NLP without Annotated Dataset

Text Embeddings and Transfer Learning in NLP

Sowmya Vajjala

Seminar für Sprachwissenschaft, University of Tübingen, Germany

20 January 2021

Class Outline

- Reminders + Last class - a quick recap
- Text representation - a quick recap
- Neural text embedding -word2vec to BERTs: an overview
- Using embeddings for downstream NLP tasks: examples
- Transfer learning for NLP: an example with Fine-Tuning BERT

Note: This is perhaps going to be a long expository session. Be prepared!

Group Discussion schedule

- We are starting this Friday!
- Teams/Papers info on the forum.
- Presentation schedule (also posted on forum)
  1. 22nd January: Teams 1–3
  2. 25th January: Teams 4–6
  3. 27th January: Teams 7–9
- Format: 15-20 minutes of presentation, Around 10 min of discussion per team (everyone can ask questions. Presenters don't have to know all answers. I will also try to answer some of the questions that come up!)

Questions on this?

Last Class

- Spam classification with snorkel:
  - Data labeling with labeling functions
  - Data augmentation with transformation functions
  - How do they both compare with the original training data (with labels)

- Questions or comments on this?

Text Representation: A quick recap

- Bag of words
- Bag of n-grams
- TF-IDF

note: code examples in following slides taken from Chapter 3 of Practical NLP Book. (github repo)

Words and Vectors

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 6.3 The term-document matrix for four words in four Shakespeare plays. The red boxes show that each document is represented as a column vector of length four.

We can think of the vector for a document as a point in $|V|$-dimensional space; thus the documents in Fig. 6.3 are points in 4-dimensional space. Since 4-dimensional spaces are hard to visualize, Fig. 6.4 shows a visualization in two dimensions; we've arbitrarily chosen the dimensions corresponding to the words *battle* and *fool*.

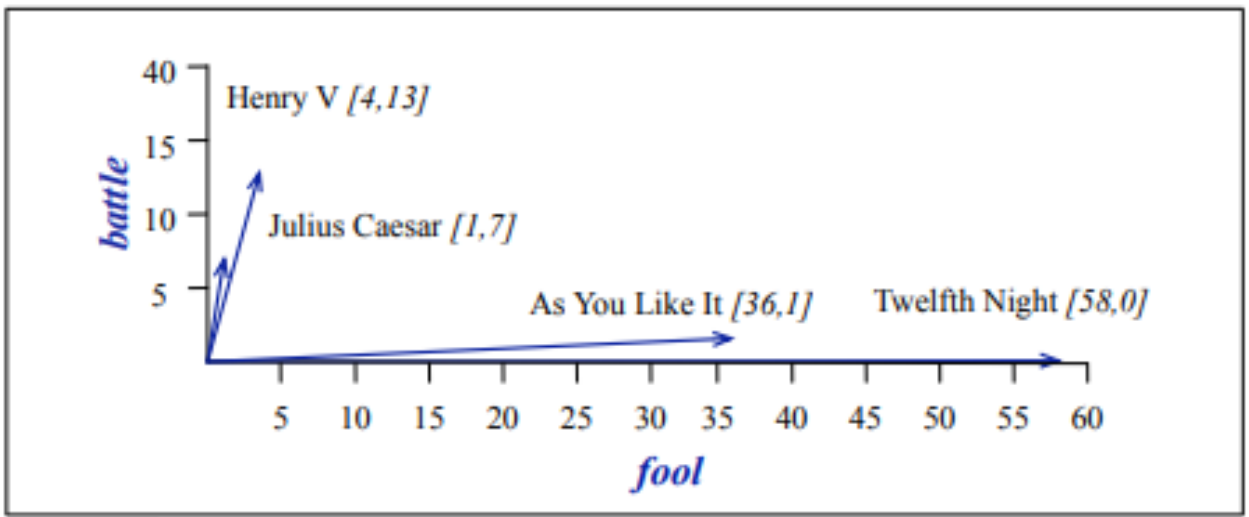
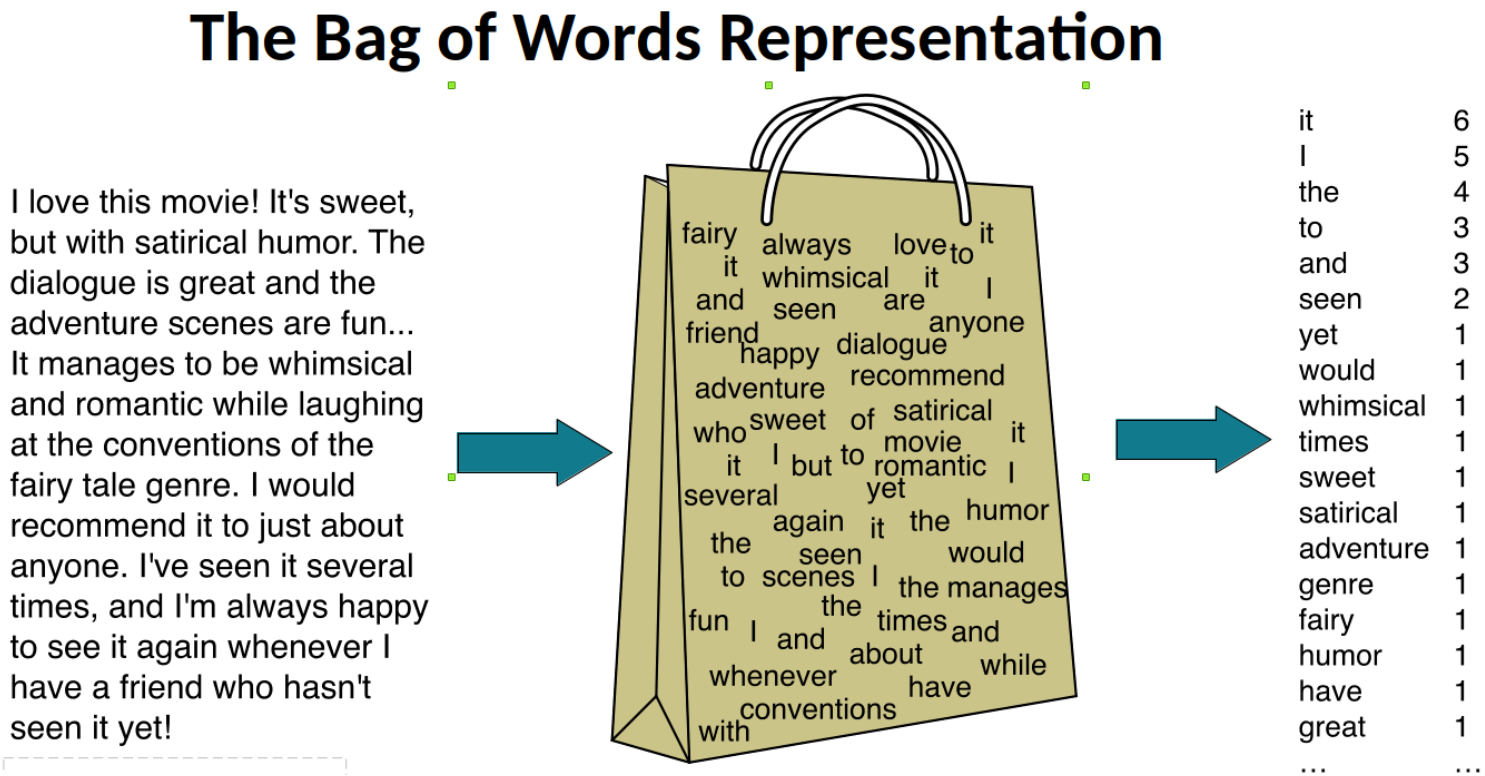


Figure 6.4 A spatial visualization of the document vectors for the four Shakespeare play documents, showing just two of the dimensions, corresponding to the words *battle* and *fool*. The comedies have high values for the *fool* dimension and low values for the *battle* dimension.

source: https://web.stanford.edu/~jurafsky/slp3/6.pdf

Bag of Words



source: https://web.stanford.edu/~jurafsky/slp3/4.pdf

Toy Corpus

```
documents = ["Dog bites man.", "Man bites dog.",
             "Dog eats meat.", "Man eats food."]
processed_docs = [doc.lower().replace(".","")
                  for doc in documents]
print(processed_docs)
```

output: ['dog bites man', 'man bites dog', 'dog eats meat', 'man eats food']

BOW Example

```
from sklearn.feature_extraction.text import CountVectorizer

#look at the documents list
print("Our corpus: ", processed_docs)

count_vect = CountVectorizer()
#Build a BOW representation for the corpus
bow_rep = count_vect.fit_transform(processed_docs)

#Look at the vocabulary mapping
print("Our vocabulary: ", count_vect.vocabulary_)

#see the BOW rep for first 2 documents
print("BoW representation for 'dog bites man': ", bow_rep[0].toarray())
print("BoW representation for 'man bites dog: ",bow_rep[1].toarray())

#Get the representation using this vocabulary, for a new text
temp = count_vect.transform(["dog and dog are friends"])
print("Bow representation for 'dog and dog are friends':", temp.toarray())
```

```
Our corpus:  ['dog bites man', 'man bites dog', 'dog eats meat', 'man eats food']
Our vocabulary:  {'dog': 1, 'bites': 0, 'man': 4, 'eats': 2, 'meat': 5, 'food': 3}
BoW representation for 'dog bites man':  [[1 1 0 0 1 0]]
BoW representation for 'man bites dog:  [[1 1 0 0 1 0]]
Bow representation for 'dog and dog are friends': [[0 2 0 0 0 0]]
```

BOW Example with Binary Values

```
#BoW with binary vectors
count_vect = CountVectorizer(binary=True)
bow_rep_bin = count_vect.fit_transform(processed_docs)
temp = count_vect.transform(["dog and dog are friends"])
print("Bow representation for 'dog and dog are friends':", temp.toarray())
```

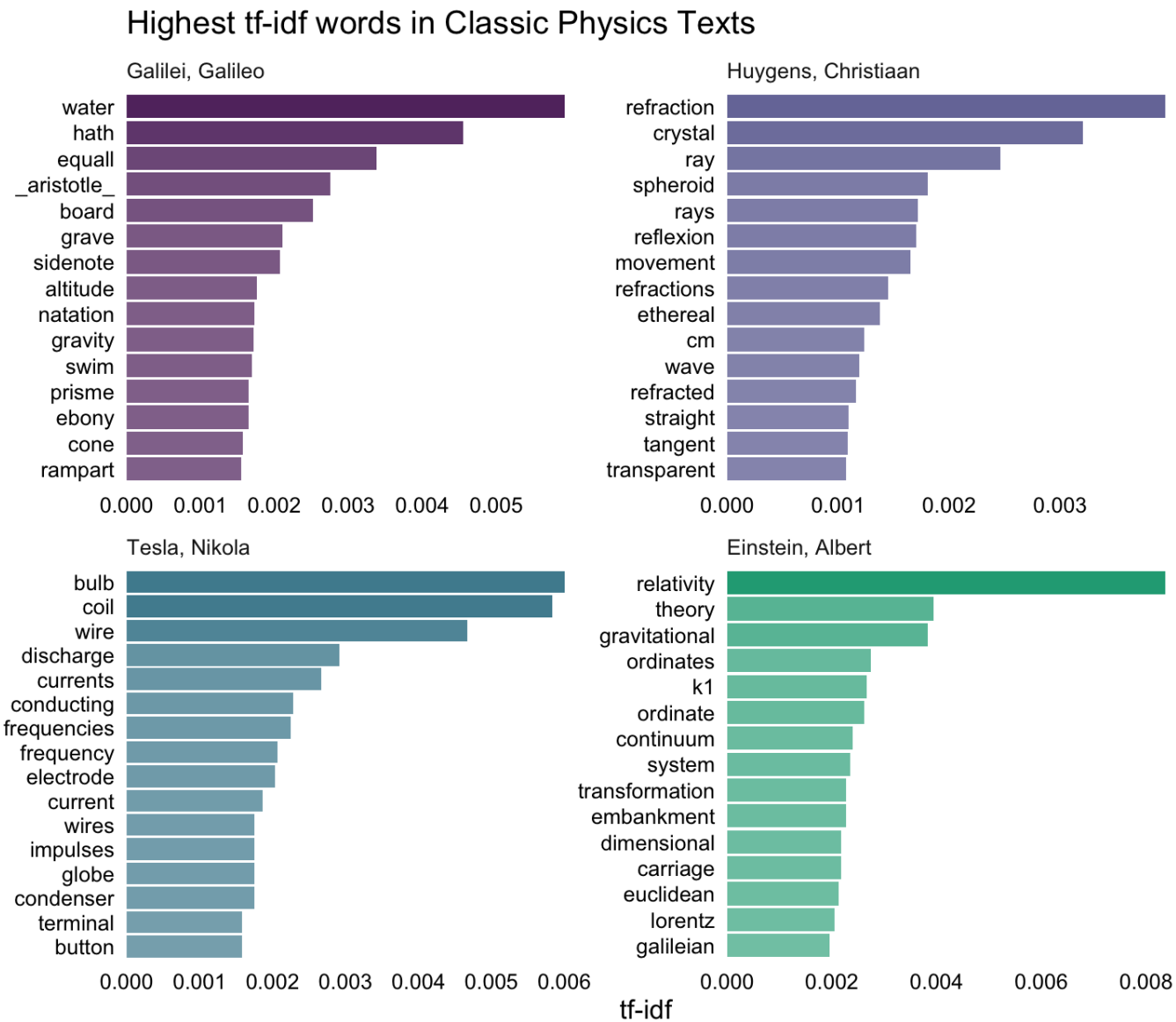
```
Bow representation for 'dog and dog are friends': [[0 1 0 0 0 0]]
```

BON Example

```
from sklearn.feature_extraction.text import CountVectorizer

#look at the documents list
print("Our corpus: ", processed_docs)

count_vect = CountVectorizer()
#Build a BOW representation for the corpus
bow_rep = count_vect.fit_transform(processed_docs)

#Look at the vocabulary mapping
print("Our vocabulary: ", count_vect.vocabulary_)

#see the BOW rep for first 2 documents
print("BoW representation for 'dog bites man': ", bow_rep[0].toarray())
print("BoW representation for 'man bites dog: ",bow_rep[1].toarray())

#Get the representation using this vocabulary, for a new text
temp = count_vect.transform(["dog and dog are friends"])
print("Bow representation for 'dog and dog are friends':", temp.toarray())
```

```
Our corpus:  ['dog bites man', 'man bites dog', 'dog eats meat', 'man eats food']
Our vocabulary:  {'dog': 1, 'bites': 0, 'man': 4, 'eats': 2, 'meat': 5, 'food': 3}
BoW representation for 'dog bites man':  [[1 1 0 0 1 0]]
BoW representation for 'man bites dog:  [[1 1 0 0 1 0]]
Bow representation for 'dog and dog are friends': [[0 2 0 0 0 0]]
```

TF-IDF



source: https://www.r-bloggers.com/2016/06/term-frequency-and-tf-idf-using-tidy-data-principles/

TF-IDF Example

```
from sklearn.feature_extraction.text import CountVectorizer

#look at the documents list
print("Our corpus: ", processed_docs)

count_vect = CountVectorizer()
#Build a BOW representation for the corpus
bow_rep = count_vect.fit_transform(processed_docs)

#Look at the vocabulary mapping
print("Our vocabulary: ", count_vect.vocabulary_)

#see the BOW rep for first 2 documents
print("BoW representation for 'dog bites man': ", bow_rep[0].toarray())
print("BoW representation for 'man bites dog: ",bow_rep[1].toarray())

#Get the representation using this vocabulary, for a new text
temp = count_vect.transform(["dog and dog are friends"])
print("Bow representation for 'dog and dog are friends':", temp.toarray())
```

```
Our corpus:  ['dog bites man', 'man bites dog', 'dog eats meat', 'man eats food']
Our vocabulary:  {'dog': 1, 'bites': 0, 'man': 4, 'eats': 2, 'meat': 5, 'food': 3}
BoW representation for 'dog bites man':  [[1 1 0 0 1 0]]
BoW representation for 'man bites dog:  [[1 1 0 0 1 0]]
Bow representation for 'dog and dog are friends': [[0 2 0 0 0 0]]
```

What is good about these representations?

- Easy to understand.
- Easy to implement programmatically.
- Documents sharing vocabulary will be close to each other in the representation space.
- We have a fixed length vector for a sentence/text of any length. (What does this mean?)

What are some drawbacks?

- Size of the vector: If we take Bag of Words/TF-IDF, it is the number of unique words in the corpus. If we take N-grams, it is the number of unique n-grams.
  $\Rightarrow$ This can get pretty big (unless we decide to cut it at top-N words/ngrams).
- Most of the items in the vector are zeroes (so, these vectors are sparse) - why is this a problem?

What are some drawbacks?

- Size of the vector: If we take Bag of Words/TF-IDF, it is the number of unique words in the corpus. If we take N-grams, it is the number of unique n-grams.
  $\Rightarrow$ This can get pretty big (unless we decide to cut it at top-N words/ngrams).
- Most of the items in the vector are zeroes (so, these vectors are sparse) - why is this a problem?
- If we encounter a new word later while using this representation on new texts, we don't know what to do.

What are some drawbacks?

- Size of the vector: If we take Bag of Words/TF-IDF, it is the number of unique words in the corpus. If we take N-grams, it is the number of unique n-grams.
  $\Rightarrow$ This can get pretty big (unless we decide to cut it at top-N words/ngrams).
- Most of the items in the vector are zeroes (so, these vectors are sparse) - why is this a problem?
- If we encounter a new word later while using this representation on new texts, we don't know what to do.

- How do we address these issues?

What are some drawbacks?

- Size of the vector: If we take Bag of Words/TF-IDF, it is the number of unique words in the corpus. If we take N-grams, it is the number of unique n-grams.
  $\Rightarrow$ This can get pretty big (unless we decide to cut it at top-N words/ngrams).
- Most of the items in the vector are zeroes (so, these vectors are sparse) - why is this a problem?
- If we encounter a new word later while using this representation on new texts, we don't know what to do.

- How do we address these issues? Text embeddings are a step in that direction.

Before learning about embeddings...

Distributional Hypothesis

- Words that occur in similar contexts may be related in some way (synonyms, antonyms etc)
- The link between similarity in how words are distributed and similarity in what they mean is called the distributional hypothesis.
- This hypothesis was studied in 20th century linguistics in 1950s and it extended to the concept of "vector semantics" which forms the basis of neural text representations known as embeddings in NLP.

Slides that follow are based on Chapter 6 in Jurafsky & Martin, 3rd Edition

Words and their relationships

If we then take every occurrence of each word (say **strawberry**) and count the context words around it, we get a word-word co-occurrence matrix. Fig. 6.6 shows a simplified subset of the word-word co-occurrence matrix for these four words computed from the Wikipedia corpus (Davies, 2015).

	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...

Figure 6.6 Co-occurrence vectors for four words in the Wikipedia corpus, showing six of the dimensions (hand-picked for pedagogical purposes). The vector for *digital* is outlined in red. Note that a real vector would have vastly more dimensions and thus be much sparser.

Note in Fig. 6.6 that the two words *cherry* and *strawberry* are more similar to each other (both *pie* and *sugar* tend to occur in their window) than they are to other words like *digital*; conversely, *digital* and *information* are more similar to each other than, say, to *strawberry*. Fig. 6.7 shows a spatial visualization.

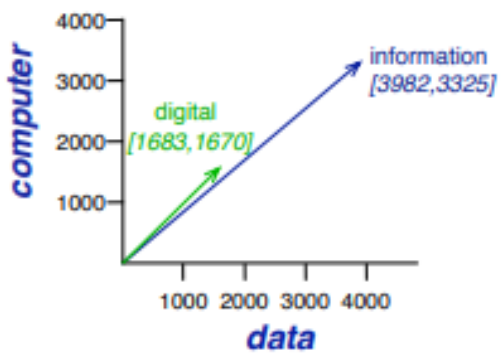


Figure 6.7 A spatial visualization of word vectors for *digital* and *information*, showing just two of the dimensions, corresponding to the words *data* and *computer*.

source: https://web.stanford.edu/~jurafsky/slp3/6.pdf

Similarity between two words

The cosine value ranges from 1 for vectors pointing in the same direction, through 0 for orthogonal vectors, to -1 for vectors pointing in opposite directions. But since raw frequency values are non-negative, the cosine for these vectors ranges from 0–1.

Let's see how the cosine computes which of the words *cherry* or *digital* is closer in meaning to *information*, just using raw counts from the following shortened table:

	pie	data	computer
cherry	442	8	2
digital	5	1683	1670
information	5	3982	3325

$$\cos(\text{cherry}, \text{information}) = \frac{442*5+8*3982+2*3325}{\sqrt{442^2+8^2+2^2}\sqrt{5^2+3982^2+3325^2}} = .017$$

$$\cos(\text{digital}, \text{information}) = \frac{5*5+1683*3982+1670*3325}{\sqrt{5^2+1683^2+1670^2}\sqrt{5^2+3982^2+3325^2}} = .996$$

The model decides that *information* is way closer to *digital* than it is to *cherry*, a result that seems sensible. Fig. 6.8 shows a visualization.

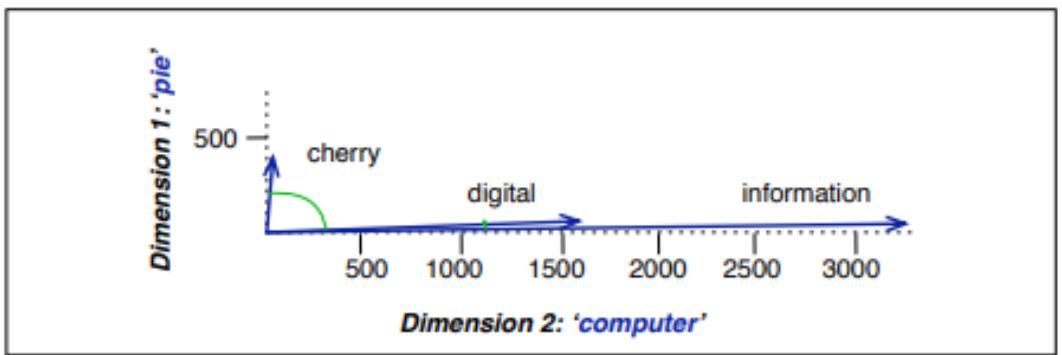


Figure 6.8 A (rough) graphical demonstration of cosine similarity, showing vectors for three words (*cherry*, *digital*, and *information*) in the two dimensional space defined by counts of the words *computer* and *pie* nearby. Note that the angle between *digital* and *information* is smaller than the angle between *cherry* and *information*. When two vectors are more similar, the cosine is larger but the angle is smaller; the cosine has its maximum (1) when the angle between two vectors is smallest (0°); the cosine of all other angles is less than 1.

source: https://web.stanford.edu/~jurafsky/slp3/6.pdf

Intuition

For example, suppose you didn't know the meaning of the word *ongchoi* (a recent borrowing from Cantonese) but you see it in the following contexts:

(6.1) Ongchoi is delicious sauteed with garlic.

(6.2) Ongchoi is superb over rice.

(6.3) ...ongchoi leaves with salty sauces...

And suppose that you had seen many of these context words in other contexts:

(6.4) ...spinach sauteed with garlic over rice...

(6.5) ...chard stems and leaves are delicious...

(6.6) ...collard greens and other salty leafy greens

The fact that *ongchoi* occurs with words like *rice* and *garlic* and *delicious* and *salty*, as do words like *spinach*, *chard*, and *collard greens* might suggest that ongchoi is a leafy green similar to these other leafy greens.[1] We can do the same thing computationally by just counting words in the context of *ongchoi*.

source: https://web.stanford.edu/~jurafsky/slp3/6.pdf

Vector Semantics

- Vector semantics is the standard way to represent word meaning in NLP.
- Vector semantic approaches "learn" to define the meaning of a word by its distribution in language use i.e., its neighboring words or grammatical environment.

Vector Semantics

- Vector semantics is the standard way to represent word meaning in NLP.
- Vector semantic approaches "learn" to define the meaning of a word by its distribution in language use i.e., its neighboring words or grammatical environment.
- The goal of such a model is to represent a word as a point in a multidimensional semantic space (called an "embedding") that is derived from distributions of its neighbors.
- The question of how to "learn" this multidimensional space resulted in various approaches to word embeddings in NLP.

What do Embeddings learn?

Figure 6.1 A two-dimensional (t-SNE) projection of embeddings for some words and phrases, showing that words with similar meanings are nearby in space. The original 60-dimensional embeddings were trained for sentiment analysis. Simplified from Li et al. (2015) with colors added for explanation.

source: https://web.stanford.edu/~jurafsky/slp3/6.pdf

Earlier text representations vs embeddings

- Count based vs learnt representations
- Sparse vectors with mostly zeros vs Dense vectors with non-zero values in all dimensions.
- Long vectors, scaling with vocabulary size vs Short vectors with limited dimensions.

Why bother about embeddings?

- Because embeddings are everywhere now!
- It turns out that dense vectors typically work better in every NLP task than sparse vectors.

Why bother about embeddings?

- Because embeddings are everywhere now!
- It turns out that dense vectors typically work better in every NLP task than sparse vectors.
- Although we don't fully know why yet, it is possible that:
  - Shorter and denser vectors will need fewer parameters to learn better for ML/DL models.
  - Dense vectors may be capturing synonymy and other relations between words.

Text Embeddings: Word2Vec

- Intuition: instead of counting how often each word w occurs near, say, apricot, we'll train a classifier on a binary prediction task: "Is word w likely to show up near apricot?"

Text Embeddings: Word2Vec

- Intuition: instead of counting how often each word w occurs near, say, apricot, we'll train a classifier on a binary prediction task: "Is word w likely to show up near apricot?"
- We don't actually care about this prediction task; instead we'll take the learned classifier weights as the word embeddings.

Text Embeddings: Word2Vec

- Intuition: instead of counting how often each word w occurs near, say, apricot, we'll train a classifier on a binary prediction task: "Is word w likely to show up near apricot?"
- We don't actually care about this prediction task; instead we'll take the learned classifier weights as the word embeddings.
- The revolutionary intuition here is that we can just use running text as implicitly supervised training data for such a classifier;
- a word c that occurs near the target word apricot acts as gold 'correct answer' to the question "Is word c likely to show up near apricot?"
- This method, often called self-supervision, avoids the need for any sort of hand-labeled supervision signal.

How do we use them?

- Pre-trained word2vec embeddings are available for various kinsd of text.
- If you want to train your own word2vec embeddings for a specific problem domain, it is easy and fast.
- Instead of using bag of words/ngrams etc, you can just use them as features.
- How?: A common strategy is to get embeddings for individual words from this model, and average them to get full text feature representation

Using pre-trained word2vec

```
from gensim.models import Word2Vec, KeyedVectors
pretrainedpath = '/tmp/input/GoogleNews-vectors-negative300.bin.gz'
w2v_model = KeyedVectors.load_word2vec_format(pretrainedpath, binary=True)
#loads the model

#What is the vector representation for a word?
w2v_model['computer']
```

source

Training a word2vec model

```
from gensim.test.utils import common_texts
from gensim.models import Word2Vec
model = Word2Vec(sentences=common_texts, vector_size=100, window=5, min_count=1, workers=4)
model.save("word2vec.model")
```

More details: https://radimrehurek.com/gensim/models/word2vec.html

What do Embeddings learn?

```
In [6]: #What is the vector representation for a word?
        w2v_model['computer']
```

```
Out[6]: array([ 1.07421875e-01, -2.81171875e-01,  1.23046875e-01,  2.11914062e-01,
               -9.13085938e-02,  2.16796875e-01, -1.31835938e-01,  8.30078125e-02,
                2.02148438e-01,  4.78515625e-02,  3.66210938e-02, -2.45361328e-02,
                2.39257812e-02, -1.68156250e-01, -2.61230469e-02,  9.71679688e-02,
               -6.34765625e-02,  1.84570312e-01,  1.70898438e-01, -1.63085938e-01,
               -1.89375000e-01,  1.49414062e-01, -4.65393066e-04,  9.61914062e-02,
                1.68945312e-01,  2.60925293e-03,  8.93554688e-02,  6.49414062e-02,
                3.56445312e-02, -6.93359375e-02, -1.46484375e-01, -1.21093750e-01,
               -2.27539062e-01,  2.45361328e-02, -1.24511719e-01, -3.18359375e-01,
               -2.20703125e-01,  1.38859375e-01,  3.66210938e-02, -3.63769531e-02,
               -1.13281250e-01,  1.95312500e-01,  9.76562500e-02,  1.26953125e-01,
                6.59179688e-02,  6.93359375e-02,  1.02539062e-02,  1.75781250e-01,
               -1.68945312e-01,  1.21307373e-03, -2.98828125e-01, -1.15234375e-01,
                5.66406250e-02, -1.77734375e-01, -2.08984375e-01,  1.76757812e-01,
                2.38037109e-02, -2.57812500e-01, -4.46777344e-02,  1.88476562e-01,
                5.51757812e-02,  5.82929688e-02, -1.06933594e-01,  1.89453125e-01,
               -1.16210938e-01,  8.49609375e-02, -1.71875000e-01,  2.45117188e-01,
               -1.73828125e-01, -8.38078125e-03,  4.56542969e-02, -1.61132812e-02,
                1.86523438e-01, -6.85468750e-02, -4.17480469e-02,  1.82617188e-01,
                2.20703125e-01, -1.22558594e-01, -2.55126953e-02, -3.88593750e-01,
                9.13085938e-02,  1.68156250e-01,  1.70898438e-01,  1.19628906e-01,
                7.08007812e-02, -2.64892578e-02, -3.08837891e-02,  4.06250000e-01,
               -1.01562500e-01,  5.71289062e-02, -7.26318359e-03, -9.17968750e-02,
               -1.58398625e-01, -2.55859375e-01,  2.16796875e-01, -3.63769531e-02,
                2.24609375e-01,  8.08781250e-02,  1.56250000e-01,  5.27343750e-02,
                1.58398625e-01, -1.14746094e-01, -8.64257812e-02,  1.19140625e-01,
               -7.17773438e-02,  2.73437500e-01, -1.64062500e-01,  7.29370117e-03,
                4.21875000e-01, -1.12792969e-01, -1.35742188e-01, -1.31835938e-01,
               -1.37695312e-01, -7.66601562e-02,  6.25000000e-02,  4.98046875e-02,
               -1.91406250e-01, -6.83027344e-02,  2.27539062e-01,  5.88378906e-02,
               -3.24218750e-01,  5.41992188e-02, -1.35742188e-01,  8.17871094e-03,
               -5.24902344e-02, -1.74713135e-03, -9.81445312e-02, -2.86865234e-02,
                3.61328125e-02,  2.15820312e-01,  5.98144531e-02, -3.08593750e-01,
               -2.27539062e-01,  2.61718750e-01,  9.86328125e-02, -5.07812500e-02,
                1.78222656e-02,  1.31835938e-01, -5.35156250e-01, -1.81640625e-01,
                1.38671875e-01, -3.18546875e-01, -9.71679688e-02,  1.31835938e-01,
```

What does word2vec learn?

A small exercise

Go to: http://bionlp-www.utu.fi/wv_demo/ and explore the interface with the English word2vec model provided (10 minutes). Share your observations on what the model does afterwards.

Share your observations.

How to get from word to document?

```
import spacy

# Load the spacy model that we already installed in Chapter 2. This takes a few seconds.
%time nlp = spacy.load('en_core_web_md')
# process a sentence using the model
mydoc = nlp("Canada is a large country")
#Get a vector for individual words
#print(doc[0].vector) #vector for 'Canada', the first word in the text
print(mydoc.vector) #Averaged vector for the entire sentence
```

Out of vocabulary words

What do we do when we encounter a new word not seen in the training vocabulary?

- Leave it out of your feature vector calculation for the text (a common strategy)
- Randomly initialize a vector
- "Learn" a representation for a OOV word by randomly replacing words during training with a new word token.
- ....

A better way: FastText embeddings

- FastText deals with unknown words and sparsity in languages with rich morphology, by using subword models.
- Each word in fasttext is represented as itself plus a bag of constituent n-grams, with special boundary symbols $<$ and $>$ added to each word.
- For example, with n $=$ 3 the word where would be represented by the sequence ¡where¿ plus the character n-grams: $<$wh, whe, her, ere, re$>$
- Then an embedding is learned for each constituent n-gram (instead of words themselves), and the word where is represented by the sum of all of the embeddings of its constituent n-grams.
- A fasttext open-source library, including pretrained embeddings for 157 languages, is available at https://fasttext.cc.

using FastText

Using a pre-trained model

```
import fasttext
model = fasttext.train_unsupervised('data/fil9', thread=4)
[model.get_word_vector(x) for x in
    ["asparagus", "pidgey", "yellow"]]
```

source:
https://fasttext.cc/docs/en/unsupervised-tutorial.html

note: You can also use gensim to train or use pre-trained fasttext models.

Many more

- There is more work in this direction.
- .. in multiple languages
- ... cross lingual word embeddings too exist, where multiple language vocabulary is mapped into common space, so that words with same meaning in the two languages are closer together.
- There is also some work on generating paragraph/document embeddings directly instead of individual words. (doc2vec - check Notebook 8 in Practical NLP's Chapter 3 @Github)

Visualizing Embeddings

- Visualizing embeddings is an important goal in helping understand, apply, and improve these models of word meaning.
- But how can we visualize a (for example) 300-dimensional vector?

Visualizing Embeddings

- Visualizing embeddings is an important goal in helping understand, apply, and improve these models of word meaning.
- But how can we visualize a (for example) 300-dimensional vector?
  1. Check the most similar words to a given word, using some measure of distance (cosine)
  2. (More common): t-sne, which can project high dimensions into lower dimensions, like we saw earlier with sentiment words example.

Code examples: Notebooks 9 and 10 in Practical NLP's Chapter 3 @Github

Bias and Embeddings

- In addition to their ability to learn word meaning from text, embeddings, also reproduce the implicit biases and stereotypes that were latent in the text.
- Some past research showed that the closest occupation to 'man' - 'computer programmer' + 'woman' in word2vec embeddings trained on news text is 'homemaker', and that the embeddings similarly suggest the analogy 'father' is to 'doctor' as 'mother' is to 'nurse'.
- Recent research focuses on ways to try to remove this kind of biases in embedding representations.

Another fun exercise

Here is another visualization tool, which uses a different approach to build these word embeddings (GloVe): https://lamyiowce.github.io/word2viz/ Like before, spend about 10 minutes and explore this tool. See if you can also notice some biases/stereotypes in these!

Share your observations

Evaluating Embeddings

- Intrinsic evaluation: performance on similiarity datasets (word similarity ratings of humans and algorithms are compared).
- Extrinsic evaluation: use these representations as features for some task (e.g., text classification) and see if it is better than other representations (e.g., bag of words).

A summary so far:

- These embedding methods are very fast, efficient to train, and easily accessible in terms of both code and pre-trained models.
- Drawback: Word2vec like embeddings are static embeddings i.e., there is a fixed embedding for each word in the vocabulary.
  - What about the multiple senses of a word?
  - Will a word's representation remain same irrespective of the context of its use?

Solution: Contextual Word Embeddings

- Idea: A word's embedding representation need not necessarily capture all possible uses of the word. It should capture only what it means in that context.
- How?: One approach - train a neural network, that takes a word, looks at its left/right context in the sentence, and estimate a vector representation of this context.

Solution: Contextual Word Embeddings

- Idea: A word's embedding representation need not necessarily capture all possible uses of the word. It should capture only what it means in that context.
- How?: One approach - train a neural network, that takes a word, looks at its left/right context in the sentence, and estimate a vector representation of this context.
- So, we can start with word2vec like embeddings, and then let the neural network "tune" these representations based on context too!

Solution: Contextual Word Embeddings

- Idea: A word's embedding representation need not necessarily capture all possible uses of the word. It should capture only what it means in that context.
- How?: One approach - train a neural network, that takes a word, looks at its left/right context in the sentence, and estimate a vector representation of this context.
- So, we can start with word2vec like embeddings, and then let the neural network "tune" these representations based on context too!
- Depending on how this context is learnt, many different contextual word embeddings were proposed in the past 3 years.
- BERT, the now famous NLP model, is based on this idea of contextual embeddings.

(note: one of the teams will present more on this on Friday!)

How do we get these representations?

Language Model pre-training

- Language modeling is a task of predicting the probability of a given sequence of words in a language.
- It is useful in a range of application scenarios, from speech recognition to spelling correction.

How do we get these representations?

Language Model pre-training

- Language modeling is a task of predicting the probability of a given sequence of words in a language.
- It is useful in a range of application scenarios, from speech recognition to spelling correction.
- In recent years, neural language models became popular in NLP.
- Why?:

How do we get these representations?

Language Model pre-training

- Language modeling is a task of predicting the probability of a given sequence of words in a language.
- It is useful in a range of application scenarios, from speech recognition to spelling correction.
- In recent years, neural language models became popular in NLP.
- Why?: they just need large amounts of text (which is now available for several languages), and learn to predict this probability in an unsupervised manner.
- In this process, they also learn effective contextual representations of text.

BERT-1

Bidirectional Encoder Representations from Transformers

- BERT is perhaps the most popular language model in NLP right now.
- learns contextual representations of text by training on "unsupervised" prediction tasks - masked language modeling, and next sentence prediction.

BERT-1

Bidirectional Encoder Representations from Transformers

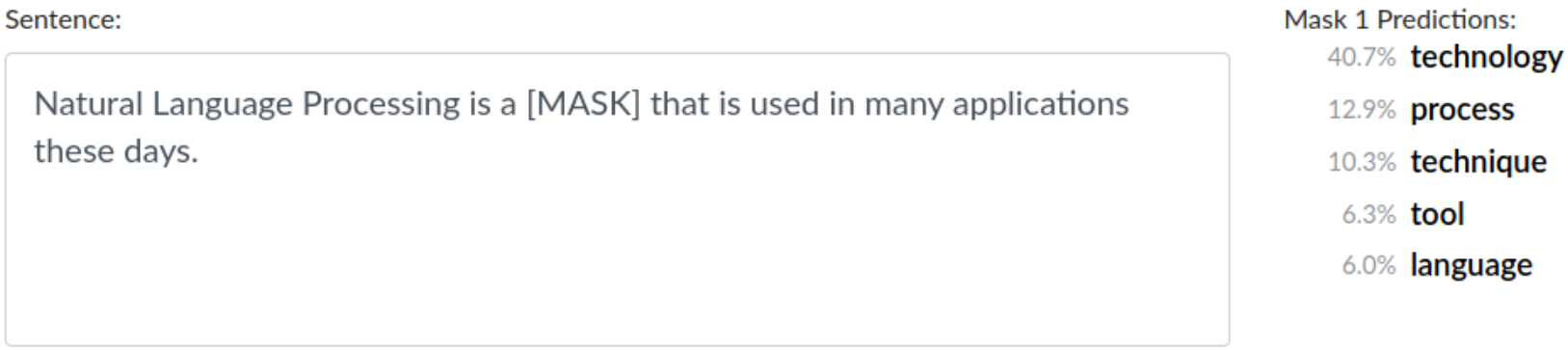
- BERT is perhaps the most popular language model in NLP right now.
- learns contextual representations of text by training on "unsupervised" prediction tasks - masked language modeling, and next sentence prediction.
- "Masked language modeling". It is similar to a fill-in-the-blank task, where a model learns to predict what the [MASK] token is, based on the context words surrounding it.
- How does learning like this capture anything beyond one sentence? next sentence prediction task: learning to predict the next sentence, given a sentence.

BERT-1

Bidirectional Encoder Representations from Transformers

- BERT is perhaps the most popular language model in NLP right now.
- learns contextual representations of text by training on "unsupervised" prediction tasks - masked language modeling, and next sentence prediction.
- "Masked language modeling". It is similar to a fill-in-the-blank task, where a model learns to predict what the [MASK] token is, based on the context words surrounding it.
- How does learning like this capture anything beyond one sentence? next sentence prediction task: learning to predict the next sentence, given a sentence.
- These are shown to be better representations of text than previous methods in various use cases.

BERT -Masked Language Model Demo



https://demo.allennlp.org/masked-lm

What do they learn?

- Depends on the model's neural network architecture.
- Various research studies showed that such contextual representations learn different kinds of syntactic properties (even though they are not explicitly trained for that).
- "probing" models to understand what they learn is an active area of research in NLP now.

How can we use these learned representations?

- As feature extractors: instead of using static embeddings, or BOW/TF-IDF or hand crafted representations, these representations can directly be used as features for downstream tasks such as text classification, information extraction etc.

How can we use these learned representations?

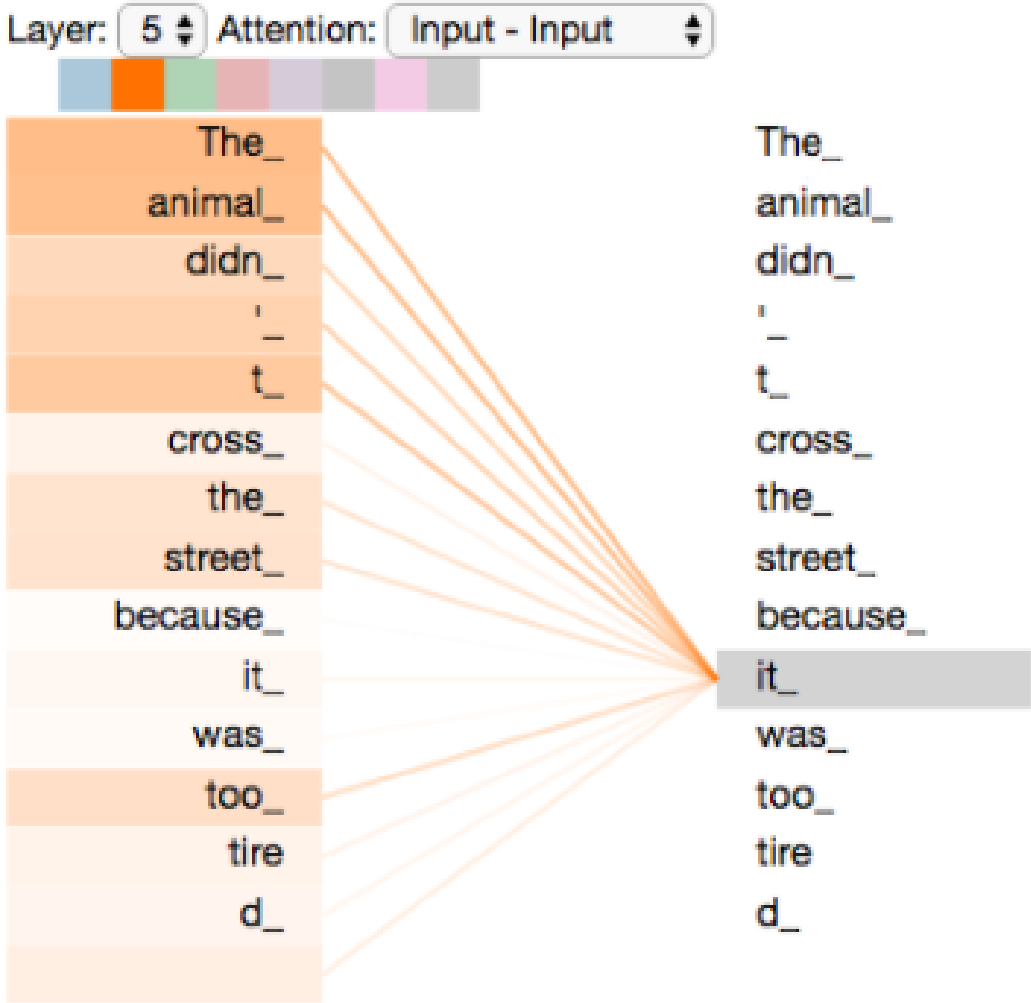
- As feature extractors: instead of using static embeddings, or BOW/TF-IDF or hand crafted representations, these representations can directly be used as features for downstream tasks such as text classification, information extraction etc.
- For fine-tuning: start with these representations (which were learnt from training on language modeling tasks) and let a neural network "tune" them to suit the current task (e.g., text classification).

What is fine-tuning?

- The representations learned from training a BERT model can be used as inputs/features to any other task (e.g., text classification)
- When we are using a neural network to learn this task, it transforms these input representations to suit that particular task.
- The pre-trained model's weights are then altered ("fine-tuned") while training for the task i.e., this pre-trained architecture is then "retrained" to suit this task.

What is happening during fine-tuning?

Let us start with "before" fine-tuning



Inferring association between tokens using attention. source: http://jalammar.github.io/illustrated-transformer/

What happens during fine-tuning?

A case study of using BERT to fine tune "aspect based sentiment analysis"

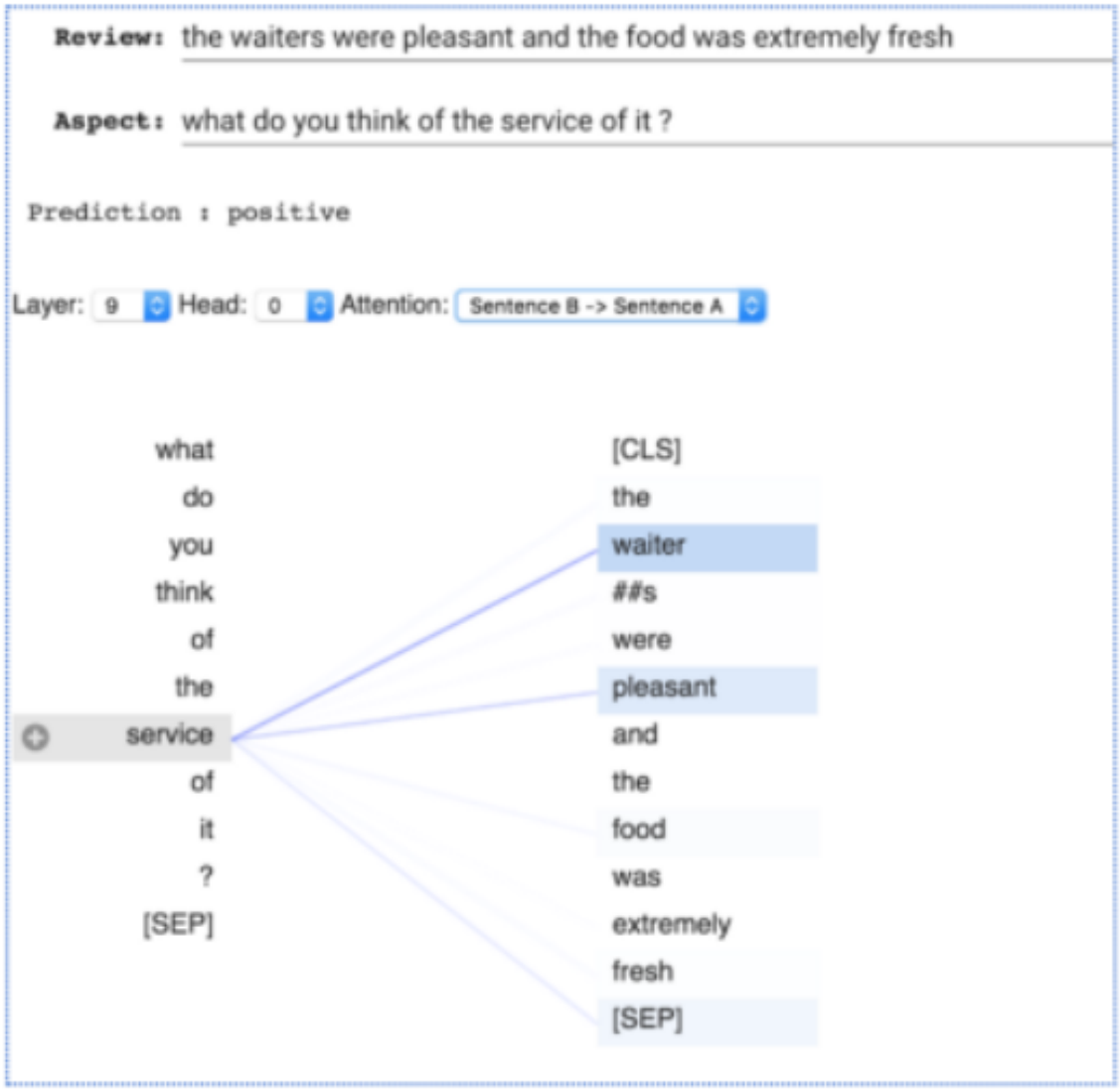
Review Sentence	Price	Food
The restaurant was too expensive	Negative	None
The restaurant was expensive, but the menu was great	Negative	Positive

Review Sentence	Question	Sentiment
The restaurant was too expensive	What do you think of the price of it ?.	Negative
The restaurant was too expensive	What do you think of the food of it ?.	None
The restaurant was expensive, but the menu was great	What do you think of the price of it ?.	Negative
The restaurant was expensive, but the menu was great	What do you think of the food of it ?.	Positive

Aspect-based sentiment analysis as QA — https://arxiv.org/pdf/1903.09588v1.pdf

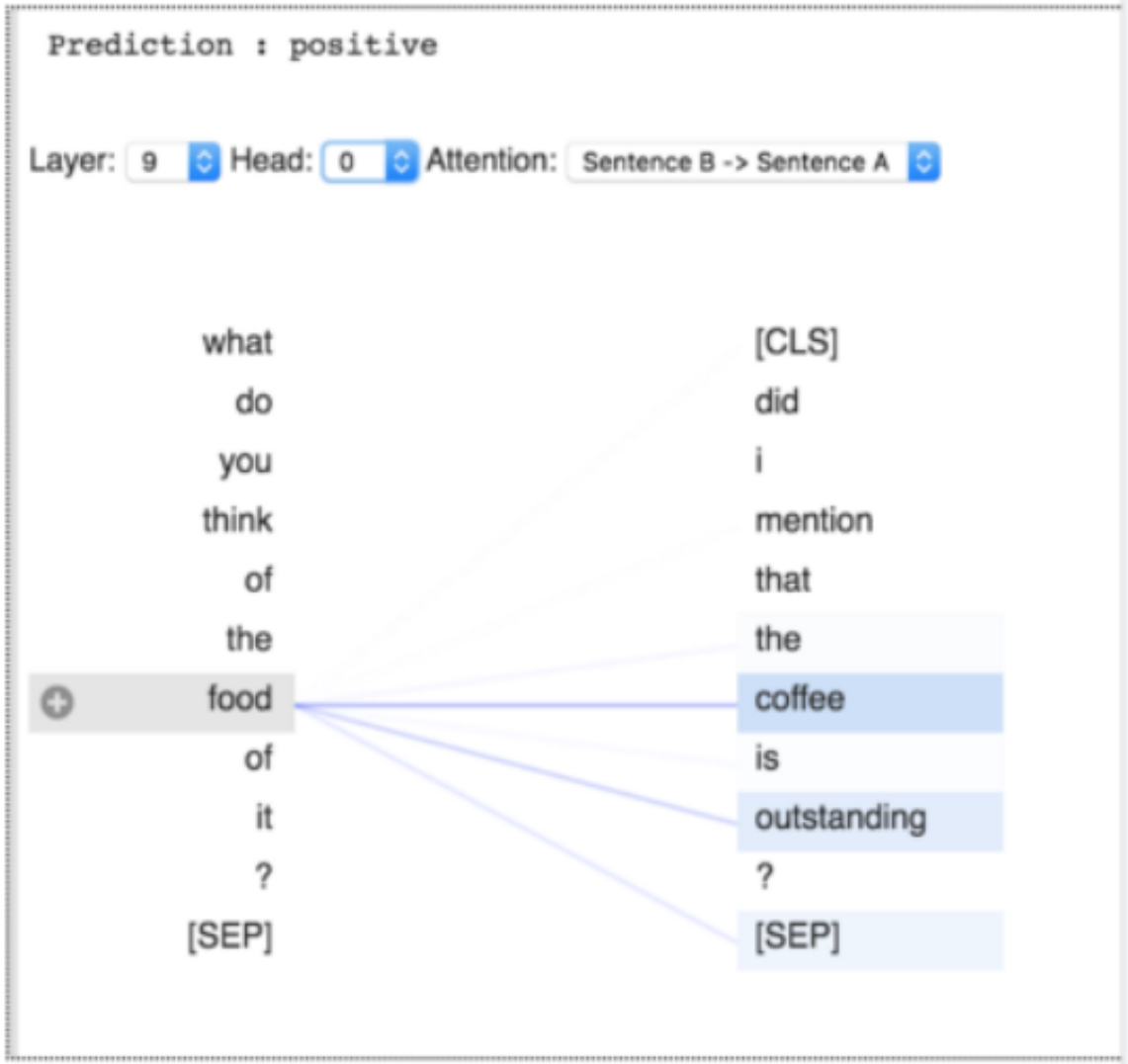
"What does a fine-tuned bert model look at?"

What happens during fine-tuning?



"What does a fine-tuned bert model look at?"

What happens during fine-tuning?



"What does a fine-tuned bert model look at?"

Does this really work in practice?

A small exercise

Go to your breakout rooms, and browse through the BERT based demos at: https://demos.pragnakalp.com/. Pick one of them, and play with it for sometime (10 minutes). When you back, each team can share their observations.

Share your observations.

Beyond the original BERT

- Improvements to model architecture (e.g., RoBERTa)
- Language specific BERT models (e.g., CamemBERT)
- Multilingual BERT models (e.g., mBERT)
- Compressed BERT (e.g., DistillBERT)

.....

Things to keep in mind

- It is expensive and resource intensive to train your own BERT or any other such language model.
- "Training GPT-3 would cost over $4.6M using a Tesla V100 cloud instance." (source)
- Using the pre-trained model + fine tuning it is how it is used in general purpose cases.
- However, even the pre-trained models are super large, and fine-tuning can also get resource intensive (and there may be challenges with deploying these).

Things to keep in mind

- It is expensive and resource intensive to train your own BERT or any other such language model.
- "Training GPT-3 would cost over $4.6M using a Tesla V100 cloud instance." (source)
- Using the pre-trained model + fine tuning it is how it is used in general purpose cases.
- However, even the pre-trained models are super large, and fine-tuning can also get resource intensive (and there may be challenges with deploying these).
- DistillBERT and other such recent developments address this problem to some extent.

Using embedding representations for text classification - examples

Source: Chapter 4 of Practical NLP book (its Github repo, that is)

Using word2vec

```
# Creating a feature vector by averaging all embeddings for all sentences
def embedding_feats(list_of_lists):
    DIMENSION = 300
    zero_vector = np.zeros(DIMENSION)
    feats = []
    for tokens in list_of_lists:
        feat_for_this =  np.zeros(DIMENSION)
        count_for_this = 0
        for token in tokens:
            if token in w2v_model:
                feat_for_this += w2v_model[token]
                count_for_this +=1
        feats.append(feat_for_this/count_for_this)
    return feats


train_vectors = embedding_feats(texts_processed)
print(len(train_vectors))
```

```
3000
```

(full code)

Then what?

- Once you extracted these features, it is the same as any other classification example we saw in earlier classes.
- Note: In this example, we used gensim (in the textbook).
- But the same representation can be obtained with a one liner in spacy (without having to write the document level aggregation code).

using FastText

```
from fasttext import train_supervised
model = train_supervised(input=train_file, label="__class__",
                         lr=1.0, epoch=75, loss='ova',
                         wordNgrams=2, dim=200, thread=2,
                         verbose=100)

results = model.test(test_file,k=1)
print(f"Test Samples: {results[0]} Precision : {results[1]*100:2.4f} Recall: {results[2]*100:2.4f}")
```

(Full code)

What's different?

- fasttext library itself natively supports classification task.
- It is also blazing fast. So, for larger datasets, where something like logistic regression may take forever to train, fasttext trains within a minute.

Training Doc2Vec (my favorite even in BERT era)

```
#prepare training data in doc2vec format:
train_doc2vec = [TaggedDocument((d), tags=[str(i)])
                 for i, d in    enumerate(train_data)]
#Train a doc2vec model to learn tweet representations.
model = Doc2Vec(vector_size=50, alpha=0.025,
                min_count=5, dm =1, epochs=100)
model.build_vocab(train_doc2vec)
model.train(train_doc2vec,
            total_examples=model.corpus_count,
            epochs=model.epochs)
model.save("d2v.model")
print("Model Saved")
```

(Source code)

Using Doc2Vec

```
#Infer the feature representation for training and test data using the trained model
model= Doc2Vec.load("d2v.model")
#infer in multiple steps to get a stable representation.
train_vectors =  [model.infer_vector(list_of_tokens, steps=50)
                  for list_of_tokens in train_data]
test_vectors = [model.infer_vector(list_of_tokens, steps=50)
                for list_of_tokens in test_data]

#Use any regular classifier like logistic regression
from sklearn.linear_model import LogisticRegression
myclass = LogisticRegression(class_weight="balanced")
#because classes are not balanced in training data!
myclass.fit(train_vectors, train_cats)
preds = myclass.predict(test_vectors)
print(classification_report(test_cats, preds))
```

(Source code)

using BERT: no fine tuning

```
from transformers import AutoTokenizer, AutoModel, pipeline
model = "bert-base-uncased"
#all models at: https://huggingface.co/models
model = AutoModel.from_pretrained(modelpath)
tokenizer = AutoTokenizer.from_pretrained(modelpath)
nlp = pipeline('feature-extraction')
sample_text = "This is a sample sentence"
feat_vector = nlp(sample_text)[0][0]
```

- do this for all your training data, and use it with some classifier!

BERT: fine-tuning

```python
from transformers import DistilBertForSequenceClassification, Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=3,              # total number of training epochs
    per_device_train_batch_size=16,  # batch size per device during training
    per_device_eval_batch_size=64,   # batch size for evaluation
    warmup_steps=500,                # number of warmup steps for learning rate scheduler
    weight_decay=0.01,               # strength of weight decay
    logging_dir='./logs',            # directory for storing logs
    logging_steps=10,
)

model = DistilBertForSequenceClassification.from_pretrained("distilbert-base-uncased")

trainer = Trainer(
    model=model,                         # the instantiated 🤗 Transformers model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=val_dataset             # evaluation dataset
)

trainer.train()
```

source: https://huggingface.co/transformers/custom_datasets.html

BERT: fine-tuning

Save and use this model

```
save_directory = "/saved_models"
model.save_pretrained(save_directory)
tokenizer.save_pretrained(save_directory)

loaded_tokenizer = DistilBertTokenizer.from_pretrained(save_directory)
 loaded_model = TFDistilBertForSequenceClassification.from_pretrained(save_directory)

 predict_input = loaded_tokenizer.encode(test_text,
                            truncation=True,
                            padding=True,
                            return_tensors="tf")

  output = loaded_model(predict_input)[0]

 prediction_value = tf.argmax(output, axis=1).numpy()[0]
```

Source Code

BERT: fine-tuning - other options

In our book, we showed fine-tuning before hugging face's simple interface was released. Two examples are below:

1. Using ktrain, a light weight deep learning library
2. using keras and pytorch

Other useful things todo

- Visualizing embeddings (using tools such as t-sne, bertviz etc) to understand what the models are learning.
- Interpreting model predictions (using tools such as LIME, SHAP, Anchors etc) and explaining why a prediction was made by the model

Resources for learning further:

- "Embeddings in NLP" book and recent (Dec 2020) tutorial at COLING 2020 (video is in the website)
- Chapters 6, 7 in "Speech and Language Processing" 3rdEdition
- "Illustrated *" posts by Jay Alammar
- A Primer in BERTology: What we know about how BERT works

Plan for Friday's class

- Teams presenting:
  - Yixuan, Kuan and Ting-Yu: "The Multilingual Amazon Reviews Corpus".
  - Nelly and Hebah: " Data augmentation using machine translation for fake news detection in the Urdu language."
  - Leyre, Lorena, Mourhaf and Siena: " Contextual word representations: putting words into computers".
- Rest of the time: I will try to summarize a some relevant stuff to the papers presented, which I think is useful for you.
- Potentially: Some group exercises.