EECS 445 Introduction to Machine Learning
Mini Project
Sentiment Analysis
Due Nov 24, 5pm
[Last Updated: Nov 2]

Overview:

In this mini project, you will experiment with and select different machine learning algorithms to do sentiment analysis on a movie review dataset. You will also play with various text preprocessing techniques to do feature engineering. This document includes the following sections:

- Section 1: Introduction
  - (a) Introduction to the files on Ctools
  - (b) Programming language
  - (c) Helpful Python libraries
    - i. scikit-learn
    - ii. re
  - (d) Handy Python functions/methods
    - i. split(sep)
    - ii. zip(iterable1, iterable2)
  - (e) Policies on use of external libraries
- Section 2
  - (a) Overview of the pipeline
  - (b) [30 points] Text preprocessing
    - i. [5 points] Lemmatization
    - ii. [5 points] Stemming
    - iii. [5 points] Remove Stopwords
    - iv. [5 points] tf (term frequency)
    - v. [5 points] tf-idf (term frequency-inverse document frequency)
    - vi. [(extra credit) 10 points] Additional preprocessing
    - vii. [5 points] Wrap-up
  - (c) [20 points] Apply basic machine learning algorithm
  - (d) [20 points] Control experiments with different options
    - i. [10 points] tf v.s. tf-idf
    - ii. [10 points] Remove stopwords or not
  - (e) [10 points] Using word2vec
  - (f) [20 points] Wrap everything up
- Section 3: Instructions on submission
- Section 4: Updates

Section 1: Introduction

(a) Introduction to the files on Ctools

In the same directory as this specification file on ctools you can find a zip file called miniproj.zip. Unzip it you will get a folder called miniproj/ and inside it there are four files: main.py, train.txt, validation and test.txt.

In main.py is the starter code in Python and there lists all the functions that you need to fill in. If you are trying to use a language different than Python, please define the corresponding functions using the same name as in main.py.

The train.txt file has 14,942 lines of text, each of which makes a training data point. Every line is in the [label][tab][raw text] format. At the beginning of the line, we can see a **binary** class label either +1 or -1, with +1 standing for positive emotion and -1 for negative emotion. These labels are assigned by human and considered as ground truth. After that we will see a tab to separate the label from the raw text. The last part is the raw text which consists of several paragraphs of review from a person towards a particular movie. Notice paragraphs are separated by a
 symbol rather than a newline because newline in this file is used to separate different data points.

The validation.txt file has 9,962 lines of text that are in the same format as those in train.txt. The data in this file are mainly used for hyper-parameter tuning in section 2 part c. They will also be used to determine the vocabulary in section 2 part b.

The test.txt file has 25,000 lines of unlabeled raw test for you to do classification on. Most of the time you don't need this piece of data except when you want to determine the vocabulary and in section 2 part f where you will choose the best model you have obtained and apply it to the test data.

(b) Programming language

In this document, the Python language and some libraries in Python are used for illustration purpose. You are welcome to use other programming language you are comfortable with.

(c) Helpful Python libraries

i. scikit-learn

scikit-learn is a library in Python that contains most of the major machine learning algorithms such as logistic regression, naive Bayes and support vector machine. One nice thing about scikit-learn is that inside it different classes of machine learning algorithms implement a common interface that contains the most commonly operations. For example, every class, regression or classification, implements a fit and a predict method. After you create an instance of a particular class, you can train it by calling the fit method, providing the feature matrix and labels as argument. Later, you can use this model to make predictions by calling the predict method, passing the feature matrix of the test data as argument. For example, below illustrates an example of using logistic regression to do classification

```
from sklearn.linear_model import LogisticRegression

X = [[1, 3], [-2, -1], [5.2, 0.4]]    # features in training data
y = [0, 1, 0]                         # labels of training data

# create an instance of logistic regression
# C is the inverse of the regularization term λ
# Smaller C means more regularization
model = LogisticRegression(C=10)
model.fit(X, y)                       # train the model
prediction = model.predict([4.9, 3])  # make prediction
```

** Note: the name of scikit-learn in Python is called "sklearn" because the term "scikit-learn" contains a "-" character which is a reserved character in Python*

ii. re

The re library provides a set of **r**egular **e**xpression tools in Python which are useful when you are doing text preprocessing. For example, one possible task in text preprocessing is to replace all the instances of float point number with a uniform token 'NUM'. We can use the sub function (for substitute) in the re library to achieve this.

```
import re
pattern = '[+-]?\d+\.\d+'
string = 'The value of pi is 3.14159'

# new_string will be 'The value of pi is NUM'
new_string = re.sub(pattern, 'NUM', string)
```

For more of regular expression, please refer to http://www.regular-expressions.info/tutorial.html. Also there is a nice tutorial on regular expression at http://perldoc.perl.org/perlretut.html. But notice the examples in the website are illustrated in perl where a pattern is embedded in two single slashes. For example, if we have `pattern = '[+-]?\d+\.\d+'` in Python, its counterpart in perl is `/[+-]?\d+\.\d+/`.

(d) Handy Python functions/methods

i. `split(sep)`

The split method of a string will split itself using the argument sep to the method as the delimiter. The default value of the delimiter is space. The output of this function is a list of tokens surrounded by the delimiter in the original string. This is useful when you are doing the basic tokenization. Below is an example to illustrate this.

```
s = 'What a nice day. I just got a mini project to do!'

# l = ['What', 'a','nice', 'day.', 'I', 'just', 'got', 'a', 'mini',
'project', 'to','do!']
l = s.split()
```

ii. zip(iterable1, iterable2)

This function is useful when you want to iterate two lists (or sets or any iterables) parallely. For example, sometimes we want to iterate the feature and its label at the same time. The snippet below achieves this.

```
for phi, t in zip(X, labels):
     w -= gradient(phi, t)
```

which is equivalent to the following (not "Pythonic"):

```
for i in range(len(labels)):
     phi, t = X[i], labels[i]
     w -= gradient(phi, t)
```

(e) Policies on use of external libraries

In this project you are free to use any external libraries to complete the tasks in section 2. You are also encouraged to discuss and share thoughts with your classmates. But you should write your code independently.

Note: For more on how to use external libraries, please refer to section 4.1.

Section 2

(a) Overview of the pipeline

Throughout this section, you may need to write a main function that drives the entire process to test your code in the following parts. When the main function is invoked, your code should

1. read training, validation and test data and extract their text and labels
2. preprocess the texts and convert them to a feature matrix like object ready for training, validation and testing
3. train the model using the features from 2 and labels from 1
4. do predictions on test data and save the predictions in a csv file called predictions.txt.

Specifically, we provide the main function main.py as a starter code. You will need to fill in this main function to test your code and get results.

(b) [30 points] Text preprocessing

In the simplest scenario, we just apply the split method (section 1, part d.ii) to the text and for every movie review we compute the number of occurrences of every word in the vocabulary. But doing this will result in an excessive vocabulary where tokens with close relationship are treated as completely different features. This can be problematic because the feature matrix can be very sparse and we may end up with more features than number of data points. To alleviate this "curse of dimensionality", we can use some techniques to assign closely related tokens to the same group and reduce duplicate in the feature space. For example, we may want to replace tokens following the pattern XXX-XXX-XXXX where X is a digit with a single token "TEL_NUM" and XXXX-XX-XX with another token "DATE". In other cases, you may think we'd better group tokens such as "do" and "did" in the same word class because "*did*" is just the past tense of the token "*do*". In the following, you will explore the techniques to reduce the size of the vocabulary and to make more meaningful features.

** Please note that doing the text preprocessing does not guarantee improvement of classification accuracy and more techniques used doesn't necessarily yield better features. It's up to you to include your optimal subset of text preprocessing tools in your final classifier to*

achieve the highest accuracy you can have. ***That said, you are still expected to finish all of the following (except the bonus part) to get full credits for (b)****.*

i. [5 points] Lemmatization

Lemmatization refers to the process of grouping the different inflected forms of a word into the same token. For example, we may want to group the tokens "*am", "are"* and *"is"* in a single form "*be".* To lemmatize a string, the CoreNLP component in the stanford_corenlp_pywrapper library in Python may be helpful to you. To find out more, please visit https://github.com/brendano/stanford_corenlp_pywrapper.

What you need to do: fill in the lemmatize function in file main.py

Note: for more instructions on how to use CoreNLP wrapper, please refer to Section4.2

[Updated Oct 31] Some lines in the data may contain non-ascii characters. Do nothing these words when lemmatizing because they are relatively few in number than other normal English words. To do so, set UnicodeDecodeError attribute to 'skip' when initializing the CoreNLP object.

ii. [5 points] Stemming

Stemming reduces a token to its **morphological** root form. For example, the token "*measure", "measures"* and *"measurement",* can be all stemmed to the root form "*measur"*. There are some overlapping of functionality between stemming and lemmatization. But they have one major difference in that stemming is rule based while lemmatization is dictionary based. That is, a stemming program has its own rules of how to reduce a token into its **morphological** root form while a lemmatizer relies on a dictionary to find out a token's "original" **semantic** form. For example, in some of the stemmers, the token "*was"* is stemmed as "*wa"* but is lemmatized to the token "*be"*. The Porter stemmer is one of the most popular stemmer in text classification. For more of the Porter stemmer and various other variations of stemmers, please visit http://www.nltk.org/api/nltk.stem.html.

What you need to do: fill in the stem function in file main.py using the PorterStemmer **provided in nltk.stem.porter.**

iii. [5 points] Remove Stopwords

Words such as "*the*", "*an*" and "*a*" are called stopwords and they appear in most of the documents without carrying significant semantic meaning. In some cases, we may want to remove stopwords to reduce the vocabulary size. In this part, you will need to implement a function called removeStopwords to do this.

What you need to do: fill in the removeStopwords function in main.py to remove stopwords as listed in the stopwords.words('english') property in nltk.corpus

iv. [5 points] tf (term frequency)

The term frequency refers to the how many times a term (token) in the vocabulary appears in a document, a movie review in our case. For example, in our second and third homework, we count the number of occurrences of every token in an email, spam or non-spam or a movie review with positive or negative emotion.

What you need to do: fill in the function tf in main.py. You are encouraged to use existing libraries.

Note: when you are invoking this function make sure the argument to the function contains all the reviews from the training, validation and test dataset because some of the tokens may not appear in all of the three and this ensures the training data points are of the same dimension with validation and test data points.

v. [5 points] tf-idf (term frequency-inverse document frequency)

One potential problem with using term frequency as features is that some words, such as stopwords, appears in most of the documents and the appearance of such words give no information to the classifier as whether this document is spam or non-spam, positive emotion or negative emotion. On the other hand, some rare words such as *"narcissism"* rarely appear in the corpus and may be very indicative of the nature of the document where it appears.

In general, we want to give more weight to the rarely occurred tokens and less to the commonly appeared words. This gives rise the the idea of idf, or inverse document frequency. The idf of the i-th token in the vocabulary can be computed by the following formula:

$$idf(i) = log(\frac{number\ of\ documents\ in\ the\ corpus}{number\ of\ documents\ where\ i-th\ token\ appears\ at\ least\ once})$$

As a result, a token will have higher idf if it appears in less documents. One implication is a token will have only one idf value while it can have different tf value in different documents.

To conclude, the tf-idf value is equal to the product of tf and idf values of a token in a document.

What you need to do: fill in the function tfidf in main.py. You are encouraged to use existing libraries.

Note: when you are invoking this function make sure the argument to the function contains all the reviews from the training, validation and test dataset because some of the tokens may not appear in all of the three and this ensures the training data points are of the same dimension with validation and test data points.

[Updated Oct 31] If you are using the TfidfVectorizer from sklearn.feature_extraction.text using the default parameters, you may not get the same answer as you implement by yourself. As a sanity check, feed the list ['good morning there', 'hello there', 'bye', 'good news there'] into the TfidfVectorizer and compare the output with your manual calculation. Read the documentation carefully and figure out the tricky part.

vi. [(extra credit) 10 points] Additional preprocessing

This is a bonus part. You may implement any preprocessing techniques that you think help improve the classification accuracy. We will grade this part by the quantity and quality of the additional preprocessing techniques you have here.

What you need to do: fill in the function additional in main.py. In the written-up file, state why you want to this backed by evidence from the dataset.

vii. [5 points] Wrap-up

Now that you have implemented all the preprocessing techniques, it's time to wrap them up to transform the raw text into meaningful features.

What you need to do: fill in the function preprocess in main.py

Note: You don't need to use all of the techniques that you have implemented so far. You can choose a subset of them that work best for you.

(c) [20 points] Apply basic machine learning algorithm

Now you have the features ready, you will want to train models using them. In this part, please use lemmatization, stopwords removal, tf-idf (but not tf) as preprocessing steps. (In part (d), you will compare between tf and tf-idf features and perform other control experiments.)

Before we decide to use some of the algorithms such as logistic regression and SVM, we are responsible for selecting the optimal hyper-parameters. To compare the effects of different value of a hyperparameter, we shouldn't rely on the accuracy of the model on the training data set. Instead, we should use separate data set for training and validation. Specifically we want to explore the effect of the values of C (inverse of the regularization term λ) on logistic regression.

In this part, for each of the C in the set {0.001, 0.01, 0.1, 1, 10, 100, 1000} train a logistic regression model using the training data and report their accuracies on the validation data respectively. Then repeat for SVM with linear kernel and the C (penalty factor, not the same as C in logistic regression) in the set {0.001, 0.01, 0.1, 1, 10, 100}. Last train a naive Bayes model and report its accuracy on the validation data.

[Updated Oct 31] When doing naive Bayes, for each of the classes (positive or negative sentiment), assume the tf-idf score follows a Gaussian distribution as opposed to multinomial distribution. To avoid zero variance, if for a class (positive or negative sentiment), a word has the same tf-idf value across all the documents in that class, estimate it as 1e-09 rather than 0.

What you need to do: fill in the sec2c function in main.py and in the written-up file report the accuracies of the three models on validation set for each value of the hyper-parameters as specified above.

(d) [20 points] Control experiments with different options

As we have much freedom in choosing what preprocessing techniques to use and what not to, it's worthwhile to compare whether using a but b is a wise choice. We illustrate this using two examples.

i. [10 points] tf v.s. tf-idf

You have implemented both the tf and tfidf function in the preprocessing stage. Let's see which of the two yields better results. One way (straightforward but not strictly correct) is listed in the following. We want to compare accuracies in two cases: A and B. On the preprocessing stage, A and B both do lemmatization and stopwords removal. Additionally A will use tf and B will use tf-idf as features. On the training stage, for each of A and B, train and validate a logistic regression model with C of values in set {0.001, 0.01, 0.1, 1, 10, 100, 1000}. Pick the best C value for A and B respectively and report their accuracies on the validation set. Note that the best C value for A may be different than that for B.

What you need to do: you don't need to write new function. Just play with the code you have to achieve the aforementioned.

ii. [10 points] Remove stopwords or not

It's also worth thinking about the effect of removing stopwords.

What you need to do: you don't need to write new function. Just play with the code you have to achieve the following. We want to compare accuracies in two cases: A and B. On preprocessing stage, A and B both do lemmatization and use tf-idf as features. Additionally A will do stopwords removel while B don't. On training stage, for each of A and B, train and validate a logistic regression model with C of values in set {0.001, 0.01, 0.1, 1, 10, 100, 1000}. Pick the best C value for A and B respectively and report their accuracies on the validation set. Note that the best C value for A may be different than that for B.

(e) [10 points] Using word2vec

The features in the previous sections are made based on the idea of bag-of-words. In this case, a document is considered a bag of words where we only care about the number of occurrences of the words without worrying about the context. In this case, a document is represented by a vector of dimension equal to the size of the vocabulary. Unlike in bag-of-words, the word2vec learns from a large corpus and as a result assign a vector to every word in the dictionary. Further we can construct a feature vector of a document by aggregating the vectors of the words which appear in this document. For more of word2vec, please visit https://radimrehurek.com/gensim/models/word2vec.html.

In this task, you will first need to install word2vec Python wrapper or wrapper in the language you use other than Python. Then you should train a word2vec model by feeding it the entire corpus including the training text and test text without any preprocessing. After that, the feature of a document is obtained by averaging the vectors of all the words which appears in this document. If a word appear N times in this document, add its vector N times when computing the average. Last you will try a SVM with linear kernel with C value being in the set {0.001, 0.01, 0.1, 1, 10, 100}. For each of the C value, train the model using the training data and report the accuracy on the validation dataset.

What you need to do: fill in the useWord2vec function in main.py as instructed above and report the accuracies in the written-up file.

For the the updates of this part, please refer to Section 4.4.

(f) [20 points] Wrap everything up

Now you have all the tools ready. Please train and validate your best model using features constructed in your favorite way and apply the model to make predictions on the unlabeled dataset found in test.txt. You should save your predictions in a file called *predictions.txt* where the first line is your prediction (+1 or -1) of the first data point and so forth. You will be graded for this part on how you are ranked compared with your classmates.

Section 3: Instructions on submission

Please submit a zip file called [your uniqueid].zip (mine would be rhzhang.zip)on ctools that contains your

1. written-up file. The file name should be README.txt.
2. your predictions on the test data. The file name should be predictions.txt
3. your source code in main.py or main.* where * stands for the extension of the language you use. Every function in this file should be filled. Also when it (and possibly other auxiliary file) is compiled and executed with train.txt, validation.txt and test.txt present in the same directory, we expect to have a identical file to predictions.txt generated and saved.
4. any other auxiliary source files that you written.

Section 4: Updates

1. [Oct 29] In this mini project, you will probably use scikit-learn, nltk, CoreNLP, gensim (for the Word2vec part) as well as some built-in Python libraries. The graders will have all these ready when grading. If you plan to use anything else, please write a bash script called ext_lib.sh which will install the additional libraries when executed.

2. [Oct 29] If you plan to use CoreNLP Python wrapper, first you will need to download and install it from https://github.com/brendano/stanford_corenlp_pywrapper. Second, download its java implementation at http://nlp.stanford.edu/software/stanford-corenlp-full-2015-04-20.zip. After you download it, unzip it and move it to the directory of your submission (please do not place it in any subdirectories). For example, one possible file placement would be:

   root/
   main.py
   rhzhang.txt
   ext_lib.sh
   stanford-corenlp-full-2015-04-20/

   In your code, assume the *stanford-corenlp-full-2015-04-20/* folder is placed in this manner and use relative path wherever you use the CoreNLP library. Last, follow the examples on the wrapper download page and have fun.

   Note that you don't need to include the *stanford-corenlp-full-2015-04-20/* folder in your submission because it's large in size. The graders will use their own when grading.

3. [Oct 29] Note that you don't need to include train.txt, validation.txt and test.txt in your submission. Assume they are placed in the same directory as *stanford-corenlp-full-2015-04-20/.* '

4. [Nov 2] To train a word2vec model, you may use something like
   `model = Word2Vec(sentences, size, window, min_count, workers)`

   The sentences variable should be your text corpus merged from the training, validation and testing dataset. Specifically, the sentences variable should be a list (outer) of lists (inner). Each of the inner list contains all the tokenized words from a single movie review. For example, if we have two reviews

   "The movie is really good."
   "I don't like it."

   Then sentences would be [['The', 'movie', 'is', 'really', 'good.'], ['I', 'don\'t', 'like', 'it.']].

   For other parameters, please use `size=100, window=5, min_count=1, workers=4`. After you have trained the model, you can have a sanity check by running `model.most_similar(positive=['woman'], negative=['man'])`

   Please also download the latest version of main.py since the arguments of the function useWord2vec has been changed.

   Please also note you don't need to include the statement to install gensim in your ext_lib.sh script because it's ready in the graders' machine.

   [Nov 2] Please set the parameter min_count as 1 when initializing a Word2Vec object.

5. [Oct 31] Added instructions on stem function (Section 2.b.ii)
6. [Oct 31] Things to be careful with when using TfidfVectorizer (Section 2.b.v)
7. [Oct 31] Ignore non-ascii words when lemmatizing (Section 2.b.i)
8. [Oct 31] Assume Gaussian distribution when doing naive Bayes (Section 2.c)