Detecting Word Issues in Course Transcripts: Progress Report

Team: The Palindromers
Theme: Free Topics
Date: Dec 11, 2021

Video Documentation:

You can view the video documenting what our program does here.

Overview of Program:

The objective of this program is to read the course's video transcripts to detect and flag potential misspellings and incorrect words, based on the context they're used in. The program has two major inputs: the course transcripts and the course textbook. These two sources are used to train a unigram language model and a bigram language model. Transcripts are read, preprocessed and placed into a list. The program will loop through the list and find the probability of the word based on the unigram model. At the same time, it will grab the words before and after the current word in question and compare these pairings with the bigram model to find the probability of this word in the context of the words surrounding it. These probabilities are combined, based on a user-defined weighting, and compared against a "typo identification threshold" that will signify if a word is potentially incorrect. The potential incorrect words are then output in a text file for the user to review for correction. The output shows which word was flagged as a potential typo, the word's index in the sentence, the word's probability score, the text data containing the word, the transcript the word came from, and the timestamp in the transcript where the word is located.

Background:

Before we executed our current implementation, we explored using a pre-trained neural language model to use for our language models. Specifically, we looked at OpenAI's GPT-3, BERT, ALBERT and CodeBERT. We chose to pursue ALBERT as a potential model for our code (GPT-3 had a waitlist that was past our due date, CodeBERT seemed geared more towards understanding code language and BERT tended to be very resource intensive, so we ruled them out). ALBERT, short for "A Lite BERT", is as it's name would suggest - a lightweight version of BERT ALBERT is a bi-directional NLP model, so it takes into account the context surrounding both sides of a word (this is different from models like GPT-3 or ELMo). As we explored this model, we found it hard to see how to incorporate it into our own program using the available resources. However, we did find there were some potential uses to our objective for correcting identified incorrect words (discussed in the 'Further Enhancements' section).

In creating our unigram and bigram models, we debated what corpus we should use to build the models. Two main sources arose: wikipedia and the course textbook. For wikipedia, we would take a sampling of various articles and use this to create our bigram model and background unigram model. Conversely, we thought of using our course textbook to do the same (using a pdf file and converting/processing through the NLTK library). We ended up choosing the course textbook for this, as we believed the words and context used in the textbook would be quite similar to the video transcripts. This would help provide a model where words were used appropriately and thus, able to better identify incorrect words in the transcript.

Code Documentation:

'textbook_processor':
The original textbook is in PDF format. We extracted the content of the textbook *Text Data Management and Analysis* (Zhai & Massung, 2016) by first using the Convertio online tool to convert it from PDF file format to text file format. Next we used a text file editor to clear out the extra space, tab, and empty line in order to make the file ready for the NLTK process.

The textbook_processor script contains the code that reads the cleaned up text format textbook file and processes the text with the sent_tokenize() function in the NLTK library to split the text so that each sentence is on a new line. To fix the UnicodeDecodeError error, we used the utf-8 character encoding when reading and writing the textbook file.

'final_unigram_bigram_script'
This code contains the key functions and classes to read the processed textbook and transcripts as well as train the required unigram and bigram models. We will discuss the main functions here -

The 'read_transcript' function: The raw transcript text data contains line numbers, timestamps, and the actual spoken words. We only need the actual spoken words to train the n-gram models. Additionally, for the bigram model we need the actual spoken words to be split by complete sentences but the raw transcript text data is broken down by timestamps. This function cleans up everything from the raw transcripts and returns only the clean spoken word text broken down by sentences. It also populates a dictionary with the transcript's file name, the timestamp, and the text associated with that timestamp to be utilized later on.

The ‘read_textbook’ function: Reads the processed textbook returned from the ‘Textbook_processing’ script and stores it in a list.

The ‘unigram_text_formatter’ function: Converts raw text into lowercase, removes punctuation and numbers, and cleans up any extra spaces in the processed text.

The ‘UnigramLanguagemodel’ class: This class iterates through the words in the transcript to store the word frequencies, total words contained in the transcript data, and total unique words. It contains the function ‘calculate_unigram_probability’ that calculates the unigram probability for a word based on its frequency.

The ‘BigramLanguagemodel’ class: This class iterates through each sentence in the transcript data and then each word in a sentence to store the frequency of two words occurring together and all of the unique word pairings. It contains the function ‘calculate_bigram_probability’ that calculates the bigram probabilities for a word pair and another function ‘sorted_vocabulary’ to store an alphabetically sorted list of all the word pairs and their frequencies.

The ‘store_unigram_probs’ and ‘store_bigram_probs’ functions: These functions calculated the required probability for the entire list of unigrams or bigrams respectively.

The ‘unigram_mixture_probs’ and ‘bigram_mixture_probs’ functions: These functions are used to take a weighted mixture of the calculated probabilities from the textbook data and the transcript data. The parameter used to control the contribution of textbook and transcript is ‘lam’. The higher the value of ‘lam’, the higher is the contribution of probabilities from textbook data.

The ‘get_timestamp’ and ‘get_timestamp_for_inaudible’ functions: The first function is utilized after the models have identified potential typo words to extract the specific transcript name and timestamp for where the potential typo is located in order to make it easier for a human user to review the specific word. The second function finds all cases where a transcript has the word ‘inaudible’ (meaning the original transcription was unable to make any guess for a correct word) and outputs a list of those cases for human review.

This script also includes some helper functions that are used to print lists, dictionaries, and tuples to text files. Some of these functions are used to store the language models while others are mainly useful for inspecting and spot-checking results at intermediate stages.

‘Typo_finder’
This code contains the key functions and class to load the transcripts to be evaluated and determines how a typo will be detected in the corpus of transcripts.

The ‘load_prob_dict’ function loads n-gram models from the local text file into a dictionary. This dictionary can then be used to pull word probabilities.

The ‘load_corpus’ function is used to process the transcript strings (make lower case, strip out punctuation, etc.) and load into a list of word values. This format will match the unigram and bigram word keys for easy dictionary lookup.

The ‘TypoFinder’ class: This class takes the unigram and bigram models, unigram and bigram weights, and a user-defined threshold as inputs when instantiated. This is the data it will use to identify incorrect words. It has two functions.

The ‘typo_flag’ function is passed the threshold and weights, as well as the word, and two word pairings (word + word before, and word + word after). It checks the bigram model for both pairings. If they’re found, they are averaged together to get an average bigram probability. If not, the probability defaults to zero. If the word is found in the unigram model, it will be assigned a probability as well (again, zero if nothing found). The function uses the weight passed in as arguments, applies them to each probability score respectively and sums them up. This combined score is returned as an output.

The ‘typo_finder’ function will loop through the corpus list, split by line, providing the words and paired words to the ‘typo_flag’ function. It compares the score of the ‘typo_flag’ output to the threshold and, if below the threshold, adds it to a list as a potential incorrect word. This list is returned from the function.

‘Main_program”
This is the entry point to the program.

User-defined variables are set up at the beginning. These can (and should) be updated to reflect where the directories are located on the local machine. Users can also modify the weights given to each n-gram model and the threshold used to identify incorrect words.

Based on the input data and parameter setup, the program will create the unigram, bigram and consolidated transcript files. First, the n-gram source data (transcripts and textbook) are read and given as arguments to the UnigramLanguageModel and BigramLanugaeModel classes. Functions from these classes are called to sort data by

vocab keys, calculate the probabilities of each model, and write them to their respective local file. Transcripts are also consolidated and written to a local file.

Once the language models are set up, the TypoFinder will load the corpus into a list. The unigram model, bigram model, unigram weight, bigram weight and threshold will all be passed into a TypoFinder object as arguments. This object then calls the 'typo_finder' function to identify typos (based on the arguments passed) and outputs them to a list.

The typo list results are then looped through and output to the results text file in a more user-friendly format. After that loop is finished, all of the cases where a transcript has the word 'inaudible' (indicating no guess was made for the spoken words) are added to the end of the results file so that they can be reviewed and updated as well.

Evaluation

The model predicts the typos for a total of 100 course transcripts containing 160k words and predicts 1500+ typos.
The team went through each of the 1500+ typos marked by the model and found 478 to be correct typos. This results in a model precision of 30%.
To calculate the recall, 20 sample transcripts were selected and the team marked all the real typos by manually reading and checking them. There were 388 typos out of which the model had detected 130 typos. So the recall of the model is 33%.
The evaluation of the predicted typos and the ground truth sample transcripts is available on this Google sheet.

This implementation can be further improved by varying some of the parameters or other methods as discussed in the further enhancements section.

Instructions on How to Run Locally:

In order to run this program on your local machine:

1. Install all needed packages:
    a. RE (regular expression - should come with python)
    b. NLTK
2. Download the code from the github repository: https://github.com/dani-richmond/CourseProject
3. Extract Transcripts.zip file (where all of the transcripts reside)
4. In the 'main_program.py' code (in Final Scripts), update the 'transcript_dir' and 'textbook_dir' variable values to where these two directories are located on your local machine (both should be in the 'CourseProject' folder)
5. Edit the 'unigram_weight', 'bigram_weight' and 'threshold' as desired

6. Run the main program
7. The potential typo results can be reviewed in the results.txt file that is output

Further Enhancements:

This program could be further refined by expanding the bigram model. The unigram model really shows us what words tend to be rarely used, which is helpful, but doesn't define an incorrect word on its own. The bigram model is what really gives the word correct context. By expanding the bigram model, we expand the context of word usage and can better refine and identify word pairing probabilities. With this enhancement, probabilities would be more unique per word (and their associated word pairings), which would give us more flexibility on defining thresholds to exclude specific words.

An additional enhancement to this program would be to leverage a pre-trained neural language model (such as ALBERT) to help suggest corrections to identified incorrect words. HuggingFace API has the capability of masking a word in a sentence and having the underlying neural language model suggest a best-fit word. Our program would go through the transcript to identify an incorrect word and flag it. Once the word is identified, the sentence string and position of the incorrect word can be passed to HuggingFace and mask the word in the given position. Then, the pre-trained model can be used to guess what the correct word should be.

Another enhancement that could be considered is to automate the PDF to the text conversion process and do the necessary trimming by code before the process with the NLTK library. In this way, the program can be more user-friendly for users who want to import the PDF files to the unigram model.

If the bigram model is expanded (such as by using both a course's textbook and a much larger corpus like Wikipedia articles), this program could be utilized by any professor that uses the Coursera platform (and so has transcripts that are formatted in the manner expected by our script). This in combination with leveraging a pre-trained neural language model to provide guesses for the correct word would be a very useful application for many professors.

Team Member Contribution:

Scott Downey: Pre-trained neural language model research, typo_finder coding, main_program coding, model and probability threshold/weight testing, documentation, model evaluation (model results only, no ground truth evaluation)

Dani Richmond: team leader (coordinated tasks, checked for status updates, facilitated team meetings, etc), NLTK library research, unigram_bigram coding, main_program coding, documentation, model evaluation (model results and ground truth evaluation)

Catherine Parker: Implemented get_timestamp function, model evaluation (model results and ground truth evaluation)

Zixiang (Steven) Li: Pre-trained neural language model research, pre-process textbook from PDF to line separated text file with NLTK, main_program coding, textbookProcessor coding, get_timestamp_for_inaudible coding, documentation, model evaluation (model results and ground truth evaluation), sample test data preparation

Jharna Aggarwal: unigram_bigram coding, main_program coding, model evaluation (model results and ground truth evaluation), project documentation, video presentation