Introduction to Data Science
Lecture 8
3 ½ more Basic Algorithms

CS 194 Fall 2014

John Canny

Outline for this Evening

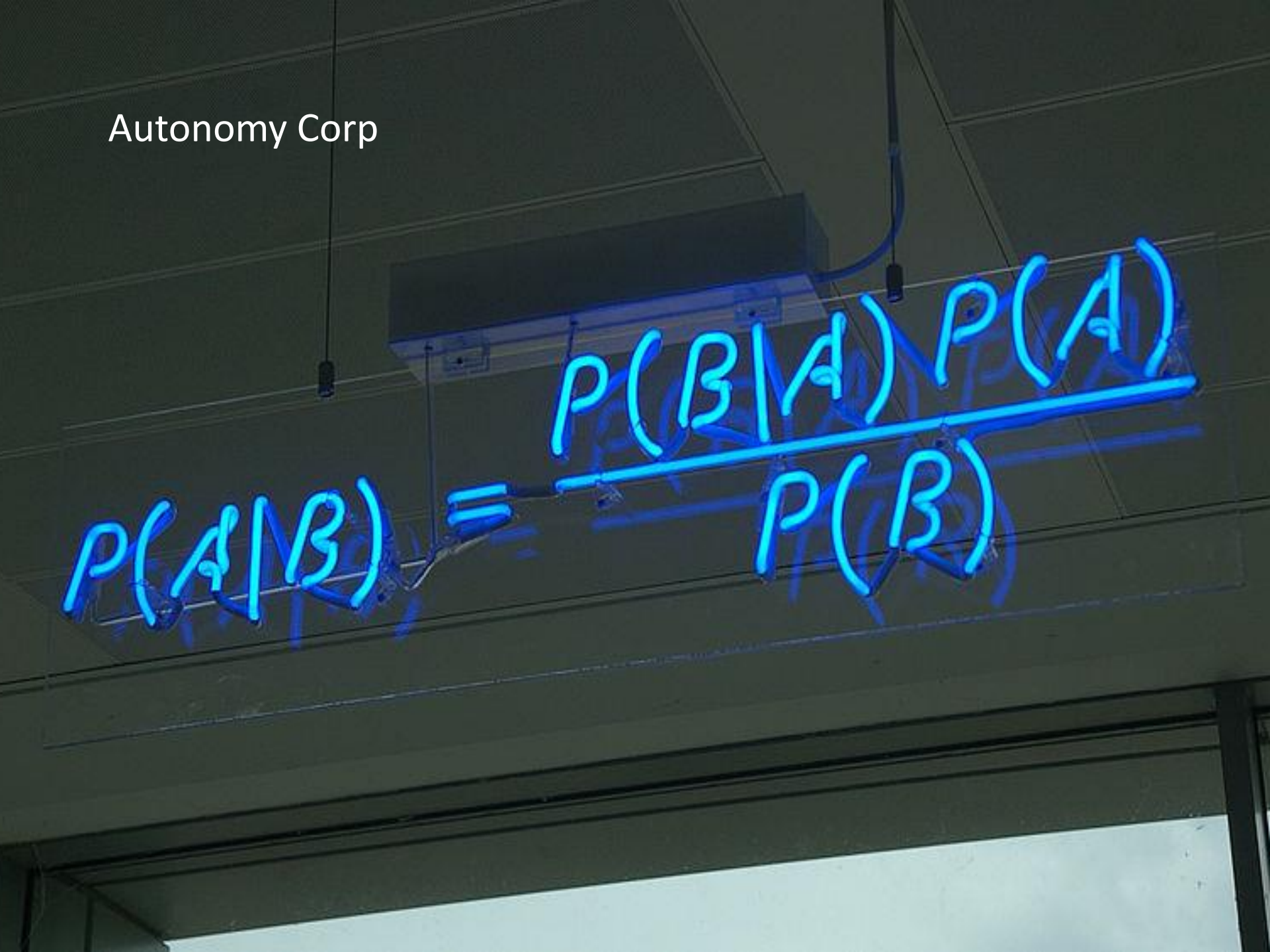
- Three (+ ½) more Basic Algorithms
  - Naïve Bayes
  - Logistic Regression (+ ½ SVM)
  - Random Forests
- Evaluation
  - Precision/Recall
  - Accuracy + weighted loss
  - ROC and AUC
  - Lift

Techniques

- **Supervised Learning:**
  - kNN (k Nearest Neighbors)
  - Linear Regression
  - Naïve Bayes
  - Logistic Regression
  - Support Vector Machines
  - Random Forests
- **Unsupervised Learning:**
  - Clustering
  - Factor analysis
  - Topic Models

Techniques

- **Supervised Learning:**
  - kNN (k Nearest Neighbors)
  - Linear Regression
  - Naïve Bayes
  - Logistic Regression
  - Support Vector Machines
  - Random Forests
- **Unsupervised Learning:**
  - Clustering
  - Factor analysis
  - Topic Models


Autonomy Corp
$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$$

Bayes' Theorem

P(A|B) = probability of A given that B is true.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

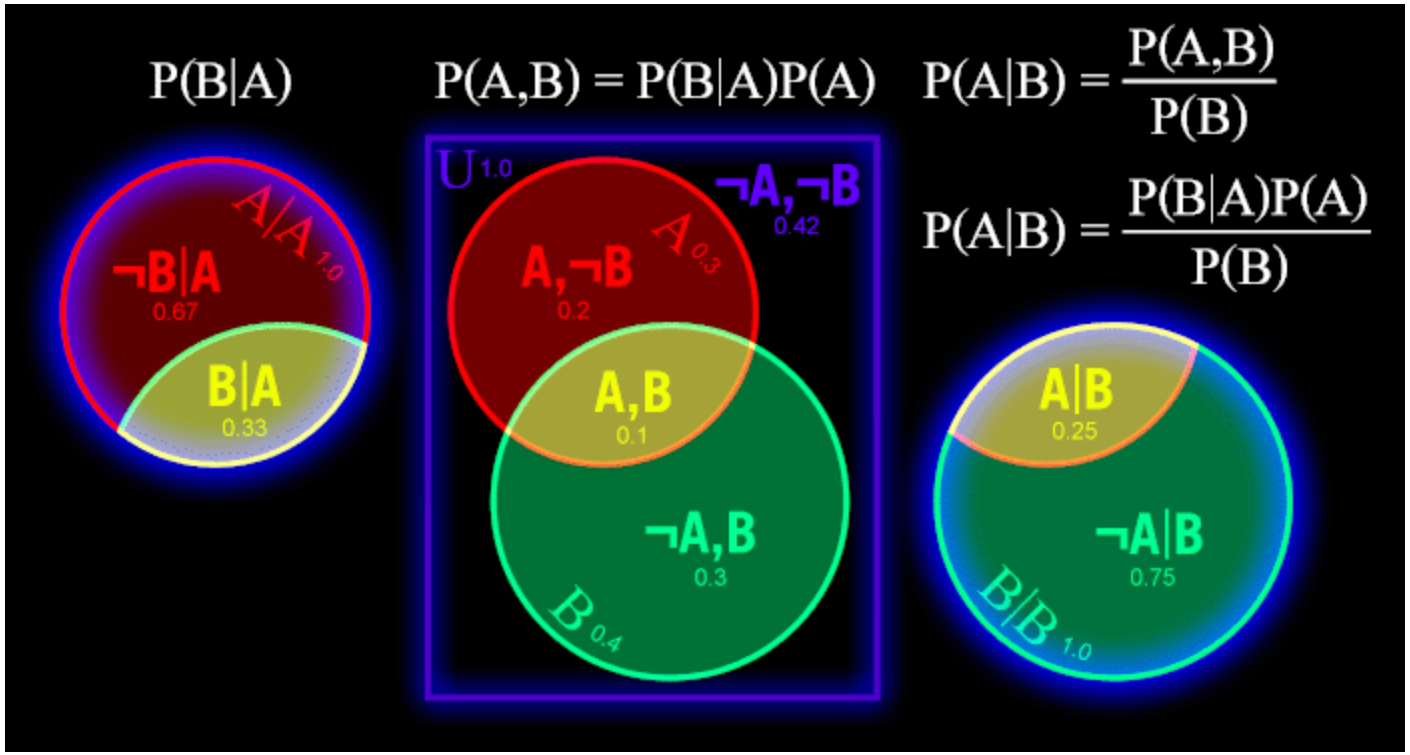
In practice we are most interested in dealing with events e and data D.

e = "I have a cold"
D = "runny nose," "watery eyes," "coughing"

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

So Bayes' theorem is "diagnostic".

Bayes' Theorem



Bayes' Theorem

D = Data, e = some event

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

P(e) is called the **prior probability** of e. Its what we know (or think we know) about e with no other evidence.

P(D|e) is the **conditional probability** of D given that e happened, or just the likelihood of D. This can often be measured or computed precisely – it follows from your model assumptions.

P(e|D) is the **posterior probability** of e given D. It's the answer we want, or the way we choose a best answer.

You can see that the posterior is heavily colored by the prior, so Bayes' has a GIGO liability. e.g. its not used to test hypotheses

Naïve Bayes Classifier

Let's assume we have an instance (e.g. a document d) with a set of features $(X_1, \dots, X_k)$ and a set of classes $\{c_j\}$ to which the document might belong.

We want to find the **most likely class** that the document belongs to, given its features.

The joint probability of the class and features is:

$$\Pr(X_1, \dots, X_k, c_j)$$

Naïve Bayes Classifier

Key Assumption: (Naïve) the features are **generated independently** given c_j. Then the joint probability factors:

$$\Pr(X, c_j) = \Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j) = \Pr(c_j) \prod_{i=1}^{k} \Pr(X_i | c_j)$$

We would like to figure out the **most likely class for** (i.e. to classify) the document, which is the c_j which maximizes:

$$\Pr(c_j \mid X_1, \ldots, X_k)$$

Naïve Bayes Classifier

Now from Bayes we know that:

$$\Pr(c_j \mid X_1, \ldots, X_k) = \Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j) \,/\, \Pr(X_1, \ldots, X_k)$$

But to choose the best c_j, we can ignore $\Pr(X_1, \ldots, X_k)$ since it's the same for every class. So we just have to maximize:

$$\Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j)$$

So finally we pick the category c_j that maximizes:

$$\Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j) = \Pr(c_j) \prod_{i=1}^{k} \Pr(X_i \mid c_j)$$

Naïve Bayes Classifier

Now from Bayes we know that:

$$\Pr(c_j \mid X_1, \ldots, X_k) = \Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j) / \Pr(X_1, \ldots, X_k)$$

A B B A A B

But to choose the best c_j, we can ignore $\Pr(X_1, \ldots, X_k)$ since it's the same for every class. So we just have to maximize:

$$\Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j)$$

So finally we pick the category c_j that maximizes:

$$\Pr(X_1, \ldots, X_k \mid c_j) \Pr(c_j) = \Pr(c_j) \prod_{i=1}^{k} \Pr(X_i \mid c_j)$$

Data for Naïve Bayes

In order to find the best class, we need two pieces of data:

- $\Pr(c_j)$ the prior probability for the class c_j.
- $\Pr(X_i|c_j)$ the conditional probability of the feature X_i given the class c_j.

Data for Naïve Bayes

For these two data, we only need to record counts:

- $\Pr(c_j) = \frac{N_d(c_j)}{N_d}$

- $\Pr(X_i|c_j) = \frac{N_w(X_i,c_j)}{N_w(c_j)}$

Where $N_d(c_j)$ is the number of documents in class c_j, N_d is the total number of documents.

$N_w(X_i, c_j)$ is the number of times X_i occurs in a document in c_j, and and $N_w(c_j)$ is the total number of features in all docs in c_j.

"Training" Naïve Bayes

So there is no need to train a Naïve Bayes classifier, only to accumulate the N_w and N_d counts.

But count data is only an approximation to those probabilities however, and a count of zero is problematic (why)?

Instead of direct count ratios, we can use **Laplace Smoothing**:

$$p = \frac{N_1 + \alpha}{N_2 + \beta}$$

With constants α and β. This reflects another layer of Bayesian inference, and pushes p toward a prior of α/β.

These constants can either be set based on prior knowledge, or learned during a training phase.

Good, Bad and Ugly of NB Classifiers

- **Simple and fast.** Depend only on term frequency data for the classes. One shot, no iteration.
- **Very well-behaved numerically.** Term weight depends only on frequency of that term. Decoupled from other terms.
- **Can work very well with sparse data**, where combinations of dependent terms are rare.
- **Subject to error and bias** when term probabilities are not independent (e.g. URL prefixes).
- **Can't model patterns** in the data.
- **Typically not as accurate** as other methods.

Logistic Regression

- We made a distinction earlier between regression (predicting a real value) and classification (predicting a discrete value).

- Logistic regression is designed as a **binary classifier** (output say {0,1}) but actually **outputs the probability** that the input instance is in the "1" class.

- A logistic classifier has the form:

$$p(X) = \frac{1}{1 + \exp(-X\beta)}$$

where $X = (X_1, \ldots, X_n)$ is a vector of features.

Logistic Regression

- Logistic regression is probably the **most widely used general-purpose classifier**.

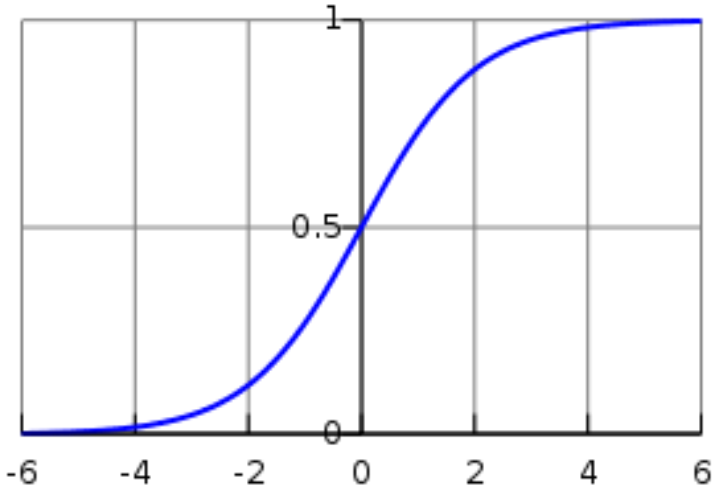
- Its **very scalable** and can be **very fast** to train. It's used for
  - Spam filtering
  - News message classification
  - Web site classification
  - Product classification
  - Most classification problems with large, sparse feature sets.

- The only caveat is that **it can overfit** on very sparse data, so its often used with Regularization

Logistic Regression

- Logistic regression maps the "regression" value $-X\beta$ in (-∞,∞) to the range [0,1] using a "logistic" function:

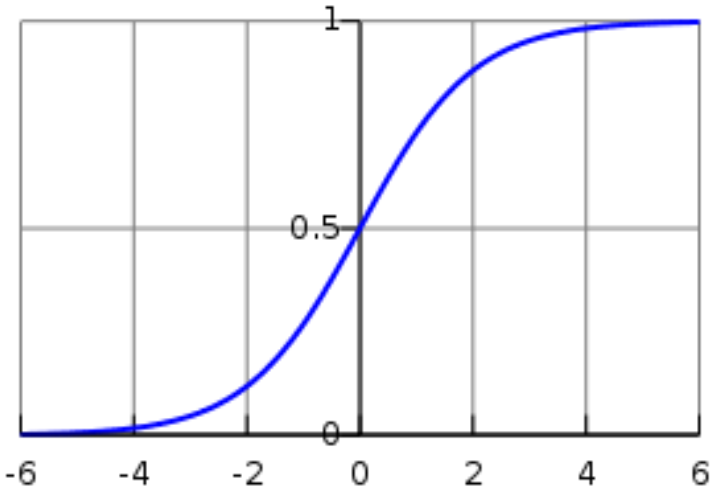
$$p(X) = \frac{1}{1 + \exp(-X\beta)}$$



- i.e. the logistic function maps any value on the real line to a probability in the range [0,1]

Logistic Regression

- Where did the logistic function come from?



Logistic Regression and Naïve Bayes

- Logistic regression is actually **a generalization of Naïve Bayes**, with binary features.
- Logistic Regression can model a Naïve Bayes classifier when the binary features are **independent**.
- Bayes rule for two classes c and $\neg c$:

$$\Pr(c|X) = \frac{\Pr(X|c)\Pr(c)}{\Pr(X)} = \frac{\Pr(X|c)\Pr(c)}{\Pr(X|c)\Pr(c) + \Pr(X|\neg c)\Pr(\neg c)}$$

- Dividing by the numerator:

$$= \frac{1}{1 + \frac{\Pr(X|\neg c)\Pr(\neg c)}{\Pr(X|c)\Pr(c)}}$$

Logistic Regression and Naïve Bayes

We have

$$\Pr(c|X) = \frac{1}{1 + \dfrac{\Pr(X|\neg c)\Pr(\neg c)}{\Pr(X|c)\Pr(c)}} = \frac{1}{1 + \exp(-X\beta)}$$

which matches if

$$\frac{\Pr(X|\neg c)\Pr(\neg c)}{\Pr(X|c)\Pr(c)} = \exp(-X\beta)$$

and assuming feature independence, the LHS factors:

$$\prod_{i=1}^{n} \frac{\Pr(X_i|\neg c)}{\Pr(X_i|c)} \frac{\Pr(\neg c)}{\Pr(c)} = \prod_{i=1}^{n} \exp(-X_i\,\beta_i)\,\beta_0$$

And we can match corresponding (i) terms to define β.

Logistic Regression and Naïve Bayes

Summary: Logistic regression has this form:

Models Naïve Bayes formula with two classes
after dividing through by one of them

$$\Pr(c|X) = \frac{1}{1 + \exp(-X\beta)}$$

Models product of contributions
from different (independent) features

Logistic Regression and Naïve Bayes

- Because it can always learn an NB model but is more general, Logistic regression should **do at least as well as** naïve Bayes*

- Logistic regression **typically does better** though because it can deal with **dependencies** between features, whereas Naïve Bayes cannot.

* - this may not be true if the Logistic model is overfit.
L1 regularization is often used with LR to avoid overfitting.

Logistic Training

For training, we start with a collection of **input values** X^i and corresponding **output labels** $y^i \in \{0,1\}$. Let p^i be the **predicted output** on input X^i, so

$$p^i = \frac{1}{1 + \exp(-X^i \beta)}$$

The **accuracy** on an input X^i is

$$A^i = y^i p^i + (1 - y^i)(1 - p^i)$$

Logistic regression maximizes either the sum of the log accuracy, or the total accuracy, e.g.

$$A = \sum_{i=1}^{N} A^i$$

Logistic Training

To find the best β, we can use gradient ascent on the derivative $^{dA}/_{d\beta}$ where

$$A = \sum_{i=1}^{N} A^i$$

The gradient is

$$\frac{dA}{d\beta} = \sum_{i=1}^{N} \left(2y^i - 1\right) p^i \left(1 - p^i\right) X^i$$

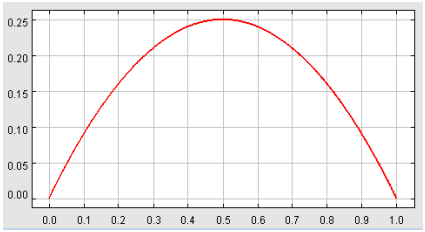
Logistic Training

The gradient is a weighted sum of input vectors X^i.

$$\frac{dA}{d\beta} = \sum_{i=1}^{N} \underbrace{(2y^i - 1)}_{\{-1,+1\}} \underbrace{p^i(1-p^i)}_{[0,0.25]} X^i$$

Positive instances (label $y^i = 1$) get weight $p^i(1-p^i)$ while negative instance get weight $-p^i(1-p^i)$

The weight $p^i(1-p^i)$ is largest for inputs with $p \approx 0.5$, which are **near the decision boundary**.

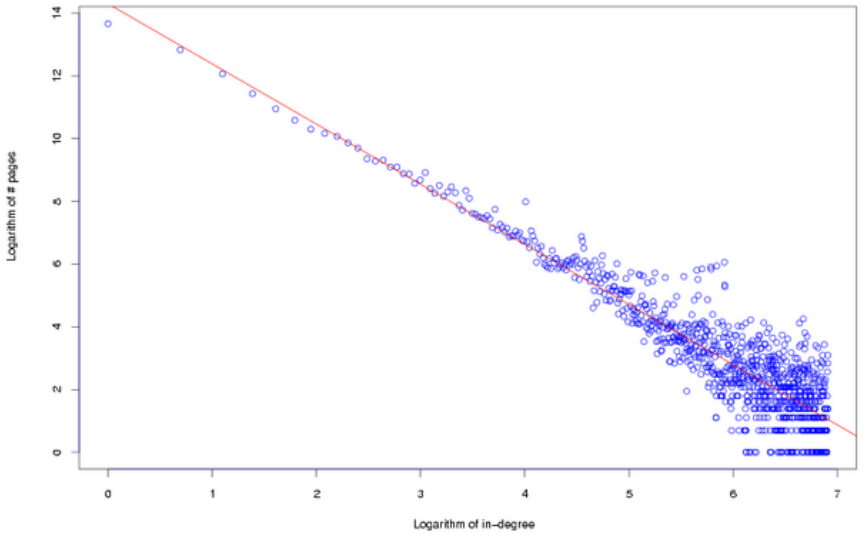


y = p (1-p)

Logistic Regression Training - SGD

- A very efficient way to train logistic models is with **Stochastic Gradient Descent** (SGD) – we keep updating the model with gradients from small blocks (mini-batches) of input data.

- One challenge with training on power law data (i.e. most data) is that the terms in the gradient can have very different strengths (because of the power law distribution)



Logistic Regression Training - ADAGRAD

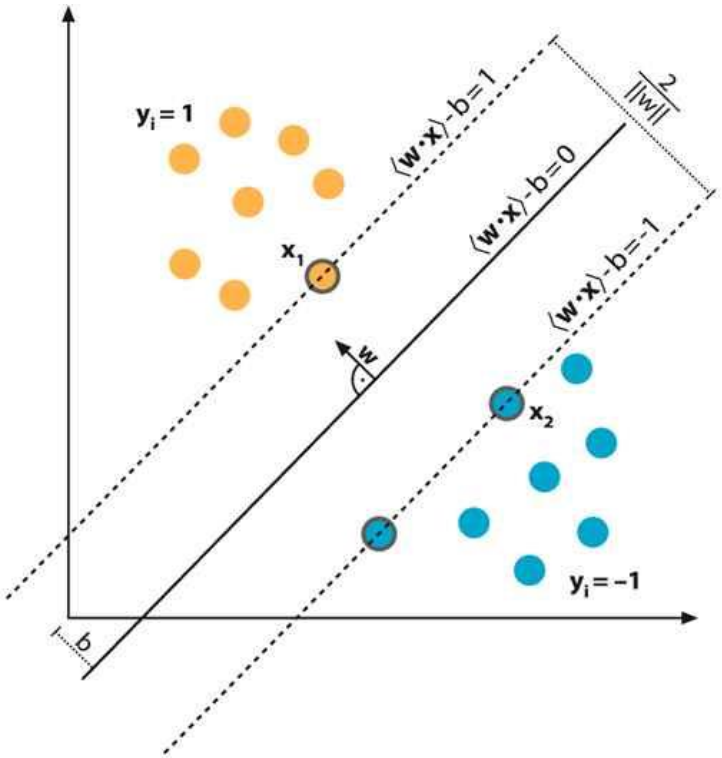
- This means that some gradient terms are 1000's of times larger than others, but the β coefficients are about the same. This imbalance **makes training very slow**.

- A recent method called ADAGRAD normalizes each coordinate of gradient by the historical (from previous iterations) magnitude of that coordinate.

- ADAGRAD often leads to extremely efficient training. On large datasets its not unusual for a model to converge **in less than one pass over the dataset**. ("less" than Naïve Bayes!)

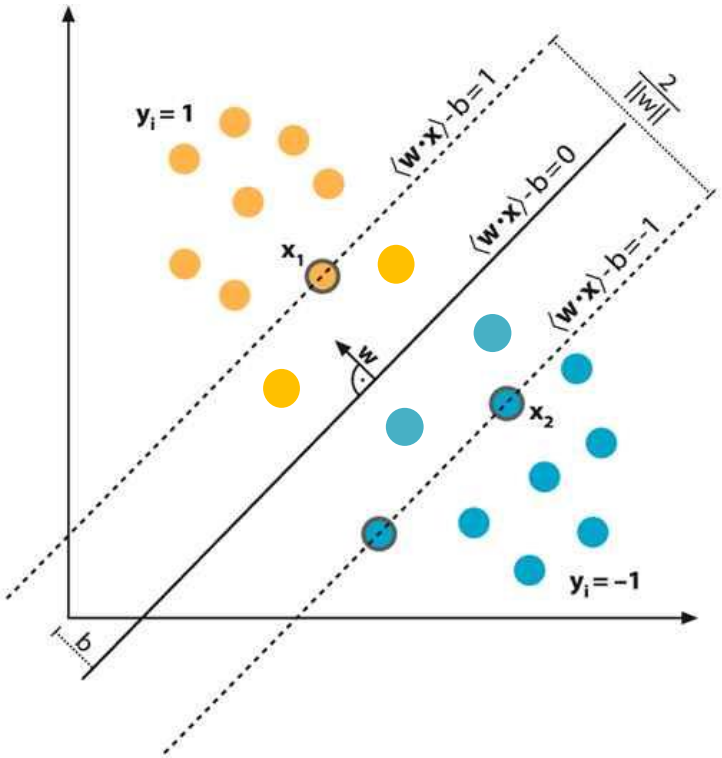
Support Vector Machines

- A Support Vector Machine (SVM) is a classifier that tries to **maximize the margin** between training data and the classification boundary (the plane defined by $X\beta = 0$)



Support Vector Machines

- The idea is that maximizing the margin **maximizes the chance that classification will be correct on new data**. We assume the new data of each class is near the training data of that type.



SVM Training

SVMs can be trained using SGD. Recall that the Logistic gradient was (this time **assuming $y^i \in \{-1, +1\}$**):

$$\frac{dA}{d\beta} = \sum_{i=1}^{N} y^i p^i \left(1 - p^i\right) X^i$$

The **SVM gradient can be defined** as (here $p^i = X^i \beta$)

$$\frac{dA}{d\beta} = \sum_{i=1}^{N} \text{if } \left(p^i y^i < 1\right) \text{ then } y^i X^i \text{ else } 0$$

The expression $\left(p^i y^i < 1\right)$ tests whether the point X^i is in the margin, and if so adds it with sign y^i. It ignores other points.

Both methods weight points "near the middle" with sign y^i.

SVM Training

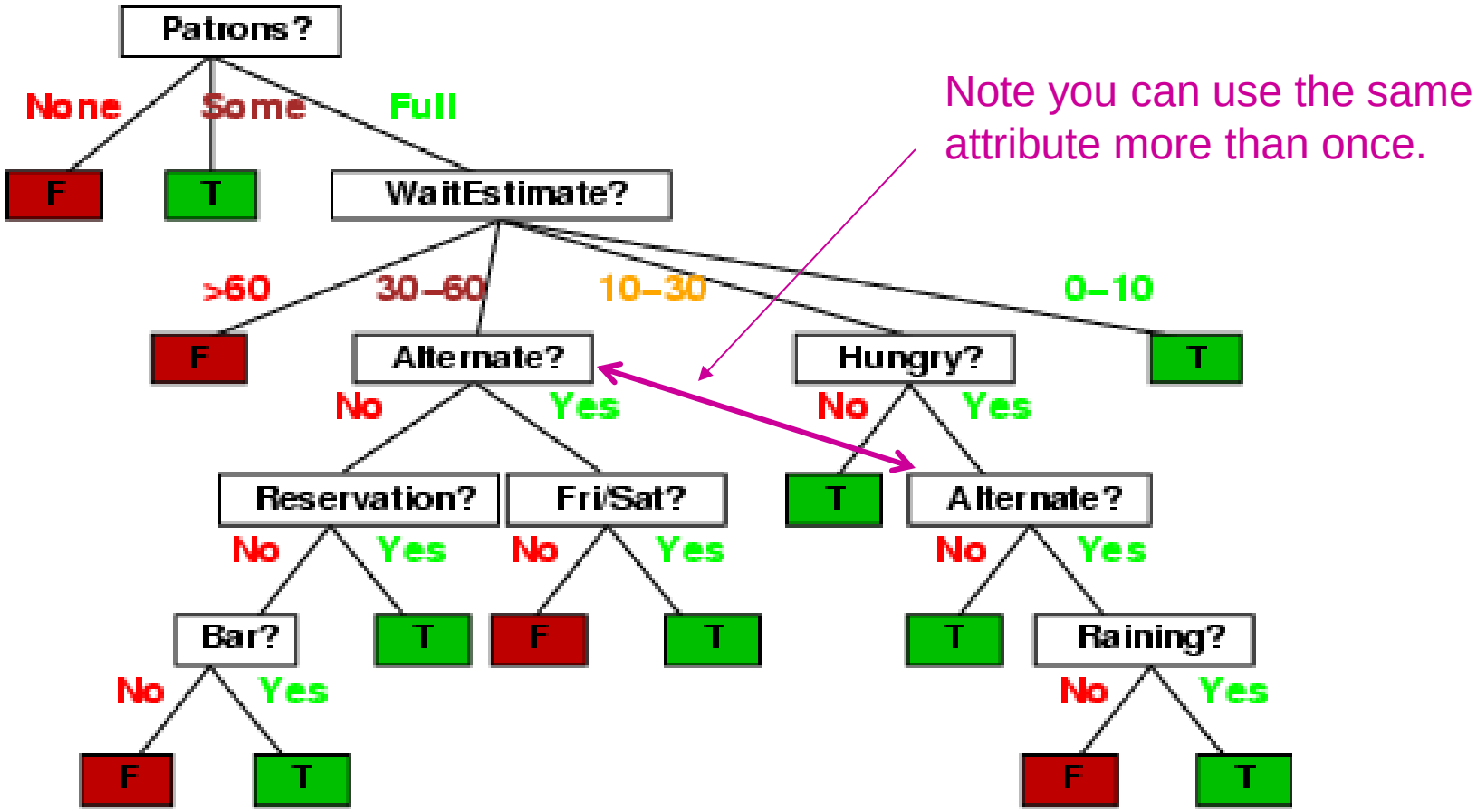
This SGD training method (called Pegasos) is much faster than previous methods, and competitive with Logistic Regression.

Its also capable of training in less than one pass over a dataset.

We'll try some of these in Lab 7.

Decision trees

- Walk from root to a class-labeled leaf.
- At each node, branch based on the value of some feature.

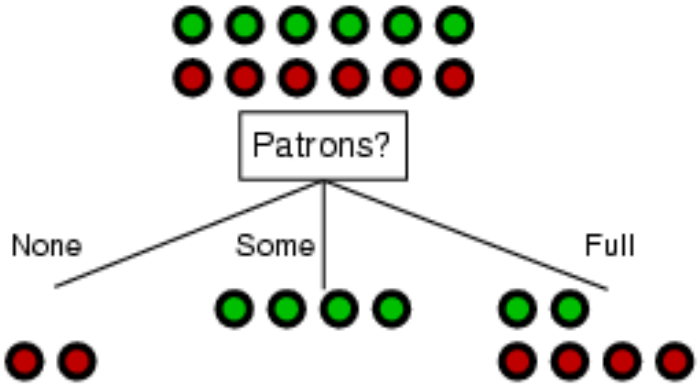


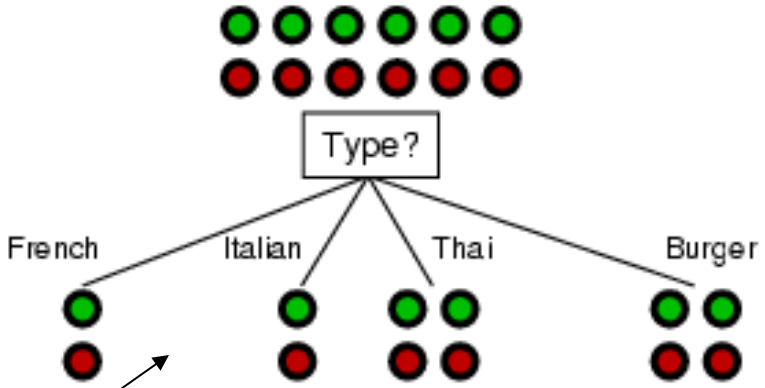
Decision tree learning

- If there are k features, a decision tree might have up to 2^k nodes. This is usually much too big in practice.
- We want to find "efficient" (smaller) trees.
- We can do this in a **greedy manner by recursively choosing a best split feature at each node**.

Choosing an attribute

- Idea: a good features splits the examples into subsets that are (ideally) "all positive" or "all negative"





- *Patrons or type?*

To wait or not to wait is still at 50%.

Using Information Theory

Entropy is defined at each node based on the class breakdown:

- Let p_i be the fraction of examples in class i.
- Let p_i^f be the fraction of elements with feature f that lie in class i.
- Let $p_i^{\neg f}$ be the fraction of elements without feature f that lie in class i

Finally let p^f and $p^{\neg f}$ be the fraction of nodes with (respectively without) feature f

Information Gain

Before the split by f, entropy is

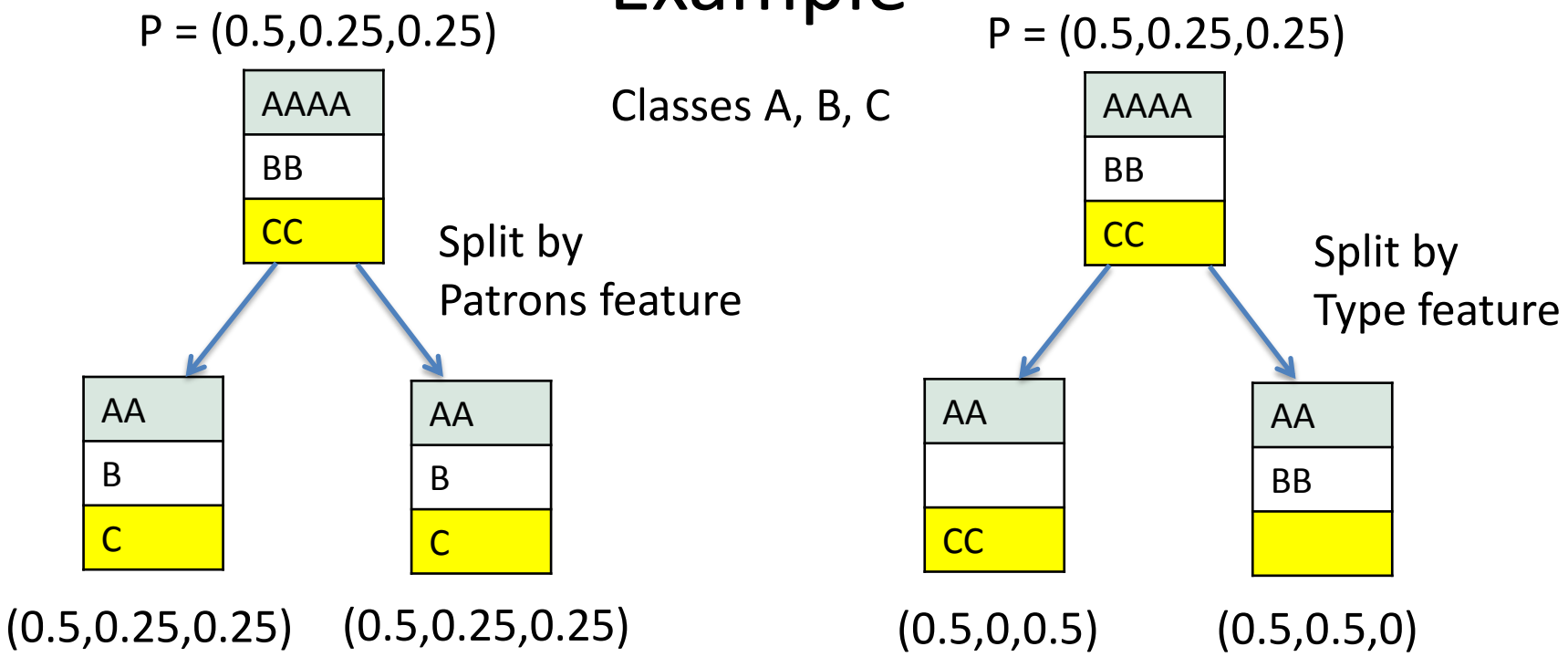
$$E = -\sum_{i=1}^{m} p_i \log p_i$$

After split by f, the entropy is

$$E_f = -p^f \sum_{i=1}^{m} p_i^f \log p_i^f - p^{\neg f} \sum_{i=1}^{m} p_i^{\neg f} \log p_i^{\neg f}$$

The information gain = $E - E_f$ (information = -entropy)

Example



$E = -\sum p_i \log p_i$ =

0.5*1+0.25*2+0.25*2 = 1.5 bits

After:

E_f = (0.5+0.5)*1.5= 1.5 bits

No gain!

Before: E = 1.5 bits

After:

E_f = (0.5+0.5)*1 bits = 1 bits

Gain = $E - E_f$ = 0.5 bits

Choosing best features

At each node, we choose the feature f which **maximizes the information gain**.

This tends to be produce mixtures of classes at each node that **are more and more "pure"** as you go down the tree.

If a node has examples all of one class c, we make it a leaf and output "c". Otherwise, when we hit the depth limit, we output **the most popular class** at that node.

Ensemble Methods

Are like **Crowdsourced machine learning algorithms**:

- Take a collection of simple or *weak* learners
- Combine their results to make a single, better learner

Types:

- **Bagging:** train learners in parallel on different samples of the data, then combine by voting (discrete output) or by averaging (continuous output).
- **Stacking:** combine model outputs using a second-stage learner like linear regression.
- **Boosting:** train learners on the filtered output of other learners.

Random Forests

Grow K trees on datasets sampled from the original dataset with replacement (bootstrap samples), p = number of features.

- Draw K bootstrap samples of size N
- Grow each Decision Tree, by selecting a random set of m out of p features at each node, and choosing the best feature to split on.
- Aggregate the predictions of the trees (most popular vote) to produce the final class.

Typically m might be e.g. sqrt(p) but can be smaller.

Random Forests

Principles: we want to take a vote between different learners so we don't want the models to be too similar. These two criteria ensure diversity in the individual trees:

- Draw K bootstrap samples of size N:
  - Each tree is trained on different data.
- Grow a Decision Tree, by selecting a random set of m out of p features at each node, and choosing the best feature to split on.
  - Corresponding nodes in different trees (usually) cant use the same feature to split.

Random Forests

- **Very popular in practice,** probably the most popular classifier for dense data (<= a few thousand features)
- **Easy to implement** (train a lot of trees). Good match for MapReduce.
- **Parallelizes easily** (but not necessarily efficiently).
- **Not quite state-of-the-art accuracy** – boosted trees generally do better – or DNNs.
- **Needs many passes over the data** – at least the max depth of the trees. (<< boosted trees though)
- **Easy to overfit** – hard to balance accuracy/fit tradeoff.

5 minute break

Outline for this Evening

- Three (+ ½) more Basic Algorithms
  - Naïve Bayes
  - Logistic Regression (+ ½ SVM)
  - Random Forests
- **Evaluation**
  - **Precision/Recall**
  - **Accuracy + weighted loss**
  - **ROC and AUC**
  - **Lift**

Precision and Recall

When evaluating a search tool or a classifier, we are interested in at least two performance measures:

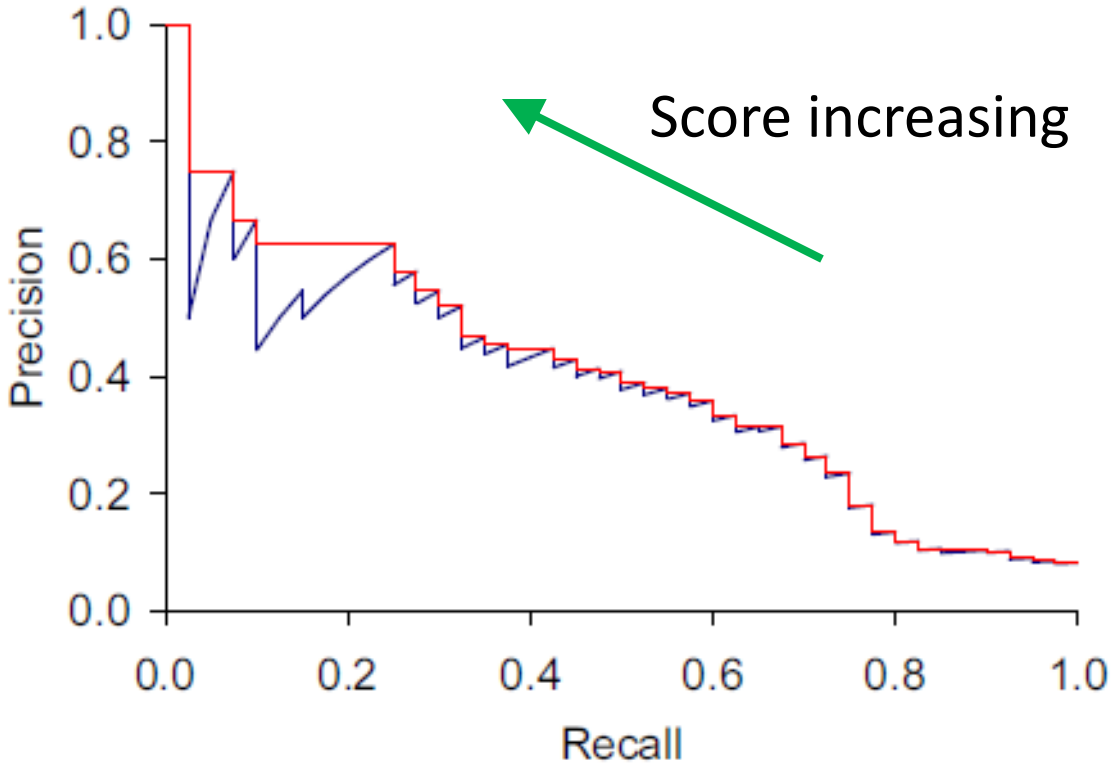
Precision: Within a given set of positively-labeled results, the fraction that were true positives = tp/(tp + fp)

Recall: Given a set of positively-labeled results, the fraction of all positives that were retrieved = tp/(tp + fn)

Positively-labeled means judged “relevant” by the search engine or labeled as in the class by a classifier. tp = true positive, fp = false positive etc.

Precision and Recall

Search tools and classifiers normally assign scores to items. Sorting by score gives us a precision-recall plot which shows what performance would be for different score thresholds.



Be careful of “Accuracy”

The simplest measure of performance would be the fraction of items that are correctly classified, or the “accuracy” which is:

$$\frac{tp + tn}{tp + tn + fp + fn}$$

But this measure is dominated by the larger set (of positives or negatives) and favors trivial classifiers.

e.g. if 5% of items are truly positive, then a classifier that always says “negative” is 95% accurate.

Weighted loss

We can instead try to minimize a weight sum:

$$w_1 \, \mathrm{fn} + \; w_2 \, \mathrm{fp}$$

And typically $w_1 \gg \; w_2$, since positives are often much rarer (clicks or purchases or viewing a movie).

The weighted “F” measure

A measure that naturally combines precision and recall is the β-weighted F-measure:

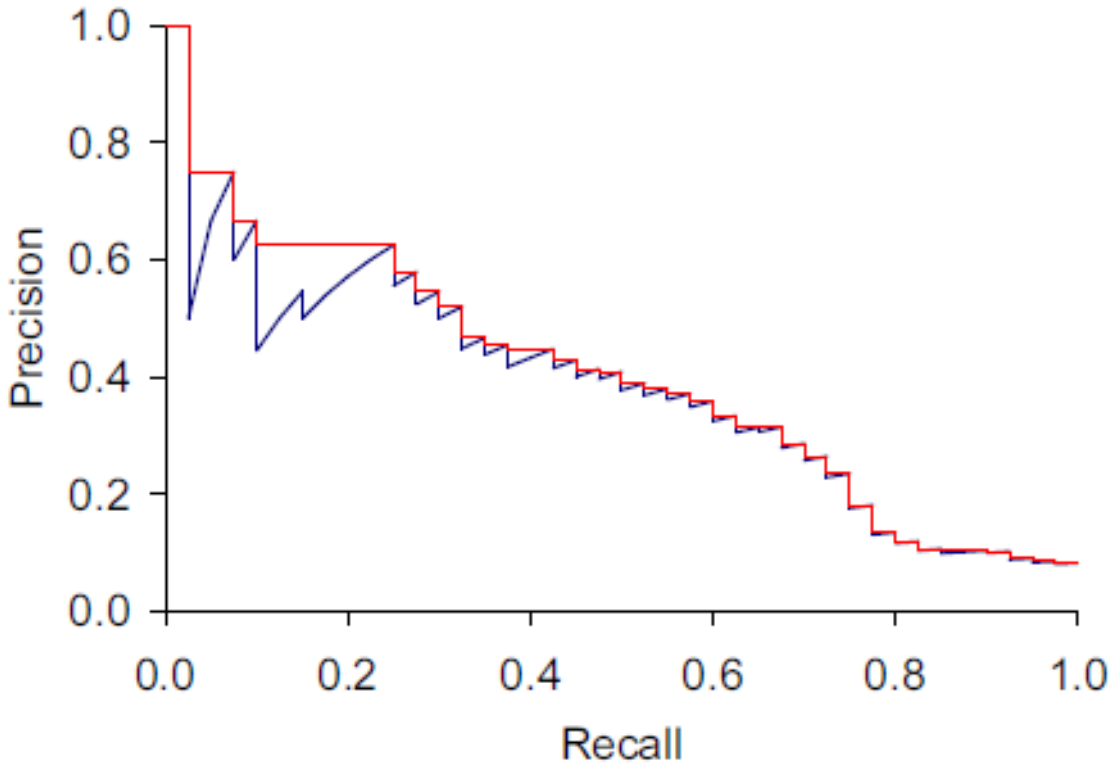
$$F = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}$$

Which is the weighted harmonic mean of precision and recall. Setting $\beta = 1$ gives us the F_1 – measure. It can also be computed as:

$$F_{\beta=1} = \frac{2PR}{P + R}$$

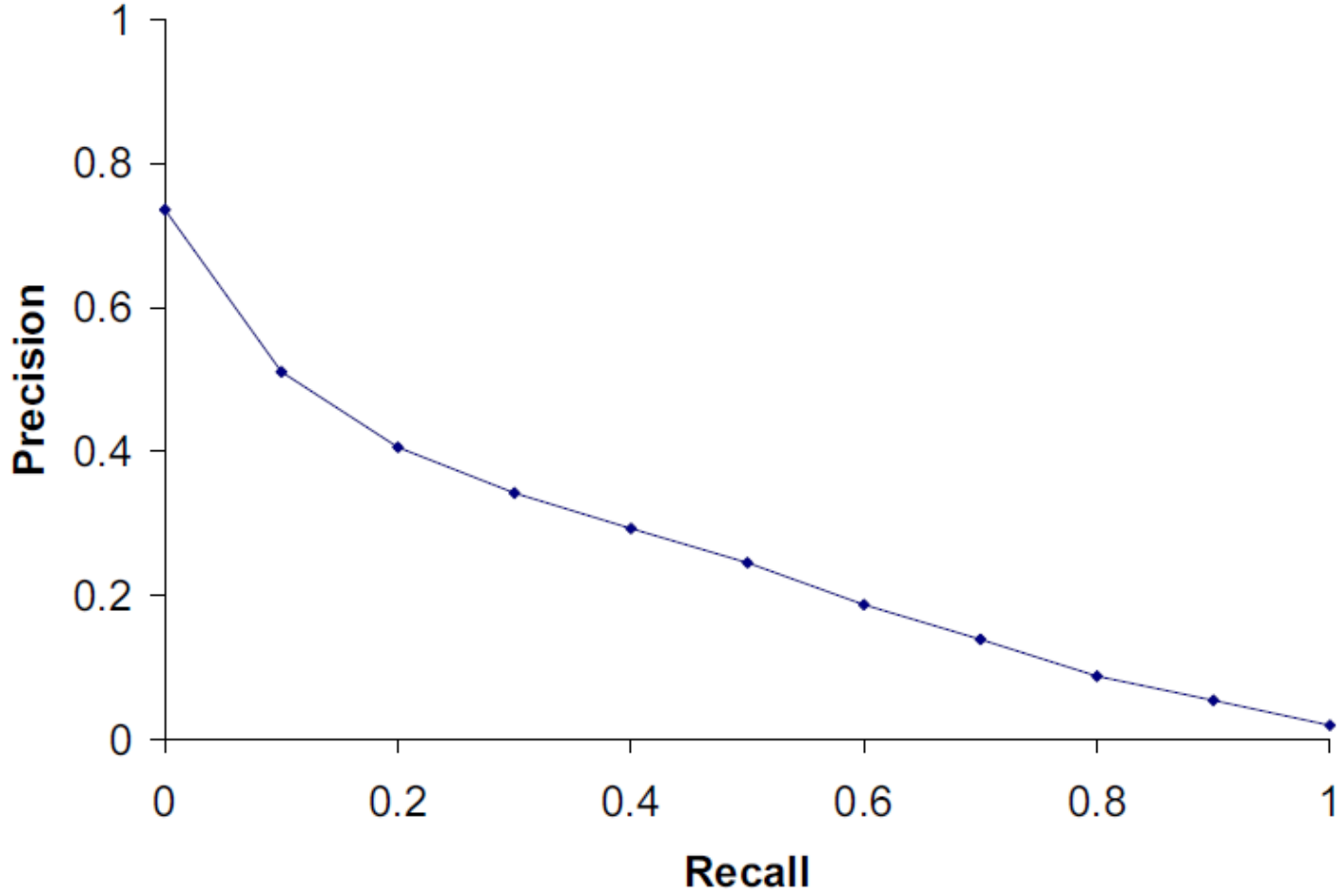
Interpolated Recall

The true precision plot (blue) necessarily dips at high precision each time a fp appears in the item ordering. These can be removed by using “interpolated precision” which is defined as the max precision at any recall value r’ > the current r. An interpolated precision-recall curve is non-increasing.



TREC Precision-Recall plots

We compute the interpolated precision values at ten values of recall, 0.1, 0.2,… 1.0.

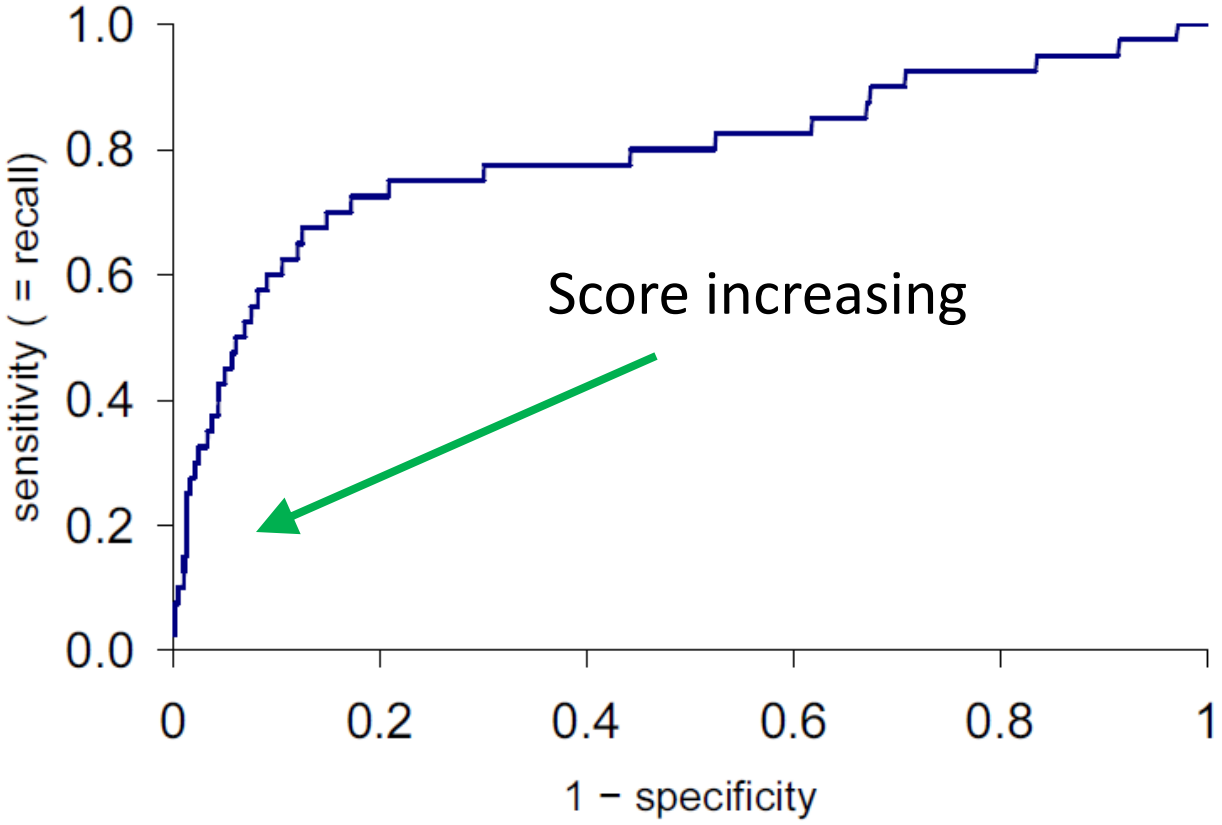


ROC plots

ROC is Receiver-Operating Characteristic. ROC plots

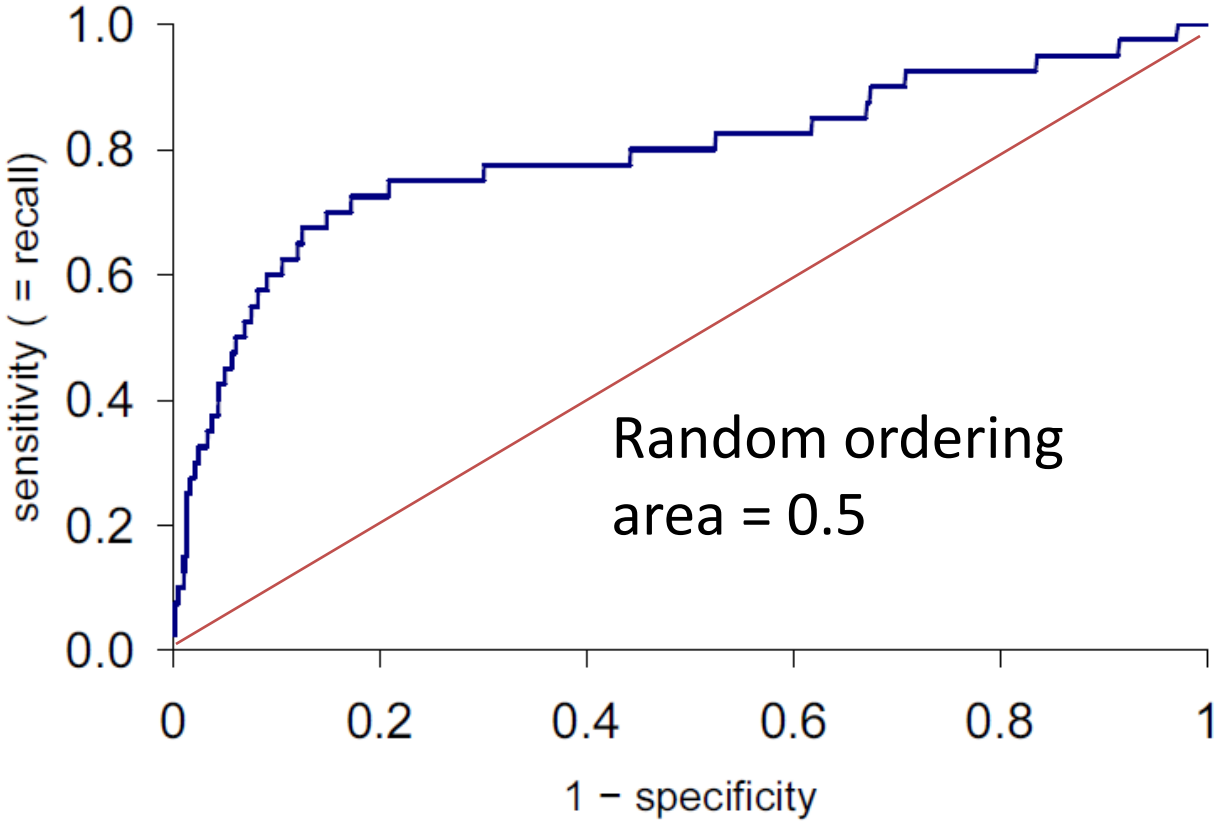
Y-axis: true positive rate = tp/(tp + fn), same as recall

X-axis: false positive rate = fp/(fp + tn) = 1 - specificity



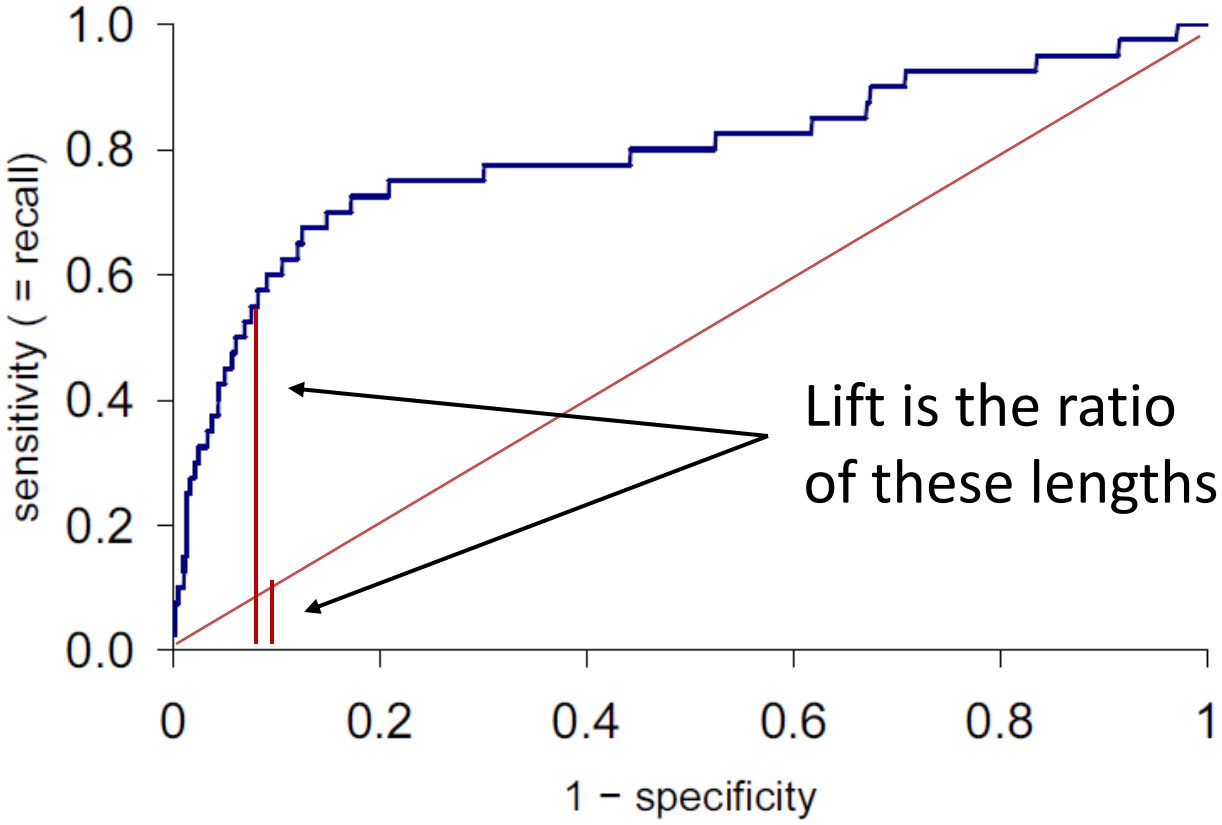
ROC AUC

ROC AUC is the “Area Under the Curve” – a single number that captures the overall quality of the classifier. It should be between 0.5 (random classifier) and 1.0 (perfect).



Lift Plot

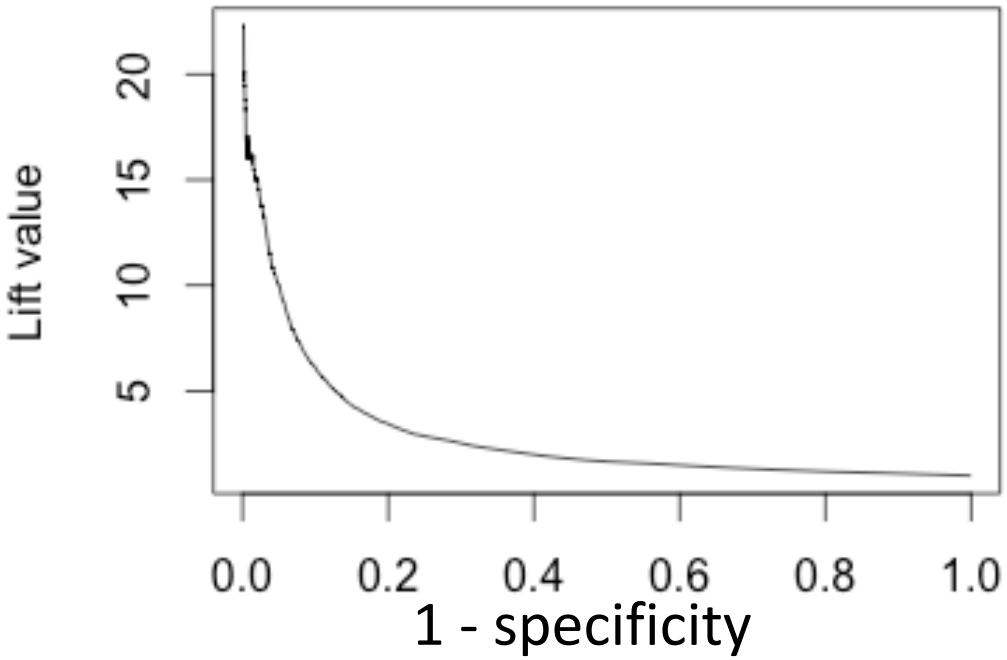
A derivative of the ROC plot is the lift plot, which compares the performance of the actual classifier/search engine against random ordering, or sometimes against another classifier.



Lift Plot

Lift plots emphasize initial precision (typically what you care about), and performance in a problem-independent way.

Note: The lift plot points should be computed at regular spacing, e.g. 1/00 or 1/1000. Otherwise the initial lift value can be excessively high, and unstable.



Summary

- Three (+ ½) more Basic Algorithms
  - Naïve Bayes
  - Logistic Regression (+ ½ SVM)
  - Random Forests
- Evaluation
  - Precision/Recall
  - Accuracy + weighted loss
  - ROC and AUC
  - Lift