Assignment 2 – Language Models and Embeddings

CS499 - Intro to NLP
Antonis Anastasopoulos

January 2021

Due Date: 2/19/2021 (eod)
Submission instructions: see at the end of the assignment.

Make sure to download the assignment materials from the class page.

1 Language Modeling (7 credits total)

Implement an n-gram count-based language model, and answer all questions below.

It will be easier for the rest of this exercise if you implement a general class of models that receives n as a parameter and computes the necessary probabilities accordingly.

[2 credits] (a) Read in the Reuters train corpus, and report the following unsmoothed unigram, bigram, and trigram probabilities:

n-gram	probability	value
free	p(free)	
market	p(market)	
language	p(language)	
free market	p(market | free)	
market shortage	p(shortage | market)	
president reagan	p(reagan | president)	
carter administration	p(administration | carter)	
net profit	p(profit | net)	
net loss	p(loss | net)	
programming language	p(language | programming)	
endangered language	p(language | endangered)	
the language of	p(of | the language)	
the paris accord	p(accord | the paris)	

[**1 credit**] **(b)** Add an option to do simple add-k smoothing, so that your model can handle unknown n-grams. Report the updated probabilities when using **add-1** smoothing for the above n-grams.

[**2 credit**] **(c)** Add a function to compute the perplexity of your model over an unseen dataset. Read in the Reuters test set, and report the perplexity of n-gram models for $n \in [2, 6]$ (using add-one smoothing).

[**2 credits**] **(d)** An adversary (named Eve) scrambled the original sentences in the `scrambled.7.txt` file. For example, the first sentence in the file is "You this why do even would ?", although the original sentence was "Why would you even do this ?" Use your best-performing language model to find the most likely un-scrambling of the scrambled sentences.

Hint: a well-formed sentence should have higher probability than an ill-formed (scrambled) sentence.

Your code should produce the unscrambled version of each sentence in the `scrambled.7.txt` file, and write them out, one sentence per line, in the same order, in a file named `unscrambled.7.txt`.

In your report, explain how you implemented the unscrambling, and also report the accuracy you obtain in unscrambling these sentences. The produced `unscrambled.7.txt` file should be part of your assignment submission.

You can evaluate how good your model is by running the `check_unscrambling.py` script. Run as follows:

```
#python3 check_unscrambling.py [your_output] [gold file]
python3 check_unscrambling.py unscrambled.7.txt unscrambled-gold.7.
    txt
```

Note: the evaluation script checks your output against the `unscrambled-gold.7.txt` file (which has the correct answers). It simply compares each line and checks whether they are identical or not – hence, it is important you write the sentences in order. Ideally, you'll never look at the gold unscrambled sentences.

2 Embeddings and Word Similarity (3 credits total)

[**1 credit**] Download the English fasttext embeddings (1 million words, trained on Wikipedia and other datasets) from here:
`https://dl.fbaipublicfiles.com/fasttext/vectors-english/wiki-news-300d-1M-subword.vec.zip`
You can simply run:

```
wget [above link] .
```

In Python3, you can read in the embeddings using this piece of code:

```
import io

def load_vectors(fname):
    fin = io.open(fname, 'r', encoding='utf-8', newline='\n',
    errors='ignore')
    n, d = map(int, fin.readline().split())
    data = {}
    for line in fin:
        tokens = line.rstrip().split(' ')
        data[tokens[0]] = map(float, tokens[1:])
    return data
```

Implement a function that computes the cosine distance between two vectors. You may use packages like scipy or similar. Complete the following table with the cosine similarity of the following word pairs:

word1	word2	cosine distance
horseradish	spinach	
lingonberry	strawberries	
pikachu	charizard	
charizard	charmander	
math	algorithm	

[**2 credits**] Also download the pretrained Glove embeddings from here: `http://nlp.stanford.edu/data/glove.6B.zip`.
You can simply run:

```
wget http://nlp.stanford.edu/data/glove.6B.zip .
```

After extracting the file, you will have 4 different embeddings, with 50, 100, 200, and 300 dimensions. The file format should be the same as the fasttext embeddings.

You are given a file `analogy.txt`, with analogy data. Each line has four tab-separated columns, each with a single word, as follows:

```
[word 1]    [word 2]    [word 3]    [word 4]
pen ballpoint rodent  nanosecond
bird  robin appliance refrigerator
...
```

The way to interpret this line is as follows: "word1 is to word2 as word3 is to word4". For example, "bird is to robin as appliance is to refrigerator".

Unfortunately, Eve has intervened again and added noise to the file. She maliciously substituted the fourth word in *some* lines, like in the example in line 2 above. Your goal is to identify which lines represent true analogies and which have been tampered with by Eve. Use your word embeddings (either fasttext, or any of the glove, or any combination of these) to implement the intuition behind the word analogy task, and try to identify which lines are "Correct" and which lines are "Wrong".

Submit a file named `analogy-predictions.txt` with a single word in each line: your prediction of whether the corresponding line in the `analogy.txt` file is "Correct" or "Wrong". Your `analogy-predictions.txt` file should have exactly 1000 lines. **In your report, explain (a) how you modeled the analogy task, and (b) how you decided on how to make the final prediction. Submitting the file (1 credit) and answering the above two questions (1 credit) will give you all credits. Additionally, we will evaluate your outputs and the top-3 performing submissions will receive 2 extra credits.**

Hint 1: You will probably have to decide on a threshold between correct and wrong.
Hint 2: Creating some "development" examples by hand so that you can evaluate different versions of your models is probably a good idea. Labeling all 1,000 examples by hand is, on the other hand, not a good use of your time...
Hint 3: Eve tampered with more than 20% of the lines, but less than 50% of them.

[BONUS] Beam-search for unscrambling (3 credits)

One way to do unscrambling for the last part of exercise 1 is to simply try all possible combinations of words. For example, for the scrambled sentence "is this suboptimal." we will compute all five possible unscramblings: "suboptimal this is.", "suboptimal is this.", "this suboptimal is.", "this is suboptimal.", "is suboptimal this.". Then we will use our LM to score each of these options, and return the one with the highest probability as our answer.

However, this is clearly, ahem, suboptimal. For a sentence with length 3, there are $3! = 3 \times 2 \times 1$ options. For a sentence of length 10, there would be 3,628,800 options!

One less accurate but more efficient solution could use **approximate search**. In this exercise we will implement two search functions: **greedy search** and **beam search**.

For each of the two exercises below, report the

[**1 credit**] **Greedy search** Greedy search is quite simple. We will search for the most likely unscrambling by generating the answer one word at a time. In each step, we will pick from our pool of words the most likely one to be the continuation of what we have generated so far.

Let's say we have our initial bag of words: {is,this,suboptimal}, and that (for simplicity) we have a bigram model. Before we start our search, the only history we have is the beginning of sentence symbol $h = \langle s \rangle$. Our process will be as follows:

1. pool: {is,this,suboptimal}, $h = \langle s \rangle$

Get the probabilities of all possible continuations, using the bigram probabilities $p(\text{is}|\langle s\rangle)$, $p(\text{this}|\langle s\rangle)$, and $p(\text{suboptimal}|\langle s\rangle)$.
Pick the highest one (let's say is the one with "this") and update our pool and history.

2. pool: {is,suboptimal}, $h = \langle s\rangle$ this
Get the probabilities of all possible continuations, using the bigram probabilities $p(\text{is}|\text{this})$ and $p(\text{suboptimal}|\text{this})$.
Pick the highest one and update our pool and history.

3. Continue until the pool is empty, and return the history.

Beam search [2 credits] Greedy search is prone to errors, because it cannot recover from any errors it might make in the beginning of the sentence. One solution is to use **beam-search**, a search algorithm very commonly used in NLP. The basic idea is that again we create the output one word at a time, but we keep multiple possible "paths" that we will keep track of.[1] The implementation is based on the Viterbi algorithm that we will introduce in future classes, but basically it works as follows.

Pseudocode:

```
Start: CURRENT.STATES := initial.state
while(not finished(CURRENT.STATES)) do
    CANDIDATE.STATES := EXPAND(CURRENT.STATES)
    SCORE(CANDIDATE.STATES)
    CURRENT.STATES := PRUNE(CANDIDATE.STATES)
```

For simplicity, we will do beam search with $k = 2$ beams for the same example as above:

- **Beam step 0**
  - Again, the only history we have is the beginning of sentence symbol $h = \langle s\rangle$. Our initial (and current) state will then be just CURRENT = $\{\langle s\rangle\}$. Note that our current state has length 1.
- **Beam step 1**
  - Now, we will expand our current state to get all possible continuations. Our candidate states will be:
    CANDIDATES = $\{\langle s\rangle\text{is}, \langle s\rangle\text{this}, \langle s\rangle\text{suboptimal}\}$.
  - We score each of them,[2] and keep track of the associated scores. To do that, let's turn our CANDIDATES set to be a set of tuples: CANDIDATES = $\{(\langle s\rangle \text{ is}, 0.6), (\langle s\rangle \text{ this}, 0.3), (\langle s\rangle \text{ suboptimal}, 0.1)\}$.
  - Now we will prune the candidate states and keep only the top-k ones, and set them to be our CURRENT states. So, our current states now are: CURRENT = $\{(\langle s\rangle \text{ is}, 0.6), (\langle s\rangle \text{ this}, 0.3)\}$.

[1] Notice that greedy search is simply a special case of beam search, where we use only $k = 1$ beam – a single possible path.

[2] the example uses dummy scores

- **Beam step 2**
  - Again, we will expand our current states (all of them!) to get all possible continuations. Instead of scoring each candidate from scratch, we can simply multiply the score of the current state with the score of the new bigram we add. Our new candidate states will be: CANDIDATES = $\{(\langle s\rangle \text{ is this}, 0.3), (\langle s\rangle \text{ is suboptimal}, 0.1), (\langle s\rangle \text{ this is}, 0.35), (\langle s\rangle \text{ this suboptimal}, 0.05)\}$.
  - Again we prune the candidate states and keep only the top-k ones, and set them to be our CURRENT states. So, our current states now are: CURRENT = $\{(\langle s\rangle \text{ this is}, 0.35), (\langle s\rangle \text{ is this}, 0.3)\}$.
- **Beam step 3**
  - Again, we will expand our current states (all of them!) to get all possible continuations. We only have one word left in our pool of words-to-use, so our job is simple. The candidates will be CANDIDATES = $\{(\langle s\rangle \text{ this is suboptimal}, 0.18), (\langle s\rangle \text{ is this suboptimal}, 0.12)\}$.
- We have finished searching,[3] so we now just return the candidate with the highest score.

In your report, apply your unscrambling method to the sentences of the `scrambled.20.txt` file, and compare against the correct `unscrambled-gold.20.txt` files. Provide the outputs of your unscrambling method as well as your accuracy. Discuss the following: how does greedy search compare to beam search in terms of accuracy? how about in terms of computation time?

[Bonus Bonus] The best performing submission for this exercise will receive an additional +2 credits.

Submission Instructions

This assignment must be done individually. Discussing the assignment on the class forum is absolutely ok – posting the solutions is not. If you use someone else's code (e.g. taken from a book or a website), make sure to properly cite it in your report.

Please submit all of the following in a gzipped tar archive (.tgz or .tar.gz; not .zip or .rar) via Blackboard. If you're making a full submission, please name your file `gmuid-hw2.tgz` (for example, the instructor's submission would be `antonis-hw2.tgz`). If you're making a partial submission, please name your file `gmuid-hw2-part.tgz` where part is the part (1, 2, or 3) that you're submitting. Make sure that your .tgz file expands into a self-contained folder with your gmuid in it.

Your submission should contain:

[3] In normal unconstrained generation with beam search, we would iterate until all beam candidates produce an end-of-sentence symbol ¡/s¿, but here we only care about using the words that were scrambled.

- A PDF file (not .doc or .docx) with your responses to the instructions/questions above. The report should include your name and gmuid, and it should be self-sufficient (as in, we shouldn't have to look elsewhere to grade you). If your name or an answer does not appear in the report, you will receive 0 credits for that part.
- All of the code that you wrote.
- A brief README file with instructions on how to build/run your code. We **will** run the code and check whether it produces all required outputs and if these outputs match the results in your report.