COMP30027 Project 2 – Short Text Language Identification

Ivan Ken Weng Chee
the_leaden_limeira

Abstract

Language identification is, given a document, the task of predicting the predominant language the document is written in. This report details the algorithm used in my Twitter tweet language identification system.

1 The Task

The task given was to build a classifier which would train itself on a training dataset (dev.json) and try to predict languages in a test dataset (test.json) based on insights obtained from the training dataset. The task can be subdivided into smaller sections which detail the flow to completion.

1.1 Data Preprocessing

The data was read in using the Pandas library for improved speed and flexibility. Some of the text appeared as Unicode characters as they were of different language origins, and there were a few issues printing right-to-left languages like Arabic. Upon reading in, missing data contained NaN values, which were replaced by an empty string.

1.2 Classifier Training

Classifier training consisted of the insertion of all words from a given sentence into their respective language tries. Here words are defined as a block of characters separated by whitespace, obtained via the strip() method of strings.

1.3 Classifier Testing

Test data instances are each tested and given a score based on how many words are contained within a language trie. A total score for each language is stored and sorted, with the maximum score representing the language class the text belongs to.

2 The Technical Details of The Implementation

The following section describes the technical details of the implementation.

2.1 Classification Algorithm

The classifier I used is based on the properties of a tree-like structure called a Trie. A trie, pronounced 'try', is a tree-based data structure which is extremely suitable for storing text, as it contains many repetitions of a set of characters. The root node represents the starting point of words stored, and at each step, a node will represent a single letter of the word. These nodes branch off when the ordering of letters between words diverges, or when a word ends.

2.2 Time Complexity

Contains : O(m)
Insertion : O(m)
m : Average word length
A trie typically spends m comparisons to store and retrieve a word, where m corresponds to the length of the word in characters.
This makes insertion and access faster than imperfect hashtables, as it has no need to worry about key collisions, or hashing algorithms to guarantee a unique path to elements, as tries can be alphabetically ordered by default.

2.3 Space Complexity

Storage : O(nm)
n : Possible character count
m : Average word length
A trie can be quite memory intensive, depending on n, the character count, which adds to the branching factor of the trie. m would grow very large if using the trie to store sentences instead of words.

3 Classifier Evaluation and Error Evaluation

This section details the evaluation aspects of the classifier and how it handles errors.

3.1 Classifier Evaluation

This submission scored 0.71676 on the Kaggle Short Text Language Identification competition, which is not too high for such a classifier, but well above the majority class baseline. One fact to note

is that the score would theoretically decrease significantly with smaller training data, but increase by a large margin with larger training data, as each trie language dictionary would grow to approximate their respective true language dictionaries. However, new forms of languages and slangs such as memes and words contained in the Urban Dictionary, which may have grammar different to that of their languages, would likely contaminate other language tries and make classification harder.

3.2 Error Evaluation

The classifier scores around 71% accuracy when evaluated on the development data, much like the scores on the competition test data. There is not much to be obtained out of error evaluation as this type of classifier 'learns' through increased input data. Feeding a dictionary into the trie would in theory greatly improve the results, but would be inefficient in terms of space, although there are algorithms which deal with trie compression. Errors mostly occur due to the same spelling of a word occurring in more than one language dictionary, such as "air" in English and "air" in Malay which means "water".

3.3 Limitations and Difficulty Faced

There are several limitations with this classifier. One such is in dealing with spelling errors in the text, whereby there is no obvious way of detecting such an error and preventing it from 'polluting' the trie with incorrect words. Another would be the memory requirements for storing the tries, which may grow significantly if larger datasets are used. Some difficulties faced while building the classifier include processing the large dataset, dealing with missing values, and determining a scoring function which would give best results based on the data.

4 Insights

4.1 Levenshtein Distance

Levenshtein distance, also called edit distance, is a similarity measure between two strings, denoted s and t. The distance is the total number of deletions, insertions, or substitutions required to transform s into t. A greater Levenshtein distance is an indication that s and t are more different. In my opinion, text from different languages would exhibit a greater Levenshtein distance from each other, and thus I proposed this as a metric to be used in my classifier. I tried incorporating this metric as a scoring function classifier testing, specifically the getClass() function, which for every instance in the test data, computes the distance between each word in the string with each word in the trie. However, upon running, I quickly realized this method became largely infeasible, and would likely have taken a few days to complete, as the algorithm becomes an efficiency bottleneck in the task due to the heavy recursion. The functionality for this metric is available in the python script, but remains untested.

4.2 Algorithm Tuning

Strangely, when modifying the trie to store whole sentences, as an analogy to having repeating tweets, produced a lower score than the purely word-based method. This is also the case when storing individual characters in the trie, i.e. a-zA-Z for Latin-based languages.

4.3 Classifier Extensions

One possibility for extending this classifier to obtain more representative results is by using a radix tree (suffix tree) instead of a trie. With this, words can be checked based on common subsequences instead of letters. Another possible extension is by using a deterministic acyclic finite state automation (DAFSA) to save space, especially due to the large branching factor of some languages.

5 Conclusion

Although markedly different to Machine Learning algorithms, this classification approach tries to exploit the properties of a data structure - the trie, to accomplish a specific task – which in this case, is a complex version of string processing.

References

Klein, Bernd. "Python Advanced: Recursive And Iterative Implementation Of The Edit Distance". *Python-course.eu*.

Bird, Steven, Ewan Klein, and Edward Loper. *Natural Language Processing With Python: Analyzing Text With The Natural Language Toolkit*. 1st ed. Nanjing, Jiangsu: Southeast University Press, 2010. Print.