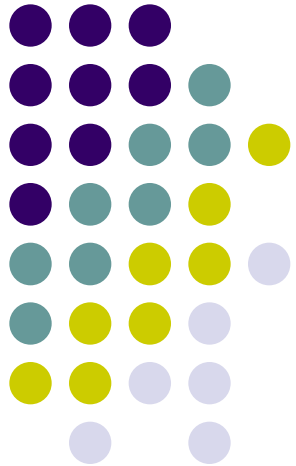
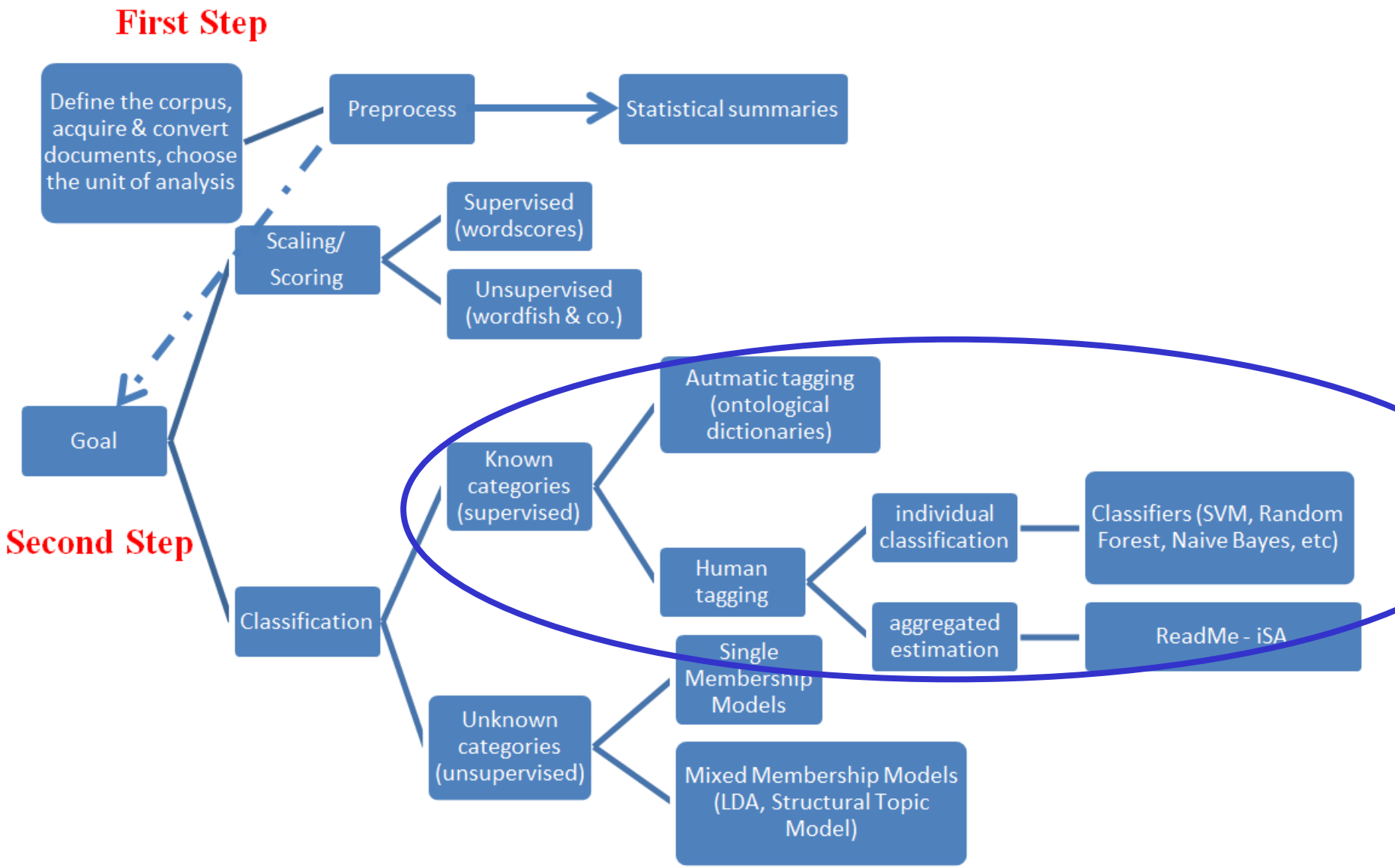
Applied Scaling & Classification Techniques in Political Science

Lecture 7

Supervised classification methods with human tagging: Machine Learning and Proportional Algorithms



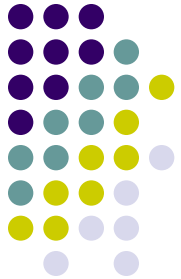
Our Course Map



References

- ✓ Grimmer, Justin, and Stewart, Brandon M. (2013). Text as Data: The Promise and Pitfalls of Automatic Content Analysis Methods for Political Texts. *Political Analysis*, 21(3): 267-297
- ✓ 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
- ✓ Curini, Luigi, and Robert Fahey (2020). Sentiment Analysis and Social Media. In Luigi Curini and Robert Franzese (eds.), *SAGE Handbook of Research Methods is Political Science & International Relations*, London, Sage, chapter 29

A classification of supervised classification methods



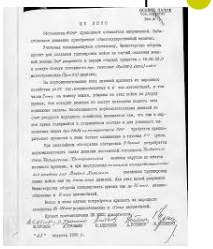
As already discussed in the previous lecture, supervised classification methods using human tagging can be either employed to:

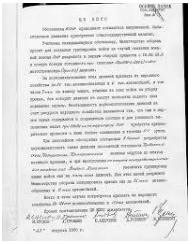
a) classify the **individual documents included in the test-set** into categories

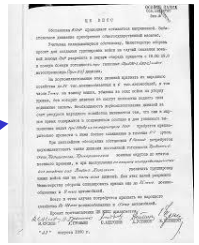
b) measure the **proportion of documents included in the test-set** in each category

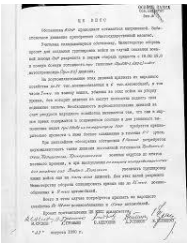
Let's start to discuss about **individual supervised classification methods** (i.e., **machine learning algorithms** applied to *text classification*)

Supervised learning: individual classification

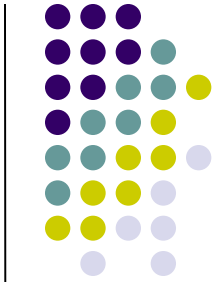








Individual methods



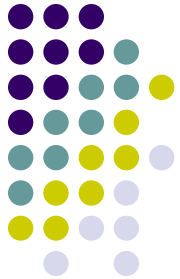
Several different possible machine learning algorithms out there

Here we will offer an intuitive introduction to the following algorithms:

- Naïve Bayes classifier
- Support Vector Machine
- Random Forest

Unfortunately we won't have time to discuss about Regularized regression, Gradient Boosting, Neural Networks and Deep Learning algorithms (among the others…)

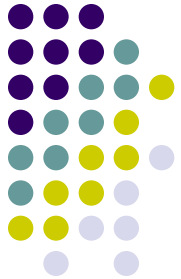
Naïve Bayes classifier



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

The model has a simple, but powerful, approach to learning the relationship between words and categories

Naïve Bayes classifier



The training set is used to learn about the distribution of words for documents from category *k*

This distribution is then used to classify each of the documents in the test set

The goal is to infer the probability that document *i* belongs to category *k* given word profile $\mathbf{W}_i$

Applying Bayes's rule: $p(C_k|\boldsymbol{W}_i) = p(C_k) * p(\boldsymbol{W}_i|C_k)$,

where we drop $p(\boldsymbol{W}_i)$ from the denominator since it is a constant across the different categories

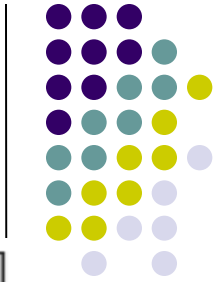
In plain English: $p(C_k|\boldsymbol{W}_i)$=posterior probability; $p(C_k)$=class prior probability; $p(\boldsymbol{W}_i|C_k)$= conditional probability or likelihood; $p(\boldsymbol{W}_i)$=evidence (the word profile we observe)

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



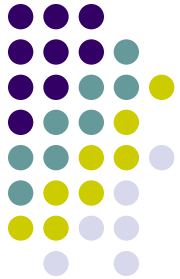
Fruit	Long	Sweet	Yellow	Total
Banana	400	350	450	500
Orange	0	150	300	300
Other	100	150	50	200
Total	**500**	**650**	**800**	**1000**

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

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

Naïve Bayes classifier

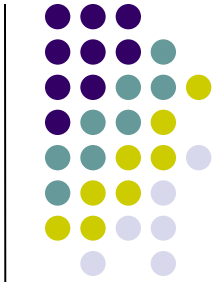


Now let's say we're given the variables of a piece of fruit and we need to predict the class

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

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

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

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

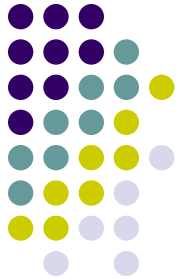
Naïve Bayes classifier

Note a possible problem here: 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 (as it happens with the feature “long” for the Orange)

What results is $p(\boldsymbol{W}_i|C_k)$ =0 for this individual variable and this zero 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 words and sparse DfM!!!!

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

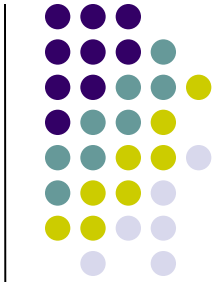
Typically, a value of one to two for the Laplace smoother is sufficient, but this is a tuning parameter to incorporate and optimize with cross validation.

Naïve Bayes classifier

Why Naïve?

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

Naïve Bayes classifier



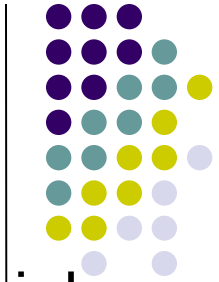
Why Naïve?

Variables, however, aren't always independent!

However, 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 independently must be 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



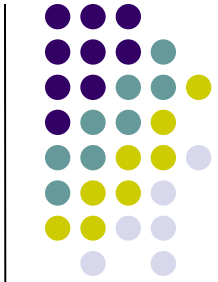
Naïve Bayes classifier algorithm is simple, fast but with high performance for text classification

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

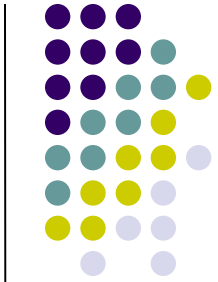


However, Naïve Bayes classifier also behaves differently on the other end of the spectrum, when provided with large amounts of training data

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

Support Vector Machine (SVM)



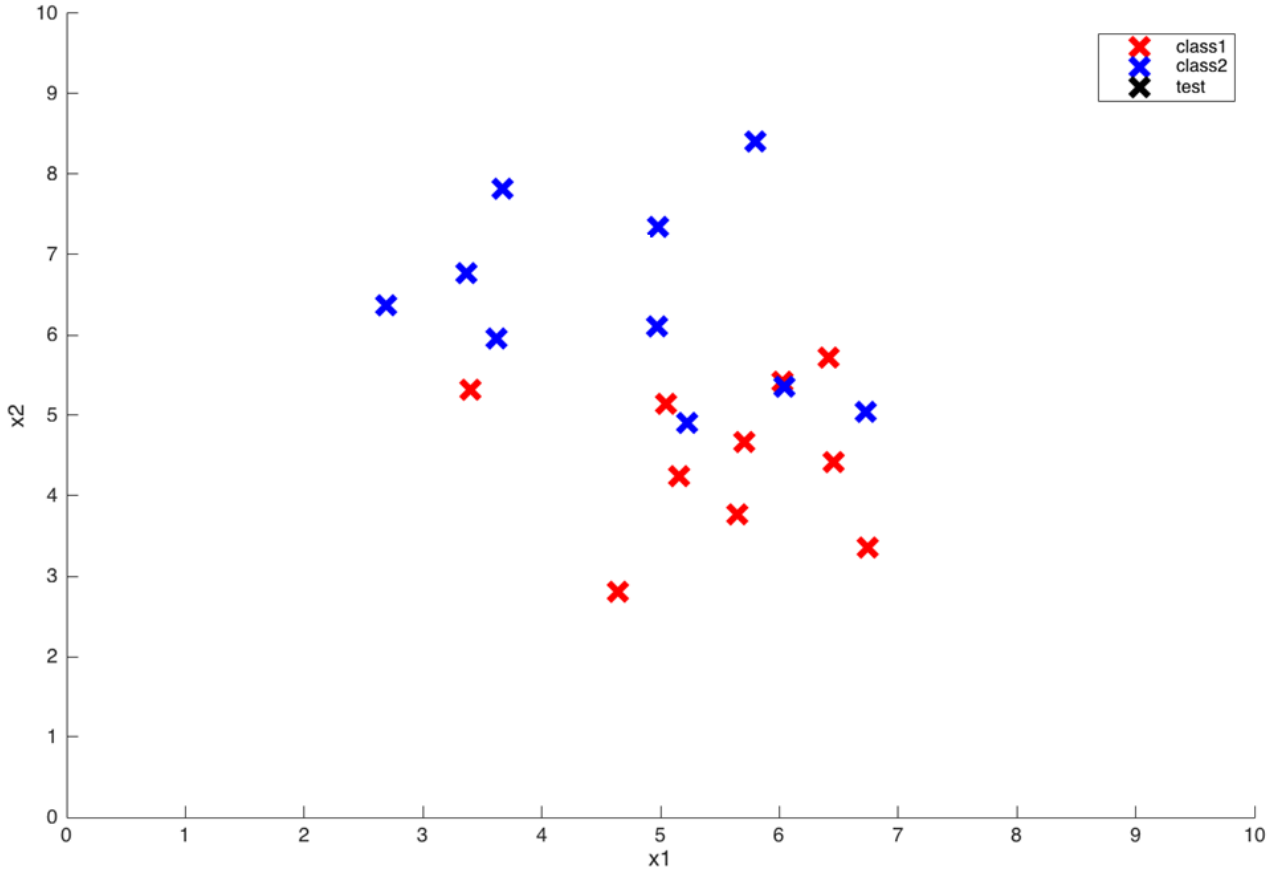
SVM is a generalization of **Nearest Neighbor** (NN) algorithm

NN is a very 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)

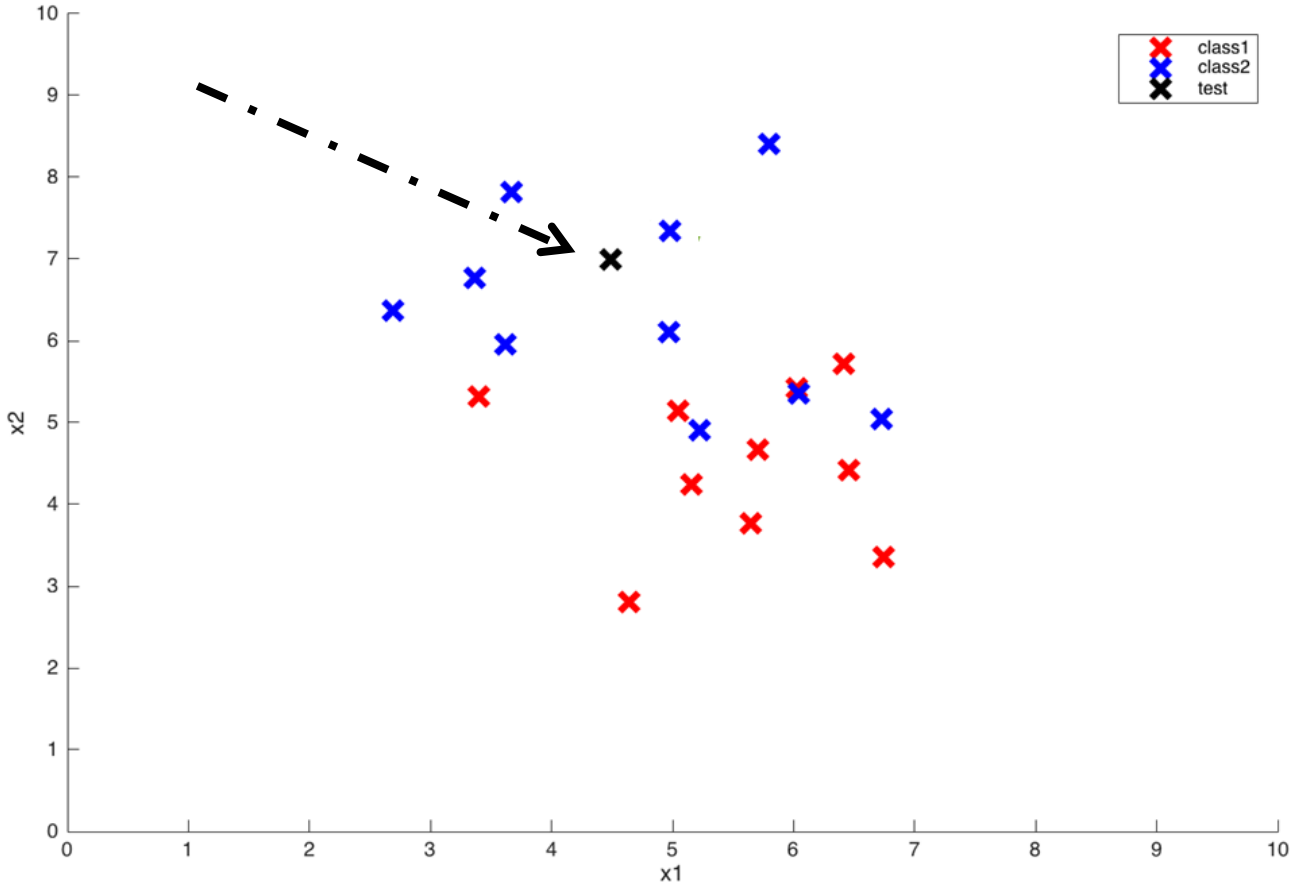
For example, in the figure below, we have two variables

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

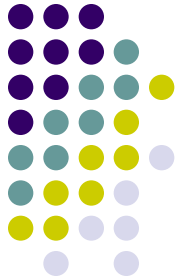


Support Vector Machine (SVM)

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)

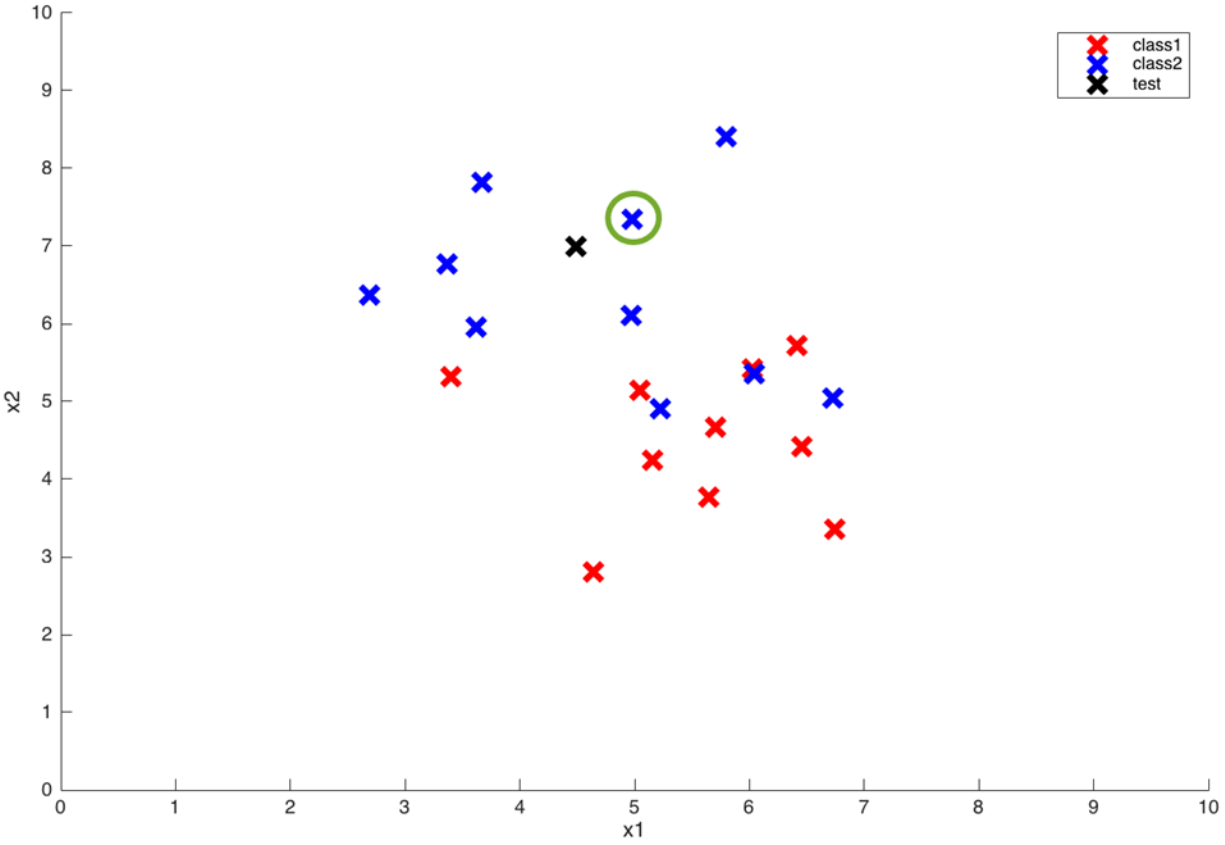


The NN algorithm finds the nearest training point to this test point (by measuring the distance of test point from **every** training point), and the class predicted of test point is the same as of nearest training point

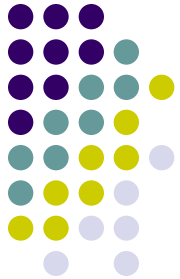
The lower the distance, the higher the **similarity** between two points

Support Vector Machine (SVM)

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



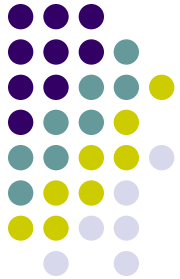
Support Vector Machine (SVM)



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)



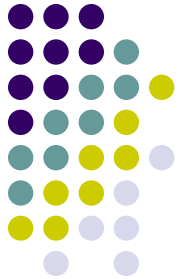
SVM remove each of these problems

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

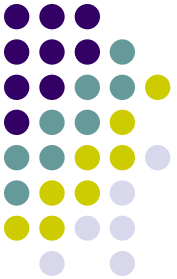
Support Vector Machine (SVM)



Moreover, once we have selected support vectors, we 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

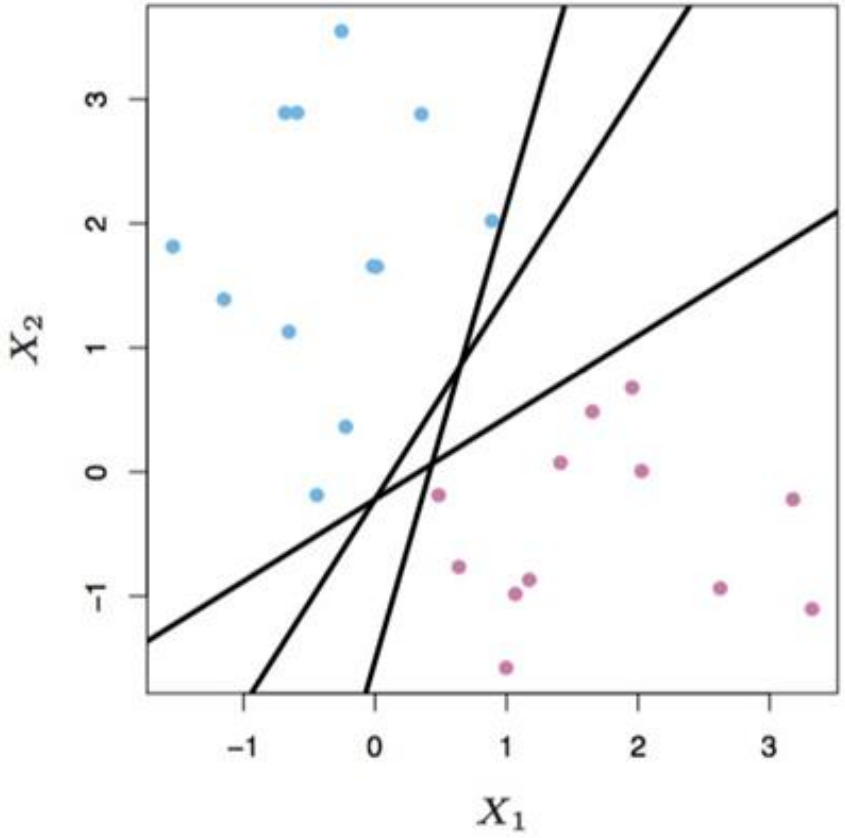
So unlike NN, we don't give importance to only a single training point (i.e., *document*, given that we are dealing with text classification), instead we give each support vector a separate importance

Support Vector Machine (SVM)



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

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



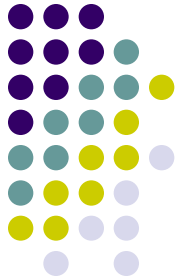
Support Vector Machine (SVM)

Harder to visualize in more than two dimensions. In this case you need an hyperplane

More formally, a 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)



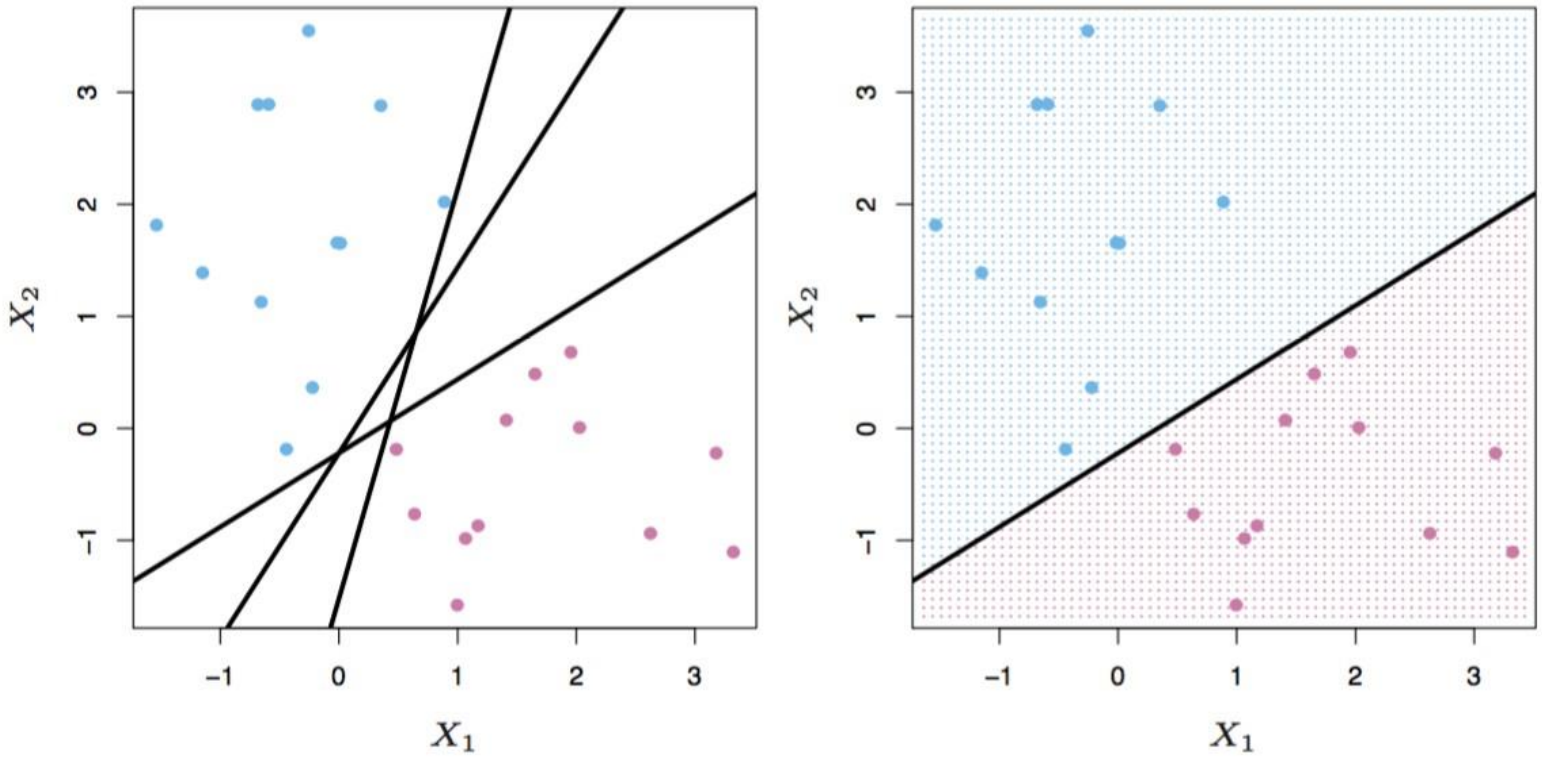
We want to find an hyperplane that **best separates two classes of points with the maximum margin** (i.e., we try to find that separating hyperplane from which distance of closest training points is maximum) thus producing the “cleanest" possible sorting of observations

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

Intuitively, this works well because the "maximum-margin" line allows for noise and is most tolerant to mistakes on either side

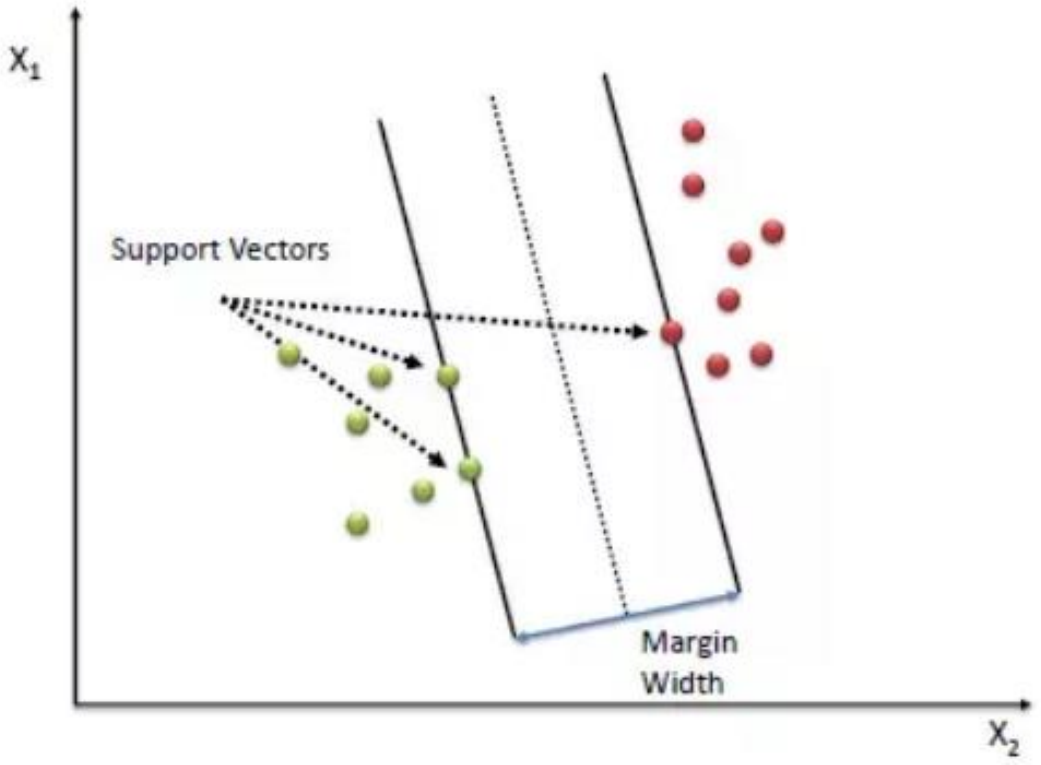
Support Vector Machine (SVM)

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

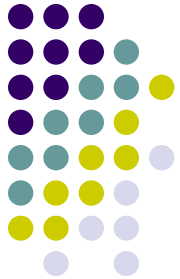


Support Vector Machine (SVM)

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



Support Vector Machine (SVM)

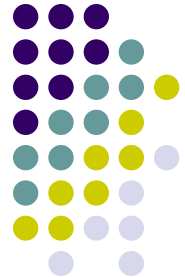


The data points that kind of "support" this hyperplane on either sides (i.e., the closest training points to the line) are called the **support vectors**

Support Vectors are simply the co-ordinates of individual observations

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

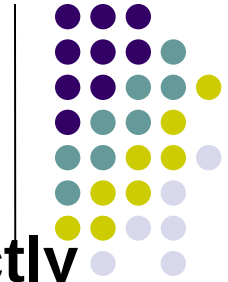
Support Vector Machine (SVM)



Fitting an SVM amounts therefore to draw a straight line through our data down the middle to separate it into two classes

An important implication of this goal is that only those **instances near the class boundary (the support vectors!) play a big role its definition**, while those that remain far away from the boundary have **little effect** on its location and direction

Support Vector Machine (SVM)



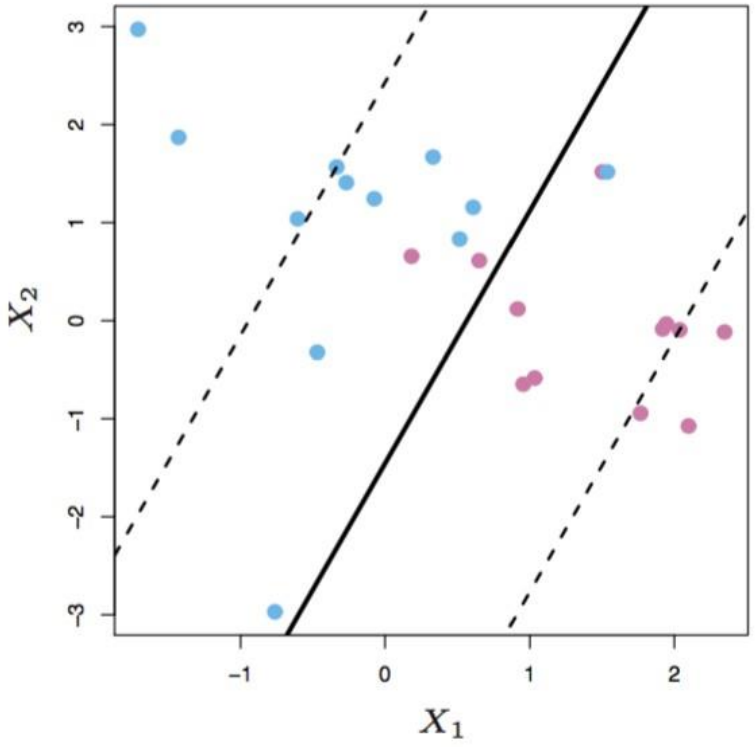
So far, we have assumed that a hyperplane can **perfectly separate instances across classes**

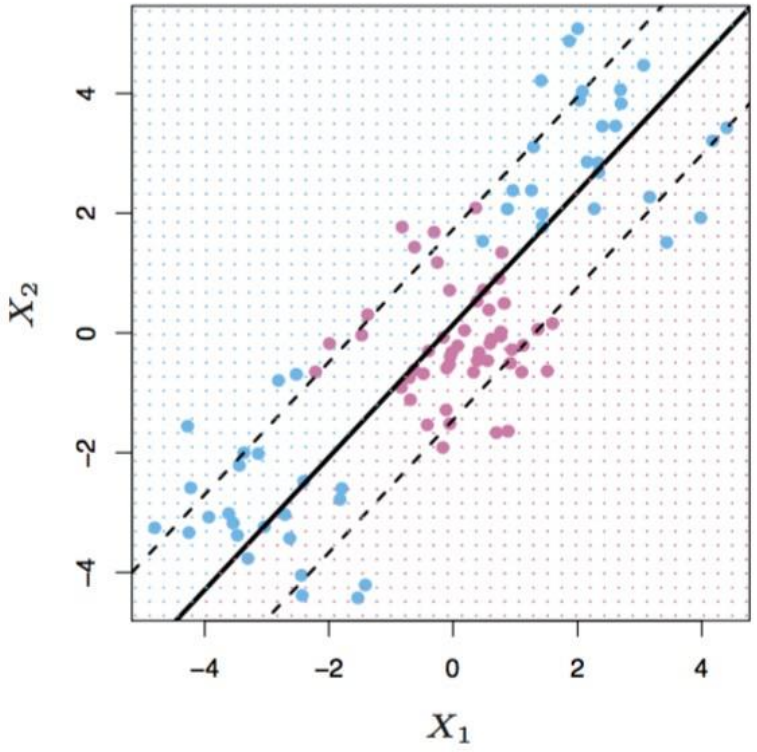
When this is not the case, we must relax the constraint imposed on the distances between points and the hyperplane, and allow for a certain amount of slack

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

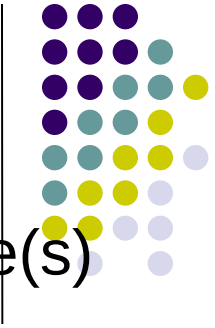
Support Vector Machine (SVM)

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





Support Vector Machine (SVM)



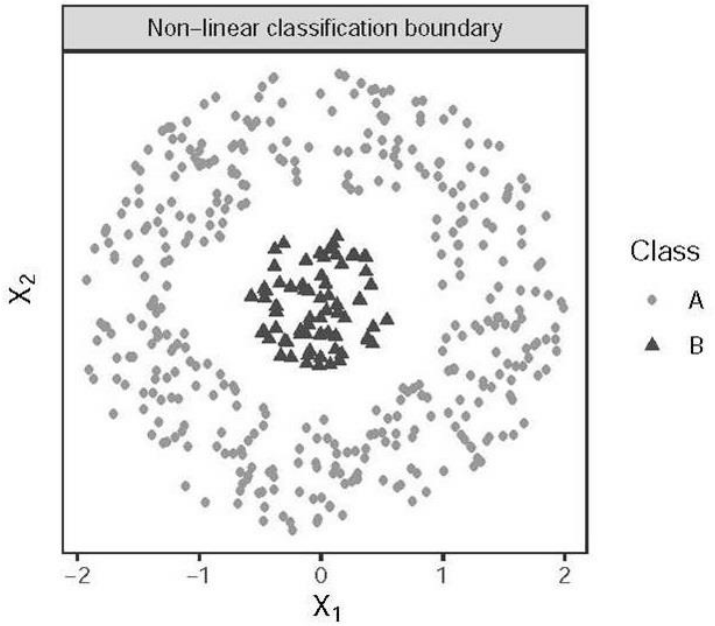
Specifically, the margin around the identified hyperplane(s) is larger for smaller values of C (ex. C = 0.01) and is smaller for larger values of C (ex. C = 1000)

Larger values of C thus result in greater focus of attention on the points located **very close** to the decision boundary, while smaller values involve data points **farther away**. It is these points that now become the support vectors

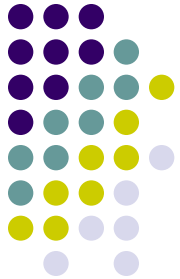
Support Vector Machine (SVM)

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

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)

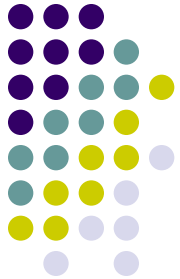


So what to do? **Move to non-linearity**!

We achieve this 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 transformed space of higher dimensions in which linear separability becomes (again) possible

Support Vector Machine (SVM)



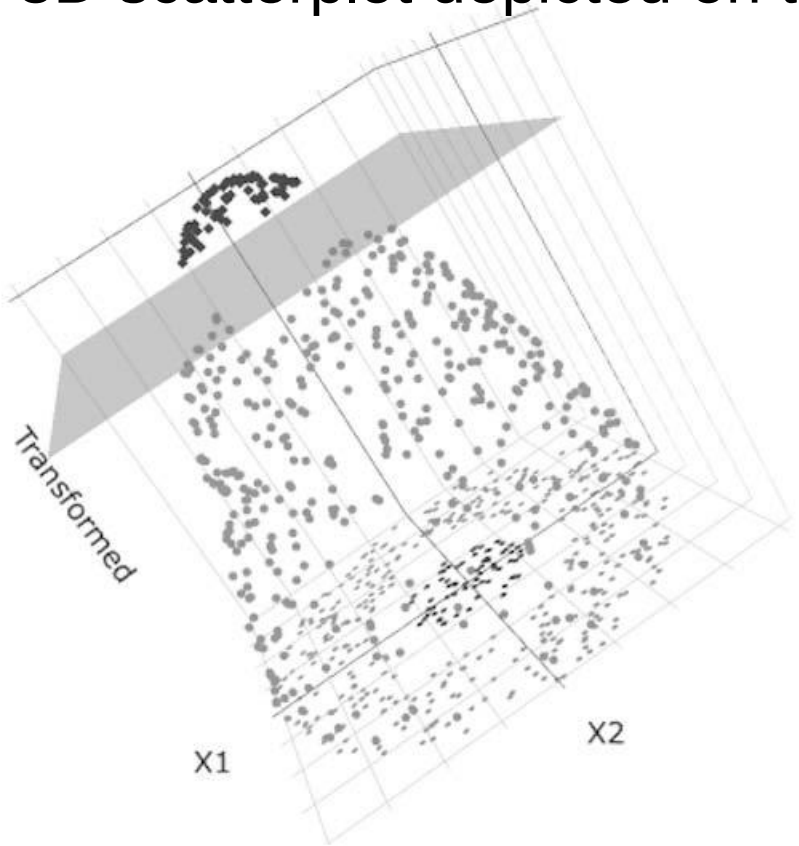
For example, if we can't draw a line in the space (x1, x2) then we may try adding a third dimension, (x1, x2, x1 * x2). If we project the "line" (in this 3D case, an hyperplane) in this higher dimension down to our original dimension, it looks like a curve

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)

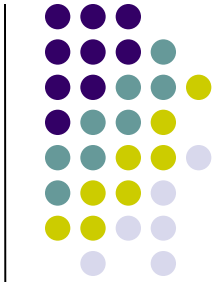
In this new, three-dimensional feature space, observations are now arrayed on a conical surface, with instances of class B (i.e., the triangles) rising to its apex

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 plane

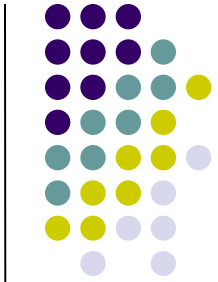
Support Vector Machine (SVM)



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

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

Random Forest classifier



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

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

The tree is created by splitting data up by variables and then counting to see how many are in each bucket after each split

Random Forest classifier

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

Intuitively, this goal is best achieved if the regions are defined by their degree of homogeneity with respect to the outcome of interest, so that region-specific predicted values are as close to the target as possible

Random Forest classifier

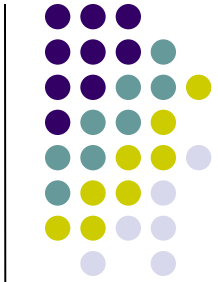
An example: Given only the gender and weight of a person, can we predict whether they are Japanese or American (our 2 classes/categories)?

Our 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



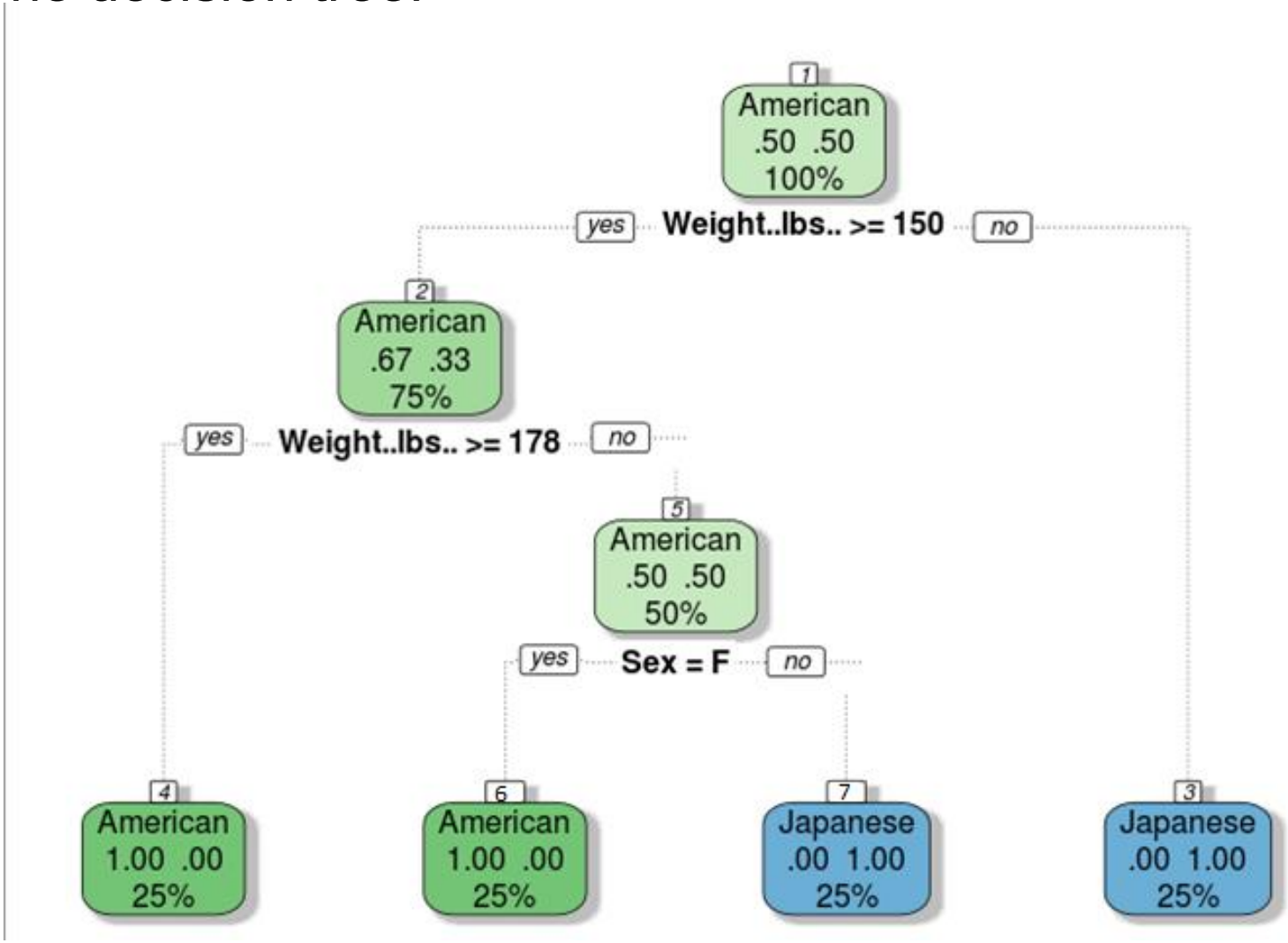
The key idea is that the procedure to create decision trees is **recursive**. For a set (**S**) of observations, the following algorithm is applied:

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*
2. If **S** is too large and it contains more than one class, 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**)

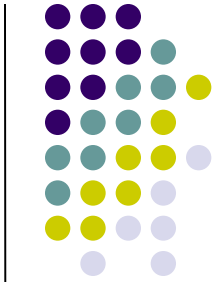
If you had to go to step 2, apply step 1 to each new subset. If your subsets need to go to step 2, apply step 1 to the sub-subsets, etc. When everything is split up appropriately (into buckets that are very small or entirely one class), you have a set of rules that look like a tree!

Random Forest classifier

The decision tree:



Random Forest classifier



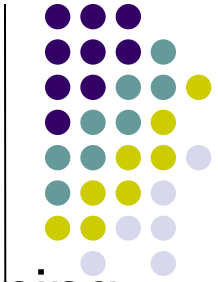
In this example the tree can **perfectly explain** the data

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

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

First, the algorithmic approach to building a tree leaves us with no means for assessing uncertainty in our estimates

Second, the sequential nature of the recursive splitting algorithm means that the structure of the tree is often highly sensitive to small changes in the observations included

So what to do (part 2)?

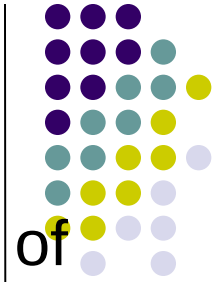
Random Forest classifier

Back to random forest (RF) now! RF is like a **bootstrapping algorithm with Decision tree model**!

Say, we have 1000 observation in the complete population with 10 variables

RF tries to build multiple Decision tree models **with different sample and different initial variables**

Random Forest classifier



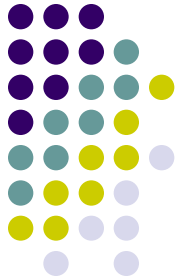
More formally, RF takes multiple simple random samples of the same data set (with replacement), of size equal to that of the original data set

It then fits a tree (with no pruning) to each bootstrapped sample by restricting the choice of each splitting variable to a random subset of predictors so that each boostrapped tree provides a truly different “perspective" on the prediction problem. All this, of course, minimizes the risk of overfitting!

Final prediction is a function of each prediction in each random sample. This final prediction can simply be **the average of each prediction**

Furthermore, measures of uncertainty can be readily produced!

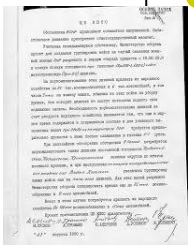
Measuring proportions

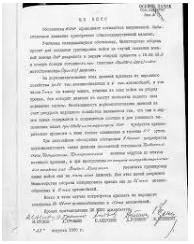


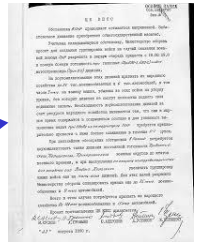
For many social science applications, only the proportion of documents in a category is needed, not the categories of each individual document

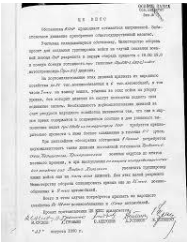
That is…

From here: individual classification



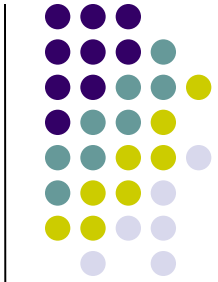






To here: proportional classification

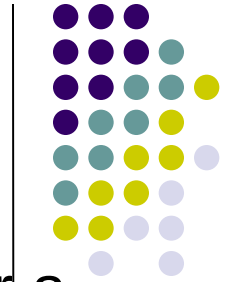
Measuring proportions



Shifting focus to **estimating proportions**, that is on $p(\boldsymbol{C})$, can lead to substantial improvements in accuracy

To understand how this approach actually works, we have to introduce a change in the DfM of a corpus as we discussed up to now

Measuring proportions



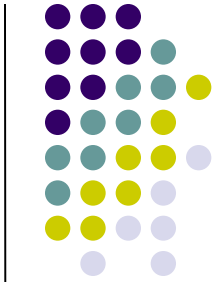
Now we include in the DfM an **indicator** (0/1) of whether a word occurred in a document, rather than the **counts** of the words

Post	Cat	Word: nuclear	Word: fear	Word: radiation	Word: pollution	Word: waste	Word: economic
post#1	like	1	0	0	0	0	1

Using this representation, let's define a multinomial probability distribution (p(**W**)) with respect to words over all possible documents in the corpus, where $p(W_1)$ in the example above is (1,0,0,0,0,1) and is called a "word profile"

p(**W**) is therefore simply the proportion of documents in the corpus observed with each pattern of word profiles

Measuring proportions



The data-generating process for the documents can then be written as:

$$p(\boldsymbol{W}) = p(\boldsymbol{W}|\boldsymbol{C}) * p(\boldsymbol{C}),$$

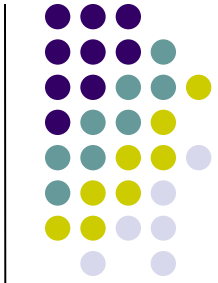
where:

$p(\boldsymbol{W})$ is the proportion of documents in the corpus observed with a given pattern of word profiles

$p(\boldsymbol{W}|\boldsymbol{C})$ is the proportion of documents in the corpus observed with a given pattern of word profiles **conditional** on categories

$p(\boldsymbol{C})$ is the proportion of documents in each class in the corpus - **the quantity of our interest**

Measuring proportions



$$p(\boldsymbol{W}) = p(\boldsymbol{W}|\boldsymbol{C}) * p(\boldsymbol{C})$$

$p(\boldsymbol{W})$ is the distribution of the features in the whole set (train + test). We have an **accurate estimation** here!

And what about $p(\boldsymbol{W}|\boldsymbol{C})$? It requires labeled documents - which are unavailable for the test set!

But if we assume that the conditional distributions **are identical in the training and test sets**, then we can estimate $p(\boldsymbol{W}|\boldsymbol{C})$ directly from the training-set

We have therefore also here an **accurate estimation** (as long as the coders did a good job!)

Estimating $p(\boldsymbol{C})$ is now therefore easy by solving the equation via standard regression algebra!

Measuring proportions

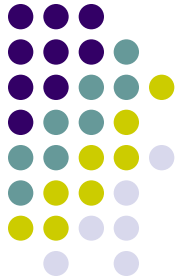
$$p(\boldsymbol{W}) = p(\boldsymbol{W}|\boldsymbol{C}) * p(\boldsymbol{C})$$

If we think of $p(\boldsymbol{C})$ as the unknown "regression coefficients" (the $\boldsymbol{\beta}$), $p(\boldsymbol{W}|\boldsymbol{C})$ as the "explanatory variables" matrix *X*, and $p(\boldsymbol{W})$ as the "dependent variable" *Y*, then this equation becomes the usual: $\mathrm{Y} = X\beta$ (with no error term)

From here, we can move to estimate $p(\boldsymbol{C})$ (via standard constrained least squares to ensure that elements of $p(\boldsymbol{C})$ are each in [0,1] and collectively sum to 1):

$$p(\boldsymbol{C}) = p(\boldsymbol{W}) * p(\boldsymbol{W}|\boldsymbol{C})^{-1}$$

Measuring proportions



In other words: instead of modeling the relation between features (i.e., words) and classes for **each single training document**, this approach uses a regression model that associates feature distribution ($p(\boldsymbol{W})$) with class distributions $p(\boldsymbol{W}|\boldsymbol{C})$ in the **entire training collection**

A key point is that this calculation does not require classifying individual documents into categories and then aggregating; it estimates the **aggregate proportions** $p(\boldsymbol{C})$ for target collections of unlabeled documents **directly**!

Measuring proportions

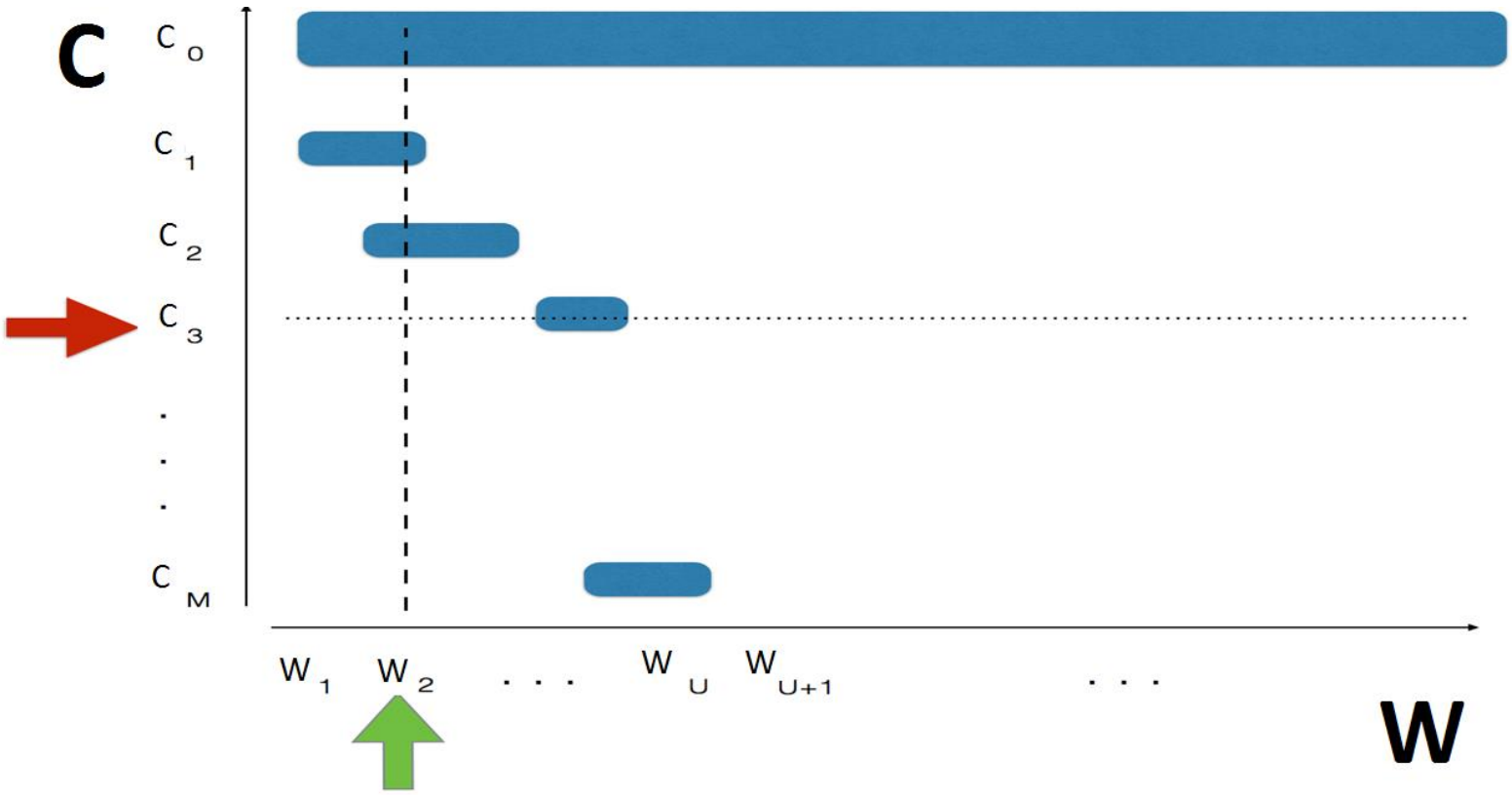
Focusing on $p(\boldsymbol{W}|\boldsymbol{C})$ rather than $p(\boldsymbol{C}|\boldsymbol{W})$ as done in the machine learning approach (remember!), has two main advantages

Theoretically: $p(\boldsymbol{W}|\boldsymbol{C})$ means: «given a post that is associated to a given content, which are the sequence of stems effectively employed to express that specific content»?

This makes a lot of sense: you do not start writing and only **afterwards** discover your sentiment toward for example a party. You start with a view, with a “category” in your mind (good, bad, support or not), and then set it out in words

Measuring proportions

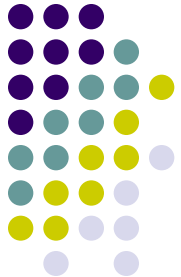
Empirically: the existence of a category C_k extremely frequent in a training-set can negatively affect $p(\boldsymbol{C}|\boldsymbol{W})$ but not $p(\boldsymbol{W}|\boldsymbol{C})$



The intuition

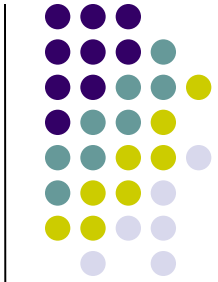
It is easier to look at the shape of the haystack rather than trying to find a needle in it!

The intuition



Moreover…choosing a classifier by maximizing the percent correctly classified at the individual level can sometimes drastically increase the **bias of aggregate quantities**

The intuition

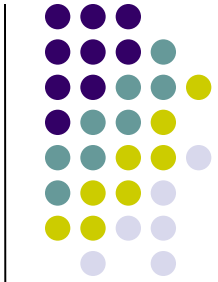


For example, the decision rule "**war never occurs**" accurately classifies **each country-year dyad** into war/no war categories with over 99% *accuracy*, but is obviously misleading for social science research purposes (remember the example about cats and dogs)!

Saying differently: a method that classifies 60% of documents correctly into 1 of 8 categories might be judged successful and useful for classification

However, because the individual category percentages still might be off by as much as 40 percentage points, the same classifier may be useless for some social science purposes (if individual-level errors do not cancel each other)

Measuring proportions



No statistical property must be satisfied by the training set for this approach to work properly: the training set is **not a representative sample** of the distribution of opinions in the population of texts to be analyzed!

However, the language used in the training-set to express some given concept is **assumed** to be the same as in the whole population of posts, i.e. social media users use the same language

✓ Is it a **reasonable assumption**?

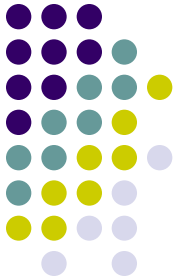
Measuring proportions

After all, in the **Oxford Dictionary** (English) you have **650k terms**

In reality, for any given topic, in the everyday language there is a tendency to use at the maximum between **200 and 500 *stems***

This is what **makes possible** the statistical analysis

R pakcages to install



install.packages("e1071", repos='http://cran.us.r-project.org')

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

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

install.packages('caret', dependencies = TRUE)

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

install.packages ("reshape2", ='http://cran.us.r-project.org')

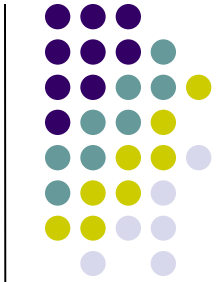
install.packages("stringr", repos='http://cran.us.r-project.org')

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

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

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

R pakcages to install



devtools::install_github("blogsvoices/iSAX")

If you have problems in installing iSAX…

For Windows ppl out there: install Java 64bit (if you have a 64bit machine!)

For Mac ppl out there: apparently isax loads Java developer kit version, instead of normal java. So install two javas in your Mac: normal java, and java developer kit