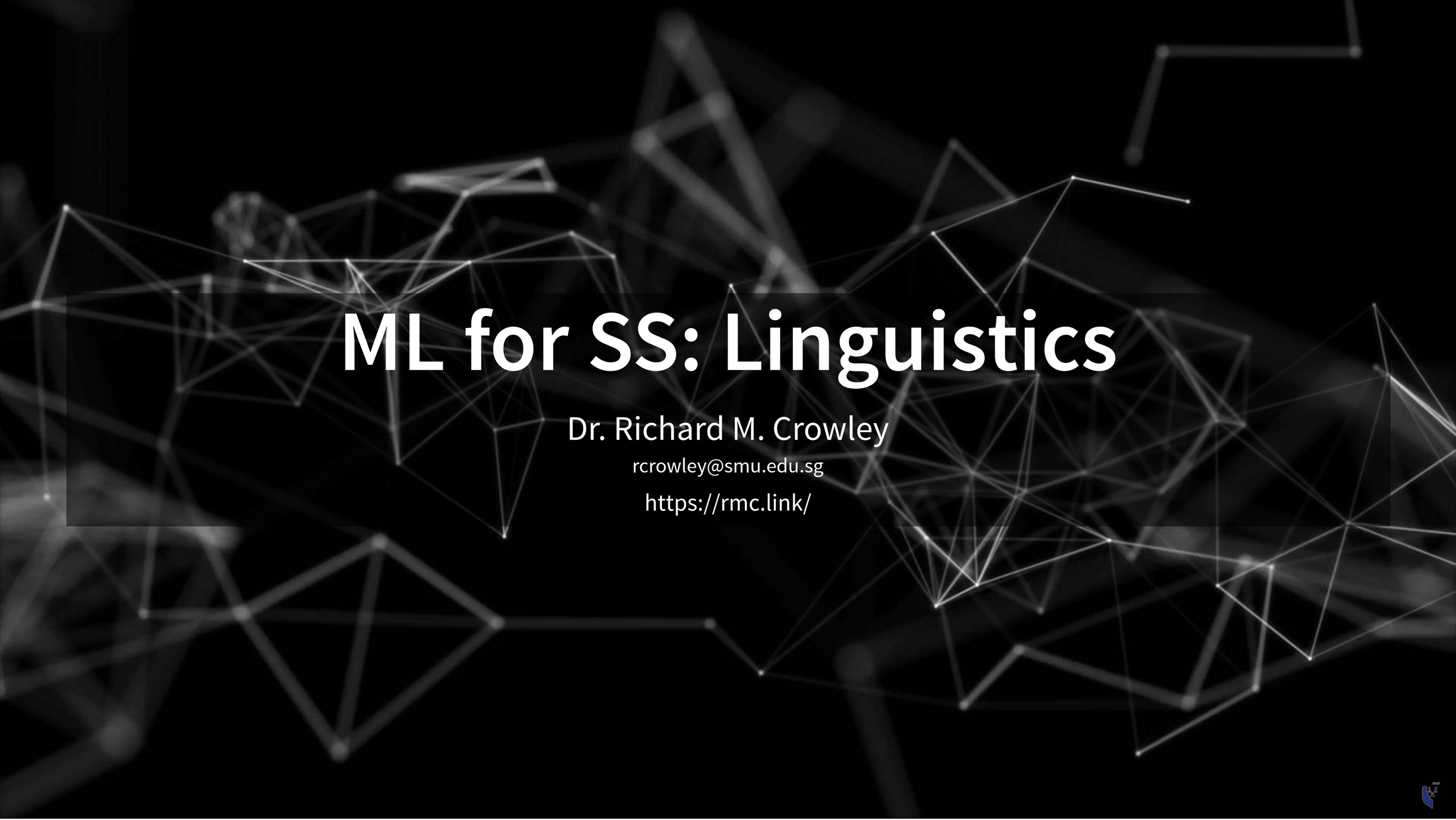
# ML for SS: Linguistics

Dr. Richard M. Crowley

rcrowley@smu.edu.sg

https://rmc.link/



# Overview

# Papers

> Harris (1954)

- Outlines the main reason for why we do most of what we do in NLP

> Garimella et al. (2019)

- Examines how social factors (gender) influence a common class of algorithms (tagging/parsing)

> Garg et al. (2018)

- Illustrates societal shifts through language by using word embeddings

# Technical Discussion: Linguistics

## Python

- NLTK for standard/statistical approaches
- SpaCy for machine learning pipelines
- Stanza for Standford NLP methods
  - They have some interesting models for narrower uses
- BeautifulSoup for HTML parsing

## R

- Call python's SpaCy package from R using `spacyr`
- `rvest` for HTML parsing

> Python is generally a bit stronger for these topics, unless your data is clean and fairly small.

> Code in python is on my website; data is on eLearn.

# Main application: Analyzing WSJ articles

- On eLearn you will find a full issue of the WSJ in text format

> Linguistic models using NLTK and SpaCy

- Tokenization and breaking documents into smaller chunks
- Part of speech tagging (grammar)
- Dependency parsing
- Named Entity Recognition (NER)
- Lemmatization

# Additional application: Word2vec models

- First we will examine the word2vec model itself
- We will then apply pre-existing word vectors to our text
- We will also train a model from scratch

**Usage of word2vec**

Many of the original uses of the model have been superseded by the models we'll talk about in sessions 9 to 11. However, if you truly need a word-level measure, it is still viable.

# Using NLP parsers: NLTK

# NLTK

- NLTK stands for Natural Language Toolkit
- It provides a bunch of handy things for text analytics
  1. Corpora that are used in research and algorithm development
     - Tagged corpora are particularly valuable
  2. Models for things like dependency parsing
  3. Useful functions for working with text

# Setting up NLTK

- When using a resource from `NLTK`, we will often have install needed datasets

> Useful parts to download using `nltk.download()`

- `'punkt'`: Used for tokenizing words (splitting apart words in a document)
- `'brown'`: A corpus that contains part of speech information based on news articles
  - Can be used to train a part of speech tagger
- `'averaged_perceptron_tagger'`: An ML model for applying part of speech tags
- `'universal_tagset'`: If you only need simple part of speech labels, this is easier to work with
- `'treebank'`: Like `'brown'` above, but based on WSJ

# Tokenizing

```
1  text = 'Hedge funds are cutting their standard fees of 2% of assets under management and 20% of pro
2  tokens = nltk.tokenize.word_tokenize(text)
3  print(tokens)
```

```
['Hedge', 'funds', 'are', 'cutting', 'their', 'standard', 'fees', 'of', '2', '%', 'of', 'assets', 'under',
'management', 'and', '20', '%', 'of', 'profits', 'amid', 'pressure', 'from', 'investors', '.', 'A', 'team',
'of', 'Ares', 'and', 'CPP', 'Investment', 'Board', 'are', 'in', 'the', 'final', 'stages', 'of', 'talks', 'to',
'buy', 'luxury', 'retailer', 'Neiman', 'Marcus', 'for', 'around', '$', '6', 'billion', '.']
```

# Part of Speech tagging:

```
# Requires previously running: nltk.download('brown')
brown_news_tagged = nltk.corpus.brown.tagged_sents(categories='news', tagset='brown')
pos_tagger = nltk.UnigramTagger(brown_news_tagged)

# tokens is our tokenized text from the previous slide
tagged1 = pos_tagger.tag(tokens)
print(tagged1)
```

```
[('Hedge', None), ('funds', 'NNS'), ('are', 'BER'), ('cutting', 'VBG'), ('their', 'PP$'), ('standard', 'JJ'),
('fees', 'NNS'), ('of', 'IN'), ('2', 'CD'), ('%', None), ('of', 'IN'), ('assets', 'NNS'), ('under', 'IN'),
('management', 'NN'), ('and', 'CC'), ('20', 'CD'), ('%', None), ('of', 'IN'), ('profits', 'NNS'), ('amid',
'IN'), ('pressure', 'NN'), ('from', 'IN'), ('investors', 'NNS'), ('.', '.'), ('A', 'AT'), ('team', 'NN'),
('of', 'IN'), ('Ares', None), ('and', 'CC'), ('CPP', None), ('Investment', 'NN-TL'), ('Board', 'NN-TL'),
('are', 'BER'), ('in', 'IN'), ('the', 'AT'), ('final', 'JJ'), ('stages', 'NNS'), ('of', 'IN'), ('talks',
'NNS'), ('to', 'TO'), ('buy', 'VB'), ('luxury', 'NN'), ('retailer', None), ('Neiman', None), ('Marcus', 'NP'),
('for', 'IN'), ('around', 'RB'), ('$', None), ('6', 'CD'), ('billion', 'CD'), ('.', '.')]
```

> Unigram tagging just uses the most common POS for a given token

# Part of Speech tagging: Neural network model

```
1  text = 'Hedge funds are cutting their standard fees of 2% of assets under management and 20% of pro
2  tokens = nltk.tokenize.word_tokenize(text)
3
4  # Requires previously running: nltk.download('averaged_perceptron_tagger')
5  tagged2 = nltk.pos_tag(tokens)
6  print(tagged2)
```

```
[('Hedge', 'NNP'), ('funds', 'NNS'), ('are', 'VBP'), ('cutting', 'VBG'), ('their', 'PRP$'), ('standard', 'JJ'),
('fees', 'NNS'), ('of', 'IN'), ('2', 'CD'), ('%', 'NN'), ('of', 'IN'), ('assets', 'NNS'), ('under', 'IN'),
('management', 'NN'), ('and', 'CC'), ('20', 'CD'), ('%', 'NN'), ('of', 'IN'), ('profits', 'NNS'), ('amid',
'IN'), ('pressure', 'NN'), ('from', 'IN'), ('investors', 'NNS'), ('.', '.'), ('A', 'DT'), ('team', 'NN'),
('of', 'IN'), ('Ares', 'NNS'), ('and', 'CC'), ('CPP', 'NNP'), ('Investment', 'NNP'), ('Board', 'NNP'), ('are',
'VBP'), ('in', 'IN'), ('the', 'DT'), ('final', 'JJ'), ('stages', 'NNS'), ('of', 'IN'), ('talks', 'NNS'), ('to',
'TO'), ('buy', 'VB'), ('luxury', 'NN'), ('retailer', 'NN'), ('Neiman', 'NNP'), ('Marcus', 'NNP'), ('for',
'IN'), ('around', 'IN'), ('$', '$'), ('6', 'CD'), ('billion', 'CD'), ('.', '.')]
```

> Unigram tagging just uses the most common POS for a given token

# Most common token/tag pairs

```
1 nltk.FreqDist(tagged2).most_common()
```

```
[(('of', 'IN'), 5), (('are', 'VBP'), 2), (('%', 'NN'), 2), (('and', 'CC'), 2), (('.', '.'), 2), (('Hedge',
'NNP'), 1), (('funds', 'NNS'), 1), (('cutting', 'VBG'), 1), (('their', 'PRP$'), 1), (('standard', 'JJ'), 1),
(('fees', 'NNS'), 1), (('2', 'CD'), 1), (('assets', 'NNS'), 1), (('under', 'IN'), 1), (('management', 'NN'),
1), (('20', 'CD'), 1), (('profits', 'NNS'), 1), (('amid', 'IN'), 1), (('pressure', 'NN'), 1), (('from', 'IN'),
1), (('investors', 'NNS'), 1), (('A', 'DT'), 1), (('team', 'NN'), 1), (('Ares', 'NNS'), 1), (('CPP', 'NNP'),
1), (('Investment', 'NNP'), 1), (('Board', 'NNP'), 1), (('in', 'IN'), 1), (('the', 'DT'), 1), (('final', 'JJ'),
1), (('stages', 'NNS'), 1), (('talks', 'NNS'), 1), (('to', 'TO'), 1), (('buy', 'VB'), 1), (('luxury', 'NN'),
1), (('retailer', 'NN'), 1), (('Neiman', 'NNP'), 1), (('Marcus', 'NNP'), 1), (('for', 'IN'), 1), (('around',
'IN'), 1), (('$', '$'), 1), (('6', 'CD'), 1), (('billion', 'CD'), 1)]
```

# What preceeds a noun?

```
1  text = 'Hedge funds are cutting their standard fees of 2% of assets under management and 20% of pr
2  tokens = nltk.tokenize.word_tokenize(text)
3
4  # Requires: nltk.download('brown') and nltk.download('universal_tagset')
5  brown_news_tagged = nltk.corpus.brown.tagged_sents(categories='news', tagset='universal')
6  pos_tagger = nltk.UnigramTagger(brown_news_tagged)
7  tagged3 = pos_tagger.tag(tokens)
8
9  bigrams = nltk.bigrams(tagged3)
10 noun_preceders = [a for (a, b) in bigrams if b[1] == 'NOUN']
11 fdist = nltk.FreqDist(noun_preceders)
12 fdist.most_common()
```

```
[(('of', 'ADP'), 3), (('Hedge', None), 1), (('standard', 'ADJ'), 1), (('under', 'ADP'), 1), (('amid', 'ADP'),
1), (('from', 'ADP'), 1), (('A', 'DET'), 1), (('CPP', None), 1), (('Investment', 'NOUN'), 1), (('final',
'ADJ'), 1), (('buy', 'VERB'), 1), (('Neiman', None), 1)]
```

# How are words and POS used

```
# Requires: nltk.download('treebank') and nltk.download('universal_tagset')
wsj_tagged = nltk.corpus.treebank.tagged_sents(tagset='universal')
pos_tagger = nltk.UnigramTagger(wsj_tagged)
tagged4 = pos_tagger.tag(tokens)
cfd_w2p = nltk.ConditionalFreqDist(tagged4)
cfd_p2w = nltk.ConditionalFreqDist([(item[1], item[0]) for item in tagged4])
```

```
cfd_w2p['funds']
```

```
FreqDist({'NOUN': 1})
```

```
cfd_p2w['NOUN']
```

```
FreqDist({'%': 2, 'funds': 1, 'fees': 1, 'assets': 1, 'management': 1, 'profits': 1, 'pressure': 1,
'investors': 1, 'team': 1, 'Investment': 1, ...})
```

# More details included in the Python file

- The python file uses a full issue of the WSJ
- Some code has to be adjusted to account for multiple files
- Includes an example of using bigram tagging for POS tags

# Using NLP parsers: SpaCy

# SpaCy

- SpaCy provides a machine-learning based approach to many of the things NLTK does
- SpaCy is also perhaps a bit more user-friendly

```python
import spacy

# From outside of python, run: python -m spacy download en_core_web_sm
nlp = spacy.load("en_core_web_sm")
# pipes enabled by default: [tok2vec, tagger, parser, ner, attribute_ruler, lemmatizer]

text = """China said its exports rose 7.2% in August from a year earlier, the latest in a series o

doc = nlp(text)
```

> `nlp()` runs all the pipes in one command

# Parse trees in SpaCy

- SpaCy has a visualization module called displaCy
- With this, we can quickly see how a sentence is structured
- To run it in a Jupyter notebook, use the below code:

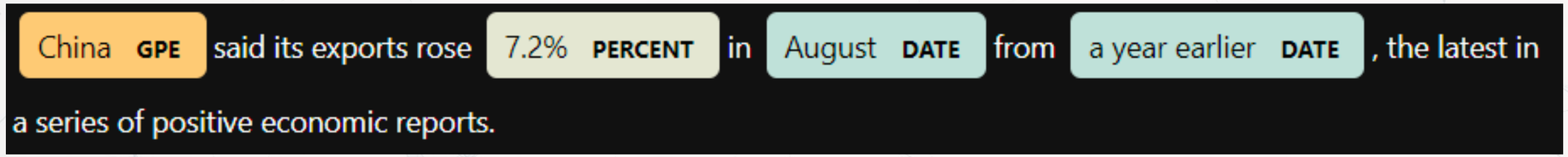
```
1  spacy.displacy.render(doc, style="dep", jupyter=True, options={'compact':True})
```

> Take a look at the code file to see the output

# NER: Named Entity Recognition

- During the `nlp()` call earlier, spaCy automatically did named entity recognition (NER)
- Using an ML algorithm + the dependency tree, it tries to determine any proper nouns in the document
  - It also tries to label them
- You can visualize these as well with `displayCy`

```
1  spacy.displacy.render(text, style="ent", jupyter=True)
```



# Extracting various components

## Entity extraction

```
1  doc.ents
```

```
(China, 7.2%, August, a year earlier)
```

```
1  for ent in doc.ents:
2      print('"'+ ent.text + '" is tag
3      ent.label_ + '" which comprises
4      spacy.explain(ent.label_))
```

```
"China" is tagged as "GPE" which comprises of
Countries, cities, states
"7.2%" is tagged as "PERCENT" which comprises
of Percentage, including "%"
"August" is tagged as "DATE" which comprises of
Absolute or relative dates or periods
"a year earlier" is tagged as "DATE" which
comprises of Absolute or relative dates or
periods
```

## Noun chunk extraction

```
1  list(doc.noun_chunks)
```

```
[China, its exports, August, a series, positive
economic reports]
```

## Lemmatization

```
1  doc[0].sent.lemma_
```

```
'China say its export rise 7.2% in August from
a year early, the late in a series of positive
economic report.'
```

# More details included in the Python file

- Using `nlp.pipe()` instead of `nlp()`
  - Allows you to apply a process to a corpus all at once (as a generator)
- Sentence boundary detection
- PoS tagging in SpaCy
- Lemmatization
- Extracting all entities from a corpus

# Words and their representations

# Words

- Words are an interim building block of language
- Words have a [somewhat] unique meaning
  - This is why methods like Naïve Bayes, cosine similarity, and SVM on word counts had some success in the literature
  - This also allows for constructing dictionaries

> But words are not independent

- Words share sounds, syllables, denotations, connotations, linguistic structure, …
  - Representing this allows for better measurement on text

**Word embeddings**

Word embeddings convert words into an $N$-dim vector capturing facets of the word, primarily its meaning in context.

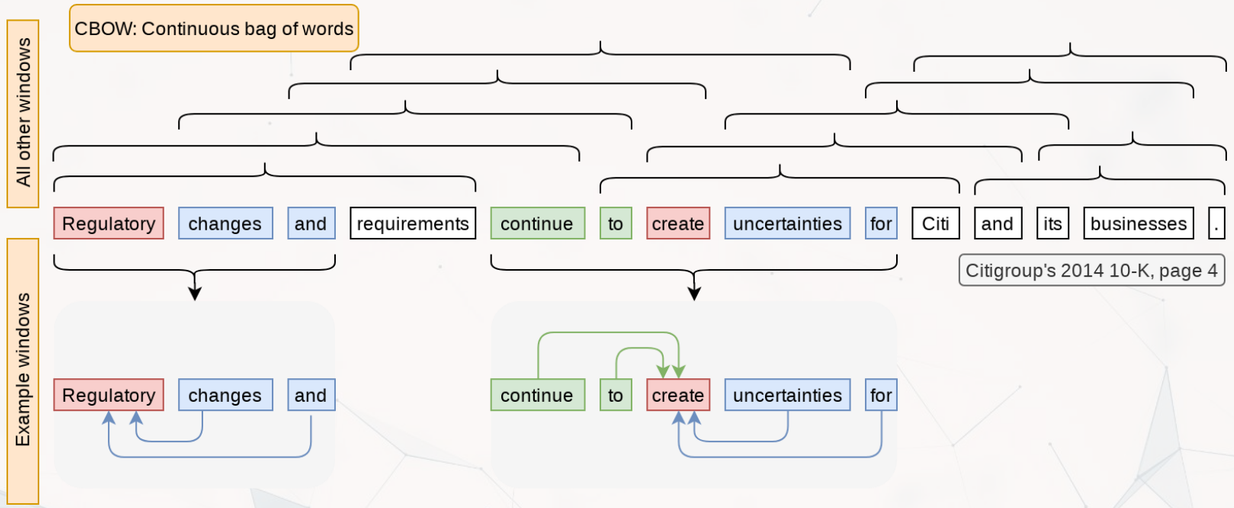
# Word embeddings: A simple example

| words | f_animal | f_people | f_location |
|---|---|---|---|
| **dog** | 0.5 | 0.3 | -0.3 |
| **cat** | 0.5 | 0.1 | -0.3 |
| **Bill** | 0.1 | 0.9 | -0.4 |
| **turkey** | 0.5 | -0.2 | -0.3 |
| **Turkey** | -0.5 | 0.1 | 0.7 |
| **Singapore** | -0.5 | 0.1 | 0.8 |

- The above is a simplified illustrative example
- Notice how we can tell apart different animals based on their relationship with people
- Notice how we can distinguish turkey (the animal) from Turkey (the country) as well

# How can we make these embeddings?

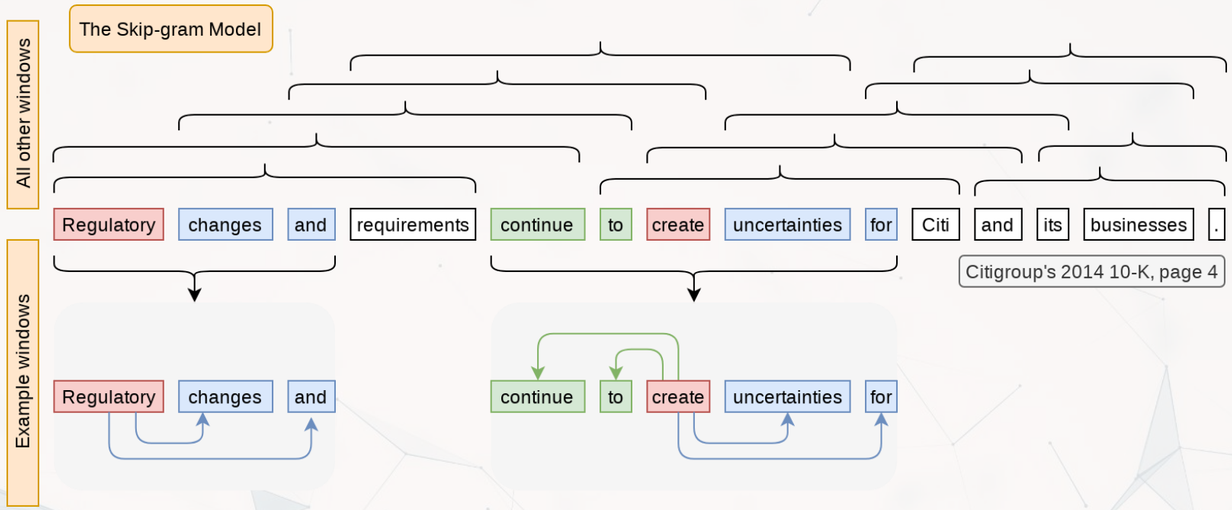
> Infer a word's meaning from the words around it




> Refered to as CBOW (continuous bag of words)

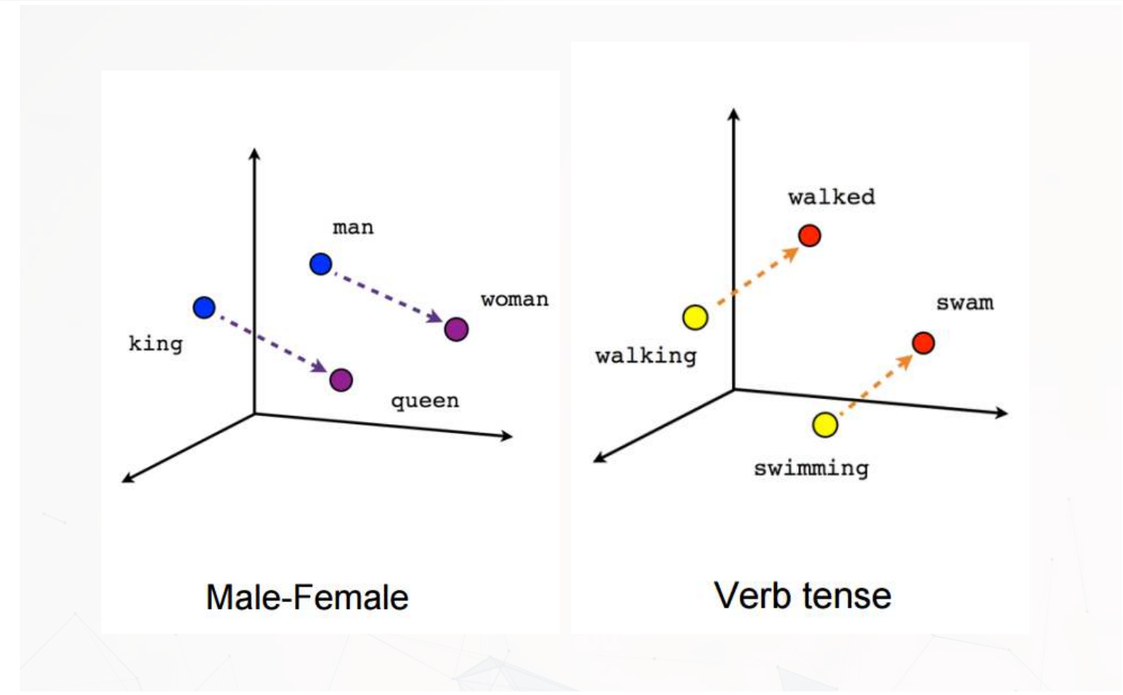
# How else can we infer meaning?

> Infer a word's meaning by *generating* words around it




> Refered to as the Skip-gram model

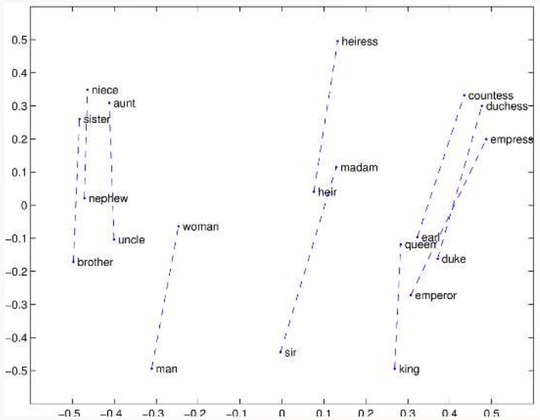
# What it retains: word2vec

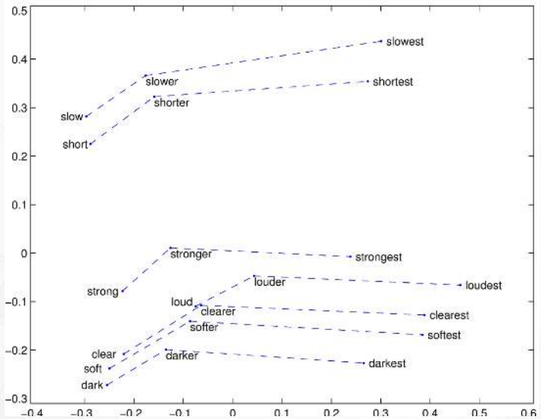


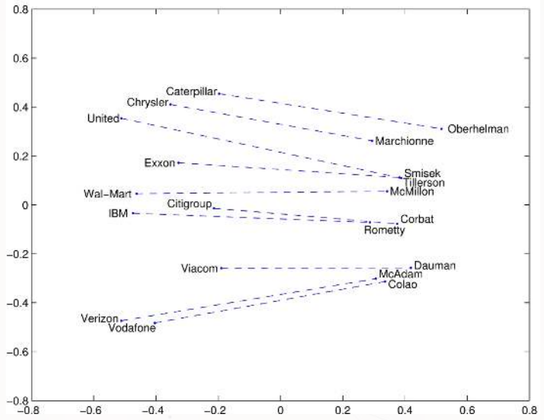


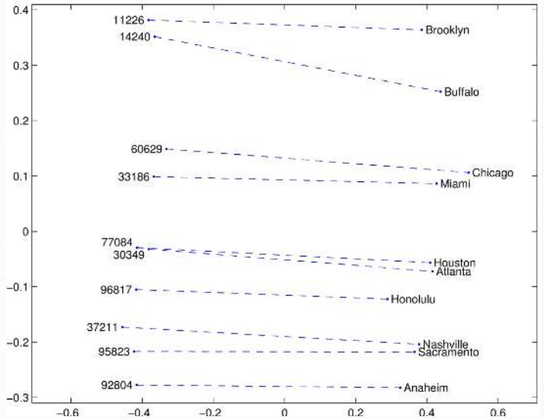
> Relations are retained as vectors between points (distance + direction)

# What it retains: GloVe



# Examining word2vec: Analogies

> man : King :: woman : ?

- Mathematically: $King - man + woman = ?$

```
1  base_w2v.most_similar(positive=['King', 'woman'], negative=['man'])
```

```
('Queen', 0.5515626668930054) ('Oprah_BFF_Gayle', 0.47597548365592957)
 ('Geoffrey_Rush_Exit', 0.46460166573524475) ('Princess', 0.4533674716949463)
 ('Yvonne_Stickney', 0.4507041573524475) ('L._Bonauto', 0.4422135353088379)
 ('gal_pal_Gayle', 0.4408389925956726) ('Alveda_C.', 0.4402790665626526)
 ('Tupou_V.', 0.4373864233493805) ('K._Letourneau', 0.4351031482219696)
```

# The sleight of hand behind this

- Word2Vec implementations usually bar a word in the analogy from being an output
  - E.g., it will never report **man : King :: woman : King**
    - But this is actually the mathematical answer

```
analogy = base_w2v['King'] + base_w2v['woman'] + base_w2v['man']
analogy = analogy / np.linalg.norm(analogy)
print('King', np.linalg.norm(analogy - base_w2v['King']))
```

```
King 1.9888592
```

```
print('Queen', np.linalg.norm(analogy - base_w2v['Queen']))
```

```
Queen 2.7364814
```

# It’s still pretty good though!

- Note that since word2vec’s original answer was Queen, this implies it was second best
  - If Queen is the closest word to King, then this would be mathematically uninteresting
    - It’s actually 7th though!

```
1    base_w2v.most_similar('King')
```

```
[('Jackson', 0.5326348543167114), ('Prince', 0.5306329727172852), ('Tupou_V.', 0.5292826294898987), ('KIng',
0.5227501392364502), ('e_mail_robert.king_@', 0.5173623561859131), ('king', 0.5158917903900146), ('Queen',
0.5157250165939331), ('Geoffrey_Rush_Exit', 0.49920955300331116), ('prosecutor_Dan_Satterberg',
0.49850785732269287), ('NECN_Alison', 0.4912859499454498 3)]
```

# What are word embeddings good for?

1. You care about the words used, by not stylistic choices
    - Abstraction
2. You want to crunch down a bunch of words into a smaller number of dimensions without running any bigger models (like LDA) on the text.
    - E.g., you can toss the 300 dimensions of the Google News model to a Lasso or Elastic Net model
        - This is a big improvement over the past method of tossing vectors of word counts at Naive Bayes
            - 300 << # of unique words
3. You want synonyms for a set of words that are selected in a less-researcher-biased fashion
    - You can even get n-gram synonyms this way
    - A popular method for augmenting small dictionaries

# Demo: Trying out word2vec

> Colab file available at https://rmc.link/colab_w2v

- This exercise is designed to help you understand about what word2vec is good at
  - As well as what it isn't good at

# Building your own word2vec model

1. Build an iterator to feed data in. This can include reading the data line-by-line to minimize memory usage

```
# Iterates document by document, sentence by sentence
# returns non-stopword lemmatized words in order
def sentence_iterator(documents):
    for document in documents:
        for sent in document.sents:
            yield [word.lemma_ for word in list(sent) \
                   if not (word.is_stop or word.is_punct)]
```

2. Feed the data in to build an index of the words

```
w2v_model = gensim.models.Word2Vec(vector_size=100, workers=8)
sentences = sentence_iterator(documents)
w2v_model.build_vocab(sentences, min_count=1)
```

3. Compile the model

```
sentences = sentence_iterator(documents)
w2v_model.train(sentences, total_examples=w2v_model.corpus_count, epochs=1)
```

# Applying our new model

- We can use this model the same as a built in model
  - Just note that we need to call `.wv` to use the underlying vectors

```
1  w2v_model.wv.most_similar('Street')
```

```
[('Media', 0.3910236060619354), ('false', 0.3888976871967315), ('fiscally', 0.3784983158111572), ('statement',
0.37563976645469666), ('farm', 0.33516165614128113), ('Drizen', 0.3255394399166107), ('Marie',
0.32349297404289246), ('sneaker', 0.32272082567214966), ('Bollore', 0.3198564648628235), ('tenant',
0.31797030568122864)]
```

```
1  w2v_model.wv.most_similar('media')
```

```
[('explanation', 0.37455514073371887), ('news', 0.3445030450820923), ('Christine', 0.3439321219921112),
('Coeur', 0.34068533778190613), ('Liberals', 0.3217358887195587), ('purchase', 0.31874051690101624), ('ballot',
0.3166697919368744), ('Roger', 0.31438490748405457), ('temporarily', 0.312235563993454), ('alike',
0.3100491762161255)]
```

# Vector space (embedding) models

- All such models convert some abstract information into numeric information
  - Focus on maintaining some of the underlying structure of the abstract information
- Examples (from smallest to largest granularity):
  - Word vectors:
    - Word2vec (available in `gensim`)
    - GloVe (available as a download from Stanford NLP)
    - fastText (available in `fastText`)
  - Sentence vectors:
    - Universal Sentence Encoder (available from `tensorflow_hub`)
  - Paragraph vectors:
    - Doc2vec (available in `gensim`)
    - BERT first stage model (available from `huggingface`, `tensorflow_hub`, `pytorch_hub`, `sentence_transformers`, etc.)
  - Topic vectors (document level):
    - Latent Dirichlet Allocation (LDA) (available in `gensim`, `sklearn`, etc.)

# Conclusion

# Wrap-up

> Linguistics is largely handled by importing specialized libraries

- `NLTK` for traditional measures
- `SpaCy` for more powerful, ML-based measures
- `Stanza` for Stanford NLP measures

> Easy to calculate many different measures, such as grammar/parts of speech or entities (NER)

- Word vectors can be implemented through `gensim` or `fastText` (which supports 157 languages)

# Packages used for these slides

**Python**

- `nltk`
- `numpy`
- `spacy`
- `gensim`

**R**

- downlit
- kableExtra
- knitr
- quarto
- reticulate
- revealjs

# References


- Harris, Zellig S. “Distributional structure.” Word 10, no. 2-3 (1954): 146-162.
- Garg, Nikhil, Londa Schiebinger, Dan Jurafsky, and James Zou. “Word embeddings quantify 100 years of gender and ethnic stereotypes.” Proceedings of the National Academy of Sciences 115, no. 16 (2018): E3635-E3644.
- Garimella, Aparna, Carmen Banea, Dirk Hovy, and Rada Mihalcea. “Women’s syntactic resilience and men’s grammatical luck: Gender-bias in part-of-speech tagging and dependency parsing.” In Association for Computational Linguistics. 2019.
- Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. “Efficient estimation of word representations in vector space.” arXiv preprint arXiv:1301.3781 (2013).