C2_W3_lecture_nb_02_building_the_language_model

August 1, 2022

Building the language model

Count matrix

To calculate the n-gram probability, you will need to count frequencies of n-grams and n-gram prefixes in the training dataset. In some of the code assignment exercises, you will store the n-gram frequencies in a dictionary.

In other parts of the assignment, you will build a count matrix that keeps counts of (n-1)-gram prefix followed by all possible last words in the vocabulary.

The following code shows how to check, retrieve and update counts of n-grams in the word count dictionary.

[1]:
```
# manipulate n_gram count dictionary

n_gram_counts = {
    ('i', 'am', 'happy'): 2,
    ('am', 'happy', 'because'): 1}

# get count for an n-gram tuple
print(f"count of n-gram {('i', 'am', 'happy')}: {n_gram_counts[('i', 'am',␣
 ↪'happy')]}")

# check if n-gram is present in the dictionary
if ('i', 'am', 'learning') in n_gram_counts:
    print(f"n-gram {('i', 'am', 'learning')} found")
else:
    print(f"n-gram {('i', 'am', 'learning')} missing")

# update the count in the word count dictionary
n_gram_counts[('i', 'am', 'learning')] = 1
if ('i', 'am', 'learning') in n_gram_counts:
    print(f"n-gram {('i', 'am', 'learning')} found")
else:
    print(f"n-gram {('i', 'am', 'learning')} missing")
```

```
count of n-gram ('i', 'am', 'happy'): 2
n-gram ('i', 'am', 'learning') missing
n-gram ('i', 'am', 'learning') found
```

The next code snippet shows how to merge two tuples in Python. That will be handy when creating the n-gram from the prefix and the last word.

[2]:
```
# concatenate tuple for prefix and tuple with the last word to create the n_gram
prefix = ('i', 'am', 'happy')
word = 'because'

# note here the syntax for creating a tuple for a single word
n_gram = prefix + (word,)
print(n_gram)
```

```
('i', 'am', 'happy', 'because')
```

In the lecture, you've seen that the count matrix could be made in a single pass through the corpus. Here is one approach to do that.

[3]:
```
import numpy as np
import pandas as pd
from collections import defaultdict
def single_pass_trigram_count_matrix(corpus):
    """
    Creates the trigram count matrix from the input corpus in a single pass␣
 ↪through the corpus.

    Args:
        corpus: Pre-processed and tokenized corpus.

    Returns:
        bigrams: list of all bigram prefixes, row index
        vocabulary: list of all found words, the column index
        count_matrix: pandas dataframe with bigram prefixes as rows,
                      vocabulary words as columns
                      and the counts of the bigram/word combinations (i.e.␣
 ↪trigrams) as values
    """
    bigrams = []
    vocabulary = []
    count_matrix_dict = defaultdict(dict)

    # go through the corpus once with a sliding window
    for i in range(len(corpus) - 3 + 1):
        # the sliding window starts at position i and contains 3 words
        trigram = tuple(corpus[i : i + 3])

        bigram = trigram[0 : -1]
        if not bigram in bigrams:
            bigrams.append(bigram)
```

```python
        last_word = trigram[-1]
        if not last_word in vocabulary:
            vocabulary.append(last_word)

        if (bigram,last_word) not in count_matrix_dict:
            count_matrix_dict[bigram,last_word] = 0

        count_matrix_dict[bigram,last_word] += 1

    # convert the count_matrix to np.array to fill in the blanks
    count_matrix = np.zeros((len(bigrams), len(vocabulary)))
    for trigram_key, trigam_count in count_matrix_dict.items():
        count_matrix[bigrams.index(trigram_key[0]), \
                     vocabulary.index(trigram_key[1])]\
        = trigam_count

    # np.array to pandas dataframe conversion
    count_matrix = pd.DataFrame(count_matrix, index=bigrams, columns=vocabulary)
    return bigrams, vocabulary, count_matrix

corpus = ['i', 'am', 'happy', 'because', 'i', 'am', 'learning', '.']

bigrams, vocabulary, count_matrix = single_pass_trigram_count_matrix(corpus)

print(count_matrix)
```

```
                  happy  because    i   am  learning    .
(i, am)             1.0      0.0  0.0  0.0       1.0  0.0
(am, happy)         0.0      1.0  0.0  0.0       0.0  0.0
(happy, because)    0.0      0.0  1.0  0.0       0.0  0.0
(because, i)        0.0      0.0  0.0  1.0       0.0  0.0
(am, learning)      0.0      0.0  0.0  0.0       0.0  1.0
```

Probability matrix The next step is to build a probability matrix from the count matrix.

You can use an object dataframe from library pandas and its methods sum and div to normalize the cell counts with the sum of the respective rows.

[4]:
```python
# create the probability matrix from the count matrix
row_sums = count_matrix.sum(axis=1)
# divide each row by its sum
prob_matrix = count_matrix.div(row_sums, axis=0)

print(prob_matrix)
```

```
                  happy  because    i   am  learning    .
(i, am)             0.5      0.0  0.0  0.0       0.5  0.0
(am, happy)         0.0      1.0  0.0  0.0       0.0  0.0
(happy, because)    0.0      0.0  1.0  0.0       0.0  0.0
```

```
(because, i)        0.0       0.0  0.0  1.0       0.0  0.0
(am, learning)      0.0       0.0  0.0  0.0       0.0  1.0
```

The probability matrix now helps you to find a probability of an input trigram.

[5]:
```python
# find the probability of a trigram in the probability matrix
trigram = ('i', 'am', 'happy')

# find the prefix bigram
bigram = trigram[:-1]
print(f'bigram: {bigram}')

# find the last word of the trigram
word = trigram[-1]
print(f'word: {word}')

# we are using the pandas dataframes here, column with vocabulary word comes␣
↪first, row with the prefix bigram second
trigram_probability = prob_matrix[word][bigram]
print(f'trigram_probability: {trigram_probability}')
```

```
bigram: ('i', 'am')
word: happy
trigram_probability: 0.5
```

In the code assignment, you will be searching for the most probable words starting with a prefix. You can use the method str.startswith to test if a word starts with a prefix.

Here is a code snippet showing how to use this method.

[6]:
```python
# lists all words in vocabulary starting with a given prefix
vocabulary = ['i', 'am', 'happy', 'because', 'learning', '.', 'have', 'you',␣
↪'seen','it', '?']
starts_with = 'ha'

print(f'words in vocabulary starting with prefix: {starts_with}\n')
for word in vocabulary:
    if word.startswith(starts_with):
        print(word)
```

```
words in vocabulary starting with prefix: ha

happy
have
```

Language model evaluation ### Train/validation/test split In the videos, you saw that to evaluate language models, you need to keep some of the corpus data for validation and testing.

The choice of the test and validation data should correspond as much as possible to the distribution of the data coming from the actual application. If nothing but the input corpus is known, then random sampling from the corpus is used to define the test and validation subset.

Here is a code similar to what you'll see in the code assignment. The following function allows you to randomly sample the input data and return train/validation/test subsets in a split given by the method parameters.

[7]:
```
# we only need train and validation %, test is the remainder
import random
def train_validation_test_split(data, train_percent, validation_percent):
    """
    Splits the input data to  train/validation/test according to the percentage␣
 ↪provided

    Args:
        data: Pre-processed and tokenized corpus, i.e. list of sentences.
        train_percent: integer 0-100, defines the portion of input corpus␣
 ↪allocated for training
        validation_percent: integer 0-100, defines the portion of input corpus␣
 ↪allocated for validation

        Note: train_percent + validation_percent need to be <=100
              the reminder to 100 is allocated for the test set

    Returns:
        train_data: list of sentences, the training part of the corpus
        validation_data: list of sentences, the validation part of the corpus
        test_data: list of sentences, the test part of the corpus
    """
    # fixed seed here for reproducibility
    random.seed(87)

    # reshuffle all input sentences
    random.shuffle(data)

    train_size = int(len(data) * train_percent / 100)
    train_data = data[0:train_size]

    validation_size = int(len(data) * validation_percent / 100)
    validation_data = data[train_size:train_size + validation_size]

    test_data = data[train_size + validation_size:]

    return train_data, validation_data, test_data

data = [x for x in range (0, 100)]

train_data, validation_data, test_data = train_validation_test_split(data, 80,␣
 ↪10)
```

```python
print("split 80/10/10:\n",f"train data:{train_data}\n", f"validation data:
↪{validation_data}\n",
      f"test data:{test_data}\n")

train_data, validation_data, test_data = train_validation_test_split(data, 98,␣
↪1)
print("split 98/1/1:\n",f"train data:{train_data}\n", f"validation data:
↪{validation_data}\n",
      f"test data:{test_data}\n")
```

```
split 80/10/10:
 train data:[28, 76, 5, 0, 62, 29, 54, 95, 88, 58, 4, 22, 92, 14, 50, 77, 47,
33, 75, 68, 56, 74, 43, 80, 83, 84, 73, 93, 66, 87, 9, 91, 64, 79, 20, 51, 17,
27, 12, 31, 67, 81, 7, 34, 45, 72, 38, 30, 16, 60, 40, 86, 48, 21, 70, 59, 6,
19, 2, 99, 37, 36, 52, 61, 97, 44, 26, 57, 89, 55, 53, 85, 3, 39, 10, 71, 23,
32, 25, 8]
 validation data:[78, 65, 63, 11, 49, 98, 1, 46, 15, 41]
 test data:[90, 96, 82, 42, 35, 13, 69, 24, 94, 18]

split 98/1/1:
 train data:[66, 23, 29, 28, 52, 87, 70, 13, 15, 2, 62, 43, 82, 50, 40, 32, 30,
79, 71, 89, 6, 10, 34, 78, 11, 49, 39, 42, 26, 46, 58, 96, 97, 8, 56, 86, 33,
93, 92, 91, 57, 65, 95, 20, 72, 3, 12, 9, 47, 37, 67, 1, 16, 74, 53, 99, 54, 68,
5, 18, 27, 17, 48, 36, 24, 45, 73, 19, 41, 59, 21, 98, 0, 31, 4, 85, 80, 64, 84,
88, 25, 44, 61, 22, 60, 94, 76, 38, 77, 81, 90, 69, 63, 7, 51, 14, 55, 83]
 validation data:[35]
 test data:[75]
```

Perplexity

In order to implement the perplexity formula, you'll need to know how to implement m-th order root of a variable.

$$PP(W) = \sqrt[M]{\prod_{i=1}^{m} \frac{1}{P(w_i|w_{i-1})}}$$

Remember from calculus:

$$\sqrt[M]{\frac{1}{x}} = x^{-\frac{1}{M}}$$

Here is a code that will help you with the formula.

[8]:
```python
# to calculate the exponent, use the following syntax
p = 10 ** (-250)
M = 100
```

```python
perplexity = p ** (-1 / M)
print(perplexity)
```

```
316.22776601683796
```

That’s all for the lab for “N-gram language model” lesson of week 3.