CS-5340/6340, Programming Assignment #1
Due: Friday, September 18, 2020 by 11:59pm

This assignment will give you experience creating machine learning systems for natural language processing. You will be writing programs to produce feature vectors to train a machine learning system for two tasks: (1) Sentiment classification of sentences, and (2) Named Entity classification. The first task involves assigning a label to an entire sentence, while the second task involves assigning a label to individual words within a sentence. Because these tasks are at different levels of granularity, you will need to write two separate programs, although their output will be used to train the same machine learning tool, which we will provide. Each task is detailed below.

---

1 Part A: Sentiment Classifier

Your first task is to train a machine learning system for Sentiment classification. Your classifier should accept sentences from movie reviews as input and label each sentence as either **Positive** or **Negative**. We will provide a software package to produce the machine learning (ML) model, but you will need to write a program that creates feature vector representations for sentences.

You should write a program called `sentiment` that accepts 4 arguments as input: (1) a file of training sentences, (2) a file of test sentences, (3) a file of word features, and (4) an integer *k* indicating the number of features to use. Your program should accept the input as command-line arguments in the following order:

```
<train_file> <test_file> <features_file> <k>
```

For example, if you are using python, then we should be able to run your program like this:

```
python3 sentiment.py train.txt test.txt features.txt 100
```

It is also perfectly fine to use Java. The arguments should similarly be accepted on the command-line in the same order as above.

1.1 Training and Test Files

The training and test files will have the same format. Each training or test instance will be a sentence paired with a gold standard class label. The integer 1 represents a **Positive** class label, and the integer 0 represents a **Negative** class label. Each instance will begin with either a 1 or a 0 indicating its class label, immediately followed by a sentence, with each word on a separate line. All words will be in lower case. One or more blank lines will separate different instances.

For example, a file of sentences might look like this:

```
1
love
this
movie
!

0
it
was
awful.
```

Important Use the words exactly as they are provided. Note that some may be isolated punctuation marks, while others may still have punctuation marks attached (e.g., “Mr.”).

1.2 Word Features File

The file of word features will consist of a list of words, with each word on a separate line. Some of these “words” may be punctuation marks or words with punctuation marks attached. Be sure to use them exactly as they are.

Based on the value of k specified on the command-line, you should extract the **first** k **words** from this file to use as the feature set for your machine learning model.

1.3 Feature Vectors

Your program should produce a k-dimensional feature vector for each sentence in the training and test files. Each position in the vector should correspond to a specific feature. The value for each feature should be binary, using a 1 to indicate that the feature is present in a sentence or a 0 to indicate that the feature is absent. For example, suppose $k = 4$ and the first 4 words in the features file are “movie”, “good”, “bad”, and “!”. We would then create this feature set:

x_1**:** movie
x_2**:** good
x_3**:** bad
x_4**:** !

For each sentence, a feature vector should then be generated using these features. For example, using the 4 features above, the sentence *“this movie is good , really good !”* should produce a feature vector indicating that “movie”, “good”, and “!” are present, but “bad” is absent, as shown below:

$$x_1{=}1,\ x_2{=}1,\ x_3{=}0,\ x_4{=}1$$

IMPORTANT: It is essential that the feature vectors are defined the same way when creating the training and test instances! For example, if x_1 represents the word “movie” when creating the training vectors, then x_1 must also represent the word “movie” when creating the test vectors.

1.4 Output Files

Your program should produce two output files: (1) a file of feature vectors for the training instances, and (2) a file of feature vectors for the test instances. These files will be the input to the ML software, so they need to be formatted **exactly** as explained below or the ML software will reject it.

Each line should represent a feature vector for one instance (sentence). Unlike the input files, there should **not** be any blank lines. The format of each line should be:

```
label feature_id:1 feature_id:1 ...
```

There should be exactly one space between `label` and `feature_id:1` and between each pair of adjacent features.

The ML tool requires that each feature ID be an integer > 0. The number after the colon is the feature value, which will be 0 or 1. For this ML tool, you only need to list features that have a value of 1. All features that are not listed will be given a default value of 0.

IMPORTANT: The ML software requires that the feature ids appear in ASCENDING order! So you will need to sort the feature ids and print them in ascending order.

1.4.1 Examples

For the Positive sentence *"this movie is good !"*, the output should be this:

1 1:1 2:1 4:1

For the Negative sentence *"this bad , bad movie is incredibly bad ."* the output should be this:

0 1:1 3:1

If a sentence does not contain ANY words in the feature set, just output the label (so all features will get the default value 0). For example, the Positive sentence *"so exciting"* should produce this:

1

1.4.2 Naming the Output Files

Each output file should be given the same name as the corresponding training or test file, but with `.vector` appended at the end. For example, given the command:

```
python3 sentiment.py train.txt test.txt features.txt 100
```

Your program should produce two files named:

```
train.txt.vector
test.txt.vector
```

2 Part B: Named Entity Classifier

Your second task is to train a machine learning system for Named Entity (NE) classification. A *Named Entity* is a proper name corresponding to a semantic category. Your classifier should recognize 3 types of Named Entities: Persons (PER), Locations (LOC), and Organizations (ORG).

Your classifier will use a **BIO** labeling scheme. A **"B"** indicates the **B**eginning of an entity's name, and **"I"** indicates the **I**nside of an entity's name. The **"O"** label represents **O**ther (i.e., not part of an entity's name). Since your system will identify 3 types of entities, we will need a **"B"** and an **"I"** label for each type. So the ML system will use 7 labels: **B-PER**, **I-PER**, **B-LOC**, **I-LOC**, **B-ORG**, **I-ORG**, and **O**. For example, the sentence *"Ellen Riloff teaches in Utah"* would be labeled as: *Ellen/B-PER Riloff/I-PER teaches/O in/O Utah/B-LOC*.

You should write a program called `entities` that accepts 3 arguments: (1) a file of training sentences, (2) a file of test sentences, (3) a sequence of *ftype* arguments indicating which types of features will be generated. Your program should recognize 6 possible *ftype* arguments: WORD POS ABBR CAP WORDCON POSCON. There will always be one *ftype*, WORD, which will appear first and may be followed by up to 5 additional *ftypes*, in any order! Your program should accept the command-line arguments as follows:

```
python3 entities.py <train_file> <test_file> WORD [... <ftype_n >]
```

We should be able to run your program with any combination of *ftype* arguments, for example:

```
python3 entities.py train.txt test.txt WORD
python3 entities.py train.txt test.txt WORD CAP POS
python3 entities.py train.txt test.txt WORD POSCON POS WORDCON ABBR CAP
```

It is also perfectly fine to use Java. The arguments should similarly be accepted on the command-line in the same order as above.

2.1 Input Files

The training and test files will have the same format. Each file will contain sentences with a class label and a part-of-speech (POS) tag assigned to each word. Each word will be one instance for the ML classifier. The information for each word will be on a separate line, formatted as: `label POS word`. One or more blank lines will separate sentences. An input file might look like this:

B-LOC	NNP	Israel
O	NN	television
O	VBD	rejected
O	DT	the
O	NN	skit
O	IN	by
O	DT	the
O	NN	comedian
B-PER	NNP	Tuvia
I-PER	NNP	Tzafir

2.2 Feature Types

Your program should be able to produce 4 types of features for a word w, corresponding to the *ftype* arguments:

WORD: the word w itself
POS: the POS tag p of w
ABBR: a binary feature indicating whether w is an abbreviation. An abbreviation must: (1) end with a period, (2) consist entirely of alphabetic characters [a-z][A-Z] and one or more periods, and (3) have length ≤ 4
CAP: a binary feature indicating whether the first letter of w is capitalized (including words in all caps such as "IBM").

CS-6340 Students: In addition to the features above, your program will need to produce 2 contextual types of features for a word w:

WORDCON: the previous word w_{-1} and the following word w_{+1}
POSCON: the previous POS tag p_{-1} and the following POS tag p_{+1}

There are a few cases where you will also need to define pseudo-words or pseudo-POS tags as feature values. Please use the following conventions:

Pseudo	**Description**
UNK	pseudo-word for words in the test but not the training data
UNKPOS	pseudo-POS for POS tags in the test but not the training data
PHI	pseudo-word for the beginning-of-sentence position
PHIPOS	pseudo-POS for the beginning-of-sentence position
OMEGA	pseudo-word for the end-of-sentence position
OMEGAPOS	pseudo-POS for the end-of-sentence position

The UNK pseudo-word should be used for words that appear in the test file but not in the training file. The UNKPOS pseudo-tag should be used for part-of-speech tags that appear in the test file but not in the training file.

The PHI, PHIPOS, OMEGA, and OMEGAPOS pseudo-words only need to be defined for the **WORDCON** and **POSCON** contextual features. They are needed when the current word is the first word of a sentence (so there is no preceding word) or when the current word is the last word of a sentence (so there is no following word). **Important:** Do not cross sentence boundaries when creating the contextual features! The PHI and OMEGA pseudo-words should never be the current word, they are only placeholders for the beginning and end of a sentence.

2.3 Output Files

Your `entities` program should produce 2 types of output files:

1. Generate human readable features for each training instance and write these features to a new output file in the same directory as your `entities` program. Give this file the same name but append the extension `.readable`. Do the same thing for the test file.

2. Generate a feature vector for each training instance and write these feature vectors to a new output file in the same directory as your `entities` program. Give this output file the same name but append the extension `.vector`. Do the same thing for the test file.

For example, given the command:

```
python3 entities.py train.txt test.txt WORD
```

Your program should produce four output files named:

```
train.txt.readable
test.txt.readable
train.txt.vector
test.txt.vector
```

2.3.1 Readable Output Files

In the Readable features files, each feature should be printed as follows:

WORD: w
POS: p
ABBR: yes or no
CAP: yes or no
WORDCON: w_{-1} w_{+1} (with exactly one space between w_{-1} and w_{+1})
POSCON: p_{-1} p_{+1} (with exactly one space between p_{-1} and p_{+1})

Everyone's program should print information for **all** 6 features, even if some of the features are not used. Print a blank line separating the features for different words. If an *ftype* was not specified in the arguments, then print "n/a" (for "not applicable") as its value.

As an example, consider the word "Israel" from the sample input file shown earlier. Given the *ftype* arguments **WORD CAP**, you should generate the following output:

WORD: Israel
POS: n/a
ABBR: n/a
CAP: yes
WORDCON: n/a
POSCON: n/a

Given the *ftype* arguments **WORD CAP WORDCON POSCON**, if "television" occurs in the test data but **not** in the training data, then the output should be:

WORD: Israel
POS: n/a
ABBR: n/a
CAP: yes
WORDCON: PHI UNK
POSCON: PHIPOS NN

2.3.2 Feature Vector Output Files

The Feature Vector files will serve as the input to the ML software, so they need to be formatted exactly as explained. Each line should represent a feature vector for one instance (word). Unlike the input files, there should **not** be any blank lines. The format of each line should be:

```
label feature_id:1 feature_id:1 ...
```

Put exactly one space between `label` and `feature_id:1` and between each pair of features. The ML classifier requires that class labels are integers, so use the following numeric ids for the labels:

0 for *O*
1 for *B-PER*
2 for *I-PER*
3 for *B-LOC*
4 for *I-LOC*
5 for *B-ORG*
6 for *I-ORG*

The ML tool also requires that each feature ID be an integer > 0, so your program will need to assign a numeric ID to each feature. The number after the colon is the feature value, which will always be 0 or 1. For this ML tool, you only need to list features that have a value of 1. All features that are not listed will be given a default value of 0. **Important:** The ML software requires that the feature ids appear in ASCENDING order! So you will need to sort the feature ids and print them in ascending order.

Important: The same feature ids must be used for both the training and test instances! For example, if the feature ABBR is assigned the ID 22, then 22 should be used for ABBR throughout ALL training and test instances.

2.3.3 How to Create Binary Features for String Values in the Vector Files

The ML tool requires binary feature values. The **ABBR** and **CAP** features are binary, so creating features for them is easy: just define a unique feature ID and use the value 1 for Yes and 0 for No.

But the **WORD, POS, WORDCON** and **POSCON** features take string values, so you may be wondering how to translate a string feature into a binary feature. The trick is to first generate the set of all possible string values in the training data. For the **word** features, this is the set of all distinct words plus an UNK value (for words in the test set that did not appear in the training set). For the **pos** features, this is the set of all distinct POS tags plus an UNKPOS value. If a string-valued feature has k possible values, then you should generate k binary features with exactly one of the features having a value of 1 and all other features having a value of 0.

IMPORTANT: For the **WORD** and **WORDCON** features, treat all words as being case sensitive. So for example, the words "apple" and "Apple" should be considered as different words.

Here is an example. Suppose we have a training set containing only two sentences (POS tags appear after the slash). Note that the POS tagger uses a period (.) as the POS tag for end-of-sentence tokens such as periods, exclamation points, and question marks.

S1: John/NNP saw/VBD Mary/NNP ./.
S2: Mary/NNP waved/VBD !/.

There are 6 distinct words, so we would create the 7 binary features shown below:

x_1**:** John
x_2**:** Mary
x_3**:** saw
x_4**:** waved
x_5**:** .
x_6**:** !
x_7**:** UNK

Given "John", we would produce the following **WORD** feature values:
x_1=1, x_2=0, x_3=0, x_4=0, x_5=0, x_6=0, x_7=0

Given "saw", we would produce the following **WORD** feature values:
x_1=0, x_2=0, x_3=1, x_4=0, x_5=0, x_6=0, x_7=0

For the **POS** feature, there are 3 distinct POS tags, so we should create 4 binary features:

y_1**:** NNP
y_2**:** VBD
y_3**:** .
y_4**:** UNKPOS

Given "John", we would produce the following **POS** feature values:
y_1=1, y_2=0, y_3=0, y_4=0

Given "saw", we would produce the following **POS** feature values:
y_1=0, y_2=1, y_3=0, y_4=0

Important: For the **WORDCON** and **POSCON** features, you'll also need to include values for PHI and OMEGA, or PHIPOS and OMEGAPOS, respectively. And you'll need to create separate sets of binary features for the previous word and following word. Essentially, you'll be creating 4 sets of binary features (i.e., one set for each of w_{-1}, w_{+1}, p_{-1}, p_{+1}).

2.3.4 Feature Vector Examples

Here are illustrations of each step using the sample input file shown earlier as training data. If the *ftype* arguments are **WORD POS POSCON CAP**, first you would generate all possible **WORD** features and assign a unique numeric identifier to each one. There are 9 distinct words in the example, so you must create a feature for word and create one feature for the UNK pseudo-word. Next, you should generate all possible **POS** features. There are 5 distinct part-of-speech tags, so you must create a feature for each tag and create one feature for the UNKPOS pseudo-tag.

For **POSCON**, you will also need to create features for the POS tags in the previous position("prev-pos-*p*") and following position("next-pos-*p*"). Remember that the **WORDCON** and **POSCON** features need to include values for the pseudo-words as well. Finally, for the **CAP** feature, you also need one binary capitalization feature. The full feature set is:

ID	Feature	ID	Feature	ID	Feature
1	word-Israel	2	word-television	3	word-rejected
4	word-the	5	word-skit	6	word-by
7	word-comedian	8	word-Tuvia	9	word-Tzafir
10	word-UNK	11	prev-pos-PHIPOS	12	next-pos-OMEGAPOS
13	pos-NNP	14	prev-pos-NNP	15	next-pos-NNP
16	pos-NN	17	prev-pos-NN	18	next-pos-NN
19	pos-VBD	20	prev-pos-VBD	21	next-pos-VBD
22	pos-DT	23	prev-pos-DT	24	next-pos-DT
25	pos-IN	26	prev-pos-IN	27	next-pos-IN
28	pos-UNKPOS	29	prev-pos-UNKPOS	30	next-pos-UNKPOS
31	capitalized				

Using the features and identifiers above, you would then generate the following feature vectors. Remember that the feature ids must be sorted in ascending order for the classifier!

Word	Label & Feature Vector
Israel	3 1:1 11:1 13:1 18:1 31:1
television	0 2:1 14:1 16:1 21:1
rejected	0 3:1 17:1 19:1 24:1
the	0 4:1 18:1 20:1 22:1
skit	0 5:1 16:1 23:1 27:1
by	0 6:1 17:1 24:1 25:1
the	0 4:1 18:1 22:1 26:1
comedian	0 7:1 15:1 16:1 23:1
Tuvia	1 8:1 13:1 15:1 17:1 31:1
Tzafir	2 9:1 12:1 13:1 14:1 31:1

3 The Machine Learning Tool

Installation: The machine learning software is available in the Files folder for Program #1 on Canvas as: `liblinear-1.93.tar.gz`. To install it, first unpackage it (`tar xvfz liblinear-1.93.tar.gz`). In the directory, type: `make`. This will produce two executable files: `train` and `predict`.

Training: to train a classification model (classifier), invoke the command:

```
train -s 0 -e 0.0001 <train_data> <classifier_name>
```

Note: the option "-s 0" indicates that you are using the logistic regression machine learning algorithm. The `train_data` argument is the name of the file containing the feature vectors for the training data. The learned classifier will be saved as a file named `classifier_name`.

Testing: to apply a learned classification model (classifier), invoke the command:

```
predict <test_data> <classifier_name> <predictions_file> > <accuracy_file>
```

The `test_data` file should contain the feature vectors for the test data, and the `classifier_name` should be the file name of the trained classification model. The classifier's predicted labels will be saved as `predictions_file` and the classifier's accuracy results will be saved as `accuracy_file`.

Together, you can use the following commands to train a classifier with training data and apply that classifier to test data:

```
train -s 0 -e 0.0001 train.txt.vector MyClassifier
predict test.txt.vector MyClassifier predictions.txt > accuracy.txt
```

Sentiment Classification Example

When your `sentiment` program is finished, you can train and test a ML sentiment classifier with these commands:

1. Generate the feature vectors for the training and test sentences:
   ```
   python3 sentiment.py train.txt test.txt words.txt 100
   ```
2. Train a ML classifier using the training instances:
   ```
   train -s 0 -e 0.0001 train.txt.vector SentClassifier
   ```
3. Use the ML classifier to label the test instances:
   ```
   predict test.txt.vector SentClassifier predictions.txt > accuracy.txt
   ```

Named Entity Classification Example

When your `entities` program is finished, you can train and test a ML entity classifier with these commands:

1. Generate the feature vectors for the training and test sentences:
   ```
   python3 entities.py train.txt test.txt WORD
   ```
2. Train a ML classifier using the training instances:
   ```
   train -s 0 -e 0.0001 train.txt.vector EntityClassifier
   ```
3. Use the ML classifier to label the test instances:
   ```
   predict test.txt.vector EntityClassifier predictions.txt > accuracy.txt
   ```

SUBMISSION INSTRUCTIONS

On CANVAS, please submit an archived (.tar) and/or zipped (.gz or .zip) file containing:

1. The source code files for your `sentiment` and `entities` programs. Be sure to include all files that we will need to compile and run your programs!

2. A README file that includes the following information:
   - how to compile and run your code
   - which CADE machine you tested your program on (this info may be useful to us if we have trouble running your program)
   - any known bugs, problems, or limitations of your program

3. Run your sentiment program on the **CADE** machines using the input files in the Program #1 Sentiment folder on CANVAS and set $k = 100$. For example:

   ```
   python3 sentiment.py trainS.txt testS.txt features.txt 100
   ```

   Then use the ML software to train an ML classifier with the trainS.txt.vector file produced by your program and apply the resulting classifier to make predictions on the testS.txt.vector file. Then submit the following 4 files:

   (a) trainS.txt.vector
   (b) testS.txt.vector
   (c) predictions.txt
   (d) accuracy.txt

4. Run your entities program on the **CADE** machines using the input files in the Program #1 Entities folder on CANVAS. For example:

   ```
   python3 entities.py trainE.txt testE.txt WORD CAP
   ```

   Then use the ML software to train an ML classifier with the trainE.txt.vector file produced by your program and apply the resulting classifier to make predictions on the testE.txt.vector file. Then submit the following 6 files:

   (a) trainE.txt.readable
   (b) testE.txt.readable
   (c) trainE.txt.vector
   (d) testE.txt.vector
   (e) predictions.txt
   (f) accuracy.txt

Important: Your program must be written in Python or Java and it **MUST** compile and run on the linux-based CADE machines! We will not grade programs that cannot be run on the linux-based CADE machines.

Your program will be graded based on new data files! **So please test your program thoroughly to evaluate the generality and correctness of your code!** Even if your program works perfectly on the examples that we give you, that does not guarantee that it will work perfectly on new files.

HELPFUL HINTS

TAR FILES: First, put all of the files that you want to submit in a directory named "program1". Then from the parent directory where the "program1" folder resides, issue the following command:

```
tar cvfz program1.tar.gz program1/
```

This will put everything that is inside the "program1/" directory into a single file called "program1.tar.gz". This file will be "archived" to preserve structure (e.g., subdirectories) inside the "program1/" directory and then compressed with "gzip".

FYI, to unpack the gzipped tarball, move the program1.tar.gz to a new location and issue the command:

```
tar xvfz program1.tar.gz
```

This will create a new directory called "program1" and restore the original structure and contents.

For more general information on the "tar" command, this web site may be useful:
https://www.howtogeek.com/248780/how-to-compress-and-extract-files-using-the-tar-command-on-linux/