June, 2014

AA ML Course – Theoretical Session 4

Assaf Hallak

Courtesy of Amitai & Tom

Advanced Analytics

We mine your business

Agenda

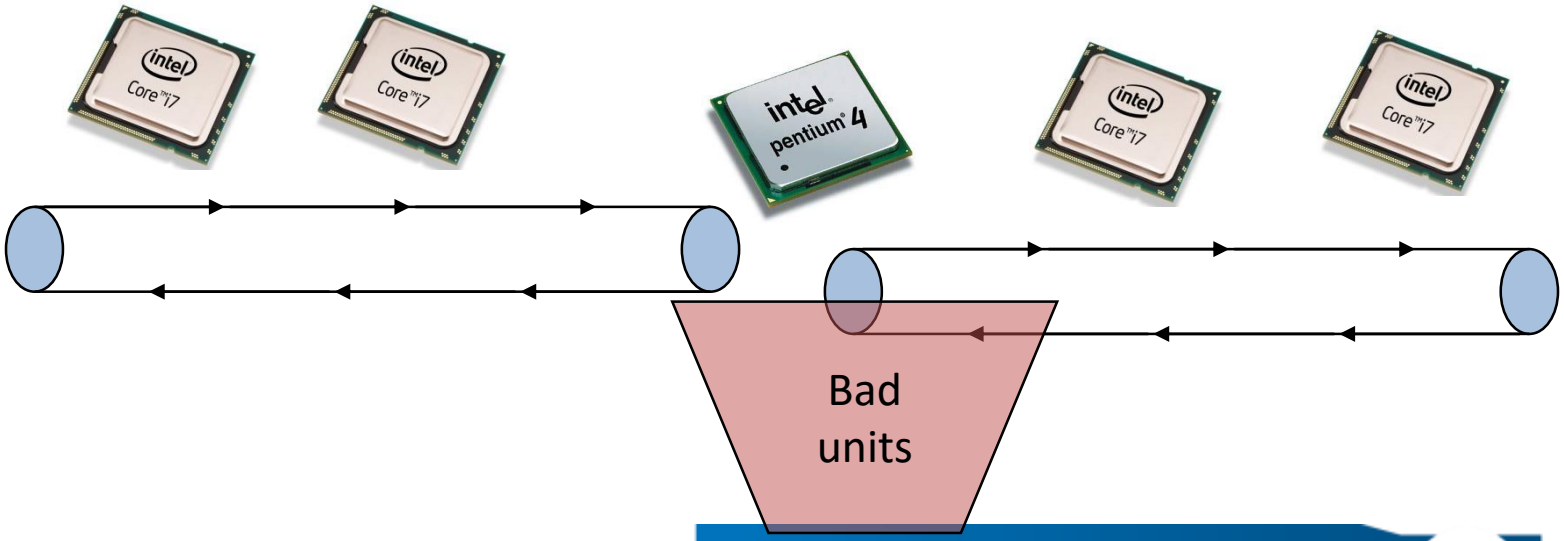
1. **Introduction**
2. Bayesian decisions
3. Naïve Bayes
4. Evaluation measures
5. K-Nearest Neighbors
6. Decision Trees
7. Support Vector Machines

Introduction

- Today we discuss prediction, mainly classification
- Classification/prediction as an optimal decision problem
- Some specific models to help us
- Different ways to evaluate ourselves

Motivating example: Classifying CPUs

- For each unit, we want to use multiple features (physical tests) to **decide** whether to discard it, or send it to market
  - Or more generally: Classify it to class ***C*** (i3/i5/i7/etc.)
- Inherent uncertainty – **probabilistic** decision
- Different decisions have different **costs**
- Formalize our problem to help us reach '**optimal**' choices



Additional Examples….

- Determine whether an email is Spam or legitimate (e.g. based on its words, sender domain)
- Determine if a Parkinson's patients disease is On/ Off, based on motion sensors data
- Determine whether a lawn mower/washing machine/refrigerator needs maintenance based on physical sensors (IOT)

Agenda

1. Introduction
2. **Bayesian decisions**
3. Naïve Bayes
4. Evaluation measures
5. K-Nearest Neighbors
6. Decision Trees
7. Support Vector Machines

Bayesian view in a (very small) nutshell

- We see **evidence X**, such as the CPU tests results
- We have **Prior** probabilities for having a bad CPU, e.g.:

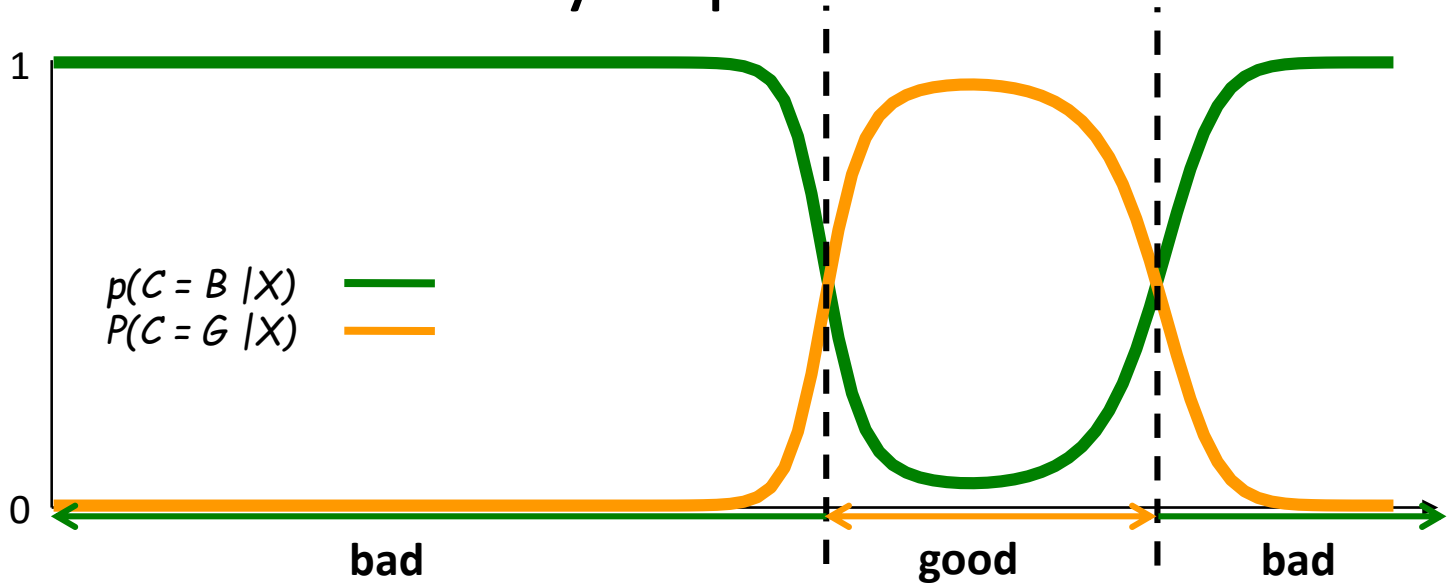
  P(C=good) = 0.99; P(C=bad) = 1-0.99 = 0.01
- We obtain the **Likelihood**: Probability of evidence, given each class, e.g.: *P(X | C= good) = 0.17*
- We compute **Posterior** probabilities: Probability of class, after seeing the evidence, e.g. *P(C=good | X)*
- Bayes rule:

posterior $P(C \mid X) = \frac{P(C)\, p(X \mid C)}{p(X)}$ *prior* *likelihood* *evidence*

, where $p(x) = \sum_c P(C)\, p(x|C)$

Classification as decision

- Our classification choice is a **decision rule,** where our **decided** '**action**' is one of {**G**ood,**B**ad}
- Our action usually depends on the evidence:



What is the Optimal Decision?

- We want optimal decisions - but optimal by what criteria?
- Need to define our costs or **loss function,** $L(C, a)$ **:** What loss we incur for prediction a when the class is C
- **0-1 loss** (misclassification loss):

Predicted/Actual	*Bad*	*Good*
Bad	0	1
Good	1	0

- Other/better appropriate losses for our CPU problem?

Optimal Decisions – Bayes decision rule

- By optimal, we mean the action that minimizes the **expected loss**
- In the Bayesian approach to decision theory, *the **optimal action** **having observed x*** is defined as the **action *a*** that minimizes the **expected loss given the data**

$$\sum_{C} L(C, a)\, p(C|x)$$

- *A decision policy that minimizes the above quantity for each X, is called* a **Bayes decision rule**
- Explanation: When ***C=good*** and we classify it as ***a=bad***, we may lose a lot (depending on our loss function). But, **the probability** of getting C = good **given the features x** could be very low… so the expected loss doesn't give much **weight** to this particular error

Bayes decision rule – 0-1 Loss

- Let's see this for **0-1 loss**
- If we decide predicted = 1 ("good"):

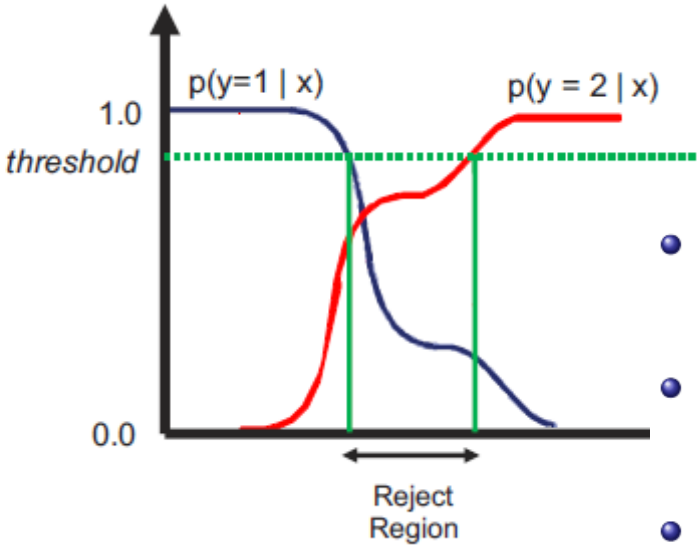
$$Loss(real = 0, pred = 1)\,\mathrm{P}(real = 0|x) + Loss(real = 1, pred = 1)\,\mathrm{P}(real = 1|\boldsymbol{x})$$
$$= P(real = 0\ and\ pred = 1|x)$$

- If we decide predicted = 0 ("bad"):

$$Loss(real = 0, pred = 0)\,P(real = 0|x) + Loss(real = 1, pred = 0)\,\mathrm{P}(real = 1|x)$$
$$= P(real = 1\ and\ pred = 0|x)$$

- **So our total expected loss for predicting '*pred*' is** $\boldsymbol{P(real \neq pred\ |x)}$
- To **minimize** the above, we simply choose *prediction* that **maximizes** $P(real = pred\ |x)$. This is the **MAP classification rule** (Maximum A-Posterior)
- The misclassification loss is minimized by selecting the class having the **largest posterior probability** (probability given the features X)

Reject option + other Loss Functions



- **0-1 Loss**: gives 0 loss when arguments agree, 1 otherwise

$$\mathcal{L}_{0-1}(x, y) = \mathrm{I}[x \neq y]$$

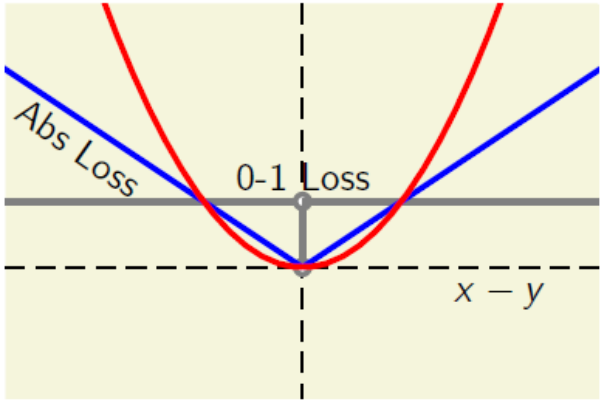
- **Absolute Loss**: gives the absolute difference in x & y

$$\mathcal{L}_{abs}(x, y) = |x - y|$$

- **Squared Loss**: gives the squared difference in x & y

$$\mathcal{L}_{sq}(x, y) = (x - y)^2$$

For some regions in input space, where the class posteriors are too low, we may prefer to say "don't know"



Bayesian decision theory: Summary

- Formulating as Bayesian decision problem tells us how to make optimal decisions, in terms of minimal risk
- But, where do we get all the 'components' to make the decision? Prior probabilities, conditional probabilities, etc…
- We need to construct models to learn these components **from data**, and combine **prior** domain knowledge if available

Agenda

1. Introduction
2. Bayesian decisions
3. **Naïve Bayes**
4. Evaluation measures
5. K-Nearest Neighbors
6. Decision Trees
7. Support Vector Machines

Naïve Bayes Classification

A simple approach for computing the required components for the Bayes rule, specifically the Likelihood

A good and robust algorithm, recommended as one of the first ‘house models’ to try

Bayesian Classification: Binary Domain

Recall our situation:

- Two classes: ***Bad***, ***Good*** (e.g., CPUs)
- Assume each instance has N attributes (test results)
  - X_n is a binary variable with value *0,1*

Example:

X_1	X_2	…	X_N	C
0	1		1	G
1	0		1	B
1	1		0	G
…	…		…	…
0	0		0	G

Binary Domain - Priors

How do we estimate $P(C)$?

- Simple Binomial estimation
  - Count # of instances with $C = B$, and with $C = G$

X_1	X_2	…	X_N	C
0	1		1	G
1	0		1	B
1	1		0	G
…	…		…	…
0	0		0	G

Binary Domain - Likelihood

How do we estimate $P(X_1,\ldots,X_N|C)$?

Two sub-problems in this case:

Compute $P(X_1,\ldots,X_N|C=G)$

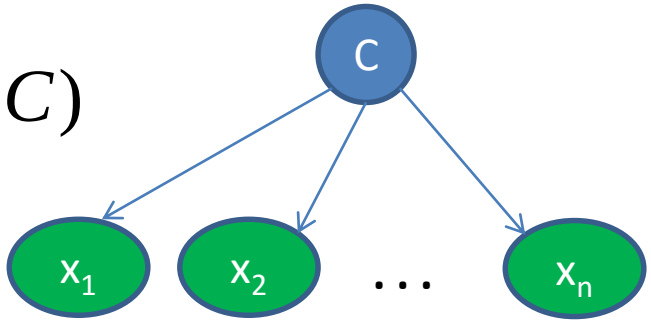
Compute $P(X_1,\ldots,X_N|C=B)$

X_1	X_2	…	X_N	C
0	1		1	G
1	0		1	B
1	1		0	G
…	…		…	…
0	0		0	G

Naïve Bayes Likelihood

- Assume

$$P(X_1,\ldots,X_N \mid C) = P(X_1 \mid C)\cdots P(X_N \mid C)$$



- This is an **independence assumption:**

Each attribute X_i is independent of the other attributes once we know the value of C

- *In our example: Test results are independent once we know whether the CPU is Good or Bad*

Naïve Bayes: Boolean Domain

- We only need to compute for each i:

$$P(X_i = 1 | C = G)$$
$$P(X_i = 1 | C = B)$$

How do we estimate $P(X_1 = 1 | G)$?

- Simple binomial estimation:

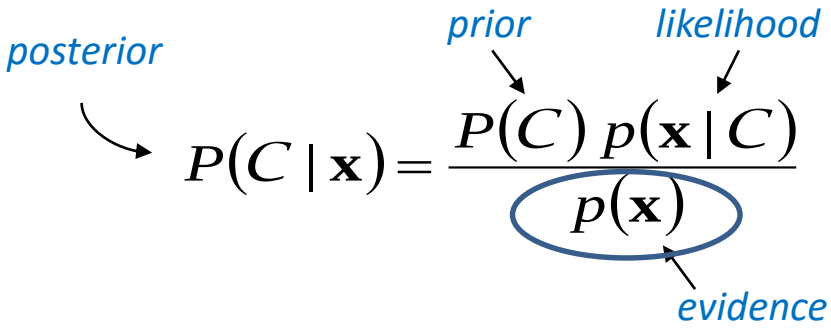
Count #1 and #0 values of X_1 in instances where $C=G$

(Similar for non-boolean discrete features)

X_1	X_2	…	X_N	C
0	1		1	G
1	0		1	B
1	1		0	G
…	…		…	…
0	0		0	G

Interpretation of Naïve Bayes

Now we can almost compute this:



Note that computing the denominator is not required, since it does not depend on ***C***

We choose the class with maximum posterior probablity

What About Continuous Features?

- If the features are continuous, the likelihoods $P(X_i|C)$ may be computed in several ways

  – **Binning to obtain discrete variables**, and then estimating by count of the bin in the training set

  – **Assuming normal distribution**:

  Computing mean μ and standard deviation σ of X_i for each class in the training set, and assuming $X_i \sim N(\mu, \sigma)$

When Is Naïve Bayes helpful?

- High input-space dimension, making it difficult to estimate probability density in that space

- Input contains both continuous and discrete variables

Agenda

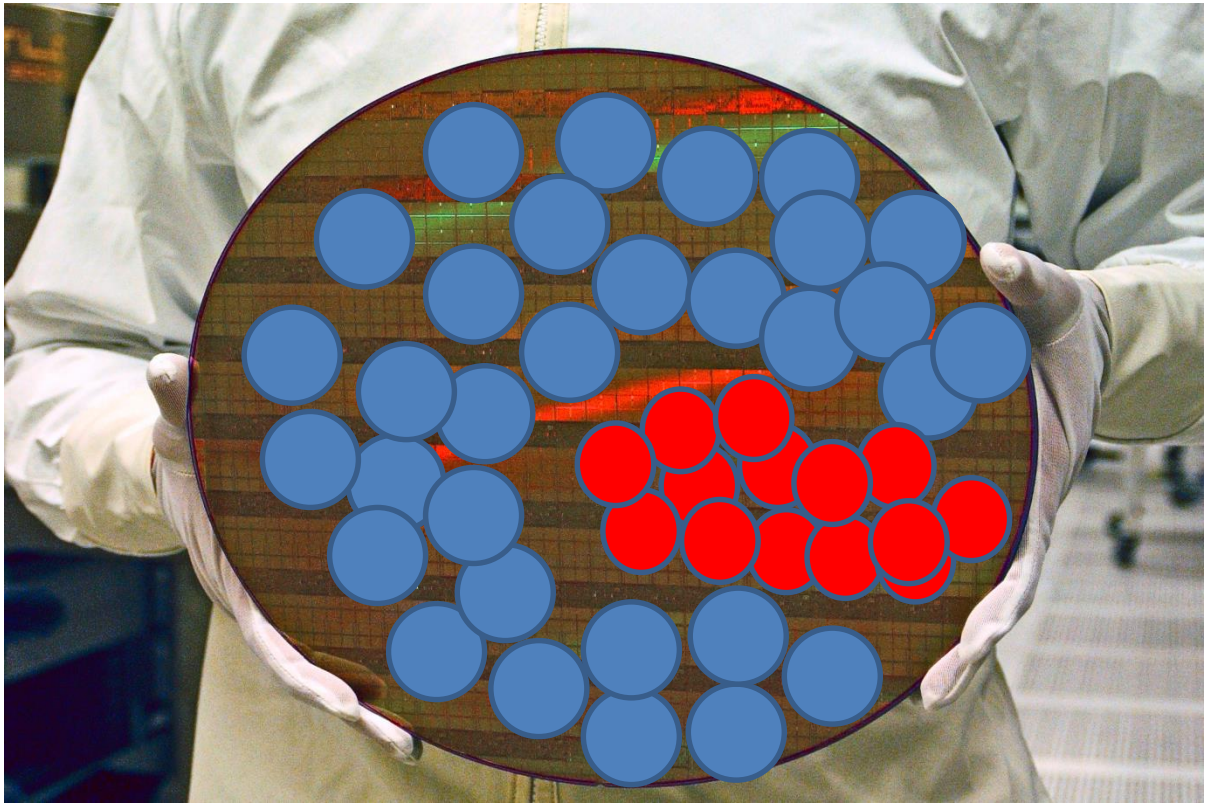
1. Introduction
2. Bayesian decisions
3. Naïve Bayes
4. **Evaluation measures**
5. K-Nearest Neighbors
6. Decision Trees
7. Support Vector Machines

Agenda

1. Introduction
2. Bayesian decisions
3. Naïve Bayes
4. Evaluation measures
5. **K-Nearest Neighbors**
6. Decision Trees
7. Support Vector Machines

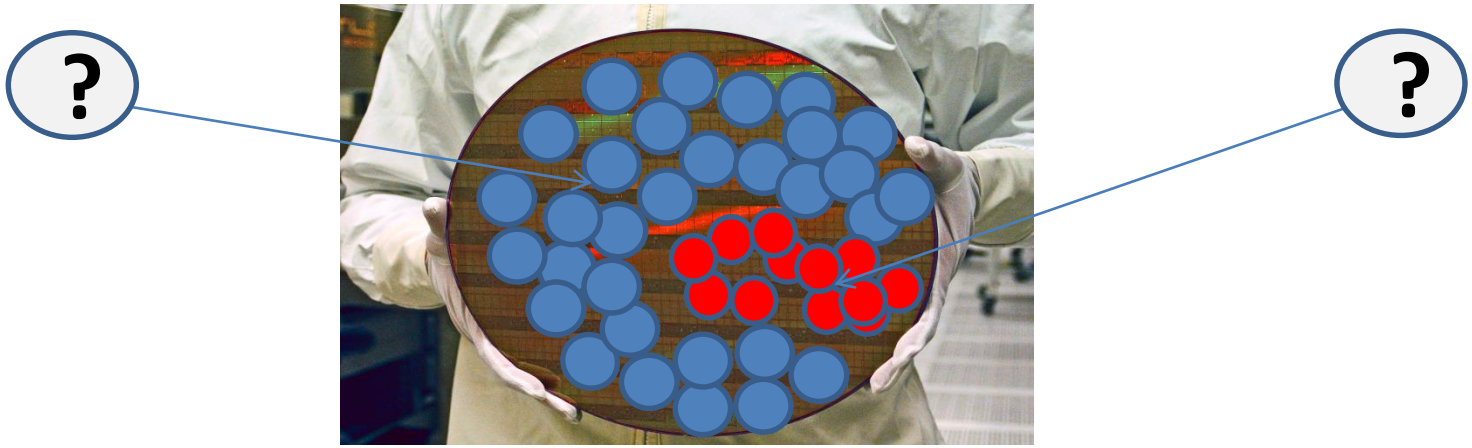
kNN – tell me who your neighbors are…

- Back to our CPU problem
- A wafer, good units, bad units



kNN – tell me who your neighbors are…

- Basic idea: Use **local** structure of **data itself**
- Store all data, and when new unit comes, find K nearest units, and classify by majority
- Known as “lazy” or “**instance-based**” learning
- 1-NN: Give new unit the label of closest data point



- Probabilistic / Bayesian decision formulation?
- ‘Nearest’ by what measure?

kNN as probability estimation

- kNN estimates the conditional probability *p(y|x)*
- Count of data points in class *y* in the neighborhood of *x*

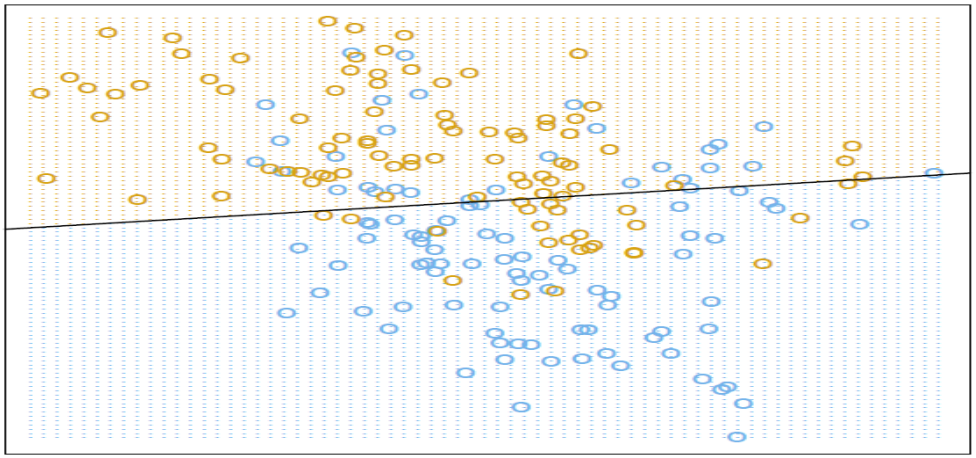
$$p(y|x) = \frac{|\{(x_i, y_i) : y_i = y, x_i \in N(x)\}|}{|N(x)|}$$

Where N(x) are the (indices of) points in the neighborhood of x

- A form of density estimation
- We can then use Bayes decision rule – **MAP classification**
- Recall that to minimize **0-1 Loss,** we choose c that **maximizes** $P(y = c\,|x)$ i.e. majority class in neighborhood

Decision Boundaries

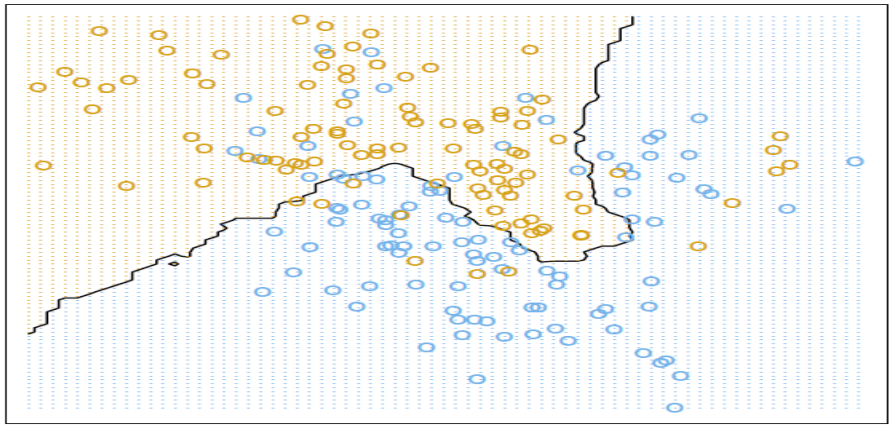
- Recall: Classification as decision problem. Classifier partitions the feature space into volumes called **decision regions**.



- The decision regions are separated by surfaces called the **decision boundaries**. These separating surfaces represent points where there are **ties** (in terms of loss) between categories.

kNN – non-linear decision boundaries

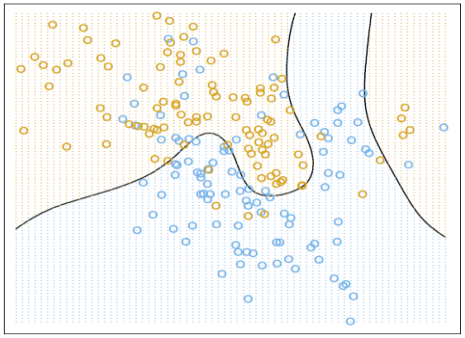
- No assumptions are made about the shape of the decision boundary



- We can expect this approach to dominate linear classifier when the real (Bayes) decision boundary is highly non-linear.

From K=1 and beyond

- **1NN** induces a **rough** boundary (previous slide)
- The larger the K, the **smoother** the decision boundary; small K could lead to **overfitting**

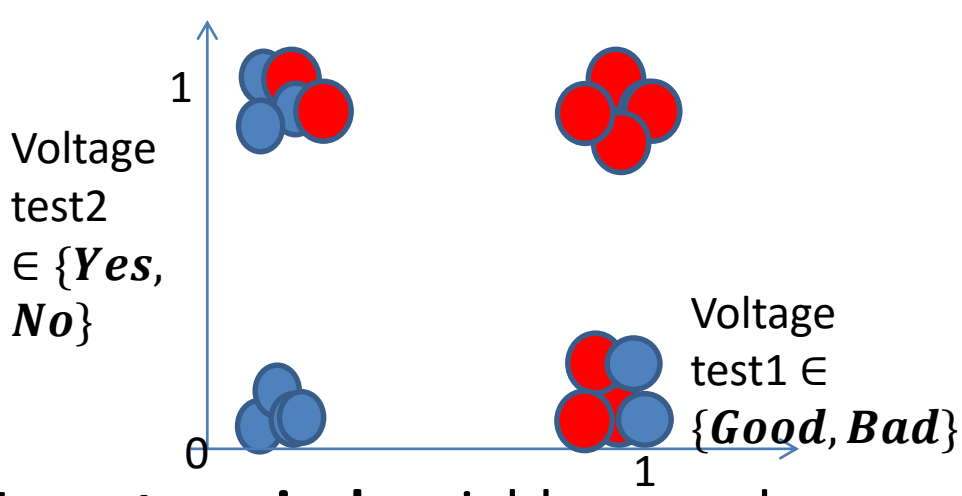


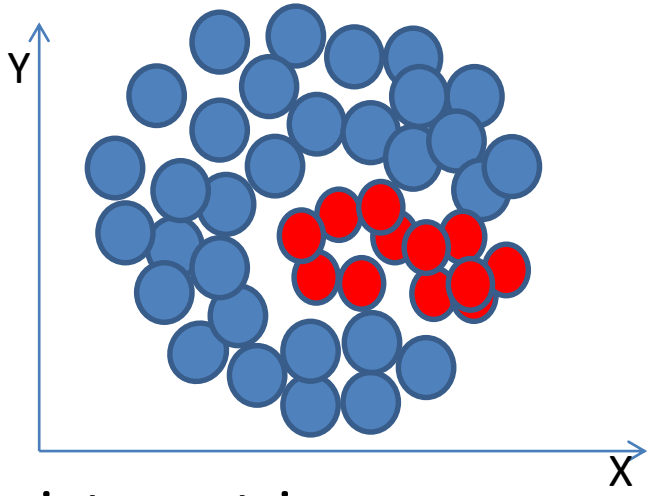
- **Bias-variance tradeoff**: Smaller K -> larger **variance** (instability of prediction);

  Larger K->larger **bias** (inaccuracy of prediction)
- If we take just a "**small neighborhood**" it's likely to be homogenous (low bias) but we are basing ourselves on a small number of points (high variance)
- For large enough dataset ($n \to \infty$), and large enough K, approaches optimal Bayes classifier (**Bayes error**)

‘Nearest’ according to what distance?

- Most common **distance metric** is **Euclidean**
- In our wafer example, with location as an input, this has the intuitive explanation of “geographic” proximity, but:





- For **categorical** variables, need an appropriate metric
- **Weights**: Give decaying importance to neighbors as they get further away
- In general, many different options and practicalities: choose according to problem and accuracy (using validation)

kNN's curse (of dimensionality)

- Main problem: kNN does not work well with **high dimensional** inputs; in high dimensions, *"everything is distant"*
- How do we find "near" people by their various "features"? Location, job, religion, education, height…
- Method is **no longer very local**, despite the name "nearest neighbor"
- The trouble with looking at "neighbors" that are so **far away** is that they may not be good predictors about the behavior of the input-output function at a given point

קרוב רחוק...

kNN summary + a note about interpretability

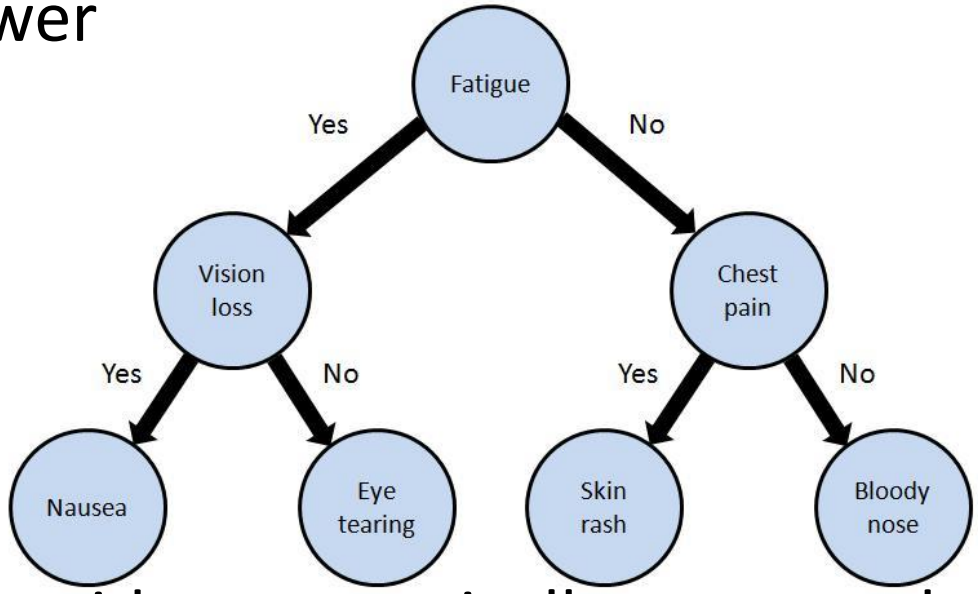
- **Local** method for complicated **decision boundaries**
- Fast training: Just store the data (but iterating and validating over **different metrics** and K-neighbors could change the picture…)
- But no assumption about decision boundary comes with a big cost: Needs a lot of data, sensitive to smoothing choice and mostly severely sensitive to dimensionality: Rapidly increasing **sparsity of 'neighborhood'** in high-dimensions
- **Interpretability/insight**: kNN does not directly tell us which predictors are important: Harder to gain understanding… **Decision Trees** to the rescue?

Agenda

1. Introduction
2. Bayesian decisions
3. Naïve Bayes
4. Evaluation measures
5. K-Nearest Neighbors
6. **Decision Trees**
7. Support Vector Machines

Motivating example: Medical diagnosis

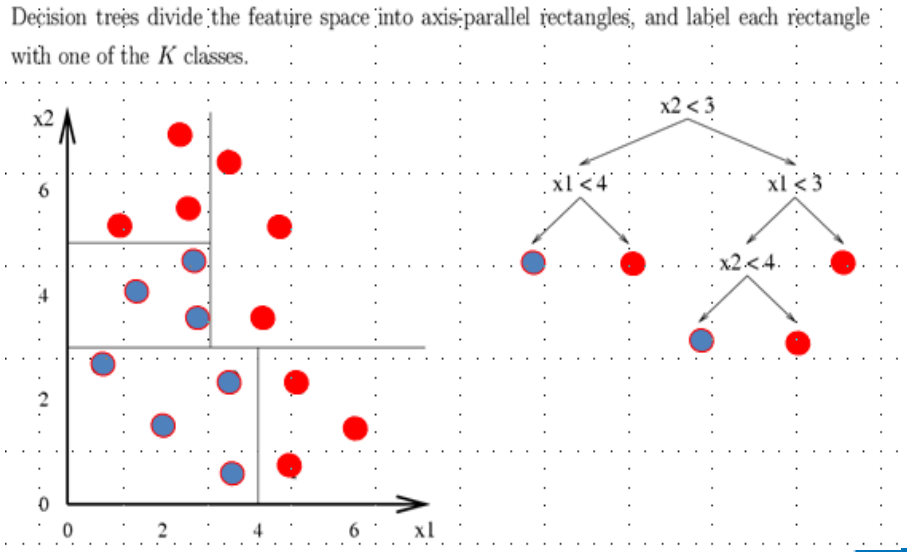
- Build a Healthcare diagnosis system for doctors
- Goal: Predict (diagnose) illness with **sequence of questions**
- Each question leads to another question based on previous answer

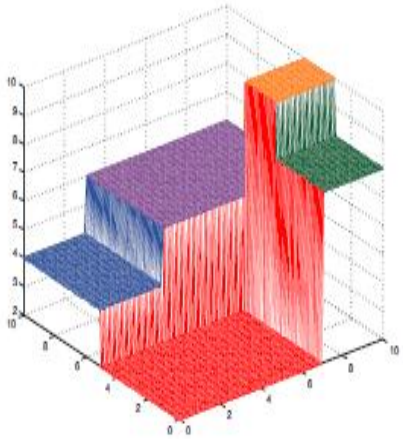


- We aim to provide automatically generated, accurate and **interpretable** (transparent) sequence of such questions

Decision Trees: Prediction by partition

- **Binary partitions** of the input space, and defining a local model in each resulting **decision regions (boxes)** of input space, represented by **leaf nodes**
- In trees used for **classification**, boxes hold class labels – for predictions we associate each region with **majority class**
- We can easily also use trees for **continuous target** - we simply use the **average** in each box for prediction

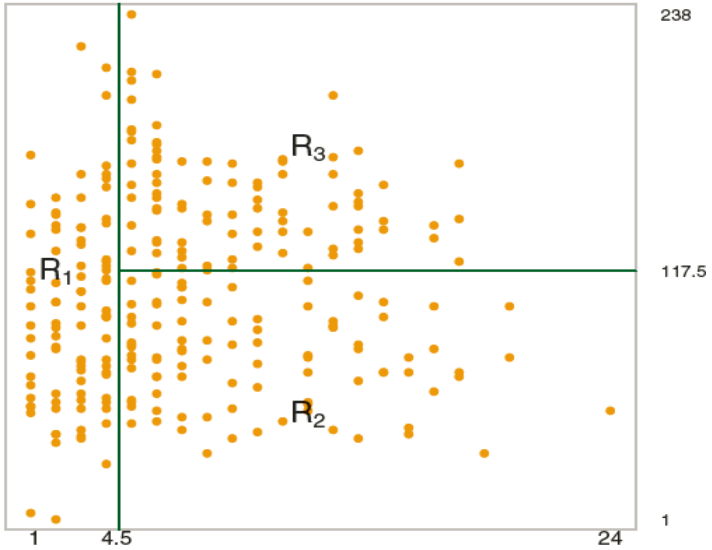




Decision Trees – big picture

For example, predict tablet sales with clear insight into model (screen size, weight, battery life)

- **Big picture**: Two main steps:

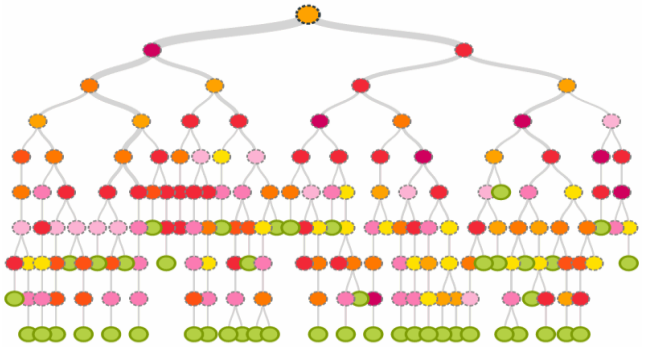


1. We divide the feature space into *J* distinct and non-overlapping **regions** $\boldsymbol{R_1, R_2, \ldots R_J}$

2. For every observation falling in region R_j, predict (majority/average) for training observations in R_j

How to partition?

- Divide predictor space into high-dimensional **boxes**
- Find boxes $(R_1, R_2, \ldots R_J)$ that minimize **Loss Function –** a **key component** for building the tree, determining its type and inner workings – we will see examples
- Computationally infeasible to consider **every** possible partition of the feature space
- Heuristic solution: **Recursive binary splitting**



1. **Top-down**: Begin at the top of the tree (all points in same region), **successively split** the predictor space
2. **Greedy:** At each step, **best feature for split** is found **locally** – for the immediate split, rather than **looking ahead** and picking a split that may lead to a better tree in some **future step** further down

Recursive partitioning

- Basic method (example with continuous target – tablet sales)
- **Loss function**: $\sum_{Boxes} \sum_{i \in Box} (y_i - \boldsymbol{average}(y_{box}))^2$ (*Sum of Squared Errors*)
- e.g. sum in one box: (3.1 – 3)^2 + (2.9-3)^2

1. **Initialization**: First select feature X_j and split (s) that leads to greatest reduction in SSE:

$$R_1(j,s) = \{X | X_j < s\} \text{ and } R_2(j,s) = \{X | X_j \geq s\}$$

That minimize

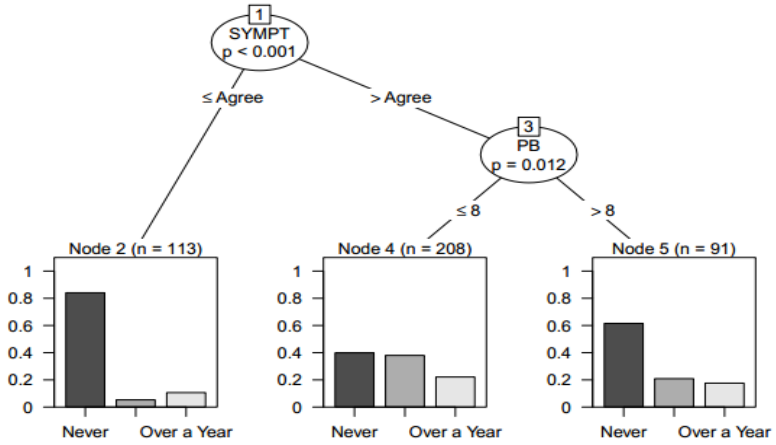
$$\sum_{i:\, x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:\, x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

2. **Recursion**: Split one of the two previously identified regions in same way

3. Continue until a **stopping criterion** is reached

Classification – node purity

- Criteria for splitting (loss function) changes, basic idea same!
- Define **node purity** as a measure of how **homogenous** the node is
- We classify by which *class* ***k*** maximizes $\hat{p}_{mk}$, the **proportion of class k observations** in box m – the **majority class**
- Just like SSE, purity will be used for our loss function



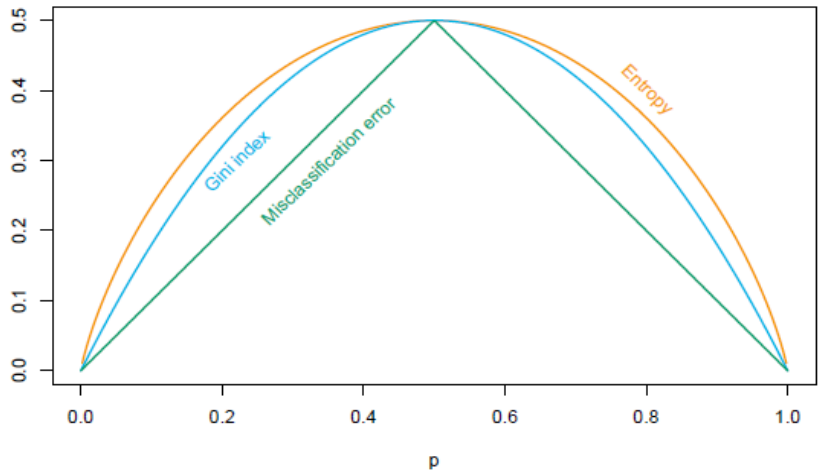
Purity measures

- Unlike in continuous problem, there are several “natural candidates” to choose from

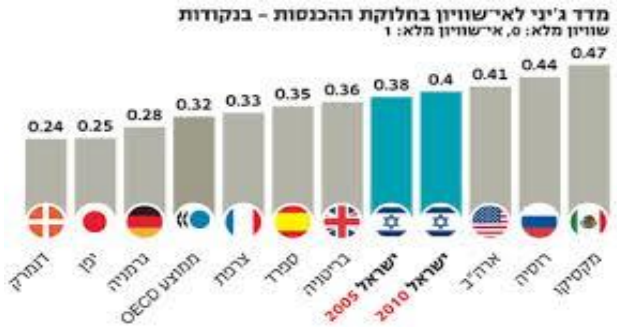
Misclassification error: $\frac{1}{N_m}\sum_{i\in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk(m)}.$

Gini index: $\sum_{k\neq k'} \hat{p}_{mk}\hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1-\hat{p}_{mk}).$

Cross-entropy or deviance: $-\sum_{k=1}^{K} \hat{p}_{mk}\log\hat{p}_{mk}.$



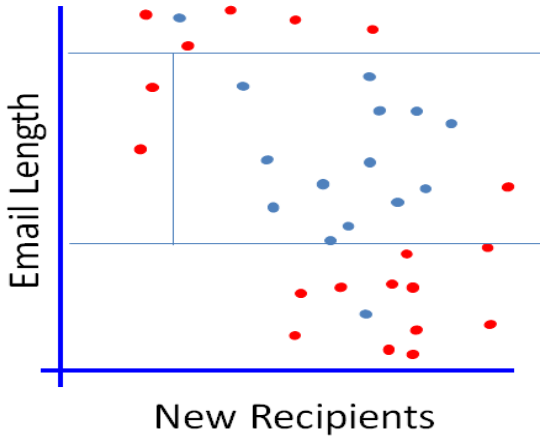
- In words:
- **Misclassification**: Fraction of points in box for which we were wrong
- **Entropy, Gini** : Behave similarly, measure of general **uncertainty** in box
- Gini, entropy are more **sensitive** criteria than misclassification for splitting - often better to use them

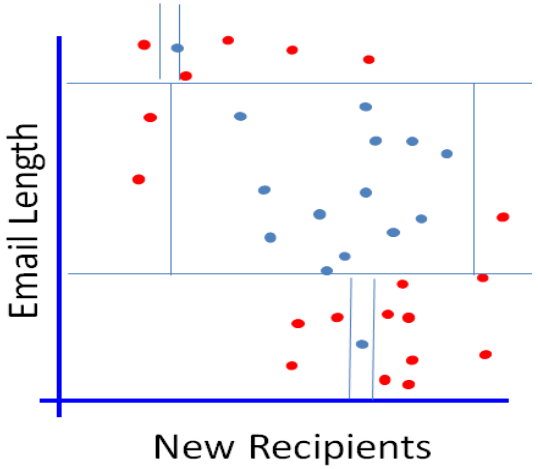


Gain measures

- Rather than decreasing **Loss** in splitting criteria, we can increase **Gain**
- **Information Gain:** By how much will we reduce the entropy after splitting a node using attribute *X? We want this as large as we can*
- *H(Y) – H(Y|X).* Recognize this?
- Maximizing infoGain is equivalent to **minimizing entropy**
- **Categorical features**: Number of partitions grows exponentially in number of values; the more choices we have, the more likely we can overfit – **infoGain weakness**
- **GainRatio:** In practice, correct for the entropy of the attribute itself by reducing its weight in infoGain

How much partitioning?

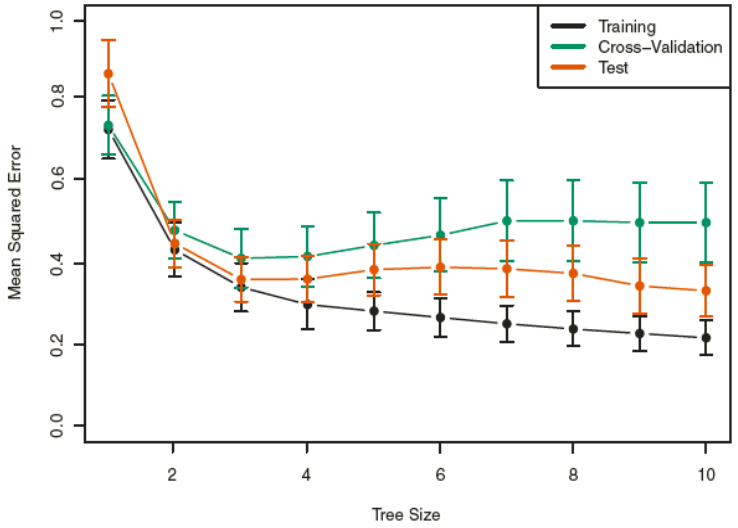




- Too much splitting likely to **overfit** data – tree might be too complex; Fewer splits -> Simpler decision boundaries
- Should we grow only while decrease in loss is large enough?
- **No –** too short-sighted. Small decreases early in splitting process could lead to **big decreases** further down in tree

Pruning

- Better solution: Grow a **large tree** until each node contains a small number of instances
- **Then use pruning** to **remove nodes** (splits) that do not provide additional information
- One natural way is to penalize size of tree (**rpart package** in **R**)



- Also use prediction **accuracy** – prune by node contribution to misclassification rate

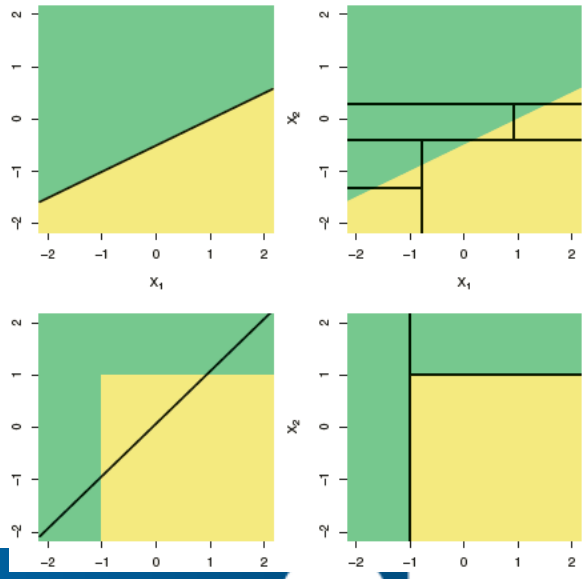
DT issues: White box, but at what cost?

- Very transparent (with related advantage of **Feature Selection**)
- But aside from previous issues we saw, 2 major problems
- **Instability:** Small changes in data can cause very different splits; the effect of an error in the top split is **propagated down** to all of the splits below it (in next session, you will see how **Random Forests** fix this)
- **Lack of smoothness:** Boundary

  or true function could be smooth,

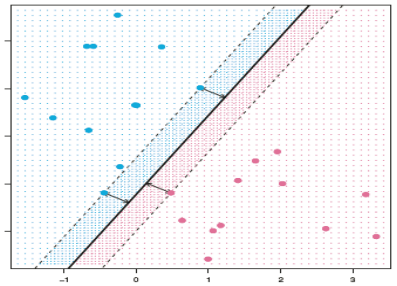
  but we search for partitions.

  Real-world examples?



Decision Trees summary

- **Explanation** for end-use by far strongest point
- Suitable for **segmented** (disjoint) decision boundaries
- Fast learning, easy handling of discrete features
- **Instable**, prone to overfitting
- Difficulty with **smoother structures**
- Generally not a state-of-the-art method, but a sub-routine in some that are
- Can we trade-in for the interpretability, and get a model that can both predict very well and capture complex structures?

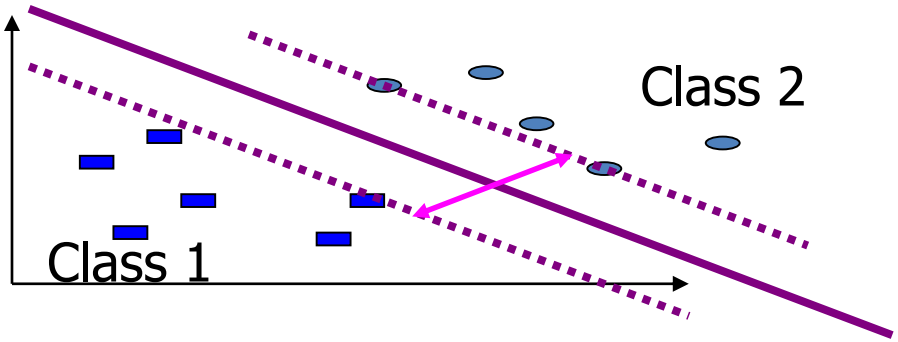


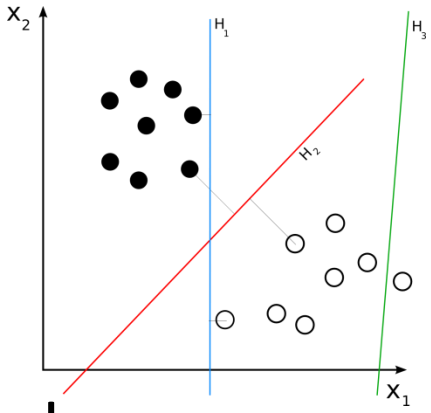
Agenda

1. Introduction
2. Bayesian decisions
3. Naïve Bayes
4. Evaluation measures
5. K-Nearest Neighbors
6. Decision Trees
7. **Support Vector Machines**

Support Vector Machines – the basics

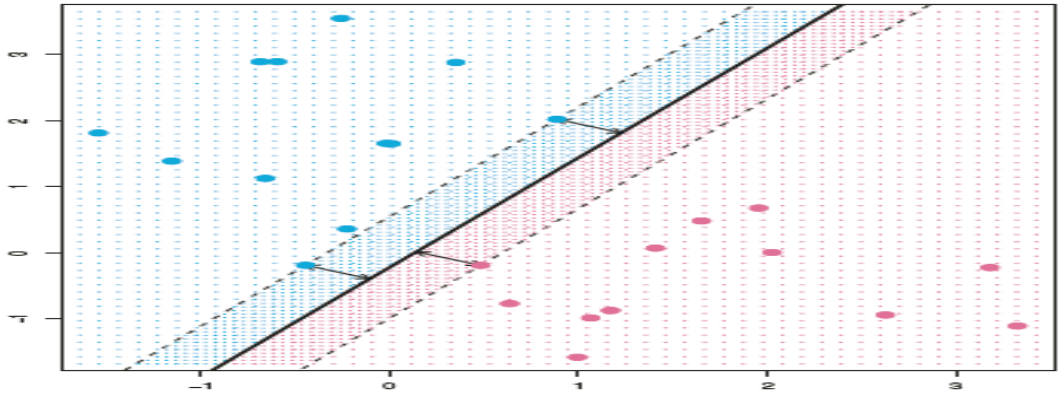
- First thing, some key building blocks for SVM
- What is a **hyperplane**? A high-dimensional plane
- **Separating hyperplane** – a hyperplane that separates the classes





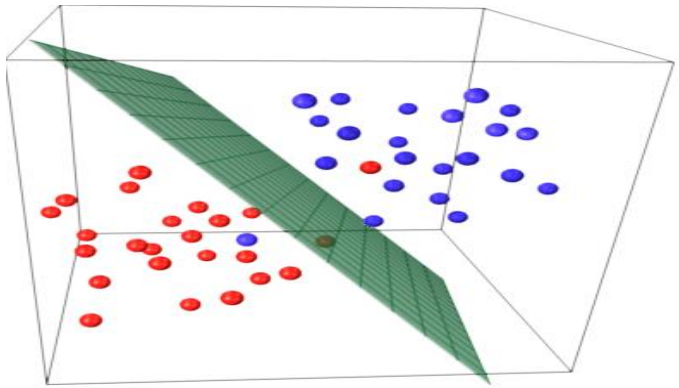
- **Margin** defined by points nearest to hyperplane
- The **maximal margin hyperplane** is the separating hyperplane for which the margin is **largest**
- **Why?** Greater generalization ability, more stability

Support Vectors – what do they 'support'?



- Our data points are just vectors in *p*-dimensional space (*feature1…featurep*)
- **Support vectors:** if these data points were moved slightly, hyperplane would move as well; others "don't matter"
- Our decision (prediction) rule potentially based on **only a small subset** of the observations; **stable** to the behavior of observations that are **far away** from the hyperplane – less sensitive to outliers

Can we really have total separation?

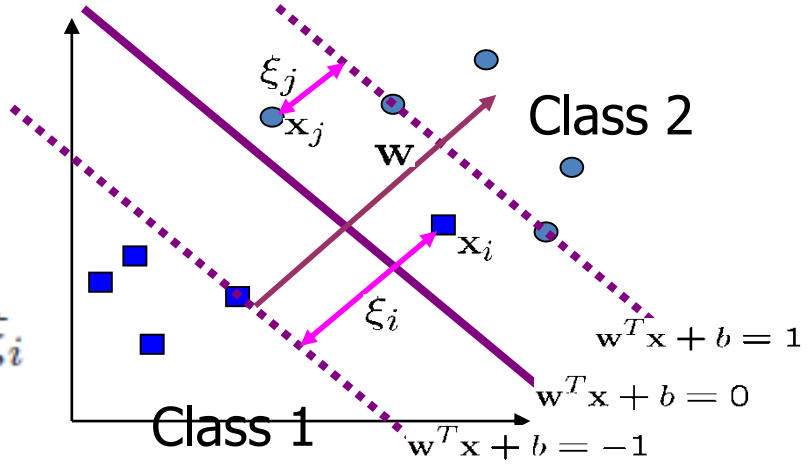


- Real-world: Some points will be on **wrong side** of the hyperplane
- **Soft margin:** Allow some observations to be on the **incorrect side**
- Why not just always find a **perfectly** separating hyperplane (if at all possible)? This is called a **hard margin -** usually leads to **instability** (hence **poor generalization**)
- Example: add just 1 new point. **Margin changes** drastically, and is now very narrow
- With soft margin, get better classification for **most points** in exchange for error on **a few**

Formulization as optimization problem

$$\min_{\mathbf{w},w_0,\boldsymbol{\xi}} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{N}\xi_i$$

$$\text{s.t.} \quad \xi_i \geq 0, \quad y_i(\mathbf{x}_i^T\mathbf{w} + w_0) \geq 1 - \xi_i$$

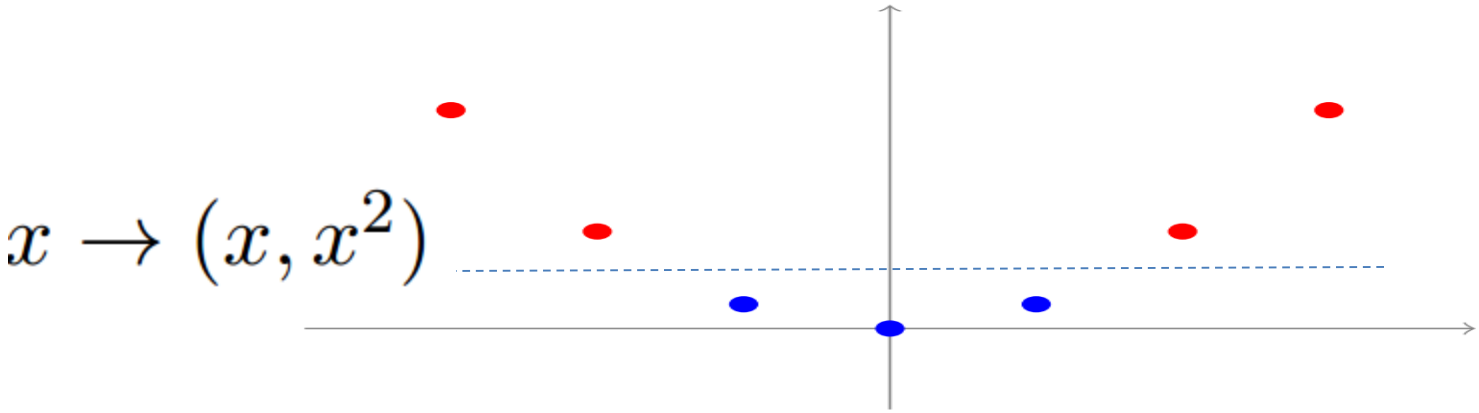


- In words: Maximize **margin** ($\frac{2}{||w||^2}$) while allowing some **deviations** (ξ_i), and **penalize** their sum with **C**
- **C** determines the number and severity of the violations, controls **bias-variance tradeoff/overfitting**: large C -> narrow margins rarely violated; small C -> margin is wider, fitting less hard

Kernels - for nonlinear boundaries

- Data are not linearly separable

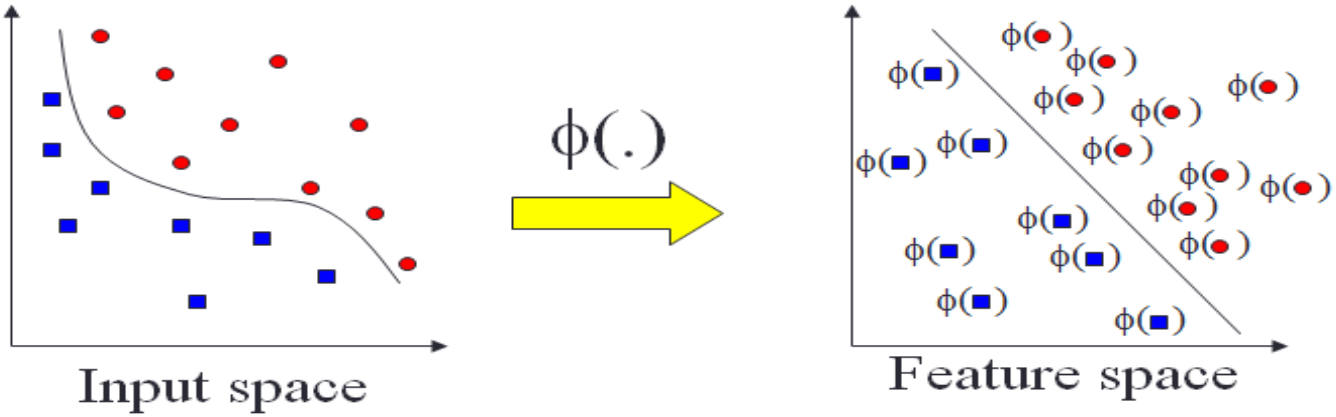
- Possible Solution: Add quadratic transformation of features



- Problem: Feature space grows exponentially
- Better solution: **Kernels** to avoid carrying out transformations explicitly (avoid computing and storing **massive number of extracted features**)

Kernel Trick

- With a few mathematical tricks, kernels allow us to fit **linear** SVM classifier in **higher-dimensional space**, **without actually computing** all these features



- Turns out that SVM easily allows us to do this mapping
- **Polynomial kernel**: Map feature space to polynomials (and their products i.e. interactions), for example $(x_1, x_2) \longmapsto (x_1^2, \sqrt{2}x_1x_2, x_2^2)$

RBF kernel – one important kernel

- **Radial Basis Function** Kernel
- Often first kernel to try (when no prior knowledge)

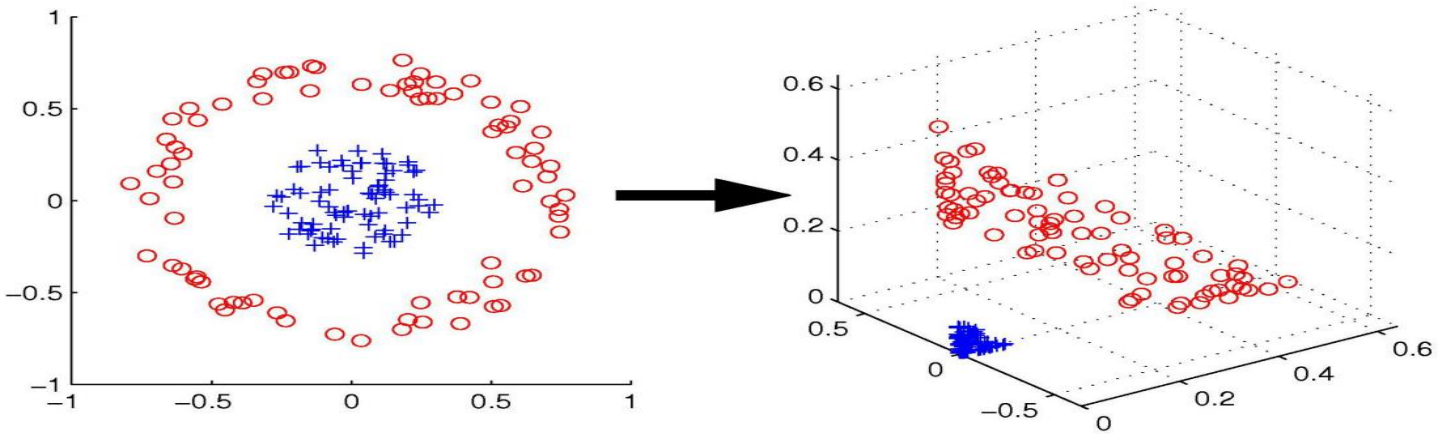


Figure 1: Transforming the data can make it linearly separable

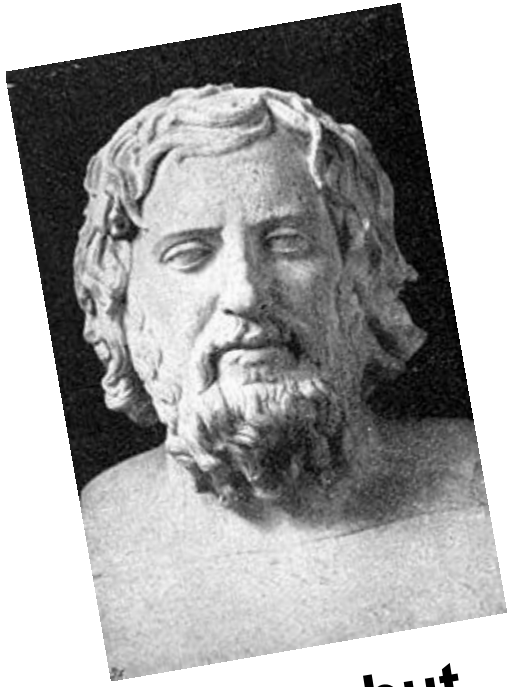
- For a new observation we predict on, points that are **far** from it play essentially no role (kernel gives **exponential** decay in distance) – captures **local ("circular") structure**

SVM summary

- Classifier with strong performance, “general-purpose”
- Natural, efficient use of high-dimensional feature space
- Flexible use of **kernel** types – capture many structures
- But no “analytic” way of finding right one
- **Sparse** solution that depends on **support vectors** only
- This is also a computational problem – need to keep carrying SVs (for kernel SVMs)
- Especially when using kernels, not immediately clear how to interpret/extract insights

Evaluation: Algorithm Preference

- Criteria (Application-dependent):
  - Accuracy
  - Misclassification error, or risk (loss functions)
  - Training time/space complexity
  - Testing time/space complexity
  - Interpretability (Model Complexity, for instance number of hidden units)
  - Easy tuning
  - Easy programmability
  - Easy embedding

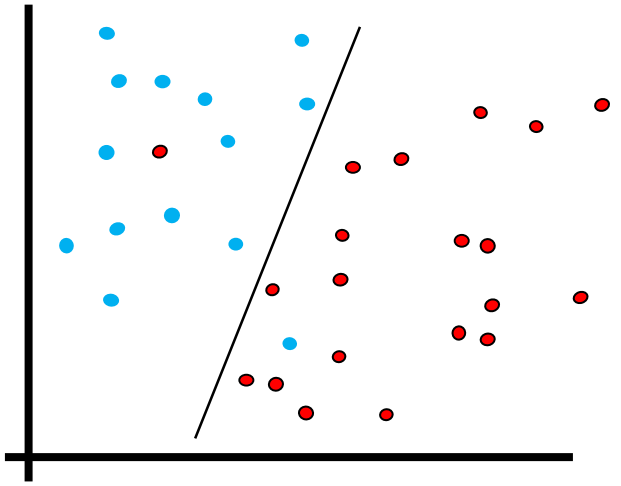


"Fast is fine, but accuracy is everything."
Xenophon
(Greek historian,
430 – 354 BC)

Evaluating What's Been Learned

The Test Set Method

- Randomly select a portion of the data to be used for training (the *training set*)
- Train the model on the training set
- Once the model is trained, run the model on the remaining instances (the *test set*) to see how it performs



Confusion Matrix

Actual \ Predicted	Blue	Red
Blue	13	1
Red	1	19

Confusion Matrix – Terms

		Predicted	
		0	**1**
Actual	**0**	True Negative (**TN**) *A*	False Positive (**FP**) *B*
	1	False Negative (**FN**) *C*	True Positive (**TP**) *D*

If we have more than two classes, we can compute this matrix for each class vs. the others, and then average all matrices

Evaluation Considerations

- Higher accuracy does not necessarily imply better performance on target task
- Classes are often very imbalanced
  - Buying customers are much rarer than non buying customers
  - What if 99:1 split? Always say “no” -- 99% default accuracy
- Errors have different ***costs***
  - Cost of eating poisonous mushroom is greater than cost of going hungry
  - Goals need to be defined by business understanding

Confusion Matrix – Error Measures

		Predicted	
		0	1
Actual	0	True Negative (TN) *A*	False Positive (FP) *B*
	1	False Negative (FN) *C*	True Positive (TP) *D*

- **Error rate** = $\frac{FN+FP}{TN+FP+FN+TP} = \frac{C+B}{n}$
- **Accuracy** = $\frac{TN+TP}{TN+FP+FN+TP} = \frac{A+D}{n}$
- **Recall** = $\frac{TP}{FN+TP} = \frac{D}{C+D}$ (= **Sensitivity = Hit rate**)
- **Specificity** = $\frac{TN}{TN+FP} = \frac{A}{A+B}$ (= **1- False Positive rate**)
- **Precision** = $\frac{TP}{FP+TP} = \frac{D}{B+D}$

Confusion Matrix – Error Measures

- **Specificity** = $\frac{TN}{TN+FP} = \frac{A}{A+B}$
- **False positive rate** = $\frac{FP}{TN+FP} = \frac{B}{A+B}$
- **Recall** = $\frac{TP}{FN+TP} = \frac{D}{C+D}$ (= **True positive rate**)

		Predicted	
		0	1
Actual	0	True Negative (TN) *A*	False Positive (FP) *B*
	1	False Negative (FN) *C*	True Positive (TP) *D*

- When there are few positives, or when every positive should be detected, then high recall is crucial.

- When there are many positives and we just need some of them, we settle for lower recall and require high specificity.
For example: Find 1000 programmers according to their LinkedIn profiles.

Confusion Matrix – Error Measures

F-Measure = $2 * \frac{Precision * Recall}{Precision + Recall}$

		Predicted	
		0	1
Actual	0	True Negative (TN) *A*	False Positive (FP) *B*
	1	False Negative (FN) *C*	True Positive (TP) *D*

F-measure is the harmonic average of precision and recall.

Provides a unified measure for both of them.

Worst value is 0, best value is 1, when both precision and recall are 1.

Commonly used in document retrieval/classification.

ROC Curve

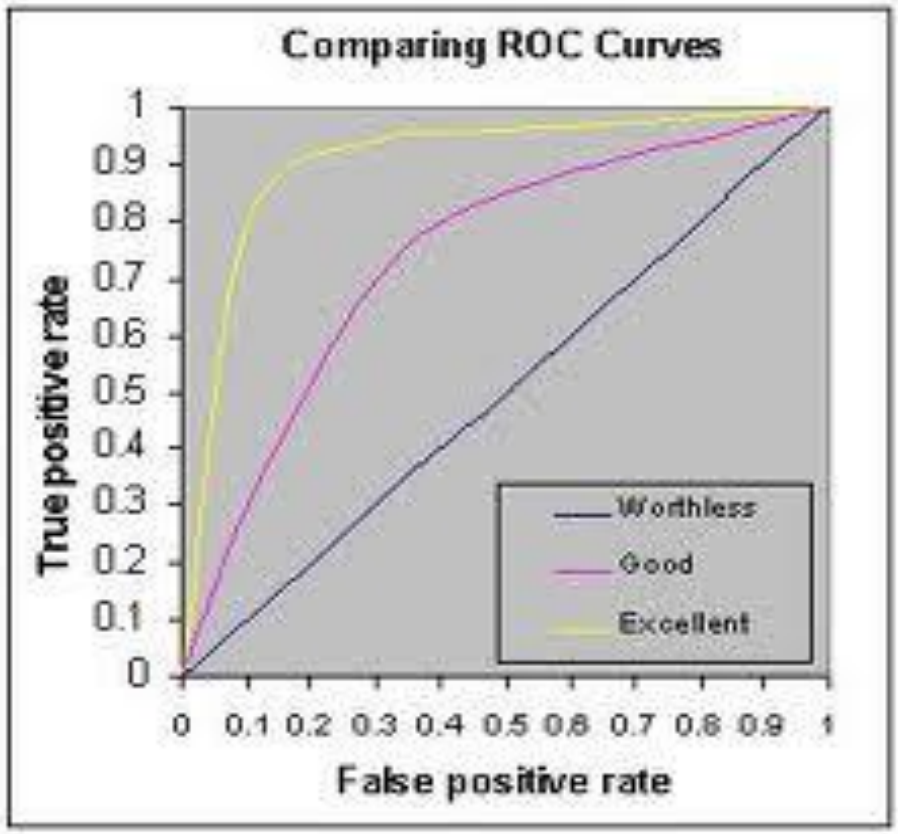
A way to assess and visualize the tradeoffs between classifiers and choose between them

False positive rate = $\frac{FP}{TN+FP}$

(e.g. fraction of bad CPUs considered good)

True positive rate = $\frac{TP}{FN+TP}$

(e.g. fraction of good CPUs considered good)



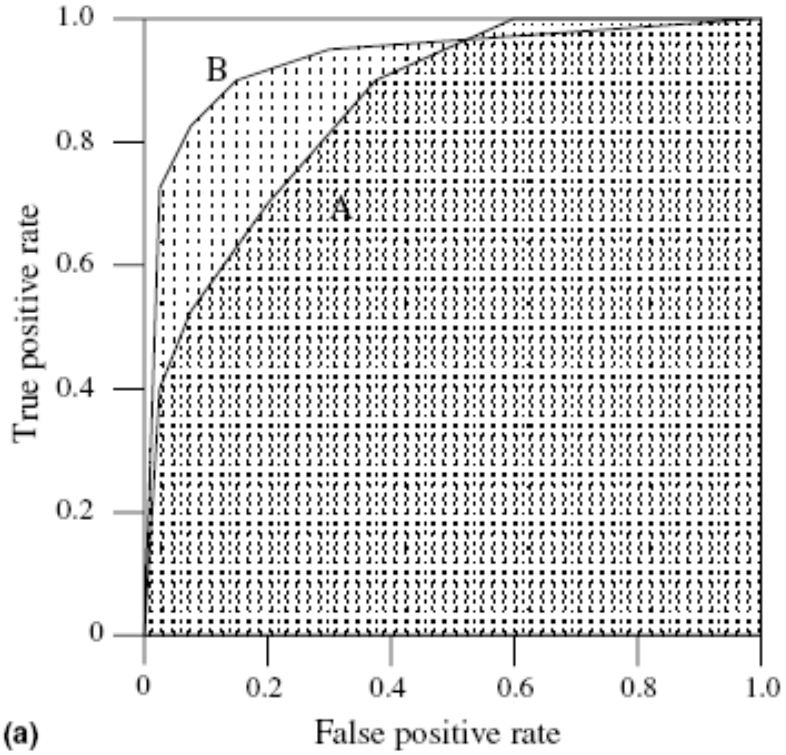
Area under ROC curve (AUC)

Comparing two ROC curves:

- The graph represents the areas under two ROC curves, A and B. Classifier B has greater area and therefore better average performance

- Note: Since the AUC is a portion of the area of the unit square, its value will always be between 0 and 1.

- AUC is equivalent to the probability that the classifier will rank a positive instance higher than a negative instance



ROC and AUC - Summary

- ROC curve provides a popular visualization for the tradeoff between false positive rate (cost) and true positive rate (benefit) of a classifier

- AUC reduces the ROC curve to one number, reflecting the classifier's quality (probability that true will rank higher than false).

- It is better to review the full ROC curve, as a single number does not reflect it in full. For example, high false positives are incorporated in AUC but less interesting.