Chapter 4

Corpora Processing and Linear Classifiers (Naive Bayes)

4.1 History of Methodologies

4.1.1 Pre-Statistical (1650s/1950s through approx. 1980s)

- Mostly about modeling specific linguistic phenomena in a small number of sentences, sometimes using code
- Linguists/highly trained coders wrote down fine-grained detailed rules to capture various aspects, e.g. ‘ “swallow” is a verb of ingestion, taking an animate subject and a physical object that is edible...’
- Very time-consuming, expensive, limited coverage (brittle), but high precision
- Academically satisfying, but not good at producing systems beyond the demo phase

4.1.2 Statistical

- Empirical approach: learn by observing language as it’s used “in the wild”
- Many different names:
  - Corpus Linguistics
  - Empirical NLP
  - Statistical NLP
- Central tool:
  - corpus
  - thing to count with (i.e. statistics)
  - (later on) machine learning methodologies, software/hardware for helping with scale

- Advantages
  - Generalize patterns as they actually exist (i.e. bottom-up, not top-down)
  - Little need for knowledge (just count)
  - Systems are robust and adaptable (change domain by changing corpus)
  - Systems degrade more gracefully (corner cases captured in data)
  - Evaluations are (more) meaningful
- Limitations
  - Bound by data – can't model what you can't see – "I held the book with my arm stretched out and opened my hand. It (floated away), (fell to the ground)"
  - Big Data methods fail when the data is small or wrong – sometimes you want to try to translate Oromo news to English with 50,000 words of bible when you want 10m+words of news
  - more computationally expensive (but less human-expensive) (usually a good trade-off)
  - Methods don't have the same pattern-recognition and generalization abilities of humans learning (and putting into rule-based methods) which can lead to unintuitive brittleness.

4.1.3 Corpus (pl: corpora): a collection of (natural language) text systematically gathered and organized in some manner

- Features:
  - Size
  - Balanced/domain
  - Written/Spoken
  - Raw/Annotated
  - Free/Pay
- Some Famous (text) examples:
  - Brown Corpus: 1m words balanced English text, POS tags
  - Wall Street Journal: 1m words English news text, syntax trees
  - Canadian Hansards: 10m words French/English parliamentary text, aligned at sentence level
  - Wikipedia (all of it): 3b words
  - Google books ngrams: 500B words
  - Common Crawl: 1T words
  - Any others of particular interest?

4.1.4 How Big Does it need to be?

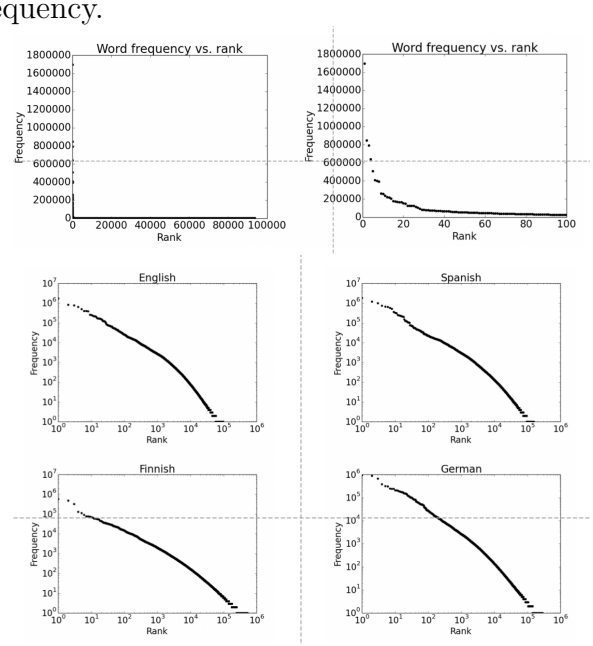
- We'd like to get examples of all linguistic phenomena, ideally several times so we know how likely they are to occur
- How big should a corpus be to get every possible sentence in English?
  - Every possible idea?
  - Every 5-word phrase?
  - Every word?
- None of these are possible!

4.1.5 corpus processing

```
# word counts and ngram counts

# 10 most frequent words in the text
sed 's/ /\n/g' sawyr11.txt | sort | uniq -c | sort -k1nr | head
# 10 most frequent words in the text after removing blank lines
sed 's/ /\n/g' sawyr11.txt | grep -v "^$" | sort | uniq -c | sort -k1nr | head
# 10 most frequent bigrams (2 word sequences) in the text

sed 's/ /\n/g' sawyr11.txt | grep -v "^$" > ts.words
tail -n+2 ts.words > ts.2pos
paste ts.words ts.2pos | sort | uniq -c | sort -k1nr | head
# count number of words/ngrams, number of word/ngram types, number of
# 1-count word/ngram types

# words in the text without blank lines, one word per line, saved to a file
# (for convenience)
sed 's/ /\n/g' sawyr11.txt | grep -v "^$" > ts.words
# number of word tokens
wc -l ts.words
# number of word types
sort ts.words | uniq | wc -l
# number of one-count words
sort ts.words | uniq -c | awk '$1==1{print}' | wc -l
# number of two-word sequence (bigram) tokens
# (based on the answer to the number of word tokens you should know this
# without running the command...
paste ts.words <(tail -n+2 ts.words) | wc -l
# number of bigram types
paste ts.words <(tail -n+2 ts.words) | sort | uniq | wc -l
# number of one-count bigrams
paste ts.words <(tail -n+2 ts.words) | sort | uniq -c | awk '$1==1{print}' | wc -l
```

4.1.6 Zipf's law

Take a naturally occurring corpus (in this case, of text). Count frequency of words. Order words by frequency, to form ranks. Then the rank of a word is inversely proportional to its frequency.



For frequency f and rank k, $f \approx \frac{1}{k^a}$ for some constant a. Thus $-a \approx \frac{\log(f)}{\log(k)}$.
Consequences:

- There will always be a lot of infrequent or unseen words
- This is true at all levels of linguistic structure
- So we have to find clever ways of generalizing so we can get reasonable estimates for things we haven't seen often enough

4.2 Nominal task (also HW1): text classifier

We have some book reviews. We want to know automatically if they're positive or negative.
Positive review:

```
I loved this book. The food was really good and fast (which is good
because I have a very packed schedule). Also great if you're on a
budget. The recipes have variations so I could eat different things but
not have to buy a whole new set of ingrediants.
```

Negative review:

```
I tried reading this book but found it so turgid and poorly written that
I put it down in frustration.  It reads like a translation from another
language by an academic bureacrat. The theme is interesting, the
execution poor.  Cannot recommend
```

4.2.1 Evaluation

We should always ask 'how do we know we are doing well at what we are trying to do?' This time the answer is fairly simple (it won't always be). We collect many reviews along with their labels (Positive/Negative, which we may have to create out of some other labels, like a numerical score).

We'll calculate simple *accuracy*: $\frac{\#\text{ correct}}{\#\text{ total}}$.

4.2.2 Data

We will divide our data into *train*, *development* (dev) (also sometimes called 'validation') and *test* corpora. train is used to build a model and is the largest data set (usually 80% of the data). dev is not used to train but is frequently consulted during optimization (where relevant) to avoid *overfitting* on training data. test is usually not evaluated or looked at except for when optimization is done, to ensure even less overfitting. Sometimes an even more super-secret *blind* test set is not even available to you but held off to test your final model.

4.2.3 Framework

Here's a simple framework for this problem:

```
import operator
def evaluate(sentence, option, model):
  # fill me in
  pass

def classify(sentence, options, model):
  scores = {}
  for option in options:
    scores[option] = evaluate(sentence, option, model)
  return max(scores.items(), key=operator.itemgetter(1))[0]
```

Let's consider methods for the 'evaluate' function:

4.3 Rule-Based (top-down?) Model

Positive reviews should have positive words, negative reviews should have negative words. Thankfully, people have compiled such *sentiment lexicons* for English. See `http://www.enchantedlearning.com/wordlist`.

Positive words example: {absolutely, adorable, bountiful, bounty, cheery}

Negative words example: {angry, abysmal, bemoan, callous}

Let's put the intuition and data together:

```
# externally constructed; don't actually structure your code this way
```

```
# probably shouldn't structure your code this way...
model = {}
model['good'] = set(['yay', 'love', ...])
model['bad'] = set(['terrible', 'boo', ...])

def evaluate(sentence, option, model):
  score = 0
  for word in sentence.split():
    if word in model[option]:
      score+=1
  return score
```

4.4 Empirical Model

Our intuitions about word sentiment aren't perfect and neither are those of the people who made the word list. But we do have many examples of reviews and their sentiments. So we can make our own list of good and bad words. Instead of hard-coding the model as I did above, we can create a *training* function that takes in sentences *with their labels* and then returns a model:

```
from collections import Counter, defaultdict
def train(labeled_sentences):
  scores = defaultdict(lambda: Counter()) # doubly nested structure
  for sentence, label in labeled_sentences:
    for word in sentence.split():
      scores[word][label]+=1
  model = defaultdict(lambda: set())
  for word, table in scores.items():
    # the most frequent label associated with the word
    label = max(table.items(), key=operator.itemgetter(1))[0]
    model[label].add(word)
  return model
```

We now have our first *supervised* model. We should back up and consider what we are actually trying to model from a probabilistic view. For one thing let's consider the actual probability of each label, rather than our ad-hoc adding method above.

Our classifier should choose $\text{argmax}_y P(y|s)$ for sentence s where y is one of a fixed set of labels. But we didn't consider s as some monotone thing; we considered the occurrence of each word as an event.

So we'll make an assumption called the *bag of words assumption* which is that for sentence $s = w_1 w_2 \dots w_n$, we say $P(y|s) = P(y|w_1, w_2, \dots, w_n)$. Note this now doesn't depend on the order of the words.[1]

[1]This assumption isn't really part of Naive Bayes, it's an assumption about the feature set being used. It's probably better to say that for x, y, we calculate $f(x, y) = f_1, \dots, f_n$ and then proceed from there. The above is a bit of a simplification. The reading avoids this simplification.

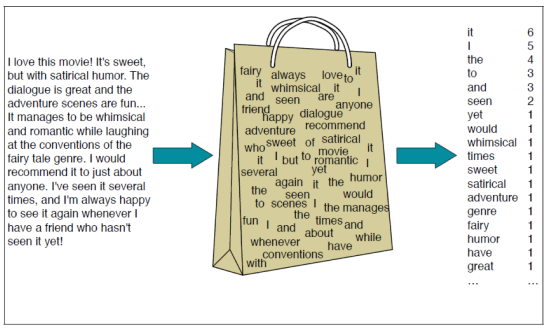


Figure 7.1 Intuition of the multinomial naive Bayes classifier applied to a movie review. The position of the words is ignored (the *bag of words* assumption) and we make use of the frequency of each word.

Figure from J&M 3rd ed. draft, sec 7.1

But this model will only be valid if we can calculate probabilities of a label for the exact multiset of each sentence's words. We need another assumption, the *Naive Bayes assumption* which is that the probability of a label given a word is conditionally independent of the other words.

Note from Bayes' rule:

$$P(y|w_1, w_2, \ldots, w_n) = \frac{P(w_1, w_2, \ldots, w_n|y)P(y)}{P(w_1, w_2, \ldots, w_n)}$$

and since the word sequence itself is constant, we can say

$$\text{argmax}_y\, P(y|w_1, w_2, \ldots, w_n) = \text{argmax}_y\, P(w_1, w_2, \ldots, w_n|y)P(y)$$

The conditional probability assumption, which makes up the Naive Bayes assumption, can then be applied, so we assume

$$P(w_1, w_2, \ldots, w_n|y)P(y) = P(w_1|y)P(w_2|y), \ldots, P(w_n|y)P(y)$$

Is this a good model? George Box, statistician: "All models are wrong, but some models are useful."

What are problems with the bag of words assumption?

What are problems with the naive bayes assumption?

Does it work? Yes, for many tasks actually. And it's very simple so it's usually worth trying.

Here's a new trainer:

```
from collections import Counter, defaultdict
def train(labeled_sentences):
  wscores = defaultdict(lambda: Counter()) # doubly nested structure
  cscores = Counter()
  for sentence, label in labeled_sentences:
    for word in sentence.split():
      wscores[label][word]+=1
      cscores[label]+=1
  model = {'cprobs':{}, 'wprobs':{}}
  for label in cscores.keys():
```

```
    model['cprobs'].append(cscores[c]/len(labeled_sentences))
    wprob = {}
    for word, score in wscores[label]:
      wprob[word] = score/cscores[label]
    model['wprobs'].append(wprob)
  return model
```

And a new, more appropriate classifier

```
def evaluate(sentence, option, model):
  score = model['cprobs'][option]
  for word in sentence.split():
    score *= model['wprobs'][option][word]
  return score
```

4.4.1 Practicalities: smoothing

`score *= model['wprobs'][option][word]` is going to be problematic if we have never seen a word with a particular class. Solution: smoothing!

Laplace (add-1) smoothing: assume you've seen every word (even words you haven't seen before) with every class!

Before (assume 10k words in training set of negative items):

$P(\text{amazing}|\text{negative}) = 0/10,000 = 0$ (seen the word but not with class 'negative')

$P(\text{blargh}|\text{negative}) = 0/10,000 = 0$ (never seen the word)

Introduce a new term, 'OOV', and if you haven't seen your test word during training, pretend your word is 'OOV'. Then, since you add 1 for each vocabulary word and the OOV with each class, if your vocabulary size was 500, you now get:

$P(\text{amazing}|\text{negative}) = 1/10,501$

and

$P(\text{blargh}|\text{negative}) = 1/10,501$

You may want to actually *introduce* some OOV into your training set, by replacing words that appear fewer than some k times with OOV. This is so that your model can learn how to behave with OOVs.

4.4.2 Practicalities: underflow

Recall:

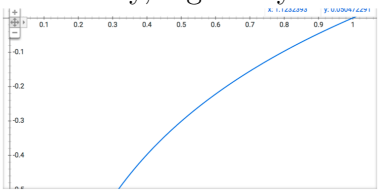
```
  for word in sentence.split():
    score *= model['wprobs']['option']['word']
```

Sentence may be long! Probabilities may be small! It's very easy to run into underflow: try this:

```
a=1
for i in range(100):
    a*=.0001
    if i % 10 == 0:
        print(i, a)
```

```
0 0.0001
10 1.0000000000000003e-44
20 1.0000000000000007e-84
30 1.000000000000001e-124
40 1.0000000000000015e-164
50 1.0000000000000021e-204
60 1.0000000000000029e-244
70 1.0000000000000036e-284
80 0.0
90 0.0
```

Thankfully, logs are your friend, because of the following graph of $\log(x)$:



So instead rewrite the function as

```
from math import log

def evaluate(sentence, option, model):
  score = log(model['cprobs'][option])
  for word in sentence.split():
    score += log(model['wprobs']['option']['word'])
  return score
```

(Note the operator change)