CSC-439 Final Project: Automatic Sentence Splitting

Matthias Guenther

December 7, 2017

Introduction

A classifier was developed to determine whether or not content could be removed from sentences—and possibly placed elsewhere—to simplify their structure. Success in this endeavor would allow more accessible texts for public dissemination, enhancing the typical individual's access to many classes of information. This applies to some extent regardless of the reader's native language and level of education: even individuals adept in reading comprehension might simply choose not to read an excessively verbose document, cutting them off just as effectively as if they couldn't grasp the content.

Approach

The given data set contained parsed sentences and whether or not they were split. While there was some additional information, none of it was relevant to this project. There were, in particular, simplified versions of the sentences, but these were not of use becuase they did not correspond directly to the structures of the original sentences. If they had been exact copies with the unwanted subtrees removed, they would have been useful in training the classifier on where to split a sentence; unfortunately, this was not feasible given the differing sentence structures.

The sentences in the data set had already been parsed into syntactic trees, but some processing was still required to convert the string representations into usable data structures. This was done primarily with regular expressions. Since the model assumes the input sentences take the form of parse trees, it will require that the same parser as used on the data set be run before it can classify a new sentence.

The model was designed as a logistic-regression classifier meant to analyze the structure of parse trees. As the structure of parse trees is not constant, it was not possible to train a separate classifier for each node. Instead, the same classifier was used on each node; this was the inception for using the tree's structure as most of the features, as doing so would allow it to produce meaningful results despite having the same weights for different parts of the sentence. It became evident from printing out a few examples in tree form that sentences could generally be split if they had multiple large, distinct, high-level subtrees, as these tended to represent different topics or points. Therefore, the classifier was designed to analyze the shape of the tree to make its decision. The syntactic significance of the node connecting multiple subtrees could also have relevant information, so the tags produced by the parser were included as features along with structural information.

The classifier's features included each node's tag generated by the parser, the node's depth, its numbers of siblings, children, and descendants, and the maximum depth of any descendant. Additionally, it calculated sentence-splitting probabilities node-by-node from the bottom up, passing each node's result on to its parent. The parent node would then use all of its children's outputs to calculate the probability that nothing should be dropped from the sentence, per the following formula:

$$P_{parent}(drop) = 1 - \sum_{i=1}^{n} (1 - P_i(drop))$$

This probability was then used as another feature in the parent; bottom-level nodes used zero for this value.

The original hope was to calculate the probability at each node that the current node should be dropped, then to procede as described above. Unfortunately, given the differing structures between the data set's

original and simplified sentences, there was no feasible way to train the lower-level nodes. Instead, only the top node was trained directly, allowing the descendants to provide whatever input they may through the child-based input. While this may seem arbitrary, the results will show that it had a significant effect. Additionally, keeping child input in the model means that if a data set with the aforementioned additional data should ever become available, modifying the code to utilize the additional information should require minimal additional effort on the part of whoever might be using the classifier.

Code

The code is divided into three modules: s_split, parsetree, and binarylr. The s_split module contains functions for training and testing classifiers; the classifiers themselves are contained in binarylr and used by parsetree. Specifically, binarylr contains the BinaryLR class, representing a binary logistic-regression classifier. The parsetree module, on the other hand, contains the ParseTree class, representing the syntactic tree of a particular sentence. The former of these classes is a generic classifier designed not to make assumptions about the task at hand. It is the latter of these two classes that specializes the program for the task of predicting whether content should be removed from a sentence.

BinaryLR is more or less a standard implementation of logistic regression; as mentioned above, it is designed not to make assumptions about the problem at hand, making an extended discussion of its workings superfluous. The only methods used externally in thie project are the constructor, the classify() instance method, and the learn() instance method. The classify() method classifies a feature vector—either as a binary output or, if True is added as the optional second argument, as a probability. The learn() method simply takes the correct label for a training example and updates the classifier's weights based on its most recent output.

ParseTree, on instantiation, reconstructs the parse tree represented by the string provided to the constructor. It also takes the set of tags that can be produced by the parser, the BinaryLR object to be used for classifying each node, and two optional arguments. The first of these optional arguments is the threshold of $P_{parent}(drop)$ that will result in the top node returning a True (split) classification. This threshold defaults to 0.5, and it only applies to cases where a binary output is requested, rather than a probability ranging from 0 to 1. The second optional argument is only intended for use within the module itself: the tree is constructed recursively, with each ParseTree instance creating its children and passing them references to itself.

Once constructed, a ParseTree instance has five main methods of interest from an external perspective: predictDropProb(), predictSplitProb(), predictSplit(), learn(), and printIndented(). The first of these simply runs the BinaryLR classifier to determine whether that node thinks anything should be dropped from the sentence. The second is only intended to be run on the ROOT node, and it calls predictDropProb() on the node's first child (the S node). The predictSplit() method is almost the same as predictSplitProb(), but it outputs True if the probability exceeds the split-probability threshold or False if not. The learn() method simply calls the classifier's learn() method after ensuring that the argument is either 1 or 0. Finally printIndented() is intended for direct use by a human for visualizing a parse tree; it was among the first methods written, as it helped to analyze what tendencies of the parse trees typically led to positive classifications. The manual parse-tree analysis mentioned in the Approach section was performed with the help of this method.

The s_split module acts as a wrapper for ParseTree enabling training and testing of models. It should not be necessary for implementing a model into another application; it is intended only for training, tuning, and testing new models. Its usage can be found in the README file.

Results

Tuning

The baseline set by a previous attempt was described only in vague terms. It was approximately 70% by some metric, but further details as to the specific value or the performance metric used were unavailable. Therefore, the testing procedure was designed to output both accuracy and F1 score; precision and recall were included as well in order to aid in tuning the balance of errors between false postives and false negatives.

Various learning rates and training periods were used for tuning against the development partition, and it was determined that a learning rate of 1.0 and a training period of 100 epochs were optimal. Models were then trained with split-probability thresholds ranging from 0.1 to 0.9 in increments of 0.1. They were tested on the development partition of the data set, yielding the results shown in Table 1.

It should be noted that the model files consistently showed weights for child outputs around -1500. While it was a surprise that the value would be negative, the magnitude—and consistency—of the value showed that the input of the children was likely important in some manner.

Prob. Threshold	Accuracy	Precision	Recall	F_1 Score
0.1	62.8099899048%	83.3701831965%	41.5702023781%	55.4779044717%
0.2	69.5650265549%	69.6797979108%	80.3685329554%	74.6434579097%
0.3	65.8078391783%	80.6620861961%	50.8465233483%	62.3744204019%
0.4	65.3030768556%	82.1658615137%	48.2163949917%	60.7711776091%
0.5	70.2190229557%	71.3193943764%	77.8958973147%	74.4627197109%
0.6	67.8971162709%	76.2401325407%	61.6032758485%	68.1445993031%
0.7	63.7931791248%	61.1194402799%	96.3067958107%	74.7806414137%
0.8	67.1816705438%	64.2319851739%	92.7947082447%	75.9156063778%
0.9	68.7398498881%	66.4503569111%	88.6998976297%	75.9797639123%

Table 1: Dev-Set Accuracy by Probability Threshold

Possibly stemming from the different split-probability thresholds, the different models had varying balances of precision vs. recall. Regardless of the reason, the variation made them promising candidates for a voting ensemble. Such a model was created to see if it could improve upon the individual development-set performance values. The results are tabulated by voting threshold in Table 2. (Note that the test was performed with the split-probability threshold set to 0.9.)

Voting Threshold	Accuracy	Precision	Recall	F_1 Score
1	60.0184347979%	58.2925251779%	99.3621545004%	73.4779443878%
2	66.0668041961%	62.7241061367%	96.4249153477%	76.0063312746%
3	70.3111969451%	67.1799918949%	91.3772738011%	77.4322701188%
4	72.7384453320%	71.6122584943%	84.6444601937%	77.5849002129%
5	72.9491287363%	74.2649242649%	78.7621072525%	76.4474337907%
6	71.2285476013%	78.7910028116%	66.2020631546%	71.9500192563%
7	66.8349207743%	82.5877582056%	51.3190014962%	63.3025740651%
8	63.1742966247%	84.5771144279%	41.4993306560%	55.6788166931%
9	57.9642716060%	87.8332525448%	28.5376801323%	43.0787518574%

Table 2: Dev-Set Accuracy by Voting Threshold

As shown in Table 2, voting thresholds of four and five performed the best overall when considering all of the displayed statistics. A voting threshold of five was chosen in order to minimize the difference between precision and recall, thus keeping the error probabilities balanced between false positives and false negatives.

Testing

The voting ensemble with a threshold of five votes, having been determined the best choice based on the development partition, was tested against the testing partition in order to ensure its performance wasn't due to hyperparameter overfitting. The results are described in Table 3.

Voting Threshold	Accuracy	Precision	Recall	F_1 Score
5	71.3132022472%	72.0122956965%	77.4097356544%	74.6135321992%

Table 3: Testing-Set Accuracy of Final Model

Statistical Significance

Although the model performed slightly worse on the testing partition than on the development partition, it still outperformed the baseline of 70% in all statistics. In order to determine statistical significance, it was tested again using bootstrap resampling with 10000 samples. Remarkably, each of the four statistics from Table 3 showed a p-value of exactly 0.0 that the results had beaten the baseline by chance. This is to say that in none of the 10000 samples did it fall at or below 70% accuracy, precision, recall, or F1 score.

As a sanity check, the bootstrap resampling procedure was performed again with a baseline of 90% to confirm that the procedure worked as intended. As one would expect from a properly functioning test, the results showed p-values of 1.0 for all statistics. This result confirmed that the procedure did not produce zeros due to a flaw in the program. Still, as one final check, the resampling was performed one last time with a baseline of 0.71313, just barely below the accuracy from Table 3. This time, accuracy gave a p-value of 0.4329—not statistically significant, as one would expect—precision gave 0.0369, and the other values both gave zeros. This is all as one would expect from a working resampling procedure.

Given the p-values of zero for all measured performance metrics, the improvement over a 70% baseline is highly statistically significant by each such measure.