Computer Science 401 — 11 January 2016
St. George Campus — University of Toronto

Homework Assignment #1
Due: Friday, 12 February 2016 at 19h00 (7pm)

Sentiment analysis with tweets

TAs: Stefania Raimondo (c3raimon@cdf.toronto.edu) and Erin Grant (t5grante@cdf)

Introduction

This assignment will give you experience with web corpora (i.e., a collection of tweets from Twitter), Python programming, part-of-speech (PoS) tags, machine learning with WEKA, and IBM Watson on BlueMix.

Your task is to split the tweets into sentences, tag them with a PoS tagger that we will provide, gather some feature information from each tweet, learn models, and use these to classify topics and sentiments. Sentiment analysis is an important, still emerging topic in computational linguistics in which we try to identify subjective information. This can range from binary opinions in social media, as in this assignment (which can be useful in marketing), to a spectrum of cognitive behaviours, to help diagnoses in health care.

You should check the course bulletin board and @SPOClab frequently for announcements and discussion pertaining to this assignment.

Tweet Corpus

Founded in 2006, Twitter (`www.twitter.com`) now provides one of the most popular social media services. In this assignment, we will use the Sentiment140 corpus (`http://help.sentiment140.com/for-students`), which was accumulated automatically, using the Twitter Search API. It was also *annotateed* automatically, assuming that any tweet with positive emoticons, like :), were positive, and tweets with negative emoticons, like :(, were negative[1] . These data are split into two files on the CDF servers under `/u/cs401/A1/tweets/`:

- `training.1600000.processed.noemoticon.csv`: 1,600,000 tweets, used for **training**.
- `testdata.manualSUBSET.2009.06.14.csv`: 359 tweets, used for **testing**.

To save space in your CDF account, these files should *only* be accessed from that directory.

Each line of these files is a single tweet, which may contain multiple sentences despite their brevity. The tweets have been collected directly from the web, so they may contain html tags, hashtags (e.g., #NLCAss1), and user tags (e.g., @user). The comma-separated fields of each line are:

0 the polarity of the tweet (0 = negative emotion, 4 = positive emotion)
1 the id of the tweet (e.g., 2087)
2 the date of the tweet (e.g., Sat May 16 23:58:44 UTC 2009)
3 the query (e.g., lyx). If there is no query, then this value is NO_QUERY.
4 the user that tweeted (e.g., robotickilldozr)
5 the text of the tweet (e.g., Lyx is cool)

You evaluate your system with *all* the test data. You train your system with 11,000 lines from the training data. Specifically, concatenate lines $[GID \times 5500 \ldots (GID + 1) \times 5500 - 1]$ (class 0) and lines $800,000 + [GID \times 5500 \ldots (GID + 1) \times 5500 - 1]$ (class 4), where GID is your group ID.

[1] See details at `http://cs.stanford.edu/people/alecmgo/papers/TwitterDistantSupervision09.pdf`

Your tasks

1 Pre-processing, tokenizing, and tagging [25 marks]

The tweets, as given, are not in a form amenable to feature extraction for classification – there is too much 'noise'. Therefore, the first step is to write a Python program named *twtt.py* (tweet tokenize and tag), in accordance with the "General specifications" section below, that will take a tweet file in its provided form and convert it to a normalized form where:

1. All html tags and attributes (i.e., `/<[^>]+>/`) are removed.
2. Html character codes (i.e., `&...;`) are replaced with an ASCII equivalent.
   See `http://www.asciitable.com`.
3. All URLs (i.e., tokens beginning with `http` or `www`) are removed.
4. The first character in Twitter user names (@) and hash tags (#) are removed.
5. Each sentence within a tweet is on its own line.

   - This will require detecting end-of-sentence punctuation. Some punctuation does not end a sentence; see standard abbreviations here: `/u/cs401/Wordlists/abbrev.english`. It can be difficult to detect when an abbreviation ends a sentence; e.g., in *Go to St. John's St. I'll be there.*, the first period is used in an abbreviation, the last period ends a sentence, and the second period is used both in an abbreviation and an end-of-sentence. You are not expected to write a 'perfect' pre-processor here (none exists!), but merely to use your best judgment in writing heuristics; see section 4.2.4 of the Manning and Schütze text for ideas.

6. Ellipsis (i.e., '...'), and other kinds of multiple punctuation (e.g., '!!!') are *not* split.
7. Each token, including punctuation and clitics, is separated by spaces.

   - Clitics are contracted forms of words, such as *n't*, that are concatenated with the previous word.
   - Note: the possessive *'s* has its own tag and is distinct from the clitic *'s*, but nonetheless must be separated by a space; likewise, the possessive on plurals must be separated (e.g., *dogs '*).

8. Each token is tagged with its part-of-speech.

   - A tagged token consists of a word, the '/' symbol, and the tag (e.g., dog/NN). See below for information on how to use the tagging module. The tagger can make mistakes.

9. Before each tweet is demarcation '<A=#>', which occurs on its own line, where # is the numeric class of the tweet (0, 2, or 4).

   - We will later need to differentiate between sentences in a tweet, hence the explicit demarcation.
   - This demarcation refers to the tweet below it, so the first line in the output should be such a demarcation.

Note that:

- If a tweet is empty after pre-processing, it should be preserved as such.
- Periods in abbreviations (e.g., e.g.) are not split off from the rest of their tokens. Therefore, *e.g.* stays as the token *e.g.*.
- You can handle single hyphens (-) between words as you please. E.g., you can split *non-committal* into three tokens or leave it as one.

The *twtt.py* program takes three arguments: the input `.csv` filename, your group number, and the output filename. For example, if you are in group 42 and want to create the `train.twt` from the train data, you would run:

```
python twtt.py /u/cs401/A1/tweets/training.1600000.processed.noemoticon.csv 42 train.twt
```

It would be advisable to perform these normalization tasks in the order listed (e.g., you should tag tokens only after punctuation and clitics have been isolated). For preprocessing and tokenization, you may only use standard Python libraries. For tagging, we are providing a tagger for you to use in */u/cs401/A1/tagger/*. Copy the contents of that directory to your working directory and invoke *NLPlib.py* in *twtt.py*. For example,

```
import NLPlib

tagger = NLPlib.NLPlib()
sent = ['tag', 'me']
tags = tagger.tag(sent)
```

The `tag` method takes an array of strings (i.e., the tokens in the order that they appear in the sentence) and returns an ordered list of PoS tags (one for each token).

Table 2 shows an example of an original tweet and its normalized version.

Run *twtt.py* after debugging it on both the training and testing `csv` files separately. **Save the normalized test data in the file test.twt – you will have to submit this.** The output of *twtt.py* will be used in Task 2.

2 Gathering feature information [20 marks]

The second step is to write a Python program named *buildarff.py*, in accordance with the "General specifications" section below, that takes tokenized and tagged tweets from Task 1 and builds an 'arff' datafile that will be used to classify tweets in Task 3. Feature extraction is basically the process of analyzing the preprocessed data in terms of variables that are indicative or discriminative of the classes. For example, if the use of the word *me* is indicative of a negative affect, then the number of first-person pronouns in a tweet will tell you something about the mood of the tweet.

The arff file consists of two main sections: the definition of features (attributes) and the instances (data points). Attributes include a name and a type; for this assignment, we will only use numeric types. Each line in the data section represents a single tweet encoded by a list of numbers (i.e., the features) separated by commas, and the class (affect) of the tweet, which is the last entry, e.g., in

```
0,1,3,0,0..., 4
```

the final 4 refers to the positive class. The feature values must appear in the same order in which they are defined in the attribute definition section. An example arff file is shown in Table 5. For details, see the online specification at `http://www.cs.waikato.ac.nz/ml/weka/arff.html`.

The *buildarff.py* program can take two or three arguments: **1)** the input filename (i.e., the output of *twtt.py*, **2)** the filename of the output *arff* file, and **(optional) 3)** the maximum number of tweets from **each** class that will be used to build the arff; all tweets are used if this argument is missing. For example:

```
buildarff.py train.twt train.arff
```

produces an arff file with as many data points as tweets in *train.twt*, and

```
buildarff.py test.twt some.arff 50
```

produces an arff file with at most 100 data points (50 for each class). Note: your mark here will depend partially on your implementation. To get full marks, you must implement the feature extraction in a modular way which allows for easy addition of new features with minimal modification.

The features to be gathered

For each tweet, you need to extract 20 features and write these, along with the class, to the arff file. These features are listed in Table 3. Many of these involve counting tokens in a tweet that have certain characteristics that can be discerned from its tag. For example, counting the number of adverbs in a tweet involves counting the number of tokens that have been tagged as RB, RBR, or RBS. Table 4 explicitly defines some of the features in Table 3; these definitions are available on CDF in the directory */u/cs401/Wordlists/*. You may copy and modify these files, but do not change their filenames. Be careful about capitalization; in all cases you should count both capitalized and lower case forms (e.g., both *he* and *He* count towards the number of third person pronouns).

When your feature extractor works to your satisfaction, build an arff file, *all.arff*, from all of the training data. **Also build a test.arff from the testing data – you will need to submit this.**

3 Classifying tweets using WEKA [30 marks]

The third step is to use the feature extraction program from Task 2 to classify tweets using the WEKA machine learning package. To use this package, you will invoke WEKA by indicating the data, classifier, and other options on the command line, as described in Appendix 1.

We will now experiment a little.

3.1 Classifiers

Build two arff files: one for all of the training and one for all of the testing data. Report classification accuracy on test data among support vector machines (SVMs), naïve Bayes, and decision trees. Pipe the output of WEKA for the best classifier to the file *3.1output.txt*. *Use the best classification algorithm for the following questions unless instructed otherwise.*

3.2 Amount of training data

Many researchers attribute the success of modern machine learning to the sheer volume of data that is now available. Modify the amount of data that is used to train your system in increments of 500, starting at 500 (i.e., $n = 500, 1000, ..., 5500$) and report the resulting accuracies in a table in *3.2output.txt*. In that file, also comment on the changes to accuracy as the number of training samples increases, including at least two sentences on a possible explanation.

3.3 Feature analysis

For each of $n = 500$ and $n = 5500$ in Section 3.2, run WEKA's information gain attribute selector and copy the output of WEKA into the file *3.3output.txt*. What features, if any, retain their importance at both low and high(er) amounts of input data? Provide a possible explanation as to why this might be.

3.4 Cross-validation

Many papers in machine learning stick with a single data set for training and another for testing (occasionally with a third for validation). This may not be the most honest or generalizable approach. Is the best method from subtask 3.1 *really* the best? For each of the classifiers in subtask 3.1, run 10-fold cross-validation on the arff file containing all 11,000 training tweets. Unfortunately, WEKA does not store results from each fold, so you will have to implement this yourself. Split the 11,000 training instances from the arff file into 10 partitions, d_i ($i = 1..10$), each with an equal number of positive and negative examples. Then,

> **for** i=1..10,
> Concatenate all other partitions, $d_{j \neq i}$, together in a new arff and use for training.
> Obtain accuracy, precision, and recall on partition d_i, in another arff.

See Appendix 2 for a description of accuracy, precision, and recall. Report all results in *3.4output.txt*. Next, for each pair of classifiers (i.e., 3 pairs), check if any one system is significantly different than another. I.e., given vectors `a` and `b`, one for each classifier, containing the accuracy values for each of the respective 10 folds, report the p-value in the `S` output below.

```
from scipy import stats
S = stats.ttest_rel(a, b)
```

4 IBM Watson on BlueMix [10 marks]

To ease into Watson, we will simply replicate subtask 3.2, above, using IBM BlueMix. Write a script, *ibmTrain.py*, using our template at `/u/cs401/A1/`, with basic error handling, to do the following:

1. Convert the original training `.csv` file into the two-field format described here:

   `https://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/doc/nl-classifier/data_format.shtml`

   Be sure to remove *all* double-quotes.

2. Save 11 subsets of this format in files called `ibmTrain#.csv`, where # is the number of training samples included, using the same subsets as subtask 3.2.

3. For each value of # (note it goes in 2 places), using your `username` and `password` from BlueMix, run the *equivalent* of:

```
curl -u "{username}":"{password}" -F training_data=@ibmTrain#.csv
 -F training_metadata="{\"language\":\"en\",\"name\":\"Classifier #\"}"
 "https://gateway.watsonplatform.net/natural-language-classifier/api/v1/classifiers"
```

Now complete *ibmTest.py*, with basic error handling, using our template in `/u/cs401/A1/`, to do this:

1. Ensure that *all* 11 classifiers are ready for testing (it can take *hours* for them to be trained). E.g.,

```
curl -u "{username}":"{password}"
 "https://gateway.watsonplatform.net/natural-language-classifier/api/v1/classifiers"
```

   lists the classifiers, and you can query their status with the *equivalent* of:

```
curl -u "{username}":"{password}"
 "https://gateway.watsonplatform.net/natural-language-classifier\
 /api/v1/classifiers/FUBAR"
```

   where `FUBAR` is the value in one of the `classifier_id` fields returned by the previous command.

2. Once all classifiers report the status `Available`, classify each tweet in the *test* data with each classifier, equivalent to taking the value of `top_class` from the output of:

```
curl -G -u "{username}":"{password}"
 "https://gateway.watsonplatform.net/natural-language-classifier\
  /api/v1/classifiers/FUBAR/classify?text=TEXT"
```

   where `TEXT` is the test tweet itself, with punctuation encoded in UTF-8, e.g., ' ' becomes '%20'. See `http://www.w3schools.com/tags/ref_urlencode.asp` for a list of codes.

3. Compute accuracy for each classifier. That is, for each classification returned by each classifier, compare its value with the known class in the test data. For each classifier, compute the proportion of tweets that classified *correctly* and report these values in the file *4output.txt*. *You should manually add to that file a brief discussion on how these results differ from those in section 3.2.*

4. Compute the *average* `confidence` of the selected class over all correct and incorrect classifications, for each classifier. I.e., there should be 22 average confidence values – 11 classifiers by two classification outcomes. You can also add any observations you find interesting to *4output.txt*, if you wish.

Note: You might have some problems with `curl` in `tcsh` on CDF, so run `bash` instead.
Note: Additional information can be found here:
`https://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/doc/nl-classifier/`
and here:
`https://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/natural-language-classifier/api/v1/#introduction`

5 Bonus [15 marks]

We will give up to 15 bonus marks for innovative work going substantially beyond the minimal requirements. These marks can make up for marks lost in other sections of the assignment, but your overall mark for this assignment cannot exceed 100%. The obtainable bonus marks will depend on the complexity of the undertaking, and are at the discretion of the marker. Imporantly, your bonus work should not affect our ability to mark the main body of the assignment in any way.

You may decide to pursue any number of tasks of your own design related to this assignment, although you should consult with the instructor or the TA before embarking on such exploration. Certainly, the rest of the assignment takes higher priority. Some ideas:

- **Identify words that the PoS tagger tags incorrectly** and add code that fixes those mistakes. Does this code introduce new errors elsewhere? E.g., if you always tag *dog* as a noun to correct a mistake, you will encounter errors when *dog* should be a verb. How can you mitigate such errors?
- Explore **alternative features** to those extracted in Task 2. What other kinds of variables would be useful in distinguishing affect? Test your features empirically as you did in Task 3 and discuss your findings.
- Explore **alternative classification methods** to those used in Task 3. Explore different parameters that control those methods. Which parameters give the best empirical performance, and why? It may help to explore the WEKA GUI for this task, as described in Appendix 1.
- **Topics** are provided in field 3 of each `csv`, as described in the introduction (or at least the queries are). Is there an effect of topic on the accuracy of the system? E.g., is it easier to tell how someone feels about Barack Obama or about LaTeX? People or companies? As there may be class imbalances in the groups, how would you go about evaluating this? Go about evaluating this.
- If the license permits it, play around with **AlchemyAPI** on BlueMix to unlock additional functionality for sentiment analysis.

6 General specifications

As part of grading your assignment, the grader may run your programs and/or arff files on test data and configurations that you have not previously seen. This may be partially done automatically by scripts. It is therefore important that each of your programs precisely meets all the specifications, including its name and the names of the files that it uses. **A program that cannot be evaluated because it varies from specifications will receive zero marks.**

If a program uses a file, such as the word lists, whose name is specified within the program, the file must be read either from the directory in which the program is being executed, or from a subdirectory of */u/cs401* whose path is completely specified in the program. Do **not** hardwire the absolute address of your home directory within the program; the grader does not have access to this directory.

All your programs must contain adequate internal documentation to be clear to the graders.

7 Submission requirements

This assignment is submitted electronically. You should submit:

1. All your code for *twtt.py* and *buildarff.py* (including helper scripts, if any).
2. The tagged and tokenized file *test.twt* from Task 1. **Do not submit other twt files.**
3. The *test.arff* file from Task 2. **Do not submit other arff files.**
4. *3.1output.txt*: Report on classifiers.
5. *3.2output.txt*: Report on the amount of training data.
6. *3.3output.txt*: Report on feature analysis.
7. *3.4output.txt*: Report on 10-fold cross-validation.
8. Any lists of words that you modified from the original version.

In another file called *ID* (use the template on the course web page), provide the following information:

1. your first and last name.
2. your student number.
3. your CDF login id.
4. your preferred contact email address.
5. the BlueMix module name and credentials (username and password) used by your programs
6. whether you are an undergraduate or graduate.
7. this statement: *By submitting this file, I declare that my electronic submission is my own work, and is in accordance with the University of Toronto Code of Behaviour on Academic Matters and the Code of Student Conduct, as well as the collaboration policies of this course.*

You do not need to hand in any files other than those specified above. The electronic submission must be made from CDF with the *submit* command:

`submit -N a1 csc401h` *filename filename ...*

Do **not** `tar` or `compress` your files, and do not place your files in subdirectories. Do not format your discussion as a PDF or Word document — plain text only.

8 Working outside the lab

If you want to do some or all of this assignment on your laptop or home computer, for example, you will have to do the extra work of downloading and installing the requisite software and data. You take on the risk that your computer might not be adequate for the task. You are strongly advised to upload regular backups of your work to CDF, so that if your home machine fails or proves to be inadequate, you can immediately continue working on the assignment at CDF. When you have completed the assignment, you should try your programs out on CDF to make sure that they run correctly there. **A submission that does not work on CDF will get zero marks.**

The course web page, *http://www.cs.toronto.edu/∼csc401h*, will offer links to versions of Python for various systems, the WEKA software suite, a gzipped file of the tweet corpus, and the lists of word-category definitions. WEKA can be run on any system with a recent Java interpreter.

Appendix 1: Using WEKA

WEKA is available on CDF in */u/cs401/WEKA*.

1. Classifier command line

The general syntax for running WEKA from the command line is:

`java -cp` **weka.jar path* *weka classifier* *classifier options**

The path to the jar file on CDF is: `/u/cs401/WEKA/weka.jar`. The three classifiers you will use are:

weka.classifiers.functions.SMO (for SVMs)
weka.classifiers.bayes.NaiveBayes (for naïve Bayes)
weka.classifiers.trees.J48 (for decision trees)

The key options are:

-t *arff file* (Choose the training file)
-*x* *X* (Use cross-validation, X is the number of folds; this is not useful here)
-T *arff file* (Choose the testing file, for section 3.2)
-no-cv (Do not perform cross-validation)
-o (Do not output the classifier. Use this option for your outputs!)

A full list of options are available at `http://www.cs.waikato.ac.nz/~remco/weka_bn/node13.html`, but this is all you need to complete this assignment.

2. Information gain attribute selector

The options for running WEKA's information gain attribute selector (Section 3.5) are different than the classifiers and a bit more complex; ideally, attribute selection is done through the WEKA GUI (and you may do so, see below). Since there is no variation in the parameters, however, we can simply provide you with a sh script that will do the job. To run it on one of your arff files and send the output to a file, use this command:

`sh /u/cs401/WEKA/infogain.sh` **arff file** `>` **output file**

3. WEKA GUI

WEKA includes an excellent GUI for quickly testing various options associated with classification. Since you need to submit correct command line arguments in your discussion, we do not recommend that you use the GUI for producing your final output; however, you may use the GUI for early testing and additional exploration as part of the bonus. If you are on a CDF lab computer or logged in using NX, the GUI is available using the following command:

```
java -jar /u/cs401/WEKA/weka.jar
```

Select the explorer, and open your arff file using the open file button. Then go to the Classify tab to carry out a classification experiment, or Select Attributes to do feature selection. The options which can be selected otherwise correspond to those available at the command line. See
`http://www.cs.waikato.ac.nz/ml/weka/index_documentation.html` for more information.

Appendix 2: Accuracy, precision, and recall

WEKA does not compute precision and recall for you, but it does give you a 'confusion table', C, summarizing how the tweets in the test data were classified. These figures allow you to compute accuracy, precision and recall yourself. Here is an example table:

```
 (a)  (b)  (c)  (d)      <- classified as
---- ---- ---- ----
 132    6    9    4      (a): class I
   2   75    1    6      (b): class II
   8    9  119   12      (c): class III
   8    3    4   97      (d): class IV
```

An element at row i, column j (i.e., $c_{i,j}$) is the number of instances belonging to class i that were classified as class j. For example, here 75 tweets belonging to class II were correctly classified, but 1 tweet belonging to class II was mistakenly classified as class III.

Here, **accuracy**, A is the overall measure of the quality of the classification. Specifically, it is the total number of correctly classified instances over all classifications, or

$$A = \frac{\sum_i c_{i,i}}{\sum_{i,j} c_{i,j}}.$$

Precision and recall have to be measured separately for each of the classes. For class κ, **precision** $Prec$ is the fraction of cases classified as κ that truly are κ. In other words, precision is, for each column, the ratio of the diagonal element to the sum of the column,

$$Prec(j) = \frac{c_{j,j}}{\sum_i c_{i,j}}.$$

For each row, the **recall** Rec is the fraction of cases that are truly κ that were classified as κ,

$$Rec(i) = \frac{c_{i,i}}{\sum_j c_{i,j}}.$$

If need be, the sets of precision measures and recall measures can then each be averaged over the classes to get an overall estimate of the precision and recall of the method.

Appendix 3: Tables

Table 1a: The Penn part-of-speech tagset—words

Tag	**Name**	**Example**
CC	Coordinating conjunction	*and*
CD	Cardinal number	*three*
DT	Determiner	*the*
EX	Existential *there*	*there [is]*
FW	Foreign word	*d'oeuvre*
IN	Preposition or subordinating conjunction	*in, of, like*
JJ	Adjective	*green, good*
JJR	Adjective, comparative	*greener, better*
JJS	Adjective, superlative	*greenest, best*
LS	List item marker	*(1)*
MD	Modal	*could, will*
NN	Noun, singular or mass	*table*
NNS	Noun, plural	*tables*
NNP	Proper noun, singular	*John*
NNPS	Proper noun, plural	*Vikings*
PDT	Predeterminer	*both [the boys]*
POS	Possessive ending	*'s, '*
PRP	Personal pronoun	*I, he, it*
PRP$	Possessive pronoun	*my, his, its*
RB	Adverb	*however, usually, naturally, here, good*
RBR	Adverb, comparative	*better*
RBS	Adverb, superlative	*best*
RP	Particle	*[give] up*
SYM	Symbol (mathematical or scientific)	+
TO	*to*	*to [go] to [him]*
UH	Interjection	*uh-huh*
VB	Verb, base form	*take*
VBD	Verb, past tense	*took*
VBG	Verb, gerund or present participle	*taking*
VBN	Verb, past participle	*taken*
VBP	Verb, non-3rd-person singular present	*take*
VBZ	Verb, 3rd-person singular present	*takes*
WDT	*wh*-determiner	*which*
WP	*wh*-pronoun	*who, what*
WP$	Possessive *wh*-pronoun	*whose*
WRB	*wh*-adverb	*where, when*

Table 1b: The Penn part-of-speech tagset—punctuation

Tag	Name	Example
#	Pound sign	*£*
$	Dollar sign	$
.	Sentence-final punctuation	*!, ?, .*
,	Comma	
:	Colon, semi-colon, ellipsis	
(	Left bracket character	
)	Right bracket character	
"	Straight double quote	
‘	Left open single quote	
“	Left open double quote	
’	Right close single quote	
”	Right close double quote	

Table 2: Conversion from raw tweets to tagged tweets

Raw tweet:

```
Meet me today at the FEC in DC at 4. Wear a carnation so I know
it’s you. <a href="Http://bit.ly/PACattack" target="_blank"
class="tweet-url web" rel="nofollow">Http://bit.ly/PACattack</a>.
```

Output from *twtt.py*:

```
...
|
Meet/VB me/PRP today/NN at/IN the/DT FEC/NN in/IN DC/NN at/IN 4/NN ./.
Wear/VB a/DT carnation/NN so/RB I/PRP know/VB it/PRP ’s/POS you/PRP ./.
|
...
```

Table 3: Features to be computed for each text

- Counts:
  - — First person pronouns
  - — Second person pronouns
  - — Third person pronouns
  - — Coordinating conjunctions
  - — Past-tense verbs
  - — Future-tense verbs
  - — Commas
  - — Colons and semi-colons
  - — Dashes
  - — Parentheses
  - — Ellipses
  - — Common nouns
  - — Proper nouns
  - — Adverbs
  - — *wh*-words
  - — Modern slang acroynms
  - — Words all in upper case (at least 2 letters long)
- Average length of sentences (in tokens)
- Average length of tokens, excluding punctuation tokens (in characters)
- Number of sentences

Table 4: Definitions of feature categories

First person:
I, me, my, mine, we, us, our, ours
Second person:
you, your, yours, u, ur, urs
Third person:
he, him, his, she, her, hers, it, its, they, them, their, theirs
Future Tense:
*'ll, will, gonna, going+to+*VB
Common Nouns:
NN, NNS
Proper Nouns:
NNP, NNPS
Adverbs:
RB, RBR, RBS
wh-words :
WDT, WP, WP$, WRB
Modern slang acronyms:
smh, fwb, lmfao, lmao, lms, tbh, rofl, wtf, bff, wyd, lylc, brb, atm, imao, sml, btw, bw, imho, fyi, ppl, sob, ttyl, imo, ltr, thx, kk, omg, ttys, afn, bbs, cya, ez, f2f, gtr, ic, jk, k, ly, ya, nm, np, plz, ru, so, tc, tmi, ym, ur, u, sol

Table 5: Example `weather.arff` for WEKA

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature numeric
@attribute humidity numeric
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,85,85,FALSE,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
rainy,70,96,FALSE,yes
rainy,68,80,FALSE,yes
rainy,65,70,TRUE,no
overcast,64,65,TRUE,yes
sunny,72,95,FALSE,no
sunny,69,70,FALSE,yes
rainy,75,80,FALSE,yes
sunny,75,70,TRUE,yes
overcast,72,90,TRUE,yes
overcast,81,75,FALSE,yes
rainy,71,91,TRUE,no
```