Programming for Computational Linguistics

N-gram Language Model

Pavel Smirnov

Introduction

The tasks for this project are to design and implement an n-gram language model in python. This language model should be able to learn from provided training text, and then generate new text that is statistically similar to the training text.

Implementation

The program is split into three main separate modules: **corpus.py** (responsible for processing and tokenizing of the text), **lm.py** (implement the core logic of the language model, responsible for predicting probability and generating text) and **main.py** (responsible for user interaction). In addition, the program contain additional modules such as **text_preprocessing.py** (execute the text pre-processing tasks) and **utests_ngramm.py** (contains unit tests of the whole program).

Let us review each module in details:

Courpus.py contains two methods **tokenize** and **detokenize**. This module also proceed some pre-processing tasks, which will be described further. **tokenize** accept as input a string of text and return a list of tokens in list format. In opposite **detokenize** method accept a list of tokens and returns a single string consisting of all those tokens together.

Text_preprocessing.py implements tasks relative with pre-processing of the input text such as deleting extra whitespaces, numbers, special chars, stop words, convert to lowercase and proceed lemmatization process. Return the list of preprocessed tokens, which provide better result for evaluation.

Lm.py provide the definition of class **LanguageModel** that contains all methods for realization of the core logic our language model. Constructor accepts as a parameter n that specify the number of elements in sequence. Method **get_ngrams** accept list of tokens and parameter n as input parameters, after execution returns token sequence represented as a tuple of tokens. In addition this method append PAD symbols represented by None element for better evaluation. The class contain instance variables: **counts** represented by dict, which contain dicts and **vocabulary**, a set. These variables initialized after executing **train** method. This method accept token sequences, after that **vocabulary** fills by the words which occur in input sequences and **counts** dict contains the statistic of how many times some words follow by their respective (n - 1) – gram in the text.

Method **p_next** allows making predictions. It takes as input a sequence of tokens and output is a probability distribution for the next token in the text. Inside **p_next** is called method **normalize**. This method accept a dict, where the keys are words and the values are numbers, representing how many times that word occurred. After executing normalize returns a new dict with words as keys and probabilities as values. The probability for each word is proportional to its count and all probabilities add up to 1. **p_next** extracts the final n – 1 tokens from input sequence taking PADs into account and for this dict of tokens as keys take numbers of the following words also represented as a dict from **counts**. This dict follows as an input parameter in **normalize** method. In case the final n – 1 tokens do not occur in **counts**, than find the first key which ends on the same word as key n – 1 token.

As we have the estimated probabilities for what word comes next, we can generate a text based on this statistic. **generate** method implements this functionality. The method starts with an empty list of tokens and use **p_next** method to get the probabilities for the next word. To pick a word according to those probabilities helps **sample** method. It accepts probability distribution as an input and returns a key from the input dict, chosen according to its probability with using **random** functionality. **generate** method works and appends words to the list of tokens until it will not predict None as the following word. In this case, this function ends and return this list of tokens as generated text.

Main.py provide the user instructions, and ask what they would like to do. All functionality expose through this module. **choose_option** imitate switch operator and run the following functionality depends on the users choice. **find_file** helps find entered file in file system. **create_language_model** create language model instance with entered n-parameter and starts to train it based on the text from entered file. **generate_text** and **generate_text_and_save** generate text. First one generates text and presents it on the screen and second one generates entered number of texts and saves them in the file **new_shakespeare.txt**.

Utests_ngramm.py covers the main functionality of the program with unit tests.

Documentation

For starting the program necessary run **main.py**. After launching of the program in console presents the instruction for the user. User should enter the number of the row for desired operation. The program checks user input. You can only enter the line number of provided operations. In case of invalid input, you will get an info about incorrect input.

For starting generating the text, it is necessary to create an instance of language model. For this, you should enter number **1**. Further, you need to enter the n-param. It's possible enter the number only in the range from 2 to 10. After that, you will be asked to provide the file-path to the file for training the language model. You can enter the full file-path for your desired file or just click “Enter” key (or enter text “None” / “none”) to read **train_shakespeare.txt** file which located in the same directory with **main.py**. In case you try to generate text before creating language model, then creation of the language model will start automatically. Wait a few seconds until the language model will be trained, you will see status in the console. Every time you will be displayed the status of the program in the console.

After creating language model and training of it, you can generate text. For generating text and present it in the console you need to enter number **2**. You will be asked to enter desired beginning of the generating text. Further generating will be based on the n – 1 tokens of entered text. In addition, you can skip this step by just click “Enter” key (or enter text “None” / “none”). In this case, the text will be generated randomly. After that, you can see generated text in the console.

Also, generation of the text is possible with saving these texts in the **new_shakespeare.txt** file, row number **3**. File will be located in the same directory with **main.py**. After choosing this option, you will be asked to enter the number of texts, which you desire to save in the file. It is possible to enter no more than 1000 texts.

You can execute these options until the moment when you enter row number **5**. In this case, the program will be closed. After each operation, the list with instructions will appear again. To find generated text just scroll up in the console, because the instructions can overlap the results of last operation.

In addition, operation on the row number **4** let you create language model with smoothing. The instructions are the same as in the case of the usual creation of a language model (row number **1**).

To execute unit tests open the **utests_ngramm.py** file and run the module or enter **python -m unittest -v utests_ngramm.py** in console. Note the current directory when enter this command.

Extensions

1. Better text processing

Text preprocessing is traditionally an important step for natural language processing tasks. It transforms text into a more digestible form so that algorithms can perform better. So how do we go about doing text preprocessing? Generally, there are 3 main components:

- Tokenization;
- Normalization;
- Noise removal.

Tokenization is about splitting strings of text into smaller pieces. **Normalization** aims to put all text on a level playing field, e.g., converting all characters to lowercase. **Noise removal** cleans up the text, e.g., remove extra whitespaces.

Based on the general outline above, there were performed a series of steps in files **text_preprocessing.py** and **corpus.py**:

- Removed extra whitespaces;
- Removed numbers;
- Removed stop words. Such words as “a”, “for”, “at” and etc. which do not help at all in NLP tasks; It also helps to save computing time and number of efforts in processing large volume of text;
- Punctuation marks and special characters were removed;
- Converted to lowercase;
- Lemmatization. Process of converting a word to its base form, e.g., “caring” to “care”.

All of the above operations were implemented to increase the accuracy of the identification task.

2. Kneser-Ney smoothing

Following smoothing is implemented in nltk library. The whole point of smoothing, to reallocate some probability mass from the n-grams appearing in the corpus to those that don't so that you don't end up with a bunch of 0 probability n-grams.

If in usual calculating of probabilities the sum is up to 1, then in the case of calculation of probabilities with smoothing it is less than 1. Rest probability reserves for n-grams, which do not appear in the provided corpus (In our case it is None).

For example, the reason why random sum 0.72 is less than 1 is that the probability is calculated only on trigrams appearing in the corpus where the first word for example is "I" and the second word is "love" The remaining 0.28 probability is reserved for context which do not follow "I" and "love" in the corpus.

You can access this functionality if enter row number **4** in the list of instructions. After that will be created and trained a language model with smoothing. As a result, the probability to predict None as a next word is higher, since None in our case is reserved for n-grams which are not appear in the corpus. Therefore, the number of word in major part of cases in a result token sequence is lower.