Core

Jump to bottom

alexander871015 edited this page a day ago · 28 revisions

Core Description

The Core is where you'll build your learning models. We will work with Beatles song lyrics and Gamecube midi music. Your models will receive lyric or data from a specific creator as input and generate new lyrics or songs in that same style. We'll be taking advantage of an NLP concept called n-grams and an NLP technique called language modeling.

Much of the Core will be easier if you have completed the Warmup. You are expected to complete the Warmup for yourself **even if your teammates already have.** You will also need to be familiar with classes and inheritance in Python. For the music generation part of the core you will need to understand how PySynth works.

The Core does not require you to include any external libraries beyond what has already been included for you. Use of any other external libraries is prohibited.

Before You Code!

Before you start coding, make sure you fully read the Python 3 tutorial.

Table of Contents

Core Structure

Everything relating to the Language Models are contained in the `models/` folder. You are responsible for implementing the following functions in the following files:

```
languageModel.py
  * selectNGramModel()
  * weightedChoice()
  * getNextToken()

unigramModel.py
  * trainModel()
  * trainingDataHasNGram()
  * getCandidateDictionary()

bigramModel.py
  * trainModel()
  * trainingDataHasNGram()
  * getCandidateDictionary()

trigramModel.py
  * trainModel()
  * trainingDataHasNGram()
  * getCandidateDictionary()
```

The Creative AI Learning Models are held as members in the LanguageModel. A call to languageModel.getNextToken will use the underlying member models to produce to the best output.

The driver of the project is `generate.py` in the root project folder. You are responsible for the following:

```
generate.py
  * generateTokenSentence()
```

`main()` has been implemented for you and only needs to be partially uncommented when you have finished the rest of the project.

We recommend that one half of the team implement `LanguageModel.py` and `generate.py` while the other half implement the guts of the learning models. We also recommend that you implement the functions in the order they are described in the spec.

Functions to Implement

`trainModel`

This function trains the NGramModel classes on the input data by building a dictionary of n-grams and respective counts, `self.nGramCounts`. Finally, the function returns the self.nGramCounts dictionary.

- The unigram model dictionary, `self.nGramCounts`, will be a one-dimensional dictionary of {*unigram*: *unigramCount*} pairs, where each unique unigram is somewhere in the input data, and *unigramCount* is the number of times the model saw that particular unigram appear in the data.
- The bigram model dictionary will be a two-dimensional dictionary of {*unigramOne*: {*unigramTwo*: *bigramCount*}} entries. *bigramCount* is the count of how many times this model has seen *unigramOne* + *unigramTwo* appear as a bigram in the input data. For example, if the only song you were looking at was *Strawberry Fields Forever*, part of the BigramModel's `self.nGramCounts` dictionary would look like this:

```
self.nGramCounts = {
    'strawberry' : {'fields' : 10},
    'fields' : {'forever' : 6, '$:::$' : 4}
}
```

- The trigram model dictionary will be a three-dimensional dictionary of {*unigramOne*: {*unigramTwo*: {*unigramThree*: *trigramCount*}}}, entries. *trigramCount* is the count of how many times this model has seen *unigramOne* + *unigramTwo* + *unigramThree* appear as a trigram in the input data.

Special tokens

The text this function receives will contain three special tokens, two at the beginning, `^::^` and `^:::^`, and one at the end, `$:::$`. These tokens are how we remember which words start and end sentences. For example, the sentence `"^::^ ^:::^ the quick brown fox $:::$"` contains the trigram `"^::^ ^:::^ the"`. Our trigram model will count this as one trigram. When we start generating sentences we'll begin with the sentence `"^::^ ^:::^"`, which means that it's more likely that our model will first choose the word "the". This will be important when we later go to generate sentences.

The ending token works similarly -- when our sentence contains the word "fox" it is much likelier that we'll generate the ending `$:::$` token, which means that our generated sentences will end in similar ways to the sentences we trained on.

We must consider these tokens separately in each model:

- The unigram model should not consider the special symbols `'^::^'` and `'^:::^'` as words to count, but it should consider the ending symbol `'$:::$'`.
- The bigram and trigram models should consider all three special symbols as words.

Each trainModel function should return the dictionary that you trained in the body of the function. Be aware that while your code will work without returning, it will be impossible to grade your submission without returning the dictionary.

`trainingDataHasNGram`

This function takes a sentence, in the form of a list of strings, and returns True if a particular language model can be used to determine the next token to add to that sentence. If not, it returns False.

- For the unigram model, this function returns True if the unigram model knows about any words at all: in other words, its `self.nGramCounts` dictionary is not empty. This is because a unigram model only looks at the frequency of single words, regardless of their context.

- For the bigram model, this function returns True if the model has seen the last word in the current sentence at the start of a bigram. *Hint: which "dimension" of the bigram model's `self.nGramCounts` would contain this information?*

- For the trigram model, this function returns True if the model has seen the second-to-last and last words in the current sentence at the start of a trigram, *in that order*.

`getCandidateDictionary`

This function returns a dictionary of candidate next words that can be added to the current sentence. It considers a sentence, examines the last words in the sentence, and returns that set of words and matching counts.

- For the unigram model, this function does not consider the input sentence at all, because we are only generating unigrams. It should return the dictionary of tokens and counts, `self.nGramCounts`.

- For the bigram model, this function will consider the last word in the input sentence. It will then use that word to select the tokens which may follow that word, based on the model's bigram data.

- For the trigram model, this function will consider the last two words in the input sentence. It will then use those words to select the tokens which may follow, based on model's trigram data.

Hint: the indexing method you use here will be syntactically very similar to what you did in `trainingDataHasNGram`.

`weightedChoice`

This function takes a dictionary as input and chooses a key in that dictionary to return, using the dictionary's values to compute probabilities of each key being chosen. It will be used in `getNextToken` as a helper function in choosing a next word for a randomly generated sentence.

Suppose your input dictionary was this:

```
{'north': 4, 'south': 1, 'east': 3, 'west': 2}
```

Here's how to choose a key to return from the above dictionary. First, make two lists: one to represent all the keys in the dictionary, and one to represent all the values in the dictionary. Here's a table where the "token" column is our list of keys, and the "count" column is our list of values.

index	token	count
0	north	4
1	south	1
2	east	3
3	west	2

We can now create a third list which is a *cumulative* count list, shown in the fourth column:

index	token	count	cumulative
0	north	4	4
1	south	1	5
2	east	3	8
3	west	2	10

Now that we have a cumulative count list that corresponds directly to our list of tokens, we generate a random integer in the interval [0, 10) (where 10 is the maximum value in the cumulative list) using Python's `random.randrange(min, max)` function. This will return an integer *x* such that min <= *x* < max. We then walk through the token list and return the first token whose corresponding cumulative count is larger than our random number.

For example, using the table above, say we generated 7 as our random number. Then we see the word at index 2, "east", is the first word with a cumulative count greater than 7, since the word's cumulative value is 8. Therefore, we return the token at index 2.

`getNextToken`

This function does three main things:

1. Call `getCandidateDictionary` with the current sentence as an argument.
2. If filter is `None`, pass the return value of `getCandidateDictionary` to `weightedChoice`.
3. If filter is `None`, return whatever `weightedChoice` returns.

At a high level, the effect of doing this is getting a list of candidate next words for the current sentence, choosing a next word for the sentence based on the weights of each candidate word, and then finally returning that chosen word.

If filter is not `None`, instead of passing the result of `getCandidateDictionary` to `WeightedChoice`, we will build a new `filteredCandidates` dictionary first. We should build this `filteredCandidates` dictionary by iterating through the return result of `getCandidateDictionary`, and if a key in that result is also present in our filtered list (or is some special token), we will add to our `filteredCandidates` the same key, and value associated with that key. If at the end our `filteredCandidates` is still empty, we should return a random item in `filter`. If `filteredCandidates` is not empty, we want to return the result from passing `filteredCandidates` to `WeightedChoice` instead.

`selectNGramModel`

This function takes a sentence, which is a list of strings. It first checks if the trigram model in `self.models` can be used to pick the next word for the sentence; if so, it returns the trigram model. If not, it attempts to back off to a bigram model and returns the bigram model if possible. As a last resort, it returns the unigram model. *Remember that you wrote a function that checks whether or not a particular NGramModel can be used to pick the next word for a sentence.*

`generateTokenSentence`

This function adds a word one at a time to a sentence until it decides that the sentence is done. *Remember that you wrote a function that picks a next word for a sentence using a language model!*

Determining whether a sentence is done can come about one of two ways: either the `sentenceTooLong` function, which is written for you, returns True because the sentence is too long, or the next token chosen for the sentence is the special ending symbol `$:::$`. If either of these two events happen, the sentence is done. **When considering the current length of the sentence, you should not count any of the special symbols `^::^`, `^:::^`, or `$:::$`.**

This function returns a list of tokens representing a sentence (musical or not). **The returned list should not contain any of the starting or ending symbols!**

Provided Functions

`runLyricsGenerator`

This function takes a list of trained language models and calls the functions you wrote above to generate a verse one, a verse two, and a chorus. Each verse/chorus is a list containing 4 sentences, where sentences are lists of strings. Note: when you call `generateTokenSentence` in this function, you can choose what value you would like `desiredLength` (the goal length of the sentence) to be - play around with different values and see what gets the best output.

After you create the `verseOne`, `verseTwo`, and `chorus` lists, pass those lists into the `printSongLyrics` function, which is written for you. This will print out the song that you created in a nice song-like format.

runMusicGenerator

This function works exactly the same as `runLyricsGenerator` with the exception that it also takes in a song name.

It calls `pysynth.make_wav(tuplesList, fn=songName)` after your list of PySynth tuples is generated in order to actually make the .wav file. The `tuplesList` parameter will be the list of PySynth tuples that comprise your song. The `fn=songName` part tells PySynth what to name the .wav output file, which will be stored in the `wav/` directory when PySynth is finished making your song.

sentenceTooLong

`sentenceTooLong()` uses gaussian probabilities to randomly decide when a sentence is long enough. It would be unnatural if all sentences were the same length, so `sentenceTooLong` checks if your sentence reaches a desired length, plus or minus a small amount.

This function takes two parameters, an integer `desiredLength` and an integer `currentLength`. `desiredLength` is how the length that you would like your sentence to be. The `random.gauss` function generates a random number close to `currentLength`; if the random number is larger than `desiredLength`, the function returns `True`. Otherwise the function returns `False`.

Increasing the value of `STDEV` in this function will increase the randomness, leading to more varying sentence lengths.

printSongLyrics

This function takes three parameters which are lists of lists of strings: `verseOne`, `verseTwo`, and `chorus`. It then prints out the song in this order: verse one, chorus, verse two, chorus.

trainLyricsModels

This function takes a list of directories from your `data/lyrics` subfolder where artists' lyrics are stored - for example, `[the_beatles]` - and returns those lyrics as a list of lists. It then makes a list of language models and trains those language models using the returned lyrics.

This function ultimately returns the list of trained language models.

`trainMusicModels`

This functions works exactly the same as `trainMusicModels` except that it uses information found in `data/midi` instead.

`main`

In normal mode, main prompts the user for input, 1 to generate Music, 2 for Song Data, 3 to quit. These capabilities have been implemented for you, but may not function properly until `generateTokenSentence` is finished.

How to Run

If you are using PyCharm, open `generate.py` and click "Run..." in the top navigation bar.

If you are working from the command line, navigate to the root directory where your CreativeAI project is stored and type:

```
python generate.py
```

Even if you have not implemented any of the functions in the project, the starter code should work out of the box. Therefore, you can play around with it and get a feel for how the driver in `main` works.

▶ Pages 11

Project Overview

1. Proposal
2. Warmup
3. Core
4. Reach
5. Grading and Key Dates

Appendix A. Testing

Appendix B. Concepts

Appendix C. Advanced Concepts

Clone this wiki locally

```
https://github.com/eecs183/creative-ai.wiki.git
```