CIS 530 Fall 2013 HW 3

Instructor: Ani Nenkova
TA: Kai Hong, Jessy Li

Released: October 28, 2013
Due: 11:59PM November 12, 2013

In this assignment you will experiment with an off-the-shelf Support Vector Machine 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 define most of the features and you will have to implement modules that compute these features for a given text. You will also have a chance to come up with your own feature.

While doing the assignment, you will become familiar with several resources and tools: two subjectivity lexicons (lists of positive and negative words), Stanford CoreNLP (a flexible tool capable of doing most NLP tasks we cover in the course) and the LibSvm classifier (our off-the-shelf classifier).

The data necessary to complete the homework is placed in:

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

You will submit the code for the functions you have implemented for grading. We strongly encourage students to work in teams of two. Each submission should start with lines including the name and Penn email address of each person in the team. For example, if Kai works with Jessy, the first lines of the code submission should be:

```
# Kai Hong:  hongkai1@cis.upenn.edu
# Jessy Li:  ljunyi@cis.upenn.edu
```

NOTE: To run your code, please use the Biglab machines. Large jobs will be automatically terminated on Eniac. All students enrolled in CIS530 should have access to Biglab. To log in, use:

`% ssh your_penn_key@biglab.seas.upenn.edu`

Please contact the TAs immediately if you do not have access to Eniac or Biglab.

Submitting your work

The code for your assignment should be placed in a single file named `hw3_code_yourpennkey.py`. You also need to submit **two other files** for this homework. The first one is a xml file after processing the raw texts using Stanford CoreNLP: `98856.txt.xml`. The second file is a sample of your feature representation, representing a text in terms of the adjectives and verbs that appear in it: `train_4_postags.txt`.

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 an 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.

Let's assume you have your code organized within a folder on Eniac. You want to submit everything within your current folder, this could be done with:

```
% turnin -c cis530 -p hw3 -v *
```

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 file(s) you have submitted.

Code Guidelines

Your submission should include implementation for each of the functions described below. In addition, you may write any extra functions that you find necessary. Please add descriptive comments to your code. In the comments, detail any choices you have made regarding the treatment of the textual data. In general, avoid doing any preprocessing of the input text unless the homework explicitly asks you to make a choice. For example when we refer to "all tokens in the input" and ask you to define the types you work with, you may choose to ignore numeric strings; however in general all tokens would mean all alphanumeric strings.

Your code will be graded both automatically and manually. Please make sure your function can be called, otherwise you will lose points.

Throughout the assignment we give sample output for some of the functions that we ask you to implement. These are meant to simply give you a sense of what your answer should **look like** and is not necessarily the only correct answer. Often times there are multiple, equally reasonable alternatives for handling the data, each resulting in a slightly different answer. Whenever you encounter such situations, simply document your choice in the comments.

Data

For this assignment, we will use the same data as the one in homework 2. Our corpus contains summaries of events after which there was a 5% drop or increase of the excess return of the company, within two days following the news. You can find the training corpus in the
`/home1/c/cis530/hw3/data/` directory.

Our testing data, which you will use to evaluate the SVM-models you create, is placed in the
`/home1/c/cis530/hw3/test_data/` directory.
It contains 50 files with news which was were followed by $> 5\%$ drop of the excess return and 50 files of news followed by $> 5\%$ increase.

The excess returns for a 2-day period associated with each text file can be found in:
`/home1/c/cis530/hw3/xret_tails.txt`
You need to map the files with these prerformance values the same way you did for homework 2.

1 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.

1.1 Creating a list of words that occur at least 5 times in the training data (5 points)

Write a function `extract_top_words(directory)` that takes a path to the training data directory and return a list of all words that appear at least 5 times (≥ 5) in the given directory. Lower case all words before counting the number of occurrences.

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

For this homework, we will pass only the path to our training data as parameter `/home1/c/cis530/hw3/data/`. However, do not hard-code the path and implement a function that can in general take any path as an argument.

1.2 Lexical representation of a specific text (5 points)

Write a function, `unigram_map_entry(filename, top_words)` that takes in a file and the list generated from the previous function. It returns a list of integers with the same size as `top_words`. Each component in the returned list (denoted as `output_list`) is the number of times the corresponding word from `top_words` appeared in the input file, i.e. `output_list[i]` = number of times word `top_words[i]` appeared in `filename`. If a word does not appear in the input file, the corresponding value in the output list should be 0.

```
>>> unigram_map_entry(filename, top_words)
[1, 0, 3, .... ]
```

2 Sentiment Features (18 points)

Presumably the occurrence of a positive or a negative event associated with a company would be predictive of the company performance. To experiment to what extent word polarity, the crudest measure of the desirability of an event, can help predict company performance, we will rely on two hand-crafted widely-used sentiment lexicons: the MPQA lexicon and the General Inquirer lexicon.

The MPQA lexicon includes word which have been tagged with their perceived subjective polarity (**negative**, **positive** or **neutral**) and subjectivity type (**strong**, **weak**). The polarity class roughly indicates if the word is related to a desirable property or event; the subjectivity type reflects the degree of the positive or negative reaction. It can be found at:
`/home1/c/cis530/hw3/mpqa-lexicon/subjclueslen1-HLTEMNLP05.tff`
A readme file is also available in the same directory, and explains the organization of the data.

2.1 Loading the MPQA lexicon (4 points)

Write a function, `get_mpqa_lexicon(lexicon_path)` to access the MPQA corpus file above and return a dictionary which maps a word to a list of tuples. The first element of the tuple is the *word type* of the word, the second element is the *polarity* of the word. For ambiguous words with multiple polarities and types, all possible interpretations should be returned. The dictionary should look like:

```
>>> mpqa = get_mpqa_lexicon(lexicon_path)
>>> mpqa['best']
[('strongsubj', 'positive')]
```

The word **mean** for example can be either *strong negative* or *weak neutral*, thus the output entry corresponding to this word looks like this:

```
>>> mpqa['mean']
[('strongsubj', 'negative'), ('weaksubj', 'neutral')]
```

2.2 Feature vector from MPQA lexicon (6 points)

Write a function, `get_mpqa_features(file, dictionary)`, that given a text and the dictionary produced from the previous function, produce a list of three elements. These three elements are the number of **word tokens** with positive polarity, negative polarity and neutral polarity, respectively. If a word has multiple polarities, we should augment with one +1 all of the applicable element in the output list. For example an occurrence of the word "mean" in the text will result in the counts of both the negative and neutral components of the output to be incremented by one.

```
>>> mpqa_dict = get_mpqa_lexicon(lexicon_path)
>>> get_mpqa_features(file, mpqa_dict)
[10, 1, 0]
```

Write another function, `get_mpqa_features_wordtype(file, dictionary)`, that given a text and the dictionary produced from the previous function, produce a list of six elements. The first three elements are the number of words of strong subjectivity, with positive, negative and neutral polarity respectively; the other three elements are the analogous counts for weakly subjective words with positive, negative and neutral polarity. We take the word **mean** as an example here. Since it is both *strong negative* and *weak neutral*, you should get a vector of `[0, 1, 0, 0, 0, 1]` if a given file only includes this one word.

The lists generated by the two functions above create the MPQA-lexicon related feature representation of a text.

2.3 Loading the General Inquirer lexicon (4 points)

The General Inquirer (GI) is a lexicon of words tagged with their perceived subjective polarity, which can be either negative or positive. in GI, there are also various other tags, including *Pleasure*, *Pain*, *Arousal*, etc. More information can be found at:
`http://www.wjh.harvard.edu/~inquirer/homecat.htm`

The General Inquirer lexicon on Eniac resides here:
`/home1/c/cis530/hw3/gi-lexicon/inquirerTags.txt`

Write a function, `get_geninq_lexicon(lexicon_path)` to access the corpus file above and return a dictionary in which the value for a given word key is a tuple of four numbers. These numbers are binary indicators if the word has a property (value = 1) or not (value = 0). The four properties of interest are: (1) Positive polarity or not? (GI calls this **Pstv** or **Pos**) (2) Negative (**Ngtv** or **Neg**) polarity or not? (3) Strong (**Strng**) intensity or not? (4) Weak (**Weak**) intensity or not? Here we will ignore all other properties of the word described in the GI. Example below:

```
>>> gi_dict = get_geninq_lexicon(lexicon_path)
>>> gi_dict["make"] = (1, 1, 1, 0)
>>> gi_dict["malady"] = (0, 1, 0, 1)
```

2.4 Feature vector from General Inquirer lexicon (4 points)

Write a function, `get_geninq_features(filename, dictionary)`, that given a filename and the dictionary produced from the previous function, creates a **list** of four elements. These four elements are the number of tokens in the file provided as a first argument with positive polarity, negative polarity, strong intensity and weak intensity.

```
>>> geninq_dict = get_geninq_lexicon(path)
>>> get_geninq_features(filename, geninq_dict)
[4, 0, 2, 1]
```

3 Named Entity Features (15 points)

Systems for Named Entity Recognition (NER) identify words or phrases in a text which refer to entities of some predefined semantic categories such as *Person*, *Company*, *Location*, *Date*, *Money*, etc. Here we will create a text representations that reflects the presence of various named entities in the text.

For this and the following problem, we will rely on the Stanford CoreNLP system to derive the text mark-up necessary to create this representation. This system performs a number of NLP tasks such as part of speech tagging, syntactic parsing, lemmatization and named entity recognition. We will use named entity types, part of speech tags and dependency parse information as features in our classification task.

3.1 Running Stanford CoreNLP (10 points)

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 within 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 passed on 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.

NOTE (turnin): Please turn in the file `98856.txt.xml`, the CoreNLP output for `98856.txt`. Make sure that you run the coreNLP toolkit on BigLab. It requires a lot of memory and the process will be terminated automatically if you try to run it on Eniac. Start early! It takes about an hour to generate all the xml files.

3.2 Extract Named Entity Features (5 points)

We can now extract named entity information from the CoreNLP annotations. We will count the number of companies, people names, locations, money and dates mentioned in each file. Write a function named `extract_named_entities(xml_file_name)`, which takes as input the path to the XML file name and returns a list $[A, B, C, D, E]$. Here A, B, C, D, E are the number of companies, people names, locations, money entities and date entities for word tokens within the input file.

```
>>>extract_named_entities(xml_file)
[10, 5, 3, 4, 7]
```

Hint: Look for *Organizations, Person, Location, Money, Date* tags in the xml file.

4 Part of Speech Features (14 points)

Another set of features we are going to use here are Part of Speech tags. Adjectives and verbs are of particular interest, given that they describe the key properties and the event predicates. The motivation for this representation is the intuition that adjectives (such as *great*, *poor*) or some verbs (such as *improve*, *reaffirm*) could be helpful in predicting excess return.

4.1 Extracting Adjectives and Verbs (7 points)

Write a function `extract_adjectives(xml_directory)` which takes as input a directory with .xml files generated from Stanford CoreNLP and returns a list of **all the adjectives** that appear in the training data.

```
>>> adj_list = extract_adjectives(training_xml_path)
>>> adj_list
['good', 'bad', 'decent', 'nervous', ...]
```

For this homework, the only input we will test is the directory to the xml for the training data.

Similarly, write a function `extract_verbs(directory)`, which returns all verbs that appear in the training data.

Hint: Look for "JJ", "JJR", "JJS" for adjectives; "VB", "VBD", "VBG", "VBN", "VBP", "VBZ" for verbs, following the part-of-speech tagset in Penn Treebank:
`http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`.

4.2 Adjectives and Verbs Text Representation (7 points)

Write a function `map_adjectives(xml_filename, adj_list)`. This function takes a filename and a list of adjectives as input, returns a list of the same length. The value of element $a[i]$ from the output list is equal to 1 if the ith word in the adjective list is present in the xml file and is an adjective; 0 otherwise.

```
>>> map_adjectives(xml_filename, adj_list)
[1, 0, 1, .... ]
```

Similarly, write a function `map_verbs(xml_filename, verb_list)`, to generate a list `verb_list` that indicates the presence or absence of the verbs we track.

5 Dependency Features (15 Points)

Our syntactic features will represent a text in terms of the dependency relations associated with verbs 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. In most cases, verbs are regarded as head words in the relation. The dependents of verbs in dependency representations can be the arguments of the verb, or adverbial modifiers of the verb. The intuition is that knowing the arguments of the verb will be more helpful in evaluating an event.

For instance, 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)
```

The extracted verb-related 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>
```

Here, for simplicity, we find all verb-related relations by identifying some dependency relation types given in a pre-defined list. The list could be found at: `/home1/c/cis530/hw3/verb_deps.txt`

By using the dependencies listed in this file, you are free of the effort to analyze if the governor or dependent is a verb or not. You also do not need to check the dependency types not listed in the file, even if one end of the dependency relation is a verb.

5.1 Extracting Verb Related Dependency Relations (10 Points)

Write a function `extract_verb_dependencies(xml_path)` which returns all dependency and word tuples which appeared **at least** 5 **times** in the input. The dependencies are extracted by mapping the file listed above, where most of the cases involves a verb as a governor, removing duplicates. 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_verb_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. Note however, if you are only parsing, the Stanford Parser (`http://nlp.stanford.edu/software/lex-parser.shtml`) has an option which would allow you to list all dependencies directly, without analysis of the the xml as we perform here at Stanford CoreNLP. To streamline the extraction of features, however, we will derive all our features from the xml output directly.

5.2 Mapping Dependency Relations (5 Points)

We are now ready to produce a representation that reflects the occurrence of verb-related dependencies in each text. Write a function `map_verb_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.

6 SVM Classification (20 points)

6.1 Data Format

Using the features generated from previous steps, we can now predict the drop or increase of excess return for the files in the test set. We will use the LIBSVM toolkit, which provides off-the-shelf implementation of Support Vector Machines for classification and regression. More details about the package can be found here:

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

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

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

Each line represents a single instance (news item) 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 excess return of one file is more than zero, **-1** otherwise. The features for each news summary are generated from the functions you have written above. Basically, given a feature vector v for each instance, you represent every non-zero component within this vector in SVM format files. Suppose we have $v_k \neq 0$ within 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
```

Here the indexes for one instance are strictly monotone increasing.

6.2 Making the SVM-format files for different feature sets (10 points)

Write another function, `process_corpus(txt_dir, xml_dir, feature_mode)` which uses the provided data and the features you have extracted in Section 1-5. The `txt_dir` and `xml_dir` specify the place where your function loads the news summaries and CoreNLP results. The parameter `feature_mode` should be an integer, and its value specify the group of features to use, as defined below:

```
1 -> Only Lexical
2 -> Only Sentiment words
3 -> Only Named Entity features
4 -> Only Adjectives and Verbs from Part of Speech
5 -> Only Dependency related features
6 -> All features combined
```

Here we give an example of how to generate SVM feature files with this function. Consider the case when we only use lexical features. You need to first create the feature list to be used. Since feature space(list) is always derived from training data, you may want to hard-code the path to the training data: Denote `training_path = '/home1/c/cis530/hw3/data/'`. The feature list is derived from `top_words = extract_top_words(training_path)`. Then for each file F in `txt_dir`, you call `v = unigram_map_entry(F, top_words)` to get the feature vector for this file. The label of this file can be derived by mapping the file name with excess returns in `/home1/c/cis530/hw3/xret_tails.txt` . When you are combining vectors of several feature groups, you can just extend the vector first, then produce the `index:feature` pair.

Suppose you have generated the svm feature files for training, the model file can thus be created using the training data and the svm-train program:

```
svm-train [options] training_svm_feature model_file
```

We suggest you use linear kernel here, which `[options] = -t 0` while training. There are some other parameters in SVM, you can play with it and select according to your preferences. Record the parameters you choose as comments in your script.

NOTE (turnin): For grading, turn in the feature files generated from adjectives and verbs representations, for the news articles in the training set. Name the file `train_4_postags.txt`.

6.3 Predicting Excess Returns based on News Summaries (10 points)

Now that you have obtained the model using Svm-train, we can use the model file to generate predictions on the held-out test data. For testing, use the following command:

```
svm-predict testing_svm_file model_file output_file
```

The file `testing_svm_file` is generated by calling `process_corpus(txt_dir, xml_dir, feature_mode)`. Model file is the one you have generated using `svm-train` command. After executing it, you can see your result in the `output_file`, which includes the excess return predicted given the model and the testing features. The predictions are -1 or 1, corresponding to the class labels we passed on in the input to training the model.

Train a model, then test on the held-out data based on the six groups of features as specified above. Compute Precision (P), Recall (R), and F-Score (F) by comparing the prediction and the actual return on the test set for both Positive (Pos) and Negative (Neg) samples. Report your results with the six values: `Pos-P, Pos-R, Pos-F, Neg-P, Neg-R, Neg-F`, each listed in a single line, separated by space. The performance of different features should be listed in this order: *lexical, sentiment, adjectives+verbs, named entities, dependency, combined*. **Record the result in your python script.** The result should look like this:

```
# 0.6 0.4 0.48 0.571 0.75 0.648  - Lexical (example)
# ......      - Sentiment
# ......      - Adjectives and Verbs from Pos-tagging
# ......      - Named Entities
# ......      - Dependency
# ......      - Combined
```

NOTE: You can expect some of the feature group to perform poorly, i.e. assigning every output the same label. However, there are some features which would give a performance significantly better than random.

7 Use Your Own Features (8 points)

There are definitely some other features which might be useful for predicting the excess return of the news entries. Write functions which would generate the features of your own. You may follow the approach of producing features the way you have done in previous problems. Describe the features you have extracted and the reason why you use these features as comments in the script you submitted. Also record the performance of testing on the held-out test data with your features as comments in your script.

NOTE: You don't need to worry if your features do not perform very well. Select the features which you think would be useful, and justify your choice.