Trigram Model: Theory and Practice

1 Preprocessing each line

Our main task is to train a character-based language model identify different languages. Inconsistent symbols, or redundant punctuations do not convey much information and will bias our identification task; therefore, we intend to preprocess each training text by lines, retaining only English Alphabets, spaces, digits (all converted to 0) and periods. In addition, we reckon extra attention paid to the new line character '`\n`', which can be easily mistaken for a space.

Based on phonotactics and morphotactics of natural languages, some morphemes or phoneme combinations are inhibited; naturally, some characters are not likely to occur at the beginning or the end of a sentence. To capture this property, we add start symbols and end symbols to each line. From the perspective of probability, the character-based language model is aiming for predicting the next character based on the context (previous two characters), in which start symbols and end symbols can be the forward reference and backward reference to make the trigram grammar a valid probability distribution.

```python
def _preprocess_line(self, raw_line):
    '''
    Convert raw lines to the required format". Requirements are as follows:
        1. Retain characters like English alphabets, spaces, digits, or '.' characters exclusively.
        2. Lowercase all characters
        3. Convert all digits to 0
    param:
        raw_line (str): a sentence in its raw format
    Output:
        processed_line (str): the sentence in the required format
    '''
    char_list = [char for char in raw_line]
    processed_list = []
    for char in char_list:
        if re.match(r"[a-zA-Z]", char) or char.isspace():
            processed_list.append(char.lower())
        elif char.isdigit():
            processed_list.append('0')
        elif char == '.':
```

```
            processed_list.append('.')
    processed_line = ''.join(processed_list).replace('\n', '')

    return processed_line

def _add_symbols(self, processed_line):
    '''
    Add start and end symbols to processed lines. In other words, adding
    ## to the start and the end of previously processed line
    param:
        processed_line (str): a sentence in the required format but
    without symbols
    Output:
        complete_line (str): the sentence in the required format with
    symbols.
    '''
    start_symbol, end_symbol = self.symbol, self.symbol
    num_symbol = self.n_gram - 1
    added_start_symbols, added_end_symbols = start_symbol * num_symbol,
        end_symbol * num_symbol

    complete_line = added_start_symbols + processed_line +
        added_end_symbols

    return complete_line
```

2 Examining a pre-trained model

In the given language model, we would assume that it has implemented Maximum Likelihood Estimation(MLE) along with add-α smoothing estimation. In the trigram distribution, we noticed loads of trigrams share the same estimated probability; for example, any character given the previous two characters are '#', space, has the probability of 3.448e-02, that is, these trigrams share the same frequency in the corpus, which is quite rare in natural language data. Another counterintuitive observation is that $P(\mathit{space} \mid \texttt{\#\#})$ is not zero, whereas most space will not be the first character of a sentence.

Besides, we also conducted some experiments to narrow the range of α based on the given training data. According to the perplexity shown in Section 5, there is a huge gap of perplexity between our implemented add-α (where $\alpha = 0.8$) smoothing algorithm and the given model. We assumed it is a result of a large α, since if the probability of each trigram is nearly the same, then the model will guess the third character according to a uniform distribution by giving the first two characters. Essentially, the model is not meaningful when the trigram distribution is uniform, as it does not learn anything but only takes stochastic guesses. To achieve the uniform distribution, the α is supposed to be an extremely large number; therefore, the frequency of each trigram is approximately the same. Moreover, the value of α should relate to the corpus size, whereas we do not have for given model thus fail to take it into consideration. By setting α to $10^1, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7$, the perplexities are 10.14, 16.17, 25.37, 29.28, 29.91, 29.98, 29.98. Following the increase of α, the perplexity increases, which means the performance of the character-based language model becomes poorer. We narrowed the range of α and found that when setting α to 400, the perplexity of our implemented model (22.02) approximately equals that of given model (22.09). Hence, we guess the pre-trained model is trained through add-α smoothing estimation but with an inappropriately larger α value.

3 Implementing a model: description and example probabilities (35 marks)

3.1 Model description

We built a character-based trigram language model, estimating the probability distribution based on MLE and add-α smoothing. The combination of MLE and add-α smoothing is a straightforward and effective way deal with error handling by mannual add α to all possible trigrams in case that some trigrams, not shown in training data, cannot be identified by the language model. In addition, add-α smoothing can help on the problems of overfitting. According to MLE, our trigram parameter estimation will follow:

$$P(w_n|w_{n-1}, w_{n-2}) = \frac{C(w_{n-2}, w_{n-1}w_n)}{C(w_{n-2}w_{n-1})} \tag{1}$$

Therefore, estimated probability of a trigram equals the frequency that it appears in the corpus, divided by its history counts - the counts of trigrams sharing the same first two characters. MLE can well reflect the probability of high-frequency trigrams, whereas, if a trigram (in testing data) has never occurred in the training data, the method ignores the probability on the existence of the trigram, which is not the fact in most cases since training data could be limited or language keeps evolving. For solving this issue, we add a value α (initially set to 1 and finally adjusted to 0.8, full details are in Appendix: Experimental Results) to each possible trigram (the size of possible trigrams is marked as V below) to smooth our results from MLE. After introducing a hyperparameter to our original estimation, each trigram parameter changed based on:

$$P(w_n|w_{n-1}, w_{n-2}) = \frac{C(w_{n-2}, w_{n-1}w_n) + \alpha}{C(w_{n-2}w_{n-1}) + V\alpha} \tag{2}$$

3.2 Model excerpt

For potential characters after 'ng', we usually would expect nothing since 'ng' is usually part of the morpheme 'ing'. If we consider 'ng' as one sound unit, which is usually phonemically realised as /ŋ/, it is always in a coda position and inhibited from appearing in an onset position in English. It is also not surprising that 'ngf', 'ngt', and 'ngs' are with higher probabilities because 'ng' does not have to be in the same morpheme or phoneme, which is rare(i.e., gingf, averdoingt). As for 'ngs', that could be a result of gerunds' pluralisation, which is not common as well (i.e., belongings but not swimmings).

```
{'ng ': 0.793155893536123, 'ng#': 0.0022813688212927796, 'ng.':
↪  0.026362484157161, 'ng0': 0.0010139416983523466, 'nga':
↪  0.0035487959442332124, 'ngb': 0.0010139416983523466, 'ngc':
↪  0.0010139416983523466, 'ngd': 0.004816223067173646, 'nge':
↪  0.08593155893536136, 'ngf': 0.0022813688212927796, 'ngg':
↪  0.0010139416983523466, 'ngh': 0.0010139416983523466, 'ngi':
↪  0.0022813688212927796, 'ngj': 0.0010139416983523466, 'ngk':
↪  0.0010139416983523466, 'ngl': 0.0035487959442332124, 'ngm':
↪  0.0010139416983523466, 'ngn': 0.0022813688212927796, 'ngo':
↪  0.007351077313054512, 'ngp': 0.0010139416983523466, 'ngq':
↪  0.0010139416983523466, 'ngr': 0.012420785804816245, 'ngs':
↪  0.021292775665399277, 'ngt': 0.013688212927756679, 'ngu':
↪  0.0035487959442332124, 'ngv': 0.0010139416983523466, 'ngw':
↪  0.0010139416983523466, 'ngx': 0.0010139416983523466, 'ngy':
↪  0.0010139416983523466, 'ngz': 0.0010139416983523466}
```

4 Generating from models

The basic algorithm of this trigram model is predicting the next character conditioned on the previous two characters. Therefore, in our generating function, we applied a sliding window at a position to condition on the previous two characters, and then predict the character at the position by a non-deterministic approach based on a trigram probability distribution. Every time generating a character, the sliding window will move a step further to condition on the next two new characters. The non-deterministic approach is to randomly select the characters according to the probabilities. A trigram is more likely to be selected if it has a larger probability according to the distribution of the first two characters. However, this non-deterministic approach aims not to select the trigram with the highest probability but with a high rate, which determines the non-uniqueness of the generated text. Moreover, we need to set some proper rules for start symbols and end symbols for not invalidating the text. In our function, we do not count any start symbol or end symbol into the length. For start symbols, the generation of first character should always be conditioned on two starting symbols. It is not sensible to generate the end symbols when the text does not reach the length. Hence, this function will not consider generating end symbols by removing the probability of trigrams involving end symbols, and manually adding end symbols when halting.

```
'From trained English language model'
'##ing to ambpurese of to ound intions alf auuucras rate iver.fxjpfact of the res subts ou the betich evies.ysw.0zcpwcgeopolvent her sm parlies wo.dsdxmosidtes a poliarow hav ea plecg0bic of mrstion ampotimosten porecesis and spartureand the its it the offeed econ nominiell lobjecord thasto coadied ach##'

'From given English language model'
'##ye hose do you.aoes it.igh.u0yubye.hnwblos agong that ducks..zip itthat turt.00cgue.n you napplethin the mor.kjgc zippy on he thi.in it.sdphou le shund we dow thes are wany.gl you does thats the tere therefragook ats to you my bund.bbigh.ehpdvcpy.jmwrrouts at to you wat.wjhn.wmally.lice.t is.rrobbit##'

```

5 Computing perplexity

To evaluate how well our models on predicting the data, we should fit our models to dev sets and calculate the perplexity based on the equation below:

$$PP_M(\vec{w}) = P_M(w_1...w_n)^{-\frac{1}{n}} \tag{3}$$

However, we do not have given dev sets to fit our models, then we have to fit them directly to the test data, which might have potential overfitting issues. Another issue is the simple product of each trigram's probability based on our model will be close to zero, which leads to some non-obvious results. Therefore, we will utilise cross-entropy, which is a logarithmic version of calculating perplexity and will give us an obvious number. All the perplexity is measured based on the add-α (where $\alpha = 0.8$) smoothing.

$$PP_M(\vec{w}) = 2^{log_2 P_M(w_1...w_n)^{-\frac{1}{n}}} \tag{4}$$

Results:

```
'perplexity based on training.en': 8.858391703180724
'perplexity based on given model': 22.094457902881697
'perplexity based on training.de' : 23.408273543425153
'perplexity based on training.es' : 22.9957778613915
```

From the perplexity we gained, we can infer that our model based on 'training.en' is better at predicting the data set we fit in, because according to our equation, the higher the perplexity, the lower $P_M(w_1...w_n)$. If we fit our English model to a new test data, and we get a higher perplexity, which means our model is not generalised enough to make predictions or the test data is noisy (perhaps not in English). However, this perplexity of our model is not sound enough, which probably results from the limited training data as only 1000 lines are involved in training data. Another reason could be the imperfect selection of smoothing algorithm as add-α is limited in many aspects.

6 Further improvements

In this section, we will seek further improvements based on some intuitions from the aspects of linguistics. Our core task is to conduct language identification. Therefore, firstly, we need to consider for a character-based model how languages are different in a 'character' sense, based on which we make further developments. Obviously, we have processed them to be the same in a semiotic sense; however, we could make provisional assumptions based on the random tri-characters, which provide helpful but subtle information for morphological parsing and word formation. Note that our proposes are some shallow aspects of morphological parsing, while the vision is much more complicated to handle. We will mainly take advantage of inflectional morphemes for two reasons: (i) Inflectional morphemes usually are less varied and have a strict selectional property (i.e., derivational affix like -ly can be attached to either a noun or a verb, whereas inflectional ones like -ing or -ed only go behind a verb.). (ii) Inflectional morphemes tend to have a broader usage, thus a higher frequency in any possible text.

Moreover, another idea might be valuable is whether there is a distinction clear enough to help us distinguish among different languages. Morpheme borrowing happens all the time, and the historical development which leads to language variation are also overlapping in the same language family (i.e., big vowel shift and umlaut in germanic languages $\rightarrow$ corresponding typological change). Despite some similarities, we can still point out the distinct morphemes if we did a morphological investigation on an individual language basis. Currently, we reckon that those annotations possibly will be done manually because of lack of understanding of cross-linguistic morphology and if we want to train a classifier to distinguish different morphemes that belong to different languages, that would be beyond the scale of this assignment. Then the basic idea of developments we could temporarily make is to highlight the language-specific morphemes, for example, we can get an average proportion of a language-specific morpheme, say, -ing from a English corpus(training data), and if that rate for another text is around or even higher, then it is likely to be in English. Similarly, we could use -ung in Geman identification task. However, there are exceptions to any rules, and these 'distinct' morphemes might happen to occur randomly (i.e., ung is also a part of English preterite stung) or in some unknown syntactic structure but at least what we proposed will increase the identification rate, and ideally in a larger corpus.

Word formation is another factor we think may give extra information about a language. However, most languages share the same word formation rules like compounding, blending and etc with the proportion differing a bit though. We believe this idea sheds light on morpheme extraction and benefit language identification on a morpheme-based model.

Appendix: your code

Include a verbatim copy of your code for questions 1-5 here. If you answered question 6, you do *not* need to include that code.

Proprocessing:

```python
import re
from math import log
from collections import defaultdict
import json
from random import choices


class Character_Model():

    def __init__(self,
                 training_data_path,
                 distribution_write_path = '',
                 write_distribution_status = False,
                 smoothing_status = True,
                 load_distribution_locally = False,
                 distribution_file_path = 'trigram_distribution.json',
                 symbol = '#',
                 alpha = 0.8,
                 n_gram = 3):
        '''
        This class is for building a character based model for language identification.
        Params:
            training_data_path (str): path of training data
            distribution_write_path (str): path to write the probabilities of trigrams
            write_distribution_status (bool): whether to write distribution
            smoothing_status (bool): whether to apply add-alpha smoothing
            load_distribution_locally (bool): for saving the time, after generating the distribution file, it's not meaningful to compute the distribution every time creating the object. Whether to load the distribution from generated json file
            distribution_file_path (str): path of a distribution
            symbol (char): start and end symbol of lines
            alpha (float): alpha value for add-alpha smooth
```

```
            n_gram (int): here we set to 3 because this coursework is aiming to solve the problems of trigram

        '''
        self.training_data_path = training_data_path
        self.distribution_write_path = distribution_write_path
        self.n_gram = n_gram
        self.alpha = alpha
        self.symbol = symbol

        self.smoothing_status = smoothing_status

        self.n_gram_frequency = self.get_n_gram_frequency(self.training_data_path)
        if self.smoothing_status:
            self.add_alpha_smoothing()

        # load distribution from local file (computing distribution consumes a lot of time when smoothing algorithms are applied)
        if load_distribution_locally:
            with open(distribution_file_path, "r") as f:
                self.n_gram_distribution = json.load(f)
            pass
        else:
            self.n_gram_distribution = self.get_n_gram_distribution(self.n_gram_frequency)

        if write_distribution_status:
            self.write_n_gram_distribution()


    def _preprocess_line(self, raw_line):
        '''
        Convert raw lines to the required format". Requirements are as follows:
        1. Retain characters like English alphabets, spaces, digits, or '.' characters exclusively.
        2. Lowercase all characters
        3. Convert all digits to 0
    param:
        raw_line (str): a sentence in its raw format
    Output:
        processed_line (str): the sentence in the required format
```

```
        '''
        char_list = [char for char in raw_line]
        processed_list = []
        for char in char_list:
            if re.match(r"[a-zA-Z]", char) or char.isspace():
                processed_list.append(char.lower())
            elif char.isdigit():
                processed_list.append('0')
            elif char == '.':
                processed_list.append('.')
        processed_line = ''.join(processed_list).replace('\n', '')

        return processed_line

    def _add_symbols(self, processed_line):
        '''
        Add start and end symbols to processed lines. In other words, adding ## to the start and the end of processed line
        param:
            processed_line (str): sentence in the required format but without symbols
        Output:
            complete_line (str): sentence in the required format with symbols.

        '''
        start_symbol, end_symbol = self.symbol, self.symbol
        num_symbol = self.n_gram - 1
        added_start_symbols, added_end_symbols = start_symbol * num_symbol, end_symbol * num_symbol

        complete_line = added_start_symbols + processed_line + added_end_symbols

        return complete_line


    def get_n_gram_frequency(self, data_path):
        '''
        1. Load the training data
        2. Preprocess and add symbols on training data. Count on the n-gram frequency
        3. return the dict of n-gram frequencies
```

```python
        '''

        n_gram_frequency = defaultdict(int)

        with open(data_path) as f:
            for line in f:
                line = self._preprocess_line(line)
                line = self._add_symbols(line)

                for j in range(len(line) - self.n_gram):
                    n_gram = line[j: j + self.n_gram]
                    n_gram_frequency[n_gram] += 1

        return n_gram_frequency

    # specific trigram function
    def add_alpha_smoothing(self):
        '''
        This function can only applied to trigram model.
        Apply add alpha smoothing to the language model:
            1. get the set of characters
            2. get all possible trgrams (removing some impossible cases such as: 'a#a' '###')
            3. Update the trigram frequency dict. Essentially, add alpha smoothing is to change the frequency of each trigram.

        Param:
            self.symbol: start and end symbol of lines
            self.alpha: the value of alpha added to frequency of every trigram
        '''
        # get the character set
        char_vocab = []
        for key in self.n_gram_frequency.keys():
            char_vocab += set(list(key))
            char_vocab = list(set(char_vocab))

        trigram_vocab = []
        # get all possibie trigrams
        for i in range(len(char_vocab)):
            for j in range(len(char_vocab)):
                for k in range(len(char_vocab)):
                    trigram_vocab.append(f'{char_vocab[i]}'+
```

```
                                         f'{char_vocab[j]}'+
                                         f'{char_vocab[k]}')

        # remove the impossible trigrams (patterns)
        trigram_vocab.remove(self.symbol * 3)
        for trigram in trigram_vocab:
            if trigram[0] == self.symbol and trigram[2] == self.symbol:
                trigram_vocab.remove(trigram)
                continue

            if trigram[1] == self.symbol and (trigram[0] != self.symbol and trigram[2] != self.symbol):
                trigram_vocab.remove(trigram)

        # update the trigram frequency dict

        for trigram in trigram_vocab:
            if trigram in self.n_gram_frequency.keys():
                self.n_gram_frequency[trigram] += self.alpha
            else:
                self.n_gram_frequency[trigram] = self.alpha

    def _calculate_MLE(self, n_gram_name, frequency_dict):
        '''
        Calculate the maximum likelihood estimation of a specific n_gram based on the n_gram frequency
        Params:
            n_gram_name (str): name of n_gram
            frequency_dict (dict): dictionary used to store the frequencies of all n-grams
        '''
        frequency = 0
        n_gram_frequency = frequency_dict[n_gram_name]

        # n_grams which shares the same n-1 characters
        n_gram_pool = [n_gram for n_gram in frequency_dict.keys() if n_gram.startswith(n_gram_name[0: self.n_gram - 1])]
        for n_gram in n_gram_pool:
            frequency += frequency_dict[n_gram]

        mle = n_gram_frequency / frequency

        return mle
```

```python


    def get_n_gram_distribution(self, n_gram_frequency):
        '''
        Based on the n-gram frequency, calculating the probability of each n-gram
        '''
        n_gram_distribution = {}
        # need to consider the unseen trigram toghter with smoothing
        for n_gram in sorted(n_gram_frequency.keys()):
            mle = self._calculate_MLE(n_gram, n_gram_frequency)
            n_gram_distribution[n_gram] = mle

        return n_gram_distribution

    def write_n_gram_distribution(self):
        '''
        Write the n-gram distribution to a file in the JSON format
        '''
        write_path = self.distribution_write_path
        with open(f"{write_path}", "w") as outfile:

            json.dump(self.n_gram_distribution, outfile)

            print(f"file has been written at {write_path}")


    # here trigram
    def get_n_gram_probability(self, n_gram_head):
        '''
        Get trigram probability based on the given first two characters (trigram head)
        '''
        result = {}
        n_gram_pool = [n_gram for n_gram in self.n_gram_distribution.keys() if n_gram.startswith(n_gram_head)]

        for n_gram in n_gram_pool:
            result[n_gram] = self.n_gram_distribution[n_gram]


        return result
```

```python


class Pre_trained_model():

    def __init__(self,
                 model_path = "./model-br.en"):
        '''
        class for the given model
        '''
        self.model_path = model_path
        self.model_distribution = self.load_pre_trained_model()



    def load_pre_trained_model(self):
        '''
        Load the pre-trained model and get distribution of n-grams
        '''
        n_grams, n_gram_distribution = [], {}

        with open(self.model_path, 'r') as f:
            for line in f:
                n_grams.append(re.sub(r"[\n\t]*", "", line))
        for n_gram in n_grams:
            n_gram_distribution[n_gram[0: 3]] = float(n_gram[3: ])

        return n_gram_distribution


def generate_from_LM(distribution, length = 300, start_symbols = '##', end_symbols = '##'):
    '''
    In this function, we manually set the start or end symdols don't count on the length. In addition,
    we manually add '##' to the start and then the generation of first character depending on '##' and trigram distribution.
    However, during generation, if the current length of a line doesn't reach the limit, we remove all the possibilities of trigrams involving '#'.

    '''
    generated_txt = ''
    # add the start symbols, if trigram => start symbols = '##'
```

```python
    generated_txt += start_symbols

    while len(generated_txt) < length + 2:

        trigram_head = generated_txt[-2: ]

        trigram_pool, trigram_probability = [], []
        for key, value in distribution.items():
            if key.startswith(trigram_head):
                if key[-1] != '#':
                    trigram_pool.append(key); trigram_probability.append(value)

        choice = choices(trigram_pool, trigram_probability)[0]
        generated_txt += choice[-1]

    generated_txt += end_symbols
    return generated_txt


def read_test_file_by_lines(path):

    with open(path) as f:
        lines = f.readlines()
        lines = [line.rstrip() for line in lines]

    return lines

def preprocess_line(raw_line):

    char_list = [char for char in raw_line]
    processed_list = []
    for char in char_list:
        if re.match(r"[a-zA-Z]", char) or char.isspace():
            processed_list.append(char.lower())
        elif char.isdigit():
            processed_list.append('0')
        elif char == '.':
            processed_list.append('.')
    processed_line = ''.join(processed_list).replace('\n', '')

    return processed_line

```

```python
def preprocess_test_data(test_data_lines):

    processed_data = []
    for line in test_data_lines:
        processed_line = '##' + preprocess_line(line) + '##'
        processed_data.append(processed_line)

    return processed_data

def calculate_perplexity(processed_data, distribution):
    # could have some problems with the formula, need a further check
    cross_entropy = 0
    count = 0
    for line in processed_data:
        for i in range(len(line) - 3):
            trigram = line[i: i + 3]
            p = distribution[trigram]
            minus_log_p = -1 * log(p, 2)
            cross_entropy+= minus_log_p
            count += 1
    cross_entropy_mean = cross_entropy / count
    perplexity = pow(2, cross_entropy_mean)

    return perplexity

def search_alpha(training_data_path, test_data_path, alpha_list):

    raw_test_lines = read_test_file_by_lines(test_data_path)
    processed_test_lines = preprocess_test_data(raw_test_lines)

    result = {}
    for alpha in alpha_list:
        lm = Character_Model(training_data_path = training_data_path,
                             load_distribution_locally=False,
                             alpha=alpha,
                             write_distribution_status=True,
                             distribution_write_path=f"{alpha}.json")
        perplexity = calculate_perplexity(processed_test_lines,
         ↪  lm.n_gram_distribution)
        result[str(alpha)] = perplexity

    return result

```

```python
# seems not valuable to build on command line based program; therefore, we choose to use the most straightforward way to run the file.

lm = Character_Model(training_data_path = "./code1st/training.es",
                     distribution_write_path = './0.8.json',
                     write_distribution_status = False,
                     smoothing_status = True,
                     load_distribution_locally = False,
                     distribution_file_path='./0.8.json',
                     symbol = '#',
                     alpha = 0.8,
                     n_gram = 3)

# model excerpt: distribution of ng
ng_distribution = lm.get_n_gram_probability('ng')
print(ng_distribution)

# txt generated  from self-built language model
generated_txt_1 = generate_from_LM(length = 300, distribution = lm.n_gram_distribution)
print(generated_txt_1)
# txt generated from
pre_trained_lm = Pre_trained_model(model_path='./code1st/model-br.en')
generated_txt_2 = generate_from_LM(length = 300, distribution=pre_trained_lm.model_distribution)
print(generated_txt_2)

# calculate the perplexity of our model
test_lines = read_test_file_by_lines('./code1st/test')
processed_lines = preprocess_test_data(test_lines)
print(calculate_perplexity(processed_lines, lm.n_gram_distribution))

# # calculate the perplexity of given model
print(calculate_perplexity(processed_lines, pre_trained_lm.model_distribution))

# searching the alpha
result = search_alpha('./code1st/training.en', './code1st/test', [10, 100, 1000, 10000, 100000, 1000000])
print(result)
```

Appendix: Experimental Results

This section shows the experimental results on different values of α and the corresponding perplexities.

α	Perplexity
0.2	8.97
0.4	8.87
0.6	8.86
0.8	8.86
1	8.87
1.2	8.89
1.4	8.91
1.6	8.93
1.8	8.96
2	8.99
10	10.14
10^2	16.17
400	22.02
10^3	25.37
10^4	29.29
10^5	29.91
10^6	29.98
10^7	29.98