Azerbaijani Unicodification

Mark Gerken

Dr. Kevin Scannell

Machine Learning – CSCI 4930

Project Description

This project aimed to predict and restore the diacritics and extended Latin characters to provided Azerbaijani text.

Data Exploration

The project data was provided in two separate files, “azj-train.txt” and “input.csv”. The text document provided a large corpus of Azerbaijani training words with correct diacritics and characters. The CSV file was used for testing and contains 272,596 lines of text that had to be parsed and evaluated for unicodification.

Evaluation Process

Evaluation was performed online through the website Kaggle. All output was placed into the file “predictions.csv”. This file is then uploaded to Kaggle, which then tests and scores it. The evaluation metric for this project is token-level accuracy, which is simply the percentage of tokens in the input file correctly restored.

Process and Results

Below I outline every attempt made throughout the Kaggle competition. Each attempt describes the current state of the project as well as the reported accuracy.

Attempt #1: Dictionary approach

This first implementation simply memorized each word in the training data. If the word is found in the test data, the word is replaced with the example seen in training.

Accuracy: 60.232%

Attempt #2/3: Single letter Unigram

Records each instance of a unique character. Every character is replaced individually with the most recently seen equivalent, diacritically decorated or not. Attempt #3 was the fix of a logic error, which caused every single character to be replaced with a decorated character.

Accuracy: ~~12.493%~~ 53.447%

Attempt #4: Dictionary and frequency List

This attempt combines previous attempts in a filter-like style approach. Code has been restructured for a future attempt of multi-line contextual analysis (not yet implemented as of 15 April 2018). Conditionals test if the word has been seen before in training, otherwise defaults to evaluating each letter against the frequency list described in attempt #2.

Accuracy: 84.613%

Attempt #5/6: 5/4/3 letter sequences, dictionary, and frequency list

Filter-like approach expanded upon. If the word is not found in the dictionary, the word is analyzed for a before seen 5/4/3 letter pattern. The word will undergo any replacements necessary, and then be passed through the frequency list comparison. Capitalization is still preserved. Attempt #5: Program currently reports never finding a letter pattern it hasn't seen in training, so suspecting bug. Attempt #6: Bug fixed. Program is currently very inefficient. Processing requires several minutes (recorded at about 2:20) to complete. Intend to look into storing processed lists externally instead of fully recreating each time.

Accuracy: ~~83.943%~~ 85.174%

Attempt #7: Two and Three Line Context

Completed dictionary to file processing, drastically reducing run time of program. All text processing is now saved to CSV files and read back. If CSV files are not present, program will simply redo the processing. I attempted to incorporate multi-line comparisons for context. Unfortunately the processing time became impossibly large, and I could not devise way to reduce efficiently. All necessary code was migrated to a single file, documented appropriately, and removed from program functionality. The goal was to consider up to previous two lines in word replacements, but was ultimately unsuccessful.

Accuracy: No submission

Attempt #8: Minor bug fixes and short string resolution

Program would skip short strings (i.e. less than 3 characters), as well as cut out lines consisting of just a comma. Additional logic and bug fixes implemented, as well as program cleaned up in general.

Accuracy: 90.558%

Attempt #9/10: Fix of default resolution

Program would return empty strings on ASCII characters not encountered before, now resolved. Changed order of restoration to Dictionary -> 5 Letter -> 4 Letter -> 3 Letter -> Frequency List -> Default. Program currently needs functionality for allowing multiple multi-letter replacements in a single line. As is, program will make the first replacement it finds in a word and then move on to the next without any further searching for letter sequences. Output has inconsistencies where any line containing just a comma is surrounded by quotes, while all other lines are not. Attempt #10 involved surrounding every line with quotes to make the file consistent with the provided testing data, and to ensure accurate scoring by Kaggle. The change had no effect.

Accuracy: ~~88.577%~~ 88.577%

Attempt #11: Support for multiple multi-letter pattern replacements

Program will now continue to look for additional letter patterns in words, and will not just move to the next word after the first pattern is found. Program first scans through word searching first for all five letter combos, then four, then three.

Accuracy: 92.202%

Attempt #12: Keras implementation

I made a final attempt to use Keras to implement LSTM solution as demonstrated in class on 4 May 2018. Unfortunately, due to the drastically different processing and the complexity of the models, I was unable to figure out how to adapt the Keras LSTM to this problem and ultimately ran out of time.

Accuracy: No submission.

Conclusion

I had a final highest score of 92.202% using the multi-tiered approach combining a dictionary, frequency lists, and letter sequence replacements. I know I have some error dealing with

ambiguous decisions by the decoder. For example, the letter ə was replaced with '@' instead of 'e'. I had to manually change this every time I decoded text. There are likely other differences between the decoder in use and the decoder that removed the diacritics from the input text initially.

I began this approach with a very naïve take and tackled the problem like I would any other. As the course progressed, I learned more about different tactics and approaches I could have taken but it became too late in the semester and would have required essentially rebuilding the project from scratch. If I had more time, I would have liked to do more research and find out ways to vectorize the input data and implement actual classifiers from scikit-learn or other modules.

Similarly, a successful implementation of LSTM would almost have assuredly raised the accuracy drastically. There is so much complexity in the implementation that I was unable to figure out how to adapt it to this problem in the short time I had left. More specifically, I could not figure out how to vectorize the testing data to the expected format. I hope to finish the implementation on my own once the semester ends so I can take more time to understand the model and how to use it.

Sources

https://github.com/keras-team/keras/blob/master/examples/lstm_seq2seq.py

GitHub Repository:

https://github.com/gerkenma/Azerbaijani-Unicodification