# Language Analytics

Session 6: Word embeddings

Ross Deans Kristensen-McLachlan
rdkm@cas.au.dk

# Course outline

- 1. Introductions
- 2. String Processing with Python
- 3. NLP for linguistic analysis
- 4. Text Classification 1
- 5. Text Classification 2
- **6. Word embeddings**
- 7. Language modelling 1
- 8. Language modelling 2
- 9. BERT
- 10. More BERT
- 11. Project pitches
- 12. Generative models
- 13. Social impact

# Plan for today

- Catch-up, assignments, and mid-term
- 1. What are word embeddings?
- 2. How do they work?
  - C-BOW vs skip-gram
  - Similarity and distance
  - Applications of word embeddings
- 3. Code-along session
  - Exploring word embeddings
  - Experimenting and testing

# What are word embeddings?

# What are word embeddings?

- Natural language consists of the kinds of thing that we call *words*
- As we've seen, most NLP tasks require the conversion of *words* into *numbers*
  - Think: CountVectorizer, TfidfVectorizer
- With these approaches, we reduce a *document* to a vector of numbers
  - Perhaps the 500 most common words; 500 highest Tfidf scores

# What are word embeddings?

- When we represent documents in this manner, we create a numerical representation of the text

- This numerical representation has a *fixed dimensionality* and correspond the same features across documents
  - This allows us to more effectively classify and cluster documents based on the similarity of their numerical representation

- But what about if we want to study individual words? How can we numerically represent words?

# One-hot encoding

- The simplest way to represent individual words numerically is through *one-hot encoding*
- We create a vector that is the size of the vocabulary and assign 0 or 1
- Each word is then a n-dimensional array of mostly zeroes, with a one appearing in the position corresponding to the word

*The dog bit the man*

| | the, dog, bit, man |
|---|---|
| The | [1, 0, 0 ,0] |
| dog | [0, 1, 0, 0] |
| bit | [0, 0, 1, 0] |
| the | [1, 0, 0, 0] |
| man | [0, 0 ,0, 1] |

# One-hot encoding

- Despite the simplicity, this approach is a bit useless
- Firstly, the vectors are incredibly *sparse*. Here our vocabulary is N=4. But imagine a vocabulary of N=10,000 or more
  - Sparse vectors are both conceptually and computationally inefficient
- Secondly, these vectors tell us nothing interesting about the words themselves

*The dog bit the man*

| | the, dog, bit, man |
|---|---|
| The | [1, 0, 0 ,0] |
| dog | [0, 1, 0, 0] |
| bit | [0, 0, 1, 0] |
| the | [1, 0, 0, 0] |
| man | [0, 0 ,0, 1] |

# Enter word embeddings

- One-hot encoding allows us to create a simple numerical (binary) vector for each word

- This representation tells us nothing of about the words themselves and their relationship to one another, either semantically or grammatically

- We need to be able to create dense numerical representations of words which also encode semantic information
  - This is exactly the purpose of word embedding models

# The distributional hypothesis

- Yep, it's back again!
  - *You shall know a word by the company it keeps*

- This states that words which occur in similar linguistic contexts share similar meanings
  - For example, *ophthalmologist* and *eye-doctor*

- Word embedding models build on this linguistic intuition by learning representations words from their distributions in texts

# Types of word embeddings

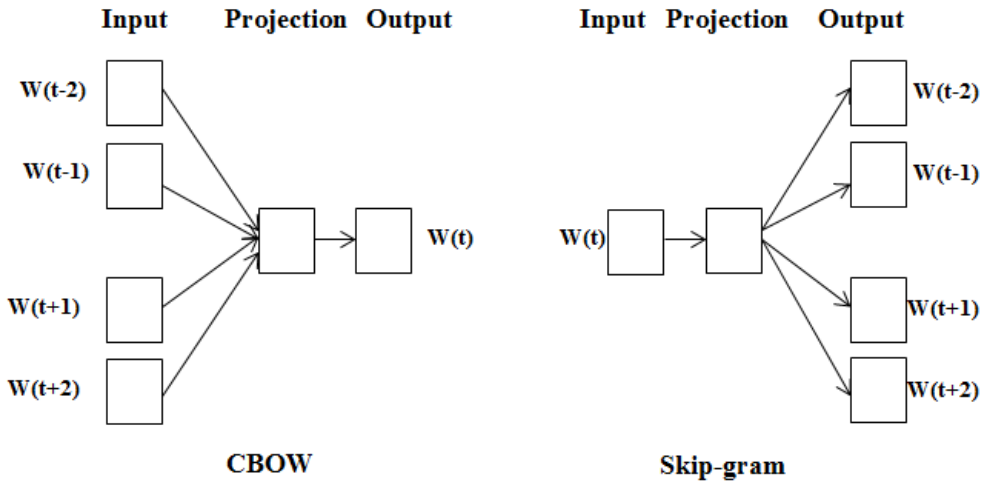
- Static word embeddings
  - word2vec, GloVe, fastText
- Dynamic word embeddings
  - Elmo, BERT
- Today, we'll be looking only at static embeddings; we'll come back to dynamic embeddings in another class
- We'll focus on *word2vec* in class but will mention GloVe and fastText in the hands-on session

# word2vec

- Think back to when we look at *collocations*
- Collocation allows us to study the words which systematically co-occur within a given window size from our target word
- The intuition behind word2vec is not entirely dissimilar
- The difference is that instead of counting co-occurrence and calculating strength of association, we instead train a classifier on a binary prediction task

# CBOW vs Skip-gram

- This ‘binary classification task’ has two distinct flavours
  - CBOW – Continuous Bag of Words
  - Skip-gram
- CBOW: “Given context *c* what is the most likely word *w*?”
- Skip-gram: “Given word *w*, what is the most likely context *c*?”
- Skip-gram tends to perform better on smaller data and with rarer words; CBOW on large data and frequent words



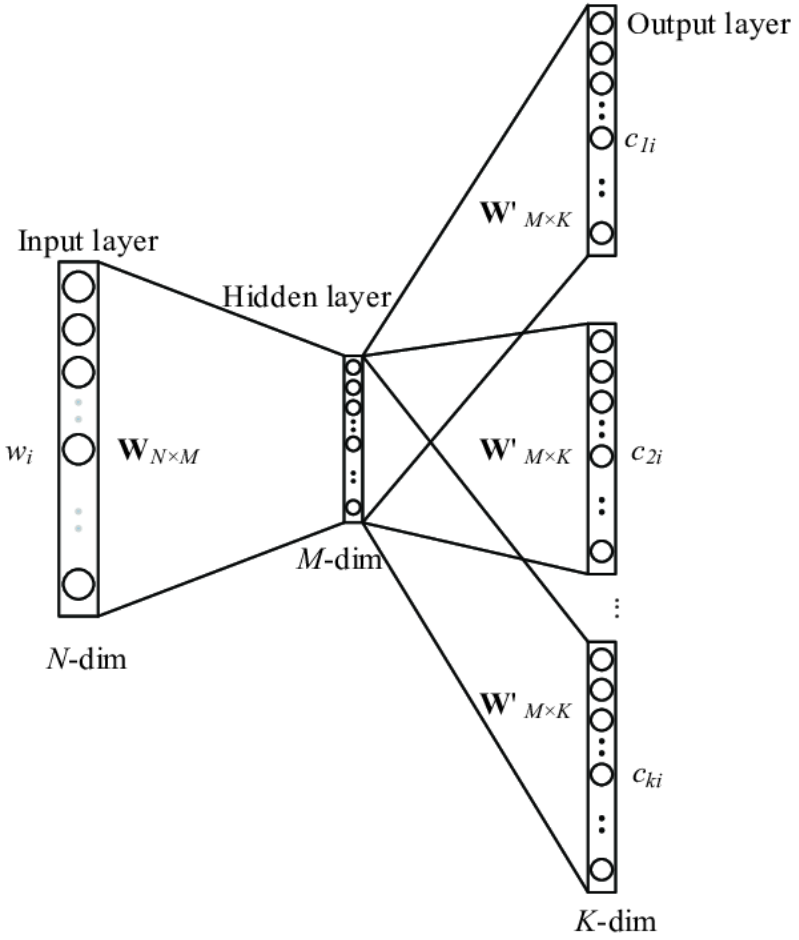
# Break

# Focusing on skip-gram

- We'll focus here on skip-gram with negative sampling (SGNS)

- SGNS is a *self-supervised*
  - 1. Treat the target word and a neighbouring context word as positive examples
  - 2. Randomly sample other words in the lexicon to get negative samples
  - 3. Use logistic regression to train a classifier to distinguish those two cases
  - 4. Use the learned weights as the embeddings

- We don't actually care about this prediction task
- Instead we take the learned classifier weights as the word embeddings – i.e. dense numerical vector representations of words



# So what?

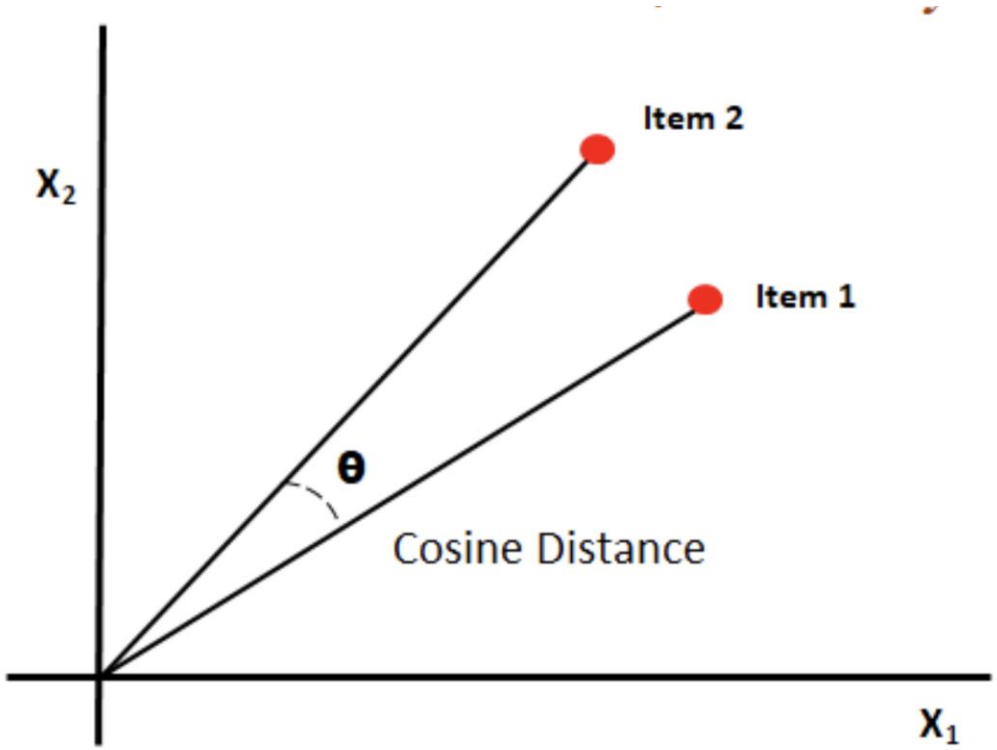
- But what actually *is* a word embedding?
- Usually we end up with a fixed dimension array of numbers
  - 50, 100, 300
- This can be thought of *geometrically*, as a point in a multidimensional space

```
>> trained_vectors[“denmark”]

array([ 0.054976 , 0.10817 , -0.50324 , -0.019769 ,
0.24695 , 1.1584 , -0.26947 , 0.0031433, 0.13016 , -
0.89177 , 1.217 , -0.63195 , -0.32401 , -0.5139 ,
0.87985 , 0.75084 , 0.84651 , -1.0341 , -0.87887 ,
1.0882 , 0.49265 , -0.59829 , -0.048051 , 1.0969 ,
0.20229 , -1.2686 , 0.29324 , -0.49264 , -0.96397 , -
0.13358 , 1.4462 , 1.1363 , 0.082198 , 0.58515 , -
0.8055 , -0.36287 , 0.44576 , 0.77141 , -0.80198 ,
0.08235 , 1.032 , 0.031318 , 0.88945 , -1.2146 ,
0.049257 , 0.85534 , -0.5116 , 0.13401 , -0.23439 , -
1.0067 ], dtype=float32)
```

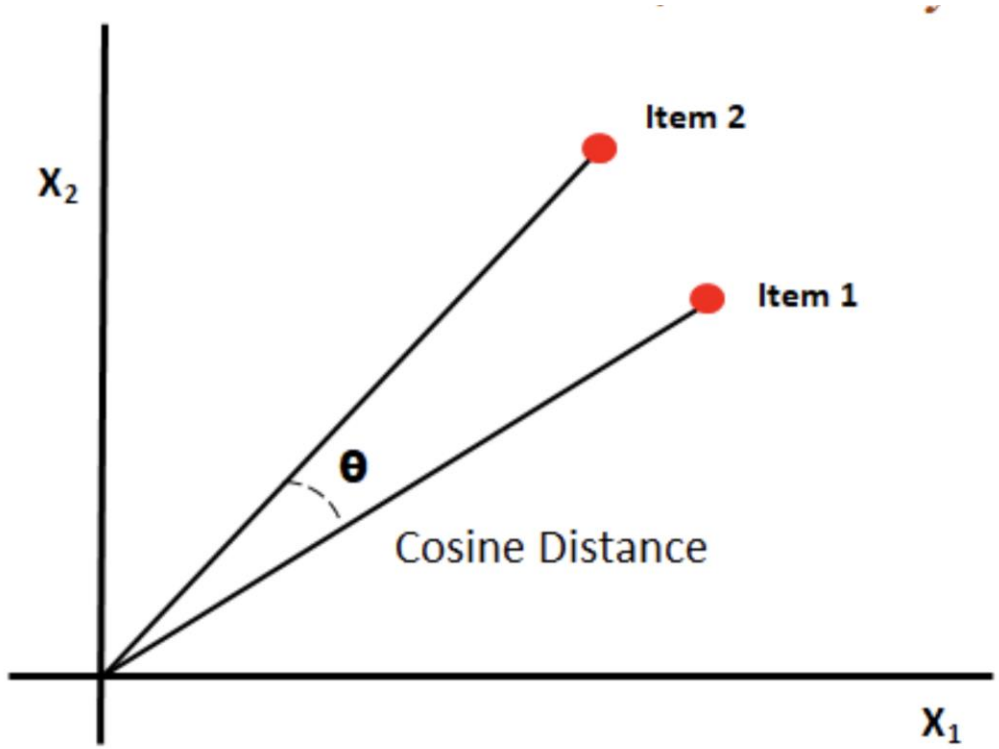
# Cosine similarity

- Consider the graph to the right
- Item 1 and 2 are separated by a certain angle θ
- A little high school trigonometry…
  - cos(0) = 1
  - cos(90) = 0
  - cos(180) = -1



# Cosine similarity

- So we can use the cosine of the angle θ between 1 and 2 to calculate their *similarity*
  - Distance = 1-cos(θ)
- This maths also works in multidimensional spaces
- Word embeddings are not simply numerical translations of words
- Instead, by formalising the distributional hypothesis, word embeddings encode something about word *meaning*



# An example

- The example to the right shows the cosine similarity for Denmark to two other countries

- This uses *pre-trained* word embeddings
  - We'll talk about this more in the hands-on session

- What this shows is that *Denmark* and *Sweden* are more similar that *Denmark* and *Scotland*

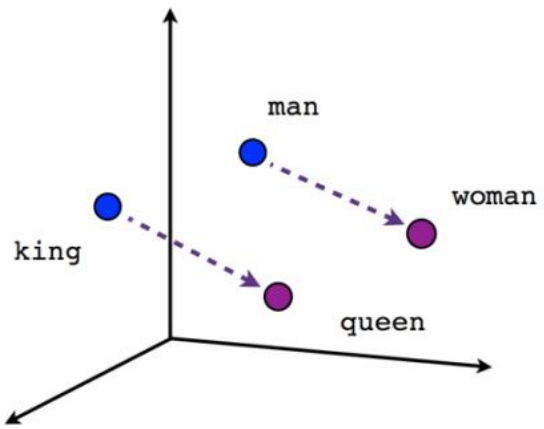
```
pretrained_vectors.similarity('denmark', 'scotland')
```

```
0.6451837
```

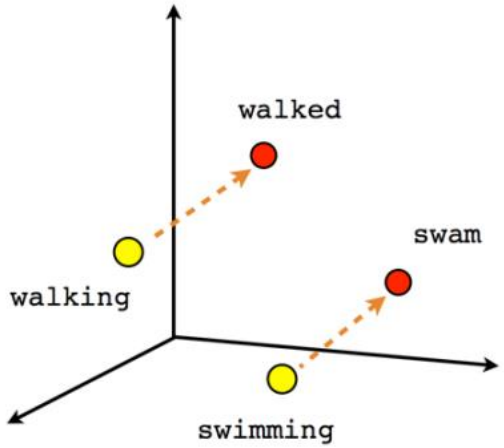
```
pretrained_vectors.similarity('denmark', 'sweden')
```

```
0.8847619
```

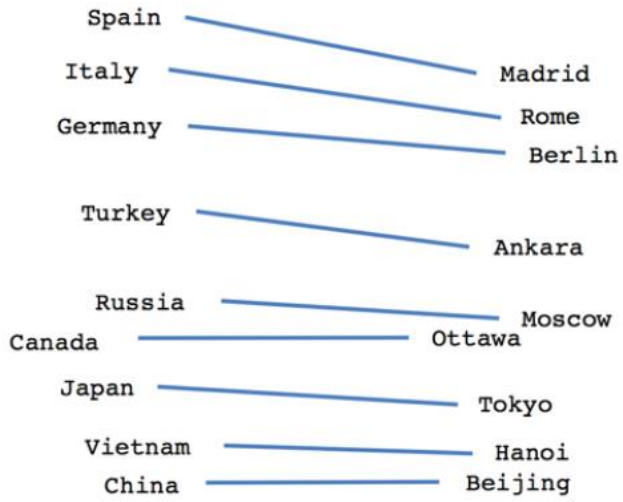
# More examples




Male-Female




Verb tense




Country-Capital

Go to http://projector.tensorflow.org/

# Group discussion

- What do you think about this approach?
  - Do word embeddings make sense, based on what we've already looked at?
  - Does the intuition make sense to you or is it completely *counter*-intuitive?
- What kind of research questions do you think word embeddings might be used to study?
  - Can you think of examples from your own disciplines?
- Can you think of any problems with the approach we've discussed so far?

# Summary

- word2vec formalises the intuition behind the distributional hypothesis and reframes it as a binary classification problem
- The output weights from the word2vec algorithm are the numerical vectors we use to represent words
- These vectors of numbers are called embeddings and are dense representations of words based on the contexts in which they occur
- These representations seem to encode meaningful linguistic information about words in a way which seems cognitively plausible
- Word embeddings are the foundation of all contemporary NLP methods

# Break

And then head over to UCloud