lesson4_languagemodel

April 29, 2018

1 Language Model Demo

Based on this demo: http://nlpforhackers.io/language-models/

1.0.1 Import modules and data

```
In [1]: import random
        from nltk import bigrams, trigrams
        from nltk.corpus import reuters, movie_reviews, shakespeare
        from nltk.tokenize import sent_tokenize, word_tokenize
        from collections import Counter, defaultdict
```

```
In [2]: # Choose a corpus: reuters, movie_reviews or shakespeare
        import nltk
        nltk.download('movie_reviews')
        corpus = movie_reviews

        if corpus == shakespeare:
            shakespeare_text = ''.join([''.join(corpus.xml(fileid).itertext()) for fileid in
        corpus.fileids()])
            words = word_tokenize(shakespeare_text)
            sents = [word_tokenize(sent) for sent in sent_tokenize(shakespeare_text)]
        else:
            words = corpus.words()
            sents = corpus.sents()

        # Lowercase everything
        words = [w.lower() for w in words]
        sents = [[w.lower() for w in sent] for sent in sents]
```

```
[nltk_data] Downloading package movie_reviews to
[nltk_data]     /Users/xiaoxing/nltk_data...
[nltk_data]   Package movie_reviews is already up-to-date!
```

1.0.2 Unigram language model

In this section, we will construct a language model based on unigrams (words).

```
In [3]: # Exercise 1. Fill in the blanks.


        # Step 1: Make a Counter from the list of words and call it "unigram_counts" (remember,
        this is easy to do!)
        unigram_counts = Counter(words)

        # Step 2: Get the total number of words and assign it to "total_count"
        total_count = sum(unigram_counts.values())
```

```
        print("Total number of words in corpus: ", total_count)

        # Print 10 most common words
        print("\nTop 10 most common words: ")

        for (word, count) in unigram_counts.most_common(n=10):
            print(word, count)
```

```
Total number of words in corpus:  1583820

Top 10 most common words:
, 77717
the 76529
. 65876
a 38106
and 35576
of 34123
to 31937
' 30585
is 25195
in 21822
```

```
In [4]: # Exercise 2. Fill in the blanks.

        # We have the Counter unigram_counts, which maps each word to its count.
        # We want to construct the Counter unigram_probs, which maps each word to its
        probability.


        # Step 1: create an empty Counter called unigram_probs.
        unigram_probs = {}

        # Step 2: using a for-loop over unigram_counts, (this will iterate over the keys i.e.
        words)
        # calculate the appropriate fraction, and add the word -> fraction pair to
        unigram_probs.
        # Remember about integer division!
        for word in unigram_counts:
            fraction = float(unigram_counts[word])/10
            unigram_probs[word] = fraction

        # Check the probabilities add up to 1
        print("Probabilities sum to: ", sum(unigram_probs.values()))

        # Print 10 most common words
        print("\nTop 10 most common words: ")
        for (word, count) in Counter(unigram_probs).most_common(n=10):
            print(word, "%.5f" % count)
```

```
Probabilities sum to:  158382.0000001032

Top 10 most common words:
, 7771.70000
the 7652.90000
. 6587.60000
a 3810.60000
and 3557.60000
of 3412.30000
to 3193.70000
' 3058.50000
is 2519.50000
```

```
in 2182.20000
```

```
In [5]: # Print the probability of word "the", then try some other words.
        print(unigram_probs['the'])
```

```
7652.9
```

```
In [6]: # Generate 100 words of language using the unigram model.
        # Run this cell several times!

        text = []  # will be a list of generated words

        for _ in range(100):
            r = random.random()  # random number in [0,1]

            # Find the word whose "interval" contains r
            accumulator = .0
            for word, freq in unigram_probs.items():
                accumulator += freq
                if accumulator >= r:
                    text.append(word)
                    break

        print(' '.join(text))
```

```
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
plot plot plot plot plot plot plot plot plot plot plot plot plot plot plot
```

1.0.3 Bigram language model

In this section, we'll build a language model based on bigrams (pairs of words).

```
In [7]: # Count how often each bigram occurs.

        # bigram_counts is a dictionary that maps w1 to a dictionary mapping w2 to the count for
        (w1, w2)
        bigram_counts = defaultdict(lambda: Counter())

        for sentence in sents:
            for w1, w2 in bigrams(sentence, pad_right=True, pad_left=True):
                bigram_counts[w1][w2] += 1
```

```
In [8]: # Print how often the bigram "of the" occurs. Try some other words following "of".
        print(bigram_counts['of']['the'])
```

```
8621
```

```
In [9]: # Transform the bigram counts to bigram probabilities
        bigram_probs = defaultdict(lambda: Counter())
        for w1 in bigram_counts:
            total_count = float(sum(bigram_counts[w1].values()))
            bigram_probs[w1] = Counter({w2: c / total_count for w2, c in
        bigram_counts[w1].items()})
```

```
In [10]: # Print the probability that 'the' follows 'of'
         print(bigram_probs['of']['the'])
```

0.25264484365384055

In [11]:
```
# Print the top ten tokens most likely to follow 'fair', along with their probabilities.
# Try some other words.
prob_dist = bigram_probs['fair']
for word, prob in prob_dist.most_common(10):
    print(word, "%.5f" % prob)
```

```
, 0.19048
to 0.15238
game 0.10476
. 0.04762
share 0.04762
amount 0.03810
enough 0.03810
bit 0.02857
warning 0.01905
town 0.00952
```

In [12]:
```
# Generate text with bigram model.
# Run this cell several times!

text = [None]  # You can put your own starting word in here
sentence_finished = False

# Generate words until a None is generated
while not sentence_finished:
    r = random.random()  # random number in [0,1]
    accumulator = .0
    latest_token = text[-1]
    prob_dist = bigram_probs[latest_token]  # prob dist of what token comes next

    # Find the word whose "interval" contains the random number r.
    for word, p in prob_dist.items():
        accumulator += p
        if accumulator >= r:
            text.append(word)
            break

    if text[-1] == None:
        sentence_finished = True

print(' '.join([t for t in text if t]))
```

```
s no genius , for me to drop down a half hour running away myself included as the cast
is being interesting about henstridge )
```

How does the bigram text compare to the unigram text?

1.0.4 Trigram language model

In this section, we'll build a language model based on trigrams (triples of words).

In [13]:
```
# Count how often each trigram occurs.

# trigram_counts maps (w1, w2) to a dictionary mapping w3 to the count for (w1, w2, w3)
trigram_counts = defaultdict(lambda: Counter())

for sentence in sents:
    for w1, w2, w3 in trigrams(sentence, pad_right=True, pad_left=True):
        trigram_counts[(w1, w2)][w3] += 1
```

In [14]:
```python
# Print how often the trigram "I am not" occurs. Try some other trigrams.
print(trigram_counts[('i', 'am')]['not'])
```

```
27
```

In [15]:
```python
# Transform the trigram counts to trigram probabilities
trigram_probs = defaultdict(lambda: Counter())
for w1_w2 in trigram_counts:
    total_count = float(sum(trigram_counts[w1_w2].values()))
    trigram_probs[w1_w2] = Counter({w3: c / total_count for w3, c in
trigram_counts[w1_w2].items()})
```

In [16]:
```python
# Print the probability that 'not' follows 'i am'. Try some other combinations.
print(trigram_probs[('i', 'am')]['not'])
```

```
0.16363636363636364
```

In [17]:
```python
# Print the top ten tokens most likely to follow 'i am', along with their probabilities.
# Try some other pairs of words.
prob_dist = trigram_probs[('i', 'am')]
for word, prob in prob_dist.most_common(10):
    print(word, "%.5f" % prob)
```

```
not 0.16364
a 0.07273
sure 0.07273
the 0.03030
willing 0.02424
going 0.02424
, 0.02424
of 0.01818
glad 0.01818
thinking 0.01212
```

In [18]:
```python
# Generate text with trigram model.
# Run this cell several times!

text = [None, None]  # You can put your own first two words in here

sentence_finished = False

# Generate words until two consecutive Nones are generated
while not sentence_finished:
    r = random.random()
    accumulator = .0
    latest_bigram = tuple(text[-2:])
    prob_dist = trigram_probs[latest_bigram]  # prob dist of what token comes next

    for word, p in prob_dist.items():
        accumulator += p
        if accumulator >= r:
            text.append(word)
            break

    if text[-2:] == [None, None]:
        sentence_finished = True

print(' '.join([t for t in text if t]))
```

```
instead of refreshing the audience with information dug up by the fact that margaret
does not borrow from a mile away .
```

How does the trigram text compare to the bigram text?

1.1 Extension exercise

N-gram language models can encounter the *sparsity problem*, especially if the data is small.

Suppose you train a trigram language model on a small amount of data (let's say the text of *The Hunger Games*), then use the language model to generate text.

On each step, you take the last two generated words (e.g. "may the") and lookup the probability distribution of what word is most likely to come next. But if your training data is small, perhaps there is only one example of the bigram "may the" in the training data (e.g. "may the odds be ever in your favor" in *The Hunger Games*). In that case, the next word will be *odds* with probability 1. This means that your language model always says "odds" after saying "may the".

1. Is the sparsity problem worse for unigram language models, bigram language models, trigram language models, or n-gram language models for n>3?
2. How might you fix this problem?
3. How might you fix this problem without access to more training data?

Try altering either the bigram or the trigram language model with your solution to question 3.