Natural Language Processing

Class 3: Early Language Modeling

Adam Faulkner

September 9, 2025

Why are language models important?

- Language models (LMs): models that assign a probability to each possible next word
- What is the most likely word in the blank? *The water of Walden Pond is so beautifully* _____.
- Modern language modelling was born at IBM in the early 1990's
- Why is the task of next-word prediction useful?
  - **Generation**: choosing contextually better words. *Their are two midterms* becomes *There are two midterms* since $p(are|There)$ is higher than $p(are|Their)$
  - **Speech-to-text**: *I will be back soonish* and not *I will be bassoon dish*, since $p(back\ soonish|I\ will\ be)$ has a higher probability.
  - **Large Language Models (LLMs)**: Contemporary LLM-based NLP, at its core, is based on next-word-prediction

N-grams

- **An n-gram is a sequence of *n* words**
- 2-gram (bigram) is a two-word sequence of words (*The water*, or *water of*),
- 3-gram (trigram) is a three-word sequence of words (*The water of*, or *water of Walden*)
- ...and so on for all n-grams

N-grams

- Generalizing, the n-gram modeling task is the task of computing $P(w|h)$, the probability of a word w given some history h
- E.g., history h is *The water of Walden Pond is so beautifully* and we want to know the probability that the next word is *blue*

$$P(\texttt{blue}|\texttt{The water of Walden Pond is so beautifully})$$

- Hard to calculate this probability using a corpus of token sequences since language is *creative* (i.e., new sentences are being created all the time) and this particular sequence might not be in the corpus

The Markov assumption

- Instead we approximate the probability that the next word will be *blue* and instead calculate

$$p(\texttt{blue}|\texttt{beautifully})$$

which is the **Markov assumption** of language modeling:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1})$$

- We compute the probability of a complete word sequence via

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1})$$

Estimating n-gram probabilities using MLE

- **Maximum likelihood estimation (MLE)**
- Get MLE estimate for the parameters of an n-gram model by getting counts from a corpus, and normalizing the counts so that they lie between 0 and 1.
- To compute a particular bigram probability of a word w_n given a previous word w_{n-1}, well compute the count of the bigram $C(w_{n-1}w_n)$ and normalize by the count of w_{n-1}

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

So, given a three sentence corpus <s>*I am Sam*</s>. <s>*Sam I am*</s>. <s>*I do not like green eggs and ham.*</s>, where `<s>` and `<>` are markers for, respectively, the beginning and end of sentences, we can calculate the bigram probability of *I am* as

$$P(\texttt{am}|\texttt{I}) = \frac{C(\texttt{I am})}{C(\texttt{I})} = \frac{2}{3} = 0.67$$

Evaluating Language Models

- **Extrinsic evaluation**: Embed the LM in an application and measure how much the application improves. If, for example, the application is a speech-to-text system, the LM would be evaluated by simply reporting out the accuracy improvement of the speech-to-text system relative to some baseline
- **Intrinsic evaluation**: Measures the quality of a model independent of any application

Perplexity

- Perplexity, or the exponentiated Average Negative Log Likelihood (NLL), is the most widely used intrinsic LM evaluation metric
- Measure of the models fluency and coherence
- Gives insights into how well the model generalizes over unseen data. A lower perplexity over unseen samples means that the model can generalize well over out-of-distribution samples
- **Note!A measure of confusion so *lower* is better**

Perplexity

- We encountered the Negative Log Likelihood (NLL) in the last lecture. Perplexity averages the NLL and then exponentiates the result

$$AverageNLL = -\frac{1}{N}\Sigma_{i=1}^{N} logP(w_i|w_1, w_2, ..., w_{i-1})$$

where N is the total number of words in the held-out dataset and $P(w_i|w_1, w_2, ..., w_{i-1})$ is the probability of word w_i given the previous words $w_1, w_2, ..., w_{i-1}$

$$Perplexity = e^{AverageNLL}$$

Perplexity

- Suppose we have an LM with a vocabulary of size 100, and the generated sequence is

```
John bought apples from the market
```

- Assume that we've already calculated the probabilities of the n-grams in this sequence using the procedure from the previous slides:

$$
\begin{aligned}
&p(John) = 0.1 \\
&p(bought|John) = 0.4 \\
&p(apples|bought) = 0.3 \\
&p(from|apples) = 0.5 \\
&p(the|from) = 0.6 \\
&p(market|the) = 0.7
\end{aligned} \tag{1}
$$

Perplexity

- The perplexity of this sequence is then

$$
\begin{aligned}
&P(\texttt{John bought apples from the market}) = \\
&0.1 \times 0.4 \times 0.3 \times 0.5 \times 0.6 \times 0.7 = 0.00252 \\
&AverageNLL = -log(0.00252)/6 = 0.99725 \\
&exp(0.99725) = \boxed{\mathbf{2.71}}
\end{aligned} \tag{2}
$$

Sampling from n-gram models to generate new text

- Another way to determine the quality of an LM is to sample sequences of text from the model, e.g., generate sequences (such as sentences) according to their likelihood as defined by the model
- In general, the power of n-gram models increases with n-gram size
- Example: Generating fake Shakespeare using 1-,2-,3-, and 4-gram LMs.

1 gram	–To him swallowed confess hear both. Which. Of save on trail for are ay device and rote life have –Hill he late speaks; or! a more to leg less first you enter
2 gram	–Why dost stand forth thy canopy, forsooth; he is this palpable hit the King Henry. Live king. Follow. –What means, sir. I confess she? then all sorts, he is trim, captain.
3 gram	–Fly, and will rid me these news of price. Therefore the sadness of parting, as they say, 'tis done. –This shall forbid it should be branded, if renown made it empty.
4 gram	–King Henry. What! I will go seek the traitor Gloucester. Exeunt some of the watch. A great banquet serv'd in; –It cannot be but so.

Sampling from n-gram models to generate new text

- How do we capture word meaning in a way that is useful for NLP applications?
- We need to be able to capture the fact that
  - some words have similar associations (*cat* is similar to *dog* since they are both domesticated pets),
  - some words are antonyms (*cold* is the opposite of *hot*)
  - some have positive connotations (*happy*) while others have negative connotations (*sad*)
  - and that the meanings of *buy, sell,* and *pay* offer differing perspectives on the same underlying purchasing event. (If I buy something from you, youve probably sold it to me, and I likely paid you.)
- More generally, a model of word meaning should allow us to draw inferences to address meaning-related tasks like question-answering or dialogue

Word senses

- Consider the word *mouse* in English. This word can be used in two different ways or *senses*
  1. any of numerous small rodents...
  2. a hand-operated device that controls a cursor...
- Polysemy (many senses) can make interpretation difficult (is someone who types ”mouse info” into a search engine looking for a pet or a tool?)
- This is an open problem and is called the problem of *word-sense disambiguation*

Semantic fields

- The notion of **word similarity** is very useful in larger semantic tasks. Knowing how similar two words are can help in computing how similar the meaning of two phrases or sentences are, a very important component of tasks like question answering, paraphrasing, and summarization.
- While the notion of **synonymy** captures groups of words with nearly identical senses (*couch/sofa*, *car/automobile*), **semantic fields** capture relations between words which cover a particular semantic domain and bear structured relations with each other

Semantic fields

Examples of semantic fields

- Hospitals: *surgeon, scalpel, nurse, anesthetic, hospital*
- Restaurants: *waiter, menu, plate, food, chef*
- Houses: *door, roof, kitchen, family, bed*

Semantic fields

- How do we discover and encode semantic fields in our NLP models?
- **Vector semantics**: the standard way to represent word meaning in NLP
- Relies on the Firth's *distributional hypothesis* of meaning:

"You shall know a word by the company it keeps."

In other words, words that tend to occur in the same context are likely to be in the same semantic field

The distributional hypothesis

- Are the words *ongchoi* and *chard* in the same semantic field? Let's take a look at a corpus and see if they occur in similar contexts
  - ongchoi is delicious sauteed with garlic.
  - ongchoi is superb over rice
  - ...ongchoi leaves with salty sauces...
  - ...chard sauteed with garlic over rice...
  - ...chard stems and leaves are delicious...
  - ...collard greens and other salty leafy greens

  Even if we haven't encountered the words *ongchoi* and *chard* before, their shared contexts puts them in the same *leafy greens* semantic field

Embeddings

- **Embeddings**: Words are represented as a point in a multidimensional semantic space that is derived from the distributions of word neighbors
- Embedding vectors can consist of hundreds of dimensions. To visualize how these embeddings group via semantic fields we can project them into 2D space via tSNE:

word2vec

- The best-known procedure for generating word embeddings from massive text corpora is the *word2vec* algorithm
- *word2vec* embeddings are static embeddings, meaning that the method learns one fixed embedding for each word in the vocabulary
- Since they're static, they are typically distributed as simple text files with NLP and ML libraries:

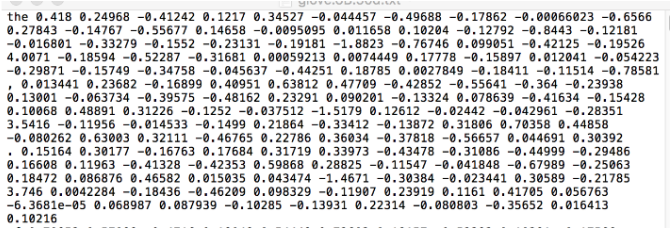


Figure 1: word2vec embedding for the word "the"

Computing embeddings via the skip-gram-with-negative-sampling procedure

- *word2vec* utilizes two learning algorithms to compute embeddings, *skip-gram with negative sampling (SGNS)* and *continuous bag-of-words*. In this course, we'll be reviewing only SGNS
- The intuition of *word2vec* is that instead of counting how often each word *w* occurs near, say, *apricot*, we'll instead train a classifier on a binary prediction task that answers the question "Is word *w* likely to show up near apricot?" We dont actually care about this prediction task; instead well take the resulting learned classifier weights and save them as the word embeddings
- The learning itself takes place via **self-supervision** which we discussed last week

Self-supervision

- **Self-supervision**: Use running text as implicitly supervised training data for the classification task; 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?"

Computing embeddings via the skip-gram-with-negative-sampling procedure

The intuition of the skip-gram is:

1. Treat the target word and a neighboring 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.

Computing embeddings via the skip-gram-with-negative-sampling procedure

Source Text	Training Samples generated from source text			
I will have orange juice and eggs for breakfast	(will, I)	(will, have)	(will, orange)	
I will have orange juice and eggs for breakfast	(have, I)	(have, will)	(have, orange)	(have, juice)
I will have orange juice and eggs for breakfast	(orange, will)	(orange, have)	(orange, juice)	(orange, and)
I will have orange juice and eggs for breakfast	(juice, have)	(juice, orange)	(juice, and)	(juice, eggs)
I will have orange juice and eggs for breakfast	(and, orange)	(and, juice)	(and, eggs)	(and, for)
I will have orange juice and eggs for breakfast	(eggs, juice)	(eggs, and)	(eggs, for)	(eggs, breakfast)
I will have orange juice and eggs for breakfast	(for, and)	(for, eggs)	(for, breakfast)	

Figure 2: Training samples generated for skip-gram learning

Computing embeddings via the skip-gram-with-negative-sampling procedure

Skip-gram embeddings are learned in the following way after first collecting a corpus of text (newspaper, books, scraped text from the internet) and choosing a vocabulary size N

1. Assign a random embedding vector for each of the N vocabulary words
2. Iteratively shift the embedding of each word w to be more like the embeddings of words that occur nearby in the texts, and less like the embeddings of words that don't occur nearby

Steps 1 and 2 are learned via a logistic binary classifier trained on both positive and negative examples

Computing embeddings via the skip-gram-with-negative-sampling procedure

- Consider the following training example:

```
... lemon,  a [tablespoon of apricot jam,      a] pinch ...
               c1         c2    w     c3       c4
```

- This example has a target word w (*apricot*), and 4 context words in the $L = ś2$ window, resulting in 4 positive training instances (on the left below). For each of these training instances well also create k negative samples, each consisting of the target w plus a "noise word" c_{neg}. A noise word is a random word from the lexicon, constrained not to be the target word w

positive examples +

w	c_{pos}
apricot	tablespoon
apricot	of
apricot	jam
apricot	a

negative examples -

w	c_{neg}	w	c_{neg}
apricot	aardvark	apricot	seven
apricot	my	apricot	forever
apricot	where	apricot	dear
apricot	coaxial	apricot	if

Training a classifier on skip-grams

- *word2vec* embeddings are generated by training a probabilistic classifier that, given a test target word w and its context window of L words $c_{1:L}$, assigns a probability based on how similar this context window is to the target word. The probability is based on applying the sigmoid function to the dot product of the embeddings of the target word with each context word.
- Let's go through this step-by-step

Training a classifier on skip-grams

- Our goal is to train a classifier such that, given a tuple (w, c) of a target word w paired with a candidate context word c (for example *(apricot, jam)*, or perhaps *(apricot, aardvark)*) it will return the probability that c is a real context word (true for *jam*, false for *aardvark*):

$$P(+|w, c)$$

  The probability that word c is not a real context word for w is just 1 minus $P(+|w, c)$

$$P(|w, c) = 1P(+|w, c)$$

- How to compute the probability P? P is based on **embedding similarity**: a word is likely to occur near the target if its embedding vector is similar to the target embedding

Calculating embedding similarity using the dot product

- Two vectors are similar if they have a high **dot product**
- Dot product: takes two equal-length sequences of numbers and return a single number.

$$Similarity(w, c) \approx c \cdot w$$

- The dot product $c \cdot w$ is not a probability — to turn it into a probability, we pass it through the sigmoid function $\sigma(x)$, which we learned about in the last class

The loss function used during skip-gram training

- Given the set of positive and negative training instances, and an initial set of embeddings, the goal of the learning algorithm is to adjust those embeddings to
  1. Maximize the similarity of the target word, context word pairs (w, c_{pos}) drawn from the positive examples
  2. Minimize the similarity of the (w, c_{neg}) pairs from the negative examples

The loss function used during skip-gram training

- These two learning goals can be expressed as the following loss function:

$$-log[P(+|w, e_{pos} \prod_{i=1}^{k} P(-|w, c_{neg_i}]$$

- In words: "Maximize the dot product of the word with the actual context words, and minimize the dot product of the word with the k negative sampled non-neighbor word"
- The training itself is standard gradient descent: The word and context matrices are initialized with random numbers and training occurs by traversing the corpus and using gradient descent to move these matrices so as to minimize the loss

Using vector arithmetic to explore the properties of embeddings

- A surprising property of word vectors is that complex relationships between words can often be expressed as vector arithmetic. Word analogy problems such as *a is to b as a* is to what?* can be solved in terms of embeddings:

  apple is to tree as grape is to what?

  Using vector arithmetic we can determine the answer as

  $$\vec{tree} - \vec{apple} + \vec{grape} \approx \vec{vine}$$

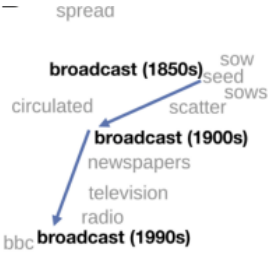
- The most famous example of this arithmetic at work is

  $$\vec{king} - \vec{man} + \vec{woman} \approx \vec{queen}$$

  Subtracting the embedding for *man* from the embedding for *king* and adding the *woman* embedding results in the *queen* embedding!

Viewing historical changes in word usage via embeddings

- Embeddings can also be a useful tool for studying how meaning changes over time, by computing multiple embedding spaces, each from texts written in a particular time period



Using embeddings in downstream NLP tasks

- **Search**. Search queries, such as *best place to get coffee*, are transformed into embeddings (if just word embeddings are being used, the average is taken) and texts in the closest embedding space (as determined by vector similarity) are returned as results: *Cafe Lalo*, for example, is closer in embedding space to *best place to get coffee* than *Sushi Nakazawa*
- **Bootstrapping NNs**. The weights of traditional NNs are initialized with random numbers. Instead, the NNs are initialized with embeddings and the NN is trained in the traditional way for classification tasks such as sentiment analysis and text classification

Next class: Sept 26

- Assignment 1 due
- Neural Networks Deep Learning
  - Perceptrons
  - Deep Feedforward Neural Networks
  - Gated Architectures: RNNs, LSTMs
- Goodfellow, Bengio, & Courville: Introduction
- Jurafsky & Martin Chapter 7: Neural Networks and Neural Language Models
- Jurafsky & Martin Chapter 9: RNNs and LSTMs