Machine learning frameworks and hands-On

Kan Min-Yen

Day 2 / Afternoon

Math Foundations

- Probability and Naïve Bayes
- Parameter Estimation
- Mixture Models and Expectation Maximization

Bayesian Interpretation

- **Belief Interpretation:**
  - View the probability P(e) as a degree of belief in event e

    "How likely is e true" – not "e is P(e)% true"

- **Frequentist (Bayesian) Interpretation:**
  - View a probability as relative success over total trials

Either is fine; scale things to [0-1] range

Conditional Probability

- **P(e|D) conditions our belief in e based on observed data D**
  - –Where D is known or perceived to be true
  - –Or where D updates our knowledge of the world over time

Bayes Theorem

Data *Likelihood* (in world where e is true)

Prior probability (before any data)

Posterior probability (after seeing data)

$$P(e \mid D) = \frac{P(D \mid e)P(e)}{P(D)}$$

Example

e = # of webpages in existence today is over 10^{10}

D = # of webpages indexed by SE_1

- P(e) =
- P(D) =
- P(e|D) =
- P(D|e) =

New Data

- D_2 = # of pages indexed by SE_2. Now what?
- Reuse Bayes Theorem!

New Posterior (after seeing data)

Prior, was our old Posterior

$$P(e \mid D, D_2) = \frac{P(D_2 \mid e, D)\,P(e \mid D)}{P(D_2 \mid D)}$$

Naïve Bayes

- Often we may have multiple pieces of evidence: D, D_2 … D_n
- Often hard to calculate how one piece of data affects other pieces.
- Sometimes OK to ignore such correlations. It's naïve but we do it to simplify our calculation.

$$P(e \mid D \ldots D_n) = \frac{\prod_i P(D_i \mid e) P(e)}{\prod_i P(D_i)}$$

- This ignores the correlation between pieces of data.
- For example $P(D,D_2)$ is simplified to $P(D)P(D_2)$

Logarithms

- As most probabilities we deal with are very small (think why?), it’s often more convenient to deal with log probabilities (why again?)
- Ex:

$$\log P(e \mid D) = \log P(D \mid e) + \log P(e) - \log P(D)$$

The role of priors

- Often we want to estimate an event probability based on data: i.e., P(e|D)
- But when we don't have much data the prior is quite helpful
- When we have a lot of data, the prior's role diminishes

$$P(e \mid D \ldots D_n) = \frac{\prod_i P(D_i \mid e) P(e)}{\prod_i P(D_i)}$$

- **Many models have parameters θ to define them**
  - –E.g., A normal (Gaussian) distribution is defined by parameters for its mean and std. dev.
- **We usually try to estimate these from data**
  - –So again we have a prior P(θ) and posterior P(θ |D)

- **Goal: find best set of parameters θ that maximizes the posterior P(θ |D)**
- **Called maximum a posteriori (MAP)**

MAP and ML Estimation

Max P(θ|D) is equivalent to min – log P(θ|D) Why?

$\mathcal{E}(\theta) = -\log P(\theta|D) \quad = -\log(D|\theta) - \log P(\theta) + \log \cancel{P(D)}$

$\quad = -\log(D|\theta) - \log P(\theta)$

Constant over maximization process, ignore

If the prior is uniform, then it is also irrelevant to our minimization.

Reverts to maximum likelihood (ML) estimation

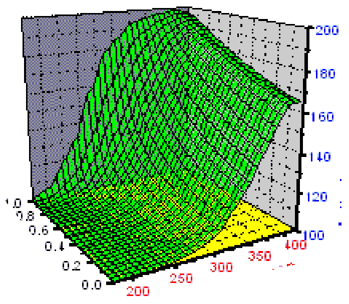
$\mathcal{E}(\theta) = -\log P(\theta|D) \quad = -\log(D|\theta)$

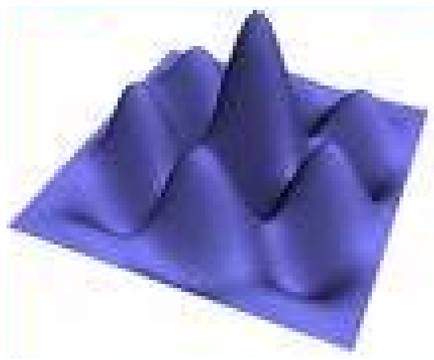
Minimizing $\mathcal{E}(\theta)$

The negative log-likelihood $\mathcal{E}(\theta)$ usually needs to be iteratively estimated

- Exact calculation usually not possible
- Use gradient descent or Expectation Maximization to find optimum

Caveat: the surface to optimize should be smooth, without lots of local minima





$\mathcal{E}(\theta)$ can also be an error function

If we measure how good a model M(θ) fits the data by f(θ,D) >= 0 then we can also frame it as a likelihood.

$$P(D \mid M(\theta)) = \frac{e^{-f(\theta,D)}}{Z}$$

Where Z is a normalization constant to make P(D|M(θ)) over all θ integrate to 1.

$$Z = \int_{\theta} e^{-f(\theta,D)} d\theta$$

Ex 1: Coin Flipping

- **Observation o={o_1, o_2,…, o_n}**
- **Maximum likelihood estimation**

Independent trials

$$b^* = \arg\max_{b\in[0,1]} \Pr(\mathbf{o} \mid b) = \arg\max_{b\in[0,1]} \prod_{i=1}^{n} \Pr(o_i \mid b)$$

E.g.: o={h, h, h, t, h,h}

Pr(o|b) = b^5(1-b)

$$b^* = \arg\max_{b\in[0,1]} b^5(1-b) \rightarrow b = 5/6$$

Ex 2: Unigram Language Model

- Observation: d={tf$_1$, tf$_2$,…, tf$_n$}
- Unigram language model
  θ={p(w$_1$), p(w$_2$),…, p(w$_n$)}
- Maximum likelihood estimation

$$\theta^* = \arg\max_{\theta\in\Theta} \Pr(d \mid \theta) = \arg\max_{\theta\in\Theta} \prod_{i=1}^{n} [\, p(w_i)\,]^{tf_i}$$

$$\rightarrow p(w_i) = tf_i / |d|, \forall i \in [1, n]$$

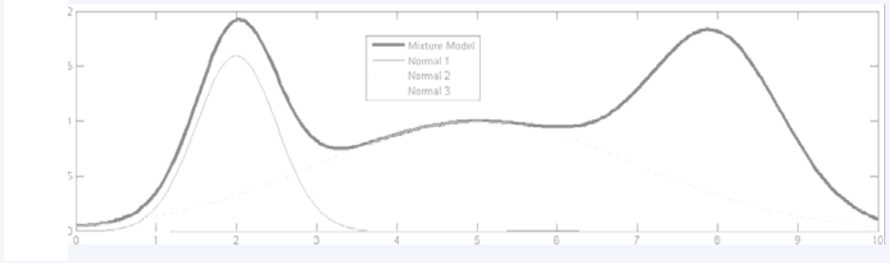
Sparse Data

- **In both examples, ML estimation yield models that are counterintuitive**
  - –You wouldn’t assume a coin is biased if you flipped it 6 times and got heads 5 times
  - –You wouldn’t assume a new web page wouldn’t contain words not seen on another page

- **That is, you have a** prior belief **in (unseen) events**
  - –Adding a prior models this (MAP vs. ML)
  - –This is why we said earlier that priors help when the amount of data is small

Mixture Modeling

- **A formalism for modeling a probability density function as a sum of parameterized functions**



- **Observed population data is complicated – not well fit by a canonical parametric distribution**
- **Assume: ‘Hidden’ subpopulation data is simple – well fit by a canonical parametric distribution**
- **Hope: 1 hidden subpopulation <-> 1 simple parametric distribution**
- **Key questions:**
  - – How many hidden subpopulations are responsible for generating the data?
  - –Which subpopulation did each data point come from?

Mixture Models

- Building a complex model out of simpler ones
- Simplest: build a linear combination of K models:

$$P(D|\theta) = \sum_{l=1}^{K} \lambda_l P(D|\theta_l) \quad \textbf{where} \quad \sum_i \lambda_i = 1$$

Mixing coefficients / weights

- Like before we want to find a set of parameters that maximizes the data (log) likelihood

Maximizing MM Likelihood

- **The problem is that the function we want to maximize is difficult to handle (because of the log of the sum):**

$$L = \sum_{i=1}^{n} \log(\sum_{l=i}^{K} \lambda_l P(d_i \mid \theta_l)) + \mu(1 - \sum_{l=1}^{K} \lambda_l)$$

- **We know neither the** assignments **or** parameters!
- **It would be a lot easier if we knew either one.**
  - –If we knew the assignments, we can compute the parameters (mean, variance) for the set of data for each component
  - –If we knew the parameters, we could assign points to each component (at least probabilistically)

Expectation Maximization

- **E step: compute expected values of λ (that component / created D_i),** pretending **current parameters are right**
  - –Calculate a "soft" assignment: assign posterior probabilities to data points for each component
- **M step: maximize the expectation computed in E-step**
  - –Calculate new θs based on λ
- **Starts with some initial guess of θ**

Expectation Maximization (EM)

- **E step: find which component / generated the data D_i**
  - –Calculate a “soft” assignment: assign posterior probability P(D|θ)
- **M step: maximize the expectation computed in E-step**
  - –Calculate new θ based on
- **Starts with some initial guess of θ.**

EM (2)

- **An algorithm where we alternate between estimating two unknowns**
  - –Assume one is correct to estimate the other, vv.

- **A simple version of this is the** K-means **algorithm for clustering, which we'll return to later in the course.**

Machine Learners and Text Classification

Nearest Neighbors

Regression

Neural Networks

Naïve Bayes

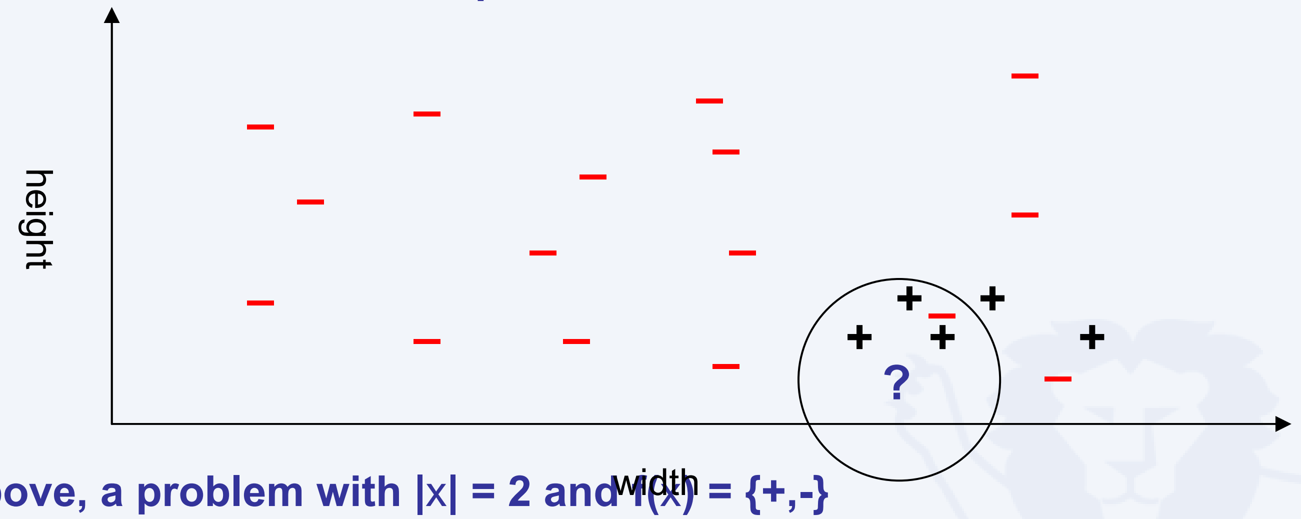
Decision Trees

Support Vector Machines

Maximum Entropy

Nearest Neighbor

- A type of instance based learning – no model
- Remembers all of the past instances
- Uses the nearest old data point as answer



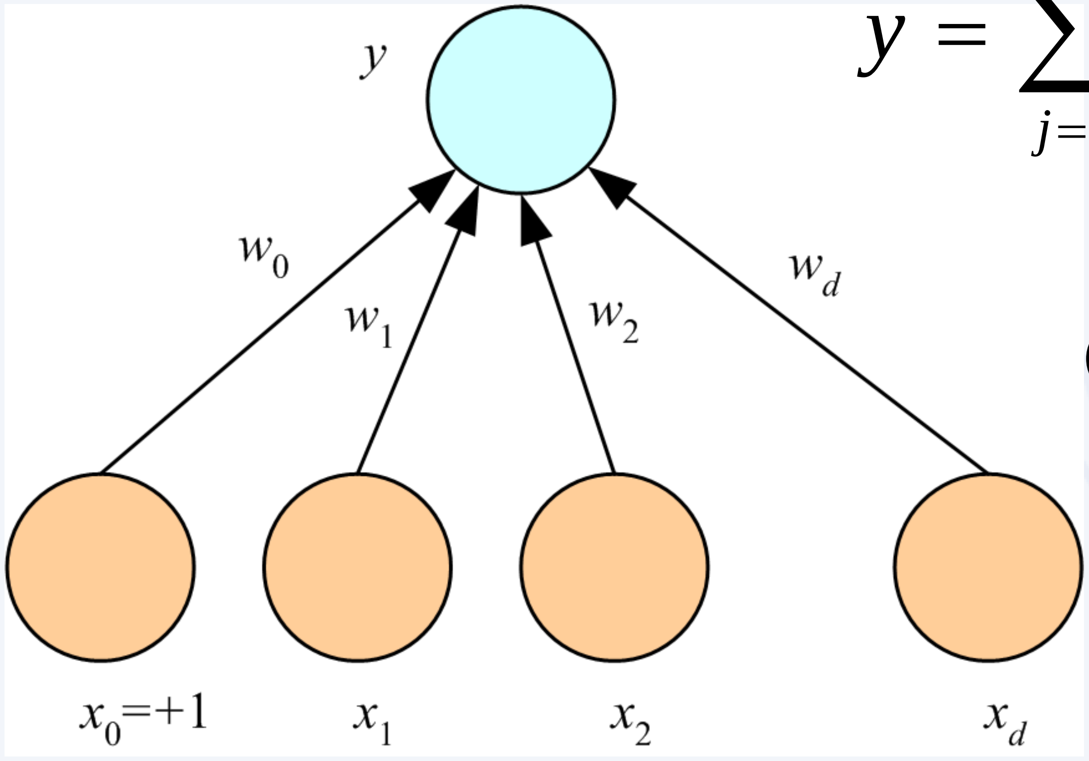
- Above, a problem with |x| = 2 and f(x) = {+,-}

- Generalize to kNN, that is, take the average class of the closest k neighbors.

Remarks on kNN

- **Inductive bias**
  - –Similar classification of nearby instances…
- **Curse of dimensionality**
  - –Similarity metric mislead by irrelevant attributes
  - –Solutions:
    - Weight each attribute differently:
      - Use cross-validation to automatically choose weights
    - Stretch each axis by a variable value.
- **Efficient memory indexing is necessary**
  - –Databases: kd-tree

Perceptrons – A basis for regression and neural networks

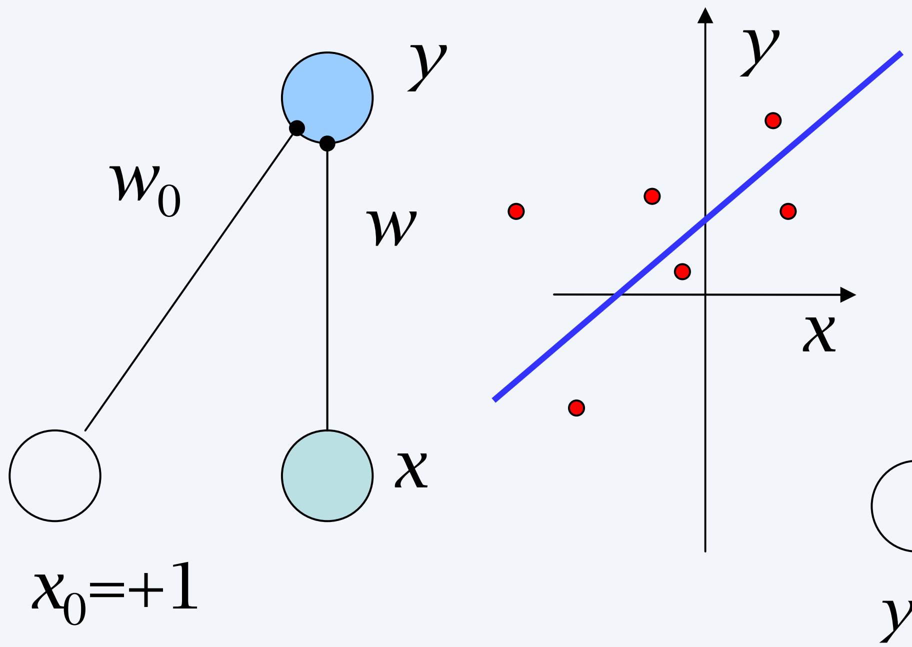


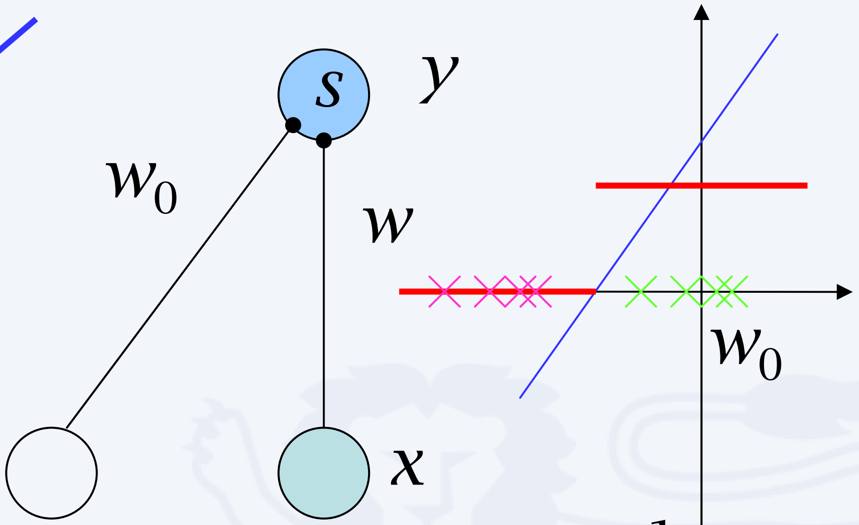
$$y = \sum_{j=1}^{d} w_j x_j + w_0 = \mathbf{w}^T \mathbf{x}$$

(Rosenblatt, 1962)

What a Perceptron Does

- Regression: $y=wx+w_0$
- Classification: $y=1(wx+w_0>0)$





$$y = \text{sigmoid}(o) = \frac{1}{1+\exp\left[-\boldsymbol{w}^T\boldsymbol{x}\right]}$$

Compute w_i from training data to minimize error

Error often measured by squared error

Multi-class classification

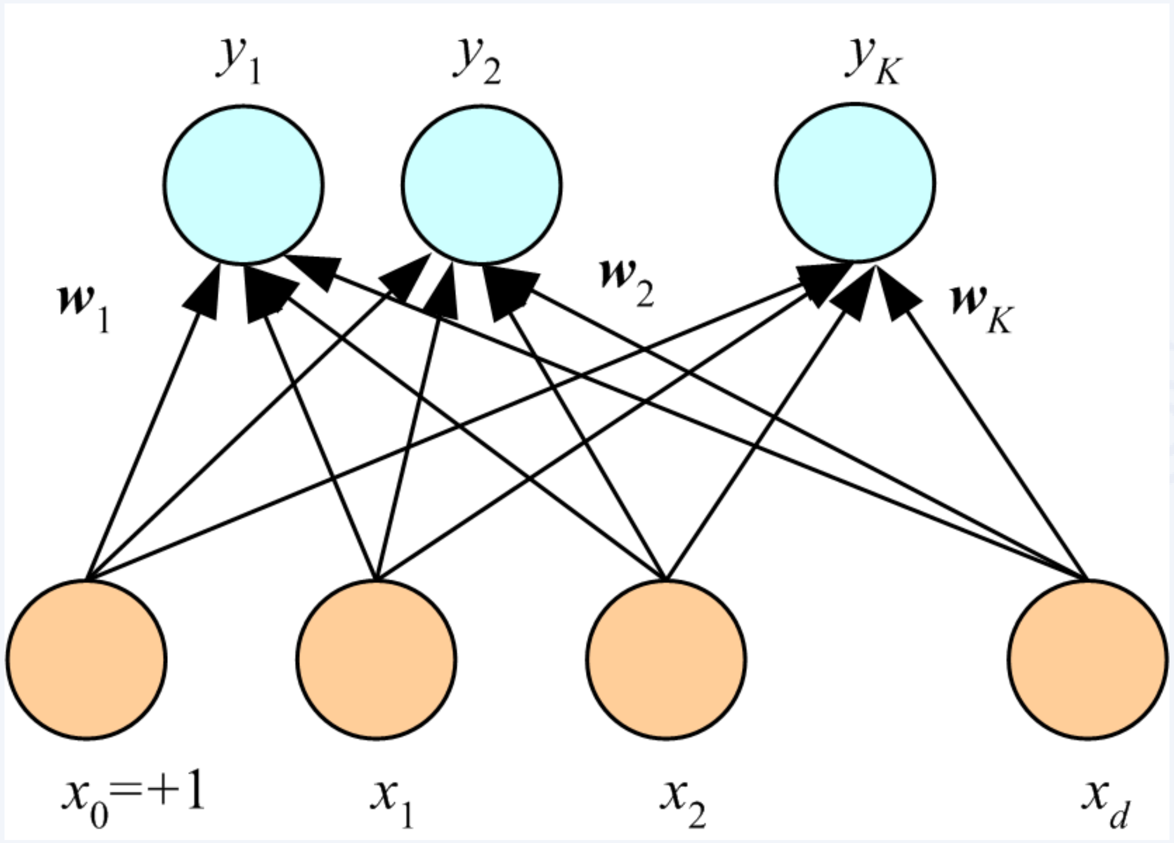
Classification:

$$o_i = \mathbf{w}_i^T \mathbf{x}$$

$$y_i = \frac{\exp o_i}{\sum_k \exp o_k}$$

choose C_i

if $y_i = \max_k y_k$



Learning weights

- **Iterative learning is applied in such algorithms**

1. **Set weights uniformly or randomly**
2. **Calculate errors**
   - Either on full batch of training data on on single instances
3. **Update weights to minimize errors and repeat**

$$\Delta w_{ij}^{t} = \eta\left(r_i^t - y_i^t\right)x_j^t$$

$$\text{Update} = \text{LearningFactor} \cdot \left(\text{DesiredOutput} - \text{ActualOutput}\right) \cdot \text{Input}$$

- **Many names for different ways of doing this:**
  - Gradient descent (delta rule, LMS)
  - Backpropagation (for networks)

Remarks on Regression/ANN

- **Perceptron units can be layered together to form networks**

- **Pros (Networks):**
  - –Robust to noise
  - –Good for high dimensional data

- **Pros (Regression):**
  - –Can predict continuous values

- **Cons:**
  - –Network versions of this can be very slow to train
  - –People generally can't interpret the resulting model

Naïve Bayes

Create a model from the training data:

```
NaïveBayesLearn(examples)
    For each target value $v_j$
        $P'(v_j) \leftarrow$ estimate $P(v_j)$
        For each attribute value $a_i$ of each attribute $a$
            $P'(a_i|v_j) \leftarrow$ estimate $P(a_i|v_j)$
```

Predict:

```
ClassfyingNewInstance(x)
    $v_{nb}$ = argmax $P'(v_j) \prod P'(a_i|v_j)$
```

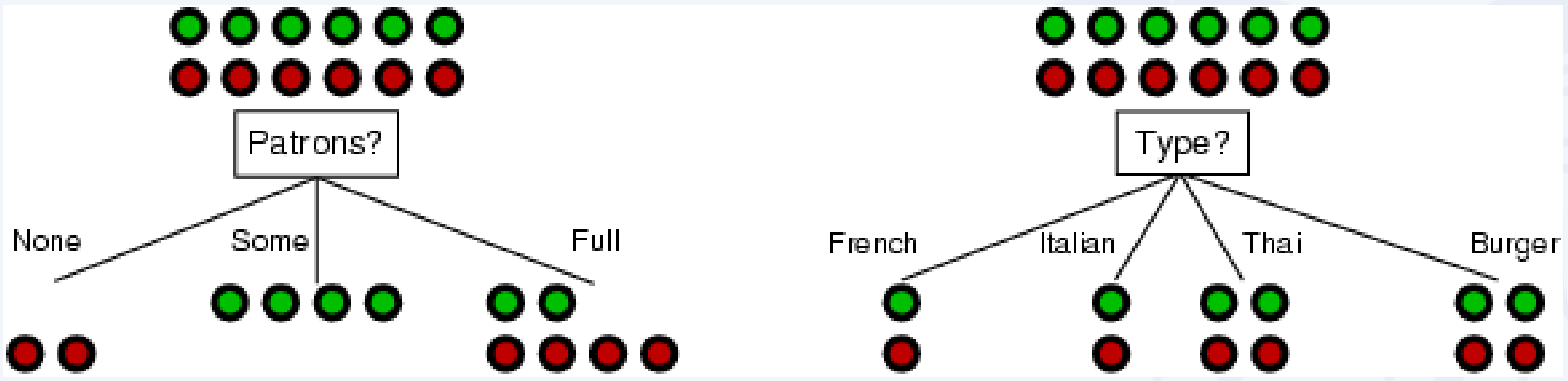
Remarks on Naïve Bayes

- **Very fast to learn and apply**
  - –Decomposes model to
    - a prior distribution of the classes, and
    - posterior distributions of features given a class
  - –A good baseline algorithm to test with

- **Has problems with correlated features**
  - –Assumes independence between features
    - Each feature's probability is simply multiplied through
  - –In practice, this doesn't seem to be too much of a problem

Decision Trees

- Divide and conquer strategy
- Sequentially choose a dimension of x to split on that makes the subproblems as easy as possible
- “easy” = information gain



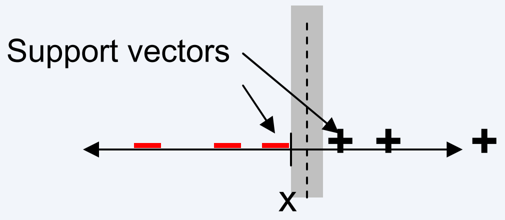
Remarks on Decision Trees

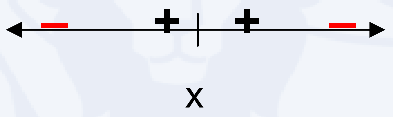
- **Normal training speed; fast testing**
  - –Complexity proportional to |**x**| and # of instances
  - –Need to compute best feature after every new rule
  - –But just need to apply tree rules in testing
- **Pros**
  - –Easy to analyze: people easily understand hypotheses, easier for post-analysis
- **Cons:**
  - –Can overfit data easily
  - –Large inductive bias: considers only on feature at a time
  - –Most methods adopt a version of pruning to give some assurance of the generalizability of its rule

Support Vector Machines

A complex topic, let's just go over the very basic

- **Basic SVMs use a line (hyperplane) to separate the classes**
  - –Draws a line to maximize the margin between the classes
  - –Only care about data instances (support vectors) near the boundary; other instances are not used



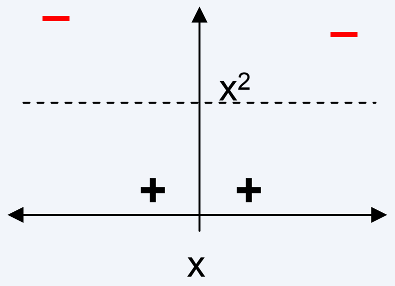


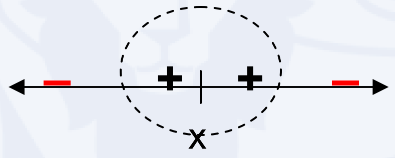
- **Left is linearly separable with one line but the right is not**

Support Vector Machines

Solution: Map the data into a higher dimensional space

- –This is called the **kernel trick**
- –This guarantees that it will be separable, allowing non linear classification
- –Relies on *k*(x,y), a *kernel* function that takes two points in the original input space and calculates their distance





- **The same data set is now separable**

Remarks on SVMs

- **A learner that seems to have good performance for many different scenarios**
- **Sensitive to choice of kernel function**
  - –That is, how to calculate how close two data points are
  - –Variety of kernel functions to try
  - –Sequence data and tree data structures can be compared using different kernels
- **Running time depends heavily on kernel function**

The Maximum Entropy Principle

A type of constraint satisfaction: find a model that fits all of the training data

- **Use an exponential model**

Weight (to be learned)

Features (usually binary-valued)

$$p_s = \frac{\exp\left(-\sum_i \lambda_i f_i(s)\right)}{Z}$$

Normalization (to make it a probability)

- **Given some set of constraints which must hold, what is the best model among those available?**
  - –Answer: the one with maximum entropy
  - –Meaning that it doesn’t assume more than what is necessary

- **Why? ...philosophical answer:**
  - –Occam’s razor, don’t pretend you know something you don’t

Example

- **Throwing the "unknown" die**
  - –do not know anything – we should assume a fair die (uniform distribution ~ max. entropy distribution)

- **Throwing unfair die**
  - –we know: p(4) = 0.4, p(6) = 0.2, nothing else
  - –best distribution?
  - –do not assume anything about the rest:

1	2	3	4	5	6
0.1	0.1	0.1	0.4	0.1	0.2

Remarks: Max Ent

- **Similar in spirit to SVM's max margins**
  - –Make hypothesis as general as possible
- **Features**
  - –Are usually binary valued
  - –Used a lot in sequence labeling tasks
  - –Often encode previous decisions in sequence learning
    - E.g., last word was labeled as an adjective
- **Is the basis for a number of more complex sequence labeling models (more on this later)**
  - –Max. Entropy Markov Models (MEMM)
  - –Conditional Random Fields (CRF)

Recap on Text Classification

- **Use a machine learning technique to assign a document d to a category c**

Some characteristics:

–|D| >> |C|, where there are numerous examples for each C

–Represent each d as a set of features $f_1 \ldots f_n$, typically each w in vocabulary is a feature, weighted by tf.idf

–Results in thousands of features

Curse of Dimensionality

Two problems:

–Some learning methods don’t work well with thousands of features.

–Many datasets don’t have enough examples to generate **sufficient statistics** for features

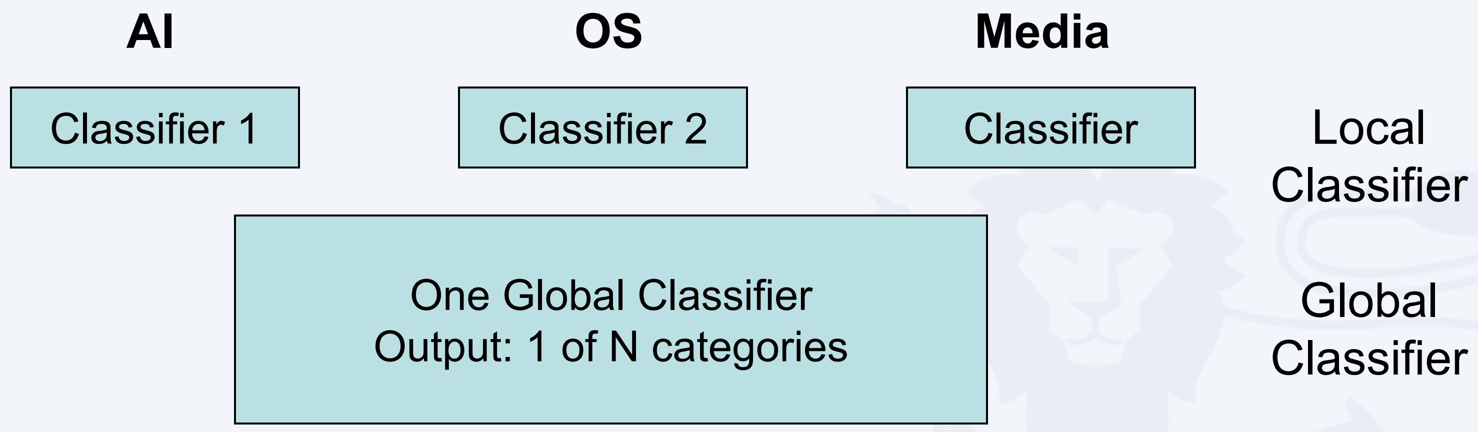
When the statistics can sub for the distribution in inference decisions

Solution?

- **Use dimensionality reduction**
- **Use feature selection**
- **Use appropriate weighting scheme**

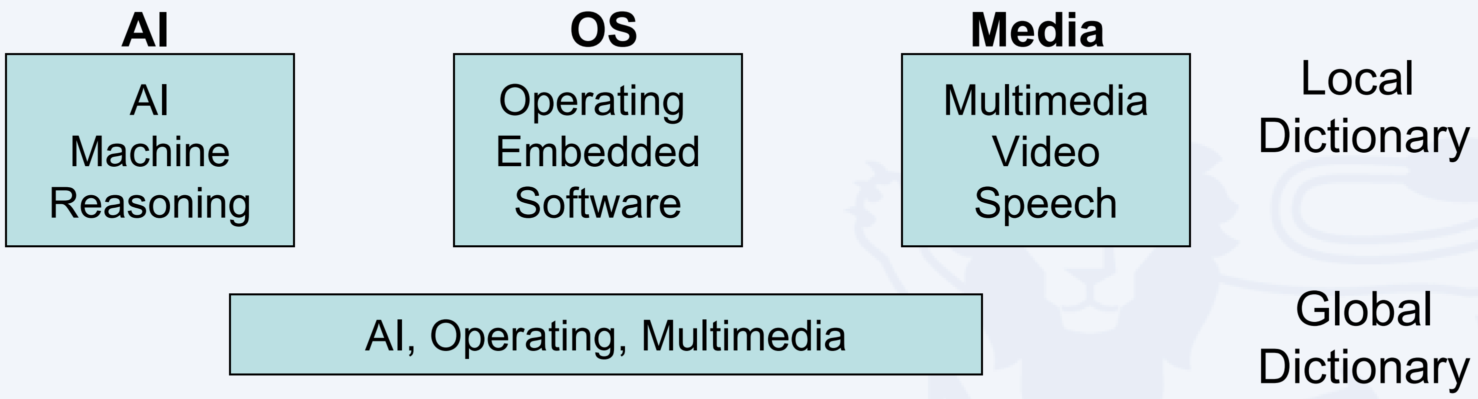
Classification Method

- **Choice of methods (Global vs. local classifier)**
  - –Global: one multi-class classifier
  - –Local: Many binary classifiers, making Y/N decisions



Feature Selection

- **Selecting / eliminating features based on criteria on a feature's (term's) distribution (or weight)**
- **Decision of local vs. global features**
  - –Global: one set of features for one or more classifiers
  - –Local: each classifier uses own (local) features



Choice of features and feature selection method have largest influence on categorization performance.

IR and TC

To think about … carefully

- **IR favors rare features**
  - –Retains all non-trivial terms
  - –Use IDF to select rare features
- **TC needs common features in each category**
  - –DF is more important than IDF

What are the differing characteristics of these two problems?

Feature Selection Methods

- **DF: Document Frequency**
- **IG: Information Gain**
- **MI: Mutual Information**
- **CHI: X^2 statistic**

Term/Class Contingency Table

	T_k=1 (Occurs)	T_k=0 (Absent)
C_i=1 (Relevant)	A	C
C_i=0 (Non-relevant)	B	D

Selection Methods, 1

- **DF: throw away all terms that occur in less than n documents**
  - –Equate noise with rare terms
  - –But IR assumes such rare terms can indicate content, so we typically don't set this too aggressively

- **IG: measure number of bits of information that can be used for category prediction**

$$G(t) = -\sum_{i=1}^{m} P(c_i)\log P(c_i)$$
$$+ P(t)\sum_{i=1}^{m} P(c_i \mid t)\log P(c_i \mid t)$$
$$+ P(\bar{t})\sum_{i=1}^{m} P(c_i \mid \bar{t})\log P(c_i \mid \bar{t})$$

Constant across all features

Bias when present

Bias when absent

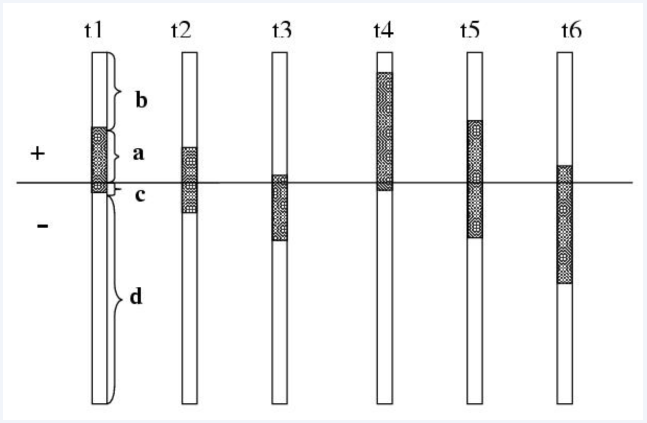
Weighting of features

- **Feature weighting plays a role in certain types of classifiers: SVM, kNN.**
  - –What about NB?
- **Support Vector Machines shown to be competitive in accuracy in classification**
  - –Shown to be attributable more to text representation than kernel function (Leopold 02)

Weighting schemes

- TF
- Log TF
- ITF

- IDF
- TF.IDF
- Log TF.IDF



	$T_k=1$	$T_k=0$
$C_i=1$	A	C
$C_i=0$	B	D

- TF.CHI
- TF.RF

Sensitive to Classification

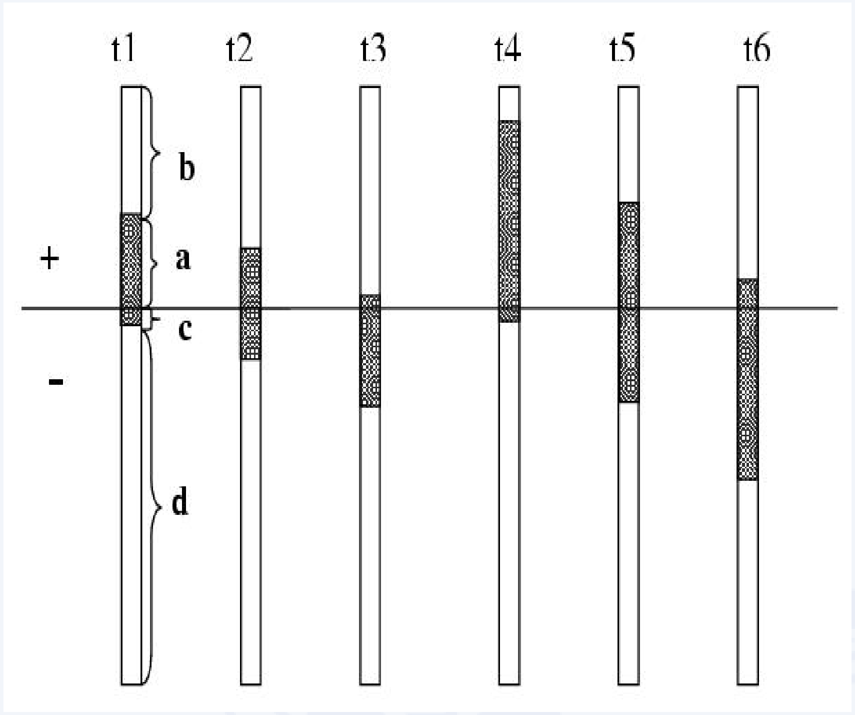
RF = log (2+a/c)

IDF = N/(a+c)

CHI =
$$\frac{N(ab-bc)^2}{(a+c)(a+b)(b+d)(c+d)}$$

Relevant Frequency

- First three = idf_1
  last three = idf_2
- RF = ratio of a to c as important, while taking into account relative rarity of term



To think about: how is this different from CHI? From IG? From MI?

Hands on with SVM^{light}

Text classification over Reuters-21578

SVM^{light}

- SVM package from Joachims (Cornell)
- Many competing packages (some embedded as libraries within frameworks)

- Deals with sparse vector format

- Data is just a set of instances:

```
class_label (feature_index:feature_value)+ # comment
```

Seven Steps

- **Reuse NLTK from Day 1**

1. **Inspect Reuters corpus from within NLTK**
2. **Build vocabulary list**
3. **Create initial vectors**
    1. Use TF as weight
    2. Set up simple training and testing splits
    3. Run SVM^{light}
4. **Create normalized, stemmed vectors**
5. **Use IDF or TF.IDF for vector weighting**
6. **Use bigram features**
7. **Build n classifiers for each category**

Summary

- Many machine learners to pick from
- Difficulty comes in picking useful features to provide to the classifier
- Weighting word features has significant impact on performance

Looking Ahead

Day 1

AM

- Applications' Input / Output
- Resources

PM

- Selected Toolkits
- Python Intro
- NLTK Hands-on

Day 2

AM

- Evaluation
- Annotation
- Information Retrieval
- ML Intro

PM

- Machine Learning
- SVM Hands-on

>>Day 3

AM

- Sequence Labeling
- CRF++ Hands-on

PM

- Dimensionality Reduction
- Clustering
- Trends & Issues