.e., the closest training points to the line) are called the **support vectors**

Support Vectors are simply the co-ordinates of individual observations (i.e., in text analysis: documents)

In the figure, there are 3 support vectors, so at the test time, we will compute similarity test point **on only these 3 support vectors**



Support Vector Machine (SVM)



So far, we have assumed that a hyperplane can **perfectly separate instances across classes**

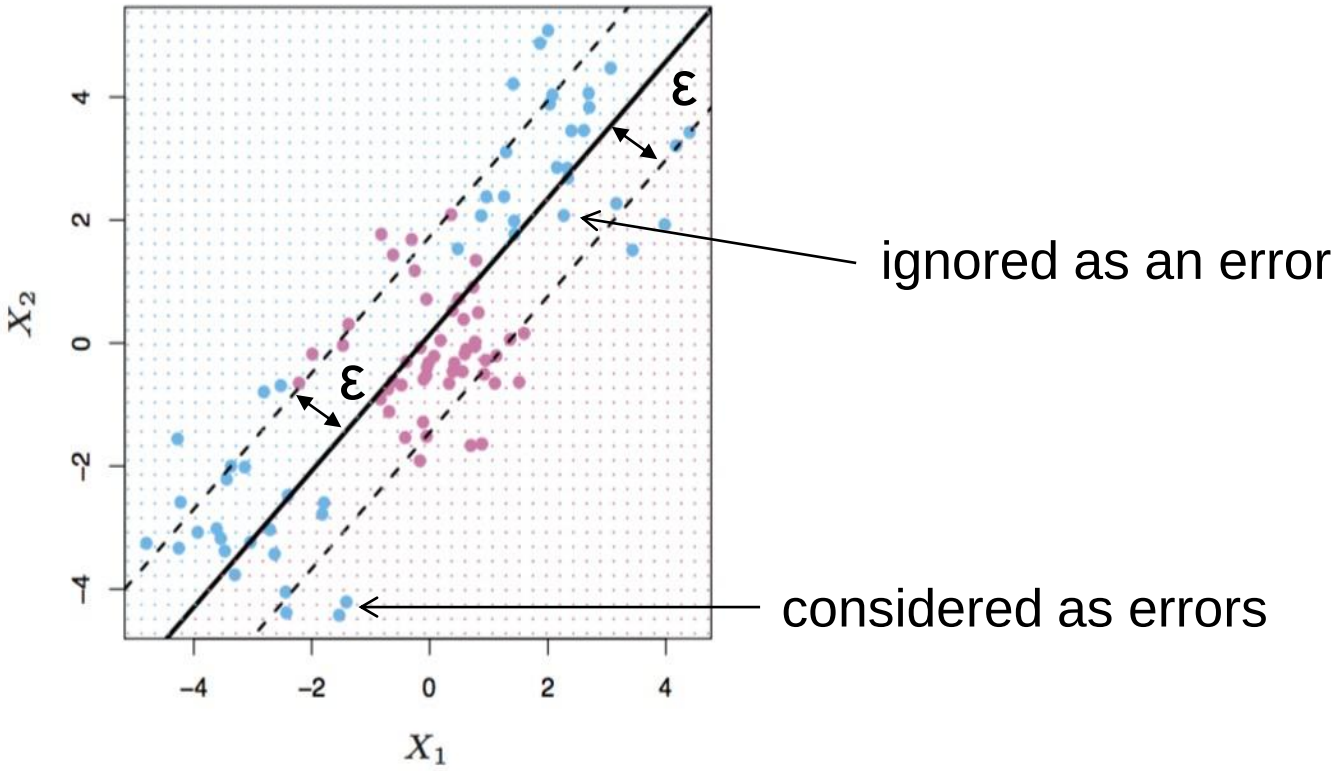
When this is not the case, we must relax the constraint imposed on the distances between points and the hyperplane, and allow for a **certain amount of slack**

This slack will allow for instances to be within the margin, or even to cross the (quasi)separating hyperplane

Support Vector Machine (SVM)

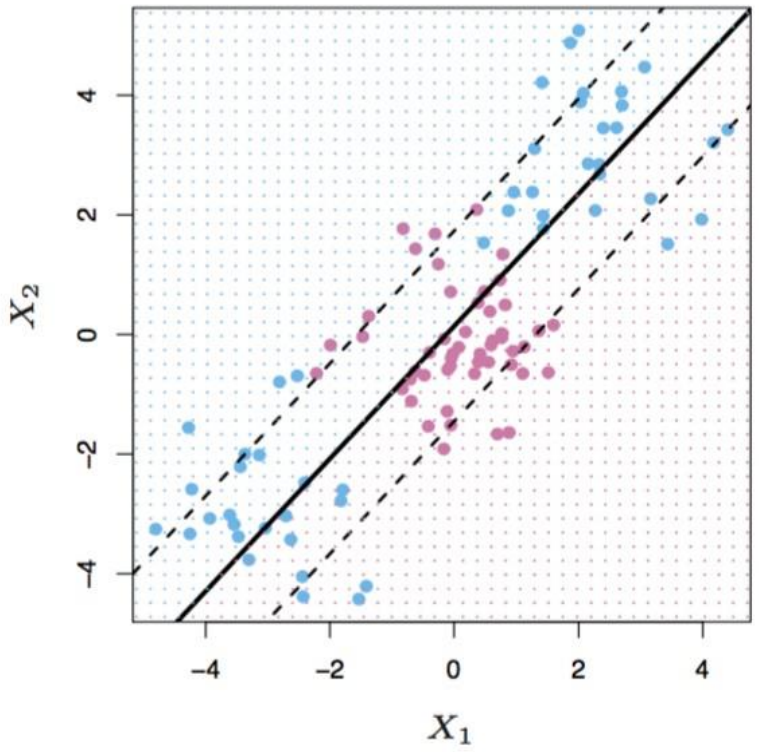
In this respect we can define a *loss function* that *ignores* those errors which are situated within the certain distance of the true value

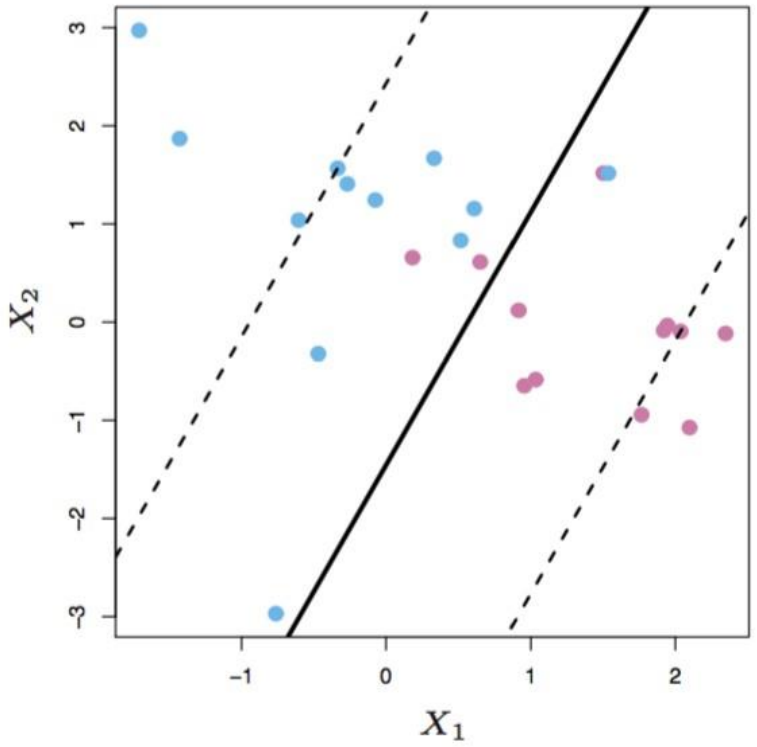
This type of function is often called “epsilon (ε) intensive-loss function”



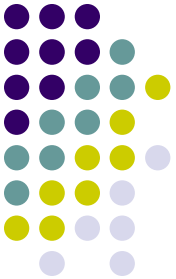
Support Vector Machine (SVM)

The width of the intensive-loss zone can of course be (very) different!





Support Vector Machine (SVM)

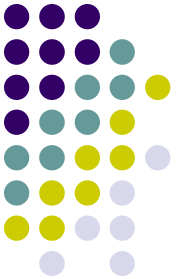


This function allows us to identify the **cost** of the errors on the training points

These are zero for all points that are inside the band (i.e., that are within ε distance of the observed value), and larger than 0 for all points outside of it

This penalty for the errors is known as *C* (i.e., cost) and it quantifies the *penalty* associated with having an observation on the wrong side

Support Vector Machine (SVM)



With no perfect separation, the goal is therefore to minimize our sum of classification errors, conditioning on the **tuning parameter** *C* (i.e., cost) that indicates tolerance to errors

In particular, parameter *C* determines the trade-off between the *model complexity* and the degree to which deviations larger than epsilon are *tolerated* in optimization formulation

Support Vector Machine (SVM)

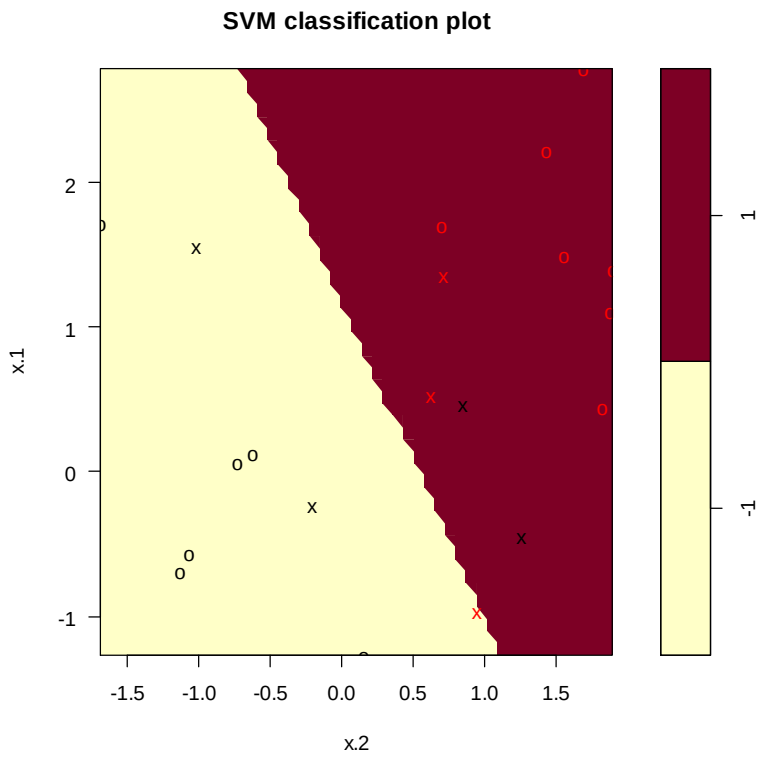
Larger values of *C* (ex. *C* = 1000 , a value which penalizes a lot the model for misclassified observations) thus result in greater focus of attention on the points located **very close** to the decision boundary (for a given ε), i.e., only those **instances near the class boundary play a big role its definition**, while those that remain far away from the boundary have **little effect** on its location and direction

…while smaller values of *C* (ex. *C* = 0.01, a value which doesn't penalize the model much for misclassified observations) involve an attention also on data points **farther away** (for a given ε). It is these points that now can also become support vectors

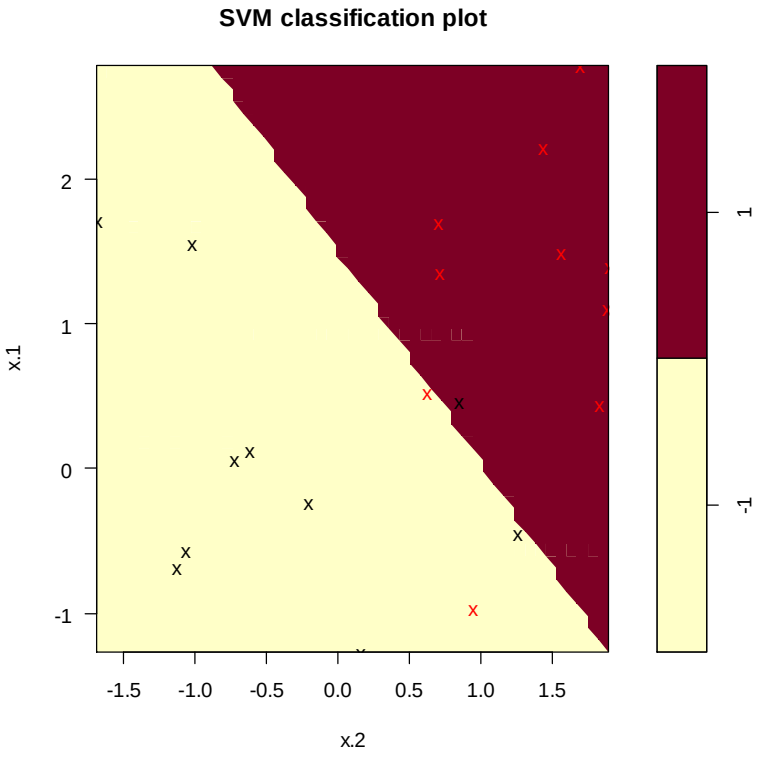
Support Vector Machine (SVM)

An example with two labels (red and black points) and two different values for *C* (with epsilon fixed to 0.1)

C=10; SV points (i.e., the x)=7



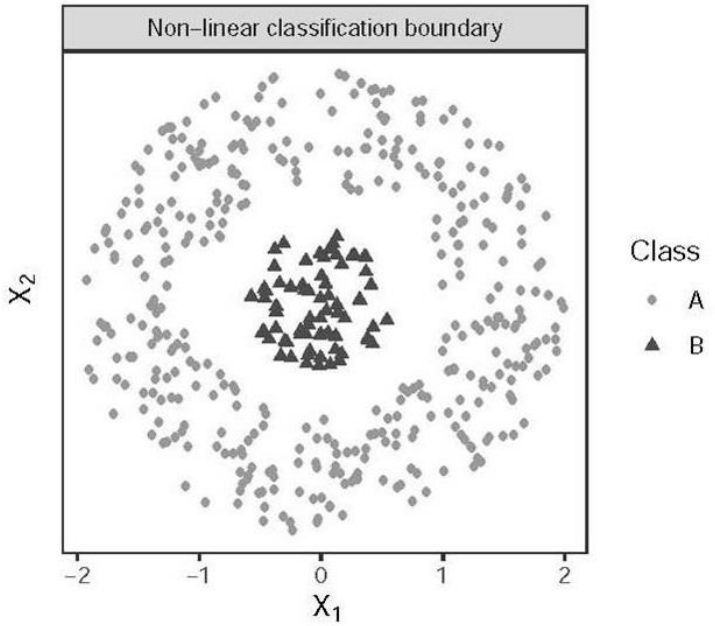
C=0.01; SV points=ALL! (although of course with a different weight!)



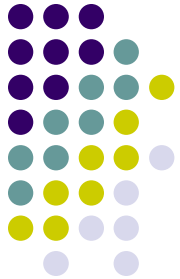
Support Vector Machine (SVM)

But what if we don't want (or **cannot**) fit a straight line to find support vectors (for example in 2 dimensions)?

For instance, in the figure below, although the two classes are easily recognized as occupying different regions of feature space, no hyperplane across it would result in a good separation, The optimal decision boundary, which in this case corresponds to a circle, is not linear



Support Vector Machine (SVM)

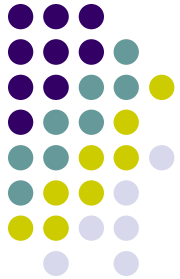


So what to do? **Move to non-linearity**!

We achieve this however *not* by drawing curves, but by "lifting" the features we observe into higher dimensions…

….i.e., instead of operating on the space defined by the *original set of predictors* (where no linear boundary can correctly separate classes of the target outcome), we can operate on a tra*nsformed space of higher dimensions* in which linear separability becomes (again) possible

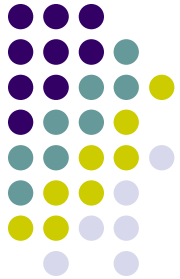
Support Vector Machine (SVM)



For example, if we can't draw a line in the space (x1, x2) then we may try adding a third dimension, (x1, x2, x1 * x2)

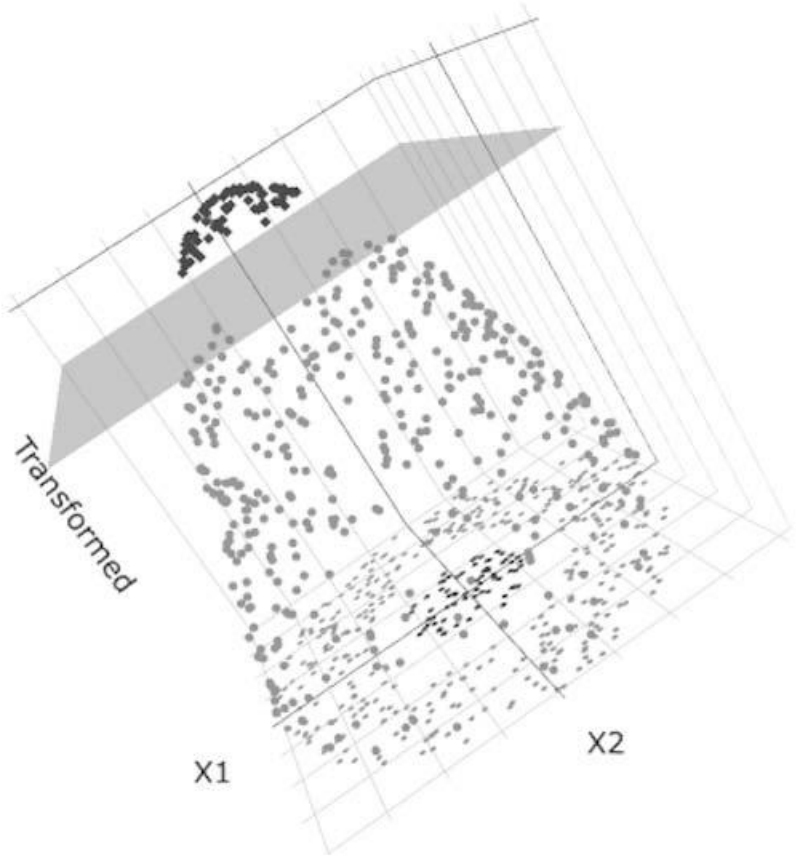
This is known as a **kernel trick**, that can be linear, radial, polynomial, etc.

Support Vector Machine (SVM)



Going back to the previous figure, suppose we add a third feature equal to the negative sum of squares of the original predictors

This results in the 3D scatterplot depicted on the figure below



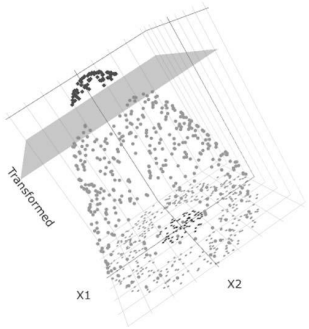
Support Vector Machine (SVM)

In this new, three-dimensional feature space, observations are now arrayed on a conical surface, with instances of class B (i.e., the triangles) rising to its apex

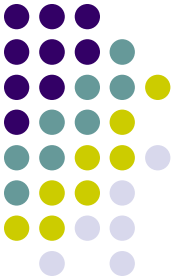
It is now easy to define a *plane*, depicted in gray, that cuts the top of this cone and separates instances of the two classes

The projection of this separating plane back onto the original two-dimensional space generates the circular decision boundary we needed

Once again, the SVM's goal is to learn this separating hyperplane



Support Vector Machine (SVM)



You can employ different Kernel transformations:

Popular SVM Kernels

Kernel	Form
Linear	$K(x, x') = \langle x, x' \rangle$
dth Degree Polynomial	$K(x, x') = (1 + \langle x, x' \rangle)^d$
Radial Basis Function	$K(x, x') = \exp(-\|\|x - x'\|\|^2/c)$

The *Linear* kernel is the simplest kernel function. It is given by the inner product <x,y> plus an optional constant

Linear SVMs are usually preferable to other kernels when the number of features in your problem is larger than the number of instances (“large p, small n”)

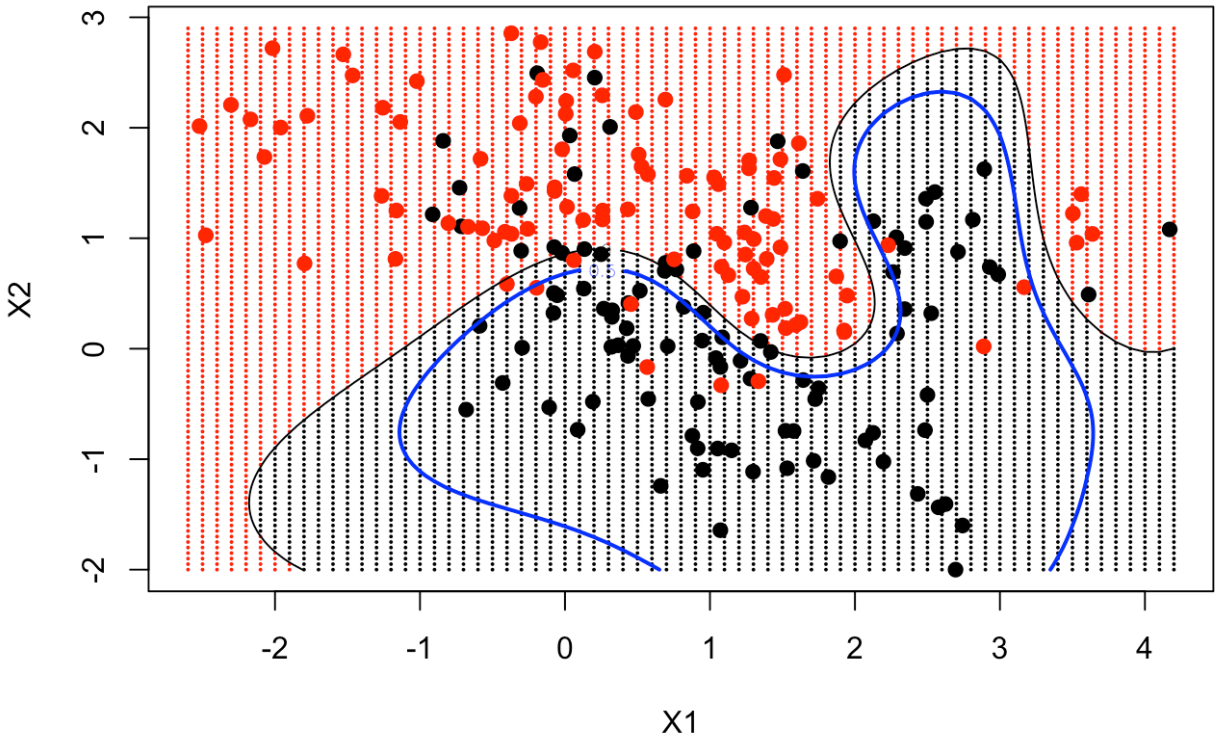
This is the typical situation you have with text-analytics!

So, a linear Kernel can be considered as the (possible) first-best when dealing with texts

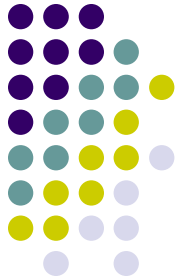
Support Vector Machine (SVM)

With other type of analysis, the RBF Kernel is a reasonable first choice

In particular, in those cases where groups are not nicely separated by lines or (hyper)planes, the RBF Kernel is able to carry out non-linear partitioning



Support Vector Machine (SVM)



How does SVM work with a multi-category classes?

In a one-vs-one SVM (as in our example), a classifier is trained to distinguish one class from another

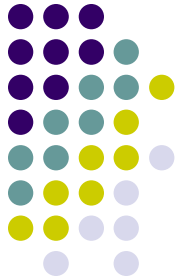
For M classes, however, you have M*(M-1)/2 combinations, which is also the number of resulting classifiers

For example, if we had 3 classes we get 3 classifiers, i.e., class1-vs-class2, class1-vs-class3, class2-vs-class3

Practically, in each case you suppose that only 2 classes exist, and you train a classifier to distinguish among them

During the prediction phase, you then choose the class with a majority vote among all the classifiers

Support Vector Machine (SVM)



SVMs have been most successfully used to solve classification problems, particularly when the number of predictive features is much larger than the number of observations n (as it happens with texts!!!)

For medium-sized datasets, Support Vector Machines (SVMs) provides, quite often, an excellent choice

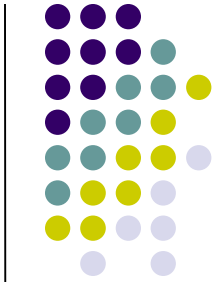
Random Forest classifier

Jordan Goldmeier
@Option_Explicit

Where do data scientists go camping?

In random forests

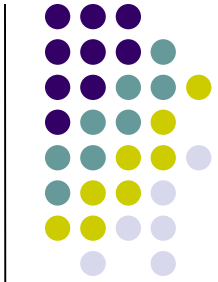
Random Forest classifier



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**

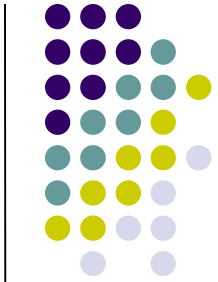
Random Forest classifier



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

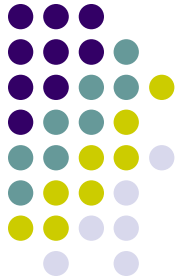
Key idea: the procedure to create decision trees is **recursive**

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 (i.e., when a given stopping criterion is reached – such as "stop when there are only % observations left in a node), the tree becomes an **endpoint or terminal-node**, 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

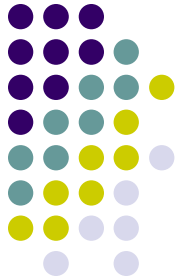


Step 2: If **S** is too large and it contains more than one class (as in our case!), find the **best rule** (i.e., the best “cutpoint”) based **on the value of one feature** to split the data into two regions

Different rules can be used in this respect. however…

- ✓ …the aim is always the same: the "best" branching rule is the one that results in the **most information gain** (i.e., the one which maximize our predictive ability – typically the proportion correctly predicted, if the DV is categorical, or mean squared error, if the DV is continuous)

Random Forest classifier



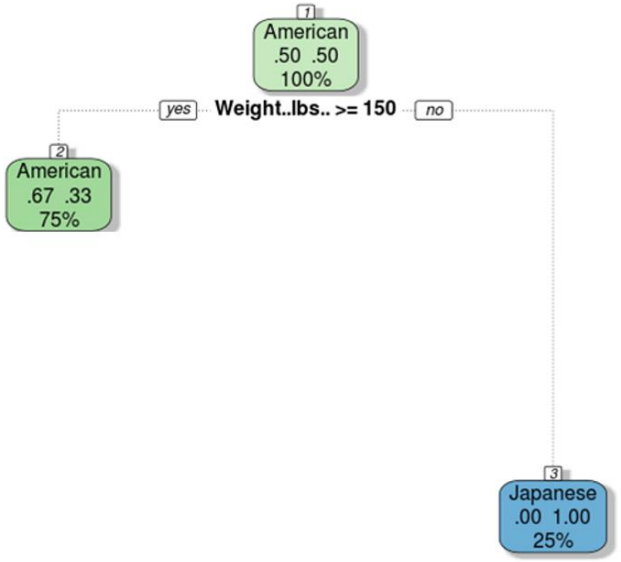
So given our initial **S**, on which feature should we focus?

- ✓ 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

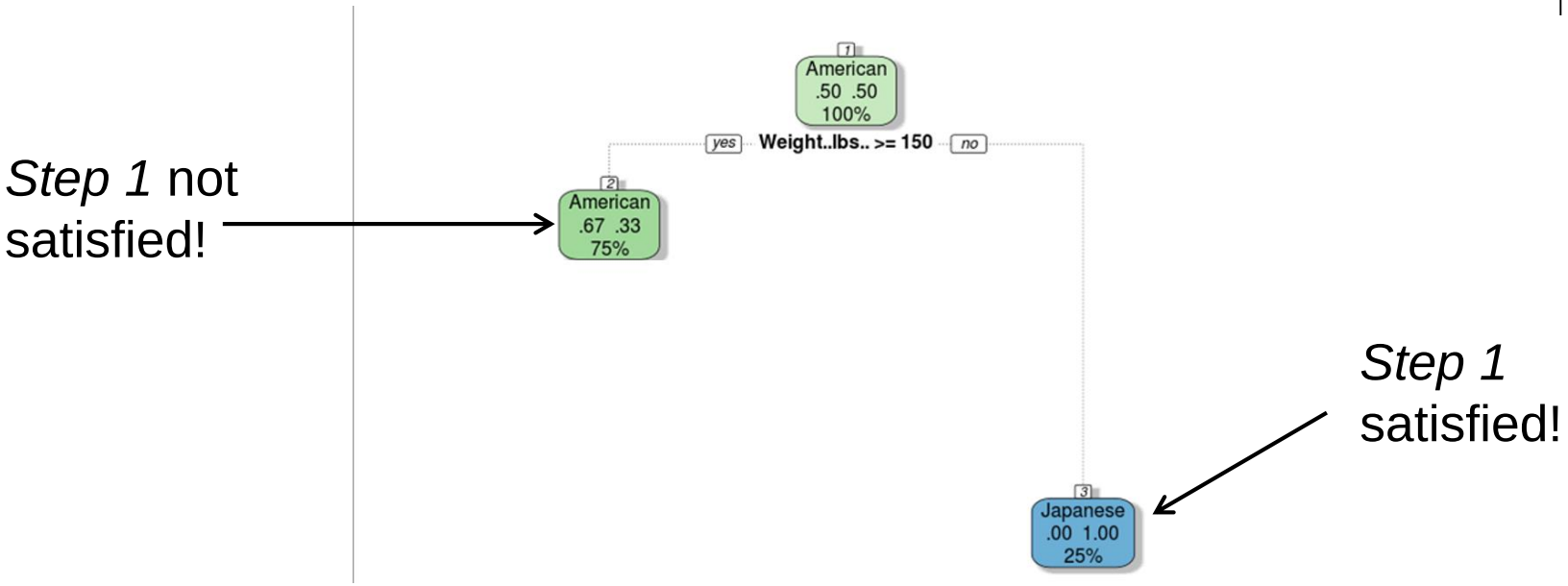
The decision tree: first feature selection rule

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

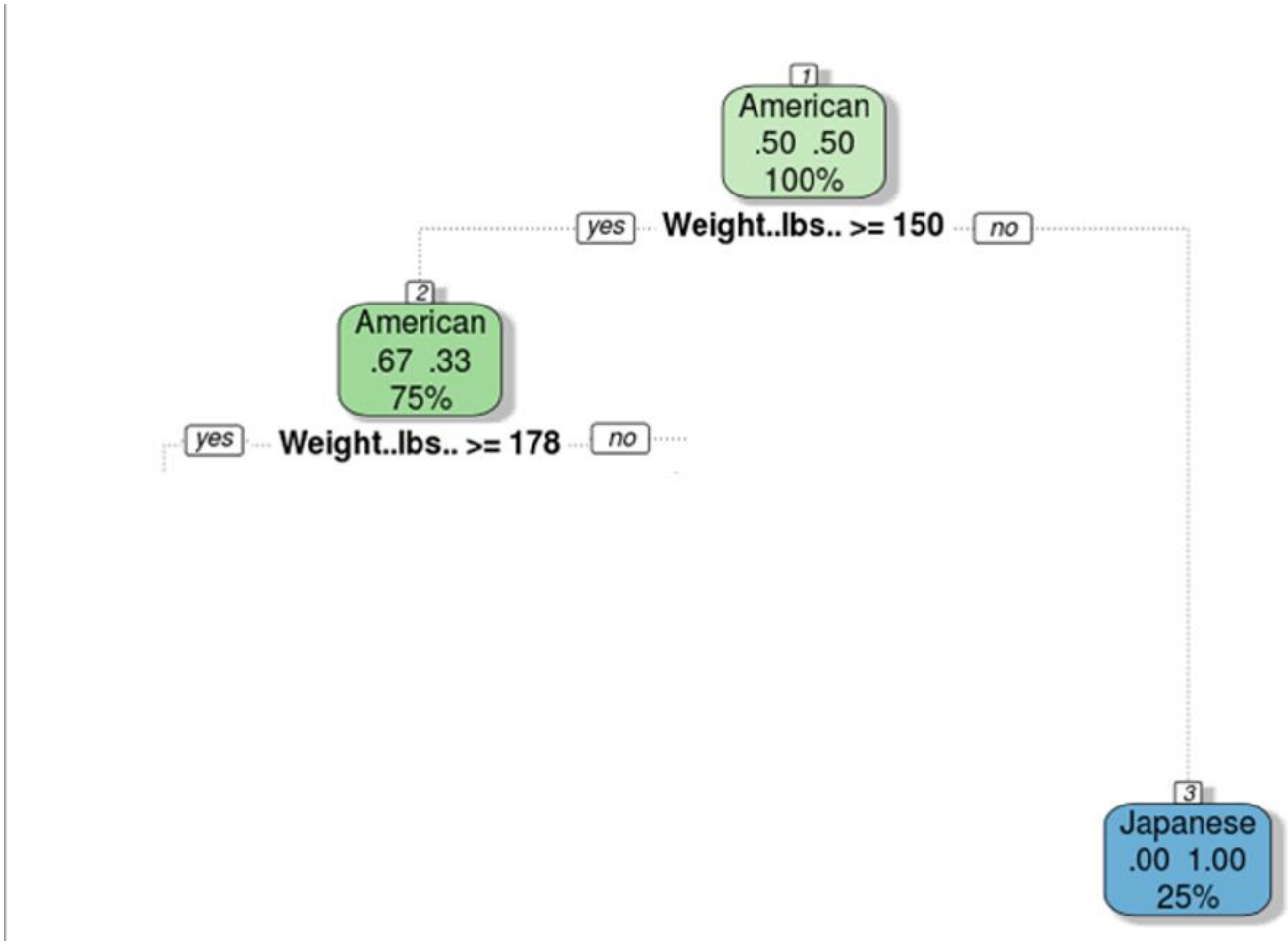


If your subsets need to go to *step 2*, apply *step 1* to the sub-subsets, etc.

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!

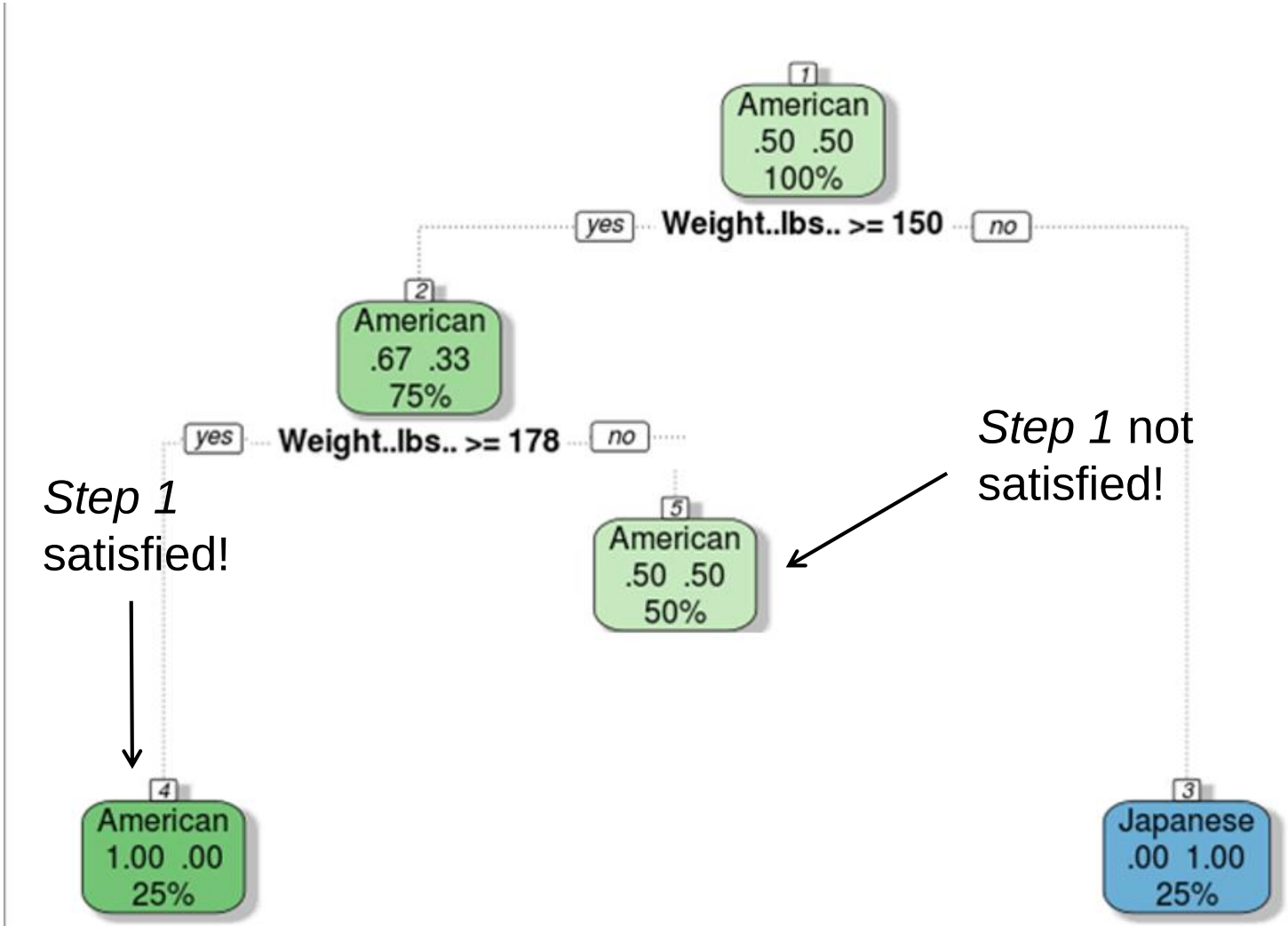
Random Forest classifier

The decision tree: second feature selection rule



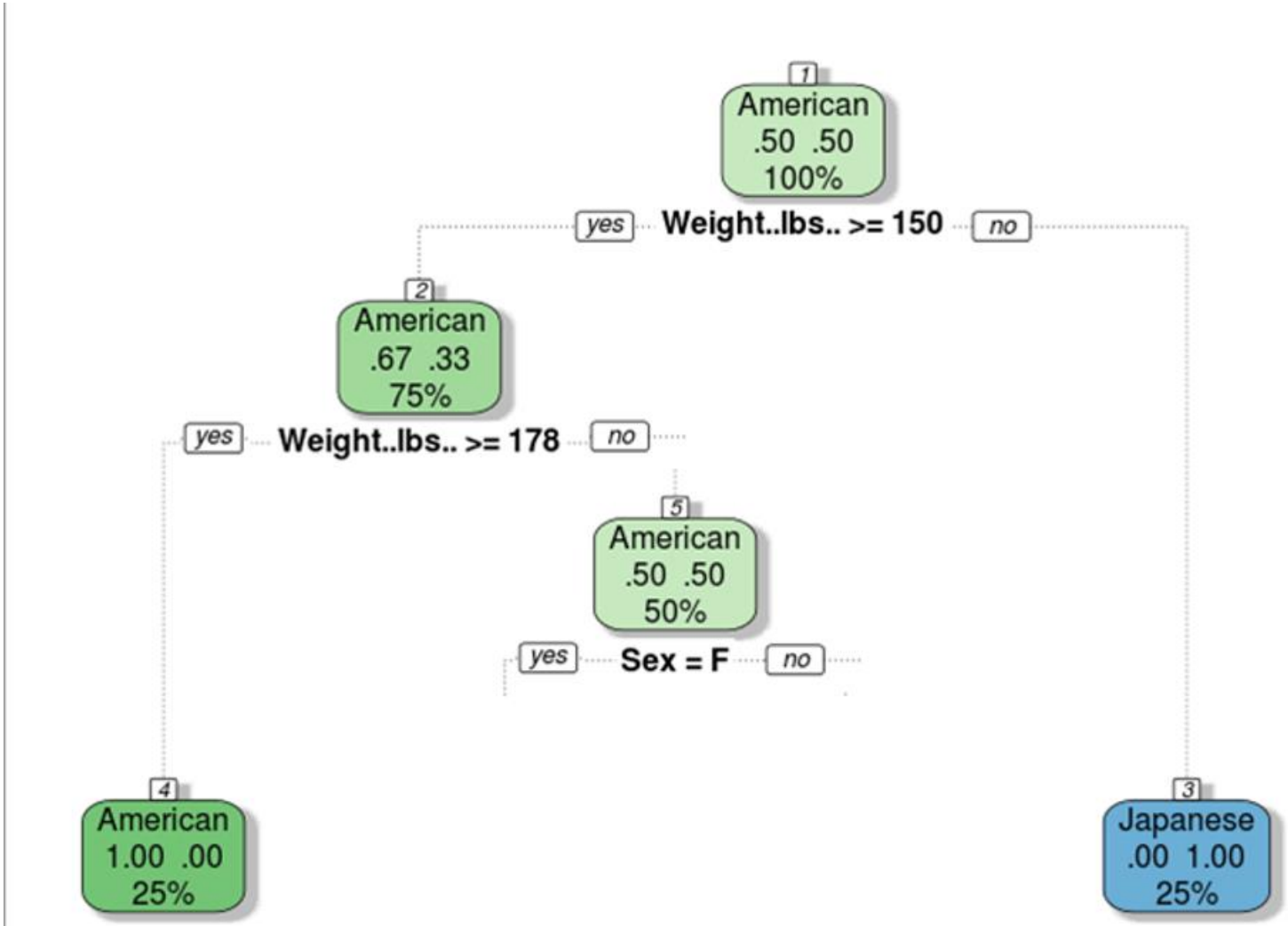
Random Forest classifier

The decision tree: second feature selection rule



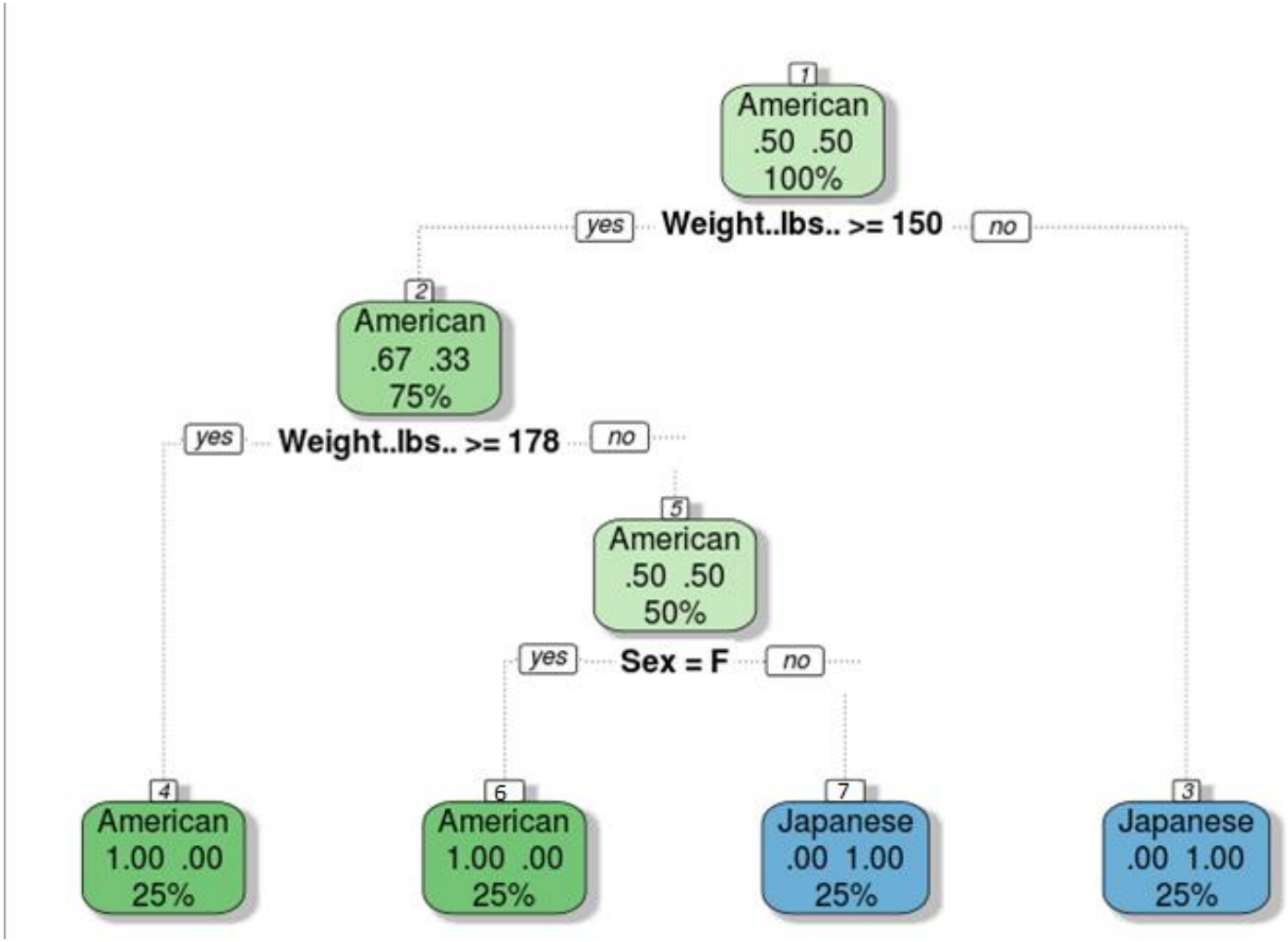
Random Forest classifier

The decision tree: third feature selection rule



Random Forest classifier

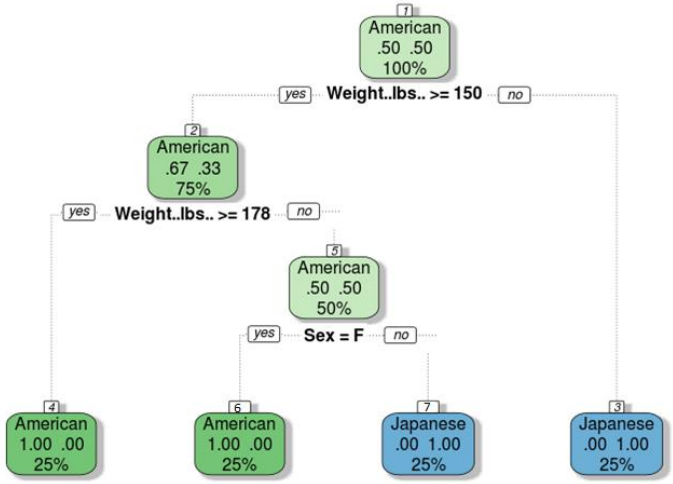
The final decision tree: all subgroups satisfy *Step 1*!



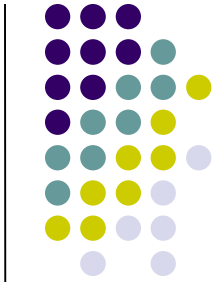
Random Forest classifier

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

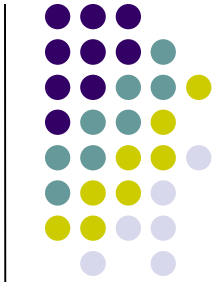


We can now use the decision tree (and its rules) just obtained to estimate the nationality of any individual not original included in our training-set

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



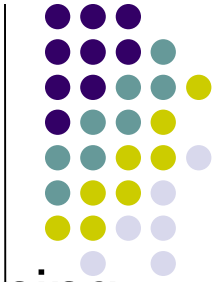
In this example the tree can **perfectly explain** the data

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**)!*

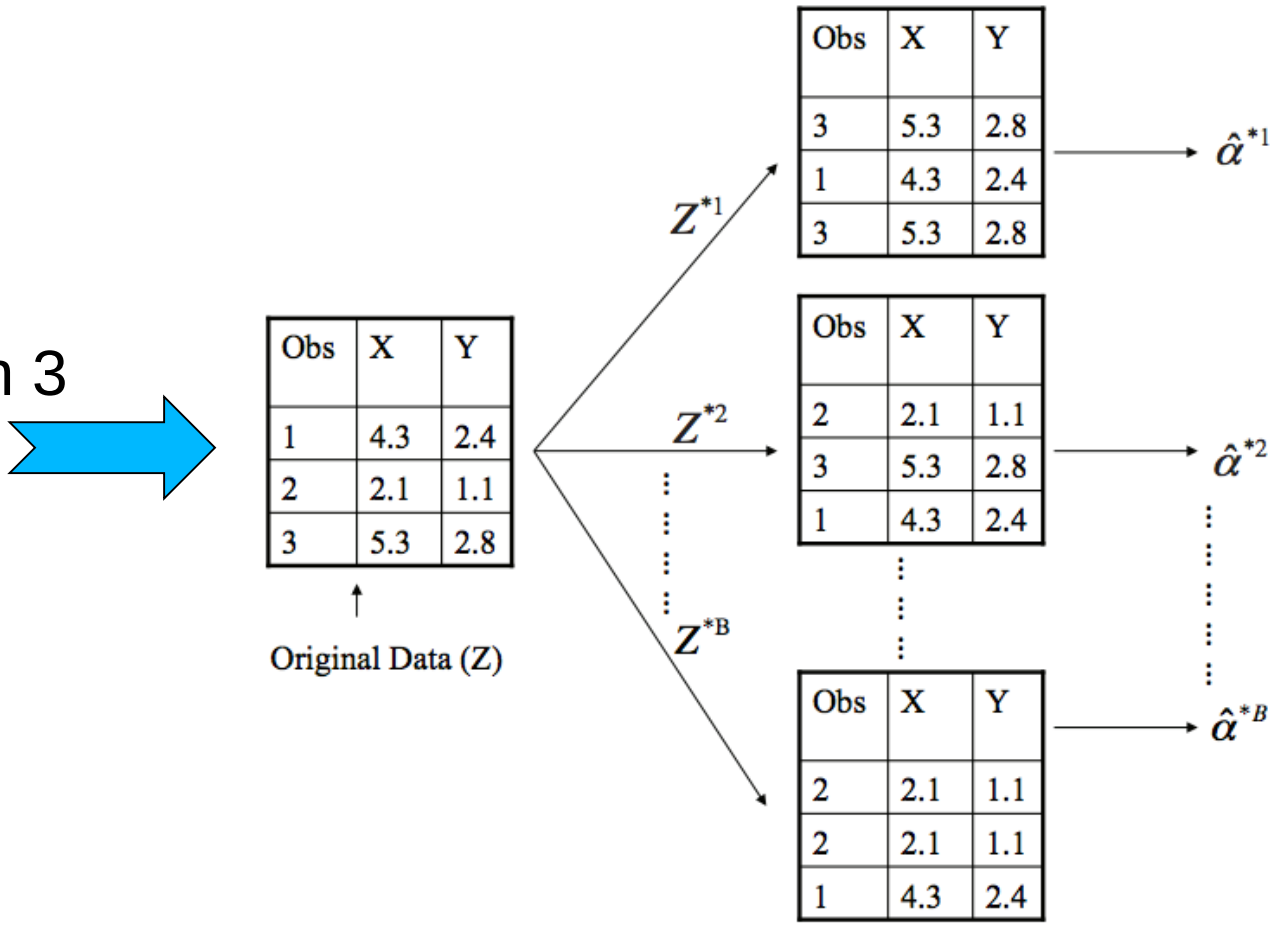
But before moving to bagging, what do you mean by **bootstrapping**?

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

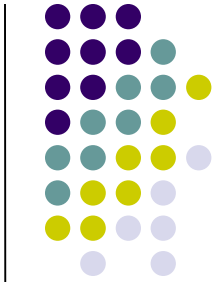
Each bootstrap is the used to compute the estimated statistic we are interested in (i.e., a mean or anything else)

Random Forest classifier

An example with 3 resamples



Random Forest classifier

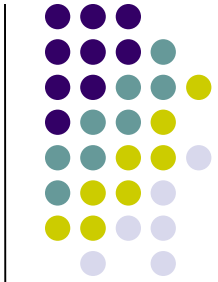


Bootstrapping is an extremely powerful statistical tool that can be used to quantify the uncertainty associated with a given estimator or statistical learning method

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
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

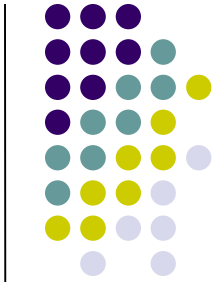
Random Forest classifier

***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



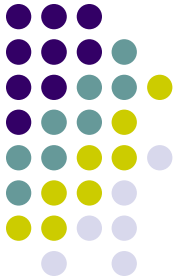
The bootstrap samples will have the same size of the original training-set. However they will be built with replacement!

More in details, bootstrapped sample will include roughly two-thirds of all unique observations in the original training-set (a parameter you can change!), meaning that the remaining one-third will be based on *replacement* (i.e., you will have some observations that will appear more than once in the bootstrapped sample)

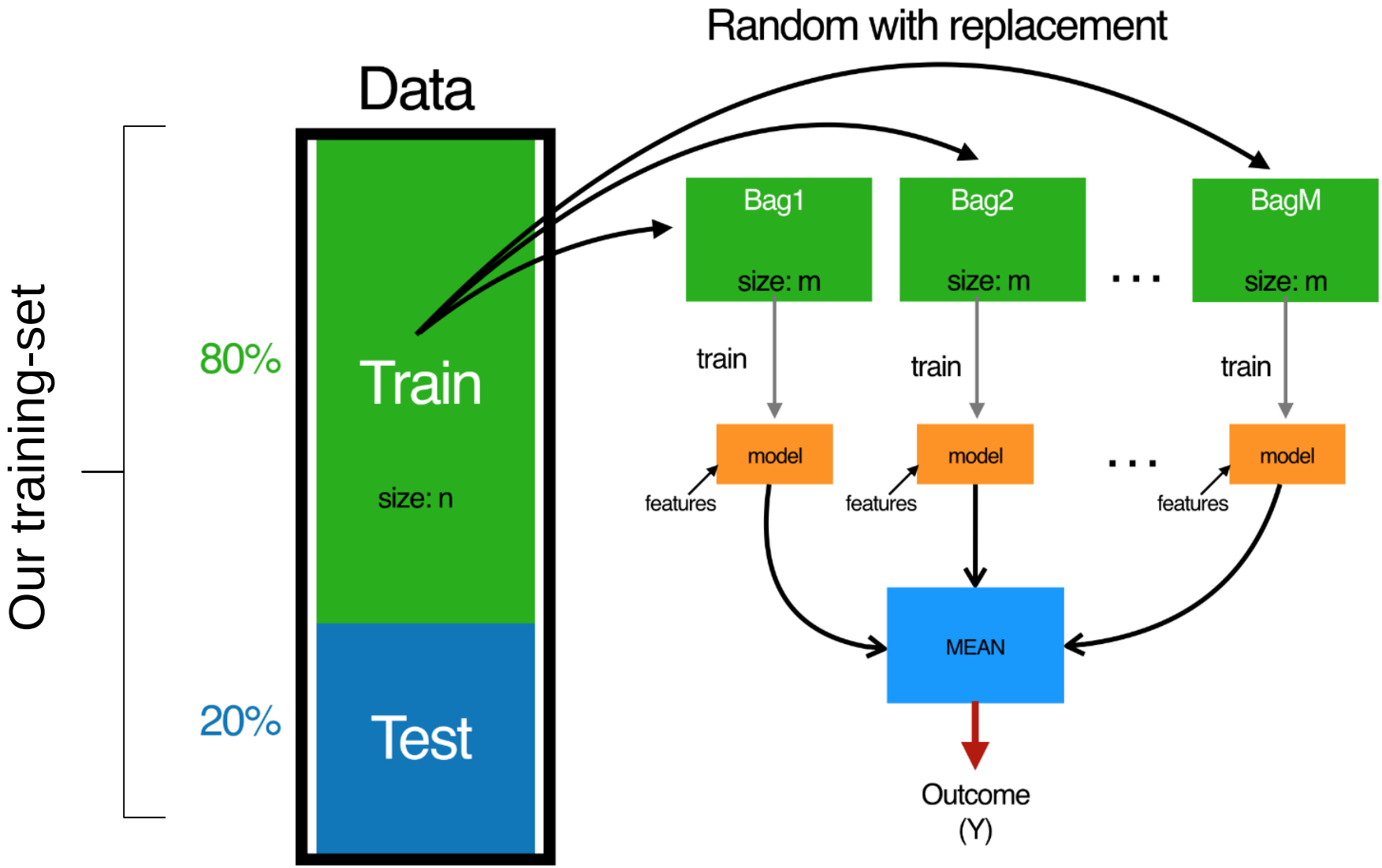
Random Forest classifier

This also means that approximately one-third of the sample observations are never drawn in a particular bootstrap sample

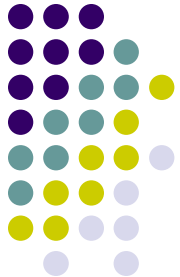
This is the *out-of-bag (OOB) sample*, since these observations are not used to train a particular regression tree (these out-of-bag samples of course differ across the bootstrapped trees, just like the bootstrap samples randomly vary from tree to tree)



Random Forest classifier



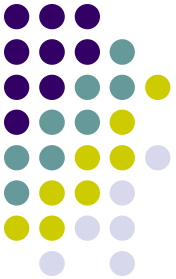
Random Forest classifier



What's the utility of having a OOB samples?

Since they were not included in the bootstrap sample for a specific tree, they can be used as an out-of-sample prediction!

Random Forest classifier



Imagine that we have 500 documents in the training-set, and imagine that our bootstrapped sample is based each time 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., $\widehat{\boldsymbol{Y}_{OOB}} = \hat{f}(\boldsymbol{W}_{OOB})$) for each specific tree we fit

Then we can contrast the prediction of our model with the "true" value of the OOB samples (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., $\widehat{\boldsymbol{Y}_{OOB}} = \boldsymbol{Y}_{OOB}$)

Then we average our results for each bootsrapped sample! We can use the **OOB observations** to produce therefore a natural cross-validation (?!?) process: more on this later on!

Random Forest classifier

Through bagging, we therefore move from a (decision) tree to a forest of (decision) trees!

Still, bagging for itself cannot be enough…

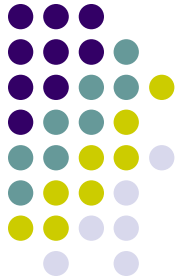
Bagging trees introduces a random component in to the tree building process that reduces the variance of a single tree's prediction and improves predictive performance

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!

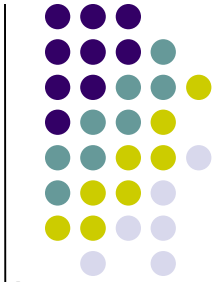
Let's inject more randomness into the tree-growing process. RF achieve this in two ways:

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,* i.e., we have a forest of (decision) trees; everytime m<p we have a random forest of (decision) trees!

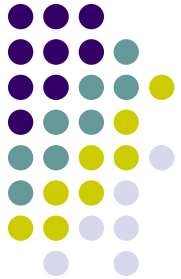
Random Forest classifier



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

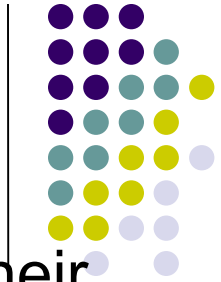


By fitting a tree (with no pruning) to each bootstrapped sample **and** by restricting the choice of each splitting variable to a random subset of predictors, we are sure that each boostrapped tree provides a truly different “perspective" on the prediction problem. All this, of course, minimizes the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

Into the Black Box



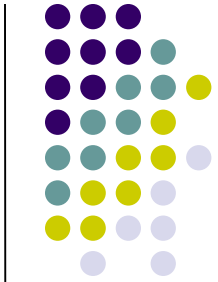
Often, ML models are considered “black boxes” due to their complex inner-workings. However, because of their complexity, they are typically more accurate for predicting nonlinear or rare phenomena

Unfortunately, more accuracy often comes at the expense of interpretability, and interpretability is crucial. It is not enough to identify a ML model that optimizes predictive performance; understanding and trusting model results is a hallmark of good (social and political) science!

Luckily, several advancements have been made to aid in interpreting ML models over the years.

Interpreting ML models is an emerging field that has become known as "*Interpretable Machine Learning*" (IML)

Into the Black Box



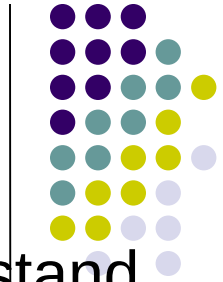
Approaches to model interpretability can be broadly categorized as providing *global or local explanations*

Global interpretations help us understand the inputs and their entire modeled relationship with the prediction target

Global interpretability in particular is about understanding how the model makes predictions, based on a holistic view of its features and how they influence the underlying model structure

It answers questions regarding which features are relatively influential as well as how these features influence the response variable

Into the Black Box

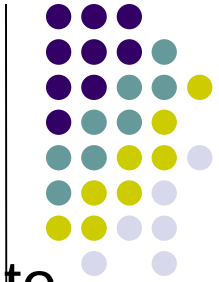


Local interpretations help us on the other hand to understand model predictions for a single row of data or a group of similar rows (i.e., the marginal impact of a feature across its values on our DV, holding all the other features at their actual value for each observation *i*)

In text analytics (given that we are dealing with huge DfM wherein the features per-se are not the main focus of our interest) we are mainly interested in global interpretations

However if you use ML for other aims, local interpretations become VERY important!

Into the Black Box



Indeed, a ML algorithm would allow you to discover quite easily non-linear relationships difficult to detect ex-ante…

For example, in a recent paper (Jordan et al. 2022), it has been suggested the following procedure:

1. Estimate a parametric model that tests theoretically grounded hypotheses

2. Use a machine learning approach on the same set of theoretical predictors to evaluate the robustness of the initial parametric tests

3. Adjust the initial parametric model to account for any nuances revealed in the machine learning approach