Massachvsetts Institvte of Technology
Department of Electrical Engineering and Computer Science
6.034 Recitation 11, Thursday, December 1

Probability & Naïve Bayes — Prof. Bob Berwick, 32D-728

Agenda:
1. Finish Boosting
2. Probability: Axioms, Conditional Probability, Chain rule, Conditional independence, Bayes' Theorem
3. Naïve Bayes: another classifier (used for, e.g., Spam Asssasin)
4. Beyond naïve Bayes: the maximum entropy stewpot

1. Boosting and the Adaboost algorithm

The idea behind **boosting** is to find a weighted combination of *s* "weak" classifiers (classifiers that underfit the data and still make mistakes, though as we will see they make mistakes on less than ½ the data), $h_1, h_2 ..., h_s$, into a **single strong** classifer, $H(x)$. This will be in the form:

$$H(\vec{x}) = sign(\alpha_1 h_1(\vec{x}) + \alpha_2 h_2(\vec{x}) + \cdots + \alpha_s h_s(\vec{x})$$

$$H(\vec{x}) = sign\left(\sum_{i=1}^{s} a_i h_i(\vec{x})\right)$$

$$\text{where: } H(\vec{x}) \in \{-1,+1\}, h_i(\vec{x}) \in \{-1,+1\}$$

Recall that the *sign* function simply returns +1 if weighted sum is positive, and –1 if the weighted sum is negative (i.e., it classifies the data point as + or –).
Each training data point is **weighted.** These weights are denoted w_i for i=1, ..., n. **Weights** are *like* probabilities, from the interval (0, 1], with their **sum equal to 1. BUT** weights are **never 0**. This implies that **all data points have some vote** on what the classification shuld be, at all times. (You might contrast that with SVMs.)

The general idea will be to pick a single 'best' classifier *h* (one that has the lowest error rate when acting all alone), as an initial 'stump' to use. Then, we will **boost** the weights of the data points that this classifier **mis-classifies (makes mistakes on),** so as to focus on the next classifier *h* that does best on the re-weighted data points. This will have the effect of trying to fix up the errors that the first classifier made. Then, using this next classifier, we repeat to see if we can now do better than in the first round, and so on. In computational practice, we use the same sort of entropy-lowering function we used with ID/classifier trees: the one to pick is the one that lowers entropy the most. But usually we will give you a set of classifiers that is easier to 'see', or will specify the order.

In Boosting we always pick these initial 'stump' classifiers so that the error rate is strictly < ½. Note that if a stump gives an error rate greater than ½, this can always be 'flipped' by reversing the + and – classification outputs. (If the stump said –, we make it +, and vice-versa.) Classifiers with error exactly equal to ½ are useless because they are no better than flipping a fair coin.

1. Here are the definitions we will use.
Errors:
The error rate of a classifier *s*, E^s, is simply the sum of all the *weights* of the training points classifier h_s gets **wrong**.
($1–E^s$) is 1 minus this sum, the sum of all the *weights* of the training points classifier h_s gets **correct.**
By assumption, we have that:
$E^s < ½$ and $(1– E^s) > ½$, so $E^s < (1– E^s)$, which implies that $(1– E^s)/E^s > 1$

Weights:
α_s is **defined** to be ½ ln[(1− E^s)/ E^s)], so from the definition of weights, the quantity inside the ln term is > 1, so all alphas must be positive numbers.
Let's write out the Adaboost algorithm and then run through a few iterations of an example problem.

Adaboost algorithm
Input: training data, $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$

1. **Initialize data point weights.**

   Set $w_i^1 = \dfrac{1}{n} \quad \forall i \in (1, \ldots, n)$

2. **Iterate over all 'stumps':** for s=1, ..., T

   a. **Train base** learner using distribution w^s on training data.
   Get a base (stump) classifier $h_s(x)$ that achieves the lowest error rate E^s .
   (In examples, these are picked from pre-defined stumps.)

   b. **Compute the stump weight**: $\alpha_s = \dfrac{1}{2} \ln \dfrac{(1 - E^s)}{E^s}$

   c. **Update weights** (3 ways to do this; we pick Winston's method)

   For points that the classifier **gets correct,** $w_i^{s+1} = \left[\dfrac{1}{2} \cdot \dfrac{1}{1 - E^s} \right] \cdot w_i^s$

   (Note from above that 1− E^s> ½, so the fraction 1/(1− E^s) must be < 2, so the total factor scaling the old weight must be < 1, i.e., the **weight of correctly classified points must go DOWN in the next round)**

   For points that the classifier **gets incorrect,** $w_i^{s+1} = \left[\dfrac{1}{2} \cdot \dfrac{1}{E^s} \right] \cdot w_i^s$

   (Note from above that E^s < ½, so the fraction 1/E^s) must be > 2, so the total factor scaling the old weight must be > 1, i.e., the **weight of incorrectly classified points must go UP in the next round**)

3. **Termination condition:**

   If $s > T$ or if $H(x)$ has error 0 on training data or < some error threshold, exit;
   If there are no more stumps h where the weighted error is < ½, exit (i.e., all stumps now have error exactly equal to ½)

4. **Output final classifier:**

$H(\vec{x}) = sign\left(\sum_{i=1}^{s} a_i h_i(\vec{x})\right)$ **[this is just the weighted sum of the original stump classifiers]**

Note that test stump classifiers that are **never** used are ones that make more errors than some pre-existing test stump. In other words, if the set of mistakes stump X makes is a **superset** of errors stump Y makes, then Error(X) > Error(Y) is **always** true, no matter weight distributions we use. Therefore, we will **always** pick Y over X because it makes fewer errors. So X will **never** be used!

Let's try a boosting problem from an exam (on the other handout).

Food for thought questions.

1. How does the weight α^s given to classifier h_s relate to the performance of h_s as a function of the error E^s?
2. How does the error of the classifier E^s affect the new weights on the samples? (How does it raise or lower them?)
3. How does AdaBoost end up treating outliers?
4. Why is not the case that new classifiers "clash" with the old classifiers on the training data?

5. Draw a picture of the training error, theoretical bound on the true error, and the typical test error curve.
6. Do we expect the error of new weak classifiers to increase or decrease with the number of rounds of estimation and re-weighting? Why or why not?

Answers to these questions:

1. How does the weight α^s given to classifier h_s relate to the performance of h_s as a function of the error E^s?

Answer: The lower the error the better the classifier h is on the (weighted) training data, and the larger the weight α^t we give to the classifier output when classifying new examples.

2. How does the error of the classifier E^s affect the new weights on the samples? (How does it raise or lower them?)

Answer: The lower the error, the better the classifier h classifies the (weighted) training examples, hence the larger the increase on the weight of the samples that it classifies incorrectly and similarly the larger the decrease on those that it classifies correctly. More generally, the smaller the error, the more significant the change in the weights on the samples.
Note that this dependence can be seen indirectly in the AdaBoost algorithm from the weight of the corresponding classifier α_t. The lower the error E^t, the larger α_t, the better h_t is on the (weighted) training data.

3. How does AdaBoost end up treating outliers?

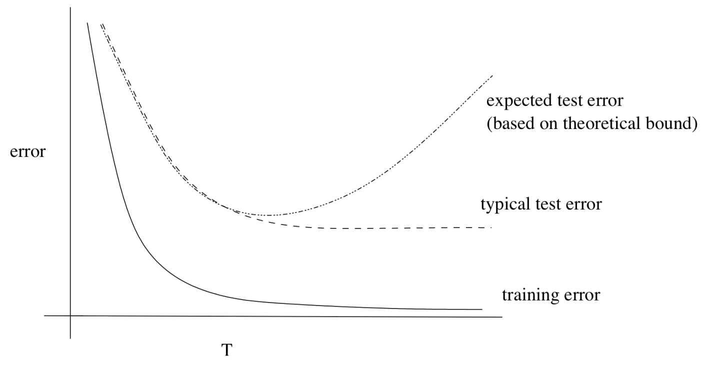
Answer: AdaBoost can help us identify outliers since those examples are the hardest to classify and therefore their weight is likely to keep increasing as we add more weak classifiers. At the same time, the theoretical bound on the training error implies that as we increase the number of base/weak classifiers, the final classifier produced by AdaBoost will classify all the training examples. This means that the outliers will eventually be "correctly" classified from the standpoint of the training data. Yet, as expected, this might lead to overfitting.

4. Why is not the case that new classifiers "clash" with the old classifiers on the training data?

Answer: The intuition is that, by varying the weight on the examples, the new weak classifiers are trained to perform well on different sets of examples than those for which the older weak classifiers were trained on. A similar intuition is that at the time of classifying new examples, those classifiers that are not trained to perform well in such examples will cancel each other out and only those that are well trained for such examples will prevail, so to speak, thus leading to a weighted majority for the correct label.

5. Draw a picture of the training error, theoretical bound on the true error, and the typical test error curve.

Answer:



6. Do we expect the error of new weak classifiers to increase or decrease with the number of rounds of estimation and re-weighting? Why?

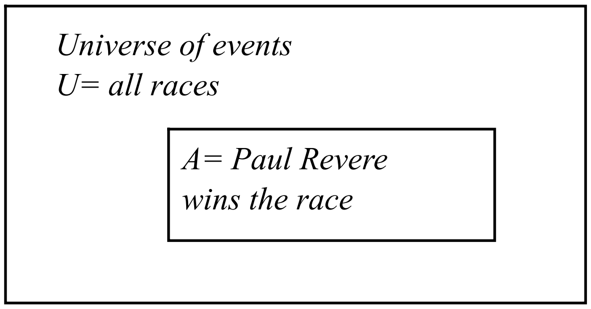
Answer: We expect the error of the weak classifiers to increase in general since they have to perform well in those examples for which the weak classifiers found earlier did not perform well. In general, those examples will have a lot of weight yet they will also be the hardest to classify correctly.

2. Basics of probability (review & pictures)

The fundamentals of probability theory: the axioms of probability. Why are these important? The power of the purse: Because while there are *other* attempts to handle the notion of 'uncertainty', e.g., 'fuzzy logic', '3-valued logic', etc., these axioms are the **only** system with the property that **if** you **gamble** with them, you **cannot** be unfairly exploited by an opponent who uses some other system (Di Finetti, 1932 theorem).

So, some first concepts.
We say that ***A* is a random variable** if *A* denotes an event and there is some uncertainty if *A* is true.
Typically, we let *U* denote the **universe** of all possible events (= all "possible worlds"). Then a subset of *U*, call it *A*, corresponds to the set of events in which *A* is true.

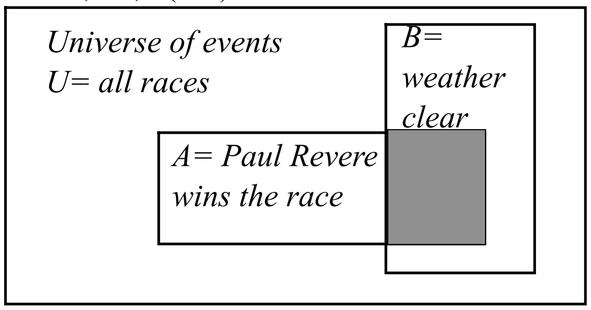
Example. Let the universe *U* be the set of all horse races. Let *Paul Revere* (abbreviation: P-R) be a horse. Then we can let *A* denote the set of racing events in which Paul Revere wins. We can draw this as a picture, where *races* labels the outer square, the universe, and the circle inside is the set of all events where Paul Revere wins the race:



Let us denote by ***P*(*A*)** the fraction of events (possible worlds in the universe of events) in which *A* turns out true. We could spend the next 2 hours on the philosophy of possible worlds and this business. But we won't.
We will compute probabilities using an informal notion of *areas* (formally, we'd use measure theory).
The Universe of all events has total area 1, $P(U)$=1, because it denotes all the events that are true. $P(A)$ then is the area of the smaller rectangle with respect to *U* (= the fraction of the total universe in which Paul Revere wins). $P(\neg A)$= the races in which Paul Revere does **not** win = the set difference between *U* and *A*. From this we will posit 3 axioms regarding $P(A)$:

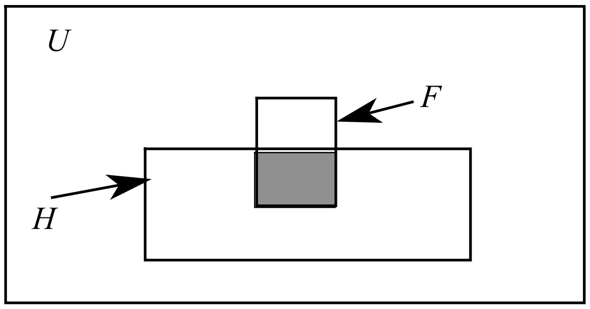
1. (1) $0 \leq P(A) \leq 1$ [because: the area of *A* cannot be < 0 or > 1]
2. (2) $P(true)$=1
3. (3) $P(false)$=0
4. (4) $P(A \vee B) = P(A) + P(B) - P(A,B)$ [where $\vee$ means "or", i.e., **either** *A* or *B* must be true; + means "add together", and the comma in *A, B* means "and", i.e., both *A* **and** *B* must be true]

To see how this last axiom works, let's look at the racing universe with event *A*= Paul Revere wins and a second event, *B*= the weather is clear. The **shaded area** represents the fraction of events when **both** *A* and *B* are true, i.e., $P(A,B)$= true:



It should be apparent that in order to figure out the probability of *A* **or** *B*, we need to add up the areas corresponding to *A* and to *B*, but then subtract out the shaded area so that it is not counted twice. In this way, we arrive at the formula for the probability of *A* **or** *B*.

We next turn to the notion of **conditional probability.**
We let *P*(*A*|*B*) denote the fraction of events/possible worlds in which *B* is true, and then *also* have *A* true. That is, we 'shrink' the universe from *U* down to *B*, focusing in on a subset possibly more relevant to our situation, and use *that* as our basis to calculate probabilities.
Example. In the figure below, we illustrate the following situation. Let *H*= probability that "I have a headache"; *F*= probability that "I am getting the flu". These are denoted by the rectangles *H* and *F* in the figure below. Let us assume:
P(*H*) = 1/10; *P*(*F*)=1/40. Now let's compute the conditional probability *P*(*H*|*F*), i.e., the probability that I have a headache **given** that I have the flu. This is the fraction of flu-events that are also headache events – that is, if we just look at the rectangle *F*, what proportion of *F* overlaps with *H*? (The answer is 1/2). Thus, *P*(*H*|*F*)=1/2.



In other words, to find *P*(*H*|*F*), we compute:
(# worlds in which *H* **and** *F* are true)/(# worlds in which *F* is true) or,
(area *H* **and** *F*)/(area of *F*), or
P(*H, F*)/*P*(*F*)

So this is the **formula for conditional probability:**

$$P(A \mid B) = \frac{P(A,B)}{P(B)}.$$

Note how *P*(*B*) is in the denominator here. Multiplying out, we obtain the important formula called the **chain rule** which we will uses in the naïve Bayes classifier:

$$P(A,B) = P(A \mid B) \cdot P(B)$$

Some other manipulations of conditional probability will be used in what follows. We consider two: (i) simplifications to the *right* of the conditioning bar symbol |; and (ii) simplifications to the *left* of the conditioning bar symbol.
Simplifications to the *right* of the bar:
Suppose we have *lots* of conditions to impose on whether or not Paul Revere wins. For example, this could depend on not only if the weather's clear, but also whether the jockey's brother is a friend of mine, whether Paul Revere won its last race, etc. In other words:

P(Paul Revere wins | weather clear, jockey's brother a friend, P-R won last race)

With more factors then, we have less *bias*, because we are focusing in on our particular situation, but we will have more *variance*, because it will become harder and harder to measure all these terms perfectly. So, sometimes we will want to reduce the number of factors to the right of the conditioning symbol to those we are more confident we can estimate; this is called *back off*. (We will see this in action soon). There is no problem in simply doing this:

P(Paul Revere wins | weather clear, ~~jockey's brother a friend~~, ~~P-R won last race~~)

And then of course just having *P*(Paul Revere wins | weather clear) remaining. But what about if there are more terms to the *left* of the bar, as in this case:

P(Paul Revere wins, Valentine loses, Epitaph loses | weather clear)

Note that if we *add* terms to the left the probability should get lower and lower every time we add a new factor. (Why? Think about intersection.) If we just care about Paul Revere, are we then allowed to simply strike out the other two horses, this way?

P(Paul Revere wins, ~~Valentine loses~~, ~~Epitaph loses~~ | weather clear)

The answer is: No! We need to carry out a more complex expansion to isolate Paul Revere on the left. To see how, let's abbreviate Paul Revere wins as *R*, Valentine loses as *V*, Epitaph loses as *E*, and the Weather is clear as *W*. Then our conditional probability:

P(Paul Revere wins, Valentine loses, Epitaph loses | weather clear)

Can be abbreviated as:

$$\frac{P(R,V,E,W)}{P(W)}$$

We can use this formula to derive the **chain rule for conditional probability:**

P(Paul Revere wins, Valentine loses, Epitaph loses | weather clear)=
P(Paul Revere wins| Valentine loses, Epitaph loses, weather clear) ×
P(Valentine loses | Epitaph loses, weather clear) ×
P(Epitaph loses | weather clear)

Proof. Writing out the 3 terms:

$$\frac{P(R,V,E,W)}{P(W)} = \frac{P(R,V,E,W)}{P(V,E,W)} \times \frac{P(V,E,W)}{P(E,W)} \times \frac{P(E,W)}{P(W)}$$

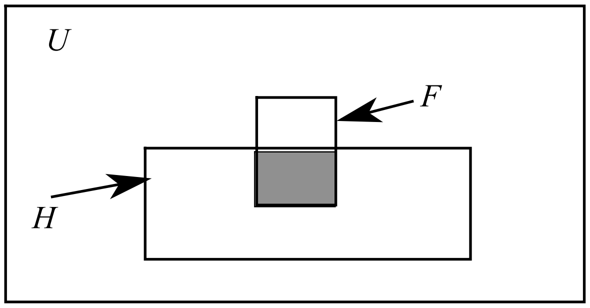
Now, supposed it is the case that the following simpler expansion holds:
P(Paul Revere wins, Valentine loses, Epitaph loses | weather clear) =
P(Paul Revere wins| ~~Valentine loses, Epitaph loses~~, weather clear) ×
P(Valentine loses | ~~Epitaph loses~~, weather clear) ×
P(Epitaph loses | weather clear)

In this case, whether Paul Revere wins or not depends *only* on whether the weather's clear…and not on what the other two horses do. They are irrelevant factors, so we can strike them out. In this case, when the probability is *unchanged* when we drop out conditioning factors, we say that the probability is **conditionally independent** (independent of the other horses, but still conditioned on the weather). More generally, if there are *n* factors *f*, and each factor is independent of the other, but still dependent on a condition *c*, we can write the following, which will be another key ingredient in our naïve Bayes classifier model:

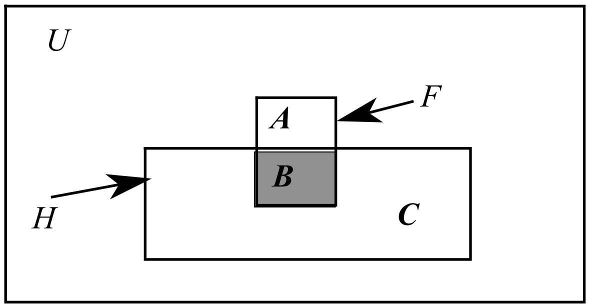
$$P(f_1,\ldots,f_n \mid c) = P(f_1 \mid c) \times \ldots \times P(f_n \mid c)$$

That is, we can just write out the probability as the product of the *n* factors, **assuming they are independent from one another** (the outcomes of these events do not affect the outcomes of one another); note the factors are still dependent on the outcome of event *c*.

OK, we come to the last ingredient we shall need, **Bayes' Law**. Again we can illustrate this with the simple picture of headache and flu as before. Recall *P*(*H*)=1/10; *P*(*F*)=1/40, *P*(*H*|*F*)=1/2.



Now we will **label** each of the distinct regions in this diagram, *A, B,* and *C*, as follows. *A*+*B*=area of *F*; *B*+*C*= area of *H*:



By the definition of conditional probability, $P(H|F)= P(H,F)/ P(F) = B/(A+B)$.
Now consider this reasoning: one day you wake up with a headache, and you think, OMG, 50% of flus are associated with headaches, so now I have a 50-50 chance of getting the flu." Is this reasoning correct?

What we *want* to compute is: $P(F|H)$. We already know the *other* conditional probability, that of headache given the flu. Further, by the definition of conditional probability, in terms of the regions *A, B,* and *C*, we have that: $P(F|H) = B/(B+C)$. To find this last ratio of regions, we can take the conditional probability $P(F|H) = B/(A+B)$, and multiply it by $(A+B)/(B+C)$, as follows:

$$\frac{B}{B+C}=\frac{B}{A+B}\cdot\frac{A+B}{B+C} \text{ i.e.,}$$

$$P(F \mid H)=P(H \mid F)\bullet\frac{P(F)}{P(H)}$$

$$\text{in our example, } \frac{1/2\times 1/40}{1/10}=\frac{1/80}{1/10}=\frac{1}{8}$$

The term $P(F)$ is called the **prior probability** (of getting the flu); the term $P(H|F)$ is called the **likelihood**; the term $P(H)$ is the **evidence** (e.g., that you have a headache); and the term $P(F|H)$ is called the **posterior probability** of getting the flu (given that you have a headache). So this updated probability is a kind of learning: given the fact (data) that you indeed have a headache, how does the probability of getting the flu change? (It increases from 1/40 to 1/8.) Inverting from $P(H|F)$ to $P(F|H)$ is called **Bayes' Law.** It follows from a very simple manipulation of the definition of conditional probability and then application of the chain rule, i.e., that $P(A,B)= P(A|B)\times P(B)$:

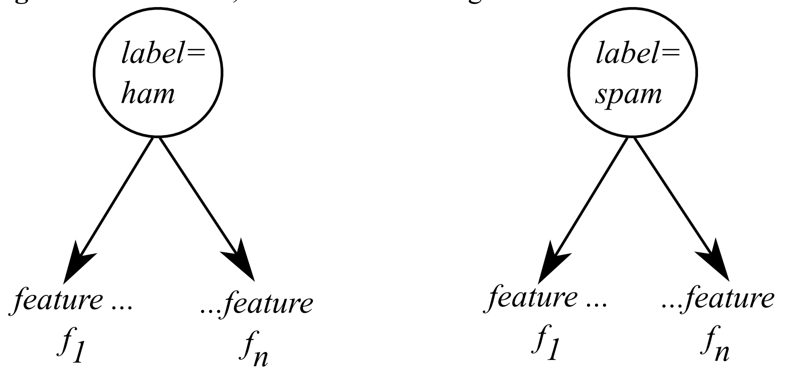
$$P(B \mid A)=\frac{P(A,B)}{P(B)} \text{ (by dfn of conditional probability)}$$

$$=\frac{P(A \mid B)\cdot P(B)}{P(B)} \text{ (by chain rule, replacing } P(A,B))$$

Or in words we can say this:

$$\text{posterior}=\frac{\text{likelihood} \times \text{prior}}{\text{evidence}}$$

Now let's put this all to work to build a classifier called **Naïve Bayes**. Like k-means and ID-trees, and Boosting, etc., this will take as input the values of some **features** and then output a classification **label**.
As our example, we will use the common, but valuable task of classifying email into 1 of 2 categories: either good email ("ham") or bad email ("spam"). The underlying probability model follows what is called a **Bayes' net**. We can imagine the following generative process: we pick a label, e.g., "ham", and given this label, email documents of this type will have a certain distribution of feature values $f_1, \ldots, f_n$. If we pick the other label, "spam", we will get another distribution for the feature values (hopefully distinct). So the picture looks like this, and the idea of course is that **given** a **new** email, we would like to figure out whether it is ham or spam:



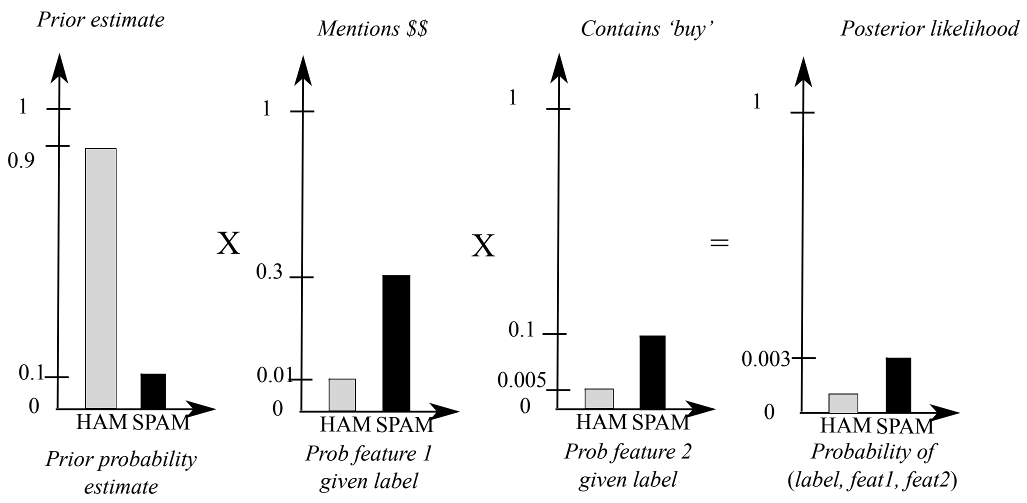
Crucially, we assume that **the features are independent from one another.** (This is the "naïve" part of Naïve Bayes.) Their values depend on (are conditioned on) **only** the value of the label. That is why we draw the networks as above, with **no links** between the features, only from the label directed down to the features.

Now here's the idea behind the classification.. Suppose we have estimated that 90% of our email is "ham" (OK), and that 10% is "spam". This gives us our **prior probability estimates** $P(label=ham)=0.9$ and $P(label=spam)=0.1$. That's what we can say about any new email **without any additional information.** (We'll see below how we get these estimates.)
Now, when we get a new email, we will get the values of its **features** and use these to adjust the prior probabilities, as with our headache example. (In our example, to keep things simple, we will use only two features.)

So, this new email comes along: "*Buy this amazing new Ginsu knife for only $39.....*"
Is this ham or spam? We'll assume that we use the following 2 features:

Feature 1: The email mentions money; this occurs in 30% of spam, and in 1% of ham
Feature 2: The email contains the word 'buy'; this occurs in 10% of spam, and in 0.5% of ham

We can picture our calculation as follows: our initial prior probabilities for each category are adjusted by **multiplying** the contribution **each feature** 'votes' (independently) as to how likely each category is. Then we pick the **most likely = biggest probability category** at the end:



So, in this case, our new email is classified as "spam" because this yields the largest posterior likelihood. Note how we got this value. It is simply this:

$P(label) \times P(f_1 \mid label) \times P(f_2 \mid label) = P(label, f_1, f_2)$ [recall from dfn of conditional prob that: $\frac{P(label, f_1, f_2)}{P(label)} = P(f_1 \mid label) \times P(f_2 \mid label)$ IF f_1, f_2 are independent of one another]

In other words, we multiple the following out to find the label likelihood, and pick the biggest likelihood:

Prior probability of a label × Probability of feature contributions = Posterior label likelihood

In our case, for the two labels "ham" and "spam":

	Prior ×	Pr(feat1 ($)| l) ×	Pr(feat 2 ('buy')|l)	= Label likelihood	
Ham:	0.9 ×	0.01 ×	0.005	= 0.000045	(log of this likelihood: –4.34)
Spam:	0.1 ×	0.30 ×	0.10	= 0.00303	(log of this likelihood: –2.52)

So, our email is more likely to be spam than ham. In fact, taking the ratios of the log likelihoods, –2.52/–4.32, the email is about 2 orders of magnitude (100x) more likely to be spam than ham. Recall that: (1) the features **must** be independent of one another; (2) we can add other features, of course…this is what a program like Spam Assassin can do, by training; and (3) one can use this method with lots more categories to **classify** documents (see the end of the handout).

Let's turn to justifying this approach probabilistically, as well as how we actually estimate the probability values above, via training, and highlighting some pitfalls.

First, why is this justified? We are computing the **maximum** probability that an input email will have a particular label (category), **given** that it has a particular set of features. We pick the label that maximizes: *P*(*l=value* | *observed features*). Let's follow out this logic. We are maximizing the following quantity over label values:

$$\max P(label \mid features) = \max \frac{P(features, label)}{P(features)}$$ [by dfn of conditional probability]

But note that the denominator in the expression above, *P*(*features*) = *P*(f_1, …, f_n) is *constant* no matter what our choice of label value. So, to maximize the above quantity, it suffices to maximize the numerator:

$$\max P(features, label) = P(f_1, \ldots, f_n, label)$$

By the chain rule, this quantity in turn is just:

$$\max P(label) \times P(f_{1,}, \ldots, f_n \mid label)$$

But given that the features are all independent of one another, this is the same as (recall our Paul Revere example!):

$$\max P(label) \times P(f_1 \mid label) \times \ldots \times P(f_n \mid label)$$
$$\max prior \quad \times 'vote' f_1 \quad \times \ldots \times 'vote' f_n$$

This is exactly the computation we have carried out. It remains to figure out how we 'train' our classifier – that is, how do we get the various estimates of the probabilities above? The simplest thing is just to estimate them from counts in training text, that is, known examples of ham and spam emails. These are the so-called *maximum likelihood estimates*:

$$P(label = ham) = \frac{count\ (\#\ ham\ emails)}{count(total\ \#\ emails)} \qquad P(label = spam) = \frac{count\ (\#\ spam\ emails)}{count(total\ \#\ emails)}$$

$$P(f_1 \mid label = ham) = \frac{count\ (\#\ ham\ emails\ mention\ \$)}{count(total\ \#\ ham\ emails)}$$

$$P(f_1 \mid label = spam) = \frac{count\ (\#\ spam\ emails\ mention\ \$)}{count(total\ \#\ spam\ emails)}$$

$$P(f_2 \mid label = ham) = \frac{count\ (\#\ ham\ emails\ contain\ \text{'buy'})}{count(total\ \#\ ham\ emails)}$$

$$P(f_2 \mid label = spam) = \frac{count\ (\#\ spam\ emails\ contain\ \text{'buy'})}{count(total\ \#\ spam\ emails)}$$

So this is how we get the estimates. For example, if we have 1000 emails, 900/1000 are ham, and 100/1000 are spam. Of the 100 spam emails, 30/100 mention money, and 1/100 contain 'buy'. For ham emails, 1/100 mention money and 5/1000 contain 'buy'.
Note that as the # of data samples (amount of training data) increases, then our estimates should get better; one of the properties of the maximum likelihood estimates is that they will converge to the 'true' values as the amount of data goes to infinity. (The mean approaches the true average.) But, if the # of training examples is small, our estimate will be very lousy, and have more noise (variance); there are a variety of things we can do to improve this, but that's for a machine learning course.

However, there is one particular case we should note. Suppose a particular count is actually 0 – that is, we *never* observe a particular feature associated with a particular label – this will happen especially if we keep adding more and more features. In this case, note that the entire probability product to find the likelihood will *all* be zero, just because one of the estimates is 0. So this is very bad!

There is a whole cottage industry devoted to fixing this problem, and it is called *smoothing*. It is basically the Robin Hood strategy: we rob probability mass from the rich and give it to the poor. In particular, the *simplest* smoothing strategy, invented by Laplace, is called *add*–1 *smoothing*: if a count is 0, we add 1 to it, so that, e.g., 0/100 goes to 1/100. (We must also *subtract* the appropriate probability mass, i.e., counts, from the *rest* of our estimates, so that the probabilities still add up to 1 in all.)

A second method of smoothing (probability mass redistribution) is due to Alan Turing. He figured this out when he was developing probability formulas for estimating the likelihood of finding German submarines in particular areas of the ocean. What if a submarine had *never* been observed in a particular spot? (Something that's actually quite likely!) Turing reasoned that a fairly good probability estimate of 'things never seen' would be quite close to the estimate of 'things seen *exactly* once'. This method, now called Good-Turing smoothing (only published until decades after WWII), works well but is finicky. There are whole books devoted to this subject, for machine learning and especially in natural language processing, where we quickly get word sequences never seen before.

One more thing. You may note that in our calculation we multiply together a (possibly long) string of probabilities, one for each feature. With a 1000 features, this value will quickly get very,

very small. So, the usual method is to operate in log space, where multiplication is just addition, so we can maintain accuracy. (That's why we used log likelihoods above.)

Beyond Naïve Bayes (Optional)
OK, this method is fine so far as it goes, but it can be improved enormously. Here we will just sketch one method, known as **maximum entropy classification** that can gobble down any set of features, even if they are not independent. Yet remarkably, as first shown by Jaynes (1957), it is the most probabilistically sound method of **combining diverse features.** It rationalizes the general notion of just 'scoring' features and adding them up. We won't prove this here, but just indicate the general approach, which is now broadly used in, e.g., figuring out the part of speech labels in text. (For instance, in the sentence, *police police police*, is the first *police* a Noun or a Verb?)

1. To begin, let's assume there are now 10 labels for documents, with categories *A, B, C, D, E, F, G, H, I, J.* (So, e.g., category *A* could be travel; *B* sports; *C* business; etc.) If we know this, and **no other information** then given an email *m*, what is our best guess for category *C* (business) given this email, i.e., *P*(*C* | *m*)?
The maximum entropy approach would claim it is 1/10: that is, we maximize the quantity in each of the 10 bins, uniformly, by spreading out the total probability mass of 1 among 10 bins.

2. Now suppose I tell you that 55% of all emails are in category *A*, travel? Now what is the quantity *P*(*C*| *m*)? I think it should not be too hard to see that *A* gobbles up 0.55 of the probability mass, leaving 0.45 to be distributed evenly over the remaining 9 categories, or 0.05 for each of the remaining categories, including category *C*, business. So the maximum entropy estimate for *P*(*C*| *m*) is 0.05.

3. Now suppose I add *another* constraint: that *in addition* to the fact in (2), we know that 10% of all emails contain the word 'buy'. What is *P*(*C*| *m*) now? This gets harder to visualize, so we'll write it out as a table, where the first row is the probability of containing 'buy' (which thus must add up to 0.1 of all emails), and second row is the probability of not containing 'buy', which we have labeled *other* (which thus must add up to 0.9). Once again following the maximum entropy idea, since we don't know anything else about the 'contains buy' row, we should distribute its 0.1 total *evenly* among the 10 bins, thus giving 0.01 to each. Next, since *all* of category *A* must add up to 0.55, and since the 'contains buy' cell holds 0.01, it must be that the cell in the row labeled *other* and in column *A* must have the value 0.54 (so that the column total is 0.55). That leaves 0.9–0.54 = 0.36 for the rest of the 9 bins in the *other* row. Once again, spreading this evenly, we get 0.36/9 = 0.04 for each of these bins (so that each column here adds to 05). Thus we have the following table:

	1	2	3	4	5	6	7	8	9	10
	A	B	C	D	E	F	G	H	I	J
buy	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
other	0.54	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04

So, *why* is this called *maximum entropy*? You should realize that by spreading out the values evenly, we are *maximizing the entropy of the cell values*: $-p \log p$ summed over all entries is at a maximum. (Below we indicate *why* this is a good thing to do.) In any case, we are maximizing the entropy *subject to the constraints specified*. (We have two so far.)

4. So let's add one more constraint. Suppose that in addition, 80% of the 'buy' emails are in *either* category *A* or category *C*. Now we want to figure out *P*(*C*| *m*). Gulp! This one is much harder to figure out – in fact, in general to do this, it is like spreadsheets, but we can indicate what has to be true in our table now: the probability of the *buy* row, column *A*, plus the probability in the *buy* row, column *C*, must add up to 0.08 (80% of the 10%). That turns out to be the values 0.051 and 0.029. Since that leaves 0.020 for the rest of the bins in the *buy* row, these must be 0.020/8=0.0025. Since column *A* must still add up to 0.55, then that leaves 0.499 for row *other*, column *A*. Since

the *other* row must still sum to 0.9, we have 0.9–0.499= 0.401 to distribute evenly over the rest of the *other* bins, so this is 0.401/9 = 0.0446. If we impose these constraints, you'll see that this is the answer (we don't say how we figured it out!)

	1	2	3	4	5	6	7	8	9	10
	A	B	C	D	E	F	G	H	I	J
buy	0.051	.0025	0.029	.0025	.0025	.0025	.0025	.0025	.0025	.0025
other	0.499	.0446	.0446	.0446	.0446	.0446	.0446	.0446	.0446	.0446

Now we know that $P(buy,C)$= 0.029; $P(C|\, buy)$= 0.29 (= 0.029/0.1); $P(A \mid buy)$= 0.51.
This is our classifier, a *maximum entropy* classifier.

The punchline. While there are many possible distributions that could yield the three observed constraints, that 55% of the emails are in category *A*, that 10% of the emails contain *buy*, and that of these 10%, 80% are in category *A* or *C*, the **one distribution** that we picked, where we have **maximized** the entropy of the probability mass subject to these constraints, turns out to be the **only one** having the following two properties, the second one quite remarkable:

1. This distribution follows the form: $P(email,label)=\frac{1}{Z(\lambda)}\exp\sum_i \lambda_i f_i(email,label)$ where the lambdas are the weights associated with each feature f_i; the function f_i returns 1 if the feature is in the email, and 0 otherwise; and *Z* is a normalizing constant to make sure the probabilities all add up to 1.
2. This distribution **maximizes the probability of the training data,** $\prod_j P(email_j,label_j)$

This is what justifies the method!