CIS 530 Fall 2014 Homework 3

Instructor: Ani Nenkova
Head TAs: Mounica Maddela and Jessy Li

Released: Nov 1, 2014
Due: 11:59PM November 15, 2014

Overview

In this assignment you will experiment with an off-the-shelf logistic regression classifier. The focus will be on feature engineering, that is representing the text as a feature vector of numeric descriptions of the input. We will use Stanford coreNLP to pre-process the text in order to get syntactic features. We will use pre-computed word2vec word representations to expand the standard lexical representations. We will define the features and you will have to implement modules that compute these features for a given text. You will train classifiers using these features and the use them to perform document classification task. As you do the homework you will get experience working with Stanford coreNLP and the LibLinear classifier, as well as training of word2vec if you choose to do the extra credit.

For this assignment, we will use the same data as the one in homework 2. Our corpus contains documents from four genres: Health, Research, Finance and Computers and Internet. The data necessary to complete the homework is placed in:

```
/home1/c/cis530/hw3/data
/home1/c/cis530/hw3/test_data
```

To run your code, please use the Biglab machines. Large jobs will be automatically terminated on Eniac. For instruction on how to use Biglab, please refer to the link provided below:
`http://www.seas.upenn.edu/cets/answers/biglab.html`

Submitting your work

The assignment should be done individually, so each student should make one submission with their own code. The code for your assignment should be placed in a **single file** named `hw3_code_yourpennkey.py`, where `yourpennkey` is the first part of your Penn email address. For this assignment you will also submit: performance results for classifiers in `results.txt` and `extra_credit_results.txt` (if you have attempted extra credit). *Please make sure you submit all requested files in the specified format.* Place all the required files in a folder named `hw3_yourpennkey` and zip it.

All submissions will be electronic, done from your Eniac account. You need to copy your files to Eniac if you choose to work elsewhere. To copy files you can use a SFTP client such as FileZilla, WinSCP or Cyberduck, or secure copy `% scp local_file yourpennkey@eniac.seas.upenn.edu:PATH` where `local_file` is the path to your homework on the local machine and PATH is the path to the location on Eniac where you wish to copy it.

Once you have the file in place in your Eniac account, connect via ssh to `seas.upenn.edu` and use the `turnin` command to submit your files for grading:

```
% turnin -c cis530 -p hw3 hw3_yourpennkey.zip
```

You will get a confirmation message. You can run `turnin` multiple times before the deadline. Each time you run `turnin`, it overwrites your previous submission for that assignment. You can check that the homework was submitted successfully:

```
% turnin -c cis530 -v
```

This will show you the list of files you have submitted.

Code Guidelines

In the assignment below we specify a number of functions. *Your submission should include implementation for each of these required functions, with the specified number and types of arguments.* In addition, you may write any extra functions that you find necessary. Avoid doing any preprocessing of the input text besides that explicitly described in the assignment. This will make grading easier.

Your code will be graded automatically. It is your responsibility to make sure your function can be called correctly and that they return exactly the type of values specified in the assignment. To make sure that your code will run correctly during grading, please include **only** function declarations in your source code or above the line `if __name__ == "__main__"`, and prefix **all** your global variables with `yourpennkey`. Before submitting, please make sure that your code runs. Further, the sample answers given for each question are not correct. They are specified to help you understand the output format.

1 Processing corpus to extract feature sets

In this homework, the task is to predict the NYT section in which a news article appeared, roughly equal to predicting the general topic of the article. We will represent the articles with three different types of features: lexical, syntactic dependencies and syntactic production rules. The first task is to preprocess the data using Standford coreNLP. Standford coreNLP will generate the xml files for the corpus which will be used to extract features.

To run Stanford CoreNLP, we have to specify a list of annotations which we want it to produce, a list of input files, and the output directory. A copy of the Stanford CoreNLP toolkit is located in the homework folder here:
`/home1/c/cis530/hw3/corenlp/stanford-corenlp-2012-07-09`

Suppose you have downloaded CoreNLP (225MB) on your local machine or Eniac and you are in the folder which includes `stanford-corenlp-2012-07-09.jar`. You can execute the following command to generate xml files with the required annotation:

```
java -cp stanford-corenlp-2012-07-09.jar:stanford-corenlp-2012-07-06-models.jar:xom.jar:
joda-time.jar -Xmx3g edu.stanford.nlp.pipeline.StanfordCoreNLP -annotators tokenize,ssplit,
pos,lemma,ner,parse -filelist file_list.txt -outputDirectory <OUTPUT DIRECTORY PATH>
```

Here `-filelist` specifies a raw text file: `file_list.txt`, which includes all the files you want to process. Each line contains a filename with an absolute path to the file. `-outputDirectory` specifies the directory where the output will be saved. `-annotators` list specifies the task which you want CoreNLP to execute. With the list of arguments listed above, we enable tokenzation, sentence splitting, POS tagging, lemmatization, NER and syntactic parsing.

```
-annotators tokenize,ssplit,pos,lemma,ner,parse
```

You can either download CoreNLP to your local machine or on Eniac directly call the version in the homework folder. If you run the CoreNLP from the homework folder, the command line will be slightly different because you will need to specify the absolute path to the tool. You may find more details regarding how to run the CoreNLP toolkit at:
`http://nlp.stanford.edu/software/corenlp.shtml`

Run the CoreNLP toolkit to produce annotations for the training and testing data. Only the `file_list.txt` and the output directory will be different in the two calls to the toolkit. After running the tool, you will see a list of `.xml` files in the output directory. They contain the annotations for the respective files listed in `file_list.txt`. Here is a sample output file:
`http://nlp.stanford.edu/software/corenlp_output.html`.

In grading, we will not call a specific function to check how you ran the tool. Nevertheless, we expect you to use python to do a system call and run CoreNLP rather than type the command on the console prompt.

2 Lexical features (10 points)

In text classification tasks, unigrams have proven to be a simple but powerful feature. A fixed vocabulary is chosen, then each document is represented in terms of the unigrams that occur in the document.

2.1 Creating a list of words that frequently occur in the training data (5 points)

Write a function `extract_top_words(xml_directory)` that takes an xml directory path and returns the 2000 most common words from the documents in the directory. Here, word refers to the string in between the `<word> </word>` tags in the xml file. Lower case all words before counting the number of occurrences.

```
>>> extract_top_words(xml_directory)
['new', 'york', 'starbucks' ... ]
```

However, do not hard-code the path and implement a function that can in general take any path as an argument.

2.2 Lexical representation of a specific text (5 points)

Write a function, `map_unigrams(xml_filename, top_words)` that takes in a file and the list generated from the previous function and returns a vector in the feature space of top_words. It returns a list of 0's and 1's with the same size as `top_words`. Each component in the returned list (denoted as `output_list`) represents the presence of the corresponding top word in the input file. If a word appears in the input file, the corresponding value is `1` and `0` otherwise.

```
>>> map_unigrams(xml_filename, top_words)
[1, 0, 1, 0, 0, .... ]
```

3 Lexical Features with expansion (25 points)

3.1 Calculate word2vec similarity (10 points)

Write a function `extract_similarity(top_words)` that takes list of words and returns information about the words in the list that have non-zero cosine similarity between each other, along with the similarity value. Specifically, the output is a dictionary where the key is a word in the input list and the value is another dictionary of words (from the original list) with non-zero similarity with the key. It is a dictionary of dictionaries.

Here, we give you precomputed word2vec vectors in the file */project/cis/nlp/tools/word2vec/vector.txt*. The similarity between two words is the cosine similarity between their corresponding vectors.

```
>>> extract\_similarity(top\_words)
{'w0': {'w1': 0.9, 'w100':0.2}, 'w1': {'w0': 0.9, 'w6': 0.5 .....}, 'w2' ": { .....
```

3.2 Lexical representation with expansion (15 points)

Write a function, `map_expanded_unigrams(xml_file, top_words, similarity_matrix)` that takes in a file, top_words created in 1.1 and the similarity matrix generated from the previous function. It returns a list of real numbers between 0 and 1 with the same size and order as `top_words`. Each zero in the initial binary lexical representation is replaced by the maximum similarity between the word corresponding to this component and and of the words with component equal to 1. In this way the representation captures semantic similarity between words that is not directly expressed in the document that we are representing.

Below we give you a pseudo-code outline of how to construct the enriched vectors:

```
Extract unigram vector for the given file.
Get the non-zero top_words from the vector
For each zero component/top_word of the vector:
   Replace its corresponding position in the vector with the maximum of similarity
   scores with the non-zero top_words.
```

```
>>> map_expanded_unigrams(xml_filename, top_words, similarity_matrix)
[1, 0.1, 1, 0.5, 0, .... ]
```

4 Dependency features (15 points)

Our syntactic features will represent a text in terms of the dependency relations in the text. Dependency relations hold between two words in a sentence. The relation is not symmetric and one word is considered as more important (the head word in the relation) and the other word is the dependent. The intuition of using such features is that knowing the arguments of the dependency will be more helpful in evaluating an event. A possible issue is sparsity because in this representation features are pairs of words in a specific relation which will occur much less often than the individual words.

For clarity, we show an example sentence and the dependencies we will extract from it. In the following sentence (with POS-tags in bracket):

```
he(PRP) reaffirms(VBZ) the(DT) debt(NN) for(IN) the(DT) full(JJ) $(MONEY) 1,500(MONEY) .(O)
```

two of the dependencies are:

```
<dep type="nsubj">
    <governor idx="2">reaffirms</governor>
    <dependent idx="1">he</dependent>
</dep>
<dep type="dobj">
    <governor idx="2">reaffirms</governor>
    <dependent idx="4">debt</dependent>
 </dep>
```

4.1 Extracting dependency relations (10 points)

Write a function `extract_top_dependencies(xml_directory)` which returns the 2,000 most frequent dependency relations from all documents in xml_directory. The output is a list of tuples. Each tuple is of the form (A, B, C): the first element is the dependency relation, the second is the governor (head) word, the third is the dependent word.

```
>>> extract_top_dependencies(xml_path)
[(nsubj, reaffirming, Corp.), (aux, reaffirming, is), ... ]
```

Again the only input we will test is the path to the directory containing the .xml files for the training data.To get the dependency relations from the .xml files, you only need to extract the information enclosed in the `<basic-dependencies>` and `</basic-dependencies>` tags. *Lowercase the words before creating tuples.*

4.2 Mapping Dependency Relations (5 points)

We are now ready to produce a representation that reflects the occurrence of dependencies in each text. Write a function `map_dependencies(xml_filename, dependency_list)` that takes an xml file and a list of dependency tuples as input and returns a list of the same length as `dependency_list`. The element in the list takes the value **1** if its corresponding tuple in `dependency_list` appeared in the xml input file, **0** otherwise.

5 Syntactic productions (20 points)

Our syntactic features are production rules from the constituency parse of the sentences in the text. Each node in the parse tree along with its children form a production rule. Here, we consider only the non-terminal nodes for production rules to reduce sparsity—no lexical features whatsoever appear in this representation.

For instance, in the following parse tree generated by coreNLP :

```
(ROOT (S (NP (DT The) (NN program)) (, ,) (VP (VB run) (ADVP (RB jointly)) (PP (IN by)
(NP (NP (DT the) (NNP  United) (NNPS Nations) (NNP University)) (CC and) (NP (NP (DT the)
(NNP University)) (PP (IN of) (NP (NNP Ulster))))))) (. .)))
```

the extracted production rules are:

```
['ROOT_S', 'S_NP_,_VP_.', 'NP_DT_NN', 'VP_VB_ADVP_PP', 'ADVP_RB',
'PP_IN_NP', 'NP_NP_CC_NP', 'NP_DT_NNP_NNPS_NNP', 'NP_NP_PP',
'NP_DT_NNP', 'PP_IN_NP', 'NP_NNP']
```

The non-terminal category of each node along with its children form a production rule. The non-terminals in the production rules should be joined with _. We consider only non-terminal nodes here, not including the grammar rules from non-terminals to terminals. This is equivalent to considering the part of speech tags as the leafs of the tree.

5.1 Extracting syntactic production rules (15 points)

Write a function `extract_prod_rules(xml_directory)` which returns all the syntactic production rules for the parse trees in the input xml directory. To get the parse trees from the .xml files, you only need to extract the information enclosed in the `<parse>` and `</parse>` tags.

Hints: You have two approaches to extract the rules, either using a stack or a recursive method. For the stack approach, split the original string w.r.t. spaces which gives rise to two cases: one parenthesis on the left indicating the start of a non-terminal node and multiple parenthesis on the right indicating a terminal node.

Each parenthesis takes off one node from the stack. This will be a child for the top-most element remaining on the stack. You can also extract the grammar rules recursively. For a given parse string, split it into children parse strings and process them recursively. For extracting the children parse strings, start with the open parenthesis immediately after the tag and search for the corresponding closing parenthesis. This will be your first child string. Similarly, extract the other children, note the grammar rule and process the children recursively. *Make sure you do not include production rules for terminal nodes. No lexical information should appear in this representation.*

These approaches are just guidelines. Feel free to implement your own algorithm. Implementing parser module as a class will simplify the code. Here is a sample template for the parser to extract production rules:

```
class Tree:
   Data structure for each node in the constructed tree
   Function for extracting production rules.

class TreeParser:
   Functions to traverse the tree.


>>> extract_prod_rules(xml_directory)
['ROOT_S', 'S_NP_,_VP_.', 'NP_DT_NN', 'VP_VB_ADVP_PP', ... ]
```

5.2 Mapping syntactic production rules (5 points)

Write a function `map_prod_rules(xml_filename, rules_list)` that takes an xml file and a list of rules in the format specified above and returns a list of the same length as `rules_list`. The element in the list takes the value **1** if its corresponding rule in `rules_list` appeared in the xml input file, **0** otherwise.

6 Logistic regression classification using LibLinear (20 points)

6.1 Data format and features

Using the features generated from previous steps, we can now predict the type of a news article (given by the newspaper section that the article was published in). In this section we will create binary classifiers to predict if an article is of a given type or not. In the next section, we will combine the binary classifiers for all of the domains to create a multi-class document classifier. We will use the LIBLINEAR toolkit, which provides off-the-shelf implementation of logistic regression for classification and regression. More details about the package can be found here:

`http://www.csie.ntu.edu.tw/~cjlin/liblinear/`

To train a classifier, we must format our data representation according to the LIBLINEAR specifications:

```
[Label] [Index1]:[Feature1] [Index2]:[Feature2] ...
[Label] [Index1]:[Feature1] [Index2]:[Feature2] ...
```

Each line represents a single instance (document) in the training set. The class of each instance is represented as a label, followed by feature index and the corresponding feature value. For labels, we use **+1** if the document is of the specified type and **-1** otherwise. The features for each document are generated from the functions you have written above. Basically, given a feature vector v for each instance, you represent every non-zero component in this vector in LIBLINEAR format files. Suppose we have $v_k \neq 0$ in vector v, then this component can be represented as $k+1 : v_k$, here $Index = k+1$, $Feature = v_k$. For example, the vector $v = [10, 0, 9, 0, 0, 8]$ is represented as:

```
1:10 3:9 6:8
```

Notice that the indexes for representing one instance are strictly monotone increasing.
Here, you will experiment with five different feature representations. The numbering of representation we give here corresponds to the feature mode in the rest of the homework.

```
1 -> binary lexical
2 -> lexical with expansion
3 -> binary dependency relations
4 -> binary syntactic production rules
5 -> All except expanded lexical features (1+3+4)
```

6.2 Processing corpus into intermediate format. (5 points)

In this section, we convert the given corpus xml files into intermediate format to simplify the process of creating multiple input files in LIBLINEAR format for each type of article. The function below ensures that you process the xml_files and construct the feature vectors only once.

Write a function, `process_corpus( xml_dir, top_words, similarity_matrix, top_dependencies, prod_rules)` which uses the provided data and the features you have extracted in Section 1-5. The `xml_dir` specifies the place where your function loads CoreNLP results. top_words is the list of words extracted in 1.1. top_dependencies is the list of dependencies extracted in 4.1. prod_rules is a list of syntactic production rules extracted in 5.1. similarity_matrix gives the word2vec similarity of each pair of top_words which is extracted in 2.1.

The function simultaneously creates intermediate format files for all five sets of features. The output will be five files, one for each feature set. The format for each file is as follows.

```
[Filename] [Index1]:[Feature1] [Index2]:[Feature2] ...
[Filename] [Index1]:[Feature1] [Index2]:[Feature2] ...
```

This format is very close to the LIBLINEAR, except that the label is replaced by filename. You will call this function twice, once for training data and once for testing data. The output files should specify if they are training or testing features and the feature set that they contain. For example, if output file corresponds to dependency features for training data, then the filename would be train_3.txt; if the output file corresponds to lexical features for test data, then the filename would be test_1.txt. The feature mode for each set is defined in 6.1. Please follow this naming convention when generating the files to facilitate grading.

Here we give an example of how to generate intermediate format files with this function. Consider the case when we only use lexical features. Then for each file F in `xml_dir`, you call `v = map_unigrams_(F, top_words)` to get the feature vector for the file.When you are combining vectors of several feature groups, you can just extend the vector first, then produce the `index:feature` pair.
Each line should look as follows:

```
Computers_and_the_Internet_2005_01_01_1638703.txt.xml 1:1 2:1 3:1 4:1 5:1 6:1 7:1 8:1 9:1
10:1 11:1 12:1 13:1 14:1 15:1 16:1 17:1 18:1 19:1 20:1
```

6.3 Creating the LIBLINEAR-format files

Now that you have obtained the intermediate format files, you can create LIBLINEAR format files for each binary classifier. In this section, you have to create input data for 20 binary classifiers (4 domains X 5 feature sets). For each (domain, feature_set) pair, you will generate two files, one for training corpus and one for test data. All you have to do is to replace the filenames in intermediate format files with labels. For each domain, all the files of that domain should get the label 1 and the rest will get the label -1.

Sample format:

```
1    1:1 2:1 3:1 4:1 5:1 6:1 7:1 8:1 9:1 10:1 11:1 12:1 13:1 14:1 15:1 16:1
-1  29:1 30:1 31:1 32:1 33:1 34:1 35:1 36:1 37:1 38:1 39:1 40:1 41:1 42:1 43:1
```

6.4 Train and test classifiers (15 points)

Write a function run_classifier(train_file, test_file) to train a classifier and test it using test data. train_file is the LIBLINEAR formatted representation of training corpus. Similarly, test_file is the LIBLINEAR formatted file for the test data. You should train the model, test it on held-out data and report the predicted labels, performance measures of classifier and probabilities of label 1. You should compute Precision (P), Recall (R), and F-Score (F) by comparing the prediction and the actual labels on the test set for both Positive and Negative samples for each classifier. You can create a tuple for performance measures `(Pos-P, Pos-R, Pos-F, Neg-P, Neg-R, Neg-F, Accuracy)`. The output is a tuple with the list of predicted labels as the first entry, the performance measures as the second entry and probabilities of label 1 as the third entry. F,P,R are real numbers between 0 and 1. Accuracy is a number between 0-100.

Note : Here, you can use liblinearutil (LIBLINEAR python API) to train and test classifiers. DO NOT USE COMMANDLINE FOR TRAINING AND TESTING MODELS. Call this function for each (domain, feature_set) pair. Report the performance measures for each (domain, feature_set) pair in results.txt. Each line in the file corresponds to performance measures of one type of article and one feature set.

The following lines should be written in results.txt:

```
# 0.53 0.63 0.57 0.8 0.4 0.533 80.3   Research:1
# 0.3 0.3 0.7 0.8 0.4 0.533  82 .5  Research:2
# 0.5 0.6 0.5 0.6 0.3 0.3   79 Research:3
# .......   - Research:4
#.........
# ......      - Finance:1
#.........
# ......      - Computers:1
#.......
# .......     - Health:1
```

Note: When training the classifier use -s and -wi options. Read the documentation to understand what they mean. Use 0 parameter for the -s option which means logistic regression. The training data for training the binary classifier is imbalanced, i.e. the number of negative instances is at least double the positive instances. To deal with this imbalance, you should specify -wi option which deals with the imbalance. Feel free to experiment with other weights but keep in mind that the weight of the negative class should be smaller than that for the positive class.

7 Document classification task (10 points)

Now we use the feature set with the best accuracy to assign a single class to each of the documents in the test data, indicating the most likely type of the article among the four possibilities. From the above experiment, you can get the probability that an article to be from each of the four newspaper sections that we consider. To decide a single most likely class, we assign to the article the the type with highest probability. Accuracy should be float number between 0 and 100.

Write a function classify_documents(health_prob, computers_prob, research_prob, finance_prob) which takes the list of probabilities for each type and returns the type with highest probability. The output would be list of labels $\epsilon\{"health","research","computers","finance"\}$. Report the accuracy of your predictions in the last line of results.txt.

8 Extra Credit : Train the word2vec vectors with given corpus.(25 points)

In the above representations, we used the pre-trained word2vec vectors for calculating similarity matrix. The vectors are extracted from wikipedia data. For extra credit, you should retrain the word2vec using our training corpus and extract expanded lexical features using the new similarity matrix. You should report the performance results in extra_credit_results.txt file in the format mentioned above. Treat this feature set as 6th set. You should also report the accuracy of the feature set for the document classification task. You should also compare this result with other results and explain how the results reflect your expectations.