NLP (CSC 306)
Winter 2025

Project 2 – Text Classification

Due: Thursday, 2/6, end of the day (Code)
Friday, 2/7, end of the day (Report)

For this assignment, you will be building a text classifier. The goal of this text classifier will be to distinguish between words that are simple and words that are complex. Examples of simple words are *heard*, *sat*, *feet*, *shops*, *town*, and examples of complex words are *abdicate*, *detained*, *liaison*, *vintners*. Distinguishing between simple and complex words is one step in a larger NLP task called text simplification, which aims to make text easier to read. Replacing complex words with simpler ones is one way in which text can be simplified. Text simplification is useful for re-writing texts so that they can be more easily understood by younger readers, people learning English as a second language, people with learning disabilities, and others with low literacy skills.

Learning Objectives

- Understand an important class of NLP evaluation methods (precision, recall and F1), and implement them yourself.
- Employ common experimental design practices in NLP. Split the annotated data into training, development, and test sets, implement simple baselines to determine how difficult the task is, and experiment with a range of features and models.
- Get an introduction to applying classification using sklearn (a widely used Python package for machine learning).

Things to Download

- starter code (Download from Nexus):
  - `evaluation.py`: This is where you will implement functions for calculating precision, recall, f-score.
  - `complex_word_classification.py`: This is where you will implement different approaches to classifying words as complex or simple.
  - `movie_review_classification.py`: This is an implementation of movie review classification using scikit learn classifiers. Consult this as an example how to use the scikit learn classifiers.
  - `syllables.py` Some helper functions for counting syllables.
- data (Download using the links below)
  - training and development data: This is a tarball with the training/dev/test sets. *Read the note below when working on the lab computers.*
  - word counts from the Google N-gram corpus *Read the note below when working on the lab computers.*

The Python files are available on Nexus. The data sets and Google word counts are available at `http://cs.union.edu/~striegnk/csc483/data.tgz` and `http://cs.union.edu/~striegnk/csc483/ngram_counts.txt.gz`.

Important! Where to save things when working on the lab machines Throughout the course you will at times work with datasets that are quite large. (Here: training/dev/test sets and the unigram counts.) You should ***not*** **save any datasets in your home directories.** When working on the lab computers, please save them in the folder `/var/csc306`. This is a shared local directory on each lab machine that all of you should be able to read and write. When you download datasets, make sure that no copies are left in your Downloads folder. For example: Download to `Downloads` in your home directory. Then, open a terminal and type: `mv Downloads/filename /var/csc306`

Ask me, if you need help with this.

Deliverables

- Your completed code. I.e. the files `evaluation.py` and `complex_word_classification.py`.
- The file `test_labels.txt` described below.
- A written report in PDF format. The report should follow the guidelines posted on Nexus. In addition, the instructions below specify some requirements of what should be contained in the report.

Please do NOT upload any data files Gradescope.

Background: Identifying Complex Words

Automated text simplification is an NLP task, where the goal is to take as input a complex text, and return a text that is easier to understand. One of the most logical first steps in text simplification is identifying which words in a text are hard to understand, and which words are easy to understand. This is an example of text classification.

You are given a labeled dataset for this assignment. The dataset consists of words and the sentences they occurred in and has been split into training, development, and test sets. The sets is disjoint, so if a word appears in the training set, it will not also appear in the test set or the development set.

This dataset was collected by taking the first 200 tokens in 200 complex texts, and crowdsourcing human judgements. Nine human annotators were asked to identify at least 10 complex words in each text. From here, words that were identified as complex by at least 3 annotators were labeled as complex. In addition, words that were identified as complex by zero annotators were labeled as simple. One thing to note is that the dataset only consists of nouns, verbs, adjectives, and adverbs. Other words, such as stopwords (i.e. common words like "the" or "and") and proper nouns were removed. The data set you can download for this homework is plit into 4,000 words for training and 800 words for development and 200 words are reserved for testing.

Shown below is an example of the training data. Note that the training data and development data files have the same formatting, and the test data does not include any information in the *label* and *annotators* columns:

word	label	annotators	sentence	sentence index
jumping	0	0	" Coleman with his jumping frog – bet stranger $ 50 – stranger had no frog & C got him one – in the meantime stranger filled C 's frog full of shot & he could n't jump – the stranger 's frog won . "	4
paths	0	0	The Cannery project will feature drought-tolerant landscaping along its bike paths , and most of the front yards will be landscaped with low-water plants in place of grass .	10
fair-weather	1	5	Months ago , many warned him not to invest in a place where fair-weather tourists flee in the fall and the big lake 's waters turn cold and storm-tossed , forcing the 100 or so hardy full-time residents of Cornucopia to hibernate for the winter .	13
krill	1	7	But unlike the other whales , the 25-foot-long young adult stuck near the surface – and it did n't dive down to feast on the blooms of krill that attract humpbacks to the bay .	27

Here is what the different fields in the file mean:

word The word to be classified

label 0 for simple words, 1 for complex words

annotators The number of annotators who labeled the word as complex

sentence The sentence that was shown to annotators when they labeled the word as simple or complex

sentence index The index of the word in the sentence (0 indexed, space delimited).

The starter code provides the function `load_file(data_file)`, which takes in the file name (`data_file`) of one of the datasets, and reads in the words and labels from these files. The `load_file()` function makes every word lowercase.

Note: While the context in which each word was found is provided, you do not need it for the majority of the assignment. The only time you may need this is if you choose to implement any context-based features in your own classifier in Step 4.

Installing Dependecies

You need two Python packages for this assignment: scikit-learn and nltk. If you are working on the lab computers, they should already be installed. Otherwise, you can install them by running the following commands from a terminal:

```
python3 -m pip scikit-learn
python3 -m pip nltk
```

You also need to download some data for nltk in order to be able to run the movie classification example. You can do that by executing the following in a Python shell:

```
import nltk
nltk.download("movie_reviews")
```

1 Implement the Evaluation Metrics

Before we start with this text classification task, we need to first determine how we will evaluate our results. The most common metrics for evaluating binary classification (especially in cases of class imbalance) are precision, recall, and f-score. For this assignment, complex words are considered positive examples, and simple words are considered negative examples.

For this problem, you will fill in the following functions:

- `get_accuracy(y_pred, y_true)`
- `get_precision(y_pred, y_true, label=1)`
- `get_recall(y_pred, y_true, label=1)`
- `get_fscore(y_pred, y_true, label=1)`

Here, `y_pred` is list of predicted labels from a classifier, and `y_true` is a list of the true labels.

You may *not* use sklearn's built-in functions for this, you must instead write your own code to calculate these metrics, using only Python's standard libaries (and numpy, if you want to). You will be using these functions to evaluate your classifiers later on in this assignment.

Make sure to test these functions well, using small made-up examples. For example:

```
>>>get_precision([1,1,0,0,1,0,0,1,0,0], [1,1,1,1,1,0,0,0,0,0], 1)
0.75
```

You should also write a function `evaluate(y_pred, y_true)`, which calculates and prints out accuracy, precision, recall, and f-score. It should calculate the accuracy for class 1/complex. This function will be helpful later on!

Please use the following format for the print-out:

```
Accuracy: 90%
Precision: 67%
Recall: 86%
F-score: 75%
```

2 Baselines

You should start by implementing simple baselines as classifiers.

2.1 A very simple baseline

Your first baseline is the simplest classifier possible: it labels all words as complex. You should complete the function `all_complex(data_file)`, which takes in the file name of one of the datasets, labels each word in the dataset as complex, and prints out the accuracy, precision, recall, and f-score. (Use the `evaluate(y_pred, y_true)` function you implemented before.)

Please report (in your written report) the accuracy, precision, recall, and f-score on the training data and on the development data.

2.2 Word length baseline

For your next baseline, you will use a slightly more complex baseline: This classifier should use the length of each word to predict its complexity.

For the word length baseline, you should write some code that tries setting various thresholds for word length to classify words as simple or complex. For example, when the threshold is 9, any words with less than 9 characters will be labeled simple, and any words with 9 characters or more will be labeled complex. Your code should try all possible thresholds. Once you find the best threshold using the training data, use this same threshold for the development data as well.

You will be filling in the function `word_length_threshold(training_file, development_file)`. This function takes in both the training and development data files, and prints out the accruacy, precision, recall, and f-score for your best threshold's performance on both the training and development data.

In your write-up, please report the best threshold you found and the evaluation metrics on the training and and on the development data for the best threshold.

2.3 Word frequency baseline

The final baseline is a classifier that is similar to the last one, but thresholds on word frequency instead of length. Google NGram frequencies are provided in the text file `ngram_counts.txt`, along with the helper function `load_ngram_counts(ngram_counts_file)` to load them into Python as a dictionary.

You will be filling in the function `word_frequency_threshold(training_file, development_file, counts)`, where `counts` is the dictionary of word frequencies. This function again prints out the accuracy, precision, recall, and f-score for the best threshold's performance on both the training and development data.

In this case, you can not try *all* possible frequency thresholds. To see why, print out the minimum and the maximum frequencies in `counts`. So you have to decide which frequency thresholds you want to try.

In your write-up, please report which thresholds you tried, what you found to be the best threshold, and all evaluation metrics on the training and and on the development data for the best threshold.

Note: Due to the size of `ngram_counts.txt`, loading the word counts into Python takes a few seconds. **Make sure to *not* save a copy of `ngram_counts.txt` in your home directory on the lab machines.**

3 Classifiers

Now, let's move on to actual machine learning classifiers!

3.1 Naive Bayes classification

For our first classifier, you will use the built-in Naive Bayes model from sklearn, to train a classifier. You should refer to the online sklearn documentation when you are building your classifier. Also, see the example given in `movie_review_classification.py`.

The first thing to note is that sklearn classifiers take in numpy arrays, rather than regular lists. Numpy is a Python library that provides an efficient implementation of multi-dimensional arrays.

To train a classifier, you need two numpy arrays:

1. `train_x`, an m by n array (m rows, n columns; you can also think of it as a nested list that contains m lists of n elements each), where m is the number of words in the dataset, and n is the number of features for each word
2. `train_y`, an array of length m for the labels of each of the words.

Once we have these two arrays, we can fit a Naive Bayes classifier using the following commands:

```
>>> from sklearn.naive_bayes import GaussianNB
>>> clf = GaussianNB()
>>> clf.fit(train_x, train_y)
```

Training and testing instances are presented in to the classifier as a list of features. For example, in the movie review classification example, each review is turned into a list of numbers that represents for each of the 2000 most common words whether that word occurs in this particular review or not.

Finally, to use your model to predict the labels for a set of words, you only need one numpy array: e.g. `dev_x`, an m' by n array, where m' is the number of words in the development set, and n is the number of features for each word. Note that the n used here is the same as the n in `train_x`. Then, we can use our classifier to predict labels using the following command:

```
>>> Y_pred = clf.predict(dev_x)
```

You should fill in the function `naive_bayes(training_file,development_file, counts)`. This function should train a Naive Bayes classifier on the training data using word length and word frequency as features. That means, each word should be represented as a list of two numbers: the first is the length of the word and the second is the word's frequency according to the Google ngrams. This is the information based on which the classifier will learn to assign a class. The function should print out your model's accuracy, precision, recall, and f-score on the training data and the development data.

In your write-up, please report all evaluation metrics on the training and development data for your Naive Bayes classifier that uses word length and word frequency.

NOTE: Before training and testing a classifier, it is generally important to normalize your features. This means that you need to find the mean and standard deviation (sd) of a feature. Then, for each row, perform the following transformation:

```
X_scaled = (X_original - mean)/sd
```

In Python:

```
x_scaled = (x - train_x.mean(axis=0)) / train_x.std(axis=0)
```

Be sure to always use the means and standard deviations from the training data (even when predicting the class of a new word).

3.2 Logistic Regression

Next, you will use sklearn's built-in Logistic Regression classifier. Again, you should use word length and word frequency as your two features. You should refer to the online sklearn documentation when you are building your classifier. To import and use this model, use the following command:

```
>>> from sklearn.linear_model import LogisticRegression
>>> clf = LogisticRegression()
```

Complete the function `logistic_regression(training_file, development_file, counts)`. This function should train a Logistic Regression classifier on the training data, and print out your model's accuracy, precision, etc. on the training data and the development data.

Again, please **report all evaluation metrics on the training and development data.**

3.3 Comparing Naive Bayes and Logistic Regression

After implementing Naive Bayes and Logistic Regression classifiers, you will notice that their performance is not identical, even though they are given the same data. Add a paragraph to your write up that discusses which model performed better on this task.

4 Build your own model

Finally, the fun part! In this section, you will build your own classifier for the complex word identification task. You will also perform an error analysis for your best performing model.

4.1 Build your model

You can choose any other types of classifier, and any additional features you can think of! For classifiers, beyond Naive Bayes and Logistic Regression, you might consider trying SVM, Decision Trees, and Random Forests, among others. Additional word features that you might consider include number of syllables, number of WordNet synonyms. For counting the number of syllables, we have provided a python script `syllables.py` that contains the function `count_syllables(word)`, which you may use. To use WordNet in Python, refer to this documentation. The following Python command gives you the number of synonyms a word has:

```
from nltk.corpus import wordnet as wn
len(wn.synsets(word))
```

You could also include sentence-based complexity features, such as length of the sentence, average word length, and average word frequency.

When trying different classifiers, we recommend that you train on training data, and test on the development data, like the previous sections.

In your writeup, please include a description of all of the classifiers and features that you tried. You are required to try *at least* 1 other type of classifier (besides Naive Bayes and Logistic Regression) and at least two other features (besides length and frequency).

4.2 Analyze your model

An important part of text classification tasks is to determine what your model is getting correct, and what your model is getting wrong. For this problem, you must train your best model on the training data, and **report the accuracy, precision, recall, and f-score on the development data.**

In addition, you need to **perform a detailed error analysis of your models.** Give several examples of words on which your best model performs well. Also give examples of words which your best model performs poorly on. Can you identify categories of words on which your model is making errors?

4.3 Run your model on the unlabeled test set

Finally, combine the training and development data and train your best model on the combined set. Then use this classifier to predict labels for the unlabeled test data and submit these labels in a text file named `test_labels.txt` (with one label per line).

5 Recommended readings

I you are looking for inspiration what features to try, have a look at the following papers: Shardlow [2013], Paetzold and Specia [2016], Kriz et al. [2018]

Submit

Submit your implementation on Gradescope.

- Your completed code. I.e. the files `evaluation.py` and `complex_word_classification.py`.
- The file `test_labels.txt`.

Do **NOT** submit the data files.

Submit your report on Nexus.

- Report in PDF format.
- The LaTeX sources.

References

Reno Kriz, Eleni Miltsakaki, Marianna Apidianaki, and Chris Callison-Burch. Simplification using paraphrases and context-based lexical substitution. In *The 2018 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL 2018)*, New Orleans, Louisiana, June 2018. URL `http://aclweb.org/anthology/N18-1019`.

Gustavo Paetzold and Lucia Specia. Semeval 2016 task 11: Complex word identification. In *10th International Workshop on Semantic Evaluation (SemEval)*, pages 560–569, 2016. URL `http://www.aclweb.org/anthology/S16-1085`.

Matthew Shardlow. A comparison of techniques to automatically identify complex words. In *51st Annual Meeting of the Association for Computational Linguistics, Student Research Workshop*, pages 103–109, 2013. URL `http://www.aclweb.org/anthology/P13-3015`.