XCS221 Assignment 1 — Sentiment Analysis

Due Friday, Nov. 18 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs221-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Also note that your answers should be in order and clearly and correctly labeled to receive credit. Be sure to submit your final answers as a PDF and **tag all pages correctly when submitting to Gradescope**.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.
- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None           <---- This error caused the test to fail.
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))       <---- In this case, start your debugging in line 23 of grader.py.
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```

Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

Introduction

In this assignment we will be working on a sentiment classification for the movies. We will build a binary linear classifier that reads movie reviews, as you would see from the sites like Rotten Tomatoes, and predicts whether the result is positive or negative.

Rotten Tomatoes has classified these reviews as ”positive” and ”negative”, respectively, as indicated by the intact tomato on the bottom and the splattered on the top. In this assignment, you will create a simple text classification system that can perform this task automatically.

Here are two reviews of Minari `https://www.rottentomatoes.com/m/minari/reviews`, courtesy of Rotten Tomatoes (no spoilers!):

Figure 1: Here are positive and negative reviews of Minari, from Rotten Tomatoes

Advice for this assignment:

- In the context of this assignment, words are simply strings separated by whitespace. This includes case and punctuation (e.g. “great” and “Great” are considered different words).
- You might find some useful functions in `util.py`. Have a look around in there before you start coding.

1. **Sentiment Classification**

In this problem, we will build a binary linear classifier that reads movie reviews and guesses whether they are "positive" or "negative."

Do not import any outside libraries (e.g. numpy) for any of the coding parts. Only standard python libraries and/or the libraries imported in the starter code are allowed. In this problem, you must implement the functions without using libraries like Scikit-learn.

(a) [**2 points (Coding)**]

Implement the function `extractWordFeatures`, which takes a review (string) as input and returns a feature vector $\phi(x)$ (you should represent the vector $\phi(x)$ as a `dict` in Python).

(b) [**5 points (Coding)**]

Implement the function `learnPredictor` using stochastic gradient descent and minimize the hinge loss. Consider printing the training error and test error after each iteration to make sure your code is working. You must get less than 4% error rate on the training set and less than 30% error rate on the dev set to get full credit.

(c) [**4 points (Coding)**]

Create an artificial dataset for your `learnPredictor` function by writing the `generateExample` function (nested in the `generateDataset` function). Use this to double check that your `learnPredictor` works!

Hint, the dictionary key space of the phi is the same as given argument weights.

(d) [**4 points (Coding)**]

Now we will try a crazier feature extractor. Some languages are written without spaces between words. So is splitting the words really necessary or can we just naively consider strings of characters that stretch across words? Implement the function `extractCharacterFeatures` (by filling in the `extract` function), which maps each string of n characters to the number of times it occurs, ignoring whitespace (spaces and tabs).

(e) [**3 points (Written)**]

Run your linear predictor with feature extractor `extractCharacterFeatures`. Experiment with different values of n to see which one produces the smallest test error. You should observe that this error is nearly as small as that produced by word features. Why is this the case?

Construct a review (one sentence max) in which character n-grams probably outperform word features, and briefly explain why this is so.

Note: There is a function in `submission.py` that will allow you to test different values of n. See the `Docstring` of `testValuesOfN(n)` how to run it. Remember to write your final written solution.

What we expect:

- a short paragraph (4-6 sentences). In the paragraph state which value of n produces the smallest validation error, why this is likely the value that produces the smallest error.
- a one-sentence review and explanation for when character n-grams probably outperform word features.

2. **K-means clustering**

Suppose we have a feature extractor ϕ that produces 2-dimensional feature vectors, and a toy dataset $\mathcal{D}_{\text{train}} = \{x_1, x_2, x_3, x_4\}$ with

- $\phi(x_1) = [10, 0]$
- $\phi(x_2) = [30, 0]$
- $\phi(x_3) = [10, 20]$
- $\phi(x_4) = [20, 20]$

(a) **[2 points (Written)]**

Run 2-means on this dataset until convergence. Please show your work. What are the final cluster assignments z and cluster centers μ? Run this algorithm twice with the following initial centers:

i. $\mu_1 = [20, 30]$ and $\mu_2 = [20, -10]$

ii. $\mu_1 = [0, 10]$ and $\mu_2 = [30, 20]$

What we expect: Show the cluster centers and assignments for each step.

(b) **[5 points (Coding)]**

Implement the `kmeans` function. You should initialize your k cluster centers to random elements of `examples`.

After a few iterations of k-means, your centers will be very dense vectors. In order for your code to run efficiently and to obtain full credit, you will need to precompute certain dot products. As a reference, our code runs in under a second on cardinal, on all test cases. You might find `generateClusteringExamples` in `util.py` useful for testing your code.

Do not use libraries such as Scikit-learn.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

1.e

2.a