NATURAL LANGUAGE PROCESSING

INTRO-WEEK1

THIRUMURUGAN.R

MAIN APPROACHES IN NLP:

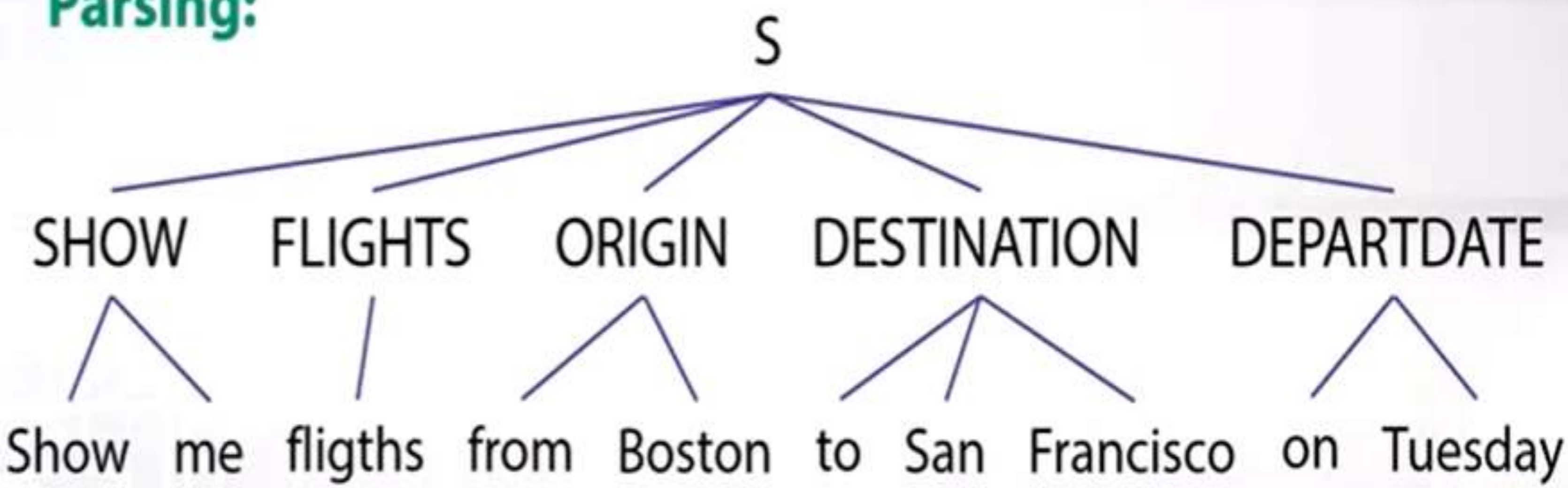
1. **Rule-based methods**
   - Regular expressions
   - Context-free grammars
   - …
2. **Probabilistic modeling and machine learning**
   - Likelihood maximization
   - Linear classifiers
   - …
3. **Deep Learning**
   - Recurrent Neural Networks
   - Convolutional Neural Networks

Semantic slot filling: CFG

Context-free grammar:

- SHOW → show me | i want | can i see |…
- FLIGHTS → (a) flight | flights
- ORIGIN → from CITY
- DESTINATION → to CITY
- CITY → Boston | San Francisco | Denver | Washington

Parsing:



Semantic Slot Filling: CRF

Training corpus:

ORIG DEST DATE

Show me flights from Boston to San Francisco on Tuesday.

Feature engineering:

- Is the word capitalized?
- Is the word in a list of city names?
- What is the previous word?
- What is the previous slot?
- ….

Semantic Slot Filling: CRF

Probabilistic graphical model:

- Conditional Random Field (CRF)

$$p(\text{tags}|\text{words}) = \ldots$$

features

parameters Θ

Training:

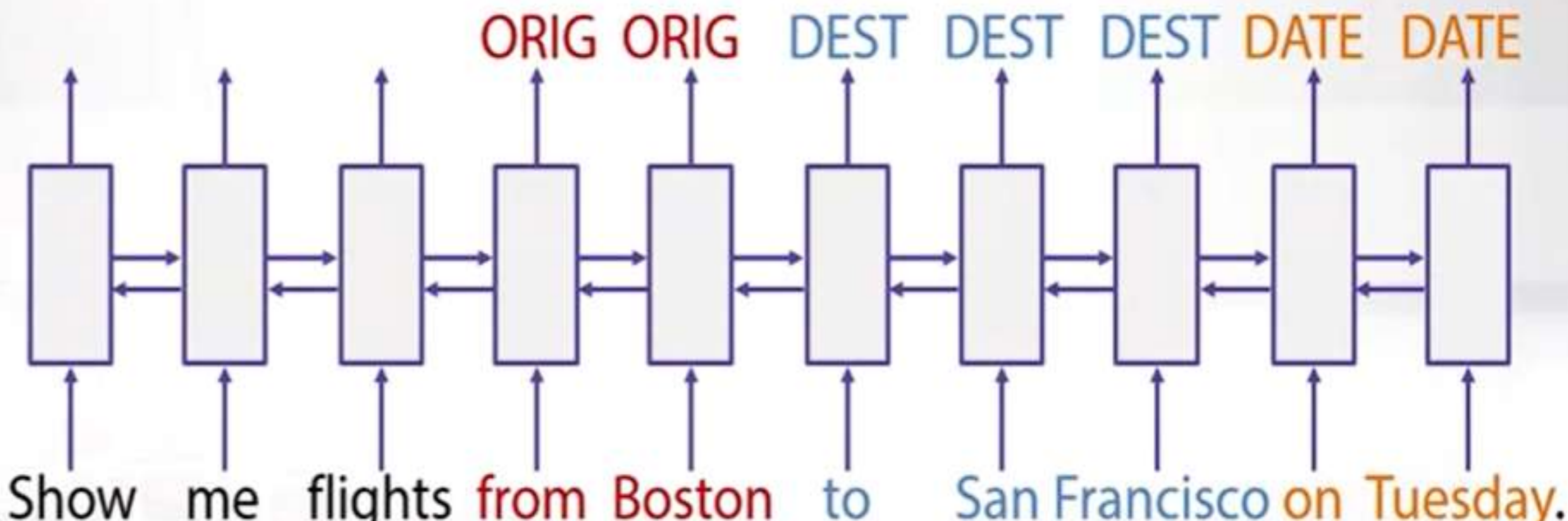
$$p(\text{tags}|\text{words}) \to \max_{\Theta}$$

Inference:

$$\text{tags}^* = \text{argmax}\ p(\text{tags}|\text{words})$$

Semantic Slot Filling: LSTM

- Big training corpus
- No feature generation
- Defining the model
- Training and inference



Why do we need to study traditional NLP?

- Perform good enough in many tasks

  Example: sequence labeling
- Allow us not to be blinded with the hype

  Example: word2vec / distributional semantics
- Can help to further improve DL models

  Example: word alignment priors in machine translation

Why do we need to study DL in NLP?

- Provide state-of-the-art performance in many tasks

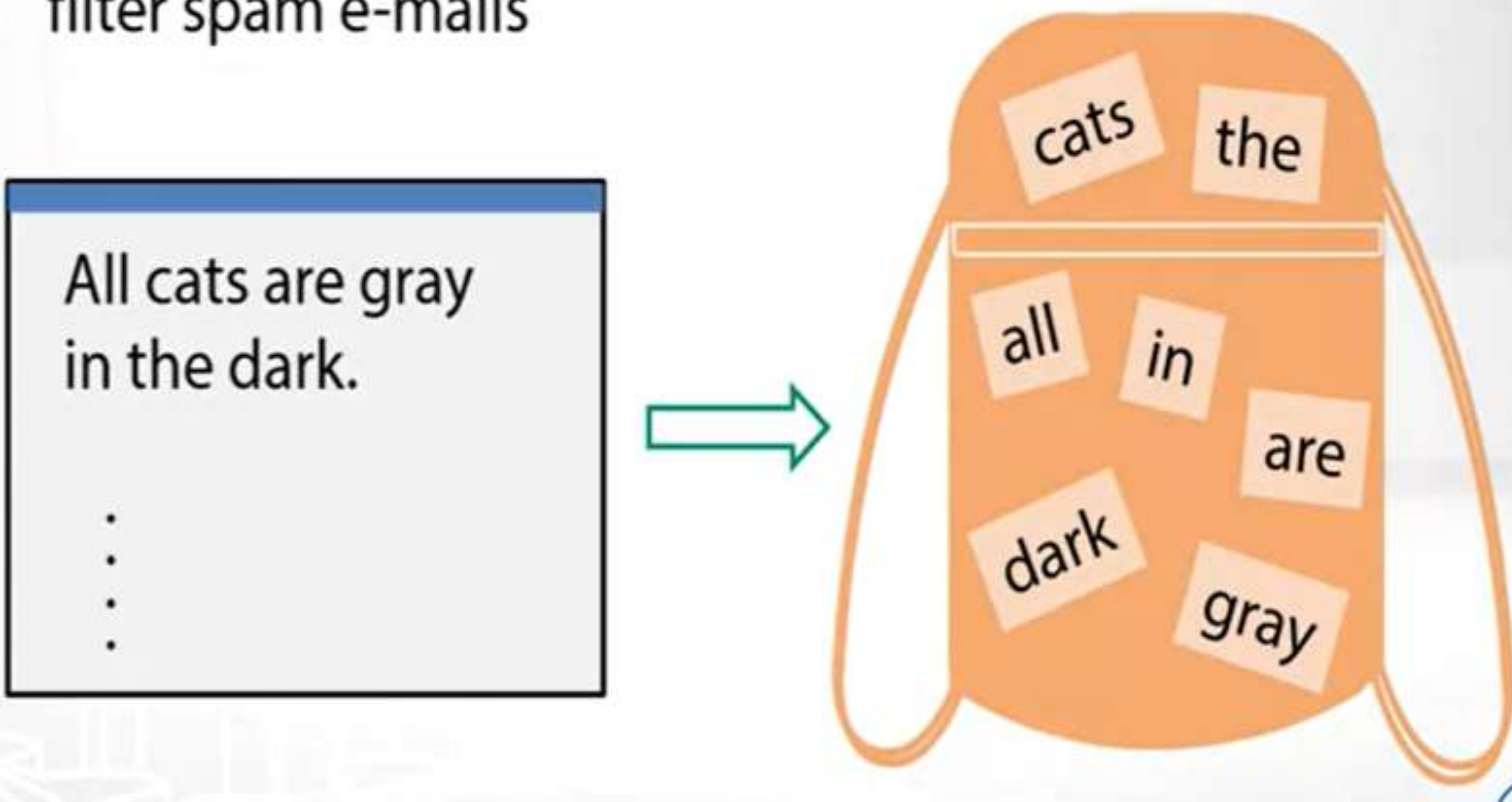
  Example: machine translation
- This is where most of research in NLP is now happening

  Example: papers from ACL, EMNLP, etc.
- Look fancy and everyone wants to know them ☺

Overview:

Text classification tasks:

- predict some tags or categories
- predict sentiment for a review
- filter spam e-mails



How to predict word sequences?

Language models are needed in chat-bots, speech recognition, machine translation, summarization…

How to predict tags for the word sequences?

- Part-of-speech tags
- Named entities
- Semantic slots

How to represent a meaning of a word, a sentence, or a text?

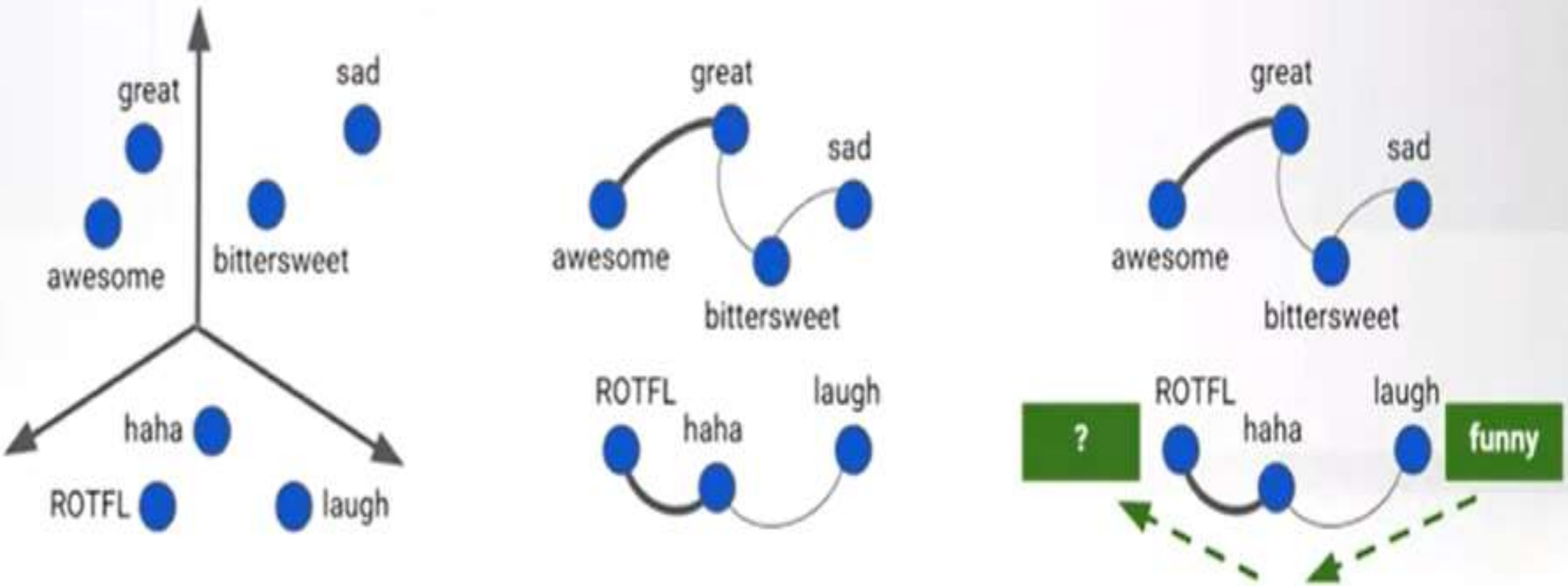
You shall know the word by the company it keeps. (Firth, 1957)

- Espresso? But I ordered a cappuccino!
- Don't worry, the cosine distance between them is so small that they are almost the same thing.

- Word embeddings
- Sentence embeddings
- Topic models

Where do we need that?

- Search, question answering, and any ranking
- Any label propagation on a word similarity graph



Word Embedding Vectors (dense, continuous space)

Word Similarity Graph

Learning Emotion Labels

Sequence to sequence tasks:

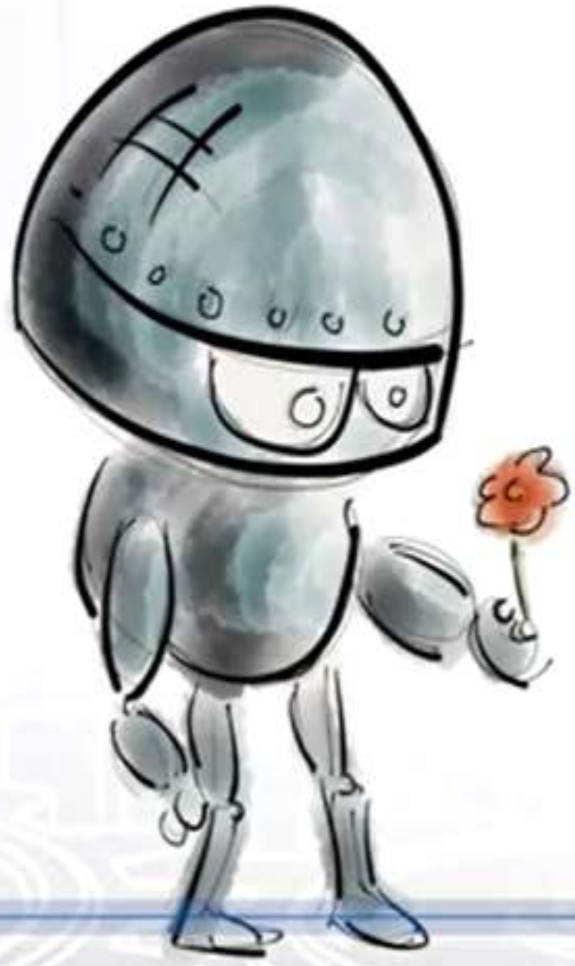
- machine translation
- summarization, simplification
- conversational chat-bot

ENCODER

DECODER

Dialogue agents become more and more popular:

- goal-oriented (e.g. help in a call-center)
- conversational (e.g. entertainment)

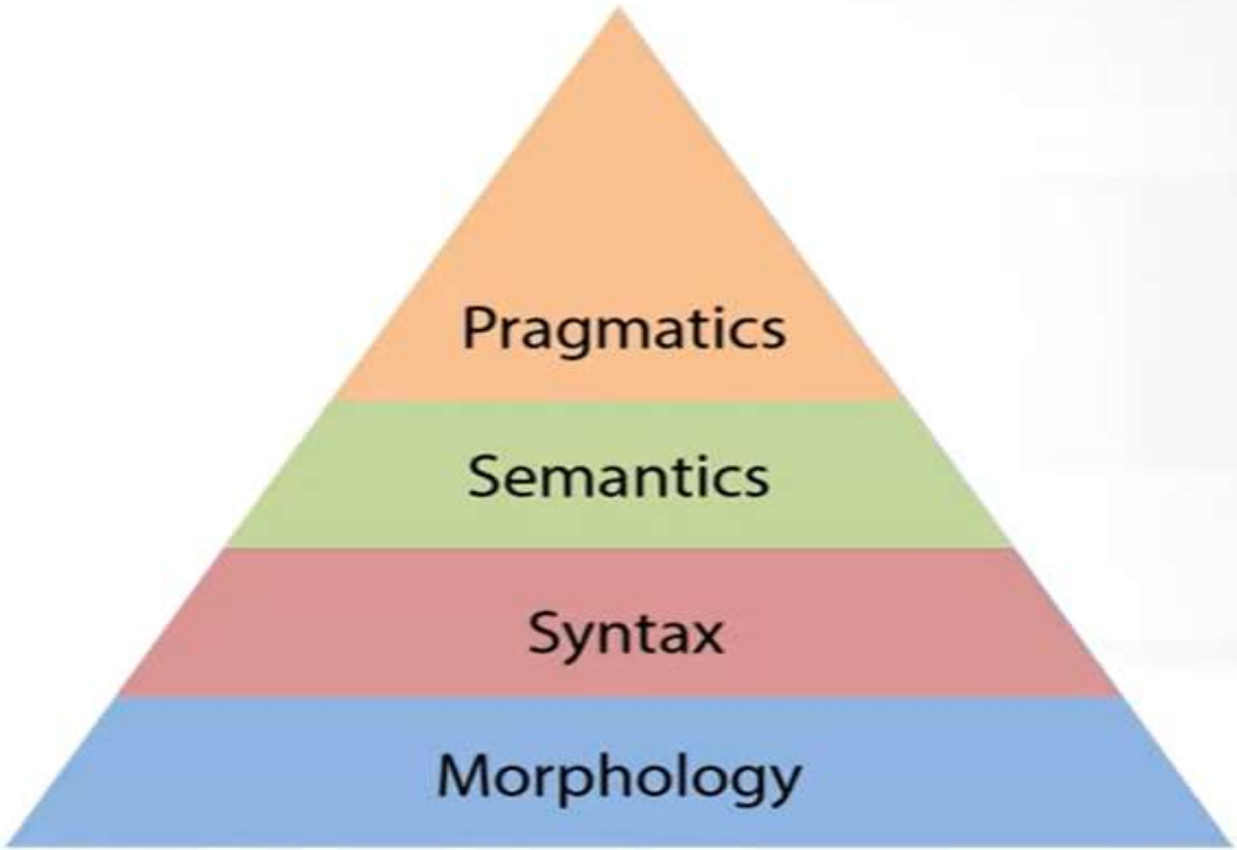


Project:

build a conversational chat-bot that assists with StackOverflow search!

LINGUISTICS INTRODUCTION:

NLP Pyramid



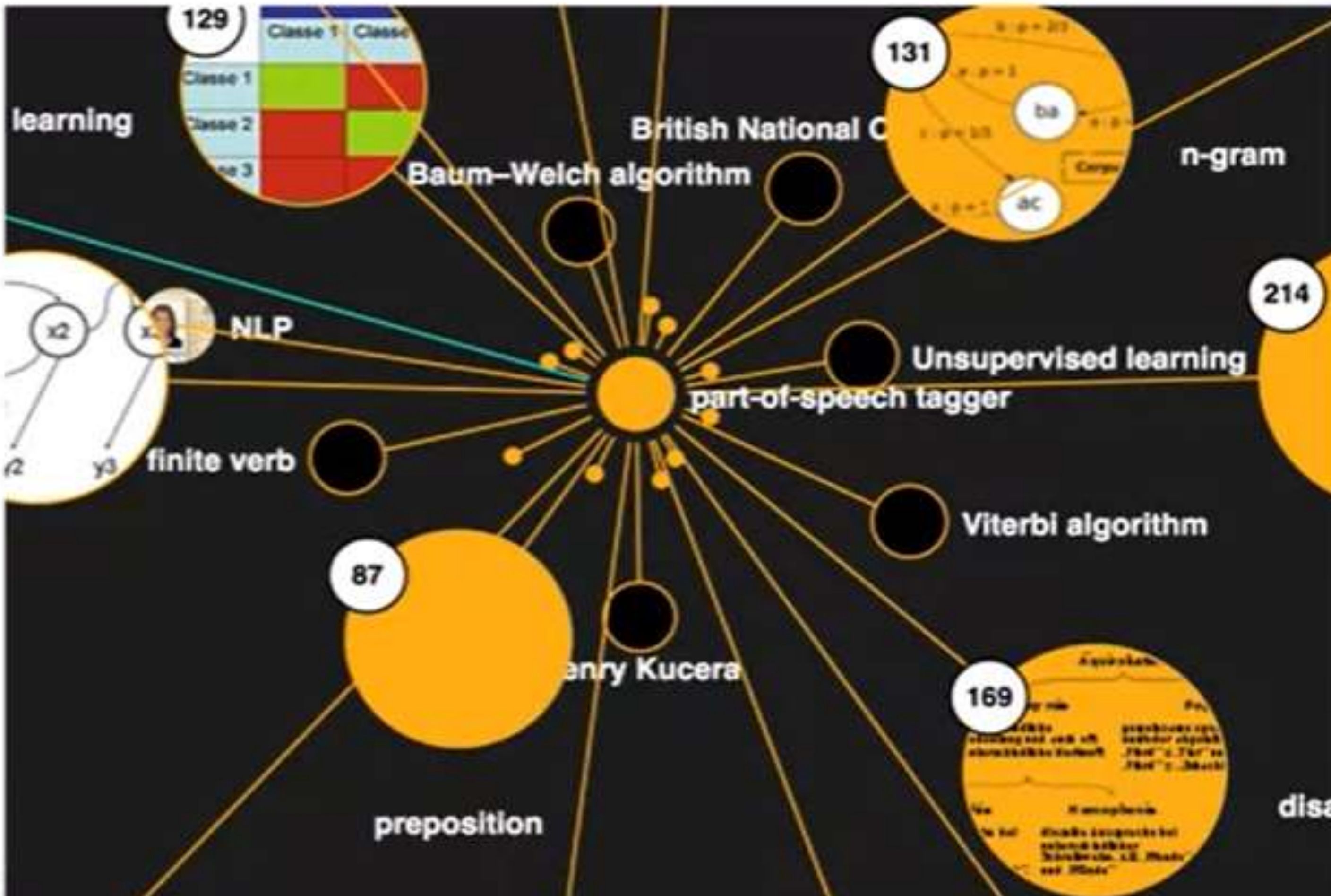
Natural Language Processing Pyramid

Libraries and tools

- **NLTK**
  - Small but useful datasets with markup
  - Preprocessing tools: tokenization, normalization…
  - Pre-trained models for POS-tagging, parsing…
- **Stanford parser**
- **spaCy**: python and cython library for NLP
- **Gensim**: python library for text analysis, e.g. for word embeddings and topic modeling
- **MALLET**: Java-based library, e.g. for classification, sequence tagging, and topic modeling

Linguistic knowledge

- Ideas and evaluation
- External resources: WordNet, BabelNet, etc.

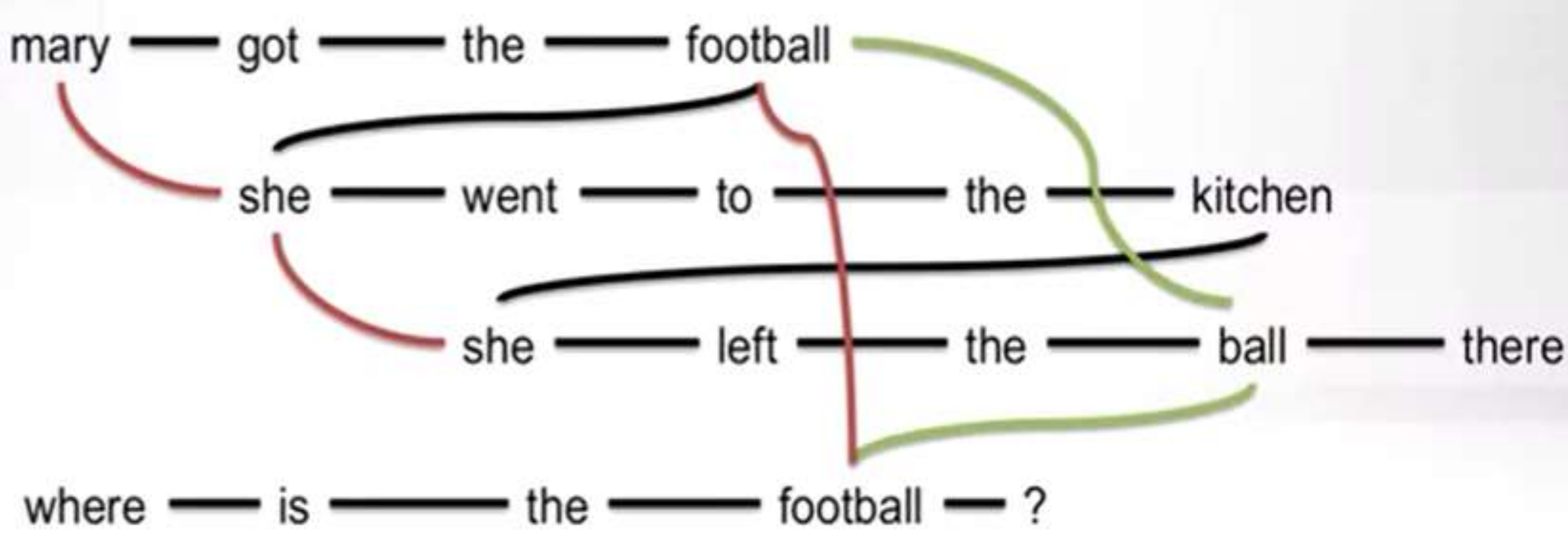


Linguistic knowledge + Deep Learning

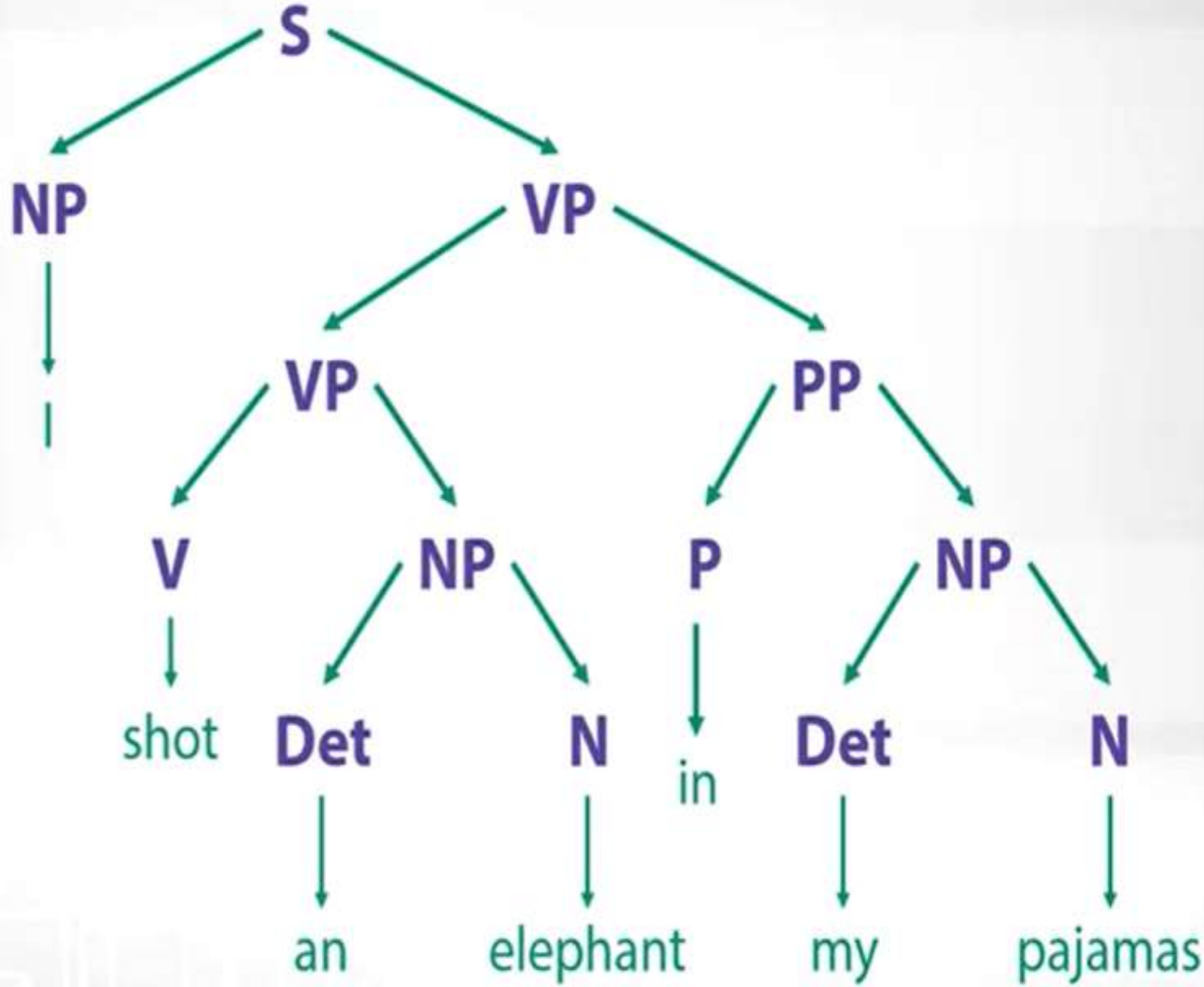
Task: Question Answering / Reasoning

Linguistic links: co-reference (red), hypernyms (green)

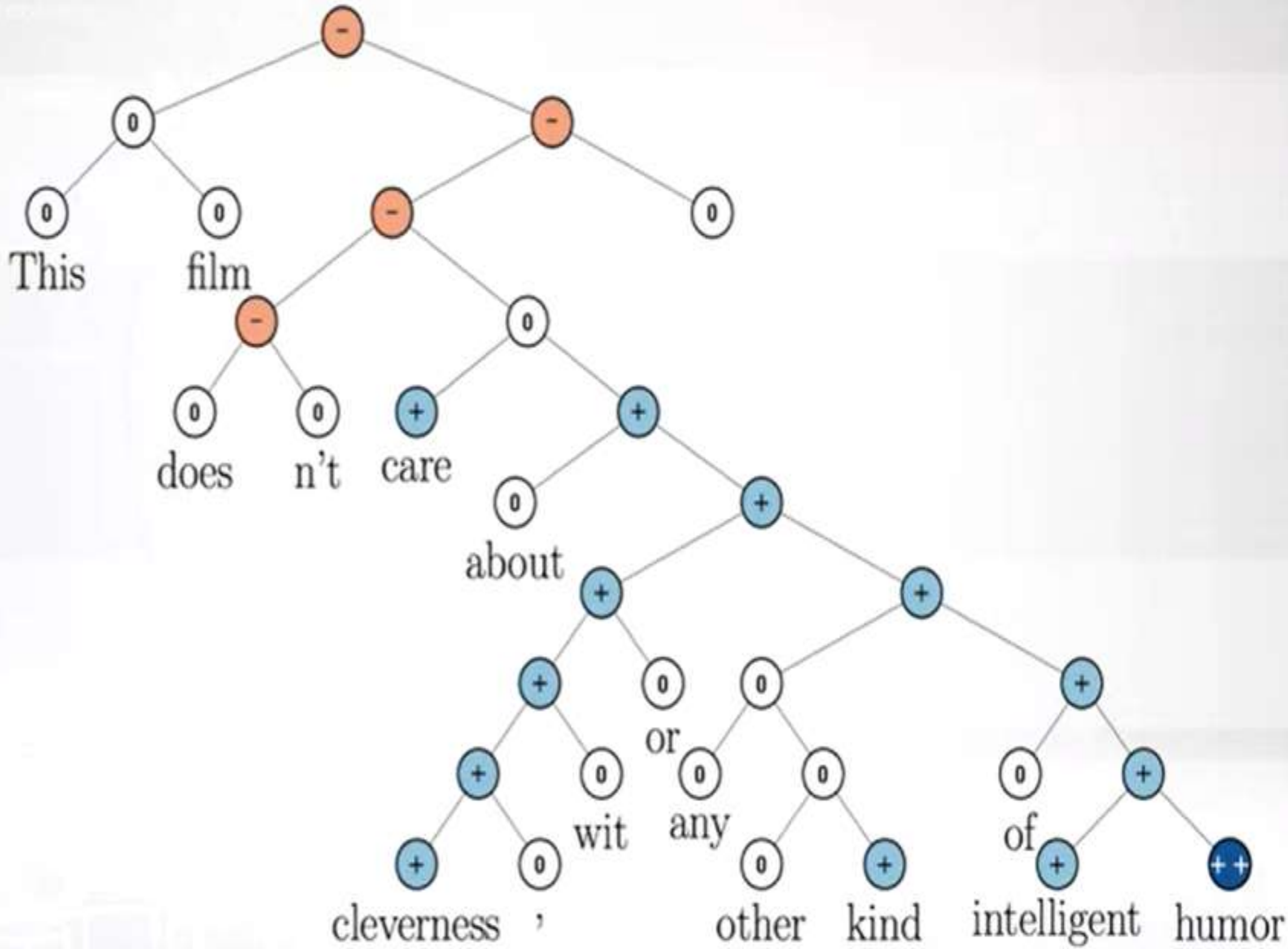
Method: DAG-LSTM



Syntax: constituency trees



Sentiment analysis



We'll focus on text classification

Example: sentiment analysis

- Input: text of review
- Output: class of sentiment
  - e.g. 2 classes: positive vs negative
- Positive example:
  - The hotel is really beautiful. Very nice and helpful service at the front desk.
- Negative example:
  - We had problems to get the Wi-Fi working. The pool area was occupied with young party animals. So the area wasn't fun for us.

TEXT PRE-PROCESSING:

What is a word?

It seems natural to think of a text as a sequence of words

- A word is a meaningful sequence of characters

How to find the boundaries of words?

- In English we can split a sentence by spaces or punctuation

Input: Friends, Romans, Countrymen, lend me your ears;

Output: Friends | Romans | Countrymen | lend | me | your | ears

- In German there are compound words which are written without spaces
  - "Rechtsschutzversicherungsgesellschaften" stands for "insurance companies which provide legal protection"
- In Japanese there are no spaces at all!
  - Butyoucanstillreaditright?

Tokenization

Tokenization is a process that splits an input sequence into so-called tokens

- You can think of a token as a useful unit for semantic processing
- Can be a word, sentence, paragraph, etc.

An example of simple whitespace tokenizer

- nltk.tokenize.WhitespaceTokenizer

This	is	Andrew's	text,	isn't	it?

- Problem: "it" and "it?" are different tokens with same meaning

Tokenization

Let's try to also split by punctuation

- nltk.tokenize.WordPunctTokenizer

This	is	Andrew	’	s	text	,	isn	’	t	it	?

- Problem: “s”, “isn”, “t” are not very meaningful

We can come up with a set of rules

- nltk.tokenize.TreebankWordTokenizer

This	is	Andrew	’s	text	,	is	n’t	it	?

- “’s” and “n’t” are more meaningful for processing

Python tokenization example

```
import nltk
text = "This is Andrew's text, isn't it?"
```

```
tokenizer = nltk.tokenize.WhitespaceTokenizer()
tokenizer.tokenize(text)
```

```
['This', 'is', "Andrew's", 'text,', "isn't", 'it?']
```

```
tokenizer = nltk.tokenize.TreebankWordTokenizer()
tokenizer.tokenize(text)
```

```
['This', 'is', 'Andrew', "'s", 'text', ',', 'is', "n't",
'it', '?']
```

```
tokenizer = nltk.tokenize.WordPunctTokenizer()
tokenizer.tokenize(text)
```

```
['This', 'is', 'Andrew', "'", 's', 'text', ',', 'isn',
 "'", 't', 'it', '?']
```

Token normalization

We may want the same token for different forms of the word

- wolf, wolves → wolf
- talk, talks → talk

Stemming

- A process of removing and replacing suffixes to get to the root form of the word, which is called the **stem**
- Usually refers to heuristics that chop off suffixes

Lemmatization

- Usually refers to doing things properly with the use of a vocabulary and morphological analysis
- Returns the base or dictionary form of a word, which is known as the **lemma**

Stemming example

Porter's stemmer

- 5 heuristic phases of word reductions, applied sequentially
- Example of phase 1 rules:

Rule	Example
SSES → SS	caresses → caress
IES → I	ponies → poni
SS → SS	caress → caress
S →	cats → cat

- nltk.stem.PorterStemmer
- Examples:
  - feet → feet　　cats → cat
  - wolves → wolv　　talked → talk
- Problem: fails on irregular forms, produces non-words

Lemmatization example

WordNet lemmatizer

- Uses the WordNet Database to lookup lemmas
- nltk.stem.WordNetLemmatizer
- Examples:
  - feet → foot   cats → cat
  - wolves → wolf   talked → talked
- Problems: not all forms are reduced

- Takeaway: we need to try stemming or lemmatization and choose best for our task

Python stemming example

```
import nltk
text = "feet cats wolves talked"
tokenizer = nltk.tokenize.TreebankWordTokenizer()
tokens = tokenizer.tokenize(text)
```

```
stemmer = nltk.stem.PorterStemmer()
" ".join(stemmer.stem(token) for token in tokens)
```

```
u'feet cat wolv talk'
```

```
stemmer = nltk.stem.WordNetLemmatizer()
" ".join(stemmer.lemmatize(token) for token in tokens)
```

```
u'foot cat wolf talked'
```

Further normalization

Normalizing capital letters

- Us, us → us (if both are pronoun)
- us, US (could be pronoun and country)
- We can use heuristics:
  - lowercasing the beginning of the sentence
  - lowercasing words in titles
  - leave mid-sentence words as they are
- Or we can use machine learning to retrieve true casing → hard

Acronyms

- eta, e.t.a., E.T.A. → E.T.A.
- We can write a bunch of regular expressions → hard

Summary

- We can think of text as a sequence of tokens
- Tokenization is a process of extracting those tokens
- We can normalize tokens using stemming or lemmatization
- We can also normalize casing and acronyms
- In the next video we will transform extracted tokens into features for our model

FEATURE EXTRACTION FROM TEXTS:

Bag of words (BOW)

Let's count occurrences of a particular token in our text

- Motivation: we're looking for marker words like "excellent" or "disappointed"
- For each token we will have a feature column, this is called **text vectorization**.

	good	movie	not	a	did	like
good movie	1	1	0	0	0	0
not a good movie	1	1	1	1	0	0
did not like	0	0	1	0	1	1

- Problems:
  - we loose word order, hence the name "bag of words"
  - counters are not normalized

Let's preserve some ordering

We can count token pairs, triplets, etc.

- Also known as n-grams
  - 1-grams for tokens
  - 2-grams for token pairs
  - …

good movie
not a good movie
did not like

good movie	movie	did not	a	…
1	**1**	0	0	…
1	**1**	0	**1**	…
0	0	**1**	0	…

- Problems:
  - too many features

Remove some n-grams

Let's remove some n-grams from features based on their occurrence frequency in documents of our corpus

- **High frequency n-grams:**
  - Articles, prepositions, etc. (example: and, a, the)
  - They are called **stop-words**, they won't help us to discriminate texts → remove them
- **Low frequency n-grams:**
  - Typos, rare n-grams
  - We don't need them either, otherwise we will likely overfit
- **Medium frequency n-grams:**
  - Those are good n-grams

There're a lot of medium frequency n-grams

- It proved to be useful to look at n-gram frequency in our corpus for filtering out bad n-grams
- What if we use it for ranking of medium frequency n-grams?
- **Idea**: the n-gram with smaller frequency can be more discriminating because it can capture a specific issue in the review

TF-IDF

Term frequency (TF)

- $\mathrm{tf}(t, d)$ – frequency for term (or n-gram) t in document d
- Variants:

weighting scheme	TF weight
binary	$0, 1$
raw count	$f_{t,d}$
term frequency	$f_{t,d} / \sum_{t' \in d} f_{t',d}$
log normalization	$1 + \log(f_{t,d})$

TF-IDF

Inverse document frequency (IDF)

- $N = |D|$ – total number of documents in corpus
- $|\{d \in D: t \in d\}|$ – number of documents where the term t appears
- $\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D: t \in d\}|}$

TF-IDF

- $\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D)$
- A high weight in TF-IDF is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents

Better BOW

- Replace counters with TF-IDF
- Normalize the result row-wise (divide by L_2-norm)

good movie
not a good movie
did not like

good movie	movie	did not	...
0.17	**0.17**	0	...
0.17	**0.17**	0	...
0	0	**0.47**	...

Python TF-IDF example

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
texts = [
    "good movie", "not a good movie", "did not like",
    "i like it", "good one"
]
tfidf = TfidfVectorizer(min_df=2, max_df=0.5, ngram_range=(1, 2))
features = tfidf.fit_transform(texts)
pd.DataFrame(
    features.todense(),
    columns=tfidf.get_feature_names()
)
```

	good movie	like	movie	not
0	0.707107	0.000000	0.707107	0.000000
1	0.577350	0.000000	0.577350	0.577350
2	0.000000	0.707107	0.000000	0.707107
3	0.000000	1.000000	0.000000	0.000000
4	0.000000	0.000000	0.000000	0.000000

LINEAR MODEL FOR TEXT CLASSIFICATION:

Sentiment classification

IMDB movie reviews dataset

- http://ai.stanford.edu/~amaas/data/sentiment/
- Contains 25000 positive and 25000 negative reviews

- Contains at most 30 reviews per movie
- At least 7 stars out of 10 → positive (label = 1)
- At most 4 stars out of 10 → negative (label = 0)
- 50/50 train/test split
- Evaluation: accuracy

Sentiment classification

Features: bag of 1-grams with TF-IDF values

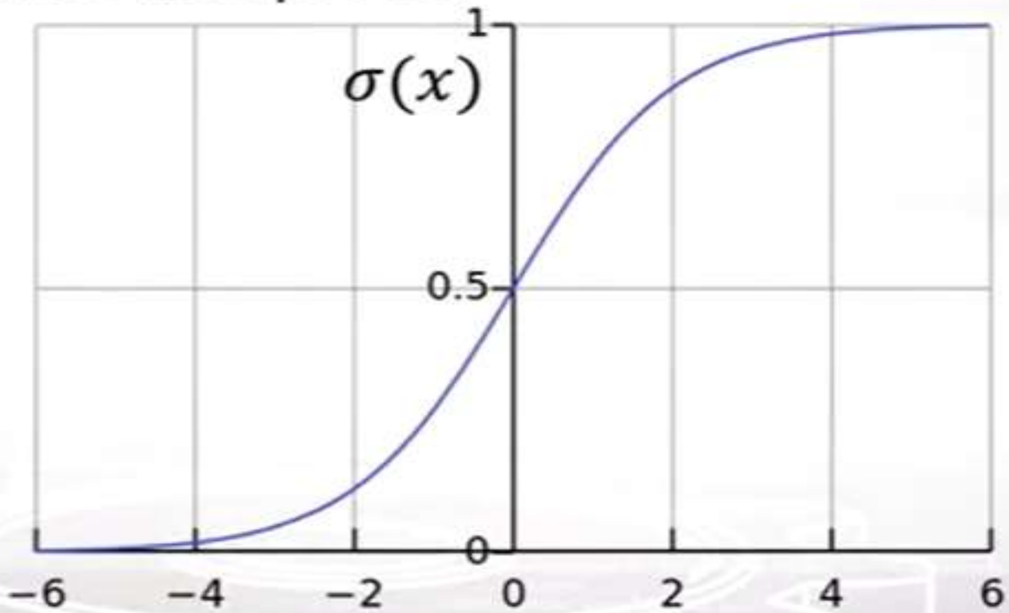
- 25000 rows, 74849 columns for training
- Extremely sparse feature matrix – 99.8% are zeros

acting	actingjob	actings	actingwise
0.000000	0.0	0.0	0.0
0.000000	0.0	0.0	0.0
0.053504	0.0	0.0	0.0
0.033293	0.0	0.0	0.0
0.000000	0.0	0.0	0.0

Sentiment classification

Model: Logistic regression

- $p(y = 1|x) = \sigma(w^T x)$
- Linear classification model
- Can handle sparse data
- Fast to train
- Weights can be interpreted



Sentiment classification

Logistic regression over bag of 1-grams with TF-IDF

- Accuracy on test set: 88.5%
- Let's look at learnt weights:

ngram	weight
great	9.042803
excellent	8.487379
perfect	6.907277
best	6.440972
wonderful	6.237365

Top positive

vs

ngram	weight
worst	-12.748257
awful	-9.150810
bad	-8.974974
waste	-8.944854
boring	-8.340877

Top negative

Better sentiment classification

Let's try to add 2-grams

- Throw away n-grams seen less than 5 times
- 25000 rows, 156821 columns for training

and am	and amanda	and amateur	and amateurish	and amazing
0.068255	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0
0.000000	0.0	0.0	0.0	0.0

Better sentiment classification

Logistic regression over bag of 1,2-grams with TF-IDF

- Accuracy on test set: 89.9% (+1.5%)
- Let's look at learnt weights:

well worth	13.788515		bad	-24.467648
best	13.633200		poor	-24.319746
rare	13.570259	VS	the worst	-23.773352
better than	13.500025		waste	-22.880340

Near top positive — Near top negative

How to make it even better

Play around with tokenization

- Special tokens like emoji, ":)" and "!!!" can help

Try to normalize tokens

- Adding stemming or lemmatization

Try different models

- SVM, Naïve Bayes, …

Throw BOW away and use Deep Learning

- https://arxiv.org/pdf/1512.08183.pdf
- Accuracy on test set in 2016: 92.14% (+2.5%)

Summary

- Bag of words and simple linear models actually work for texts
- The accuracy gain from deep learning models is not mind blowing for sentiment classification

SPAM FILTERING:

Mapping n-grams to feature indices

If your dataset is small you can store {n-gram → feature index} in hash map.

But if you have a huge dataset that can be a problem

- Let's say we have 1 TB of texts distributed on 10 computers
- You need to vectorize each text
- You will have to maintain {n-gram → feature index} mapping
  - May not fit in memory on one machine
  - Hard to synchronize
- An easier way is hashing: {n-gram → hash(n-gram) % 2^{20}}
  - Has collisions but works in practice
  - sklearn.feature_extraction.text.**HashingVectorizer**
  - Implemented in **vowpal wabbit** library

Spam filtering is a huge task

Spam filtering proprietary dataset

- https://arxiv.org/pdf/0902.2206.pdf
- 0.4 million users
- 3.2 million letters
- 40 million unique words

Let's say we map each token to index using hash function ϕ

- $\phi(x) = \text{hash}(x) \; \% \; 2^b$
- For $b = 22$ we have 4 million features
- That is a huge improvement over 40 million features
- It turns out it doesn't hurt the quality of the model

Hashing example

- $\phi(good) = 0$
- $\phi(movie) = 1$
- $\phi(not) = 2$
- $\boldsymbol{\phi(a) = 3}$
- $\boldsymbol{\phi(did) = 3}$
- $\phi(like) = 4$

Hash collision

$$\mathbf{hash}(s) = s[0] + s[1]p^1 + \cdots + s[n]p^n$$

s – string
p – fixed prime number
$s[i]$ – character code

good movie
not a good movie
did not like

0	1	2	3	4
1	1	0	0	0
1	1	1	1	0
0	0	1	1	1

Trillion features with hashing

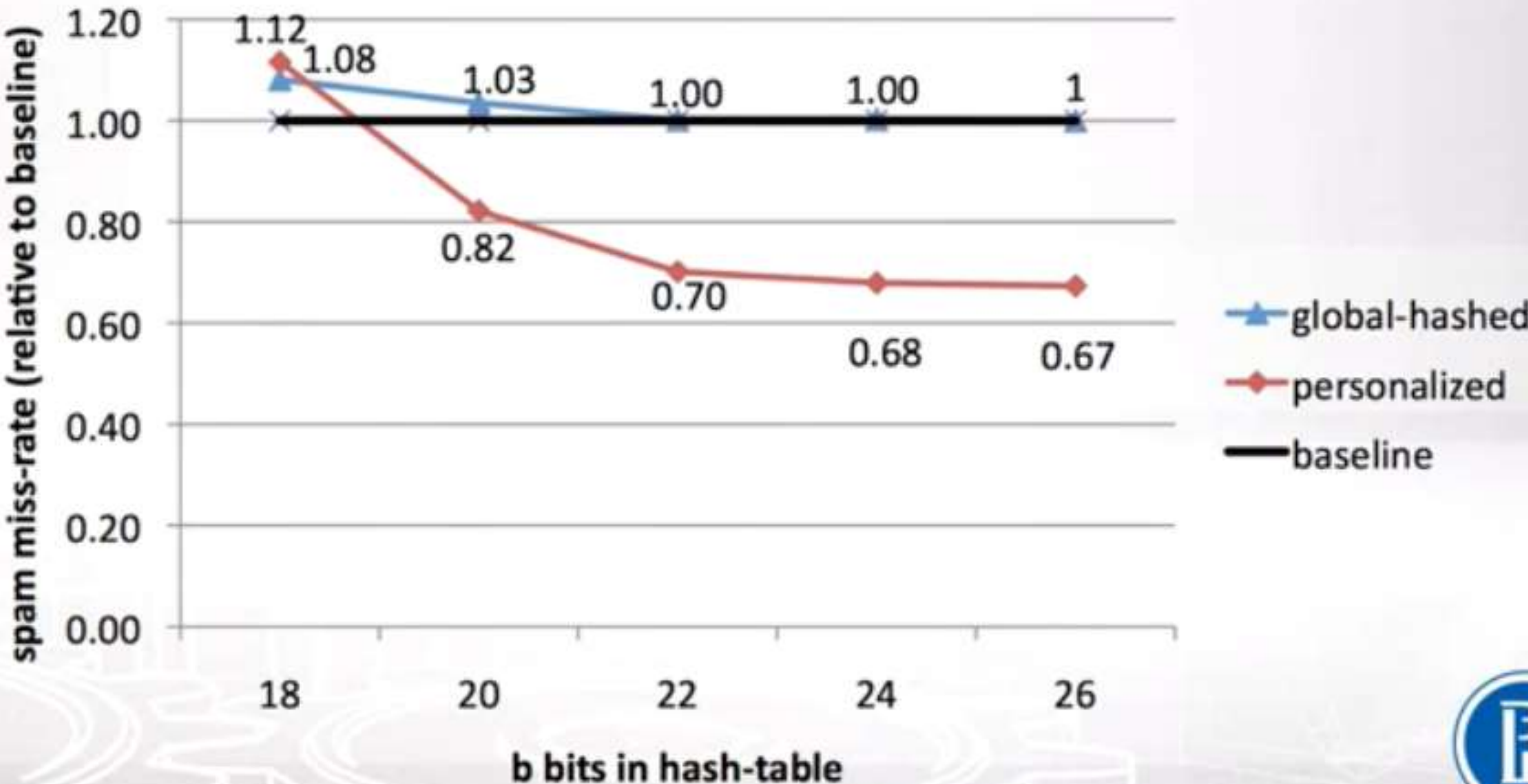
Personalized tokens trick

- $\phi_o(token) = \text{hash}(token) \,\%\, 2^b$
- $\phi_u(token) = \text{hash}(u + \text{"\_"} + token) \,\%\, 2^b$
- We obtain 16 trillion pairs (user, word) but still 2^b features

text document (email)		bag of words		bag of words (personalized)		$\phi_0(x) + \phi_u(x)$
U: **Votre Apotheke en li** -10 pilu x 100 mg + **Cila** raison gratuite tème de commande sûr	→	NEU Votre Apotheke …	→	NEU **USER123**_NEU Votre **USER123**_Votre Apotheke **USER123**_Apotheke …	→	1 0 -1 0 0 -1 0 1

Experimental results

- For $b = 22$ it performs just like a linear model on original tokens
- We observe that personalized tokens give a huge improvement in miss-rate!

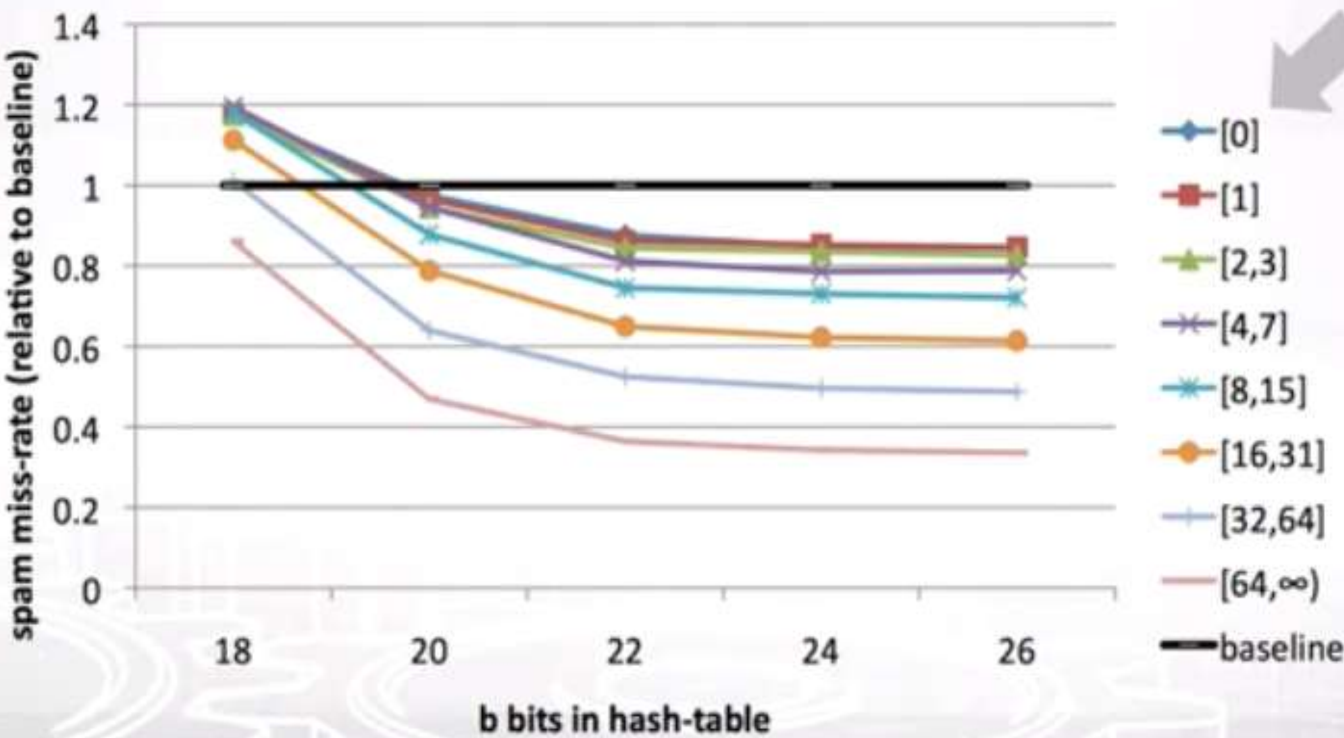


Why personalized features work

Personalized features capture "local" user-specific preference

- Some users might consider newsletters a spam but for the majority of the people they are fine

How will it work for new users?



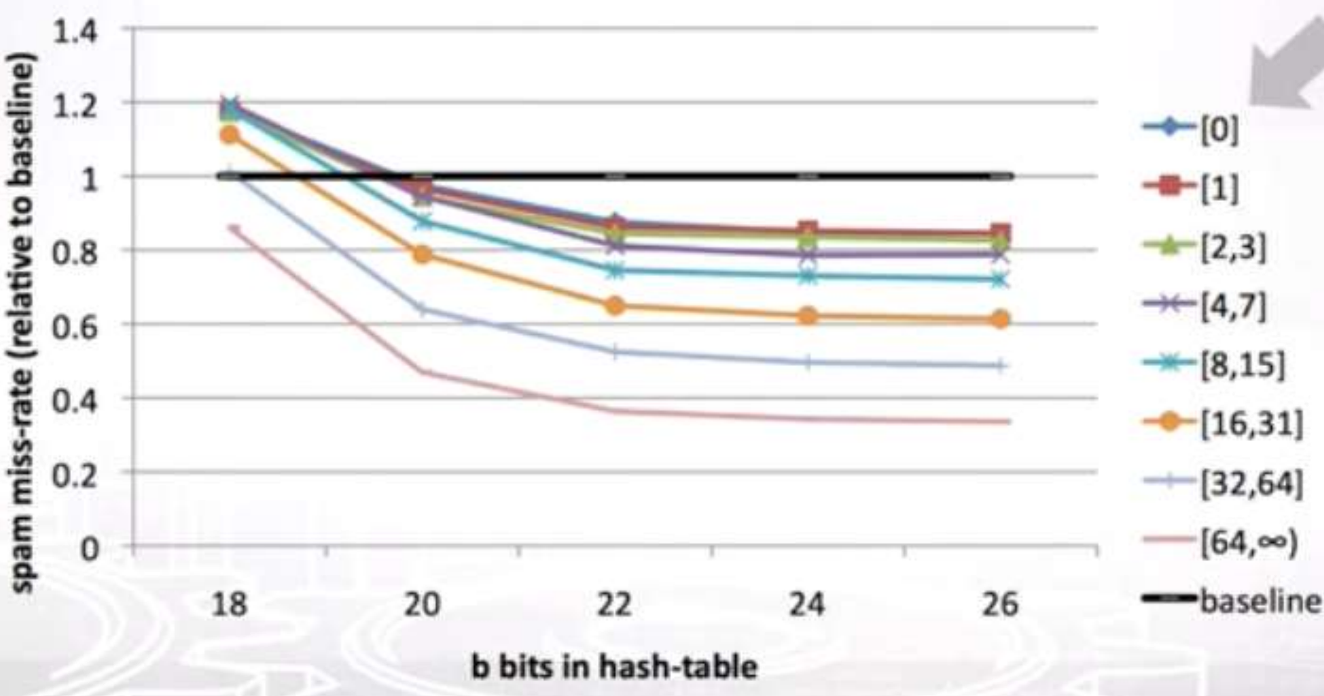
What?

Number of emails in training

Why personalized features work

It turns out we learn better "global" preference having personalized features which learn "local" user preference

- You can think of it as a more universal definition of spam



Number of emails in training

Why the size matters

Why do we need such huge datasets?

- It turns out you can learn better models using the same simple linear classifier

Ad click prediction

- https://arxiv.org/pdf/1110.4198.pdf
- Trillions of features, billions of training examples
- Data sampling hurts the model

	1%	10%	100%
auROC	0.8178	0.8301	0.8344
auPRC	0.4505	0.4753	0.4856
NLL	0.2654	0.2582	0.2554

Sampling rate

Vowpal Wabbit

- A popular machine learning library for training linear models
- Uses feature hashing internally
- Has lots of features
- Really fast and scales well

VOWPAL WABBIT

Format: label | sparse features ...
1 | 13:3.9656971e-02 24:3.4781646e-02 ...
which corresponds to:
1 | tuesday year ...
command: time vw –sgd rcv1.train.txt -c

Summary

- We've taken a look on applications of feature hashing
- Personalized features is a nice trick
- Linear models over bag of words scale well for production

Bag of words way (sparse)

~100k columns

	good	movie	very	a	did	like
very	0	0	**1**	0	0	0
+						
good	**1**	0	0	0	0	0
+						
movie	0	**1**	0	0	0	0
=						
very good movie	**1**	**1**	**1**	0	0	0

Bag of words representation
is a sum of sparse one-hot-encoded vectors

Neural way (dense)

~300 columns

	x_1	x_2	x_3
very	**0.7**	**0.4**	**0.5**
good	**0.2**	**-0.1**	**0.1**
movie	**-0.5**	**0.4**	**0.1**

Example:
word2vec
embeddings

Word2vec property:
Words that have similar context tend to have collinear vectors

Neural way (dense)

~300 columns

	x_1	x_2	x_3
very	0.7	0.4	0.5
+			
good	0.2	-0.1	0.1
+			
movie	-0.5	0.4	0.1
=			
very good movie	0.4	0.7	0.7

Example: word2vec embeddings

Sum of word2vec vectors
can be a good text descriptor already!

A better way: 1D convolutions

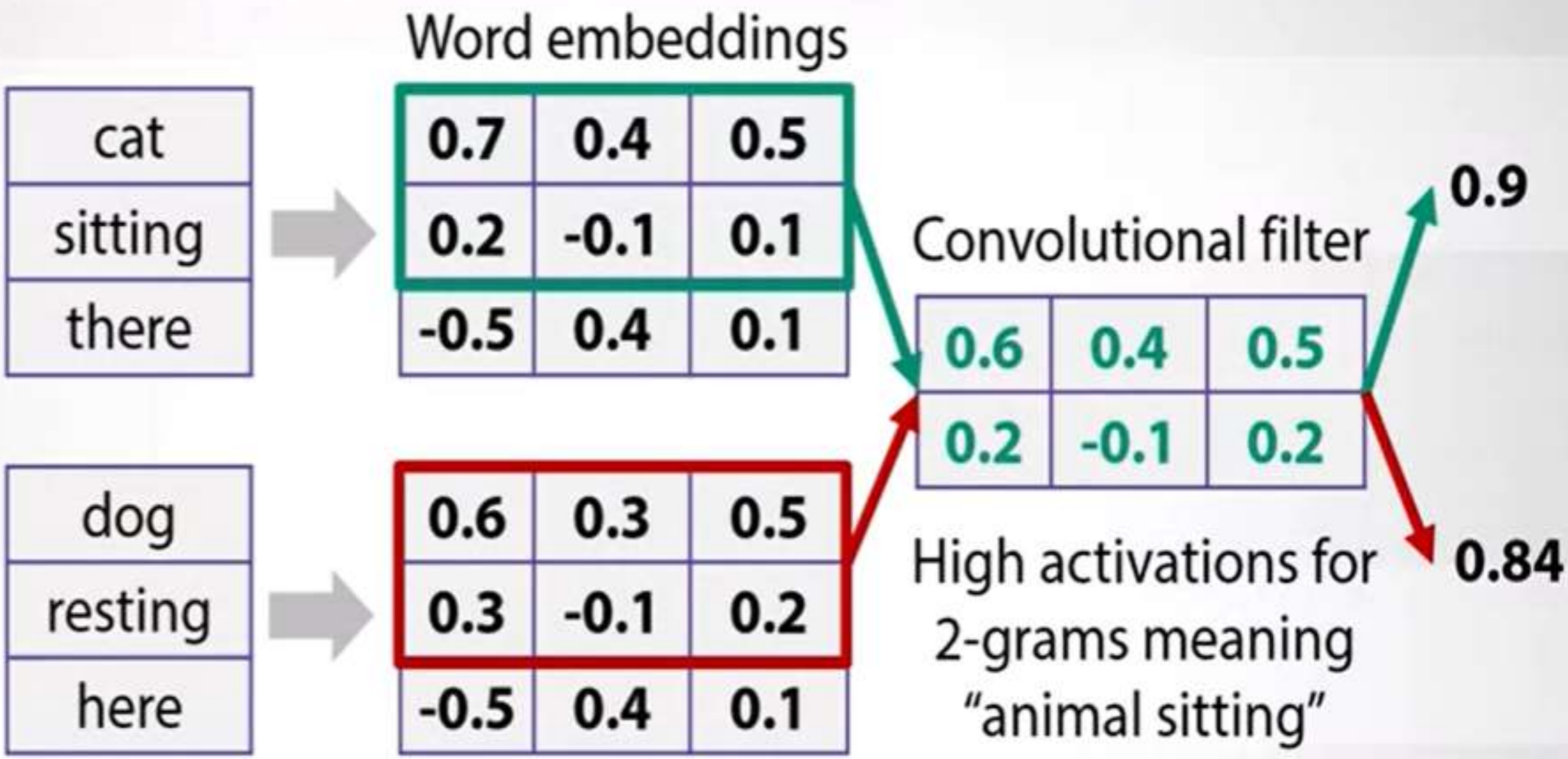
Word embeddings

cat	0.7	0.4	0.5
sitting	0.2	-0.1	0.1
there	-0.5	0.4	0.1

How do we make 2-grams?

dog	0.6	0.3	0.5
resting	0.3	-0.1	0.2
here	-0.5	0.4	0.1

A better way: 1D convolutions



- This convolution provides high activations for 2-grams with certain meaning
- Word2vec vectors for similar words are similar in terms of cosine distance (similar to dot product)

http://bionlp-www.utu.fi/wv_demo/

1D convolutions

- Can be extended to 3-grams, 4-grams, etc.
- One filter is not enough, need to track many n-grams
- They are called 1D because we slide the window only in one direction

cat	**0.7**	**0.4**	**0.5**	0.1
sitting	**0.2**	**-0.1**	**0.1**	0.3
there	**-0.5**	**0.4**	**0.1**	-0.2
or	**-0.1**	**0.8**	**-0.3**	**0.7**
here	**-0.5**	**0.3**	**0.2**	-0.4

What to do with this vector?

cat	**0.7**	**0.4**	**0.5**
sitting	**0.2**	**-0.1**	**0.1**
there	**-0.5**	**0.4**	**0.1**
or	**-0.1**	**0.8**	**-0.3**
here	**-0.5**	**0.3**	**0.2**

0.7

Maximum pooling over time!

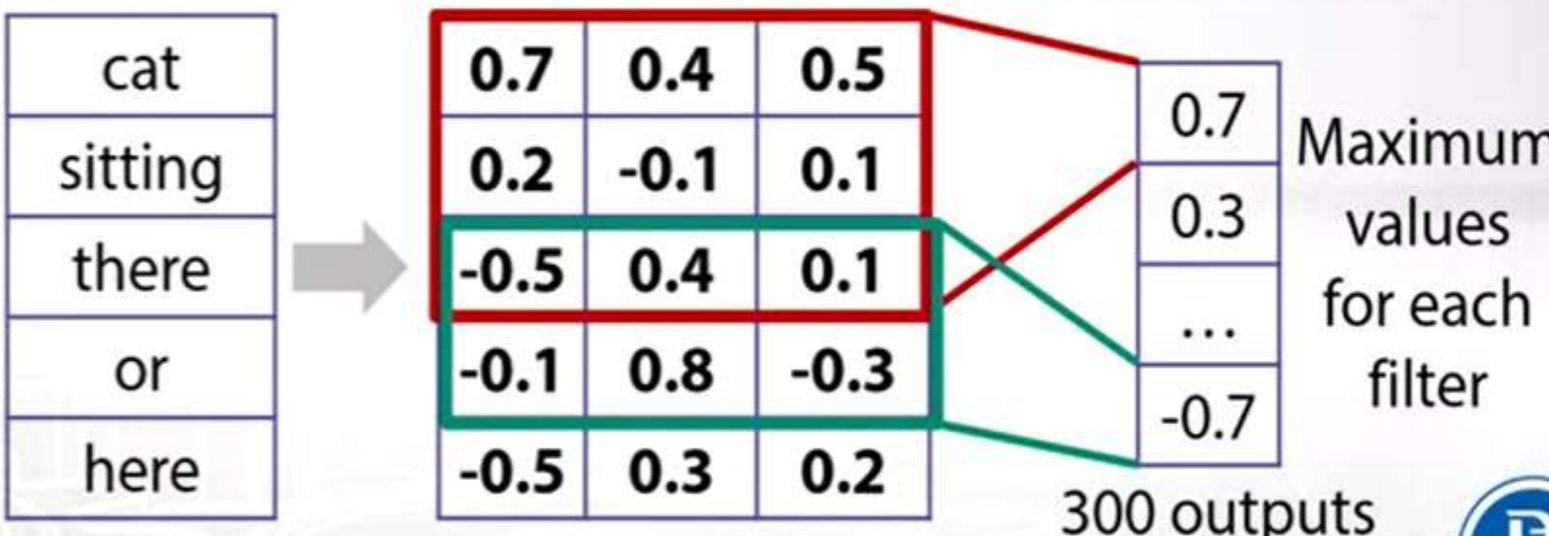
Let's train many filters

Final architecture

- 3,4,5-gram windows with 100 filters each
- MLP on top of these 300 features

Quality comparison on customer reviews (CR)

- Naïve Bayes on top of 1,2-grams – 86.3% accuracy
- 1D convolutions with MLP – 89.6% (+3.8%) accuracy



http://www.jmlr.org/papers/volume12/collobert11a/collobert11a.pdf https://arxiv.org/pdf/1408.5882.pdf

Text as a sequence of characters

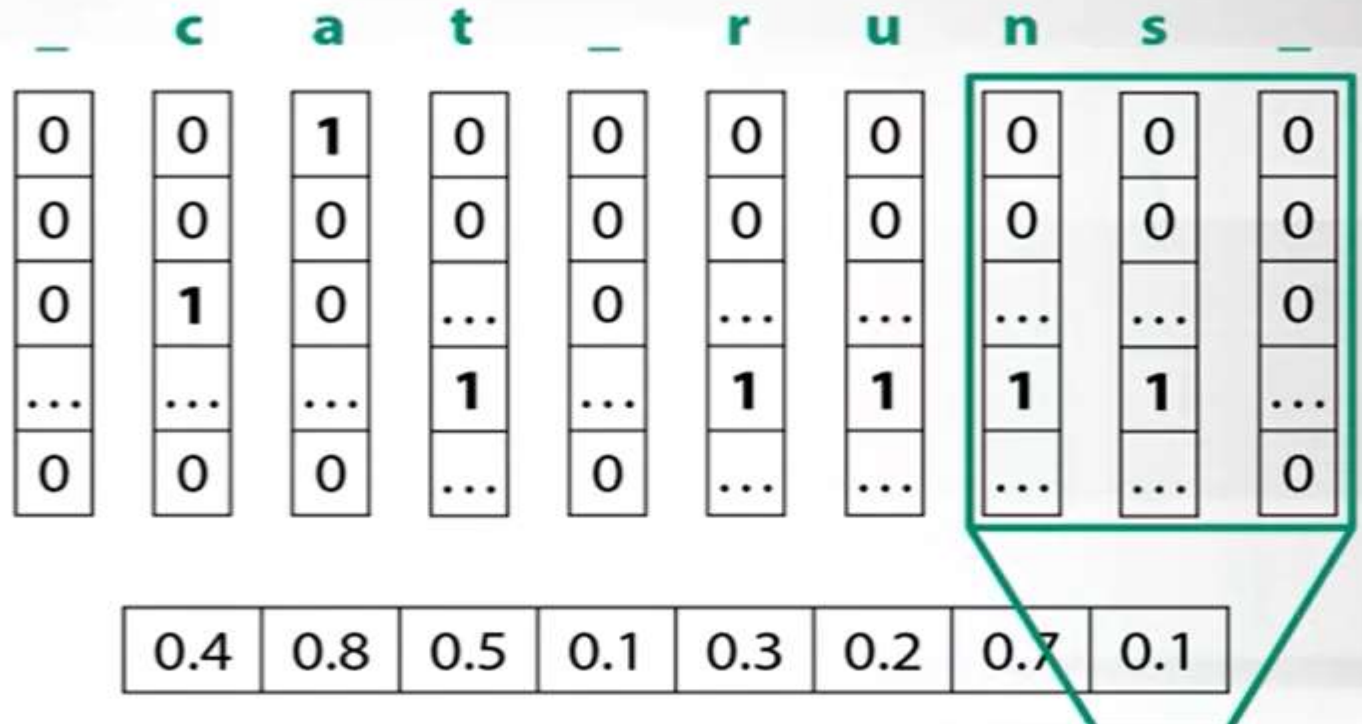
One-hot encoded characters, length ~70

_	c	a	t	_	r	u	n	s	_
0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	1	0	…	0	…	…	…	…	0
…	…	…	1	…	1	1	1	1	…
0	0	0	…	0	…	…	…	…	0

Let’s start with character *n*-grams!

1D convolutions on characters

One-hot encoded characters, length ~70



Filter #1	0.4	0.8	0.5	0.1	0.3	0.2	0.7	0.1
Filter #2	0.5	0.1	0.4	0.2	0.3	0.9	0.1	0.8

Filter #1	0.4	0.8	0.5	0.1	0.3	0.2	0.7	0.1
Filter #2	0.5	0.1	0.4	0.2	0.3	0.9	0.1	0.8
Filter #3	0.4	0.7	0.3	0.7	0.5	0.5	0.9	0.4

~1024 filters

What's next? Let's add pooling!

Max pooling

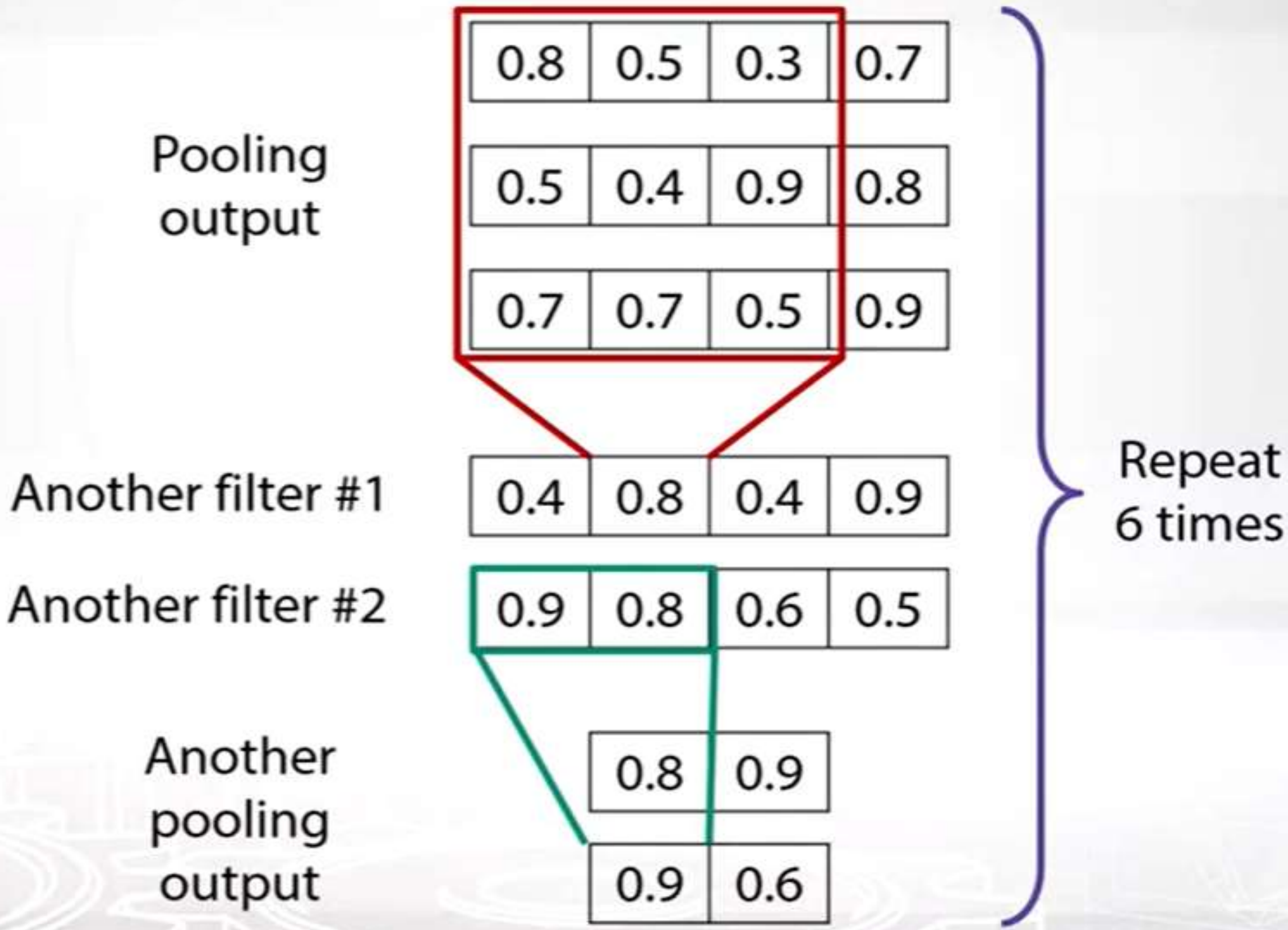
Filter #1	0.4	0.8	0.5	0.1	0.3	0.2	0.7	0.1
Filter #2	0.5	0.1	0.4	0.2	0.3	0.9	0.1	0.8
Filter #3	0.4	0.7	0.3	0.7	0.5	0.5	0.9	0.4

Pooling output

0.8	0.5	0.3	0.7
0.5	0.4	0.9	0.8
0.7	0.7	0.5	0.9

Provides a little bit of position invariance for character n-grams

Repeat 1D convolution + pooling



Final architecture

- Let's take only first **1014** characters of text
- Apply 1D convolution + max pooling **6** times
  - Kernels widths: 7, 7, 3, 3, 3, 3
  - Filters at each step: 1024
- After that we have a $\mathbf{1024} \times \mathbf{34}$ matrix of features
- Apply MLP for your task

Experimental datasets

Categorization or sentiment analysis

	Dataset	Classes	Train Samples
Smaller	AG's News	4	120,000
	Sogou News	5	450,000
	DBPedia	14	560,000
	Yelp Review Polarity	2	560,000
Bigger	Yelp Review Full	5	650,000
	Yahoo! Answers	10	1,400,000
	Amazon Review Full	5	3,000,000
	Amazon Review Polarity	2	3,600,000

Experimental results

Errors on test set for classical models:

Model	AG	Sogou	DBP.	Yelp P.	Yelp F.	Yah. A.	Amz. F.	Amz. P.
BoW	11.19	7.15	3.39	7.76	42.01	31.11	45.36	9.60
BoW TFIDF	10.36	6.55	2.63	6.34	40.14	28.96	44.74	9.00
ngrams	7.96	2.92	1.37	**4.36**	43.74	31.53	45.73	7.98
ngrams TFIDF	**7.64**	**2.81**	**1.31**	4.56	45.20	31.49	47.56	8.46

Errors on test set for deep models:

LSTM	13.94	4.82	1.45	5.26	41.83	29.16	40.57	6.10
Sm. Full Conv.	11.59	8.95	1.89	5.67	38.82	30.01	40.88	5.78
Lg. Full Conv. Th.	9.51	-	1.55	4.88	38.04	29.58	40.54	5.51
Sm. Full Conv. Th.	10.89	-	1.69	5.42	**37.95**	29.90	40.53	5.66
Lg. Conv.	12.82	4.88	1.73	5.89	39.62	29.55	41.31	5.51
Sm. Conv.	15.65	8.65	1.98	6.53	40.84	29.84	40.53	5.50
Lg. Conv. Th.	13.39	-	1.60	5.82	39.30	**28.80**	40.45	**4.93**
Sm. Conv. Th.	14.80	-	1.85	6.49	40.16	29.84	**40.43**	5.67

Deep models work better for large datasets!

Summary

- You can use convolutional networks on top of characters (called learning from scratch)
- It works best for large datasets where it beats classical approaches (like BOW)
- Sometimes it even beats LSTM that works on word level