College of Computer Science and Engineering
Department of Computer Science and Artificial Intelligence

CCAI-413: Natural Language Processing
Lab#3 N-Gram Language Models

Objectives

- Understand and Apply N-Gram Language Models
- Use the N-Gram Model to Predict the Next Word
- Use the N-Gram Model to Generate a Sentence

Lab Tool(s)

https://www.kaggle.com/

N-gram language models

Language modeling (LMs) is a way to determine the probability of sequence of words. N-gram models is one type of LMs that depends on sequence of n-words: 1-gram (one word) called unigram, 2-gram (two words) called bigram, 3-gram (three words) called trigram, and so on.

LMs Applications:

- Automatic speech recognition
- Handwriting and character recognition
- Spelling correction
- Machine translation
- And many more.

NLTK provides the n-gram language model, type the following command to import required libraries:

[6]:
```
from nltk import ngrams
from nltk import bigrams
from nltk import trigrams
```

+ Code + Markdown

To turn the text into n-grams, type the following commands:

Unigram

```
text = [['This', 'is', 'NLP','Lab'], ['The', 'Lab', 'is', 'NLP']]
list(ngrams(text[1], n=1))
```

: [('The',), ('Lab',), ('is',), ('NLP',)]

Bigram

```
list(bigrams(text[0]))
```

[('This', 'is'), ('is', 'NLP'), ('NLP', 'Lab')]

Trigram

```
list(ngrams(text[1], n=3))
```

]: [('The', 'Lab', 'is'), ('Lab', 'is', 'NLP')]

Build N-gram Language Model

In this lab you will learn how to build a n-gram language model by training the model, calculate the probability, and then use the model in NLP application.

Dataset

In this lab, we will use the Brown corpus, which is million-word electronic corpus of English, created in 1961 at Brown University. Brown corpus contains text from around 500 sources. Type the following command to import the data:

```
from nltk.corpus import brown
```

Display the dataset in form of sentences:

```
brown.sents()
```

```
[['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', 'Friday', 'an', 'investigation', 'of', "Atlanta's", 'recen
t', 'primary', 'election', 'produced', '``', 'no', 'evidence', "''", 'that', 'any', 'irregularities', 'took', 'p
lace', '.'], ['The', 'jury', 'further', 'said', 'in', 'term-end', 'presentments', 'that', 'the', 'City', 'Execut
ive', 'Committee', ',', 'which', 'had', 'over-all', 'charge', 'of', 'the', 'election', ',', '``', 'deserves', 't
he', 'praise', 'and', 'thanks', 'of', 'the', 'City', 'of', 'Atlanta', "''", 'for', 'the', 'manner', 'in', 'whic
h', 'the', 'election', 'was', 'conducted', '.'], ...]
```

Build the n-gram model

Before building the model, create a defaultdict object, which is a dictionary in Python that store a collection of data values.

```
from collections import Counter, defaultdict

model = defaultdict(lambda: defaultdict(lambda: 0))
```

Split text into trigrams using NLTK and then calculate the frequency. *Pad_right* and *pad_left* functions help to ensure that each combination of the trigrams occurs in the dataset.

```
# Count frequency of co-occurance
for sentence in brown.sents():
    for w1, w2, w3 in trigrams(sentence, pad_right=True, pad_left=True):
        model[(w1, w2)][w3] += 1
```

Then, transform the count to probabilities as follows:

```
# Let's transform the counts to probabilities
for w1_w2 in model:
    total_count = float(sum(model[w1_w2].values()))
    for w3 in model[w1_w2]:
        model[w1_w2][w3] /= total_count
```

Use The Model

Use the model to predict the next word

After building the model, now let's use the model to predict the next word given two words

```
dict(model["City","of"])
```

```
{'Atlanta': 0.07142857142857142,
 'San': 0.07142857142857142,
 'Miami': 0.07142857142857142,
 'Warwick': 0.14285714285714285,
 'Sandalwood': 0.07142857142857142,
 'Bullet': 0.07142857142857142,
 'Buffalo': 0.07142857142857142,
 'Lions': 0.07142857142857142,
 'Silkworms': 0.07142857142857142,
 'London': 0.07142857142857142,
 'Palaces': 0.07142857142857142,
 'Washington': 0.07142857142857142,
 'God': 0.07142857142857142}
```

Sort the predicted values in a descending order to bring the word with the highest probability at the top:

```
dict(sorted(model["City","of"].items(), key=lambda item: item[1],reverse = True))
```

```
{'Warwick': 0.14285714285714285,
 'Atlanta': 0.07142857142857142,
 'San': 0.07142857142857142,
 'Miami': 0.07142857142857142,
 'Sandalwood': 0.07142857142857142,
 'Bullet': 0.07142857142857142,
 'Buffalo': 0.07142857142857142,
 'Lions': 0.07142857142857142,
 'Silkworms': 0.07142857142857142,
 'London': 0.07142857142857142,
 'Palaces': 0.07142857142857142,
 'Washington': 0.07142857142857142,
 'God': 0.07142857142857142}
```

As you can see from the result, the word "Warwick" has the highest probability, so it most likely to be the next word.

Use the model for text generation

In this section, we will use the trained model to generate a random text.

Import the random library that will be used to create a random threshold later:

```
import random
```

Select two words (as our model is Trigram) in which the sentence will began:

```
text = ["today", "the"]
```

The following code will generate a random sentence from the word starts by the above two words. The code will take the last two words only from the sentence and loop over the existing words. The words that have a probability ***accumulator*** above the threshold ***r*** will be added to the sentence. Then the code will take the last two words and search for another word, and so on. The code will be repeated until no word above the threshold is found.

```
sentence_finished = False

while not sentence_finished:
  # select a random probability threshold
  r = random.random()
  accumulator = .0

  for word in model[tuple(text[-2:])].keys():
      accumulator += model[tuple(text[-2:])][word]
      # select words that are above the probability threshold
      if accumulator >= r:
          text.append(word)
          break


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

Print the selected words in on sentence

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

```
today the only young man of principle and with no previous issue of March 4 .
```

The generated sentence may seem a little confusing, but it's quality can be improved by selecting the appropriate threshold, using more data, and using an appropriate data.

Note: The model will generate different sentence every time as long as the threshold ***r*** is selected randomly and not constant. By giving the threshold a constant value, the model will generate the same sentence every time.

References:

- https://web.stanford.edu/~jurafsky/slp3/3.pdf
- https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-language-model-nlp-python-code/