n_gram_language_model_with_nltk

Install

```
#!pip install -U pip
!pip install -U dill
!pip install -U nltk==3.4
```

- dill
  - a Python package that **extends the capabilities of another Python library called pickle.**

    Pickle allows you to save (serialize) Python objects to a file and then **load (deserialize) them back into your Python session later.**
- nltk
  - **nltk is a toolkit** designed for **working with human language data (text).** It's like a Swiss Army knife for natural language processing (NLP) tasks.

    With nltk, you can **perform a variety of operations on text**, such as **classifying, tokenizing, stemming, tagging, parsing, and semantic reasoning.** This helps computers understand, interpret, and even generate human language.

N-grams Language Models (N-grams LM)

- Nowadays, everything seems to be going neural...

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

  We'll use the `lm` module in `nltk` to get a sense of how non-neural language modelling is done.

  (**Source:** The content in this notebook is largely based on language model tutorial in NLTK documentation by Ilia Kurenkov)

  todays --> neural network (gpt)
  in the past --> we use n-grams
- import packages — understand bigrams/ngrams

  ```
  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
  ```
- n-gram

  ```
  In [3]: text = [['a', 'b', 'c'], ['a', 'c', 'd', 'c', 'e', 'f']]
  In [4]: list(bigrams(text[0]))
  Out[4]: [('a', 'b'), ('b', 'c')]
  In [5]: list(ngrams(text[1], n=3))
  Out[5]: [('a', 'c', 'd'), ('c', 'd', 'c'), ('d', 'c', 'e'), ('c', 'e', 'f')]
  ```
  - bigrams --> (Bigrams are pairs of consecutive elements from a sequence.)
  - ngrams --> similar
- padding

  ```
  from nltk.util import pad_sequence
  list(pad_sequence(text[0],
                    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.
  ['<s>', 'a', 'b', 'c', '</s>']
  ```
  - **pad_left=True and left_pad_symbol="<s>"**

    --> mean you're adding a special symbol <s> at the beginning of your list.
  - **pad_right=True and right_pad_symbol="</s>"**

    --> mean you're adding a special symbol </s> at the end of your list.
  - **n=2 here indicates you're working with bigrams (2-grams),** so you only need to add one symbol at the beginning and one at the end, because you're only looking at pairs of items.

  ```
  padded_sent = list(pad_sequence(text[0], pad_left=True, left_pad_symbol="<s>",
                                  pad_right=True, right_pad_symbol="</s>", n=2))
  list(ngrams(padded_sent, n=2))
  [('<s>', 'a'), ('a', 'b'), ('b', 'c'), ('c', '</s>')]
  ```
  - `['<s>', 'a', 'b', 'c', '</s>']` doing bigram based on the new list
  - `[('<s>', 'a'),` --> ensure the first element of ngram only have padding or the first character 'a'

  ```
  list(pad_sequence(text[0],
                    pad_left=True, left_pad_symbol="<s>",
                    pad_right=True, right_pad_symbol="</s>",
                    n=3)) # The n order of n-grams, if it's 2-grams, you pad once, 3-grams pad twice, etc.
  ['<s>', '<s>', 'a', 'b', 'c', '</s>', '</s>']
  ```
  - ensure the first element of ngram only have padding or the first character 'a'

    -->

    so when it is trigram, we need 2 <s> at the beginning

    same thing for the last element

  ```
  padded_sent = list(pad_sequence(text[0], pad_left=True, left_pad_symbol="<s>",
                                  pad_right=True, right_pad_symbol="</s>", n=3))
  list(ngrams(padded_sent, n=3))
  [('<s>', '<s>', 'a'),
   ('<s>', 'a', 'b'),
   ('a', 'b', 'c'),
   ('b', 'c', '</s>'),
   ('c', '</s>', '</s>')]
  ```
  - `[('<s>', '<s>', 'a'),` doing trigram on the new list -->

    we can see the first element now only have padding or the first character 'a'
  - simplified padding code

    ```
    from nltk.lm.preprocessing import pad_both_ends
    list(pad_both_ends(text[0], n=2))
    ['<s>', 'a', 'b', 'c', '</s>']
    ```
    - the previous one --> we now don't need to specify pad_left/pad_right, etc

    ```
    list(bigrams(pad_both_ends(text[0], n=2)))
    [('<s>', 'a'), ('a', 'b'), ('b', 'c'), ('c', '</s>')]
    ```
  - everygrams

    ```
    from nltk.util import everygrams
    padded_bigrams = list(pad_both_ends(text[0], n=2))
    list(everygrams(padded_bigrams, max_len=2))
    [('<s>',),
     ('a',),
     ('b',),
     ('c',),
     ('</s>',),
     ('<s>', 'a'),
     ('a', 'b'),
     ('b', 'c'),
     ('c', '</s>')]
    ```
    - evergrams() --> include all grams up to max_len

      now max_len=2 --> so we need to include both unigram & bigram
  - flatten

    ```
    from nltk.lm.preprocessing import flatten
    list(flatten(pad_both_ends(sent, n=2) for sent in text))
    ['<s>', 'a', 'b', 'c', '</s>', '<s>', 'a', 'c', 'd', 'c', 'e', 'f', '</s>']
    ```
    - just **flatten the two lists in text, which processed with path_both_ends**
  - simplified padding + ngram

    ```
    from nltk.lm.preprocessing import padded_everygram_pipeline
    train, vocab = padded_everygram_pipeline(2, text)
    ```
    - **padded_everygram_pipeline --> handles two crucial steps** in one go: **padding your sequences** and **generating n-grams**
  - padded_everygram_pipeline

    ```
    training_ngrams, padded_sentences = padded_everygram_pipeline(2, text)
    for ngramlize_sent in training_ngrams:
        print(list(ngramlize_sent))
        print()
    print('#############')
    list(padded_sentences)
    ```
    - **training_ngrams** --> include unigram to bigram lists with padding for all lists in text
    - `['<s>', 'a', 'b', 'c', '</s>', '<s>', 'a', 'c', 'd', 'c', 'e', 'f', '</s>']` padded sentence --> padded

Lets get some real data and tokenize it

- just two different ways to do tokenization using different tokenizor

  ```
  from nltk import word_tokenize, sent_tokenize
  # Testing whether it works.
  # Sometimes it doesn't work on some machines because of setup issues.
  word_tokenize(sent_tokenize("This is a foobar sentence. Yes it is.")[0])
  ```
  - sometimes you may encounter error using this method

    ```
    word_tokenize
    ['This', 'is', 'a', 'foobar', 'sentence', '.']
    ```
  - naive way to do tokenization --> use regular expression matching
- find and open the language-never-random.txt
  - basically an article
  - the text is a list. each element is a sentence in the article.
- do the tokenization

  ```
  # Tokenize the text.
  tokenized_text = [list(map(str.lower, word_tokenize(sent)))
                    for sent in sent_tokenize(text)]
  ```
  - the tokenization of the first line of the article

    ```
    tokenized_text[0]
    ['language', 'is', 'never', ',', 'ever', ',', 'ever', ',', 'random', 'adam', 'kilgarriff', 'abstract', 'language', 'users', 'never', 'choose', 'words', 'randomly', ',', 'and', 'language', 'is', 'essentially', 'non-random', '.']
    ```
  - first 500 characters of the article
- use the padded_everygram_pipeline we have learnt to preprocess the whole text

  ```
  # Preprocess the tokenized text for 3-grams language modelling
  n = 3
  train_data, padded_sents = padded_everygram_pipeline(n, tokenized_text)
  ```
  - recall: padded_everygram_pipeline
    - **training_ngrams** --> include unigram to bigram lists with padding for all lists in text
    - padded sentence --> padded

Training an N-gram Model

- maximum likelihood Estimator

  ```
  from nltk.lm import MLE
  model = MLE(n) # Lets train a 3-grams model, previously we set n=3
  ```
  - a old-school model
  - basic idea

    Imagine you have a bag containing only orange and grape candies, but you don't know the exact proportion of each. You decide to conduct an experiment to estimate this:
    - **Experiment**: You randomly pick 10 candies, one at a time, and note their flavor.
    - **Results**: You find 7 orange candies and 3 grape candies.

    Now, you wonder, "What's the most likely proportion of orange to grape candies in the bag?"

    Using MLE, you calculate the likelihood of drawing 7 orange and 3 grape candies for different proportions (like 50% orange-50% grape, 70% orange-30% grape, etc.). The proportion that makes your specific draw (7 orange, 3 grape) the most likely is the MLE estimate.
  - Applying **Maximum Likelihood Estimation (MLE) to text** involves using it to build models that can **predict the next word in a sentence** based on the words that have come before it.
    - basically learn the correlation between words
      - **Step 1: Collecting Data** — First, you collect data on word sequences. Let's say your book has the following sentences: "The cat sat on the mat." "The cat chased the mouse." "The dog chased the cat." From these sentences, you gather data on how often each word follows another word.
      - **Step 2: Calculating Probabilities with MLE** — Using MLE, you calculate the probability of each word following another word. For instance: To find the probability of the word "chased" following the word "cat", you see how many times "cat" is followed by "chased" and divide it by the total number of words that follow "cat". In our data: "cat" is followed by "sat" once, "chased" once, and that's it. So, the probability of "chased" following "cat" (using MLE) is 1/2 (or 50%).
      - **Step 3: Applying MLE to Predictions** — When your model encounters the phrase "The cat", it uses the calculated probabilities to predict the next word. Since "sat" and "chased" both have a probability of 50% in our very small dataset, it might suggest either as the next word.
    - the cat is on cushion
    - the --> language model --> what is the following word?
      - vocabulary/dictionary
        - cat --> 50% / on --> 20% / cushion --> 30% / ... — unigram
        - cat / on / cushion / cat on / on cushion
          - apple
          - Apple Inc
    - the cat --> language model -->
- at the beginning, the model vocab is 0

  ```
  len(model.vocab)
  0
  ```
  - ... which gets filled as we fit the model.

    ```
    model.fit(train_data, padded_sents)
    print(model.vocab)
    <Vocabulary with cutoff=1 unk_label='<UNK>' and 1429 items>
    ```
    vocabulary cutoff= 1 here means we only include those words that occur more than once in the dataset
  - ```
    len(model.vocab)
    1429
    ```
    automatically it finds there are 1429 words in the vocabulary
- The vocabulary helps us handle words that have not occurred during training.

  ```
  print(model.vocab.lookup(tokenized_text[0]))
  ('language', 'is', 'never', ',', 'ever', ',', 'ever', ',', 'random', 'adam', 'kilgarriff', 'abstract', 'language', 'users', 'never', 'choose', 'words', 'randomly', ',', 'and', 'language', 'is', 'essentially', 'non-random', '.')
  ```
  - lookup all the tokens for the first sentence
- ```
  # If we lookup the vocab on unseen sentences not from the training data,
  # it automatically replace words not in the vocabulary with `<UNK>`.
  print(model.vocab.lookup('language is never random lah .'.split()))
  ('language', 'is', 'never', 'random', '<UNK>', '.')
  ```
  lah --> is not shown in the dataset
  --> thus the model will assign a <UNK>, i.e., unknown token to the sentence
- **Note:** For more sophisticated ngram models, take a look at these objects from `nltk.lm.models`:
  - `Lidstone`: Provides Lidstone-smoothed scores.
  - `Laplace`: Implements Laplace (add one) smoothing.
  - `InterpolatedLanguageModel`: Logic common to all interpolated language models (Chen & Goodman 1995).
  - `WittenBellInterpolated`: Interpolated version of Witten-Bell smoothing.

Using the N-gram language Model

- ngramcounter

  When it comes to ngram models the training boils down to counting up the ngrams from the training corpus.

  ```
  print(model.counts)
  <NgramCounter with 3 ngram orders and 18687 ngrams>
  ```
- how many times "language" is mentioned

  ```
  model.counts['language'] # i.e. Count('language')
  25
  ```
- how many times "language is" is mentioned

  ...and bigrams for the phrase "language is"

  ```
  model.counts[['language']]['is'] # i.e. Count('is'|'language')
  11
  ```
- how many times "language is never" is mentioned

  ... and trigrams for the phrase "language is never"

  ```
  model.counts[['language', 'is']]['never'] # i.e. Count('never'|'language is')
  7
  ```
- And so on. However, **the real purpose of training a language model** is to **have it score how probable words** are in certain contexts.

  This being MLE, the model **returns the item's relative frequency as its score.**

  ```
  model.score('language') # P('language')
  0.003916040100250626
  ```
- higher the score, higher the co-occurence ratio of the two words

  ```
  model.score('is', 'language'.split())  # P('is'|'language')
  0.44
  ```
- ```
  model.score('never', 'language is'.split())  # P('never'|'language is')
  0.6363636363636364
  ```
- words not seen in the dataset get the same score as the unknown token

  Items that are not seen during training are mapped to the vocabulary's "unknown label" token. This is "" by default.

  ```
  model.score("<UNK>") == model.score("lah")
  True
  model.score("<UNK>") == model.score("leh")
  True
  model.score("<UNK>") == model.score("lor")
  True
  ```
- To avoid underflow when working with many small score values it makes sense to take their logarithm. For convenience this can be done with the `logscore` method.

  ```
  model.logscore("never", "language is".split())
  -0.6520766965796932
  ```
  - When computers work with **very small numbers** (**especially when multiplying many of them**), it can lead to **underflow**, where the number gets so small that the computer treats it as zero. This can mess up calculations, especially in probability when you're often multiplying probabilities together.
    - $P = \left(\frac{1}{6}\right)^3 = \frac{1}{6} \times \frac{1}{6} \times \frac{1}{6}$

      Let's compute this probability:

      $P = \frac{1}{216}$

      this is approximately 0.00463
      - $\log(P) = \log\left(\left(\frac{1}{6}\right)^3\right)$

        the results now become:
        -5.375

Generation using N-gram language model

- we can let the model to randomly generate some text

  One cool feature of ngram models is that they can be used to generate text.

  ```
  print(model.generate(20, random_seed=7))
  ['ate', 'inferences', 'are', 'drawn.', '2', '.', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>']
  ```
- make the output more human-like

  ```
  from nltk.tokenize.treebank import TreebankWordDetokenizer

  detokenize = TreebankWordDetokenizer().detokenize

  def generate_sent(model, num_words, random_seed=42):
      """
      :param model: An ngram language model from `nltk.lm.model`.
      :param num_words: Max no. of words to generate.
      :param random_seed: Seed value for random.
      """
      content = []
      for token in model.generate(num_words, random_seed=random_seed):
          if token == '<s>':
              continue
          if token == '</s>':
              break
          content.append(token)
          print(content)
      return detokenize(content)
  ```
  - detokenize function is used to convert the list of tokens to a sentence

    ```
    generate_sent(model, 20, random_seed=7)
    ['ate']
    ['ate', 'inferences']
    ['ate', 'inferences', 'are']
    ['ate', 'inferences', 'are', 'drawn.']
    ['ate', 'inferences', 'are', 'drawn.', '2']
    ['ate', 'inferences', 'are', 'drawn.', '2', '.']
    'ate inferences are drawn. 2.'
    ```
    - when encounter the end padding </s> we stop
- examples of random sentence generation

  ```
  print(model.generate(28, random_seed=0))
  ['the', 'trouble', 'with', 'quantitative', 'studies', '.', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', '</s>', ...

  generate_sent(model, 28, random_seed=0)
  'the trouble with quantitative studies.'

  generate_sent(model, 20, random_seed=1)
  '29x50. manning, christopher and hinrich schütze 1999 foundations of statistical independence.'

  generate_sent(model, 20, random_seed=30)
  'information glut, is inappropriate, particularly where counts are low.'

  generate_sent(model, 20, random_seed=42)
  'not random, or to refer to items that are more common or more salient in the last paragraph is'
  ```
  some randomly generated sentences
- Saving/loading the model

  ### Saving the model

  The native Python's pickle may not save the lambda functions in the model, so we can use the `dill` library in place of pickle to save and load the language model.

  ```
  import dill as pickle

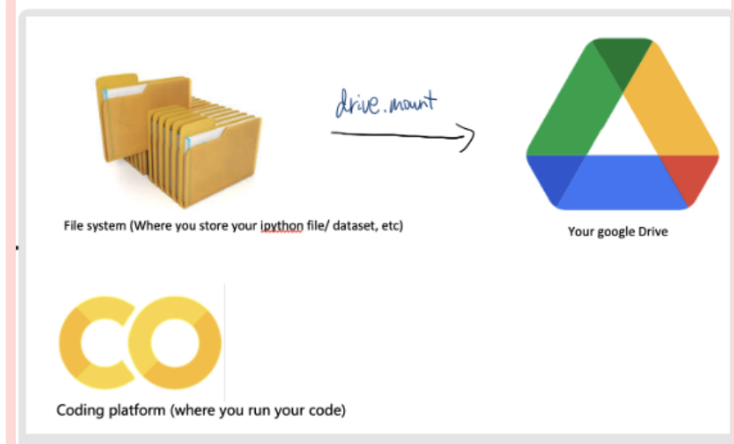
  with open('kilgariff_ngram_model.pkl', 'wb') as fout:
      pickle.dump(model, fout)

  with open('kilgariff_ngram_model.pkl', 'rb') as fin:
      model_loaded = pickle.load(fin)

  generate_sent(model_loaded, 20, random_seed=42)
  'not random, or to refer to items that are more common or more salient in the last paragraph is'
  ```

Lets try some generating with Donald Trump data!!!

- put the file in the drive and mount it

  connect me to the google drive

  For mac it's connected to your home directory

  ```
  from google.colab import drive
  drive.mount('/gdrive')
  #replace the following path according to your Google Drive path
  %cd/gdrive/My Drive/Monash-FIT-S1-2022/week_3

  Mounted at /gdrive
  /gdrive/My Drive/Monash-FIT-S1-2022/week_3
  ```
- read out the donald-tweets!.csv file and check what it looks like

  ```
  import pandas as pd
  df = pd.read_csv('Donald-Tweets!.csv')
  df.head()
  ```

	Date	Time	Tweet_Text	Type	Media_Type	Hashtags	Tweet_Id	Tweet_Url	twt_favourites_IS_THIS_LIKE_QUESTION_MARK	Retweets	Unnamed: 10	Unnamed: 11
0	16-11-11	15:26:37	Today we express our deepest gratitude to all ...	text	photo	ThankAVet	7.970000e+17	https://twitter.com/realDonaldTrump/status/797...	127213	41112	NaN	NaN
1	16-11-11	13:33:35	Busy day planned in New York. Will soon be mak...	text	NaN	NaN	7.970000e+17	https://twitter.com/realDonaldTrump/status/797...	141527	28654	NaN	NaN
2	16-11-11	11:14:20	Love the fact that the small groups of protest...	text	NaN	NaN	7.970000e+17	https://twitter.com/realDonaldTrump/status/797...	183729	50039	NaN	NaN
3	16-11-11	2:19:44	Just had a very open and successful presidenti...	text	NaN	NaN	7.970000e+17	https://twitter.com/realDonaldTrump/status/796...	214001	67010	NaN	NaN
4	16-11-11	2:10:46	A fantastic day in D.C. Met with President Oba...	text	NaN	NaN	7.970000e+17	https://twitter.com/realDonaldTrump/status/796...	178499	36688	NaN	NaN

- conduct tokenization

  ```
  trump_corpus = list(df['Tweet_Text'].apply(word_tokenize))
  ```

  ```
  trump_corpus
  Output exceeds the size limit. Open the fu
  [['Today', 'we', 'express', 'our', 'deepest', 'gratitude', 'to', 'all', 'those', 'who', 'have', 'served', 'in', 'our', 'armed', 'forces.', '#ThankAVet', 'https://t.co/wPk7QWpK8Z'],
   ['Busy', 'day', 'planned', 'in', 'New', 'York.', 'Will',
  ...
    'speech', 'transcript', ':', 'https://t.co/npswT6lbg3'],
  ...]
  ```
- preprocessing the data

  ```
  # Preprocess the tokenized text for 3-grams language modelling
  n = 3
  train_data, padded_sents = padded_everygram_pipeline(n, trump_corpus)
  ```
- haha! now you can have a model generate sentences that sounds like something Trump has spoken.

  ```
  from nltk.lm import MLE
  trump_model = MLE(n) # Lets train a 3-grams model, previously we set n=3
  trump_model.fit(train_data, padded_sents)

  generate_sent(trump_model, num_words=20, random_seed=42)
  'do so many people on television. Just another desperate move by the media pile on against me in Rome ,'

  generate_sent(trump_model, num_words=10, random_seed=0)
  'pretty sad situation. Go Jeb! You made winning MAJORS'

  generate_sent(trump_model, num_words=50, random_seed=10)
  'and many other subjects! Bad times for divided USA! +Israel2 "'

  print(generate_sent(trump_model, num_words=100, random_seed=52))
  with you sir! Your friend and supporter tonight at Trump Winery Beautiful fall foliage
  ```
  ```