Lab 07: Language Models

**References:**

1. Regular Expressions: The Complete Tutorial, by Jan Goyvaerts, 2007.
2. Speech and Language Processing, by Dan Jurafsky and James H. Martin. Prentice Hall Series in Artificial Intelligence, 2008.
3. Natural Language Processing with Python, by Steven Bird, Ewan Klein and Edward Loper, 2014.

Quick Review

**Checking and Upgrading the NLTK Version**

An upgrade is required to get certain packages functioning properly in the Text Mining & Analytics area.

import nltk

print(nltk.\_\_version\_\_)

!pip install -U pip

!pip install -U dill

!pip install -U nltk==3.4

N-gram using NLTK

Traditionally, we can use n-grams to generate language models to predict which word comes next given a history of words.

from nltk.util import pad\_sequence

from nltk.util import bigrams

from nltk.util import ngrams

from nltk.util import everygrams

from nltk.lm.preprocessing import pad\_both\_ends

from nltk.lm.preprocessing import flatten

If we want to train a bigram model, we need to turn this text into bigrams. Here's what the first sentence of our text would look like if we use the ngrams function from NLTK for this.

import nltk

text = "I am learning Text Analytics"

tokens = nltk.tokenize.word\_tokenize(text.lower())

list(bigrams(tokens))

Here's what the first sentence of our text would look like if we use the ngrams function from NLTK for this. Here we can specify the number of grams.

list(ngrams(tokens, n=3))

Add special "padding" symbols to the sentence before splitting it into ngrams. Fortunately, NLTK also has a function for that, let's see what it does to the first sentence.

from nltk.util import pad\_sequence

list(pad\_sequence(tokens, pad\_left=True, left\_pad\_symbol="<s>", pad\_right=True, right\_pad\_symbol="</s>", n=2))

# The n order of n-grams, if it's 2-grams, you pad once, 3-grams pad twice, etc.

Then,

padded\_sent = list(pad\_sequence(tokens, pad\_left=True, left\_pad\_symbol="<s>", pad\_right=True, right\_pad\_symbol="</s>", n=2))

list(ngrams(padded\_sent, n=2))

Note the n argument, that tells the function we need padding for bigrams.

Now, passing all these parameters every time is tedious and in most cases they can be safely assumed as defaults anyway.

Thus the nltk.lm module provides a convenience function that has all these arguments already set while the other arguments remain the same as for pad\_sequence.

from nltk.lm.preprocessing import pad\_both\_ends

list(pad\_both\_ends(tokens, n=2))

Combining the two parts discussed so far we get the following preparation steps for one sentence.

list(bigrams(pad\_both\_ends(tokens, n=2)))

To make our model more robust we could also train it on unigrams (single words) as well as bigrams, its main source of information. NLTK once again helpfully provides a function called everygrams.

While not the most efficient, it is conceptually simple.

from nltk.util import everygrams

padded\_bigrams = list(pad\_both\_ends(tokens, n=2))

list(everygrams(padded\_bigrams, max\_len=1))

list(everygrams(padded\_bigrams, max\_len=2))

During training and evaluation our model will rely on a vocabulary that defines which words are "known" to the model.

To create this vocabulary we need to pad our sentences (just like for counting ngrams) and then combine the sentences into one flat stream of words.

Calculating probability of n-grams in a text of sentences

import nltk

text = "I am learning Text Analytics" # can be replaced by any size of text corpus

# Tokenize the text.

tokenized\_text = [list(map(str.lower, nltk.tokenize.word\_tokenize(text)))]

print(tokenized\_text)

# Preprocess the tokenized text for 3-grams language modelling

from nltk.lm.preprocessing import padded\_everygram\_pipeline

from nltk.lm import MLE

n = 3

train\_data, padded\_sents = padded\_everygram\_pipeline(n, tokenized\_text)

model = MLE(n) # Lets train a 3-grams maximum likelihood estimation model.

model.fit(train\_data, padded\_sents)

**To get the counts:**

model.counts['i'] # i.e. Count('i')

model.counts[['i']]['am'] # i.e. Count('am'|'i')

model.counts[['i', 'am']]['learning'] # i.e. Count('learning'|'i am')

**To get the scores/probability:**

odel.score('am', 'i'.split()) # P('am'|'i')

model.score('learning', 'i am'.split()) # P('learning'|'i am')

model.score('playing', 'i am'.split()) # P('playing'|'i am')

N-gram using NLTK.

import nltk

from nltk.util import ngrams

# Function to generate n-grams from sentences.

def extract\_ngrams(data, num):

n\_grams = ngrams(nltk.word\_tokenize(data), num)

return [ ' '.join(grams) for grams in n\_grams]

text = 'A class is a blueprint for the object.'

print("1-gram: ", extract\_ngrams(text, 1))

print("2-gram: ", extract\_ngrams(text, 2))

print("3-gram: ", extract\_ngrams(text, 3))

print("4-gram: ", extract\_ngrams(text, 4))

N-gram using TextBlob

from textblob import TextBlob

# Function to generate n-grams from sentences.

def extract\_ngrams(data, num):

n\_grams = TextBlob(data).ngrams(num)

return [ ' '.join(grams) for grams in n\_grams]

text = 'A class is a blueprint for the object.'

print("1-gram: ", extract\_ngrams(text, 1))

print("2-gram: ", extract\_ngrams(text, 2))

print("3-gram: ", extract\_ngrams(text, 3))

print("4-gram: ", extract\_ngrams(text, 4))