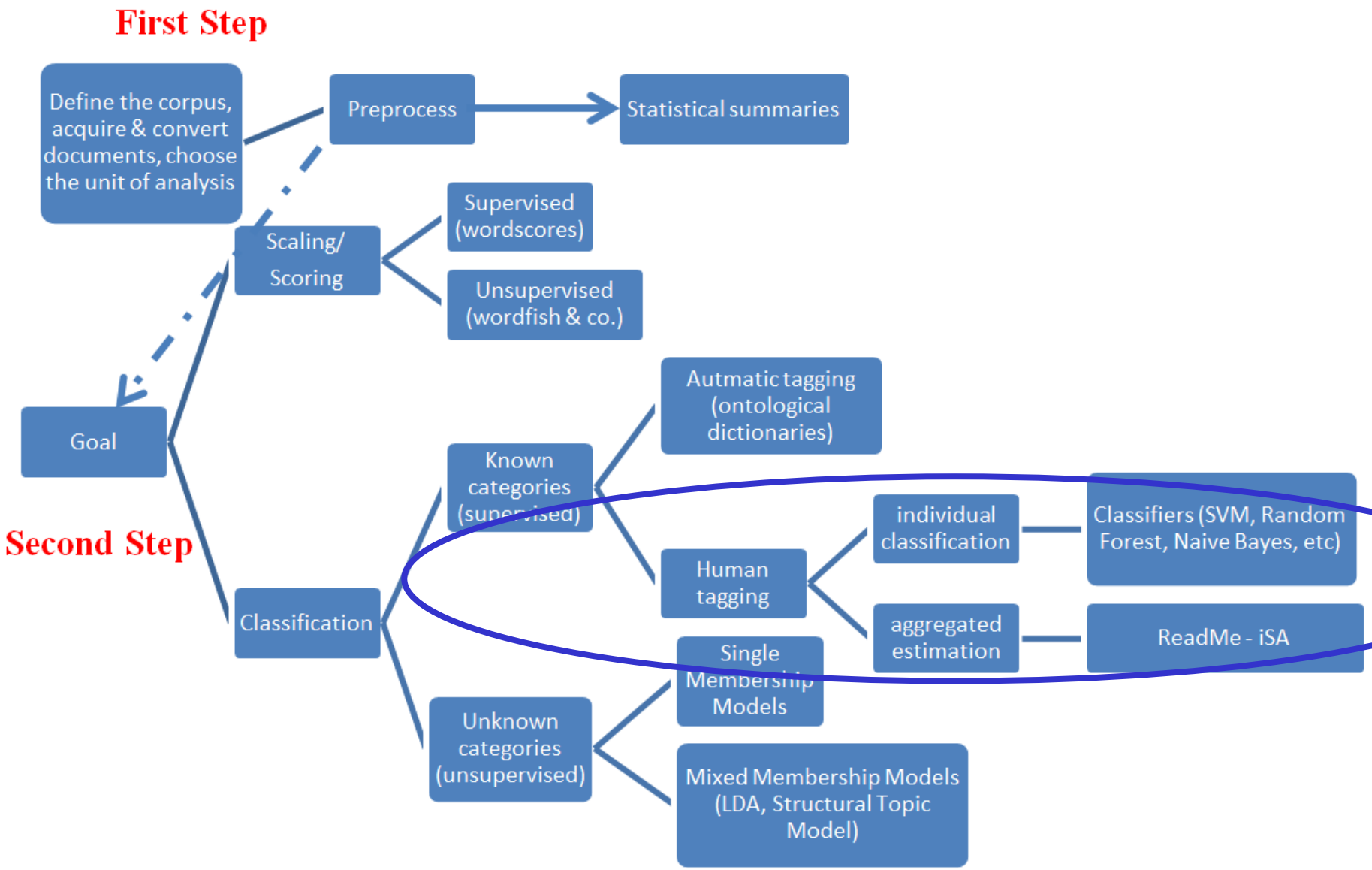
Big Data Analytics

Lecture 3/A

Supervised classification methods with human tagging

UNIVERSITÄT LUZERN

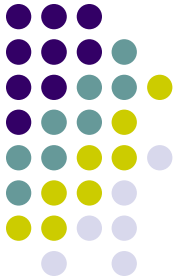
Our Course Map



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
- ✓ 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
- ✓ 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

Supervised Learning (classification) Methods



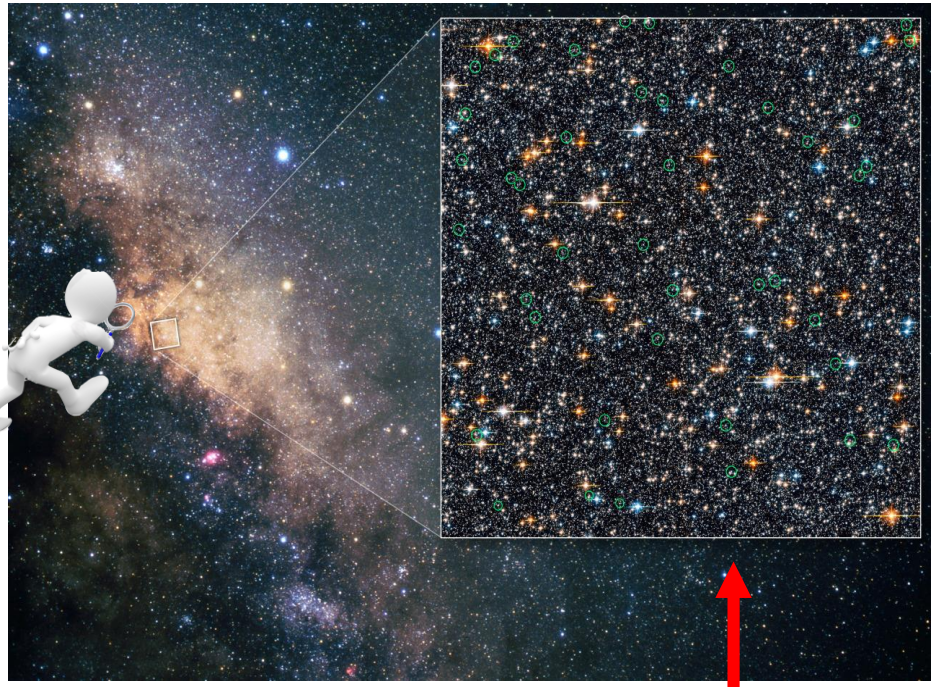
The idea of supervised learning is simple: human coders categorize a set of documents (the "**training-set**" or "**labelled-set**") by hand into a set of pre-defined categories (such as positive, negative, neutral for example)

The algorithm "learns" how to sort the documents into categories using the **training set and words**

Then, it classifies the remaining set of document not classified by hand (the "**test-set**" or "**unlabelled set**") using the characteristics (i.e., words) of the unread documents to place them into the categories

A four-step procedure

1. Data preparation: separating the training set from the test set in the corpus

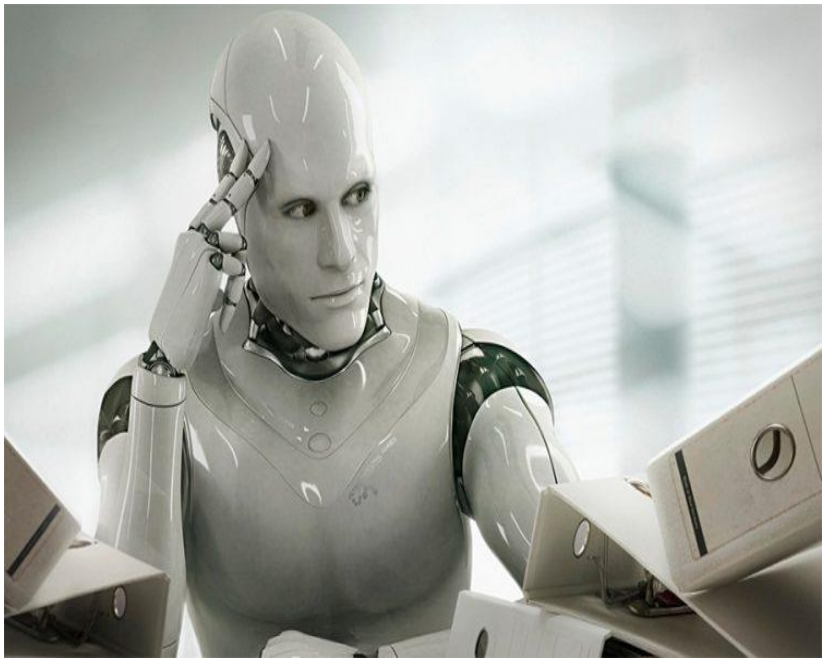


the training set

2. Human classification
of the training set on a base of a list of pre-defined categories

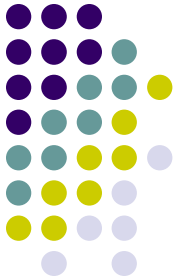
A four-steps procedure

3. Cogito ergo sum! The algorithm learns from the human classification done in the training set



4. Let's classify! The well-educated algorithm is now ready to classify all the texts in the test-set

Supervised Learning (classification) Methods



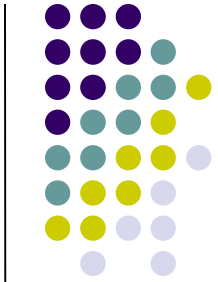
Despite the fact that the methods to do supervised classification are diverse, they share a common structure that usefully unifies the methods

Suppose there are N_{train} documents (i=1,…, N_{train}) in our training set and that we have pre-defined K categories (k=1,…,K) for our classification, such as positive, negative, neutral in the case of a sentiment analysis

Each document i's category is then represented by $Y_i \in (C_1, \ldots, C_K)$ and the entire training-set is represented as $\mathbf{Y}_{train}=(Y_1,\ldots,Y_{Ntrain})$

$\mathbf{W}_{train}$ is the term-document matrix for N_{train}

Supervised Learning (classification) Methods



Each supervised learning algorithm assumes that there is some (unobserved) function that describes the (true) relationship between the words and the labels in the training-set:

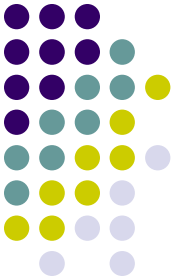
$$\boldsymbol{Y}_{train} = \mathrm{f}(\boldsymbol{W}_{train})$$

Each algorithm attempts to learn this relationship by estimating the "true" function *f* with $\hat{f}$ (the **classification function**)

$\hat{f}$ is then used to infer properties of the test set (the unlabeled set), $\widehat{\boldsymbol{Y}_{test}}$ using the test set's words $\boldsymbol{W}_{test}$:

$$\widehat{\boldsymbol{Y}_{test}} = \hat{f}(\boldsymbol{W}_{test})$$

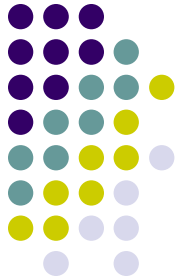
Supervised Learning (classification) Methods



Summing up…

Supervised learning models **share the same goal**: learn the potentially complicated relationships that relate (combinations of) features x to the outcome of interest y in general, using information available in the set of observations for which the pair (x; y) is fully observed (i.e., in the training-set)

Supervised Learning vs. Dictionary methods

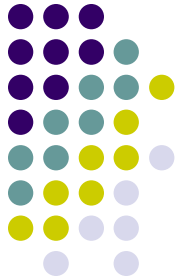


Supervised learning can be therefore conceptualized as a **generalization of dictionary methods**, where features associated with each categories (and their relative weight) are learned from the data **via human intervention**

The feature space is thus likely to be both larger and more comprehensive than that used in a dictionary

The end result is that much more information drives the subsequent classification of text

Supervised Learning vs. Dictionary methods



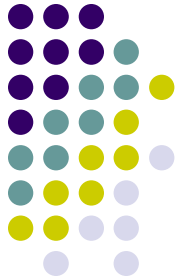
Moreover, compared to dictionary methods:

Supervised learning is **necessarily domain specific** and therefore avoids the problems of applying dictionaries outside of their intended area of use

Second, human involvement is crucial to understand the correct meaning of a text (double meaning sentences, specific jargons, neologisms, irony)

Finally, supervised learning methods are much easier to validate, with clear statistics that summarize model performance (as we will discuss)

Supervised Learning (classification) Methods

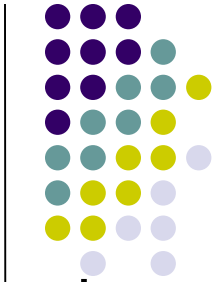


We have two broad classes of Supervised Learning Methods:

a) those who aim to classify the **individual documents** into categories

b) those who aim to measure the **proportion of documents** in each category

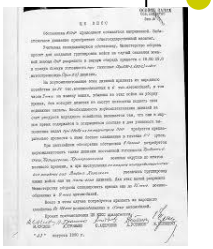
Applying a supervised learning model

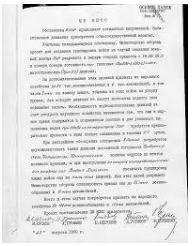


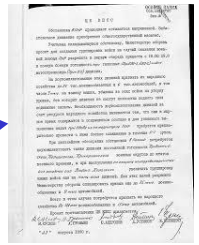
Individual approaches estimate the category of each text in a new corpus of texts (test-set) aiming at *minimizing the probability of error in individual class assignment*…

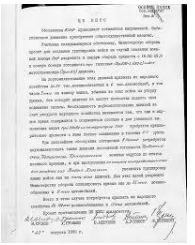
…while **aggregated approaches** estimate the class proportions into the new corpus of texts (test-set), rather than assembling the results of several individual classifications, aiming at *minimizing the error between the estimated proportions and true proportions*

Supervised learning: individual classification



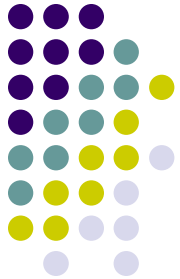






Supervised learning: proportional classification

Applying a supervised learning model



There is an important **theoretical (and statistical) difference** between individual and proportional classification:

- ✓ for some social science application, only the proportion of documents in a category is needed, not the categories of each individual document
- ✓ The opposite happens in some other application

Once again, there is no “a best method” out there. It depends on your research topic (remember the 4 rules!)

In our course, we will discuss about individual classification approaches. If you are interest about proportional classification algorithm, send me an email

Machine learning

Machine learning is defined as the “field of study that gives computers the ability to learn without being explicitly programmed” (Samuel 1959)

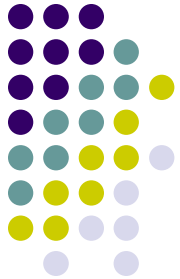
In this context “learning” can be viewed as the use of statistical techniques to enable computer systems to progressively improve their performance on a specific task from data without being explicitly programmed (Goldberg and Holland 1988)

Machine

To be able to le ecome better at it, a machine

- ✓ …be provide tion (inputs) and the desi learn a general rule to the outputs

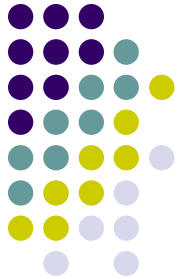
Machine learning



In our case, our aim is to do *text classification*

Therefore, **machine learning algorithms** (when dealing with text classification methods) refer to those techniques that learn how to map a set of inputs (e.g., features within documents) to a predicted class as the output in a pre-coded *training set* before classifying the data in the *test set*

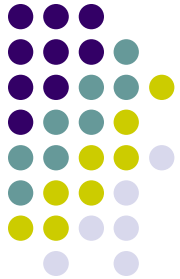
Beware of overfitting!



However…it is typically **easy** to learn even complicated relationships *in-sample* that is, relationships that are conditional on the training set

The goal, **however**, is to learn relationships for which the **expected generalization error** (i.e. the error that can be expected to ensue when learned relationships are evaluated *out-of-sample*, on a test set of observations not involved in the learning process) is low

Beware of overfitting!

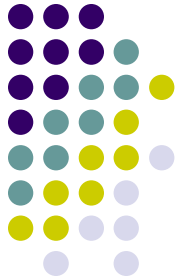


In fact, while it is always possible to arbitrarily reduce training error (i.e. error as computed using the training sample) by making models arbitrarily complex…

…such **complexity** typically results in high expected generalization error, as models start to overt their training data (i.e. they start to pick up on idiosyncratic relationships that conditional on the set of observations used to train the models)…

…that is, a supervised learning algorithm begins to **overfit the data**!

Beware of overfitting!

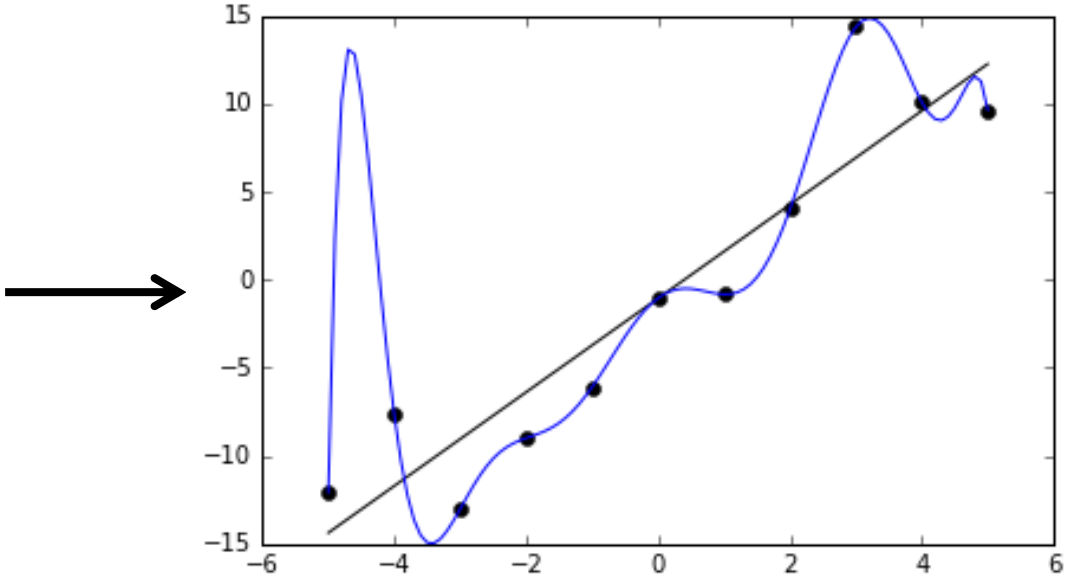


Overfitting is the production of an analysis that corresponds too closely to a particular set of data, and may therefore **fails** to fit additional data or predict future observations reliably

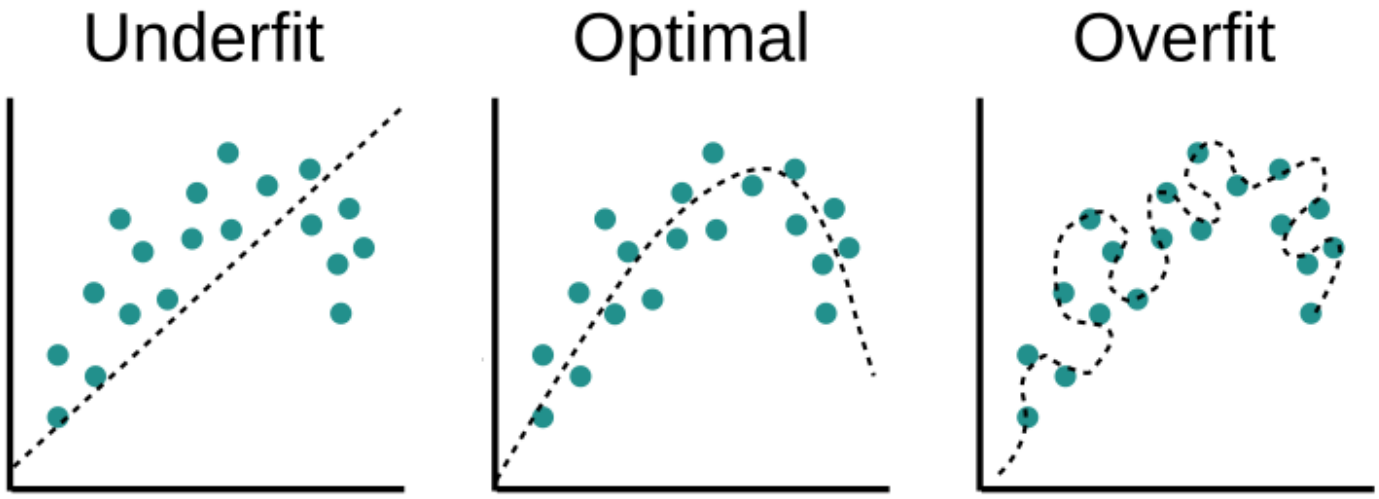
Overfitting usually arises when a *very complicated model* faithfully reflects aspects of the design data to the extent that idiosyncrasies of that data, rather than merely of the distribution from which the data arose, are included in the model

Beware of overfitting!

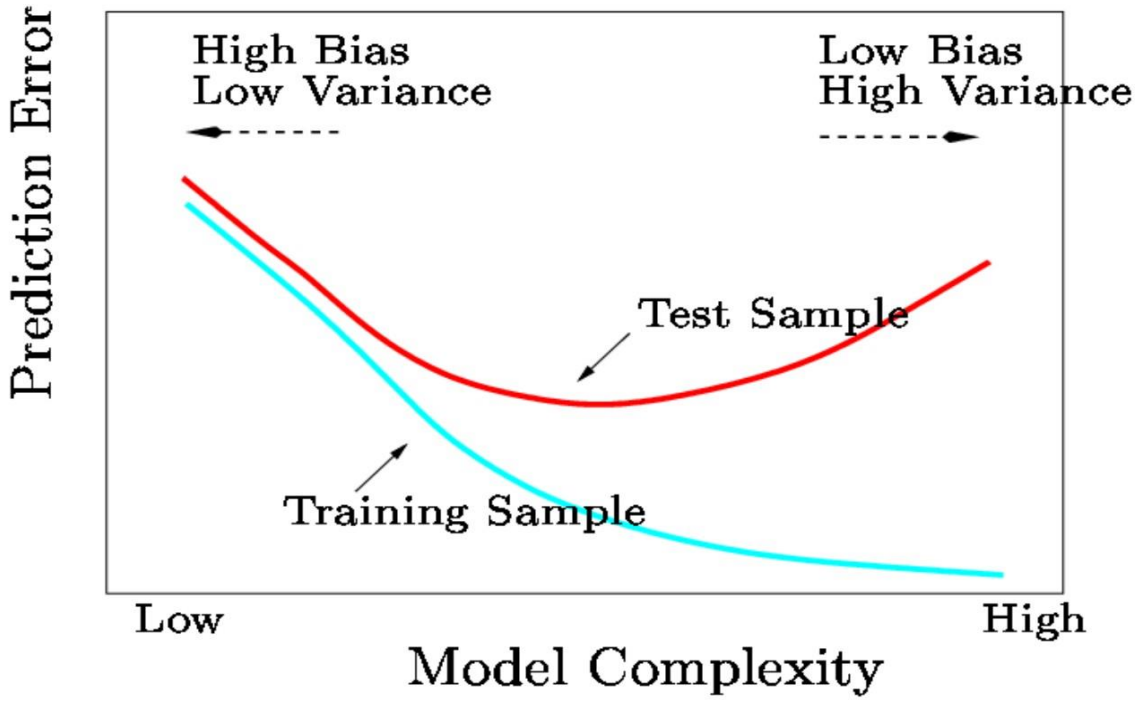
Although the polynomial function (the blue line) is a perfect fit, the linear function can be expected to generalize better beyond the fitted data!



Beware of overfitting!



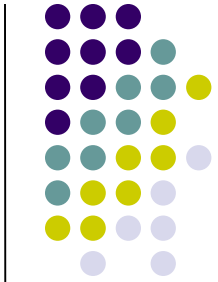
Beware of overfitting!



The Bias-Variance trade-off – *bias* refers to underfitting risk; *variance* to overfitting risks

- ✓ Model is too complex, describes noise rather than signal
- ✓ Focus on features that perform well in training-set data but may not generalize
- ✓ In-sample performance better than out-of-sample performance

Individual methods



Several different possible machine learning algorithms are available out there

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

- Naïve Bayes classifier
- Support Vector Machine
- Random Forest
- Gradient Boosting

Unfortunately we have no time to discuss about Neural Networks and Deep Learning algorithms (among the others…)

Naïve Bayes classifier

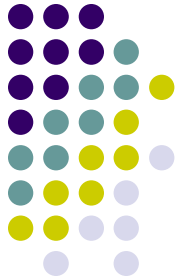
Naïve Bayes classifier: the algorithm allows us to predict a class, given a set of features using (Bayes) probability theorem

But what do we mean by Bayes probability theorem?

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

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

Naïve Bayes classifier



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

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

In plain English:

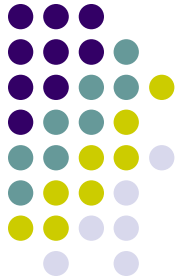
$p(C_k)$=prior probability of the outcome

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

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

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

Naïve Bayes classifier



In other words, the Bayesian formula is simply:

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

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

Naïve Bayes classifier

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

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

Naïve Bayes classifier

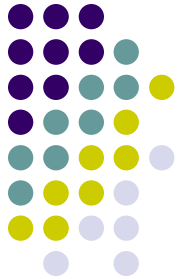
Fruit	Long	Sweet	Yellow	Total
Banana	400	350	450	500
Orange	0	150	300	300
Other	100	150	50	200
Total	**500**	**650**	**800**	**1000**

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

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

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

Naïve Bayes classifier

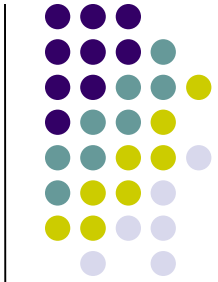


Now let's say we're given the variables of a piece of fruit and we need to predict the class (*out-of-sample prediction*!)

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

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

Naïve Bayes classifier



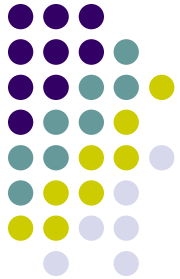
p(Banana|Long, Sweet, Yellow)=p(Long|Banana=0.8)*p(Sweet|Banana=0.7)*p(Yellow|Banana=0.9)*p(Banana=0.5)=**0.252**

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

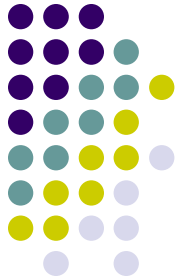


Note a possible problem here: given that naïve Bayes uses the product of variable probabilities conditioned on each class, we run into a serious problem when new data includes a variable value that **never occurs** for one or more levels of a response class (as it happens with the feature “long” for the Orange)

What results is $p(Long|Orange)$ =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 a sparse DfM (as it is usually the case)!

Naïve Bayes classifier

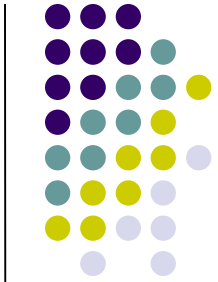


A solution to this problem involves using the ***Laplace smoother***

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

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

Naïve Bayes classifier



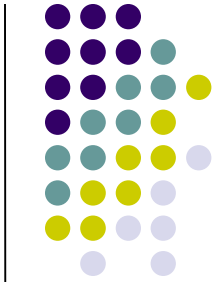
Why Naïve?

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

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

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

Naïve Bayes classifier



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 indipendently is wrong given that the use of words is highly correlated in any data set. However, Naïve Bayes classifier can still be useful (remember the First Principle of text-analysis!)

Naïve Bayes classifier

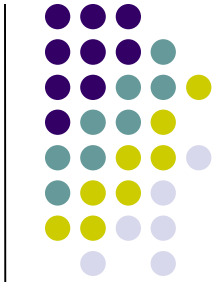
Naïve Bayes classifier algorithm has a simple, but powerful (and fast!), approach to learning the relationship between words and categories

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

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

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

Naïve Bayes classifier

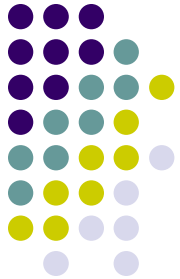


However, Naïve Bayes classifier also behaves differently on the other end of the spectrum, when provided with large amounts of training data

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

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

Support Vector Machine (SVM)



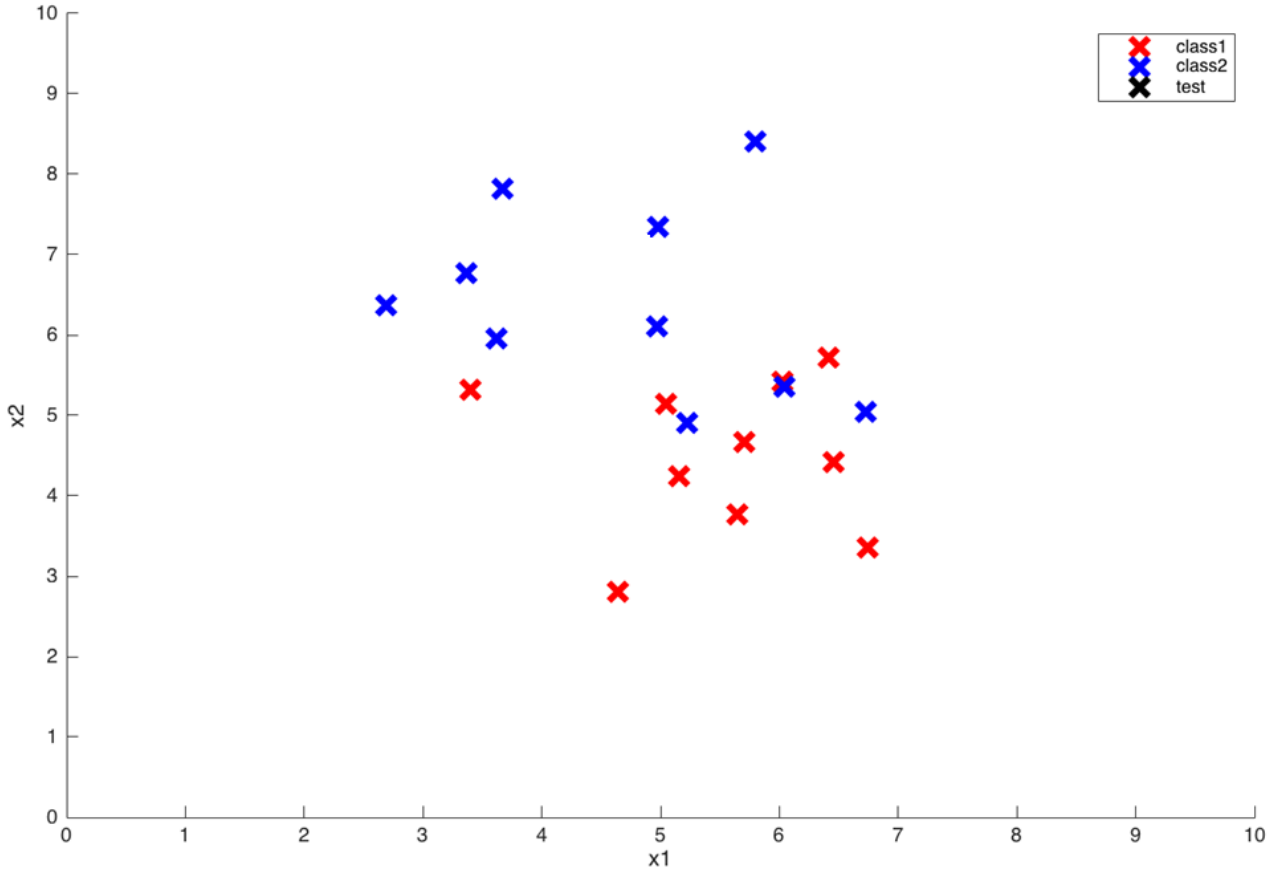
SVM is a generalization of **Nearest Neighbor** (NN) algorithm

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}

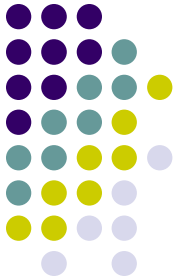
Support Vector Machine (SVM)

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

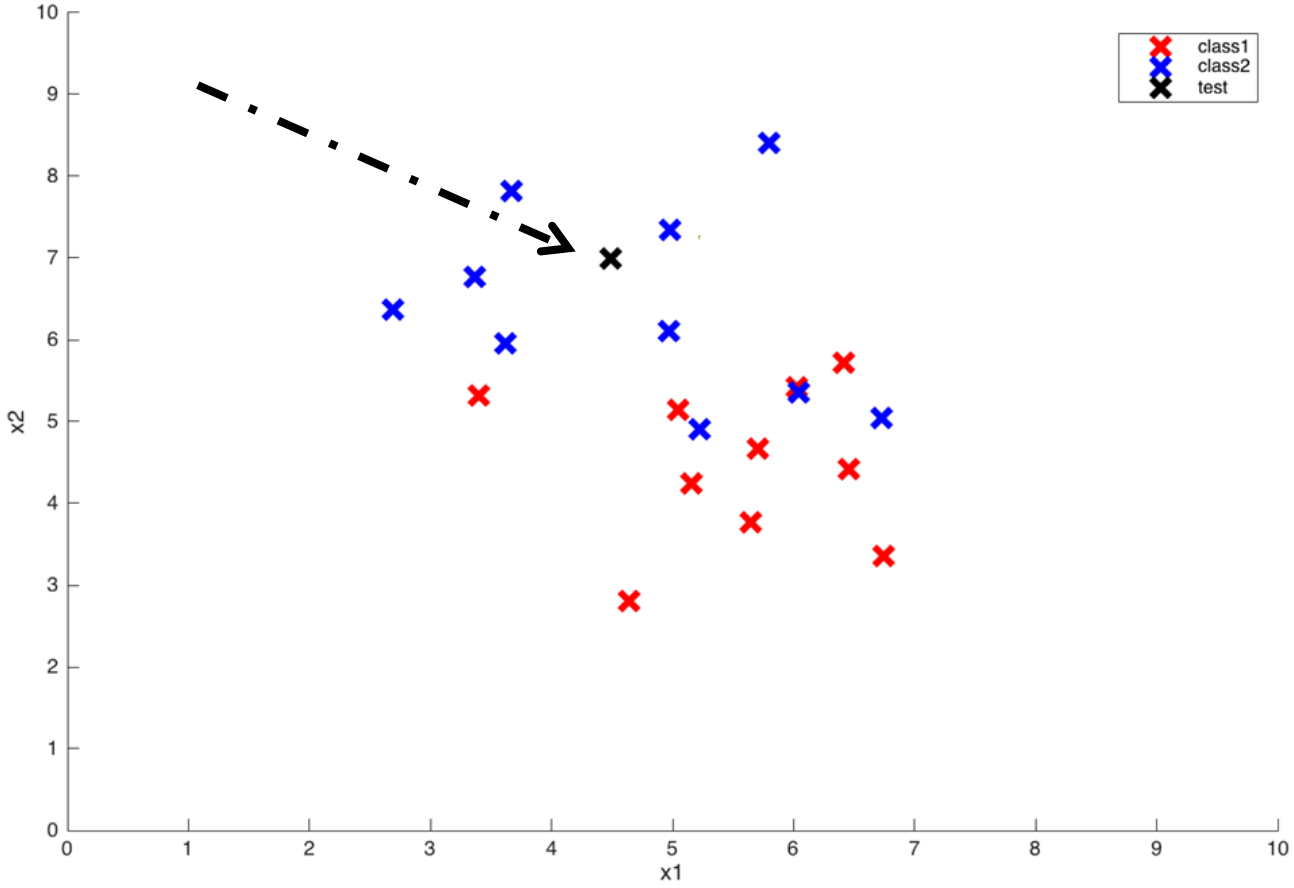
Xs: each document can be either label 1 (blue points), or label −1 (red points)



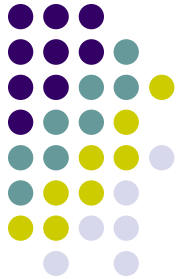
Support Vector Machine (SVM)



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)

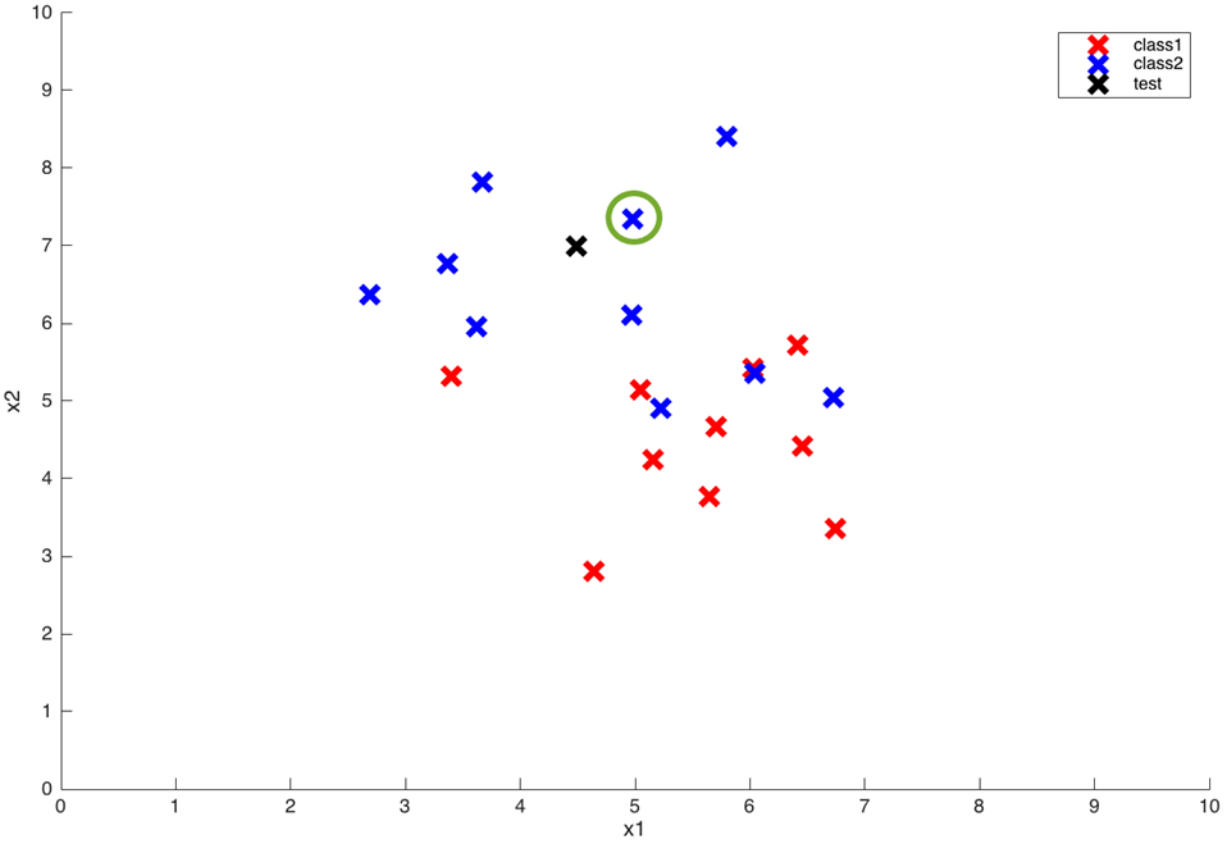


The NN algorithm finds the nearest training point to this test point (by measuring the distance of test point from **every** training point), and the class predicted of test point is the same as of nearest training point

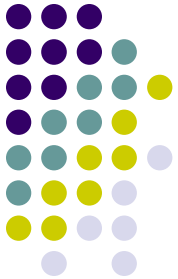
The lower the distance, the higher the **similarity** between two points

Support Vector Machine (SVM)

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



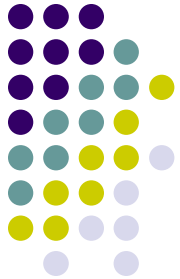
Support Vector Machine (SVM)



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)



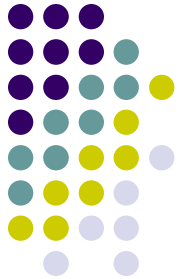
SVM remove each of these two problems

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

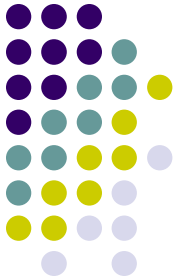
Support Vector Machine (SVM)



Moreover, once we have selected support vectors, we 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

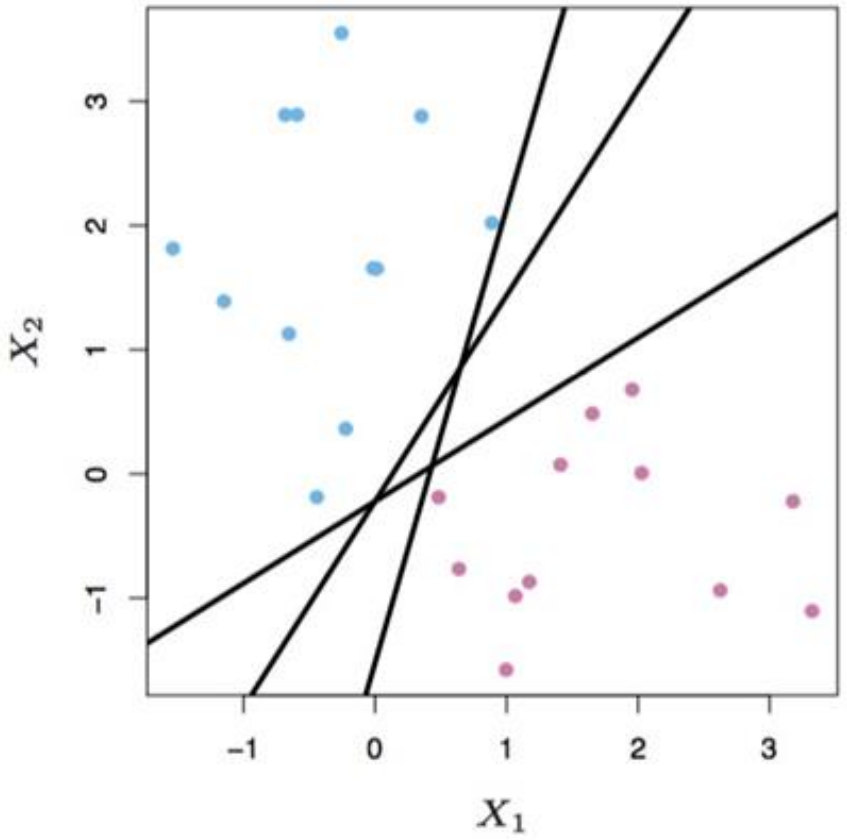
So unlike NN, we don't give importance to only a single training point (i.e., *document*, given that we are dealing with text classification), instead we give each support vector a separate importance

Support Vector Machine (SVM)

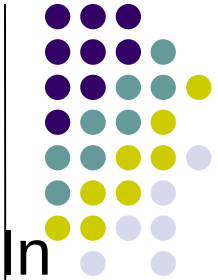


But how to find a **support vector**?

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



Support Vector Machine (SVM)

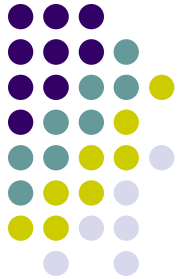


This is harder to visualize in more than two dimensions. In this case you need an hyperplane

More formally, 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

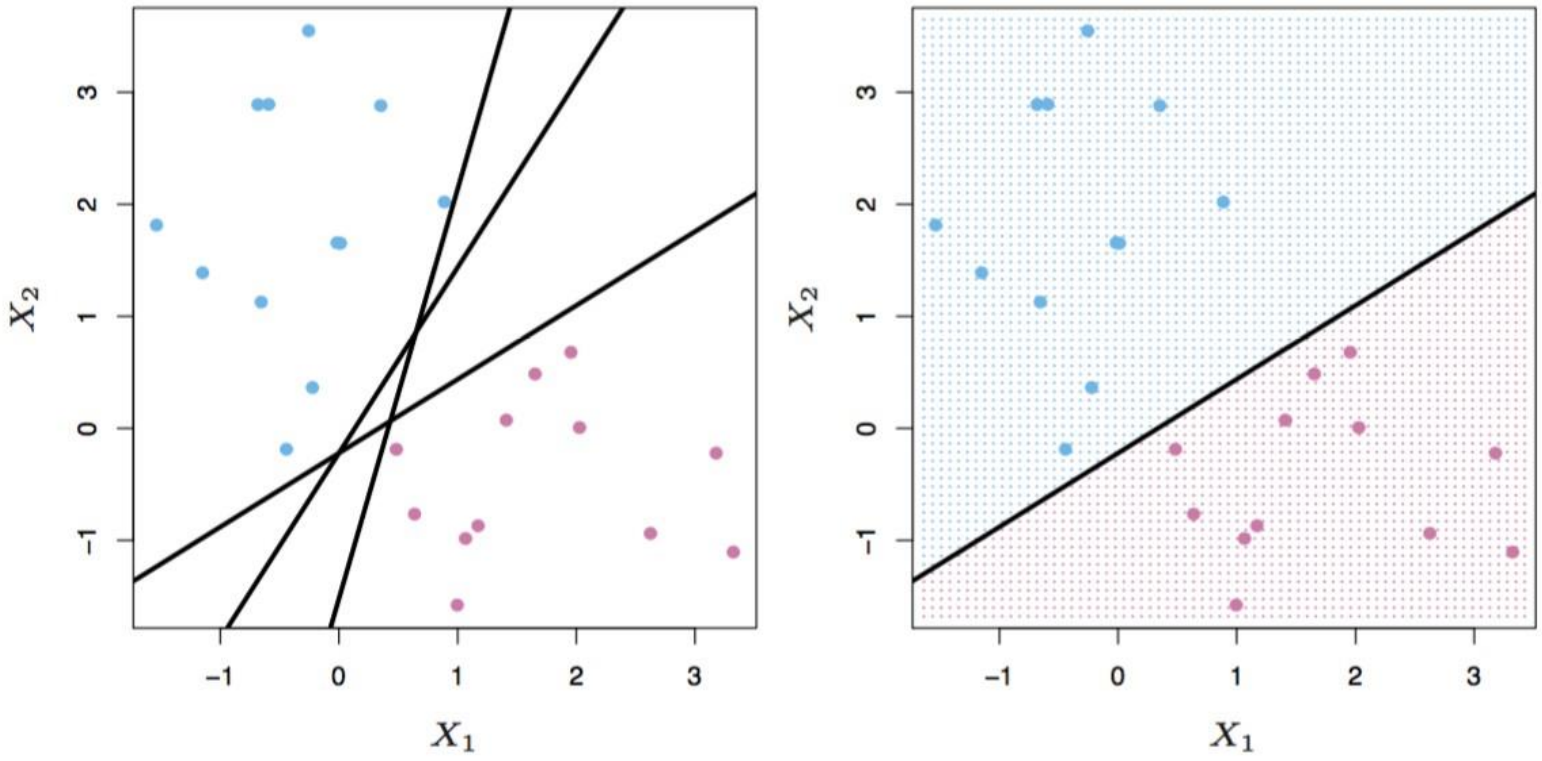
That is, our goal is to identify the hyperplane that maximizes the total distance between the line and the closest point in each class

Essentially, it 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!

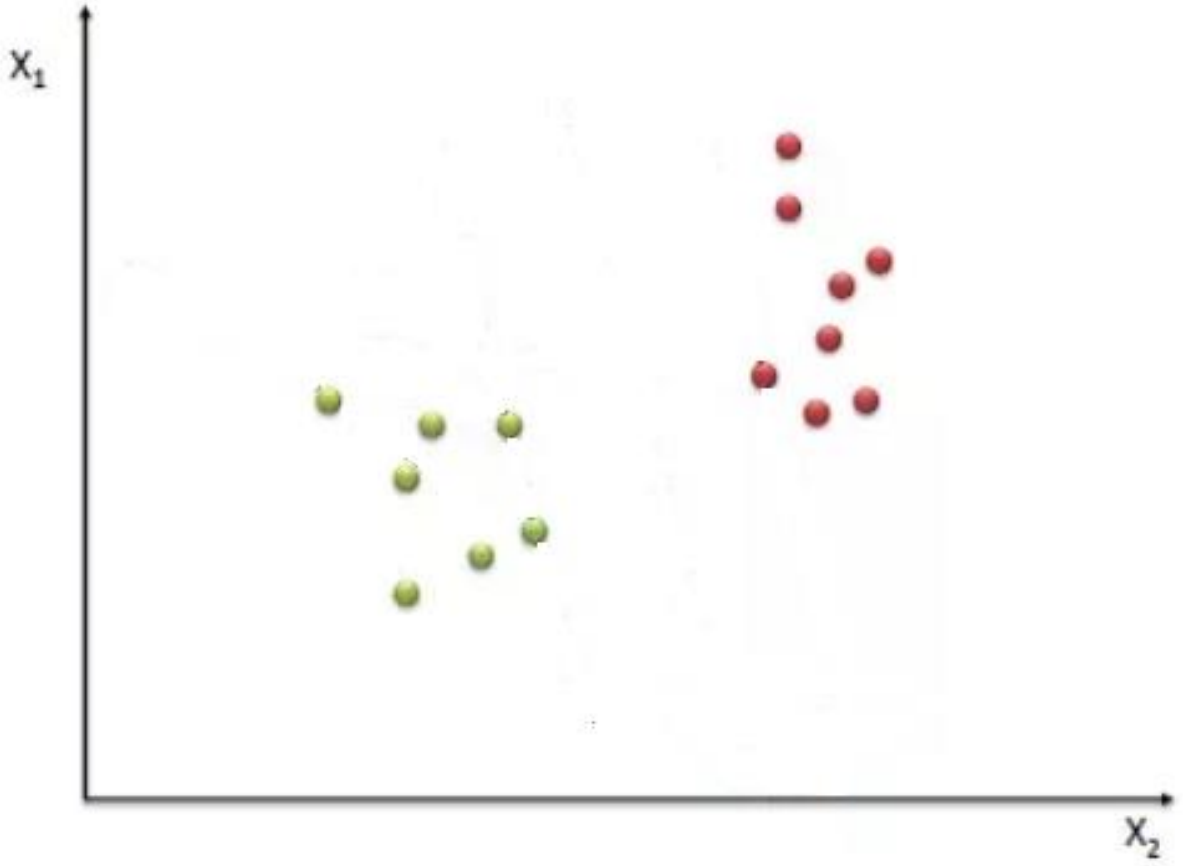
Support Vector Machine (SVM)

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

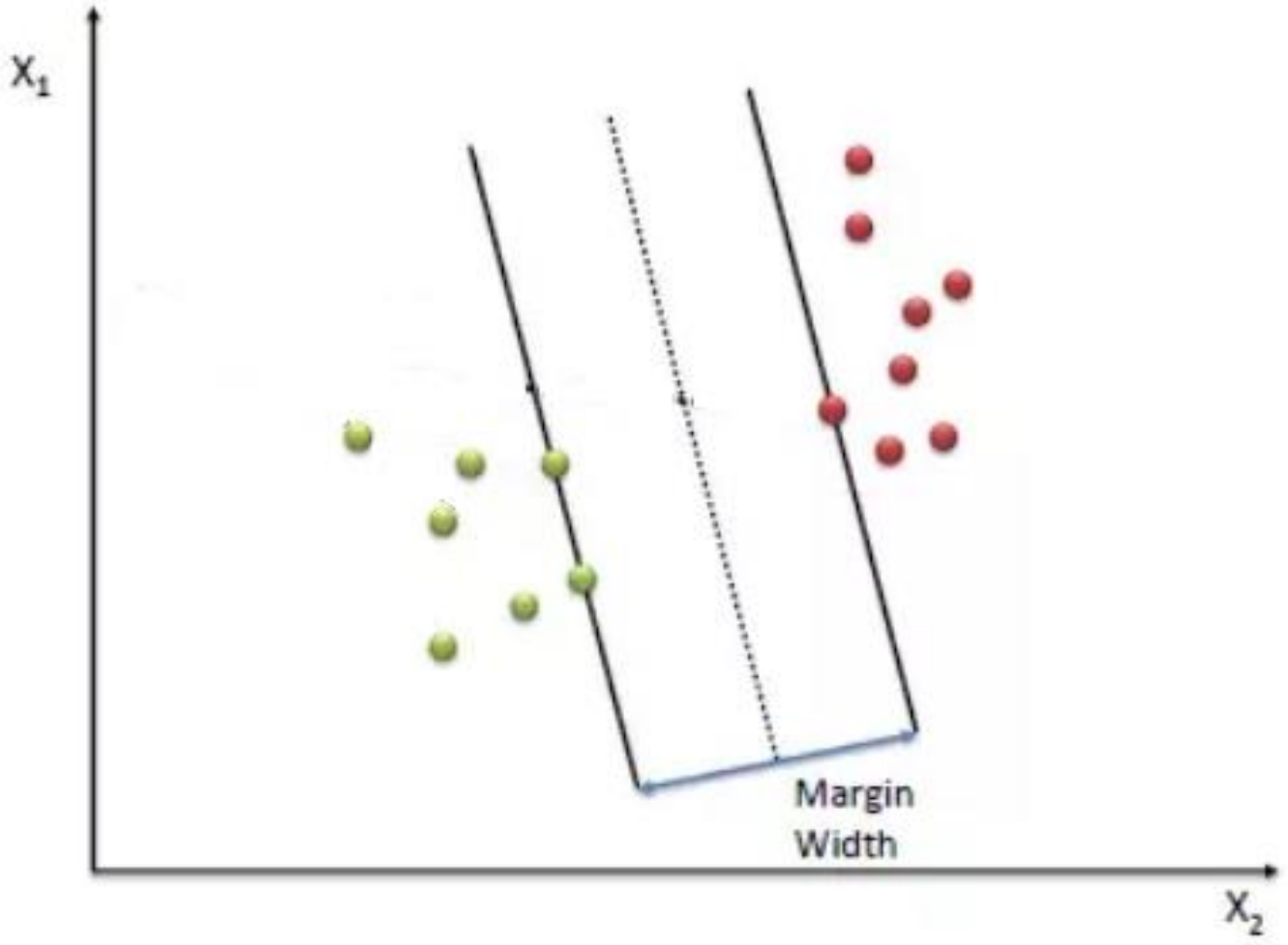


Support Vector Machine (SVM)

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



Support Vector Machine (SVM)

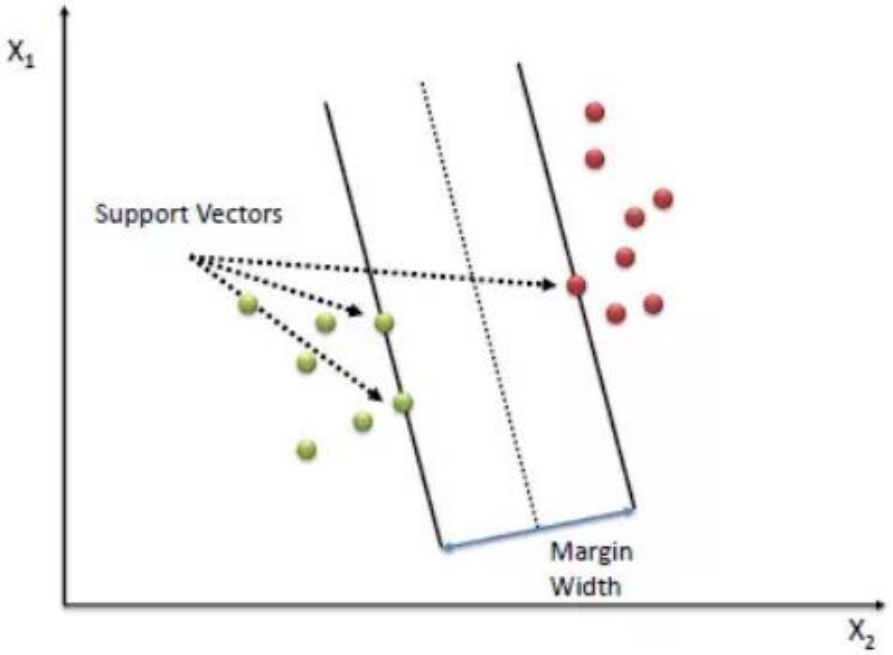


Support Vector Machine (SVM)

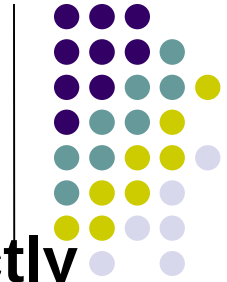
The data points that kind of "support" this hyperplane on either sides (i.e., the closest training points to the line) are called the **support vectors**

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

In the figure, there are 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)



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

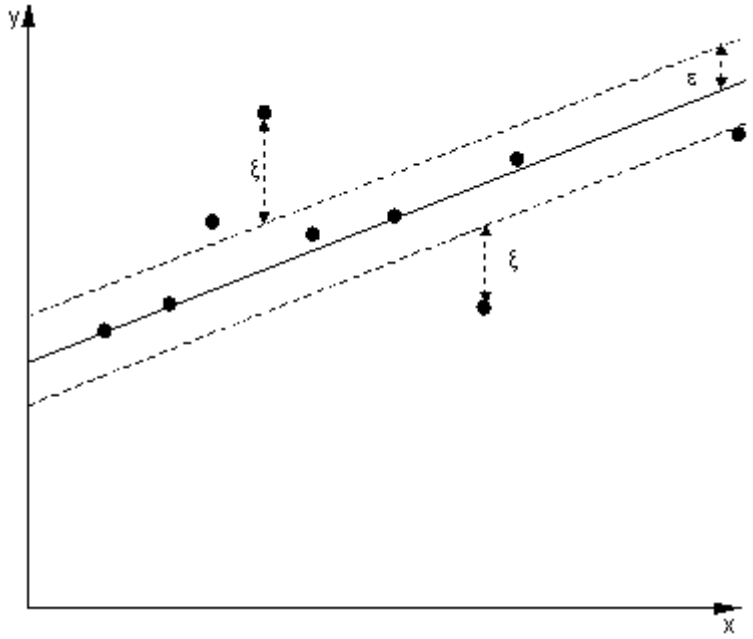
When this is not the case, we must relax the constraint imposed on the distances between points and the hyperplane, and allow for a **certain amount of slack**

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

Support Vector Machine (SVM)

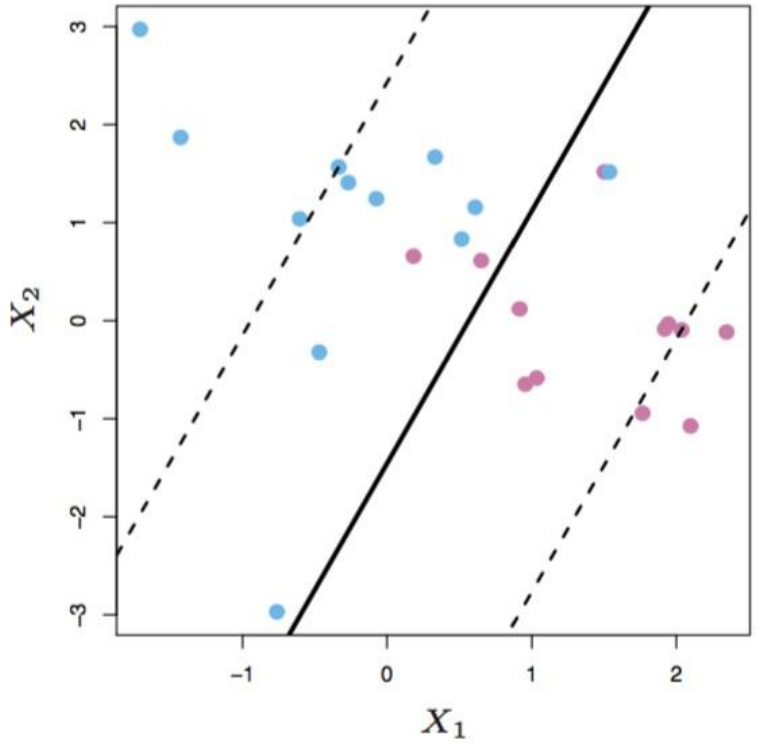
In this respect we can define a *loss function* that ignores those errors which are situated within the certain distance of the true value

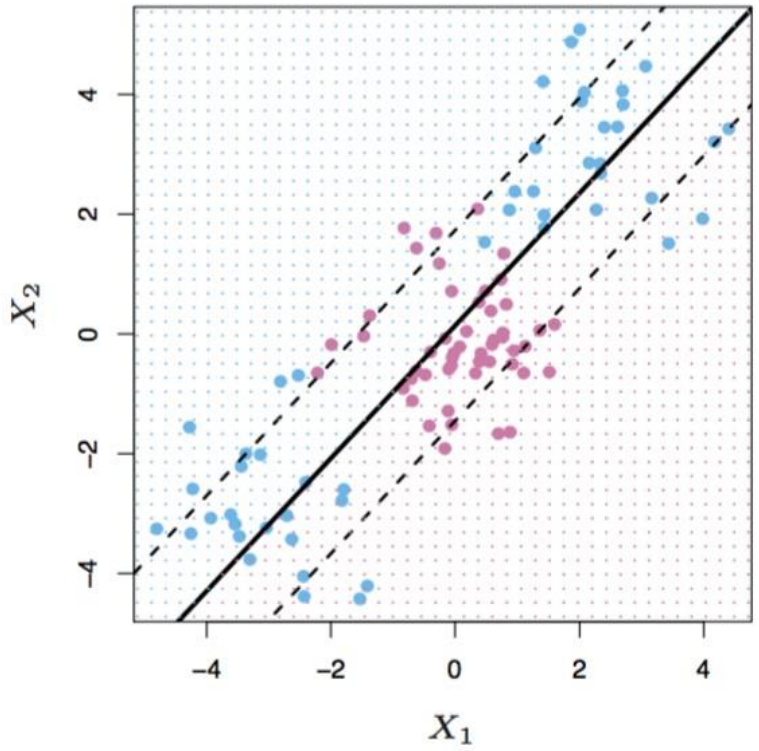
This type of function is often called – epsilon (ε) intensive-loss function. The figure shows an example of an hyperplane with epsilon intensive-loss band



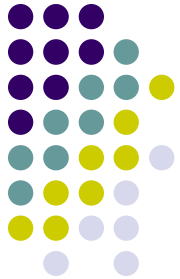
Support Vector Machine (SVM)

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





Support Vector Machine (SVM)

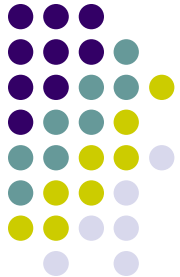


This function allows us to identify the **cost** of the errors on the training points

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)

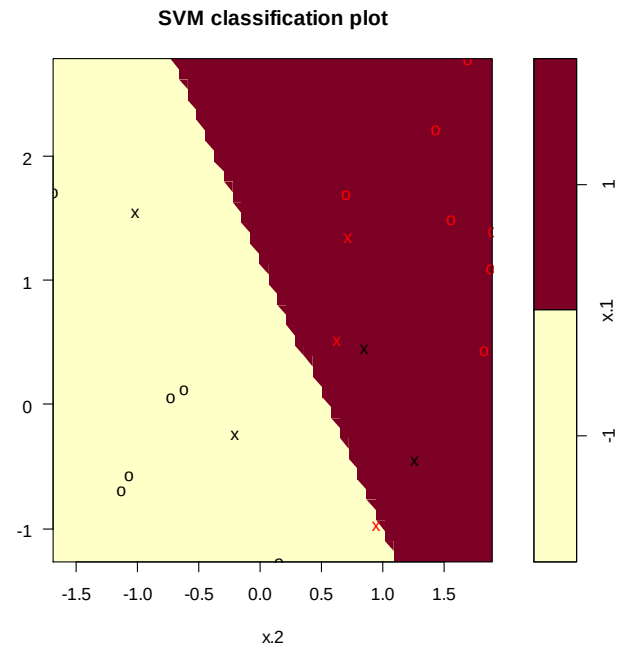
Larger values of *C* (ex. *C* = 1000 , a value which penalizes a lot the model for misclassified observations) thus result in greater focus of attention on the points located **very close** to the decision boundary (for a given ε), i.e., only those **instances near the class boundary play a big role 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

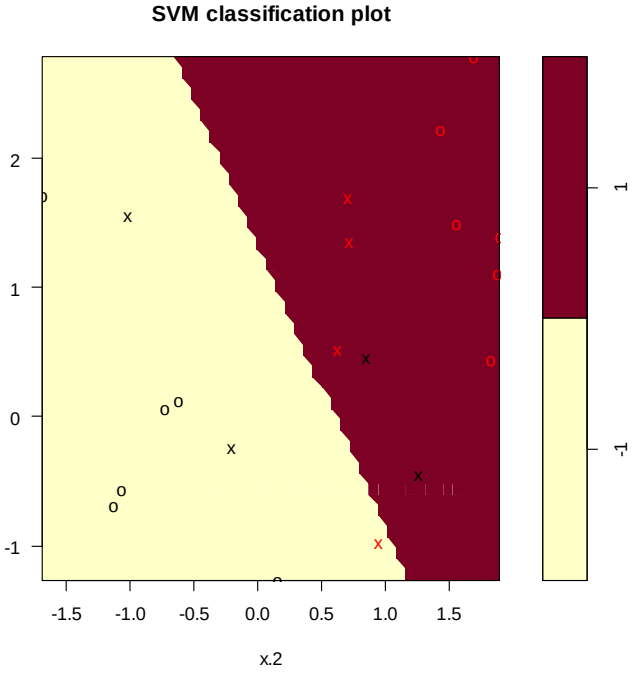
Support Vector Machine (SVM)

An example with three different values for *C* (and epsilon fixed to 0.1)

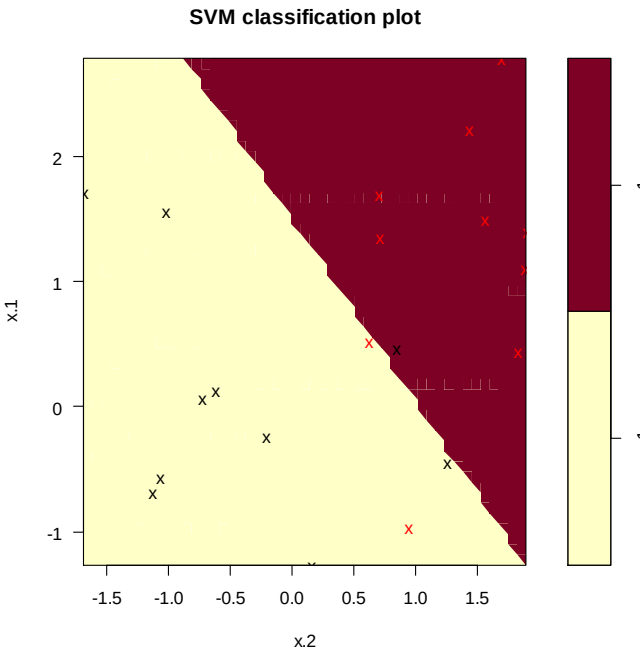
C=10; SV points (i.e., the Xs)=7



C=1; SV points=8



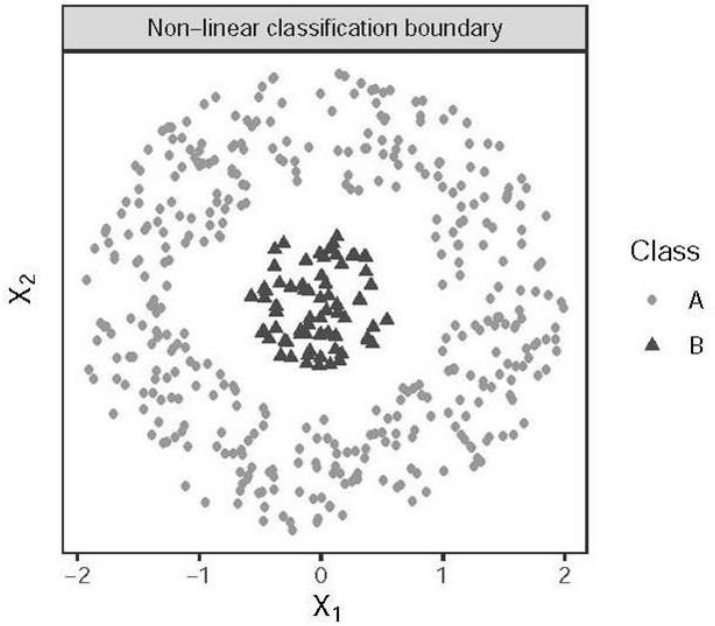
C=0.01; SV points=ALL! (although of course with a different weight!)



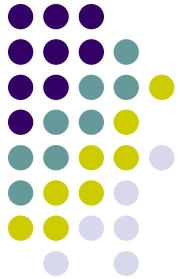
Support Vector Machine (SVM)

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)

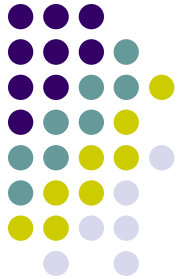


So what to do? **Move to non-linearity**!

We achieve this 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)



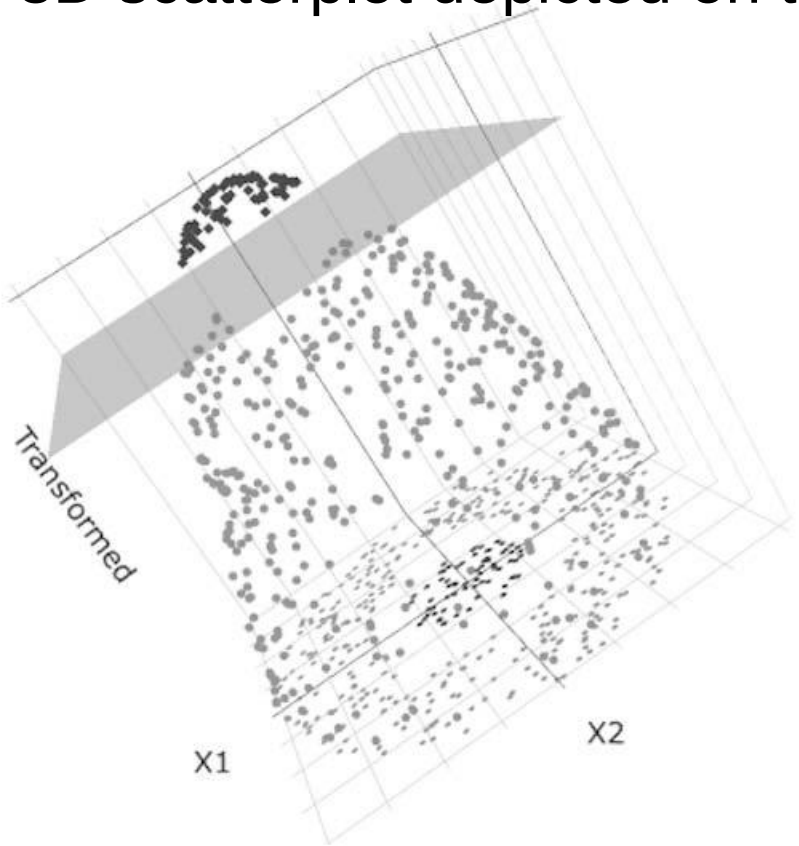
For example, if we can't draw a line in the space (x1, x2) then we may try adding a third dimension, (x1, x2, x1 * x2)

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



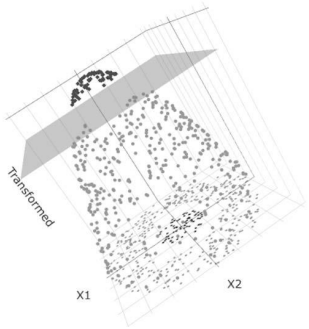
Support Vector Machine (SVM)

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

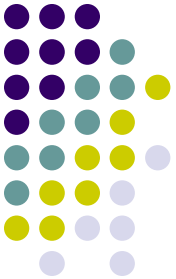
It is now easy to define a plane, depicted in gray, that cuts the top of this cone and separates instances of the two classes

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:

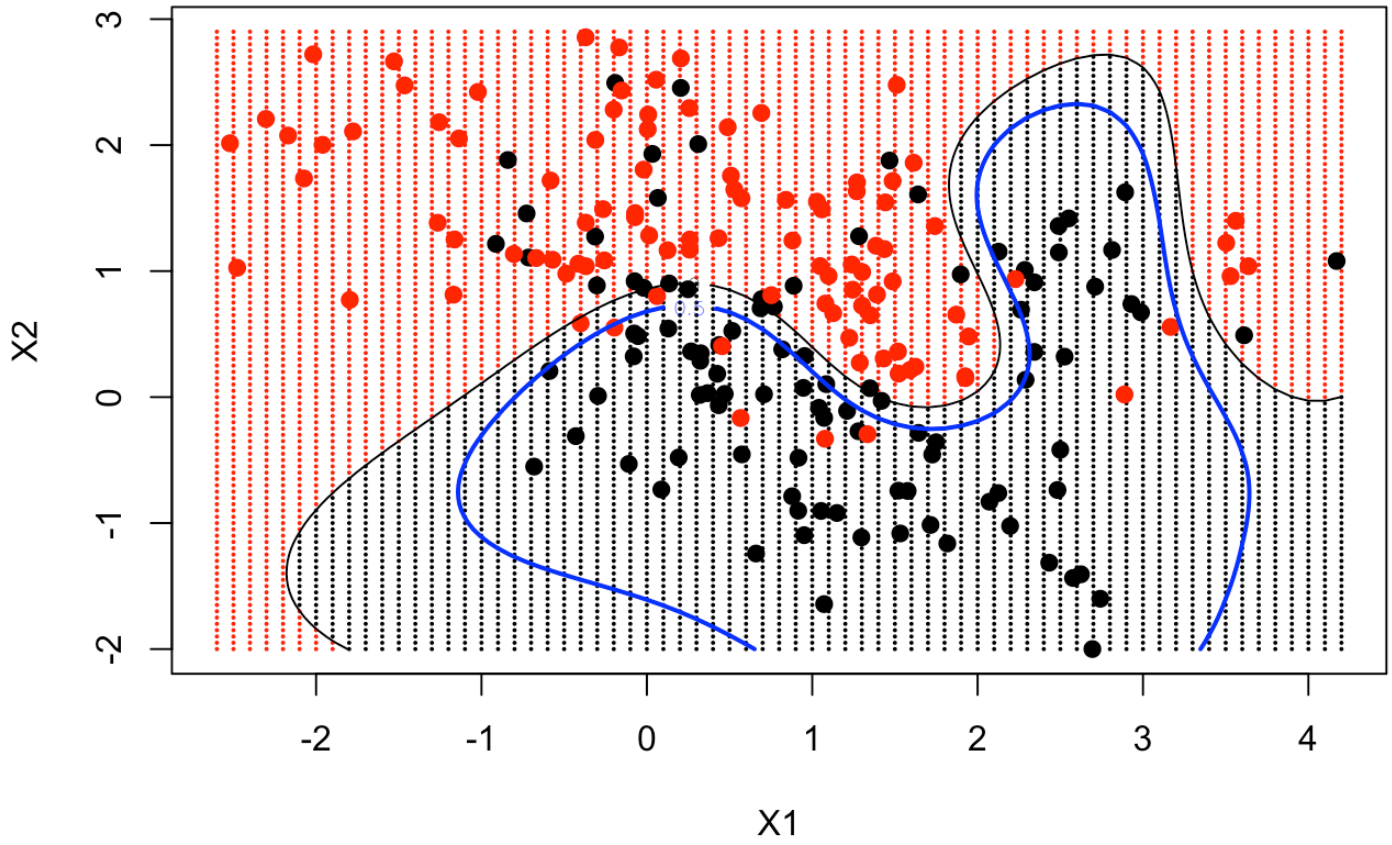
Popular SVM Kernels

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	$K(x, x') = \exp(-\|\|x - x'\|\|^2/c)$

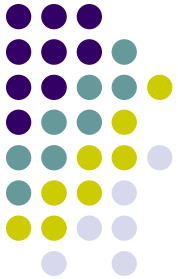
The linear one usually works fine with text classification

Support Vector Machine (SVM)

An example with a radial Kernel



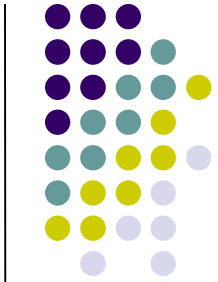
Support Vector Machine (SVM)



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



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

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

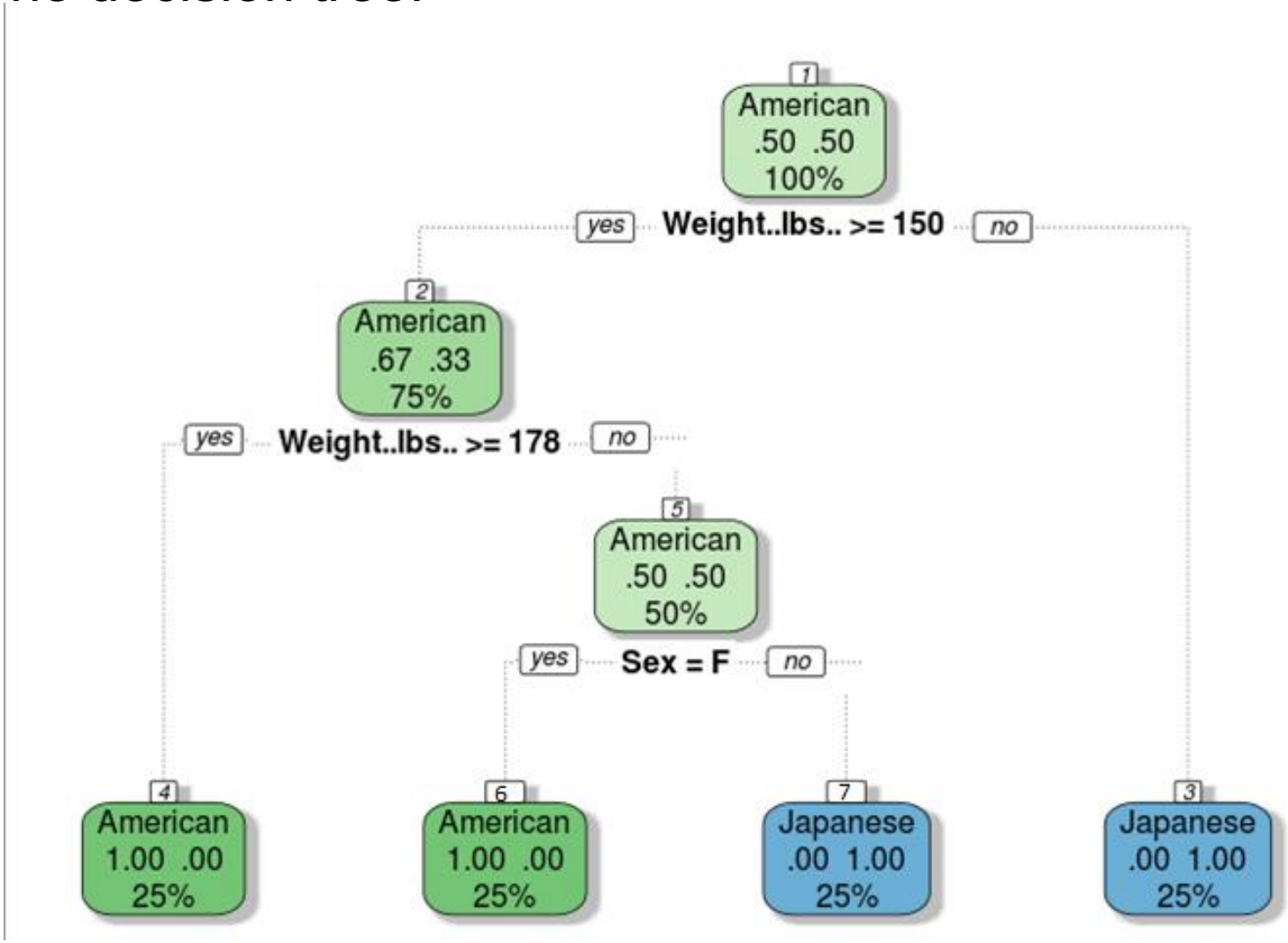
Key idea: the procedure to create decision trees is **recursive**. For a set (**S**) of observations, the following algorithm is applied:

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

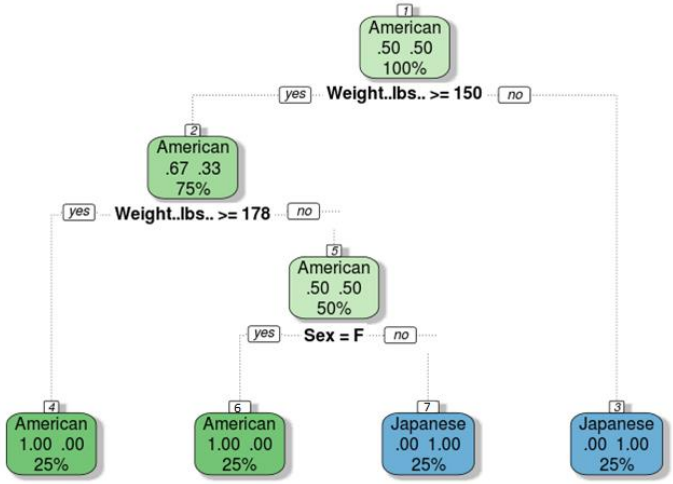
The decision tree:



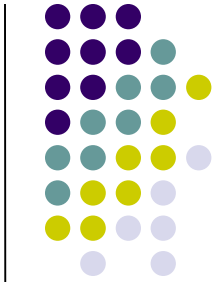
Random Forest classifier

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



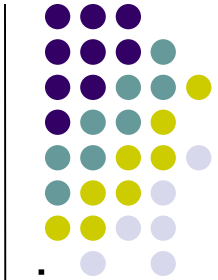
In this example the tree can **perfectly explain** the data

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



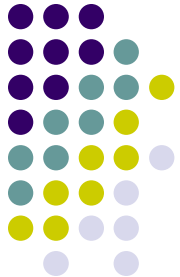
Trees are usually "pruned" to avoid **overfitting**. The pruning algorithm removes final nodes so that the model is a little more general and will tend to generalize better to new, unseen data

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

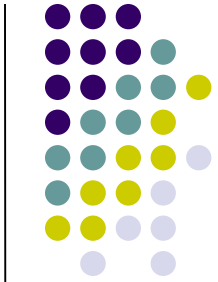
The sequential nature of the recursive splitting algorithm means in fact that the structure of the tree is often highly sensitive to small changes in the observations included

Random Forest classifier

So what to do to avoid overfitting and preventing results highly sensitive to small changes in the observations included in the training set (part 2)?



Random Forest classifier



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

Bagging combines and averages multiple models. Averaging across multiple trees reduces the variability of any one tree and reduces overfitting, which improves predictive performance. Bagging follows three simple steps:

1. Create *m bootstrap samples* from the training data.

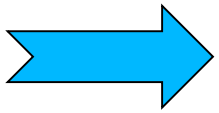
Wait…what? What do we mean by **bootstrapping**?

In essence bootstrapping **repeatedly draws independent samples** from our data set to create bootstrap data sets. This sample is 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

Random Forest classifier

An example with 3 bootstrap samples



Obs	X	Y
1	4.3	2.4
2	2.1	1.1
3	5.3	2.8

Original Data (Z)

Z^{*1}

Obs	X	Y
3	5.3	2.8
1	4.3	2.4
3	5.3	2.8

$\hat{\alpha}^{*1}$

Z^{*2}

Obs	X	Y
2	2.1	1.1
3	5.3	2.8
1	4.3	2.4

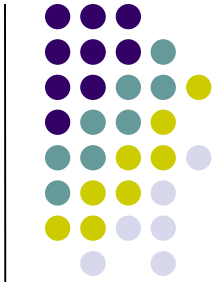
$\hat{\alpha}^{*2}$

Z^{*B}

Obs	X	Y
2	2.1	1.1
2	2.1	1.1
1	4.3	2.4

$\hat{\alpha}^{*B}$

Random Forest classifier

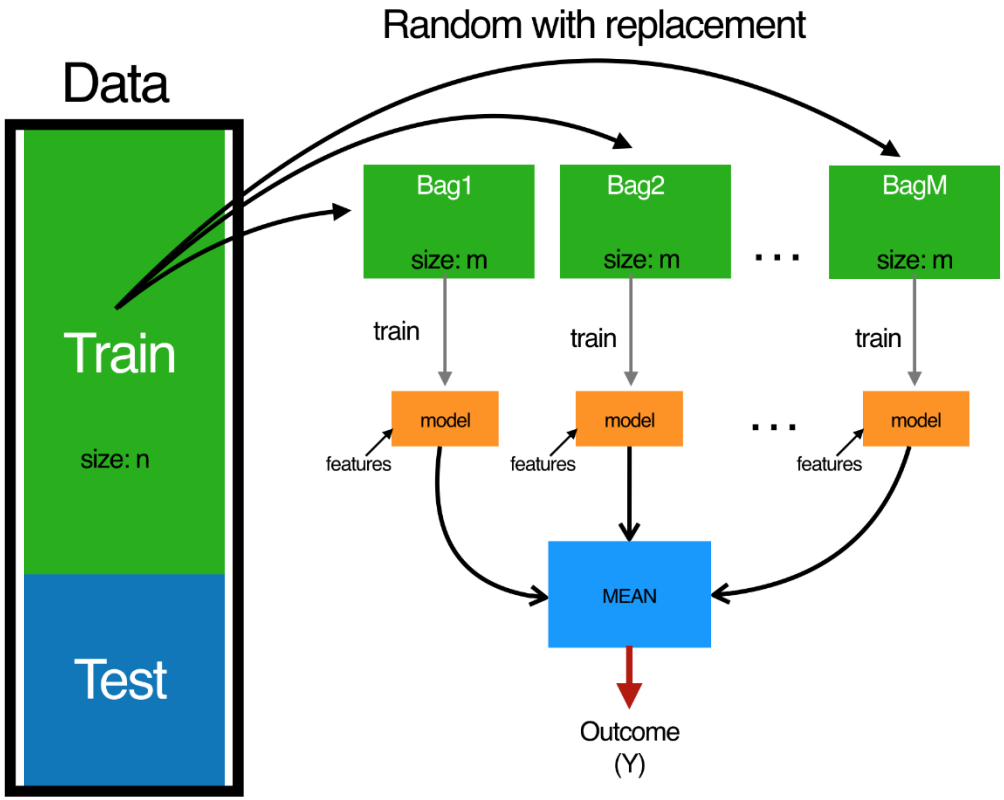


***B**ootstrap **agg**regat**ing** (**bagging**)!*

Bagging combines and averages multiple models. Averaging across multiple trees reduces the variability of any one tree and reduces overfitting, which improves predictive performance. Bagging follows three simple steps:

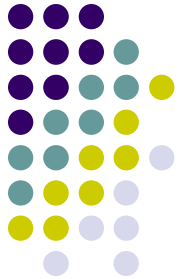
1. Create *m bootstrap samples* from the training data. Bootstrapped samples allow us to create many slightly different data sets but with the same distribution as the overall training set. The bootstrap samples must not necessary have the same size of the original training-set
2. For each bootstrap sample train a single, unpruned classification tree
3. Average individual predictions from each tree to create an overall average predicted value

Random Forest classifier



One benefit of bagging is that, on average, a bootstrap sample will contain 63% of the training data (a parameter you can change!). This leaves about 37% of the data out of the bootstrapped sample. This is the out-of-bag (OOB) sample.
As we will see, we can use the **OOB observations** to have a first idea of the model's accuracy

Random Forest classifier



Still, bagging for itself cannot be enough…

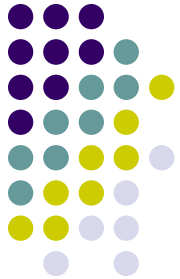
Bagging trees introduces a random component in to the tree building process that, when building an ensemble, reduces the variance of a single tree's prediction and improves predictive performance

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

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

And so? How to reduce the correlation among trees?

Random Forest classifier



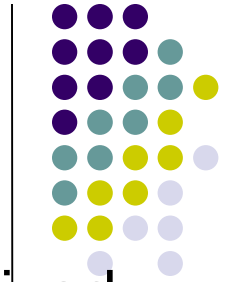
The Random Forest (RF) idea!

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

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

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

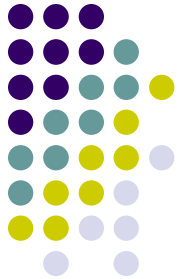
Random Forest classifier



The basic algorithm for a random forest can be 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

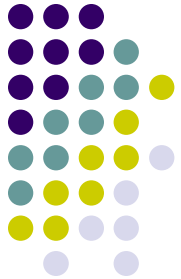


By fitting a tree (with no pruning) to each bootstrapped sample **and** by restricting the choice of each splitting variable to a random subset of predictors, we are sure that each boostrapped tree provides a truly different “perspective" on the prediction problem. All this, of course, minimizes the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

Gradient Boosting classifier

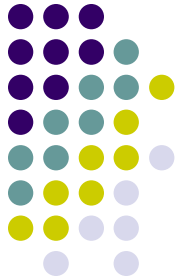


Several supervised machine learning models are founded on a single predictive model (i.e. naive Bayes, support vector machines)

Alternatively, other approaches such as bagging and random forests are built on the idea of building an *ensemble of models* where each individual model predicts the outcome and then the ensemble simply averages the predicted values

The family of **boosting methods** is based on a different, constructive strategy of *ensemble formation*

Gradient Boosting classifier



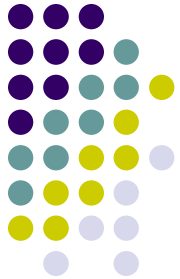
The main idea of boosting is to add new models to the ensemble ***sequentially***

At each particular iteration, as we will see, a *new weak, base-learner model* (what's that??? Give me a moment…) is trained with respect to the error of the whole ensemble learnt so far

Boosting is a framework that iteratively improves *any* weak learning model

In practice however, boosted algorithms almost **always use decision trees as the base-learner**

Gradient Boosting classifier

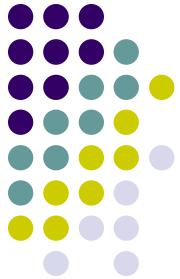


Whereas Random Forests build an ensemble of *deep independent trees*, **Gradient Boosting Machines** (GBM) build therefore an ensemble of shallow and weak *successive trees* with **each tree learning and improving on the previous**

When combined, these many weak successive trees produce a powerful “committee”

Let’s see how

Gradient Boosting classifier



Training weak models:

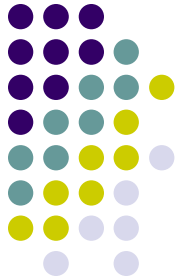
A *weak model* is one whose error rate is only slightly better than random guessing

The idea behind boosting is that each sequential model builds a simple weak model to slightly improve the remaining errors

With regards to decision trees, shallow trees represent a weak learner. Commonly, trees with only 1-6 splits are used

Combining many weak models (versus strong ones) has several benefits

Gradient Boosting classifier



Which advantages of doing it?

Speed: Constructing weak models is computationally cheap

Accuracy improvement: Weak models allow the algorithm to *learn slowly*; making minor adjustments in new areas where it does not perform well. In general, statistical approaches that learn slowly tend to perform well

Avoids overfitting: Due to making only small incremental improvements with each model in the ensemble, this allows us to stop the learning process as soon as overfitting has been detected (typically by using cross-validation)

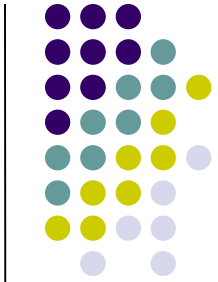
Gradient Boosting classifier

Boosted trees are **grown sequentially**; each tree is grown using information from previously grown trees

The basic algorithm for boosted classification trees can be generalized to the following where *x* represents our features and *y* represents our response

1. Fit a decision tree to the data: $F_1(x)=y$
2. We then fit the next decision tree to the classification errors of the previous: $h_1(x)=y-F_1(x)$
3. Add this new tree to our algorithm: $F_2(x)=F_1(x)+h_1(x)$
4. Fit the next decision tree to the classification errors of F_2: $h_2(x)=y-F_2(x)$
5. Add this new tree to our algorithm: $F_3(x)=F_2(x)+h_2(x)$
6. Continue this process until some mechanism (i.e. cross validation) tells us to stop

Gradient Boosting classifier



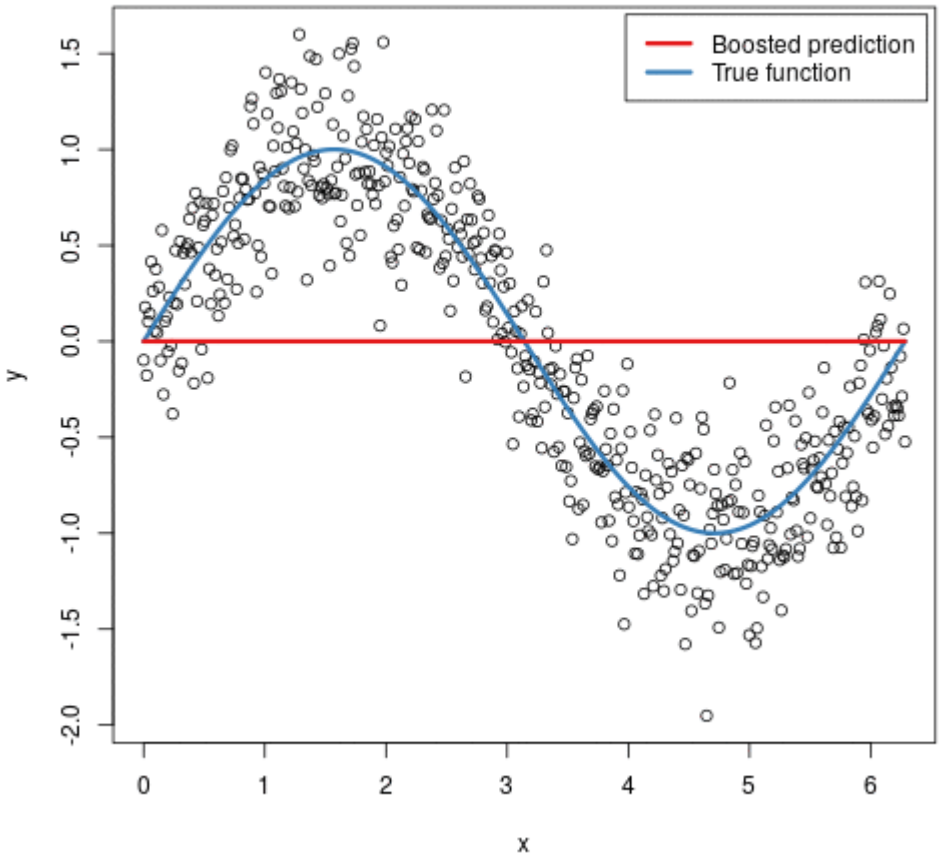
To illustrate the behavior, assume the following situation

The blue sine wave represents the true underlying function and the points represent observations that include some irreducible error

The boosted prediction (in red) illustrates the adjusted predictions after each additional **sequential tree** is added to the algorithm

Initially, there are large errors which the boosted algorithm improves upon immediately but as the predictions get closer to the true underlying function you see each additional tree make small improvements in different areas across the feature space where errors remain

Gradient Boosting classifier

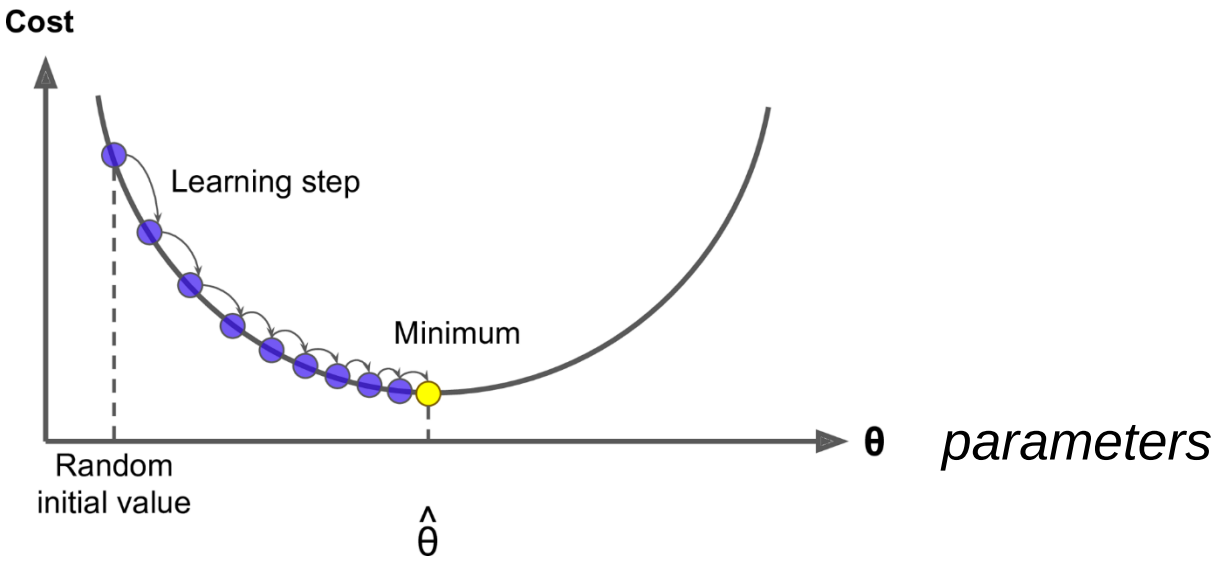


Gradient Boosting classifier

Gradient boosting is considered a ***gradient descent*** algorithm

Gradient descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems

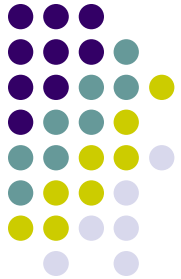
The general idea of gradient descent is to tweak parameters iteratively in order to minimize a given cost function



Gradient Boosting classifier

Gradient descent can be performed on a variety of loss functions (for example the mean squared error loss function, the mean absolute error, deviance, and, most importantly for text analytics, classification error, etc.)

Gradient Boosting classifier

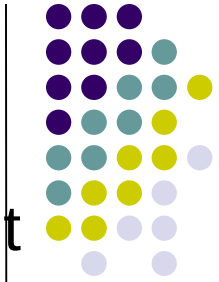


GBC are computationally expensive - they often require many trees (>1000) which can be time and memory exhaustive

The high flexibility results in many parameters that interact and influence heavily the behavior of the approach (number of iterations, tree depth, regularization parameters, etc.)

This requires a large *grid search* during tuning to find the best combination of hyperparameters given a specific task at hand

Gradient Boosting classifier

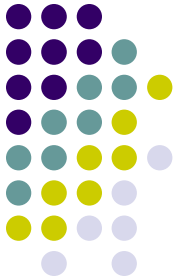


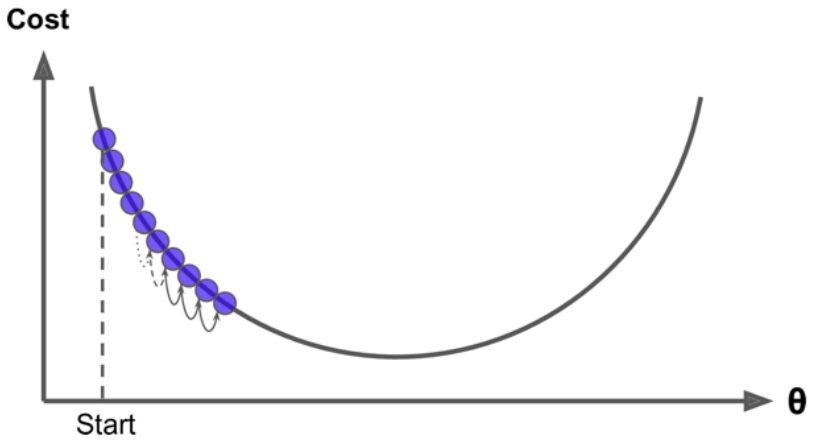
For example: an important parameter in gradient descent is the size of the steps which is determined by the *learning rate* (eta). It controls how quickly the algorithm proceeds down the gradient descent

If the learning rate is too small, the algorithm will take many iterations to find the minimum (this reduces the chance of overfitting but also increases the time to find the optimal fit)

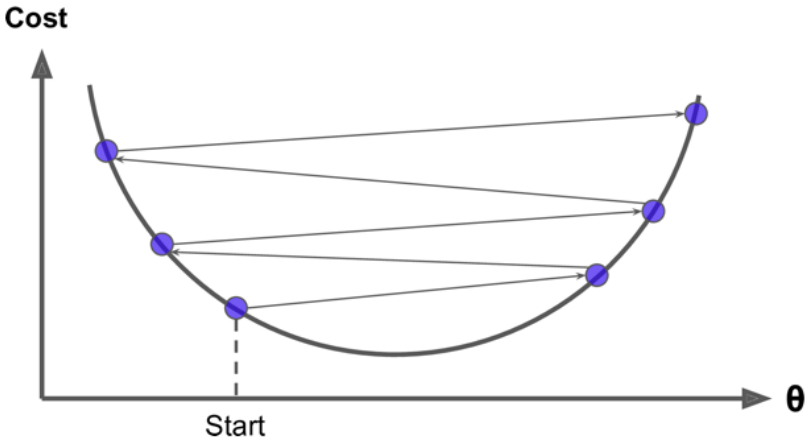
On the other hand, if the learning rate is too high, you might jump cross the minimum and end up further away than when you started

Gradient Boosting classifier



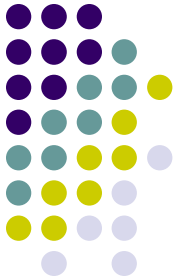


a) too small



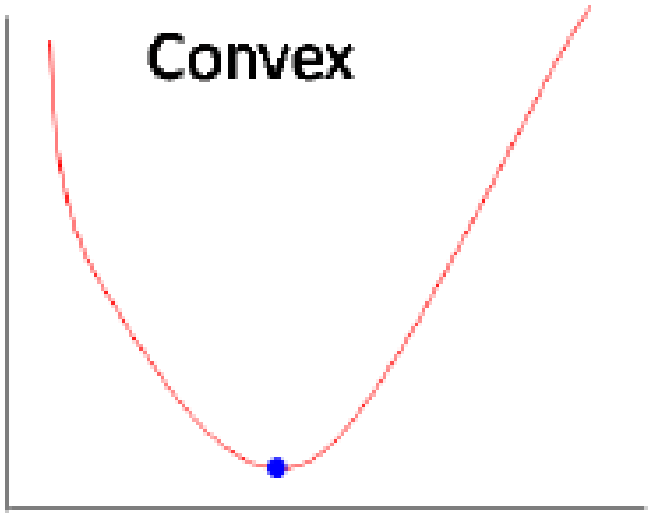
a) too big

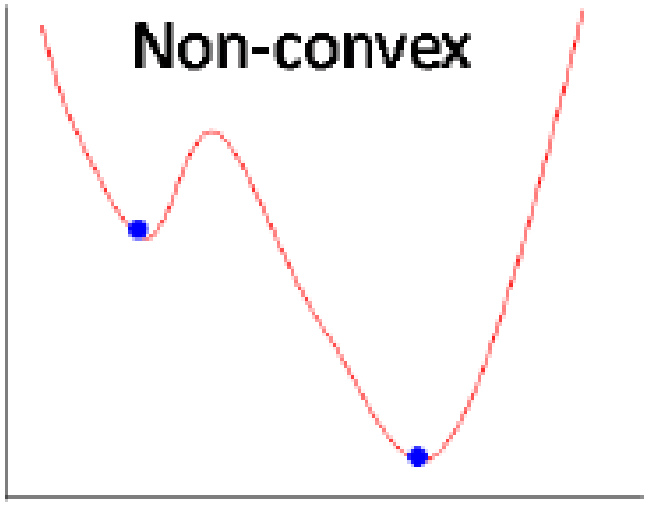
Gradient Boosting classifier



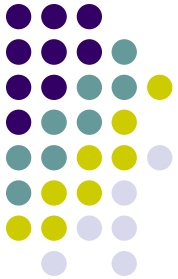
Moreover, not all cost functions are convex (bowl shaped)

There may be local minimas, plateaus, and other irregular terrain of the loss function that makes finding the global minimum difficult





Gradient Boosting classifier

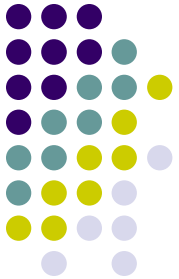


Stochastic gradient descent can help us address this problem by sampling a fraction of the training observations (typically without replacement) and growing the next tree using that subsample

This makes the algorithm faster. Moreover, the stochastic nature of random sampling also adds some random nature in descending the loss function gradient

Although this randomness does not allow the algorithm to find the absolute global minimum, it can actually help the algorithm jump out of local minima and off plateaus and get near the global minimum

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

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

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

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

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

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

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

R pakcages to install

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

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

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

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

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

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

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

install.packages("Ckmeans.1d.dp", repos='http://cran.us.r-project.org')