Fextractor.py Code Architecture

Crystal (Ying) Qin (yq37)

1 Serve Corenlp on a server

We first set up the Corenlp parser on a json rpc server by calling *jsonrpclib.Server("http://127.0.0.1:8080")*.

2 PrepareExtractor

(1) Load lists from files into sets in memory.

(2) Reading in all the articles from DB and cleaning the htmls, store the texts in self.texts, build corpus from self.texts and store it in self.collection for later use to calculate tf_idf.

3 ExecuteExtractor

(1) **generate_feature_datasets**

Goal: using this method, we generate lists

- train_set: each element is a vector of features for a word.
- targets: each element is a label (whether it's a framing word or not) for each word.
- doc_offsets: a dictionary maps doc_id to offsets into train_set and targets.
  format [(ind1_s, ind1_e),(ind2_s, ind2_e),..]: indi_s: offset of the position of the ith doc's first word's feature vector into the train_set and label into the targets. ind_e: such offsets of the next doc's first word.

Process: We go through each document:

(a) Record this doc's id in self.docID, read in the html of the document from the database, clean the html and store all the title words in self.title_words.

(b) Run corenlp on each sentence of this document to extract information of each sentence.(e.g. the dependency relations, the POS, charoffset of the words in that sentence, etc.) Then, store the information of all sentences in one doc into self.coreParsed.

(c) Fetch all the annotations for this document from DB. Store the start indices into a 2-D array self.start_indices and the end indices into a 2-D array self.end_indices.

Format: self.start_indices: [[a1_s1, a1_s2, a1_s3...], [a2_s1, a2_s2, a3_s3...] ...] ai_sj: means this doc's ith annotation's jth highlighting's start index. self.end_indices has the same format as above but for the end indices.

(d) Go through self.coreParsed, which gives information of each sentence in the doc. Then extract the feature vector for each word in the sentence by calling self.generateFeatures on each word.

self.generateFeatures :

This method generate the feature vector for each word based on its context (the sentence it resides in). By turning on/off the feat_* bits at the beginning of the code, we could control which sets of features we want to include in our study.

We first build tokens list, lemmas list and stems list for the input sentence and then use the information we stored in self.coreParsed and standard nltk tools such as TFIDF to extract features. Please refer to the code to see the list of features we are extracting.

(e) Lastly, iterate through all the annotations of this doc for that word; for each annotation on that word, append the feature vector into train_set, append label 1 to targets if the word is highlighted in the current annotation, 0 otherwise.

(2) **Evaluation and Testing**

We use Scikit-learn tools here.

(a) We first transform featuresets generated in the first step (i.e. by generate_feature_datasets()) into a sparse matrix using sklearn's DictVectorizer() method.

(b) **Then split into train set and test set:** We can choose to do a doc_level evaluation or word_level by turning on/off the doc_level bit.

If we choose doc_level, then we shuffled the doc ids (by using random.shuffle)and randomly choose say 90% documents as the train set and make the rest the test set. Then we find all the word feature vectors for our selected train/test documents by using self.offsets to offset into the complete feature datasets, then use the vstack method in python stack them together into sparse matrices.

If we choose word_level, we shuffled (by using random.shuffle) all the word data points and randomly choose say 90% word labeled feature vectors as train set and 10% as test set using sklearn's method cross_validation.train_test_split().

(c) For cross-validation option, we use Scikit-learn's cross-validation tools such as shuffle-split() in our self.crossValidation method.

(d) **Train classifiers on train sets and Benchmark the performance using test sets:**

In our runAllClassifiers() method, we choose a set of classifiers that are able to run over sparse matrices and then call self.benchmark() on each of it.

In the self.benchmark() method, we first train the classifier by calling classifer.fit() on the train sets. Then call classifier.predict() on the test sets to get prediction results of our classifiers on the test data.

Then we calculate the f_score, accurary score, precision score, recall score, confusion matrix, classification report, most infomative features of our classification by comparing the predicted and actual labels. All of these are done by using sklearn's built-in functions.

(e) We also record the train time and test time of each classifier and has a very basic draw_diagram function that visulize this information.