HEC MONTRÉAL

Theme 7

Uplift, rare classes, ensemble methods, SVM

Uplift modeling (true lift)

- For targeted actions (e.g., phone call inviting to donate)
- If we target high potential individuals, many would give without our intervention.
- The real question therefore is:

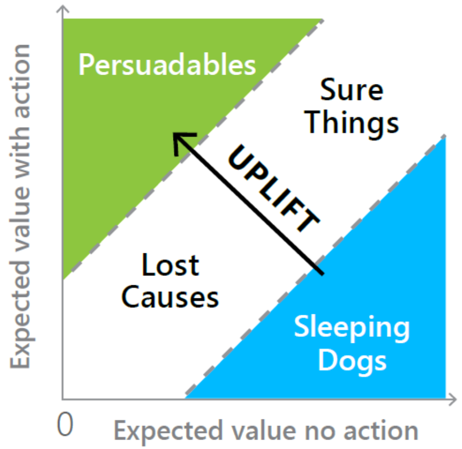
"What value do I get from calling somebody?"

Hypothetical scenario: we call individuals to invite donations:

- We remind somebody to give? 👍
- We make no difference in the donation? 👎
- After being bugged, the individual decides they will not give any more? 👎

The challenge consists in:

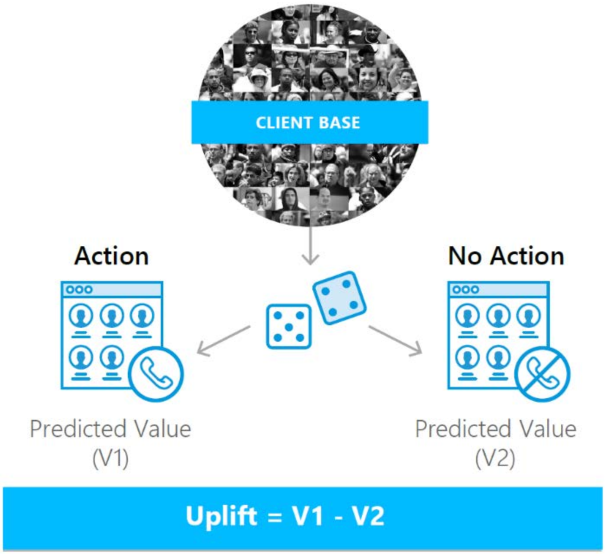
- Contacting the "persuadables"
- Avoiding the "sure things"
- Avoiding the "lost causes"
- Avoiding the "sleeping dogs" at all price



In true lift modeling, we want to compare the value of the customer with or without the treatment: here, calling or not. Data collection must be planned:

You must **randomly assign** individuals to the treatment / control

Convincing others to proceed is a colossal challenge.



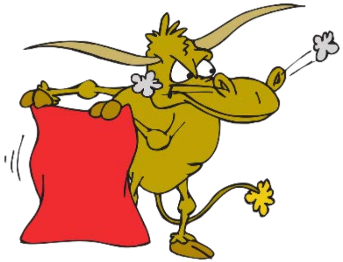
There exist other approaches, but one way is to:

- Use only the data of people who received the treatment (action) to build a model predicting the value (probability or expected value) of an individual,
- Use only the data of people who did not receive the treatment (no action) to build a model predicting the value (probability or expected value) of an individual
- Use what we learned so far on both datasets separately from each other.
- Use the two models found on the complete dataset, hence providing an estimated value for each individual, with and without the marketing treatment.
- Compute the difference between these two values. This is the uplift generated by the action.

Ensemble methods

Are multiple simple models more powerful than a single complex one?

At a fair in 1906, people were asked to guess the weight of an ox. Could the average of the guessed weights be more precise than the measurement of a scale?



Can we do the same thing with models?

- Instead of one model, we consider multiple simpler models,
- For a given individual, each model makes a prediction,
- The prediction is obtained from the average of all predictions,
- The ensemble typically provides a better prediction.

How to combine models?

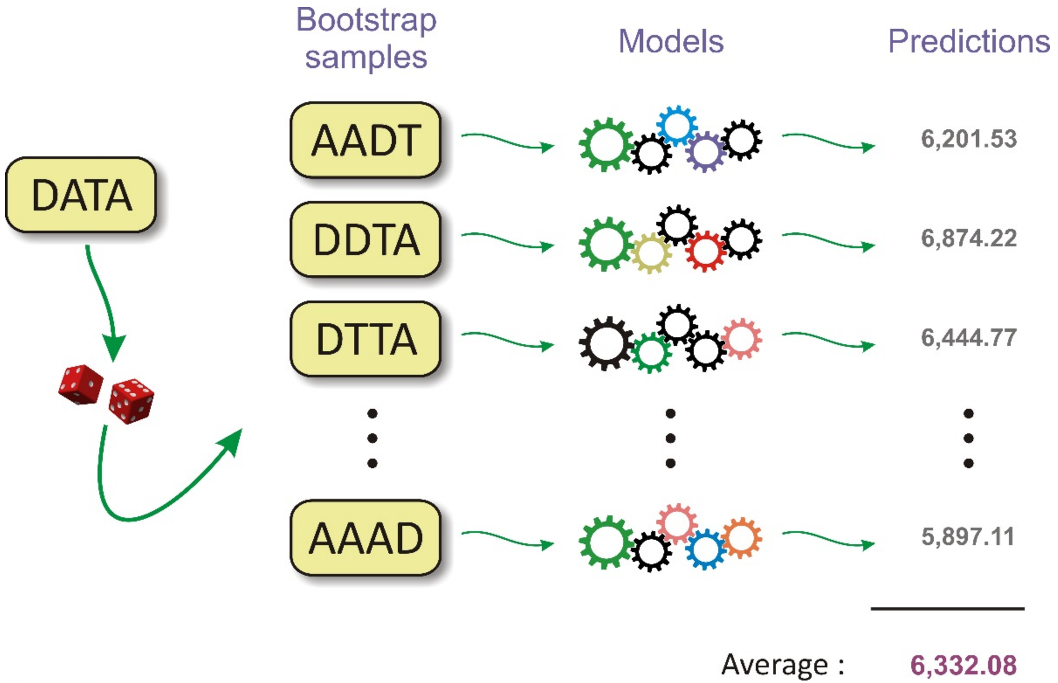
Each model makes a prediction for each individual. Then we may:

- Average the prediction (works for continuous and binary outcomes)
- Take a majority vote (for classification)

An ensemble method will yield a better model when the models in the ensemble make mistakes on different data.

In principle, any models may be combined into an ensemble.

Bagging = "Bootstrap Aggregating"



Bootstrap is used to generate disturbances in the data.

Key idea:

- Relevant variables should typically stay in the model,
- Variables that appear relevant by chance are unlikely to stay all the time,
- The bagged models should detect the true nature of the data.

The number of bootstrap samples, B, is chosen arbitrarily.

- Nonparametric bootstrap: draw individuals with replacement from the original data. Some data points will appear multiple times.
- May draw without replacement if bootstrap samples are smaller than the whole data (accelerates computation too).

Random forests

Random forests use bagging with trees.

For each split of each tree, only $\sqrt{p}$ predictors are considered instead of p:

- Speeding up computation
- Increasing the diversity of trees

For continuous targets, the default number of predictors is sometimes $p/3$ instead.

The possible predictors are chosen randomly at each split.

In addition, the trees are not pruned.

In most implementations, surrogate rules are not kept (i.e., missing values are not treated automatically).

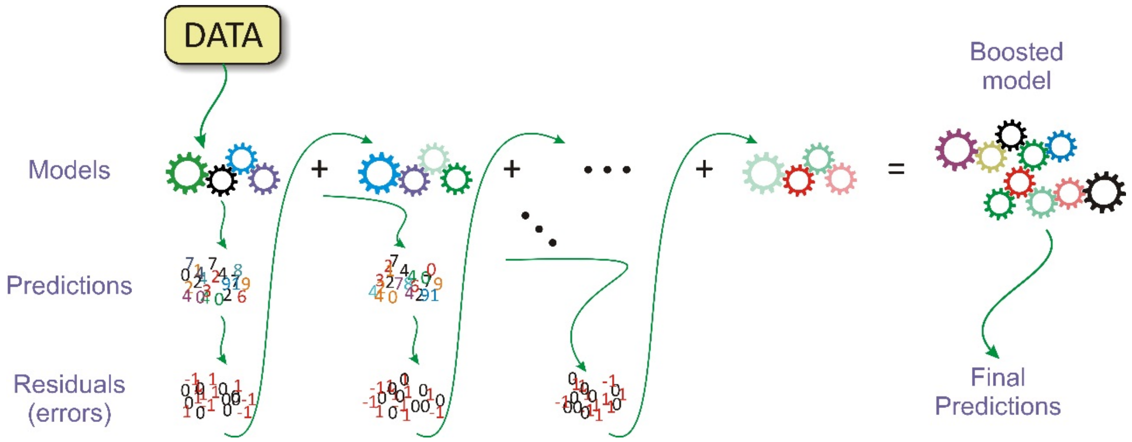
Strengths of random forests:

- Typically provides excellent predictive performance
- Seamless treatment of missing values possible (but not typically implemented),
- No need to transform the variables,
- Takes account of interactions implicitly,
- Robust to extreme values.

Weaknesses of random forests:

- Impossible to interpret (black box),
- Computationally hungry (to fit and to make predictions).

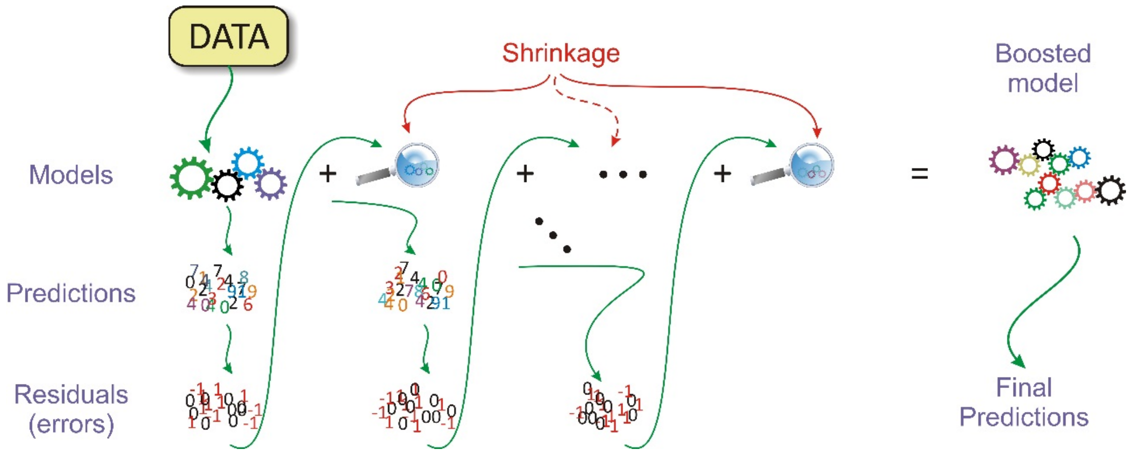
“Boosting”



“Boosting” improves the model sequentially by focusing on the errors and trying to fix them.

- Larger errors may get larger weights (but not necessarily),
- “Gradient Boosting” is based on trees.

Shrinkage



Performance is improved when a shrinkage factor (often < 0.1) is used.

The strengths and weaknesses of gradient boosting are similar to those of the random forests. As an additional weakness, gradient boosting has a stronger tendency to overfitting.

AdaBoost and Arcing

Let us see in detail two examples of boosting algorithms.

- AdaBoost (Adaptative Boosting), Freund et Schapire (1997).
- Arcing (Adaptive Resampling and Combining), Breiman (1998)

In both cases, misclassified data see their weight increase at each step:

1. Initialize sampling weights $w_1^{(1)} = \cdots = w_n^{(1)} = 1/n$.
2. At step m, draw a bootstrap sample based on weights $w_1^{(m)}, \ldots, w_n^{(m)}$
3. Build model $G_m(X)$ with the bootstrap sample, and determine a weight α_m
4. Update the weights and repeat 2 and 3 for M steps
5. The final model is given by

$$\hat{G}(X) = \sum_{m=1}^{M} \alpha_m G_m(X)$$

The two algorithms differ in the way weights are determined.

Adaboost looks only at the current model:

- Note e_m the misclassification rate of $G_m(X)$.
- The weights are $\alpha_m = \log\{(1-e_m)/e_m\}$
- The bootstrap weights are

$$w_i^{(m+1)} = w_i^{(m)} \exp\{\alpha_m I\big(y_i \neq G_m(X_i)\big)\}$$

For Arc-X4 (the name of the arcing algorithm), the whole series of model plays a role:

- Note $r_m(i)$ the number of times that i was misclassified by models $G_1, \dots, G_m$
- The weights are equal: $\alpha_m = 1/M$
- The bootstrap weights are

$$w_i^{(m+1)} = \frac{1 + r_m(i)^4}{\sum_{k=1}^{m}\{1 + r_k(i)^4\}}$$

Strengths of boosting:

- Typically provides excellent predictive performance,
- Seamless treatment of missing values,
- No need to transform the variables,
- Takes account of interactions implicitly,
- Robust to extreme values.

Weaknesses of boosting:

- Impossible to interpret (black box),
- Computationally hungry (to fit and to make predictions),
- Tendency to overfit.

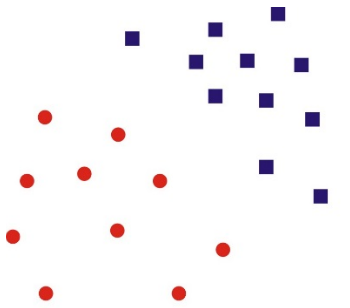
There exist multiple other options for boosting. You will see more details, e.g., on gradient boosting in the advanced data mining course.

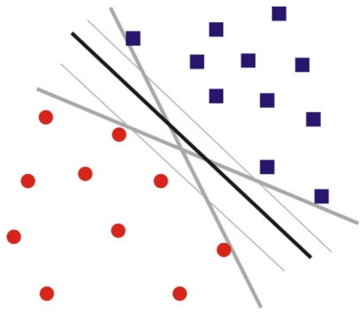
Support Vector Machines

Support vector machines are used for classification.

They are appropriate for categorical variables (including binary).

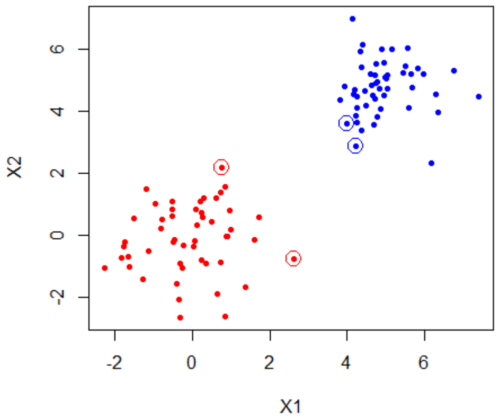
If data are separable, SVM will choose the plan that splits the groups with the largest margin among all possible options:

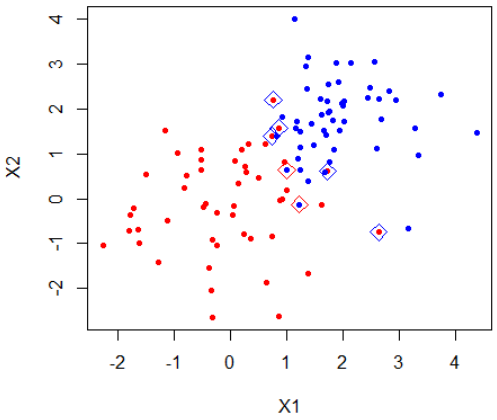




The "support vectors" are, in fact, the data points that define the optimal (hyper-) plan. They are the points closest to the separating plane.

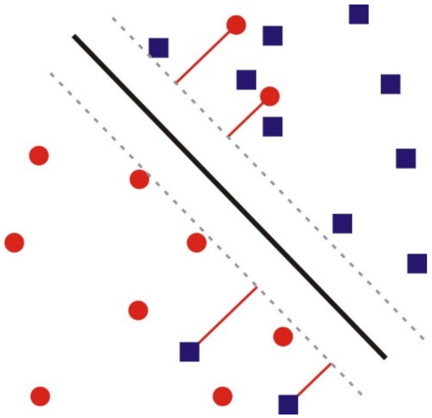
For instance, the support vectors of these datasets have been highlighted





On the right, we see the support vectors, but note that there is no separation between the two groups.

When data are not separable, a “budget” is provided to allow individuals to cross some distance over the border:



The optimization problem maximizes the margins, but a penalty is imposed with a cost on the distance between wrongly classified items and their distance to the separation plane.

The cost is a smoothness parameter akin to the width of a kernel. Indeed a lower cost will provide a smoother solution as wrongly classified individuals will pay a small premium even if they are quite far on the wrong side.

To be more specific, if $\boldsymbol{x}$ is the vector of covariates available, then

- a hyperplane may be defined as $\boldsymbol{w}^T\boldsymbol{x} + b = 0$
- $\boldsymbol{w}$ is a vector, b is a constant
- A classifier may be written as $\text{sign}(\boldsymbol{w}^T\boldsymbol{x} + b)$

Finding the best hyperplane when it is possible to separate the data is then equivalent to solving:

$$\begin{array}{ll} \text{minimize} & \boldsymbol{w}^T\boldsymbol{w}/2 \\ \text{such that} & y_i(\boldsymbol{w}^T\boldsymbol{x_i} + b) \geq 1, \forall i \end{array}$$

In other words, the condition means that all the points are well classified. With soft margins, the problem becomes:

$$\begin{array}{ll} \text{minimize} & \boldsymbol{w}^T\boldsymbol{w}/2 + c\sum_{i=1}^{n}\varepsilon_i \\ \text{such that} & y_i(\boldsymbol{w}^T\boldsymbol{x_i} + b) \geq 1 - \varepsilon_i \text{ and } \varepsilon_i > 0, \forall i \end{array}$$

- The parameter c is a penalty that controls the flexibility of the model.
- A large c may lead to overfitting.
- The dual problem for these programs is a function of the scalar products of the x_i.

In very simplified terms: in optimization, the dual problem is looking at the same problem from a different perspective, offering a different wasy to solve it. For instance:

- I want to find the winner of the 100m (lowest time),
- I look for the runner with the highest average speed instead.

Sometimes, one problem is hard and the other is easy even is they may be equivalent.

For SVM, the "Lagrangian dual" is:

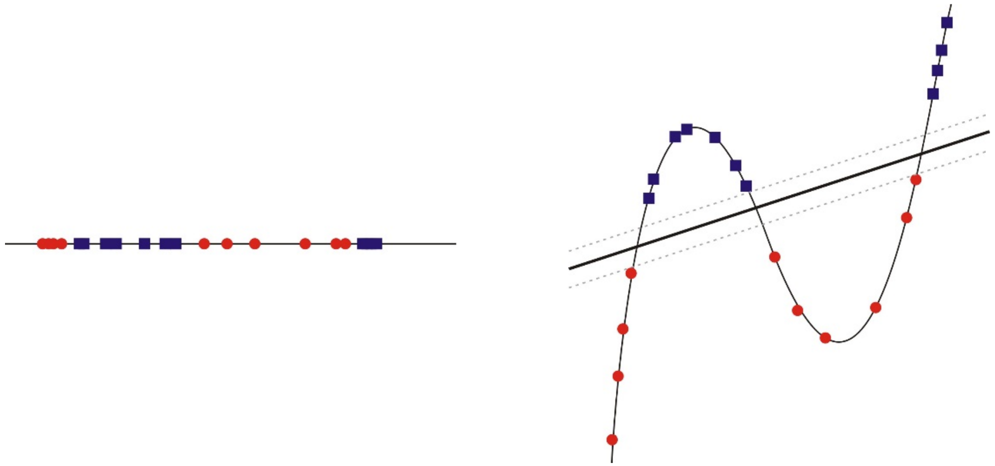
$$\begin{aligned} &\text{maximize} \quad \sum_{i=1}^{n} c_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i c_i (\boldsymbol{x_i}^T \boldsymbol{x_j}) y_j c_j \\ &\text{such that} \quad y_i(\boldsymbol{w}^T \boldsymbol{x_i} + b) \geq 1 - \varepsilon_i \text{ and } 0 \leq c_i \leq c, \forall i \end{aligned}$$

It is a quadratic problem subject to linear constraints (easy to solve efficiently). The link between the c_i and the $\boldsymbol{w}$ is

$$\boldsymbol{w} = \sum_{i=1}^{n} y_i c_i \boldsymbol{x_i}$$

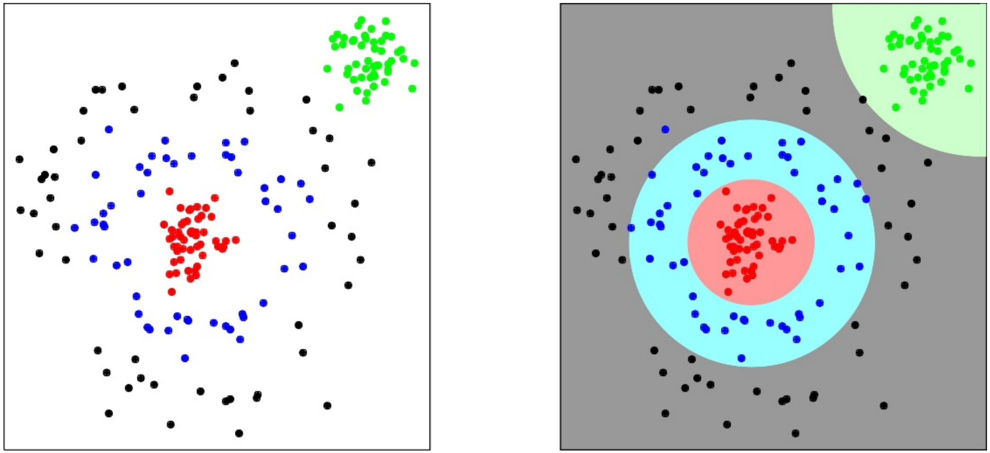
Their “kernel trick” is quite nice. Data may not be separable in their original space, but adding additional dimensions may make them separable.

For instance, going from x to polynomials of degree 3 makes this dataset separable:



We may get all kinds of nonlinear shapes even if the separation planes do remain linear!

For instance, with a radial kernel, these four groups get well separated:



Kernel Trick

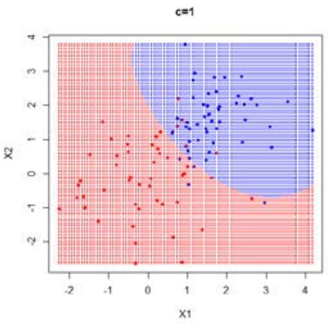
In its dual form, the maximization problem is a function of a scalar product.

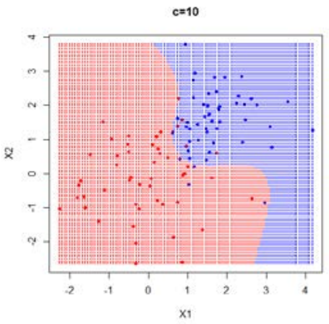
Replacing it with a kernel allows us to expand the problem to nonlinear spaces while still solving a quadratic problem with linear constraints. The following kernels are popular:

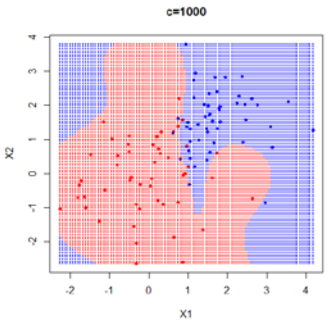
- Linear kernel: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^T \boldsymbol{x}_j$
- Polynomial of power p: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = (1 + \boldsymbol{x}_i^T \boldsymbol{x}_j)^p$
- Radial basis functions: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(-p\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2\right)$
- Sigmoid: $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tanh(\beta_1 \boldsymbol{x}_i^T \boldsymbol{x}_j + \beta_0)$

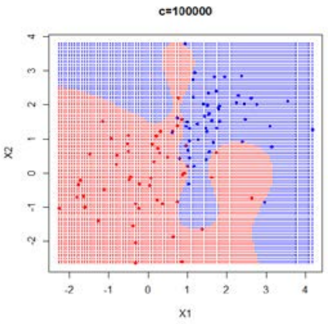
And they allow to expand the space in which the features can live.

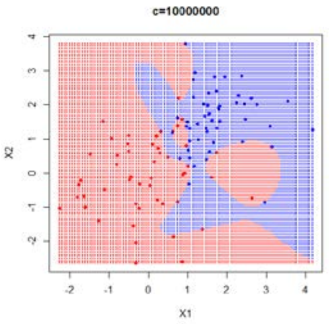
Overfitting with SVM











Choosing values of c that are too big may lead to overfitting.

The best c may be chosen by validation (cross- or out-of-bag).

Strengths of SVM:

- Can capture nonlinear phenomena,
- Sometimes feature excellent predictive performance,
- Simple optimization problem,
- Elegant kernel trick,
- Works well with a multinomial outcome.

Weaknesses of SVM:

- Impossible to interpret (black box),
- Hard to choose the right parameters (choice of kernel + value of c),
- Computationally hungry,
- Risk of overfitting,
- Does not work for a continuous target variable,
- Does not provide probabilities as an output (a function of the distance to the boundary can give a proxy), just the best separation.

Example: Converting sounds into features

Before deep learning was around, the strategy to analyze sound (as well as images) was to define manually some features that would then be measured on the data.

For images, features can be the intensity of the:

- Presence of vertical lines
- Presence of horizontal lines
- Presence of shadow
- Presence of specific colors / spectrums

This approach is still used in some contexts such as "remote sensing" (analysis of satellite images or images taken from planes).

For sounds, a similar approach is taken with the magnitude of certain frequencies, the presence of some rhythms, etc.

A dataset was built from 1856 mp3 samples from Garage Band, each converted in a few dozen characteristics.

The unstructured sound is hence transformed into a flat-file! Can SVM classify the music according to their style? Here are the predictions on the validation set:

		Prévision								
		alt	**blues**	**elec**	**folk**	**funk**	**jazz**	**pop**	**rap**	**rock**
Vérité	***alternative***	**3**	0	1	0	0	11	0	3	21
	blues	0	**6**	1	0	0	8	0	2	13
	electronic	4	1	**5**	0	0	7	0	6	5
	folkcountry	0	0	0	**61**	1	0	0	0	0
	funksoulrnb	0	0	0	7	**10**	0	0	0	0
	jazz	2	8	2	0	0	**59**	0	7	12
	pop	2	2	0	0	0	9	**2**	4	17
	raphiphop	0	6	6	0	0	4	0	**59**	5
	rock	1	5	4	0	0	9	0	7	**92**

Good classification: 297/500 = 59.4%.

Imbalanced (categorical) data

In some classification problems, we expect rare occurrences of “ones”:

- Fraud detection
- Failure of a system
- Security breach
- Response to a marketing campaign

Classification algorithms often perform better when the different groups are balanced. Rare classes can cause some trouble.

- Consider a dataset with only 1% of ones. The model that predicts 0 for everyone has a 99% rate of good classification, yet it is useless.
- Choosing the model with the lowest misclassification rate may not work.
- Logistic regression may have trouble converging with a rare class.
- Small relative numbers of ones can cause numerical challenges to all methods.

What solutions can help in these cases?

Minimizing cost

Even if the costs of making errors are unknown, it is possible to attribute a higher cost to missing (rare) ones than missing (common) zeros.

- Attributing a larger value to the rare class will boost its importance
- The costs chosen are arbitrary
- You may try many options and see which performs best

If the class is so rare that numerical issues arise, this may not solve it, but it will lead to potentially better solutions.

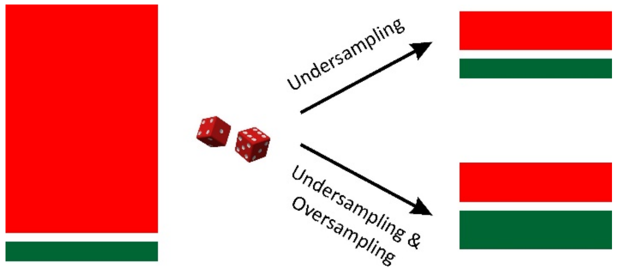
Custom algorithms

Custom algorithms that are designed for rare classes have been developed. *Rareboost* is one such example, but it does not seem readily available for R.

Undersampling / Oversampling

One option is to rebalance the dataset by:

- Undersampling (only a subset of the common class is kept)
- Oversampling (drawing from the rare class to increase its sample size)



It is not necessary to aim at a perfectly balanced dataset (no need to reach 50-50).

- Undersampling means that some information is ignored,
- Forcing 50-50 means more information is lost,
- The ideal fraction to aim for is unknown.

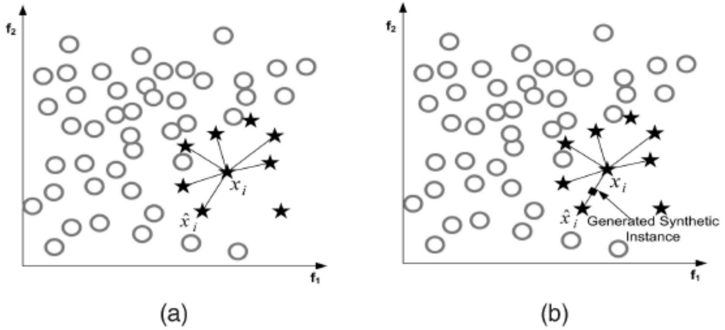
There are strategies for undersampling that attempt to keep points that are closer to the boundaries where we need to tell ones from zeroes.

- They make sense in a pure classification setting
- If we are interested in probability estimates, then samples that represent the population are preferable (just do simple random sampling).

Be aware that oversampling:

- Generates ties,
- Does not create new information.

Some methods have been proposed to oversample without creating ties, for instance
SMOTE (Synthetic Minority Oversampling Technique, (see, e.g., He H et Garcia E.A. (2009). Learning from imbalanced data. IEEE Transactions on knowledge and data engineering. Vol. 21, No.9, 1263-1284)



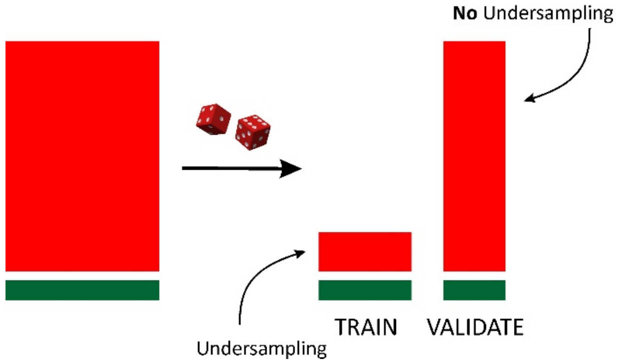
Effects of under/oversampling

When data are under/oversampled to become better balanced:

- Predicted probabilities change,
- Classification rates change,
- Methods are more stable and may better perform.

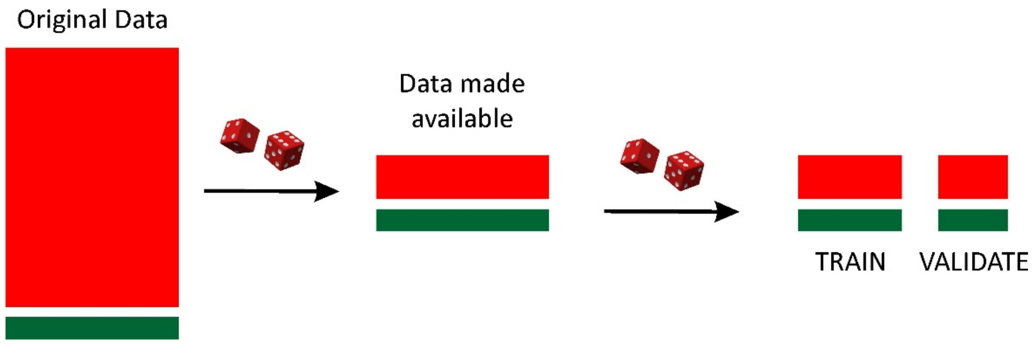
If you control the undersampling, you may apply it <u>only to your training set</u>. Then:

- The predicted probabilities will not be meaningful.
- A confusion matrix calculated on the validation set, as well as all performance measures that come from it, will represent the initial population (no need to rescale).



In that case, undersampling is just a trick to help the algorithm.

In some cases, you may already receive a dataset that has already been sampled.



As well as the information on the proportion of 0 and 1 in the original data (before over/under-sampling).

In that case, you may adjust the confusion matrix and relevant probabilities.

Note:

- π_0 and π_1 the proportions of 0 and 1 in the original data
- p_0 and p_1 the proportions of 0 and 1 in the data that was made available to you

Basic idea: each 0 in your data represents π_0/p_0 individuals, and each 1 weights π_1/p_1

Consider the home equity data. We had 80% of good payers, but suppose that, in fact, there are many more good payers and only 5% are really defaulting.

We have, in this case:

- $\pi_0 = 0.05$ and $\pi_1 = 0.95$
- $p_0 = 0.1995$ and $p_1 = 0.8005$

Yielding conversions factors:

$$c_0 = \frac{\pi_0}{p_0} = 0.2506 \quad \text{and} \quad c_1 = \frac{\pi_1}{p_1} = 1.187$$

A generic confusion matrix based on the rebalanced data,

		Prediction		
		0	1	Total
Truth	0	A	C	$A+C$
	1	B	D	$B+D$
Total		$A+B$	$C+D$	$A+B+C+D$

Then becomes:

		Prediction		
		0	1	Total
Truth	0	c_0A	c_0C	$c_0(A+C)$
	1	c_1B	c_1D	$c_1(B+D)$
Total		c_0A+c_1B	c_0C+c_1D	$c_0(A+C)+c_1(B+D)$

To represent the initial population.

Interestingly, sensitivity and specificity are unchanged (the factor c_0 or c_1) appears in both the numerator and denominator when you compute them. The rest gets adjusted.

With a 90% threshold, we had obtained:

		Prediction		
		0	1	Total
Truth	0	352	50	402
	1	878	720	1598
Total		1230	770	2000

But taking into account proportions in the original population:

		Prediction		
		0	1	Total
Truth	0	88.22	12.53	100.75
	1	1041.97	854.47	1896.44
Total		1130.19	867.00	1997.19

Misclassification rate: 52.80% (previously 46.40%)

Precision = P(truth is 1 | predict 1) = 98.55% (previously 93.51%)

Sensitivity = P(predict 1 | truth is 1) = 45.06% (previously 45.06%)

Specificity = P(predict 0 | truth is 0) = 87.56% (previously 87.56%)

Note that the total moved slightly from 2000. This is because:

- p_0 and p_1 come from the complete dataset, before the split in train-validation
- The confusion matrix is based on the validation set whose proportions of 0 and 1 are not exactly identical to the whole population due to the random split.

It is also possible to adjust probabilities. Whether you look at the leaf of a regression tree or at a single prediction for a given individual, you have estimated probabilities:

$\hat{p}_1$ that the individual is a 1 and $\hat{p}_0 = 1 - \hat{p}_1$ that it is a 0

Based on the rebalanced data set. Then in the original population, these probabilities correspond to:

$$\frac{c_1\hat{p}_1}{c_0\hat{p}_0 + c_1\hat{p}_1} \text{ and } \frac{c_0\hat{p}_0}{c_0\hat{p}_0 + c_1\hat{p}_1}$$

For instance, an individual with a predicted probability of 90% of being a good payer in the rebalanced data truly has a probability $\frac{1.187\cdot 0.9}{0.2506\cdot 0.1+1.187\cdot 0.9} = 0.977$ in the general population.

The same principles apply for classification with multiple categories.