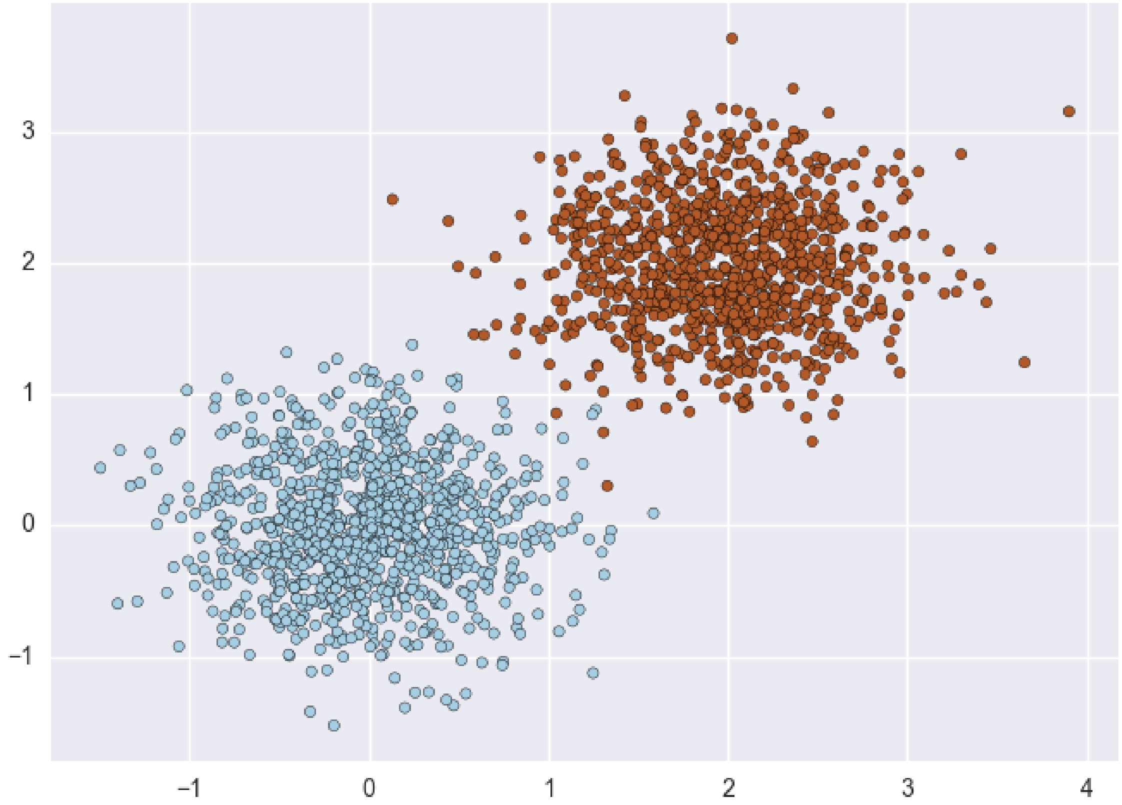
Building a classifier over imbalanced data

Sources
https://svds.com/learning-imbalanced-classes/
http://www.cs.pomona.edu/~dkauchak/classes/f13/cs451-f13/

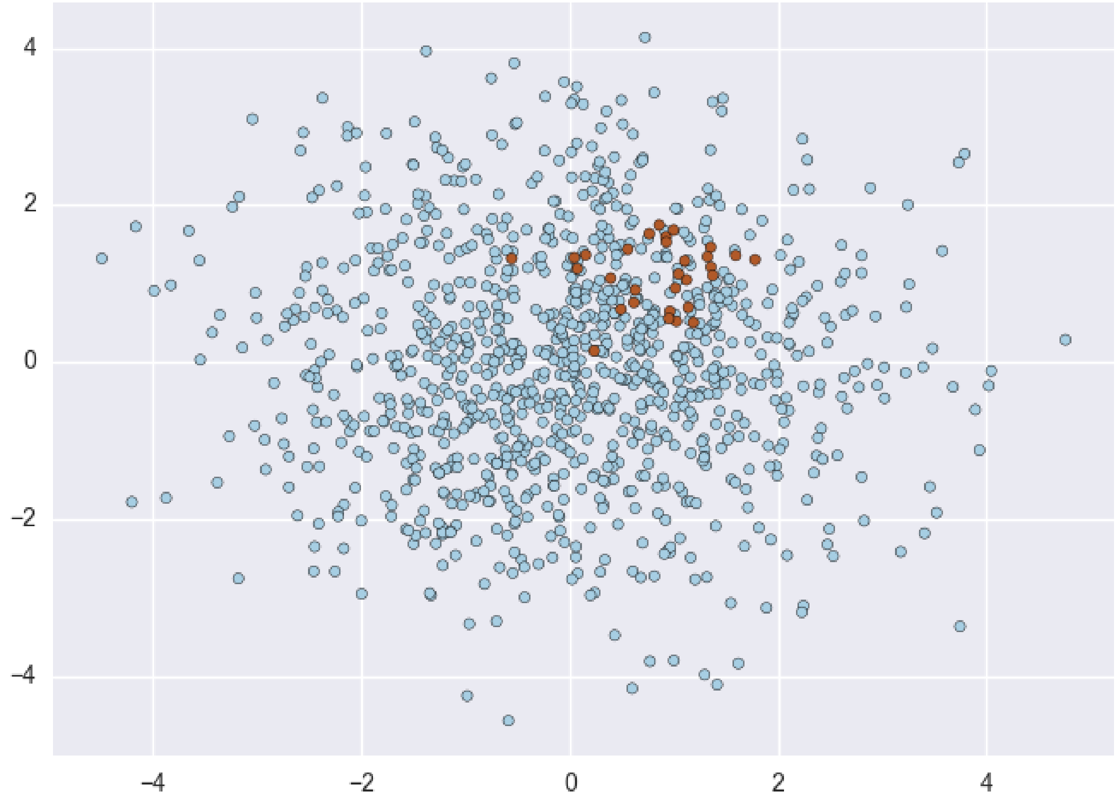
Imbalanced classes

- Most classification methods assume classes are reasonably balanced



Imbalanced classes

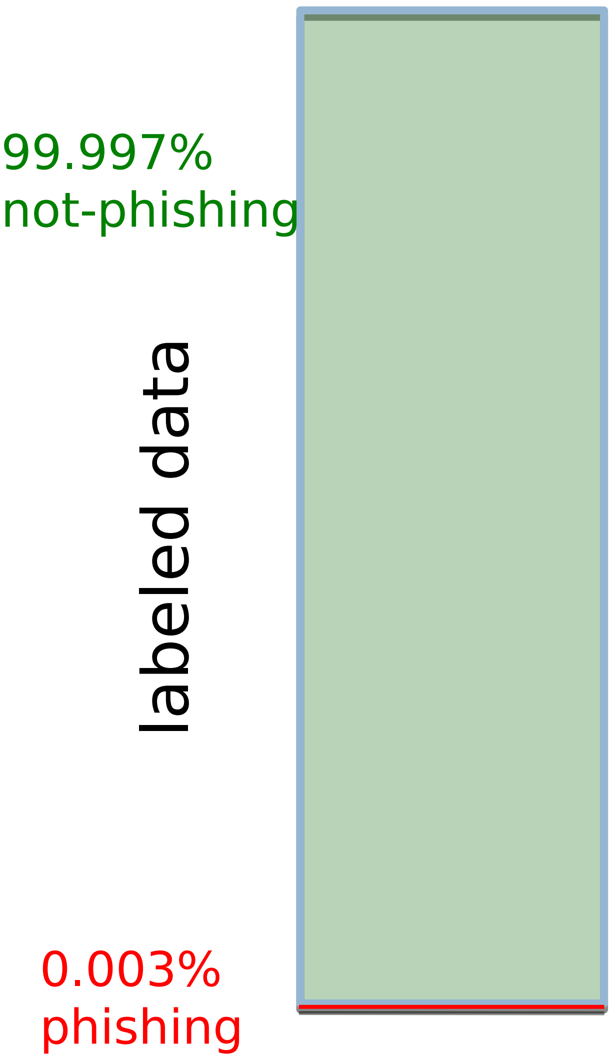
- In reality it is quite common to have a very popular class and a rare (yet interesting) one



Imbalanced classes

- Examples:
  - About 2% of **credit card** accounts are defrauded per year1. (Most fraud detection domains are heavily imbalanced.)
  - **Medical screening** for a condition is usually performed on a large population of people without the condition, to detect a small minority with it (e.g., HIV prevalence in the USA is ~0.4%).
  - **Disk drive failures** are approximately ~1% per year.
  - The **conversion rates** of online ads has been estimated to lie between 10-3 to 10-6.
  - **Factory production defect** rates typically run about 0.1%.

Imbalanced data



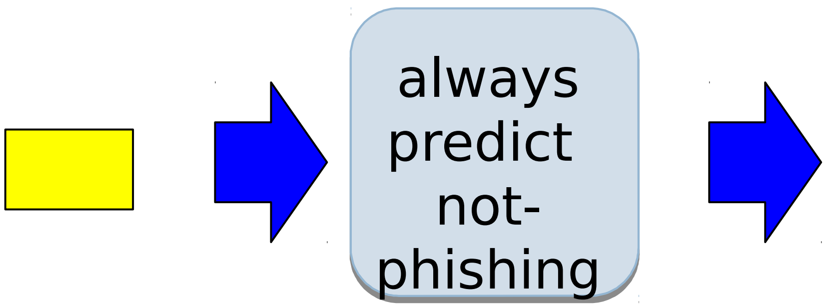
The phishing problem is what is called an **imbalanced data** problem

This occurs where there is a large discrepancy between the number of examples with each class label

e.g. for our 1M example dataset only about 30 would actually represent phishing e-mails

What is probably going on with our classifier?

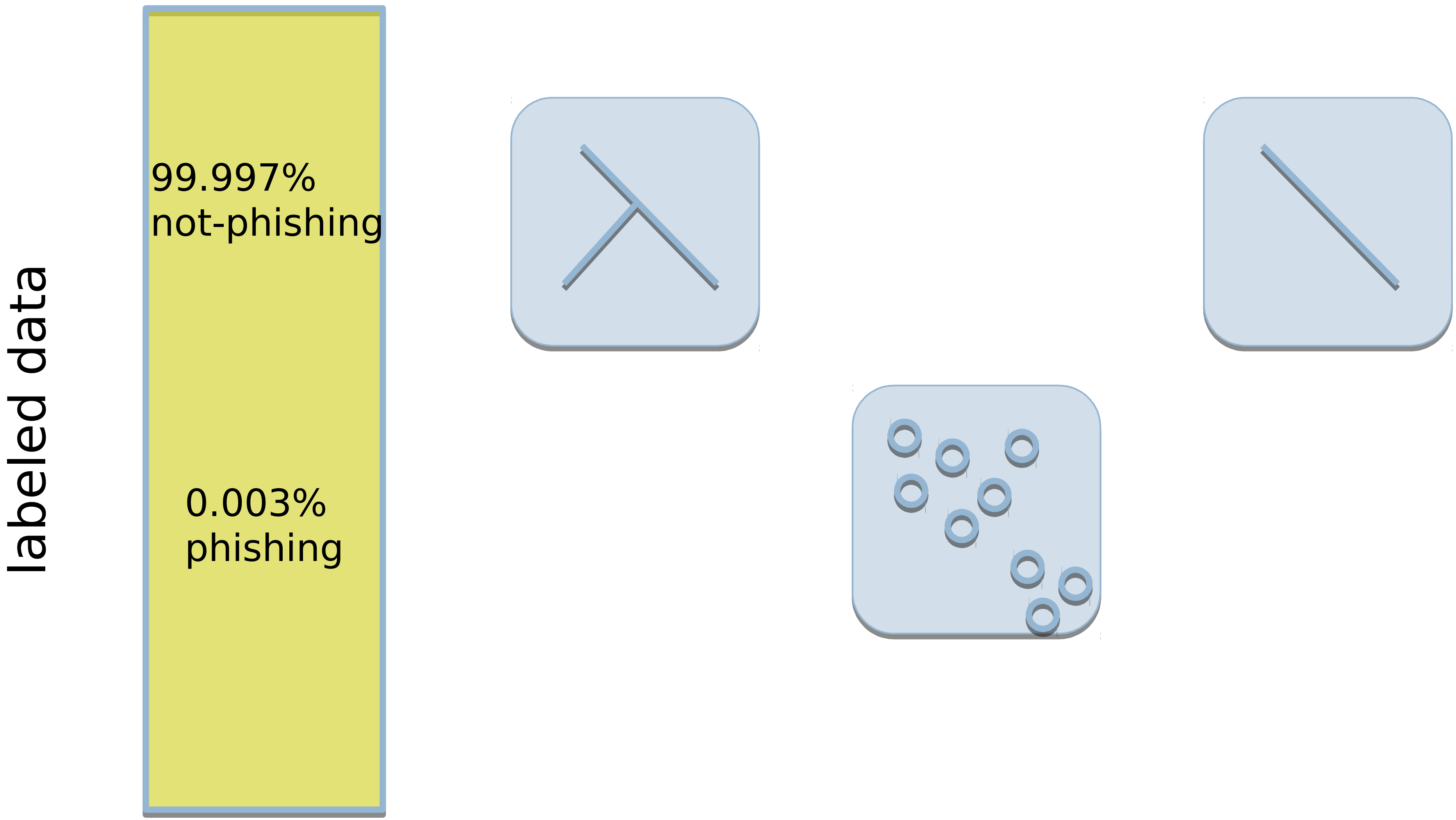
Imbalanced data



99.997% accuracy

Why does the classifier learn this?

Imbalanced data: current classifiers



How will our current classifiers do on this problem?

Imbalanced data: current classifiers

All will do fine if the data can be easily separated/distinguished

Decision trees:

- explicitly minimizes training error
- when pruning pick “majority” label at leaves
- tend to do very poor at imbalanced problems

k-NN:

- even for small k, majority class will tend to overwhelm the vote

perceptron:

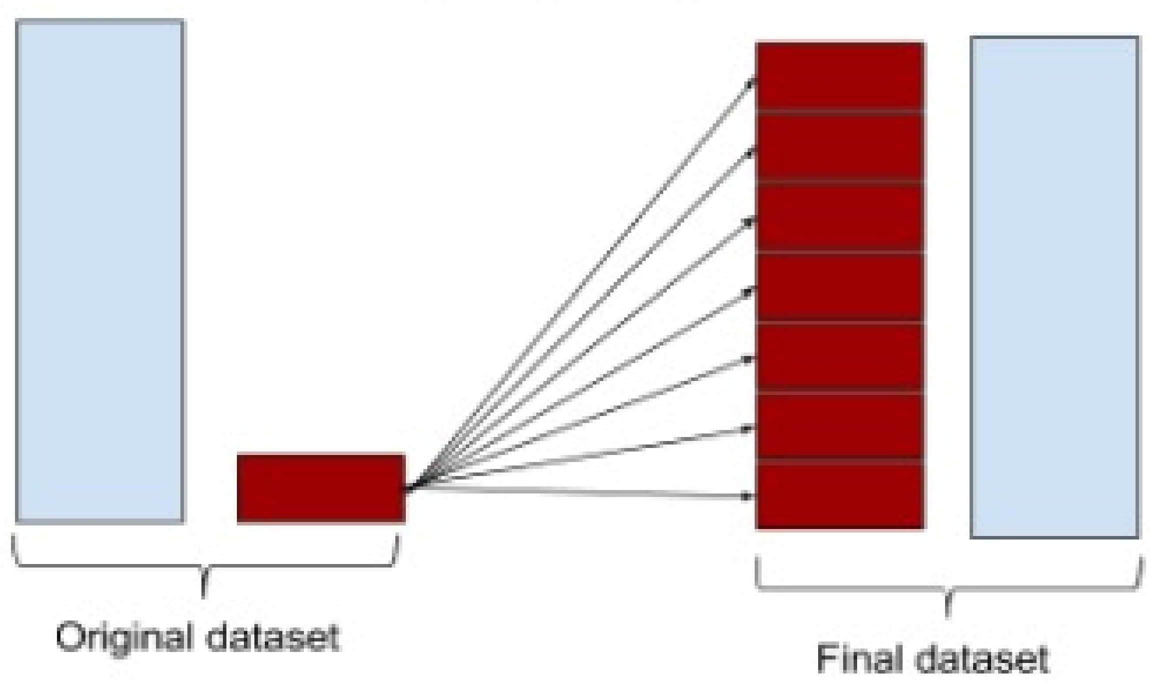
- can be reasonable since only updates when a mistake is made
- can take a long time to learn

Handling imbalanced data

- Possible alternatives
  - Do nothing and hope to be lucky
  - Balance the training set in some way:
    - Oversample the minority class
    - Undersample the majority class
    - Synthesize new minority classes
  - Throw away minority examples and switch to an anomaly detection framework
  - At the algorithm level:
    - Adjust the class weight (misclassification costs)
    - Adjust the decision threshold
    - Modify an existing algorithm to be more sensitive to rare classes
  - Construct an entirely new algorithm to perform well on imbalanced data

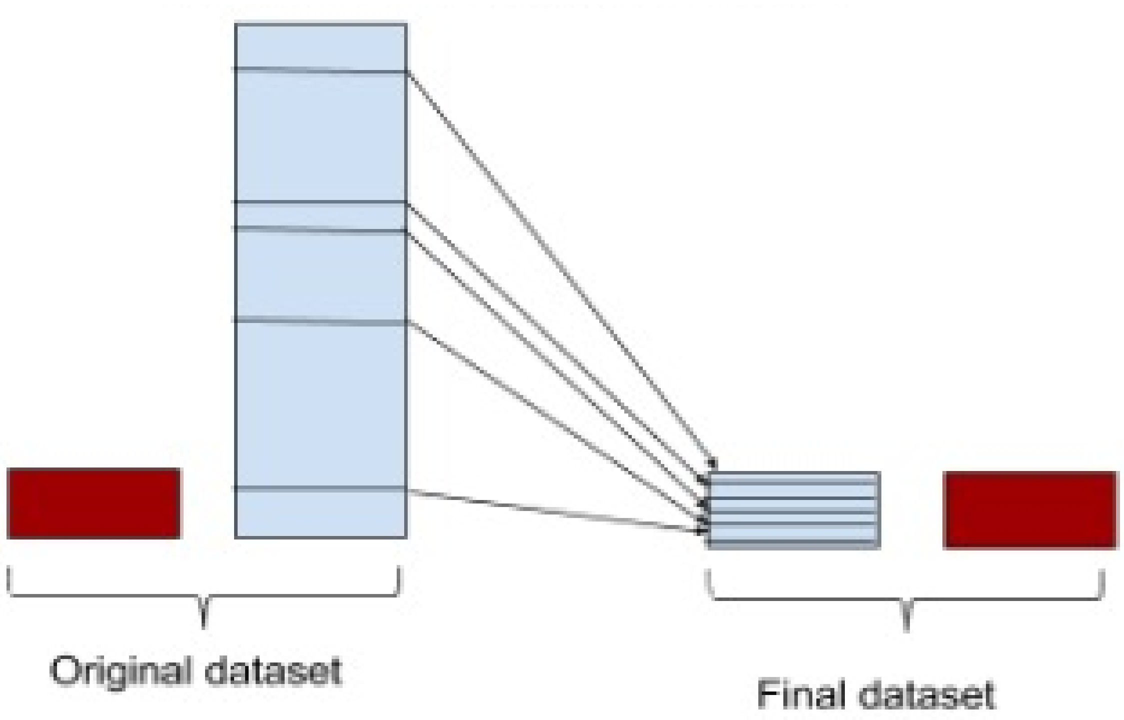
Balancing the dataset

- Oversampling the minority class



Balancing the dataset

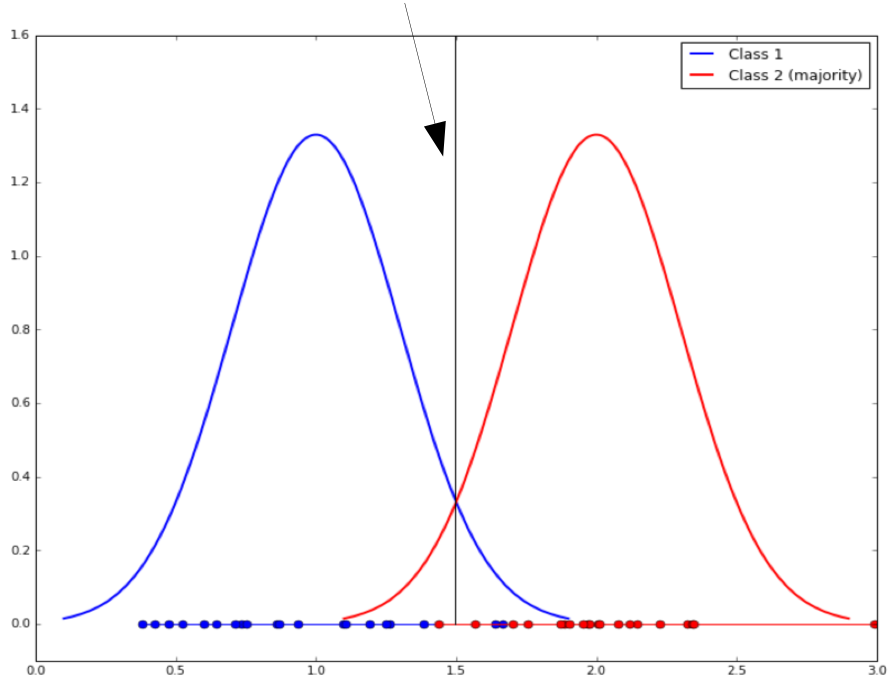
- Undersampling the majority class



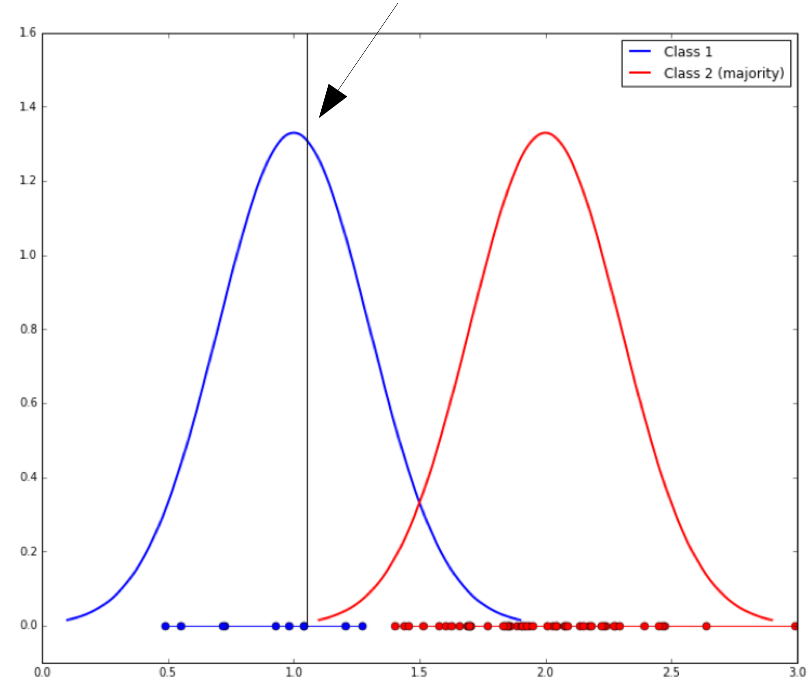
Balancing the dataset

- Undersampling the majority class
- Bayesian argument (Wallace et al., ICDM 2011)

Separation line for 1-d,
balanced data



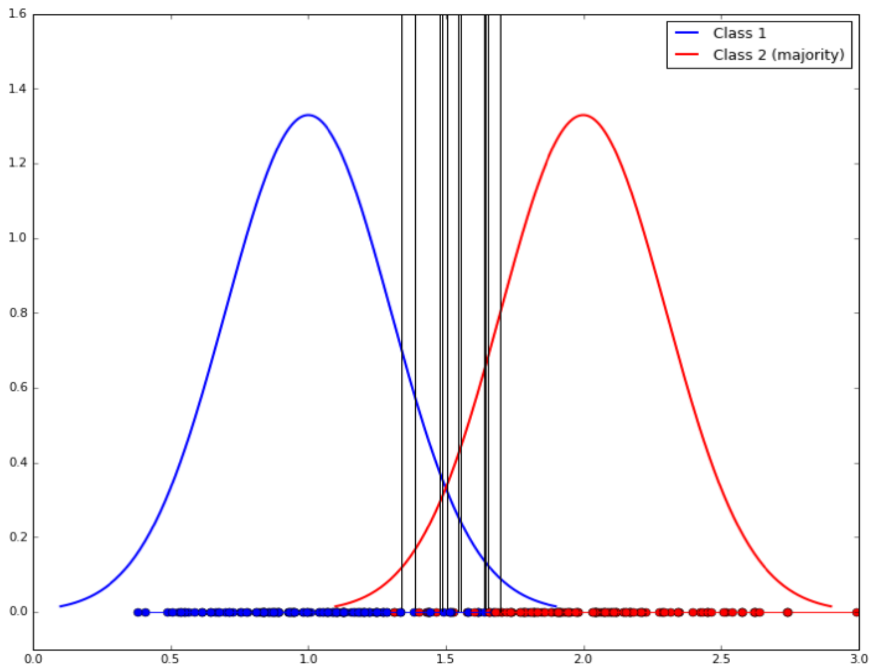
Separation line for
imbalanced data



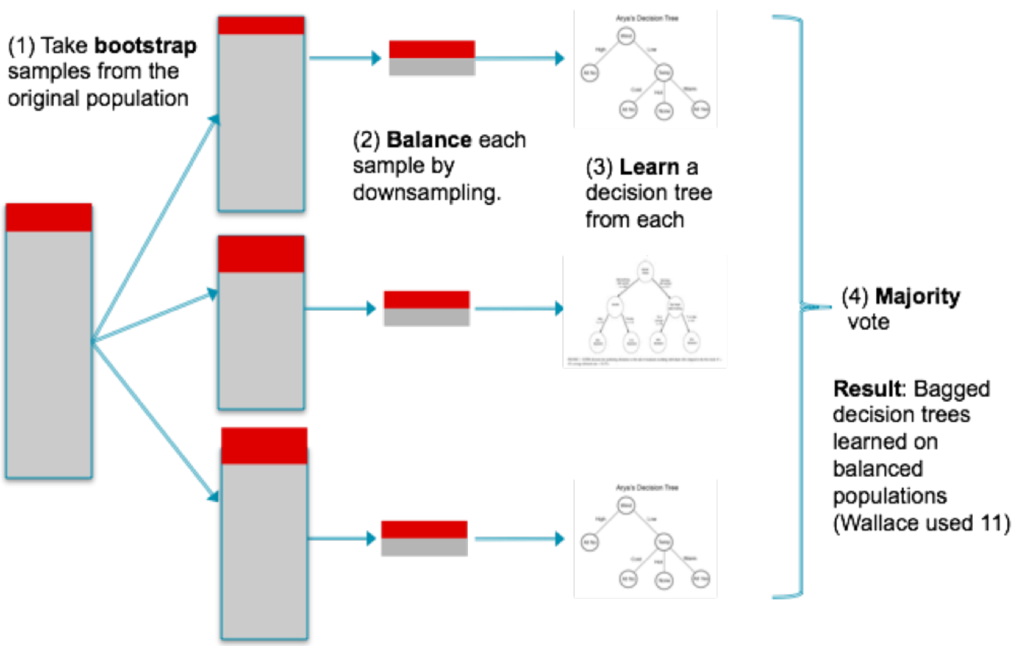
Balancing the dataset

- Undersampling the majority class
- Bayesian argument (Wallace et al., ICDM 2011)

Separation lines with Undersampling (10 examples)

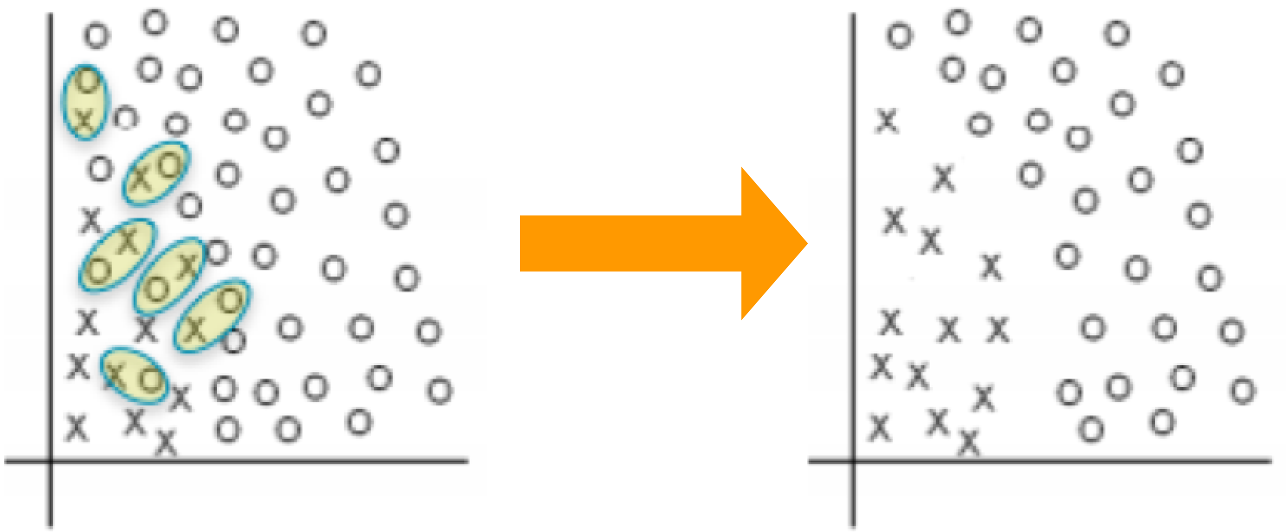


Can apply a bagging approach



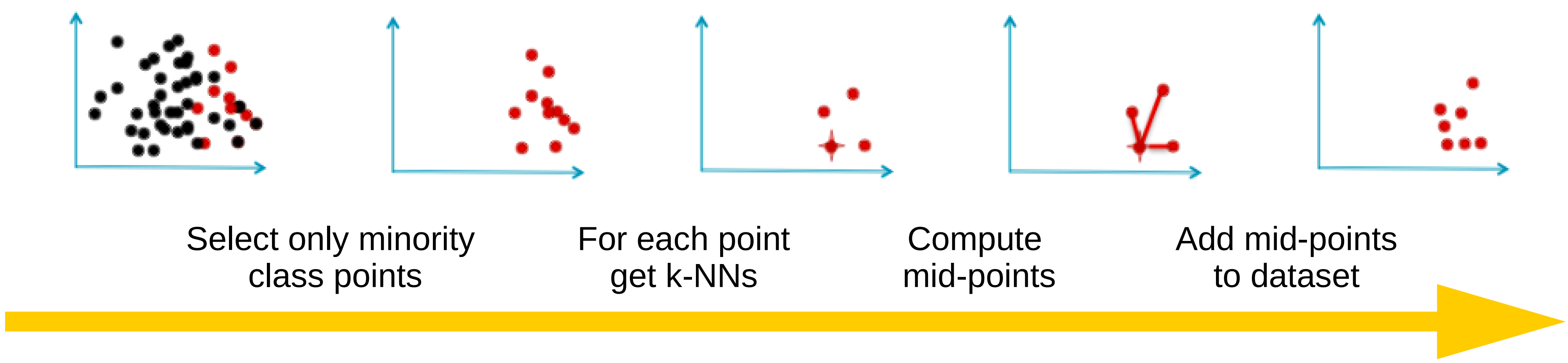
Balancing the dataset

- Smart undersampling
  - Remove some majority class points
  - Neighbor-based approaches, e.g. *Tomek links*
    - Remove majority points having as NN a minority point



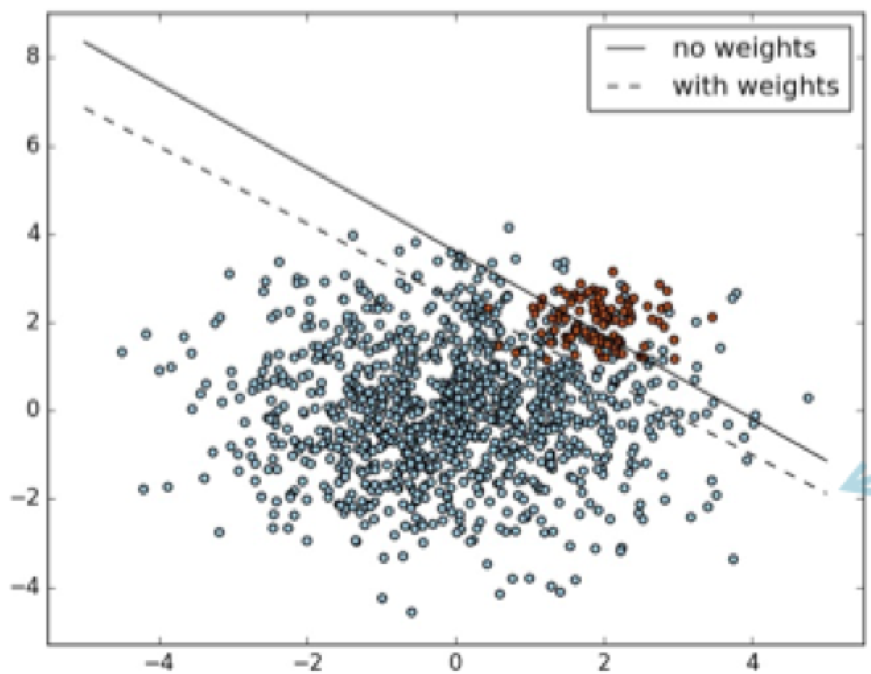
Balancing the dataset

- Smart oversampling
  - Add some minority class points
  - E.g. SMOTE (Synthetic Minority Oversampling Technique)
    - Add points through interporlation



Adjusting class weights

- Example from Python scikit-learn
  - Some classifiers have a “class_weight” parameter




```
import numpy as np
import pylab as pl
from sklearn import svm

# we create 40 separable points
rng = np.random.RandomState(0)
n_samples_1 = 1000
n_samples_2 = 100
X = np.r_[1.5 * rng.randn(n_samples_1, 2),
          0.5 * rng.randn(n_samples_2, 2) + [2, 2]]
y = [0] * (n_samples_1) + [1] * (n_samples_2)

# fit the model and get the separating hyperplane
clf = svm.SVC(kernel='linear', C=1.0)
clf.fit(X, y)

w = clf.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(-5, 5)
yy = a * xx - clf.intercept_[0] / w[1]

# get the separating hyperplane using weighted classes
wclf = svm.SVC(kernel='linear', class_weight={1: 10})
wclf.fit(X, y)

ww = wclf.coef_[0]
wa = -ww[0] / ww[1]
wyy = wa * xx - wclf.intercept_[0] / ww[1]

# plot separating hyperplanes and samples
h0 = pl.plot(xx, yy, 'k-', label='no weights')
h1 = pl.plot(xx, wyy, 'k--', label='with weights')
pl.scatter(X[:, 0], X[:, 1], c=y, cmap=pl.cm.Paired)
pl.legend()

pl.axis('tight')
pl.show()
```

Related topic: evaluating classifiers on imbalanced data

- When classes are **slightly** imbalanced, no balancing is need
- Yet, take that into consideration when evaluating performances
- E.g.: Assume the test set contains 100 records

  Positive cases = 75, Negative cases = 25

  - Is a classifier with 70% accuracy good?
  - No, the trivial classifier (always positive) reaches 75%

  Positive cases = 50, Negative cases = 50

  - Is a classifier with 70% accuracy good?
  - At least much better than the trivial classifier
- Take-home message
  - accuracy scores should be compared against some baseline classifier, e.g. Majority class classifier or a simple-yet-not-trivial one

Similar situation: multiclass problems

- Assume N classes
- If classes are perfectly balanced, a trivial classifier (e.g. majority) will yield $A_{trivial}$ ~100/N % accuracy
  - N=2 → $A_{trivial} \sim 50\%$
  - N=4 → $A_{trivial} \sim 25\%$
- Goodness of accuracy of a model should be compared against $A_{trivial}$
  - If N=5, an accuracy of 40% would look large

Again on evaluation: scoring/ranking vs. classifying

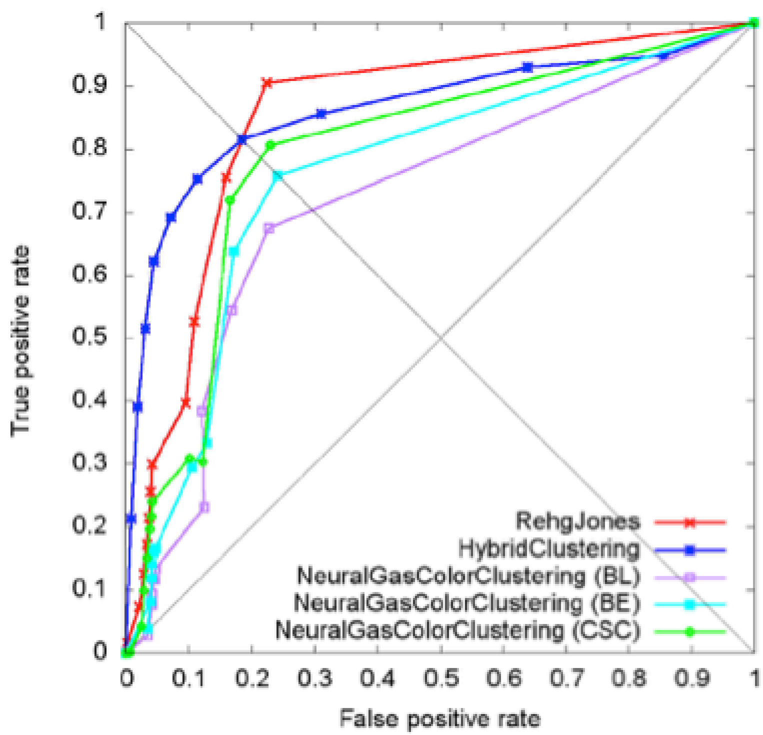
- Two slightly different objectives
  - Classifying = assigning a record to a class
  - Scoring/ranking = assigning **probabilities** of belonging to a class
- Several classification methods compute scores, and then assign class
  - Score p > 50% → class = Y
  - Otherwise → class = N
- E.g.: decision trees have p = #positive/#negative cases over each leaf

Again on evaluation: scoring/ranking vs. classifying

- What if we generalize the schema into:
  - Score p > X% → class = Y
  - Otherwise → class = N
- For each X (in [0-100]) we get a different set of predictions
  - The confusion matrix changes
  - All indicators derived from it change
    - Accuracy
    - TPR
    - TNR
    - ...

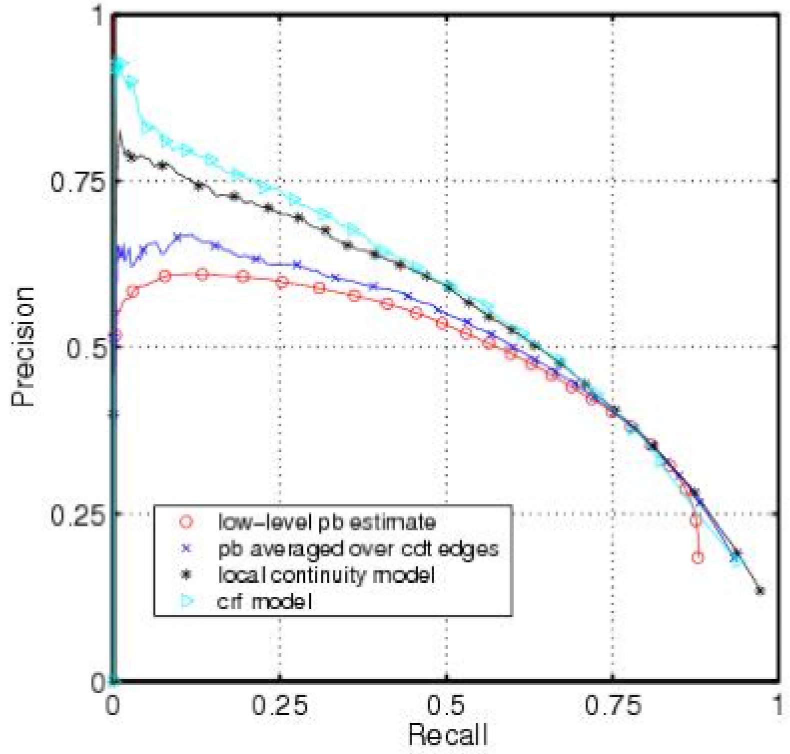
Again on evaluation: scoring/ranking vs. classifying

- Deeper insights on our model can be obtained looking at how performances change with X
  - ROC curve: plots TPR vs. FPR



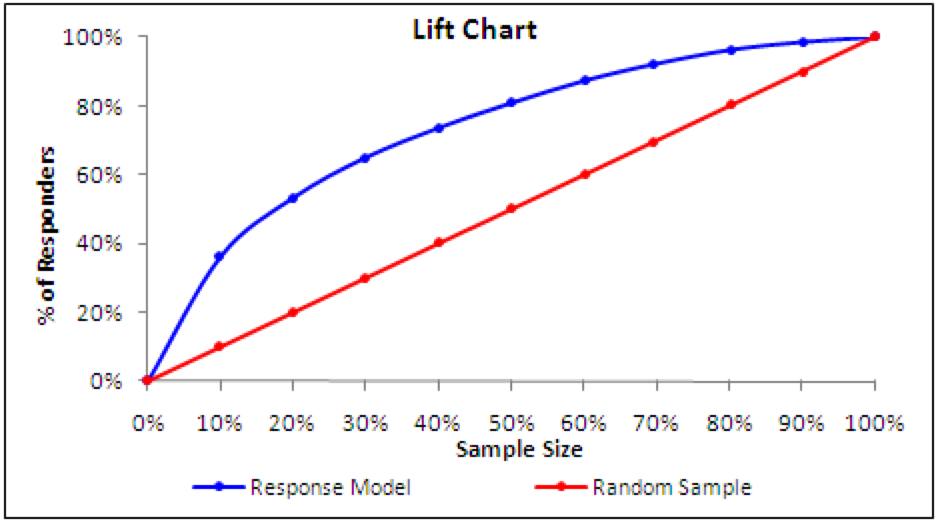
Again on evaluation: scoring/ranking vs. classifying

- Deeper insights on our model can be obtained looking at how performances change with X
  - Precision vs. recall



Again on evaluation: scoring/ranking vs. classifying

- Deeper insights on our model can be obtained looking at how performances change with X
  - Lift chart: % of positive cases vs. % of dataset classified as Y



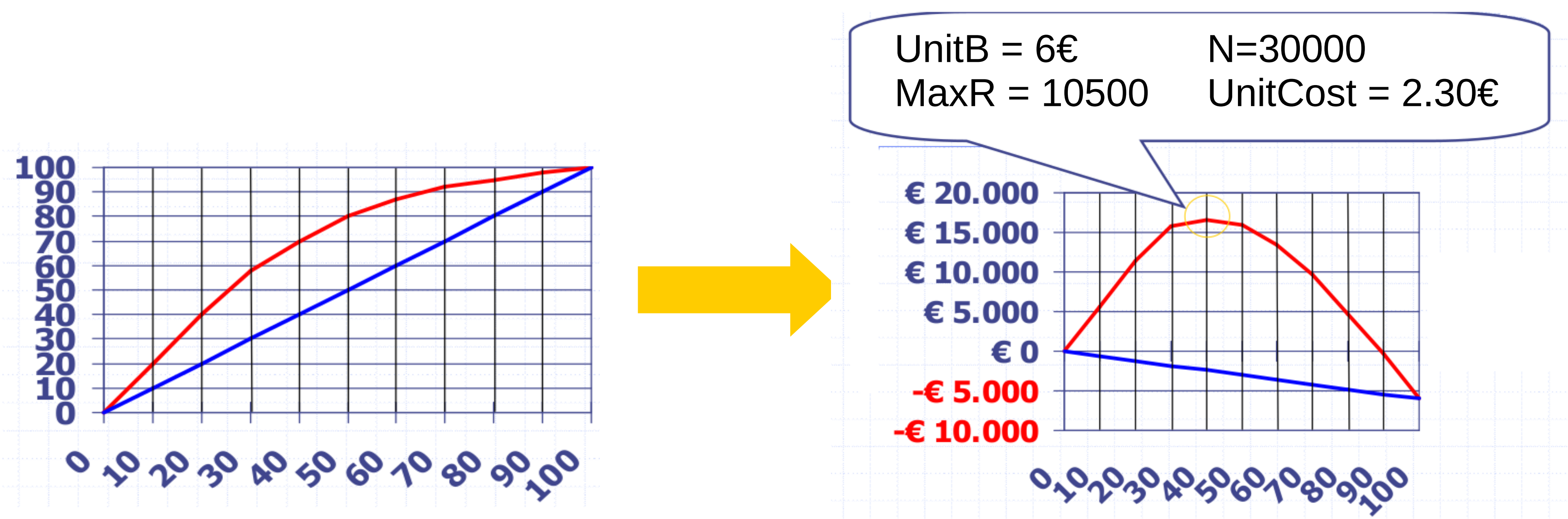
Notice: “Lift chart” is a rather general term, often used to identify also other kinds of plots. Don’t get confused!

Again on evaluation: Application example

- From Lift chart we can easily derive an “economical value” plot, e.g. in target marketing
  - Question: Given our predictive model, how many customers should we target to maximize income?
- Simple economical model
  - Profit = UnitB*MaxR*Lift(X) - UnitCost*N*X/100
    - **UnitB** = unit benefit, **UnitCost** = unit postal cost
    - **N** = total customers, **MaxR** = expected potential respondents in all population (N)
    - **Lift(X)** = lift chart value for X, in [0,..,1]

Again on evaluation: Application example

- From Lift chart we can easily derive an “economical value” plot, e.g. in target marketing
  - Question: Given our predictive model, how many customers should we target to maximize income?



Optimal X = 40%