COMP5046: Regular Expressions and Language Modelling

Joel Nothman

joel.nothman@sydney.edu.au

School of Information Technologies
University of Sydney

2018-03-13

How do we check if some string is valid language?

- Match it against **known patterns**
  - ⇒ regular expressions
  - ⇒ parse it with a grammar (later)
- Build a model of **language probability**
  - ⇒ language modelling
  - ⇒ parse it with a probabilistic grammar (later)

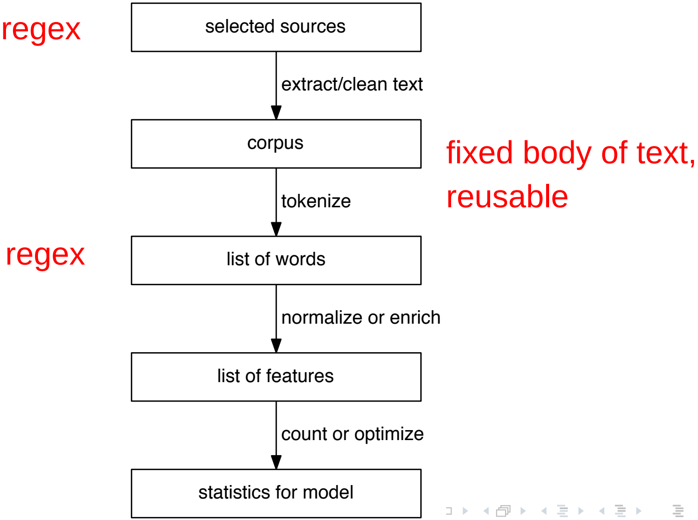
How do we check if some string is valid language?

- Match it against **known patterns**
  - ⇒ regular expressions
  - ⇒ parse it with a grammar (later)
- Build a model of **language probability**
  - ⇒ language modelling
  - ⇒ parse it with a probabilistic grammar (later)
- How do we check if some string is a past tense verb?
- How do we check if some string is a person's name?
- ...

Part I

Basic text processing

The NLP pipeline



Segmentation

- Sentence boundaries
  - needed for standard syntactic processing
  - indicated in English by [.!?:]
  - but . is ambiguous (Dr., etc., ..., 1.)
- Word segmentation: some languages are written with no spaces
- Tokenisation:
  - tools expect consistent handling of punctuation
  - how **token** is defined affects statistics
  - punctuation can be meaningful, often separate from words
  - what's → what 's
  - Joel's → Joel 's
  - didn't → did n't
  - Sydney-based → ?
  - performed with a series of regular expressions or similar

Annotation formats

- Inline markup:

```
<s><np><t pos=DT>The</t> <t pos=NN>cat</t></np>
<t pos=VBD lemma=eat>ate</t><t>.</t></s>
```

- Stand-off markup:

start	stop	type	attributes
0	3	token	pos=DT
4	7	token	pos=NN
8	11	token	pos=VBD lemma=eat
11	12	token	pos=.
0	7	phrase	type=NP
0	12	sentence	

- Fixed tokens ("CoNLL shared task style"):

```
The      the      DT      B-NP
cat      cat      NN      I-NP
ate      eat      VBD     O
.        .        .       O
# blank line ends sentence
```

We don't rely on statistics for everything

- use our linguistic/task knowledge to analyse data
- determinism can be easier to:
  - implement (initially)
  - interpret
- harder to adapt, extend
- build statistical systems around engineered analyses
- need tools for analysis and manipulation of strings

Solving crosswords

- I have a crossword puzzle which I can't finish

k			d			g					

- and being a programmer I want the computer to do the work
- One step would be to find all the words containing k, d and g

```
% grep k /usr/share/dict/words | grep d | grep g
acknowledge
acknowledgeable
acknowledged
acknowledgedly
...
% grep k /usr/share/dict/words | grep d | grep g | wc -l
277
```

- hmm, still too many words to read!

Solving crosswords in Python

- We could write a Python program to find the answer:

```
>>> for line in open('/usr/share/dict/words'):
...     if len(line) > 6 and line[0] == 'k' and \
...        line[3] == 'd' and line[6] == 'g':
...       print line,
...
kindergarten
>>>
```

- but this seems like too much work

We need a language for describing text

- such pattern languages already exist:
    - e.g. filenames with wildcards: `*.txt`, `note?ad.exe`
    - e.g. wildcards in search in Word
- even hangman (not the gallows bit)
- but we need more power!

Regular expressions describe strings

- *Regular expressions* are a very expressive pattern language
- often abbreviated to *RE*, *regex*, *regexp*
- `grep` can interpret regular expressions:

```
% grep 'k..d..g.....' /usr/share/dict/words
kindergarten
kindergartener
...
```

- The dot (`.`) is a placeholder that represents *any* character
- `grep` prints any line which *contains* a match:

```
% grep 'k..g' /usr/share/dict/words
afluking
afterking
afterworking
akazga
```

We can force where matches start and end

- The *caret* or *hat* (^) matches the beginning of string/line

```
% grep '^dog' /usr/share/dict/words
dog
dogal
dogate
dogbane
dogberry
...
```

- While dollars ($) matches the end of string/line

```
% grep 'to$' /usr/share/dict/words
adagietto
agrito
ailanto
...
```

Kleene star matches zero or more times

- What if we don't know how many characters to match
- The star (*) matches the previous char[1] zero or more times

```
% grep '^t.*o$' /usr/share/dict/words
tabanuco
taboo
tacso
...
```

- It is called the *Kleene star* operator
- The plus (+) matches one or more times
- + requires the `-E` option to `grep`
  and not all versions of grep support `-E`

[1] not true! This is a deliberate lie – for now

Matching against alternatives

- Sometimes we want to match one regular expression or another
- The pipe (`|`) matches one among alternatives
- Again it requires the `-E` option to `grep`
- or use `egrep` (which is identical to `grep -E`):

```
% egrep '^dog|^cat' /usr/share/dict/words
cat
catabaptist
...
dog
dogal
...
```

Character classes represent many alternatives compactly

- To find all words starting with vowels we could go:

```
% egrep '^a|^e|^i|^o|^u' /usr/share/dict/words
```

- This is clumsy, and what if we wanted all consonants instead?
- Regular expressions abbreviate this using *character classes*:

```
% grep '^[aeiou]' /usr/share/dict/words
```

- The square brackets (`[` and `]`) delimit the character class
- It can match any single character from the character class

Character classes represent ranges compactly

- If we wanted all lowercase characters we would need to write:

```
% grep '^[abcdefghijklmopqrstuvwxyz]' /usr/share/dict/words
```

- This is long and error prone

Character classes represent ranges compactly

- If we wanted all lowercase characters we would need to write:

```
% grep '^[abcdefghijklmopqrstuvwxyz]' /usr/share/dict/words
```

- This is long and error prone
- Did you noticed I missed `n`?

Character classes represent ranges compactly

- If we wanted all lowercase characters we would need to write:

```
% grep '^[abcdefghijklmopqrstuvwxyz]' /usr/share/dict/words
```

- This is long and error prone
- Did you noticed I missed `n`?
- Character classes support *ranges*:

```
% grep '^[a-z]' /usr/share/dict/words
```

- So we can match Python 2 variable names with:

```
% grep '^[a-zA-Z_][a-zA-Z0-9_]*$' /usr/share/dict/words
```

ensures entire line is alpha numeric or underscore

Character classes also support set complements

- The *complement* of a set is all of the elements *not* in the set
- Here this means all of the *characters not in the set*
- The caret (^) is used as the *first* character inside the character class:

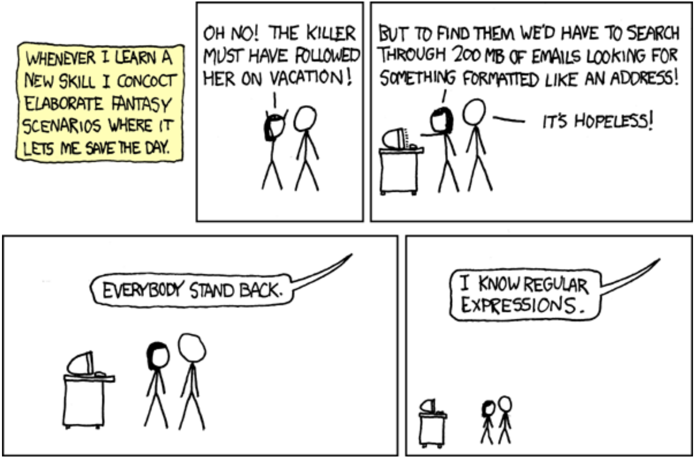
```
% grep '^[^A-Zaeiou]' /usr/share/dict/words
b
ba
baa
baahling
...
```

- the caret now means two things in regular expressions

grep command line arguments

- `-E` extended regular expressions
- `-i` case insensitive regular expressions
- `-l` only print filenames
- `-n` also print line numbers
- `-r` recursive directory search
- `-v` lines that don't match

XKCD on Regular Expressions (by Randall Munroe)



http://xkcd.com/c208.html

Regular expressions and NLP

regex can be used for basic tokenisation
+ means more than 1

- Tokenisation: `findall('[a-zA-Z-]+')`
  "We're getting intense; here's a full-blown sentence!" she rapped.
- Capturing morphology (e.g. suffixes): `sub('([a-z])\1ed$', '\1')`
  hopped → hop; skipped → skip; missed → ?mis

\1: find the thing that's in ()
not expected to know

- Patterns for extracting names and relations:
  PERSON is the TITLE of LOC
- Finding a phenomenon of interest in text with character, word or tag patterns
- Identifying types of error in a system

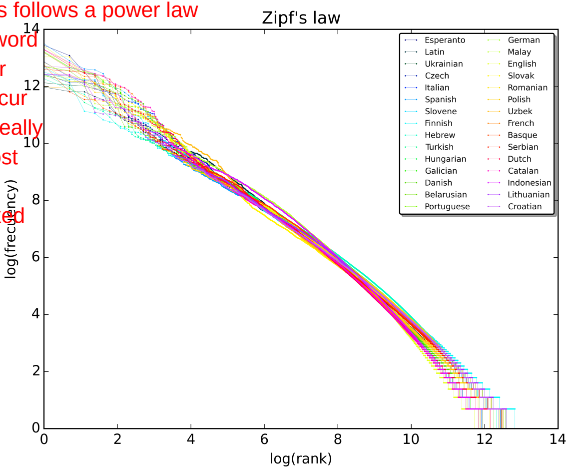
- Matching regular expressions is efficient
- Natural language is *not* a regular language

Part II

The probability of language

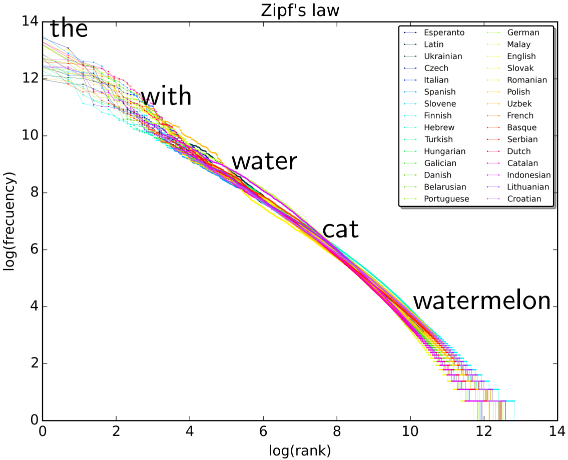
Zipf's Law: $cf_r \propto \frac{1}{r}$

distribution of words follows a power law
cumulative freq of word is proportional to 1/r
i.e. words which occur frequently, occurs really frequently - and most words do not occur frequently in any fixed corpus of text



Attribution: CC-BY-SA-4.0 by Sergio Jimenez

Zipf's Law: $cf_r \propto \frac{1}{r}$



Attribution: CC-BY-SA-4.0 by Sergio Jimenez

Some completions are more likely than others

A long long

Some completions are more likely than others

A long long way
time
?stick
*the
*orange
*.

Calculating the probability of a string as language

- Imagine a system which:
  given a string of characters
  tells us its **probability** in a particular language
  as learnt from a **corpus**

- What could we do with such a system?

- What could we do with many models trained on different corpora?

Calculating the probability of a string as language

- Imagine a system which:
  given a string of characters
  tells us its **probability** in a particular language
  as learnt from a **corpus**

- What could we do with such a system?
  correct spelling; identify the next column in a newspaper; anticipate the rest of a sentence; choose the most natural translation / summary / speech recognition

- What could we do with many models trained on different corpora?
  categorise text: language ID, authorship analysis, sentiment analysis; find the best document to match a query; ...

Reminder: Conditional Probability

- partial knowledge of an outcome that informs our model
  already seen outcome of first dice roll
- conditional probability $P(A|B)$ (said *probability of A given B*)

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{1}$$

Language Modelling

- Find the *best sequence*:
  - words
  - characters
  - tags
  - base pairs
  - ...
- **Which sequence** is the best sequence?
  $\Longrightarrow$ the most probable sequence

$$\operatorname*{argmax}_{y_1 \ldots y_n} p(y_1 \ldots y_n)$$

(this means: the setting of $y_1 \ldots y_n$ that maximises $p(y_1 \ldots y_n)$)

- we need a **probability model of language**

Language Modelling

- a Language Model is:

$$p(y_1 \ldots y_n)$$

- Chain rule expansion:

$$p(y_1 \ldots y_n) = p(y_1)p(y_2|y_1)p(y_3|y_1, y_2) \cdots p(y_n|y_1, \ldots, y_{n-1})$$

predict y_1
predict y_2 given y_1
predict y_3 given y_1 and y_2
...

Markov Assumption

- **Each prediction cannot depend on entire history!**
- But probability of each word is *most* influenced by recent history
- Markov model approximation:

$$
\begin{aligned}
p(y_1 \ldots y_n) &= p(y_1)p(y_2|y_1)p(y_3|y_1, y_2) \cdots p(y_n|y_1, \ldots, y_{n-1}) \\
&\approx p(y_1)p(y_2|y_1)p(y_3|y_2) \cdots p(y_n|y_{n-1})
\end{aligned}
$$

- In theory can use any fixed length history
- In practice a history of 2 is typically used (for English)

n-grams

- **bigram** is a pair/tuple of words (but 1st order Markov model)
- **trigram** is a 3-tuple of words (2nd order Markov model)
- **four-/quad-gram** is a 4-tuple of words (3rd order Markov model)
- **unigram** is a single word (zero-th order Markov model)

- not necessarily the adjacent words

Model Parameters

- assuming around 20,000 unique unigrams, there are (approximately):
  - $20\,000^2$ = **400 million bigrams**
  - $20\,000^3$ = **8 trillion trigrams**
  - $20\,000^4$ = **1.6×10^{17} 4-grams**
  - ...
- in a 100 million word corpus (e.g. the BNC), at most:
  - **1 in 4 bigrams** will be seen
  - **1 in 80 000 trigrams** will be seen
- the larger the n-gram: larger the n-grams, the stats data becomes sparse since counts of larger n-grams is far less
  - the less statistical evidence we have for each parameter (reliability)
  - the more contextual evidence we have for each prediction (discrimination) easier to predict next word

Google Web 1T

- in 2006, Google released web counts for 1 trillion words
  well 1 024 908 267 229 tokens

Number of sentences	95 119 665 584
Number of unigrams	13 588 391
Number of bigrams	314 843 401
Number of trigrams	977 069 902
Number of fourgrams	1 313 818 354
Number of fivegrams	1 176 470 663

- each unigram appears at least 200 times (rest replaced with `<UNK>`)
- each n-gram appears at least 40 times
- data is 24GB (compressed with gzip)

Reducing Model Size and Sparsity

- want a good trade-off between reliability and discrimination
- choice of *n* and vocabulary size

- equivalence classes: replace words in n-gram with fewer options
- clump together things that are similar for the purposes of prediction

- words with same base morpheme (stemming)
- words with similar function (POS tags)
- words with similar semantics (Pereira, Tishby and Lee, 1993)
- using WordNet synsets (Clark, 2001)

Maximum Likelihood Estimation

- **Model should match the evidence (training data)**
  $\implies$ the model that maximises the probability of training data
- called the **Maximum Likelihood Estimate** (MLE)
- Estimates are simple relative frequencies:

$$p(y_i|y_{i-1}) = \frac{p(y_{i-1}, y_i)}{p(y_{i-1})} \quad (2)$$

$$= \frac{\text{count}(y_{i-1}, y_i)}{\text{count}(y_{i-1})} \quad (3)$$

Arrggghh!

- **unreliable zero or very low counts are problematic**
  - does a zero count indicate an impossible event?
  - very small counts that are highly variable in ratios
- MLE assigns ZERO mass to unseen events!
  - no mass wasted outside of training data
  - cannot be MLE and have mass apportioned to unseen events
  - need to steal probability mass from seen events and give to unseen

Laplace's Law (or adding one)

- one solution is to add a single count for all features
- must normalise the result so that it is still a probability
- BUT now there is a mass of 1 for all possible n-grams
- typically this size is much larger than number of seen n-grams

$$p_{lap}(y_1, \ldots, y_n) = \frac{\text{count}(y_1, \ldots, y_n) + 1}{N + B}$$

- where N is number of seen n-grams, B is the number of possible n-grams

Lidstone's Law

- Lidstone proposed adding less (λ) than a full count

$$p_{lid}(y_1, \ldots, y_n) = \frac{\text{count}(y_1, \ldots, y_n) + \lambda}{N + B\lambda}$$

- can be seen as linear interpolation between MLE and uniform prior

$$p_{lid}(y_1, \ldots, y_n) = \mu \frac{\text{count}(y_1, \ldots, y_n)}{N} + (1 - \mu)\frac{1}{B}$$

where $\mu = N/(N + B\lambda)$

- equivalent to Bayesian estimator with uniform prior
- significant improvement, but fundamental problem remains

Held-out Estimation

- what is the empirical expectation of words of a given seen frequency?
  $\Longrightarrow$ look at some previously unseen data (**held-out data**)
- Church and Gale (1991) use 44 million words split into two

$$\begin{aligned}
\underset{1}{\text{count}}(y_1,\ldots,y_n) &= \text{frequency of } y_1,\ldots,y_n \text{ in training data} \\
\underset{2}{\text{count}}(y_1,\ldots,y_n) &= \text{frequency of } y_1,\ldots,y_n \text{ in held out data} \\
T_r &= \sum_{\{y_1,\ldots,y_n \mid \underset{1}{\text{count}}(y_1,\ldots,y_n)=r\}} \underset{2}{\text{count}}(y_1,\ldots,y_n) \\
p_{ho}(y_1,\ldots,y_n) &= \frac{T_{\underset{1}{\text{count}}(y_1,\ldots,y_n)}}{N_{\underset{1}{\text{count}}(y_1,\ldots,y_n)}N} \overset{abbrev}{=} \frac{T_r}{N_r N}
\end{aligned}$$

- where N_r is the number of training n-grams with frequency r

Other n-gram probability estimates

- Deleted and k-fold estimation:
  - Average over multiple held-out estimates
- Good-Turing estimation (Good, 1953; Sampson and Gale, 1995)
  - based on the ratio between the # of n-grams appearing $r + 1$ and r times
  - i.e. what is the probability of the n-gram appearing one more time?
- Modified Kneser-Ney smoothing (Chen and Goodman, 1999)
  - Estimate not only based on r
  - Francisco has high unigram MLE based on frequency alone
  - KN assigns low unigram probability because few histories precede it

Mixed n: Linear Interpolation

- Try to combine different estimates using a linear function
- typical example is to combine trigram, bigram and unigram models

$$p_{li}(y_n|y_{n-2}, y_{n-1}) = \lambda_1 p_1(y_n) + \lambda_2 p_2(y_n|y_{n-1}) + \lambda_3 p_3(y_n|y_{n-2}, y_{n-1})$$

- with $0 \leq \lambda_i \leq 1$ and $\sum_i \lambda_i = 1$
- optimise the selection of λ_i using EM or numerical optimiser

Mixed *n*: Katz's Back-off model

- **only consider more specific model if there is enough evidence**
- otherwise fall back to a smaller model
  - a smaller n-gram
  - a smaller set of equivalence classes
- then choose weights to normalise and produce distribution

Once you can model the probability of an n-gram

⇒ can calculate the probability of an 1-gram given the preceding (n-1)-gram
(from the definition of conditional probability)

⇒ can calculate the probability of a sequence
(applying the Markov assumption)

⇒ can classify text as likely coming from one population or another
(with Bayes' rule; see week 5)

Learning probabilities from an encoding of history

- Recurrent Neural Network Language Model (Mikolov et al., 2010)
  - Sparse history is mapped into a low-dimensional space
  - Similar histories are clustered
  - $P(y_n|y_1,\ldots,y_{n-1}) \approx P(y_n|\tilde{h}_n)$, $\tilde{h}_n \in \mathbb{R}^d$
  - Can also encode sub-word information
- Fixed Ordinally-Forgetting Encoding (FOFE; Zhang et al., 2015)
  - Represent each sequence of words uniquely as a vector
  - Recent words get larger weight in the vector
  - Learn a function mapping each vector to a probability
- These do not make the Markov assumption
- Parameter space not exponential in n
- How might you learn the functions from a corpus?

Take away

- Why we might model language probability
  - we do not explicitly use language models for much of this course, but many ideas are fundamental
- Sparsity and power law distribution of language
- Markov approximation and n-grams
- Smoothing for generalisation despite sparsity
- Trade-offs: reliability, discrimination and efficiency (model size)
- Combining models for multiple n

- Why consistent tokenisation matters
- Regular expression notation and applications