



Centre for Next Generation Localisation



# An Introduction to Language Model

Debasis Ganguly

Johannes Leveling

Gareth Jones



National Development Plan 2007 - 2013



science foundation Ireland  
fondúireacht eolaíochta Éireann



Dublin City University



University College Dublin



University of Limerick



Trinity College Dublin

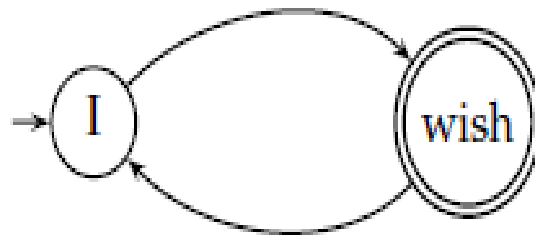
# Overview

---

- LM as a generative model
- Query generation models
- Smoothing
- Deriving term weights
- Relevance Feedback in LM
- How to implement LM yourself in Lucene

# Generative Model

- A generative model is formalized by Finite State Automata theory.
- An FSA is a 5 tuple  $M=(Q, \Sigma, \delta, q_0, q_f)$



I wish

I wish I wish

I wish I wish I wish

I wish I wish I wish I wish I wish I wish

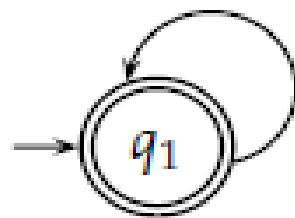
...

CANNOT GENERATE: wish I wish

## Generative Model (Contd.)

- A one state finite automaton can act as a unigram language model if the node has a probability distribution of term generation such

that  $\sum_{s \in V} P(s) = 1$



$$P(\text{STOP} | q_1) = 0.2$$

the	0.2
a	0.1
frog	0.01
toad	0.01
said	0.03
likes	0.02
that	0.04
...	...

## Generative Model (Contd.)

- To find the probability of a word sequence we multiply the individual generation probabilities
- e.g.  $P(\text{frog said that toad likes frog} | M_1) = (0.01 * 0.8) * (0.03 * 0.8) * \dots (0.01 * 0.2)$
- The Language model that gives a higher probability is more likely to generate a given sequence

Model $M_1$		Model $M_2$	
the	0.2	the	0.15
a	0.1	a	0.12
frog	0.01	frog	0.0002
toad	0.01	toad	0.0001
said	0.03	said	0.03
likes	0.02	likes	0.04
that	0.04	that	0.04
dog	0.005	dog	0.01
cat	0.003	cat	0.015
monkey	0.001	monkey	0.002
...	...	...	...

# Types of Language Models

## ● Unigram

- The probability of generation of a term is an independent event and does not depend on the previous terms being generated

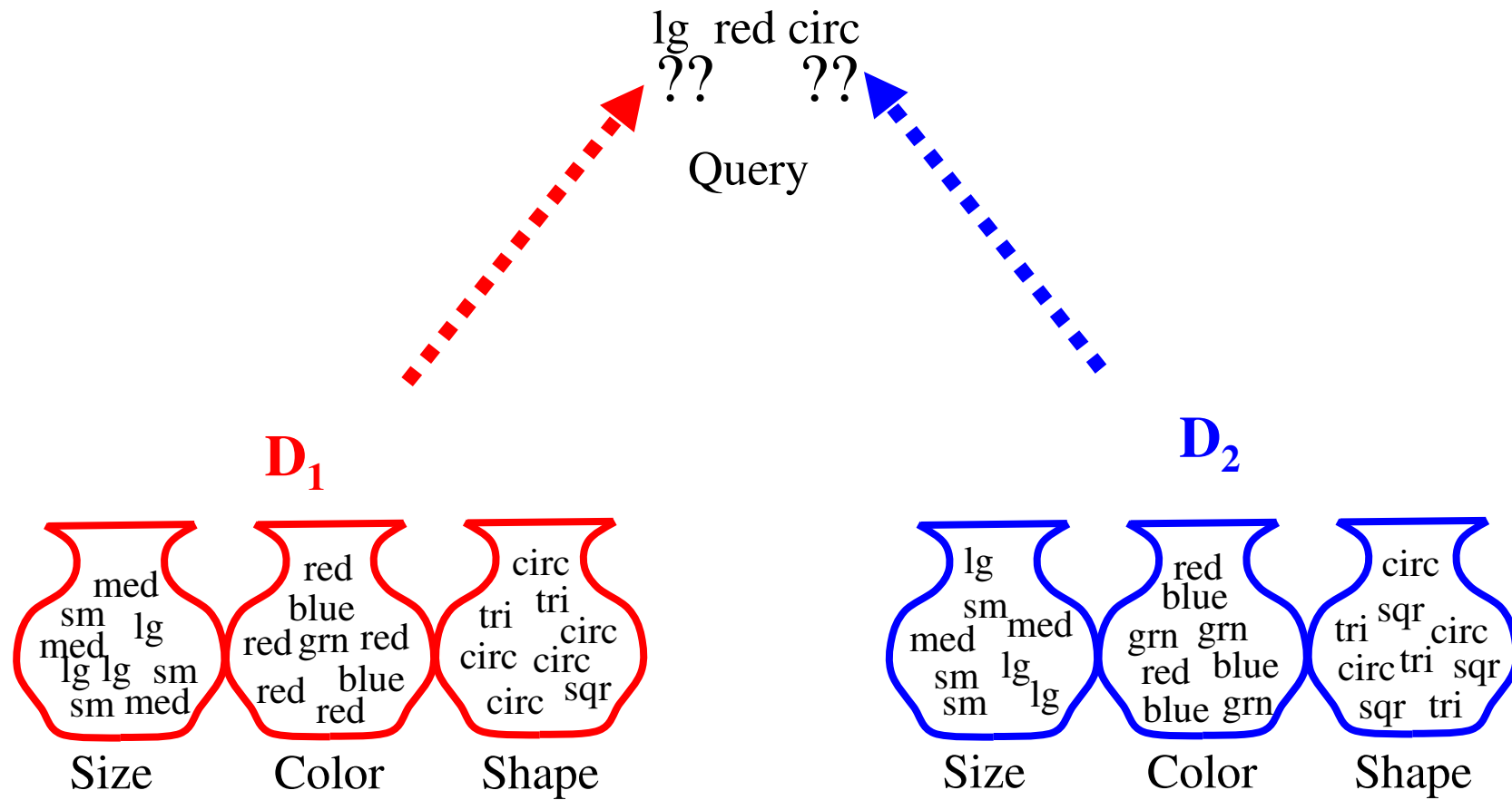
$$P(t_1 t_2 \dots t_n) = \prod_{i=1}^n P(t_i)$$

## ● Bigram

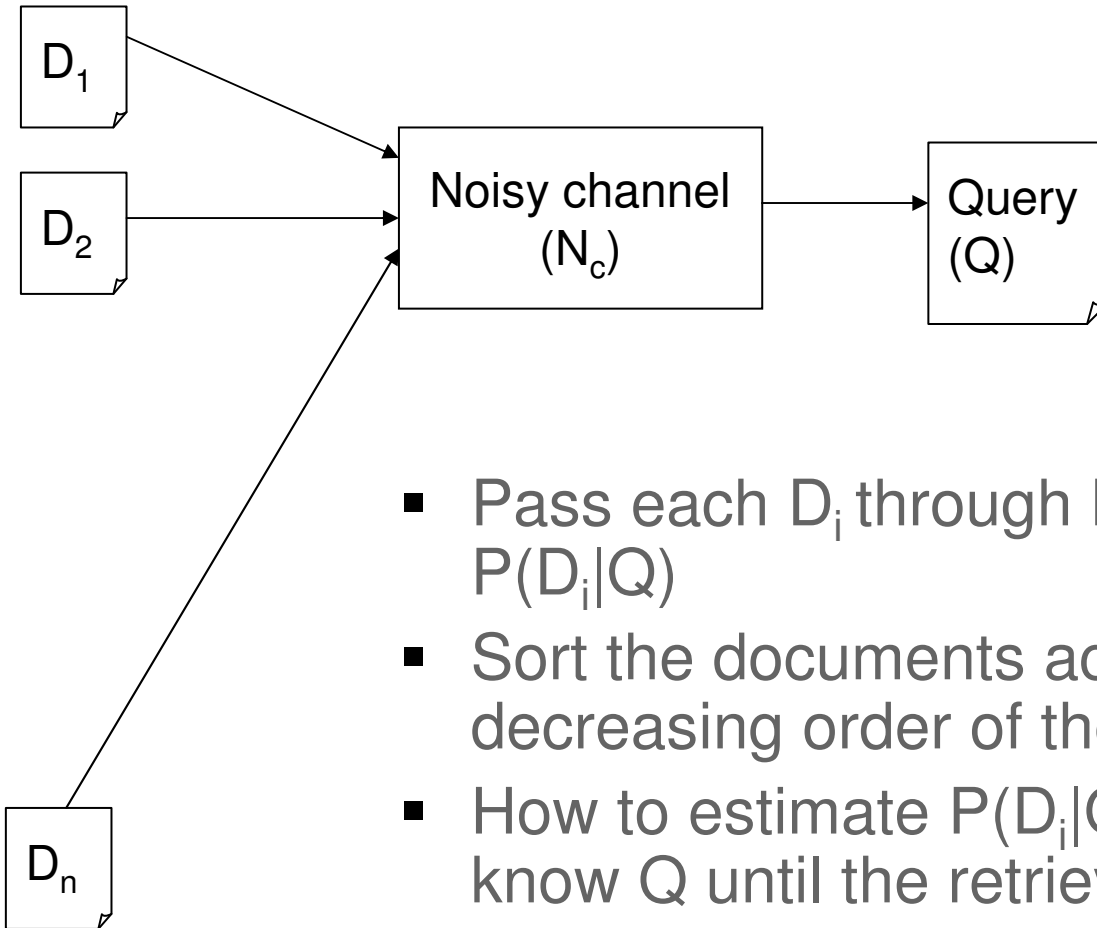
- The probability of generation of a term  $t_i$  is conditioned on the previous term being generated

$$P(t_1 t_2 \dots t_n) = P(t_1) \prod_{i=2}^n P(t_i | t_{i-1})$$

# Query likelihood model



# Query generation model in IR



- Pass each  $D_i$  through  $N_c$  and calculate  $P(D_i|Q)$
- Sort the documents according to the decreasing order of these probability values
- How to estimate  $P(D_i|Q)$  because we do not know  $Q$  until the retrieval step

# Bayes Theorem

- $P(D_i|Q) = P(D_i) * P(Q|D_i) / P(Q)$
- $P(Q)$  is constant and hence can be ignored
- The prior probability  $P(D_i)$  can be uniform for all documents or proportional to the length of a document
- How can we estimate  $P(Q|D_i)$ ?
- Let  $Q = (t_1, \dots, t_n)$  be generated by a unigram LM underlying  $D_i$

$$P(t_1 t_2 \dots t_n) = \prod_{j=1}^n P(t_j | D_i)$$

## Estimating probability of term generation

- Use the Maximum Likelihood estimate to estimate  $P(t_i|D)$
- Hence  $P(t_i|D) = \text{tf}(t_i, D) / |D|$
- Is this enough?
- What about a missing term? Even if  $n-1$  terms have a high frequency a single missing term will bring the estimated probability down to 0!
- A strict Boolean conjunctive query which is not very conducive to adhoc retrieval!
- Smoothing comes to the rescue

# Characteristics of the smoothing function

- Should give some probability mass to unseen words
- Should not overestimate the probabilities of words occurring say only once since they might occur by chance!
- Did we say chance? From where?
- The collection
- A simple way to smooth is thus to combine the events of generating a term from the document as well as from the collection.

$$P(t_1 t_2 \dots t_n) = \prod_{j=1}^n \lambda_j P(t_j | D_i) + (1 - \lambda_j) P(t_j)$$

## Another interpretation of smoothing

- The  $\lambda_i$ s are not only a coefficient of linear interpolation between two probabilities but also has a more natural interpretation
- $\lambda_i$  can be interpreted as the importance of the query term  $t_i$  to the user. A higher value of  $\lambda_i$  suggests that the user would like to attach a higher importance to the presence of the term in the document
- $\lambda_i$ s need not be constant and can typically be learnt by applying the Expectation-Maximization Algorithm
- Typically  $\lambda_i$ s are kept constant for all the query terms

## Deriving the term weights

- The weight assigned to a document  $d$  for query  $q=(t_1, \dots, t_n)$

$$P(t_1 t_2 \dots t_n | d) = \prod_{i=1}^n \lambda P(t_i | d) + (1 - \lambda) P(t_i)$$

- $P(t_i | d)$  is estimated as  $tf(t_i, d) / \text{sum of tfs}$
- $P(t_i)$  is estimated as  $df(t_i) / \text{sum of dfs}$

$$P(t_1 t_2 \dots t_n | d) = \prod_{i=1}^n \lambda \cdot \frac{tf(t_i, d)}{\sum_{t \in d} tf(t, d)} + (1 - \lambda) \cdot \frac{df(t_i)}{\sum_{t \in C} df(t)}$$

- Note that since the probabilities are small numbers and they are multiplied together, underflow may occur. Hence it's log transform is taken

## Deriving the term weights (Contd.)

$$\log P(t_1 t_2 \dots t_n | d) = \sum_{i=1}^n \log \left( 1 + \frac{\lambda}{1 - \lambda} \frac{tf(t_i, d) \sum_{t \in C} df(t)}{\sum_{t \in d} tf(t, d) \cdot df(t_i)} \right)$$

- The nice summation structure conforms to the vector dot product computation and enables storing the document and queries as vectors

# Relevance Feedback in LM

# Expectation Maximization (Hiemstra)

- Learn the  $\lambda$  s by EM algorithm

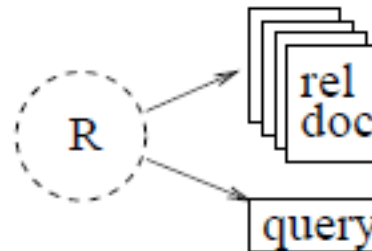
$$\text{E-step: } m_i = \sum_{j=1}^r \frac{\lambda_i^{(p)} \cdot P(T_i = t_i | D_j = d_j)}{(1 - \lambda_i^{(p)})P(T_i = t_i) + \lambda_i^{(p)} P(T_i = t_i | D_j = d_j)}$$

$$\text{M-step: } \lambda_i^{(p+1)} = \frac{m_i}{r}$$

- The more a term occurs in the set of relevant documents and the less it appears in the collection higher is the  $\lambda$

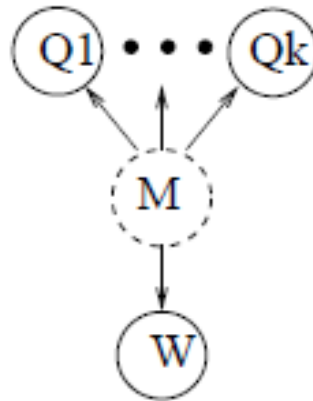
# Relevance Feedback in LM (Lavrenko)

- The generative model of a document generating a query is modified
- Aims to estimate a relevance model which generates the query and the relevant documents
- We want to estimate  $P(w|R)$ .
- But we don't know  $R$ .



# Relevance Model

- $P(w|R)$  can be estimated from the observation of  $q=(t_1, \dots, t_n)$ . Thus  $P(w|R) \approx P(w|t_1, \dots, t_n)$
- Assume a document model  $M$  which generates the query terms as well as the word  $w$



- Assume a document model  $M$  which generates the query terms as well as the word  $w$

## Relevance Model (Contd.)

$$P(w, t_1, \dots, t_n) = \sum_{M \in M_d} P(M) P(w, t_1, \dots, t_n | M)$$

$$P(w, t_1, \dots, t_n) = \sum_{M \in M_d} P(M) P(w | M) \prod_{i=1}^n P(t_i | M)$$

- The more the co-occurrence of a word  $w$  with the query terms in a document greater is the probability  $P(w|R)$
- $M_d$  is the underlying document generation model of document  $d$
- Typically restrict the set of pseudo relevant documents to top 20 documents

## Relevance Model (Contd.)

---

- After estimating the relevance model documents are then ranked by KL-divergence between  $R$  and  $M_d$

$$KL(R \parallel M_d) = \sum_w P(w/R) \log \frac{P(w/R)}{P(w/M_d)}$$

- RLM is the default retrieval model in the Indri search engine

## Implementing LM in Lucene

- Default Scoring model of Lucene is a generic tf-idf weighting model
- $\text{score}(q,d) = \text{coord}(q,d) \cdot \text{queryNorm}(q) \cdot \sum_{t \text{ in } q} ( \text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost}() \cdot \text{norm}(t,d) )$
- Extend the TermQuery class and override the createWeight() method.

```
public Weight createWeight(Searcher searcher)
throws IOException {
    return new LMTermWeight(searcher, this);
}
```

- The createWeight() method returns an Weight object which should also be subclassed
- ```
public Scorer scorer(IndexReader reader, boolean ig1, boolean ig2) throws IOException {  
    return new LMTermScorer(this.searcher, reader, this);  
}
```
- Again subclass Scorer and implement the method score()

---

# Questions?