Assignment 1

Pin-Jie Lin

pili00001@stud.uni-saarland.de
https://github.com/pjlintw/CL20/tree/main/assignment1

1 Zipf's Law

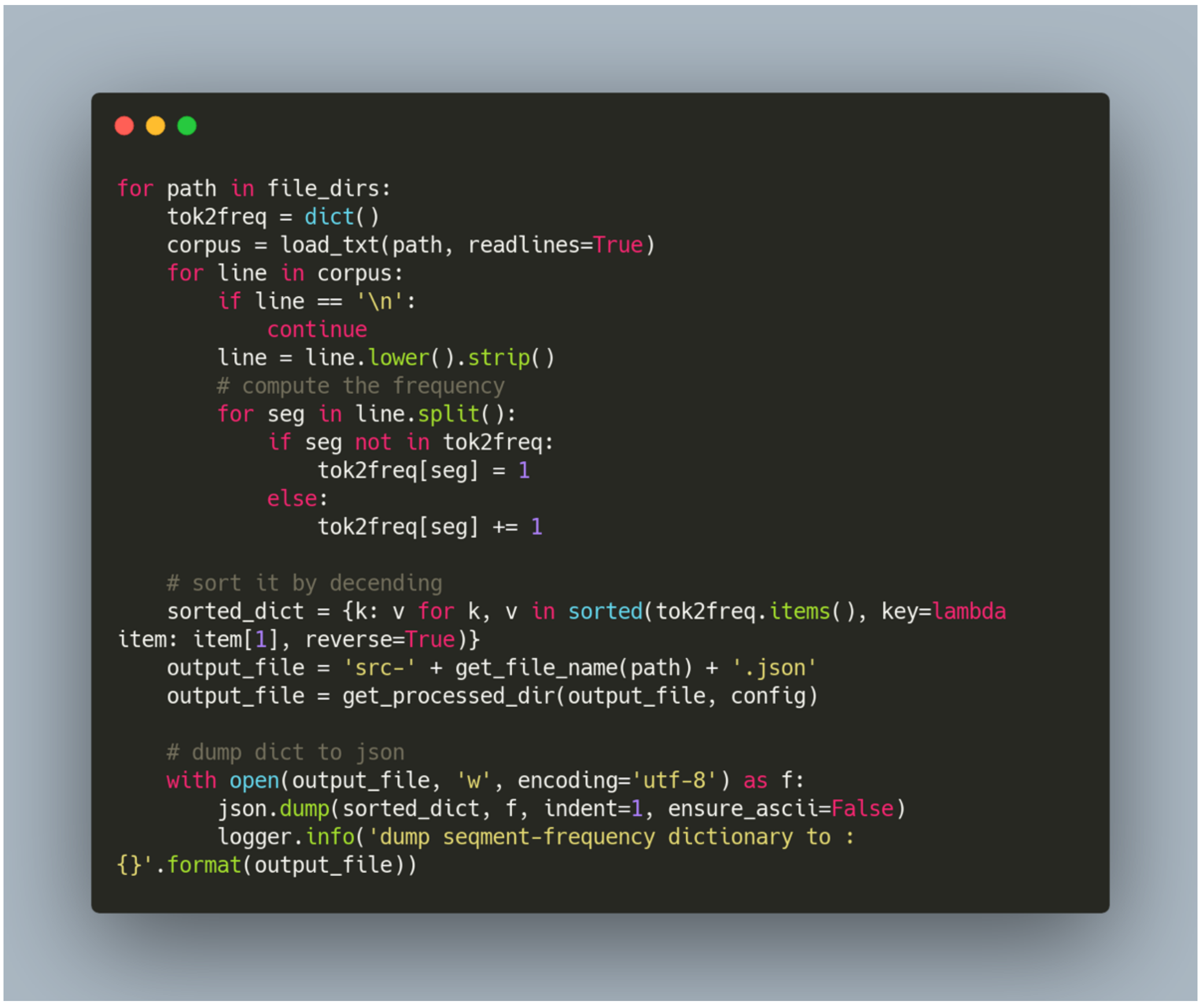
Zipf's law is one of the most powerful statistical law that the frequency distribution of words in a language obeys a linear pattern when plotting the word frequency and its rank. In this section, we empirically evaluate Zipf's law on 4 corpus and across 3 languages. We receive the similar results when plotting the Zipf's law by two charts with the linear axes and the log axes. We also demonstrate that the law holds in mostly parts of line, namely the line with mostly perfectly slope -1. We uses the King James Bible, the Jungle Book and the SEtimes Turkish-Bulgarian parallel newspaper text for our experiements.

1.1 Experiements

In this section, we briefly introduce our datasets: the King James Bible, the Jungle Book and the SETimesTurkish-Bulgarian parallel newspaper text. In addition, we descript our implemation detail here.

Dataset. In order to evalute the Zipf's law, we select four corpus to see whether it is true for different domains of text and languages. First, **the King James Bible (KJB)** is an English translation of the Christian Bible which consists of 31102 sentences. Second, **the Jungle Book** is a collection of stories which has roughly 54887 words counting from segments. For last two corpora, we use parallel news texts, **SETimes Turkish-Bulgarian corpus**, which base on the SETimes.com news portal and consist 206071 parallel sentences in both Turkish and Bulgarian language.

Experimental settings. We uses two scripts for building the data and ploting the Zipf's law charts separately. For each corpus, we count its word frequency and store word-frequency pair as kay-value mapping in a dictionary sorting by descending order. We tokenize sentence by whitespace and remain all stop words and punctuation remarks. Because it does not effect the result of zipf's law. In implementation, we collect all corpus on a folder and preprocess them within a loop. We dump the word-frequnecy pairs to a `.json` file for each corpus.



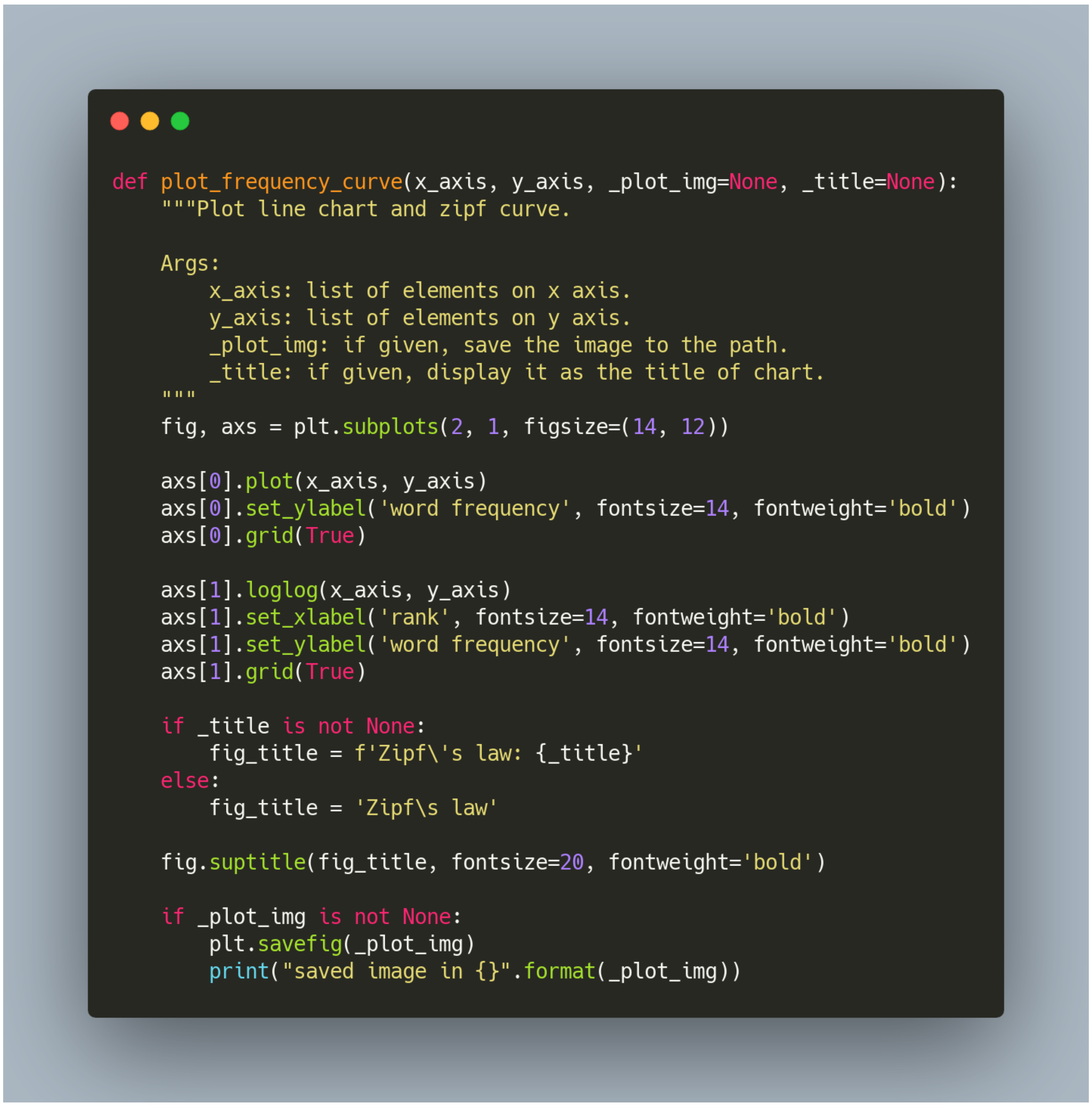
```
for path in file_dirs:
    tok2freq = dict()
    corpus = load_txt(path, readlines=True)
    for line in corpus:
        if line == '\n':
            continue
        line = line.lower().strip()
        # compute the frequency
        for seg in line.split():
            if seg not in tok2freq:
                tok2freq[seg] = 1
            else:
                tok2freq[seg] += 1

    # sort it by decending
    sorted_dict = {k: v for k, v in sorted(tok2freq.items(), key=lambda
item: item[1], reverse=True)}
    output_file = 'src-' + get_file_name(path) + '.json'
    output_file = get_processed_dir(output_file, config)

    # dump dict to json
    with open(output_file, 'w', encoding='utf-8') as f:
        json.dump(sorted_dict, f, indent=1, ensure_ascii=False)
        logger.info('dump seqment-frequency dictionary to :
{}'.format(output_file))
```

Figure 1. We create a dictionary of word-frequency pairs for each corpus aund dump it to a `.json` by descending order.

In the second program `run_zipf.py`, we load the word-frequency pairs from JSON file. In order to compare the results of different corpus, we draw the linear and the log line charts in one image using `plt.subplots` object. It creates a figure and a set of subplots for us. It's convience to us for observing the results when displaying horizontally. We will display the results and disccuss it on next section. For intance, our plotting function was designed to draw both the linear and the log line charts by using the methods `plt.subplots.plot()` and `plt.subplot.loglog()` provided by `matplotlib` library. We demontrate the function on below:



```python
def plot_frequency_curve(x_axis, y_axis, _plot_img=None, _title=None):
    """Plot line chart and zipf curve.

    Args:
        x_axis: list of elements on x axis.
        y_axis: list of elements on y axis.
        _plot_img: if given, save the image to the path.
        _title: if given, display it as the title of chart.
    """
    fig, axs = plt.subplots(2, 1, figsize=(14, 12))

    axs[0].plot(x_axis, y_axis)
    axs[0].set_ylabel('word frequency', fontsize=14, fontweight='bold')
    axs[0].grid(True)

    axs[1].loglog(x_axis, y_axis)
    axs[1].set_xlabel('rank', fontsize=14, fontweight='bold')
    axs[1].set_ylabel('word frequency', fontsize=14, fontweight='bold')
    axs[1].grid(True)

    if _title is not None:
        fig_title = f'Zipf\'s law: {_title}'
    else:
        fig_title = 'Zipf\s law'

    fig.suptitle(fig_title, fontsize=20, fontweight='bold')

    if _plot_img is not None:
        plt.savefig(_plot_img)
        print("saved image in {}".format(_plot_img))
```

Figure 2. The plotting function takes two lists as input for drawing the x-axis, y-axis in linear and log scale. If argument `_plot_img` is given, the figure will be saved in the path.

2.1 Discussion

Visualization of the word freqency In this section, we draw the results of the King James Bible and the Jungle Book and observed that high frequency words occur way more than other words for each corpus. The slope of high frequency words is vertical line comparing to the rest of word frquencies. In addition, it displays a almost perfect line with slope -1 in both figures when applying logarithm to the frequency and its rank.

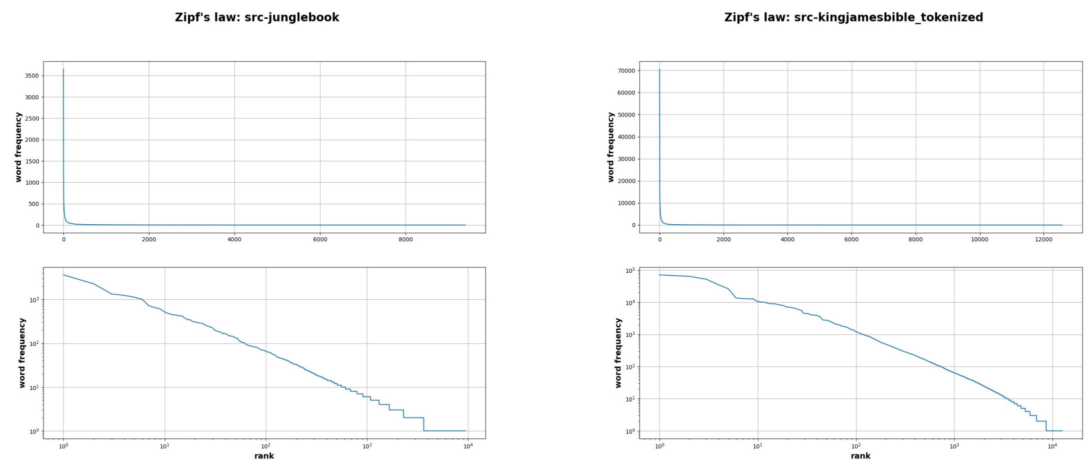


Figure 3. The plotting results of the King James Bible and the Jungle Book.

To evalute the statistical law on non-English corpus, we plot the results on the news in Bulgarian and Turkish here. Similarly, the highest frequecy words display as a vertical line and are the most majority of each text. And the log-scale line are still a line with pretty close to solpe -1. We can say that the Zipf's law holds on different genre of texts, such as religion, story and news. Meanwhile, it is true for text in different language, such as in Bulgarian and Turkish.

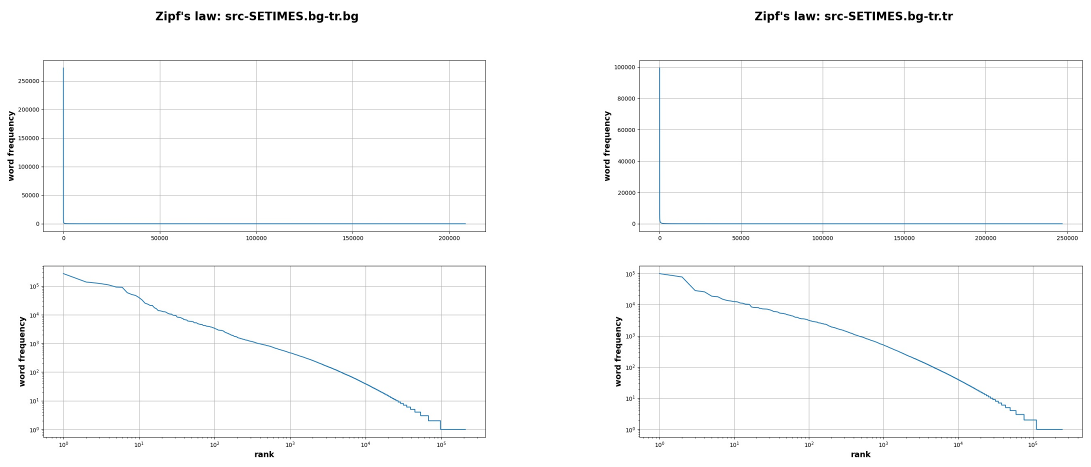


Figure 4. The plotting results of the SEtimes news in Bulgarian and in Turkish.

To conclusion, we demonstrate that 4 corpora in different domains and in different langages have similar results and the Zipf's law makes most erros for lowest frequency words.

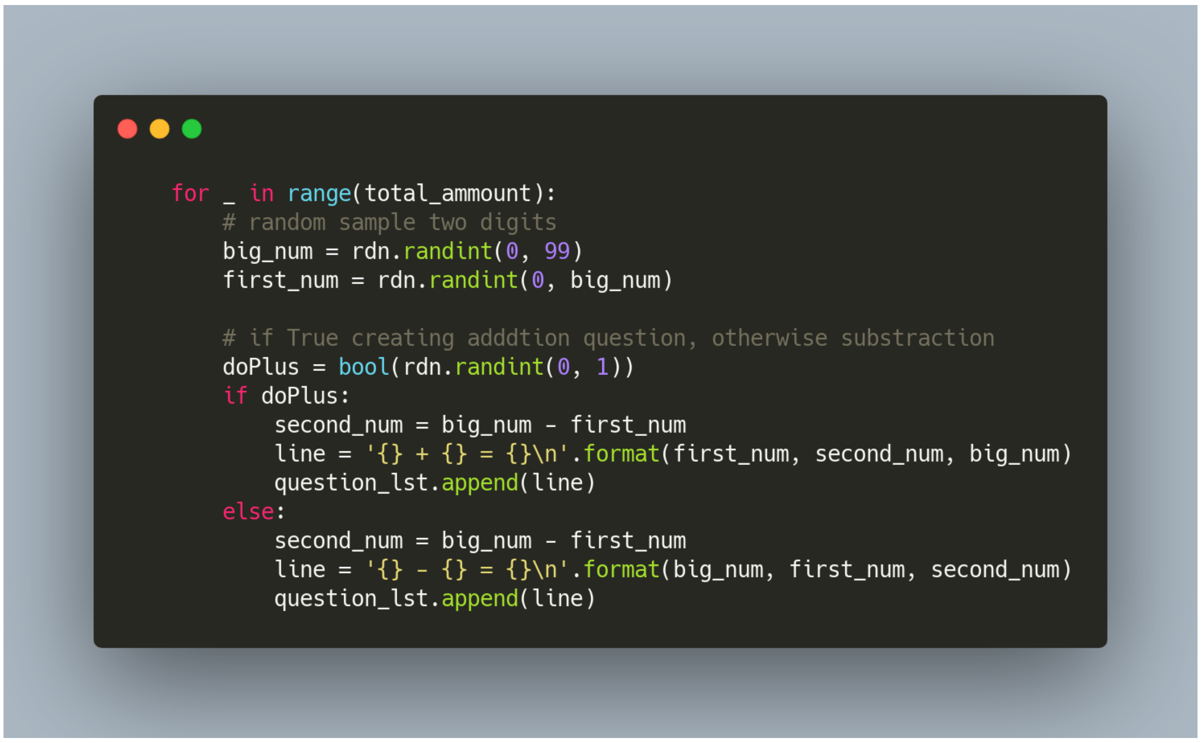
2. Random Text Generation

In this part, we train ngram model on arithmetic questions with different context size. Supprisingly, the 5-gram model is able to form correct equation and solve the questions when the model has seen the completed equation during training.

3.1 Problem Definition and Generating Training Data

Problem Definition Textual generative model has shown promissing results in recent years. In this section, we try to train our ngram model to solve simple 2 digits arithmetic questions for addition and substraction operation.

Generating Training Data We generative **10000** artifical questions by simply random sampling a number between [0, 99] and substract it by a small number to form an addtion or substraction quation. In our training data, we add a separate token `[SEP]` betwenn each equation, for example `3 + 2 = 5 [SEP] 13 - 1 = 12` , to provide the equation's boundary. We train 2-gram, 3-gram, 4-gram and 5-gram models to see whether it can generate correct token or solve the equation. We discuss the results in next section.



```python
for _ in range(total_ammount):
    # random sample two digits
    big_num = rdn.randint(0, 99)
    first_num = rdn.randint(0, big_num)

    # if True creating adddtion question, otherwise substraction
    doPlus = bool(rdn.randint(0, 1))
    if doPlus:
        second_num = big_num - first_num
        line = '{} + {} = {}\n'.format(first_num, second_num, big_num)
        question_lst.append(line)
    else:
        second_num = big_num - first_num
        line = '{} - {} = {}\n'.format(big_num, first_num, second_num)
        question_lst.append(line)
```

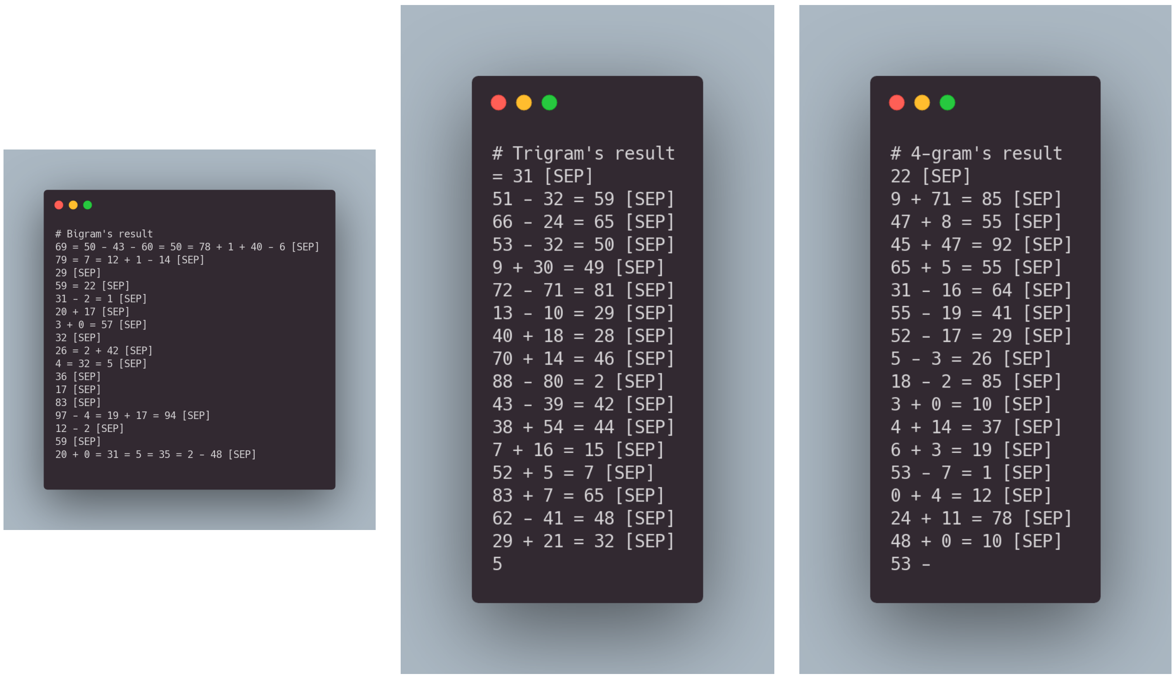
Figure 5. The function generates arithmetic questions by sampling a flag for addition and substraction.

3-2 Experment and Results

As mention before, we creates the training data for training different ngram model. Each sequence of tokens is a simple equation for addition or substraction. We concatenate all equations by a separate token `[SEP]` to provide a break signal between each equation. We hope that `[SEP]` could provide useful information to ngram model for generating correct token when given this separate token.

2, 3, 4-gram Due to the context size, these n-gram models can not seen the completed addition or substraction question during training. Therefore, they aren't able to generate correct result to the arithmetic questions.

But since the training data has the same pattern, they are able to generate "right" token after the context. For example, it always generates a digit after token `=` or after separate token `[SEP]` and don't generate operation token, like `+`, `-`, given another operation symbol. In addition, the trigram and quadrigram models could form the question in correct way althought they can't sample correct digit to the questions.



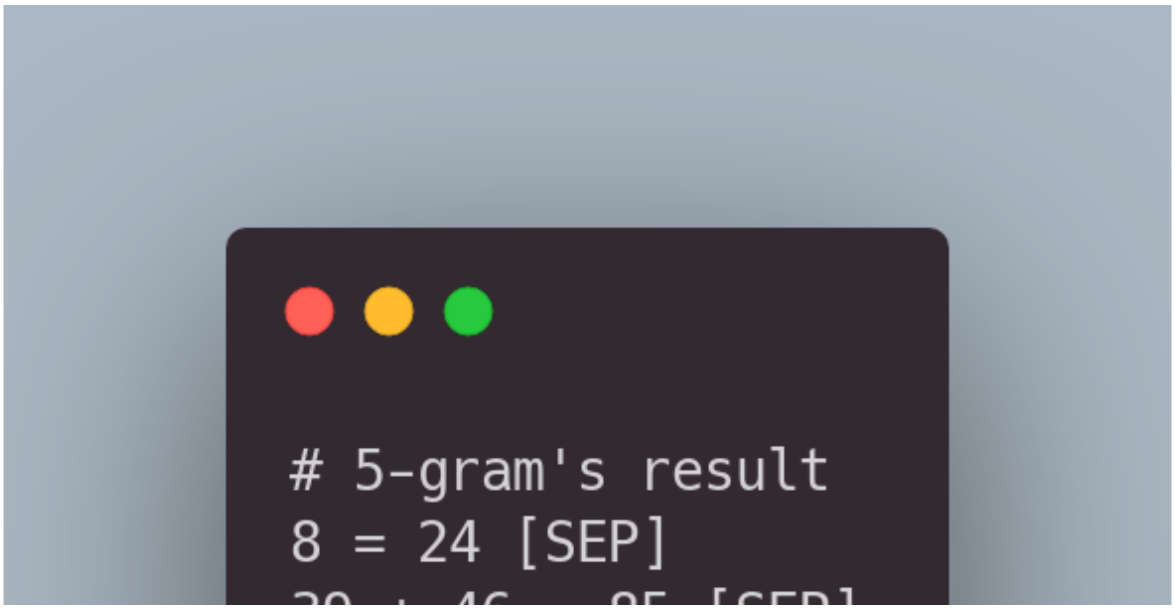
```
# Bigram's result
69 = 50 - 43 - 60 = 50 = 78 + 1 + 40 - 6 [SEP]
79 = 7 = 12 + 1 - 14 [SEP]
29 [SEP]
59 = 22 [SEP]
31 - 2 = 1 [SEP]
20 + 17 [SEP]
3 + 0 = 57 [SEP]
32 [SEP]
26 = 2 + 42 [SEP]
4 = 32 = 5 [SEP]
36 [SEP]
17 [SEP]
83 [SEP]
97 - 4 = 19 + 17 = 94 [SEP]
12 - 2 [SEP]
59 [SEP]
20 + 0 = 31 = 5 = 35 = 2 - 48 [SEP]
```

```
# Trigram's result
= 31 [SEP]
51 - 32 = 59 [SEP]
66 - 24 = 65 [SEP]
53 - 32 = 50 [SEP]
9 + 30 = 49 [SEP]
72 - 71 = 81 [SEP]
13 - 10 = 29 [SEP]
40 + 18 = 28 [SEP]
70 + 14 = 46 [SEP]
88 - 80 = 2 [SEP]
43 - 39 = 42 [SEP]
38 + 54 = 44 [SEP]
7 + 16 = 15 [SEP]
52 + 5 = 7 [SEP]
83 + 7 = 65 [SEP]
62 - 41 = 48 [SEP]
29 + 21 = 32 [SEP]
5
```

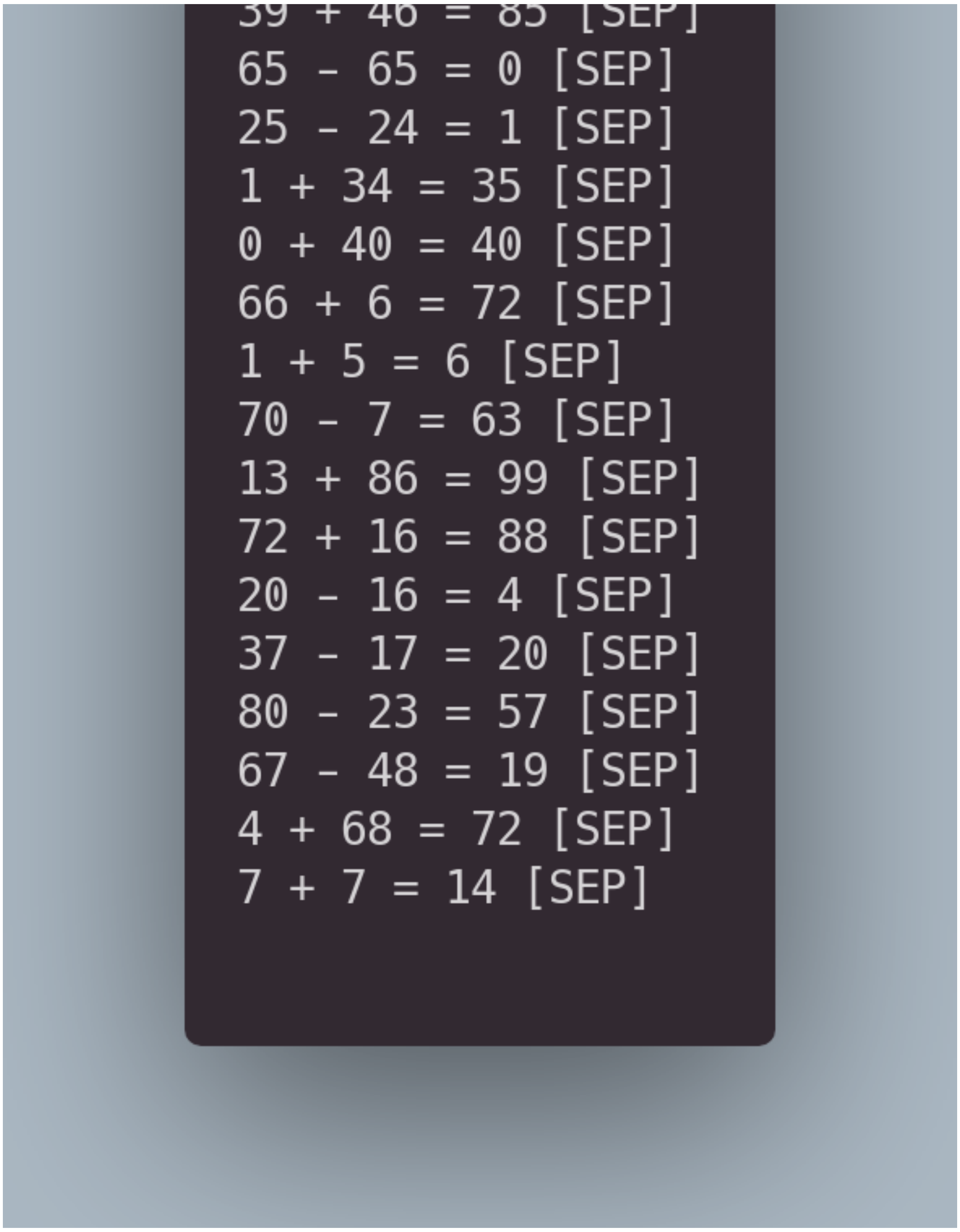
```
# 4-gram's result
22 [SEP]
9 + 71 = 85 [SEP]
47 + 8 = 55 [SEP]
45 + 47 = 92 [SEP]
65 + 5 = 55 [SEP]
31 - 16 = 64 [SEP]
55 - 19 = 41 [SEP]
52 - 17 = 29 [SEP]
5 - 3 = 26 [SEP]
18 - 2 = 85 [SEP]
3 + 0 = 10 [SEP]
4 + 14 = 37 [SEP]
6 + 3 = 19 [SEP]
53 - 7 = 1 [SEP]
0 + 4 = 12 [SEP]
24 + 11 = 78 [SEP]
48 + 0 = 10 [SEP]
53 -
```

Figure 6. The results of 2-gram, 3-gram and 4-gram models generating sequence of arithmetic questions.

5-gram The 5-gram model shows the surprising result to us. It can form 16 correct quations and give the correct number to them. Given a context like `1 + 34 =`, it can produce correct textual number 35 by sampling the distributions of token.



```
# 5-gram's result
8 = 24 [SEP]
```



```
39 + 46 = 85 [SEP]
65 - 65 = 0 [SEP]
25 - 24 = 1 [SEP]
1 + 34 = 35 [SEP]
0 + 40 = 40 [SEP]
66 + 6 = 72 [SEP]
1 + 5 = 6 [SEP]
70 - 7 = 63 [SEP]
13 + 86 = 99 [SEP]
72 + 16 = 88 [SEP]
20 - 16 = 4 [SEP]
37 - 17 = 20 [SEP]
80 - 23 = 57 [SEP]
67 - 48 = 19 [SEP]
4 + 68 = 72 [SEP]
7 + 7 = 14 [SEP]
```

Figure 7. The results of 5-gram model generating sequence of arithmetic questions.

3. Statistical Denpendence

In this section, we will apply pointwise mutual information on Yelp's reviews provided by [Li et al. (2018) (https://github.com/lijuncen/Sentiment-and-Style-Transfer)](https://github.com/lijuncen/Sentiment-and-Style-Transfer) and discuss the results fo independence assumption testing on negative reivews.

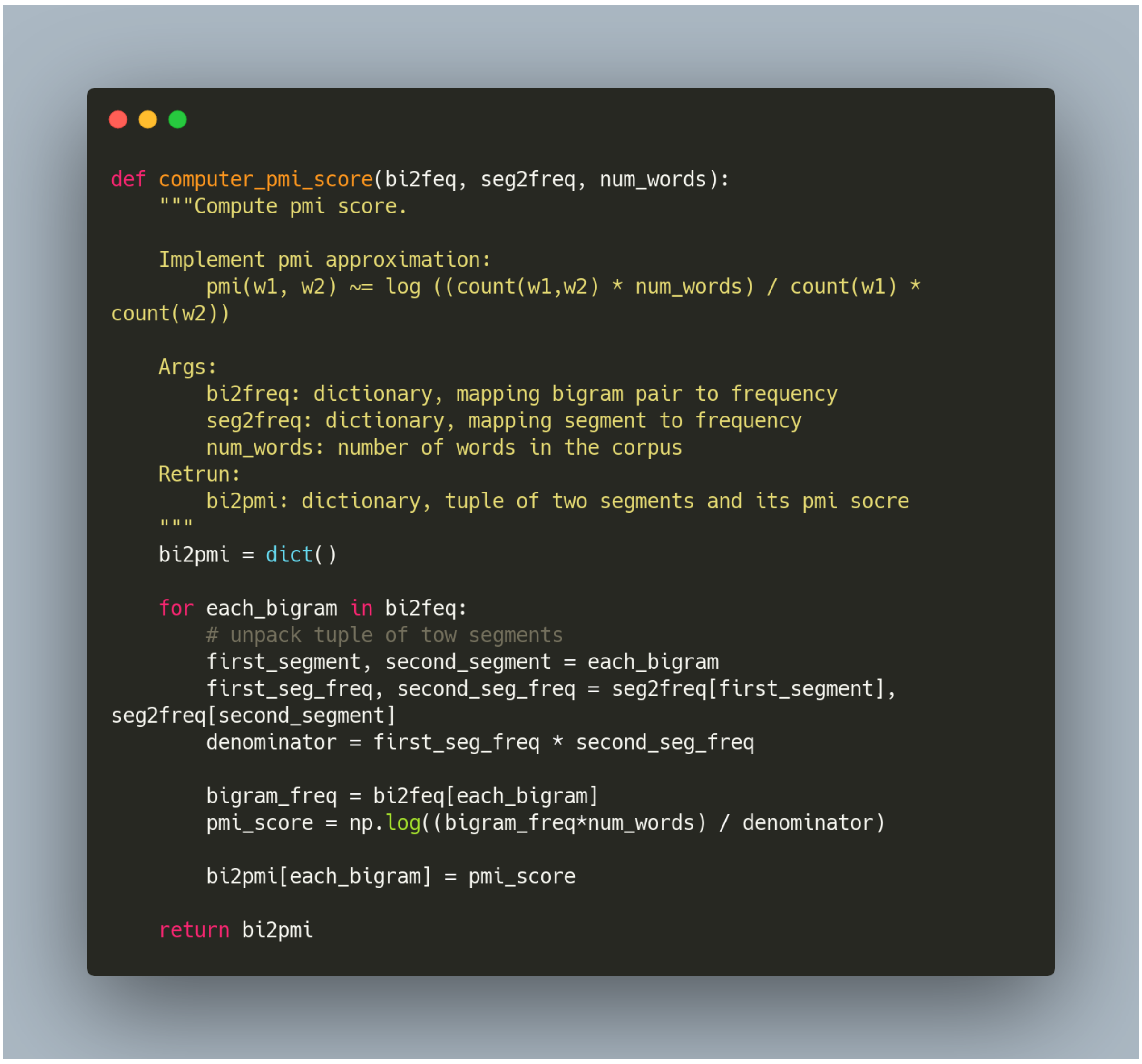
3.1 Data and experimental setting

Yelp reviews. This dataset was released for evaluating the sentiment transfer task in 2018. They consis 266041 postive and 177219 negative reviews on Yelp in the training set, 2000 reviews for each positive and negative for the validation set and 1000 reviews for testing set. We only use the 2000 negative reviews for our experiemnets.

Experimental setting. Similar to first section, we tokenize sentence by whitespace and do not remove any stop words and punctuation marks. We set a minimum frequency 10 for filtering words which occur less than the threshold. We compute the pointwise mutual information of two words by the approxiamtion below:

$$pmi(w_1, w_2) \approx log(\frac{C(w_1, w_2) * N}{C(w_1) * C(w_2)})$$

In implement, we create two dictionaries collecting bigram-frequency and segment-frequency pairs and compute all PMI score for word pairs which occur more than minimum frequency. We demonstrate the function implement in Python bellow:



```python
def computer_pmi_score(bi2feq, seg2freq, num_words):
    """Compute pmi score.

    Implement pmi approximation:
        pmi(w1, w2) ~= log ((count(w1,w2) * num_words) / count(w1) *
count(w2))

    Args:
        bi2freq: dictionary, mapping bigram pair to frequency
        seg2freq: dictionary, mapping segment to frequency
        num_words: number of words in the corpus
    Retrun:
        bi2pmi: dictionary, tuple of two segments and its pmi socre
    """
    bi2pmi = dict()

    for each_bigram in bi2feq:
        # unpack tuple of tow segments
        first_segment, second_segment = each_bigram
        first_seg_freq, second_seg_freq = seg2freq[first_segment],
seg2freq[second_segment]
        denominator = first_seg_freq * second_seg_freq

        bigram_freq = bi2feq[each_bigram]
        pmi_score = np.log((bigram_freq*num_words) / denominator)

        bi2pmi[each_bigram] = pmi_score

    return bi2pmi
```

Figure X. The function takes `bi2freq`, `seg2freq`, `num_words` as arguments and returing another dictionary object for bigram and PMI Score mapping.

3.2 Discussion

20 highest PMI socres. It showns two interesting facts in the rank. First, word pairs with higher PMI score are used to judge or express the sentiment of reviewers, like **(zero stars)**, **(worse than)**, **(avoid coming)** and **(poor quality)**. Second, they are reasonably a short phrase that people use a lot in daily life.

20 lowest PMI socres. The two adjacent words with lower PMI score occur rarely together or sometimes are typo. For example, the word pair **(to and)** is from the sentence: "*i do not steal , do n't have to and never have .*". The pair isn't a phrase and don't make sense when occurring togehter. And another example is the word pair 19 **(the the)** is a typo in the sentence: "*there is no plug for the the drain.*"

		Highest 20 sscores			**Lowest 20 scores**
rank	**words**	**PMI score**	**rank**	**words**	**PMI score**
1	zero stars	6.022394923337167	1	is .	-1.8507871228332147
2	stay away	5.91525028588584	2	to and	-1.9527187885548178
3	know how	5.752731356388065	3	and to	-1.9527187885548178
4	worse than	5.723981943102078	4	it i	-1.9583698954520934
5	walk away	5.65742117658374	5	to ,	-1.987715630591914
6	an hour	5.616599182063484	6	and ,	-1.987715630591914
7	your money	5.3850065762627475	7	i not	-1.9896923665811346
8	coming back	5.3472662482799	8	the is	-2.052036689512688
9	avoid coming	5.3472662482799	9	this .	-2.061412455790021
10	away from	5.334021021529879	10	to was	-2.071414649468561
11	else where	5.309114482315524	11	so .	-2.072973278191097
12	make sure	5.267223540606364	12	*num* the	-2.1607408911135777
13	slow slow	5.262108439939594	13	but .	-2.2021850096711035
14	poor quality	5.1978888472052995	14	not .	-2.338867996457971
15	as well	5.074274891238122	15	do .	-2.419249514908931
16	first off	5.070055475695415	16	n't .	-2.516036799856269
17	`` quality	5.051285373013424	17	for .	-2.6440177619501424
18	point where	5.051285373013424	18	i the	-3.1215207052577743
19	: where	5.051285373013424	19	the the	-3.2664807937059197
20	better than	5.0055169545578435	20	was .	-3.625658374032748

To conclusion, the probability of two random words w1 and w2 has its value when they were observed in the corpus, othwise the probability of pmi(w1, w2) will be zero. However, we use really small corpus that can not capture every grammatical correct or meaningful phrase of english language. For example, the PMI score for a common phrase **(take into)** is 0 since the word pair doesn't appear in the dataset. In general, we can add a small absolute value for C(w1, w2) to avoid the probability of these two words beiing 0. In the future, we would like to modify the approximation by some smoothing techniques.