Concept

We want to solve the Multilabel-Classification task using two different approaches. One is using a neural network the other a one vs. all multinomial naïve bayes classifier. To apply these methods, we first need to transform the input data into a format that can be used to feed to the corresponding classifier. Therefore we try different approaches. The simplest being tf-idf bag of words representation for the Multinomial naïve bayes. Another one is the bi-gram bag of words representation which offers some benefits since it catches sequential features like word order that are inherent in textual data. It comes with a price though since it drastically increases the size of the resulting feature vector. For the neural network we use a dictionary to transform every word in the document to an index which we will later use in a look-up table. Since not every document has the same length we apply batch wise padding, always padding to the size of the longest document in the corresponding batch. This representation has the benefit of keeping the complete sequence order and is not increasing the feature vector size. Those feature vectors now need to used for some model. First we will build a neural network using a LSTM model. This allows us to exploit the sequential nature of text data to a maximum since LSTMs are build for exactly that: sequential data. But we don’t just put the indices into the network but before passing them through an Embedding/Look-Up Layer. This Layer creates a meaningful word embedding feature vector for each word that has much richer information than just the index and hence provides better information for the actual LSTM layer. We then map the output through a tanh function to a number-of-labels sized output vector which should be equal to the labels. To make that possible we transform the labels to an vector as well. Here we use the one-hot-vector approach or since it is a multilabel problem ‘multi-hot-vector’. This means every label is mapped to an index and if a document has that label, so the label vector has a 1 at the corresponding indices and zeros otherwise. Since it is a Multilabel classification problem we use the bp-mll cost function (Zhang and Zhou 2006) to address this problem. To take care of the imbalances we additionally penalize the error of not so frequent instances heavier then the error for often occurring instances. For our second model the one vs. all multinomial naïve bayes classifier we first need to augment the dataset to create a binary classification task for each label in the dataset. We then train a binary mnb classifier on each label instance where the positive examples are all documents containing that label and the negative examples are the rest. We use a multinomial approach since this one is known to be best for textual data. To infer the labels of an unseen document we run the document through all the classifiers and assign the label to it if the corresponding classifier outputs a 1. To take care of the imbalances we also want to adapt the one vs. all approach to an one vs. some approach. This means we will have a proportional amount of negative examples to positive examples for each mnb classifier. This avoids cases that have 1 positive example and 19100 negative examples which will most likely result in just returning a zero for every possible document.

Implementation

Preprocessing Stuff @TODO Chetan.

To transform the preprocessed documents and labels into feature vectors we created dictionaries for all the words occurring in the documents and labels. Since the documents are supposed to be English we filtered out all words containing non-Latin letters. In the dictionaries the keys are the words/labels and the values are the indices that they are assigned to. We also build a bigram dictionary which is build analog to the word dictionary with the difference that instead of using single words we iterated over all pairs of words that are next to each other. For the bigram dictionary we threw out all bigrams that only occurred once which reduced the feature vector size from roughly 3M to 1.7M. Which is still enormous compared to the word dictionary size which is around 160K. Furthermore we calculated the IDF for each word and saved it in a dictionary to use it later for the tf-idf vector representation. For the labels we also calculated a penalize dictionary which mapped each label to the inverse of its frequency e.g. if a label occurs in 30 documents the value assigned here would be 1/30. We use this value later for the penalization in the loss function. The actual transformation to the feature vectors is done while the model is run since the feature vectors are the ones that cause trouble when it comes to memory constraints. So far the libraries used are **pickle, re, math and random**.

For the implementation of our neural network we used **tensorflow 1.13.1** and an implementation of the BP-MLL loss function in tensorflow from this repo:

<https://github.com/vanHavel/bp-mll-tensorflow>

To add the penalization term we had to adapt the bp-mll function. The bp-mll function was originally defined as



where Y is the set of labels that a document x is associated with. being the complementary set to Y and is the prediction of the i-th label based on the instance x.

To implement the penalization, we changed the formula as follows:



Where is the number of occurrences of label i with respect to the complete dataset. By that we reduce the error for labels that occur often and keep the error the same for labels that only occur once in the whole dataset. Thus penalizing rare labels compared to frequent labels. To implement this, we changed the source-code of the bp-mll implementation.

To handle the high dimensional feature vectors with constrained memory resources we build a generator function which only translates the current batch which is fed to the neural network into the feature vector representation. So we only load the whole dataset, where the documents are strings and the labels are a list of strings, into memory and then generate the memory intensive vector representations on demand.

For the multinomial naïve bayes approach we additionally use the libraries **scikit-learn, random and itertools.** Again we use a generator approach which only builds the final feature vectors, here tf-idf vectors, when its needed for training. Next we can choose if we go for the one vs. all approach or the one vs. some approach. The one vs. all is the typical text book approach where we just use the documents corresponding to a label as positive samples and all other documents as negative ones. Where as in the one vs. some approach we count how many positive instances where found for one label and then multiply that number with some scalar. The result is the number of documents we randomly sample from the dataset that remains when removing the positive instances for that label. To not run in an out of bound exception we need to be careful that the most occurring label times the scalar value does not exceeds the number of available negative examples. This allows us to assign an equal class distribution to all classifiers where we still manage to represent the skewed ratio of positive to negative samples for each of the labels. We then proceed to train one multinomial naïve bayes classifier for each label in the labelset using the dataset created for each label as described before. The trained classifiers are then saved in an array and written to disk. For inference we load the model array and run all ~4000 classifiers on each instance. Since the output is either 0 (negative example) or 1 (positive example) we can concatenate the results of all classifiers to a vector and get the predicted label vector which we can compare to our truth label vector.

References:

Zhang and Zhou 2006:

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING 2006,  
Multi-Label Neural Networks with Applications to Functional Genomics and Text CategorizationMin, Ling Zhang and Zhi-Hua Zhou