Naïve Bayes classifiers

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

Common problem: detect spam tweets

- Q. Is the following tweet spam or ham (not spam)?

- Without knowledge of content, what's the best we can do?

SPAM at big package store

Common problem: detect spam tweets

- Q. Is the following tweet spam or ham (not spam)?

Viagra sale

- Without knowledge of content, what's the best we can do?
- Use prior knowledge about relative likelihoods of spam/ham
- If *a priori*, we know 75% of tweets are spam, always guess spam
- (Note: we could also be solving topic classification not spam)

Our base model

- If 75% of tweets are spam, always guessing spam gives us a lower bound of 75% accuracy
- *Accuracy* has formal definition: % correctly-identified tweets
- A superior model must do better than 75% accuracy
- What if a priori spam rate was 99%? Our model has 99% accuracy
- That hints that accuracy can be very misleading in isolation (more later)

Better model using knowledge of content

- If we can see tweet words, we have more to go on; e.g.,

  $\boxed{\text{Viagra sale}}$ $\boxed{\text{Buy catfood}}$

- Given "Viagra sale", how do you know it's spam?
- Because among spam emails you've received, those words are highly likely to appear
- Words like "Buy catfood" are unlikely to occur in spam email
- Vice versa: "Viagra sale" low likelihood in non-spam
- P(Viagra ∩ sale | spam) is high, P(Viagra ∩ sale | ham) is low

Model based upon tweet likelihoods

- Predict spam if:

  $P(\text{Viagra} \cap \text{sale} \mid \text{spam}) > P(\text{Viagra} \cap \text{sale} \mid \text{ham})$

- Predict spam if:

  $P(\text{Buy} \cap \text{catfood} \mid \text{spam}) > P(\text{Buy} \cap \text{catfood} \mid \text{ham})$

- This model works great but makes an assumption by not taking into consideration what knowledge? The *a priori* probabilities; assumes equal priors

Model combining priors and content info

- Predict spam if:

  $P(spam)P(Viagra \cap sale \mid spam) > P(ham)P(Viagra \cap sale \mid ham)$

- Predict spam if:

  $P(spam)P(Buy \cap catfood \mid spam) > P(ham)P(Buy \cap catfood \mid ham)$

- We are weighting the content likelihoods by prior overall spam rate
- If spam-to-ham priors are .5-to-.5 the prior terms cancel out

Are these computations probabilities?

- Are P(spam)P(Viagra ∩ sale | spam) and P(ham)P(Viagra ∩ sale | ham) probabilities after weighting?
- Nope. Sum over P(spam)P($wordsequence_i$ | spam) > 1 (Imagine P(spam)=1.0 and 1000 different word sequences)
- Must normalize by likelihood of seeing specific word sequence: P(spam)P(Viagra ∩ sale | spam) / P(Viagra ∩ sale) > P(ham)P(Viagra ∩ sale | ham) / P(Viagra ∩ sale)
- Congrats: You've just reinvented *Bayes Theorem*
- I.e., how to adjust knowledge of spam rate with content evidence

Maximum *a posteriori* classifier

- Choose class/category for document *d* with max likelihood:

$$c^* = \underset{c}{argmax}\ P(c|d)$$

- Substitute Bayes' theorem:

$$c^* = \underset{c}{argmax}\ \frac{P(c)P(d|c)}{P(d)}$$

Bayes' theorem

$$P(c|d) = \frac{P(c)P(d|c)}{P(d)}$$

Simplifying the classifier

- P(*d*) is constant on both sides so we can drop it for classification:

$$c^* = \underset{c}{argmax}\ P(c)P(d|c)$$

- If P(*c*) is same for all *c* OR we don‘t know P(*c*), we drop that too
- Another way to write the classification decision rule is *Bayes test for minimum error*:

$$P(c_1 \mid d) \gtrless P(c_2 \mid d)$$

Training the classifier

- We need to estimate P(c) and P(d | c) for all c and d
- Estimating P(c)? The number of documents in class c divided by the total number of documents
- Estimating P(d | c)? E.g., we need P(Viagra ∩ sale | spam)
- That means considering all 2-word combinations (*bigrams*), which grows in size with the square!

The naïve assumption

- Naïve assumption: conditional independence; Estimate P(Viagra ∩ sale | ham) as P(Viagra | spam) x P(sale | spam)

$$P(d|c) = \prod_{w \in d} P(w|c)$$

- So, our classifier becomes

$$c^* = \underset{c}{argmax}\ P(c) \prod_{w \in d} P(w|c)$$

- where *w* is each word in *d* not the unique vocabulary words

Fixed-length word-count vectors

- Rather than arbitrary-length word vectors for each document *d*, it's much easier to use fixed-length vectors with word counts; this:

$$c^* = \underset{c}{argmax}\ P(c) \prod_{w \in d} P(w|c)$$

becomes this

$$c^* = \underset{c}{argmax}\ P(c) \prod_{unique(w) \in d} P(w|c)^{n_w(d)}$$

Estimating P(*w* | *c*)

- We have P(*c*) so how do we estimate P(*w* | *c*)?
- It's number of times w appears in all documents from class c divided by the total number of words (including repeats) in all documents from class c

$$P(w|c) = \frac{count(w,c)}{count(c)}$$

Laplace smoothing

- If P(*w* | *c*)=0 then the entire product goes to zero. Ooops

$$c^* = \underset{c}{argmax}\ P(c) \prod_{w \in d} P(w|c)$$

- To avoid, add 1 to each word count in numerator and compensate by adding |V| to denominator (to keep a probability)

$$P(w|c) = \frac{count(w,c) + 1}{count(c) + |V|}$$

- We have added +1 to each word count and there are |V| words in each word-count vector

Dealing with misspelled or unknown words

- Laplace smoothing deals with *w* that are in the vocabulary but not in class *c*, which gives P(*w* | *c*) = 0
- What should count(*w,c*) be for a word not in vocabulary? Zero doesn't seem right as we have no data but if count(*w,c*)=0 for all classes, classifier is not biased
- count(*w,c*)=0 actually won't occur due to Laplace but it's like we map all unknown *w* to a wildcard word in vocabulary, which means |V| is one bigger
- Likely not a huge factor…

$$P(w|c) = \frac{count(w,c) + 1}{count(c) + |V| + 1}$$

Avoiding floating point underflow

- In practice, multipying lots of probabilities in [0,1] range tends to get too small to represent with finite floating-point numbers
- Take log (a monotic function) and product becomes summation

$$c^* = \underset{c}{argmax}\ P(c) \prod_{w \in d} P(w|c)$$

$$c^* = \underset{c}{argmax} \left\{ log(P(c)) + \sum_{unique(w) \in d} n_w(d) \times log(P(w|c)) \right\}$$

An example

Documents as word-count vectors

- One column per vocab word, one row per document

```
d1 = "sale viagra sale"
d2 = "free viagra free viagra free viagra free"
d3 = "buy catfood and buy eggs"
d4 = "buy eggs"
```

P(spam) = 2/4
P(ham) = 2/4

column for unknown words

spam or ham

	unknown	buy	catfood	eggs	free	sale	viagra	spam
0	0	0	0	0	0	2	1	1
1	0	0	0	0	4	0	3	1
2	0	2	1	1	0	0	0	0
3	0	1	0	1	0	0	0	0

Note:
|V|=7+1=8

Laplace smoothing

- Add +1 everywhere, even for unknown (just "`df+1`" in pandas)

spam

	unknown	buy	catfood	eggs	free	sale	viagra
0	1	1	1	1	1	3	2
1	1	1	1	1	5	1	4

ham

	unknown	buy	catfood	eggs	free	sale	viagra
2	1	3	2	2	1	1	1
3	1	2	1	2	1	1	1

Pandas query to get just spam rows: `df.query('spam==1')`

Estimating probabilities: P(*w* | spam)

- 1^{st}, get total word count in spam category: sum across rows or cols then sum that result; count(spam) = `spam.sum(axis=1).sum()`
- count(spam) includes |V| (smoothing) and +1 (unknown word) for denominator of P(*w*|spam)

spam

	unknown	buy	catfood	eggs	free	sale	viagra	
0	1	1	1	1	1	3	2	10
1	1	1	1	1	5	1	4	14

count(spam) = 24

$$P(w|c) = \frac{count(w,c)+1}{count(c)+|V|+1}$$

Estimating probabilities: P(*w* | spam)

- 2nd, get total count for each word in spam docs, count(w,spam)
  `spam.sum(axis=0)`
- 3rd, divide word count in each doc by count(spam)=24

spam

	unknown	buy	catfood	eggs	free	sale	viagra
0	1	1	1	1	1	3	2
1	1	1	1	1	5	1	4
	$\frac{2}{24}$	$\frac{2}{24}$	$\frac{2}{24}$	$\frac{2}{24}$	$\frac{6}{24}$	$\frac{4}{24}$	$\frac{6}{24}$

P(unknown|spam)=1/12
P(buy|spam)=1/12
…
P(viagra|spam)=1/4

Estimate $P(c|w)$ w/o $P(d)$ normalization

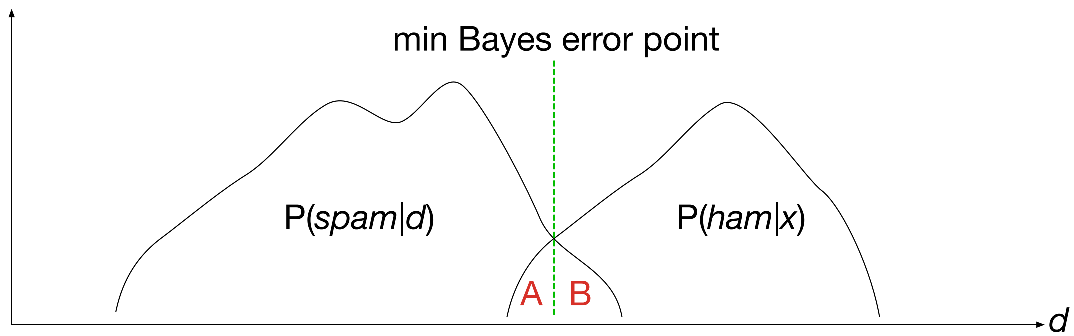
- P(spam|viagra sale) = P(spam)[P(viagra|spam)P(sale|spam)]
  = 1/2 x 6/24 x 4/24 = 0.0208
- P(ham|viagra sale) = P(ham)[P(viagra|ham)P(sale|ham)]
  = 1/2 x 2/20 x 2/20 = 0.005
- P(spam|viagra sale) > P(ham|viagra sale) so predict spam

Example with word count > 1

- $P(spam|sale\ viagra\ sale) = P(spam)[P(viagra|spam)P(sale|spam)^2]$
  $= 1/2 \times 6/24 \times 4/24 \times 4/24 = 0.0034$
- $P(ham|sale\ viagra\ sale) = P(ham)[P(viagra|ham)P(sale|ham)^2]$
  $= 1/2 \times 2/20 \times 2/20 \times 2/20 = 0.0005$
- Imagine "sale" or "viagra" repeated many times, all scores go down since $P(w|c) < 1$ but $P(ham|sale\ viagra\ sale)$ drops much faster than $P(spam|sale\ viagra\ sale)$
- Ratio of those spam to ham will grow with added spam words

Quick idea on Bayes error

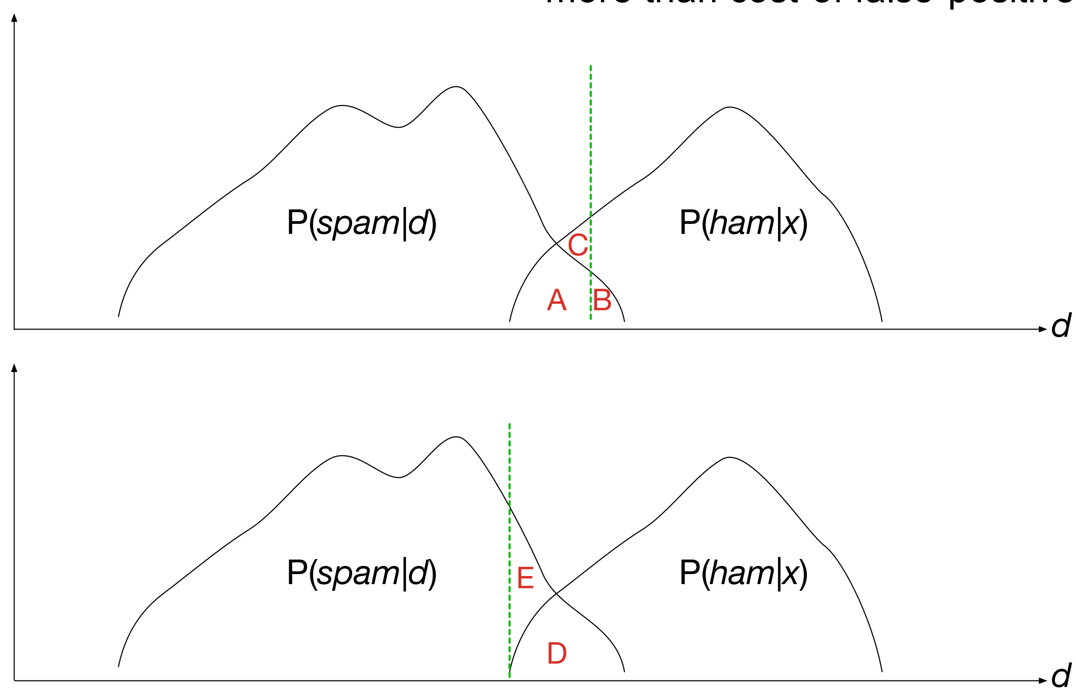
- Bayes error is the lowest possible total error from all classes
- Imagine squashing doc d down to single dimension
- Bayes error is area under P(*spam*|*d*) to right and P(*ham*|*d*) to left of decision point (A and B here)



Preview AUC ROC / PR curves

- If our "utility function" values spam or ham differently, can shift boundary to left/right
- Tradeoff: to get 0 ham error, must accept many spam, area under P(*spam*|*d*) to right: D+E error
- Later we'll examine ROC/PR curves that summarize effect of shifting boundary all over

E.g., in disease testing, might value catching disease more than cost of false positive



Key takeaways

- Naïve bayes is classifier often applied to text classification; e.g., spam/ham, topic labeling, etc…
- Less often used these days with rise of deep learning
- Fixed-length word-count vectors are the feature vectors per doc
- Bayes theorem gives formula for P(*c*|*w*)
- Naïve assumption is conditional independence $\rightarrow P(d|c) = \prod_{w \in d} P(w|c)$
- Training estimates P(*c*), P(*w*|*c*) for each *w* and *c*
- P(*c*) is ratio of docs in *c* to overall number of docs
- P(*w*|*c*) is ratio of word count of w in c to total word count in *c*
- Classifier: $c^* = \underset{c}{argmax}\ P(c) \prod_{unique(w) \in d} P(w|c)^{n_w(d)}$

Implementation takeaways

- Avoid vanishing floating-point values from product

$$c^* = \underset{c}{argmax}\left\{ log(P(c)) + \sum_{unique(w) \in d} n_w(d) \times log(P(w|c)) \right\}$$

- Avoid P(w|c)=0 via Laplace smoothing
  - add 1 to all word counts
  - adjust P(w|c) denominator with extra |V| since every doc now has every word
  - this is for missing words where w not in d but in V
- Treat unknown words as 1 / (count(c) + |V| + 1)