Sentiment Analysis

Stanford CS221 Spring 2018-2019

Owner CA: Hao Wang

Version: 4

General Instructions

This (and every) assignment has a written part and a programming part.

The full assignment with our supporting code and scripts can be downloaded as sentiment.zip.

a. This icon means a written answer is expected in `sentiment.pdf`.

b. This icon means you should write code in `submission.py`.

All written answers must be **in order** and **clearly and correctly labeled** to receive credit.

You should modify the code in `submission.py` between

```
# BEGIN_YOUR_CODE
```

and

```
# END_YOUR_CODE
```

but you can add other helper functions outside this block if you want. Do not make changes to files other than `submission.py`.

Your code will be evaluated on two types of test cases, **basic** and **hidden**, which you can see in `grader.py`. Basic tests, which are fully provided to you, do not stress your code with large inputs or tricky corner cases. Hidden tests are more complex and do stress your code. The inputs of hidden tests are provided in `grader.py`, but the correct outputs are not. To run the tests, you will need to have `graderUtil.py` in the same directory as your code and `grader.py`. Then, you can run all the tests by typing

```
python grader.py
```

This will tell you only whether you passed the basic tests. On the hidden tests, the script will alert you if your code takes too long or crashes, but does not say whether you got the correct output. You can also run a single test (e.g., `3a-0-basic`) by typing

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

We strongly encourage you to read and understand the test cases, create your own test cases, and not just blindly run `grader.py`.

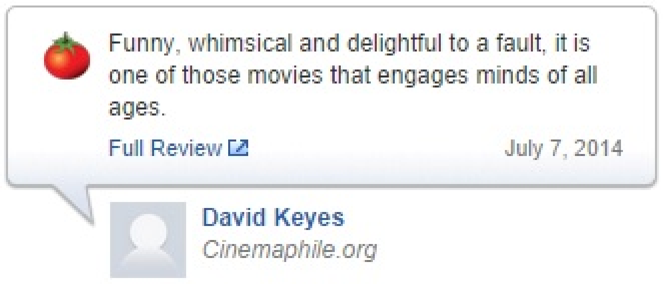
Advice for this homework:

a. Words are simply strings separated by whitespace. Note that words which only differ in capitalization are considered seperate (e.g. *great* and *Great* are considered different words).

b. You might find some useful functions in `util.py`. Have a look around in there before you start coding.

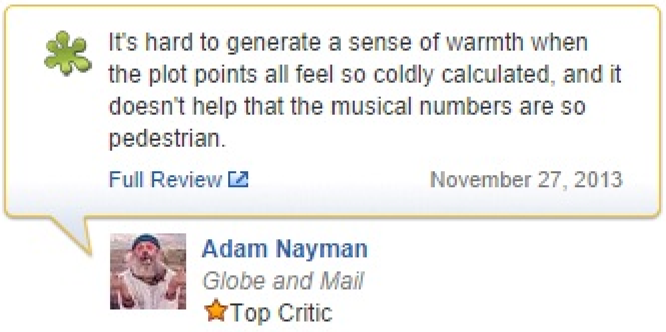
Problem 1: Building intuition

Here are two reviews of *Frozen*, courtesy of Rotten Tomatoes (no spoilers!):


Funny, whimsical and delightful to a fault, it is one of those movies that engages minds of all ages.

Full Review July 7, 2014

David Keyes
Cinemaphile.org


It's hard to generate a sense of warmth when the plot points all feel so coldly calculated, and it doesn't help that the musical numbers are so pedestrian.

Full Review November 27, 2013

Adam Nayman
Globe and Mail
Top Critic

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

We'll warm up with the following set of four mini-reviews, each either labeled positive ($+1$) or negative (-1):

1. (-1) pretty bad
2. ($+1$) good plot
3. (-1) not good
4. ($+1$) pretty scenery

Each review x is mapped onto a feature vector $\phi(x)$, which maps each word to the number of occurrences of that word in the review. For example, the first review maps to the (sparse) feature vector $\phi(x) = \{\text{pretty} : 1, \text{bad} : 1\}$. Recall the definition of the hinge loss:

$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{0, 1 - \mathbf{w} \cdot \phi(x) y\},$$

where y is the correct label.

a. [2 points] Suppose we run stochastic gradient descent, updating the weights according to

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}_{\text{hinge}}(x, y, \mathbf{w}),$$

once for each of the four examples in the order given above. After the classifier is trained on the given four data points, what are the weights of the six words ("pretty", "good", "bad", "plot", "not", "scenery") that appear in the above reviews? Use $\eta = .5$ as the step size and initialize $\mathbf{w} = [0, \ldots, 0]$. Assume that $\nabla_{\mathbf{w}} \text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = 0$ when the margin is exactly 1.

1. (score=0, loss=1) {pretty: -.5, bad: -.5}
2. (score=0, loss=1) {pretty: -.5, bad: -.5, good: .5, plot: .5}
3. (score=+.5, loss=1.5) {pretty: -.5, bad: -.5, good: 0, plot: .5, not: -.5}
4. (score=-.5, loss=1.5) {pretty: 0, bad: -.5, good: 0, plot: .5, not: -.5, scenery: .5}

(Only the weights in the last line need to be included in the given answer)

b. [2 points] Create a small labeled dataset of four mini-reviews using the words "not", "good", and "bad", where the labels make intuitive sense. Each review should contain one or two words, and no repeated words. Prove that no linear classifier using word features can get zero error on your dataset.

Remember that this is a question about classifiers, not optimization algorithms; your proof should be true for any linear classifier, regardless of how the weights are learned.
After providing such a dataset, propose a single additional feature that we could augment the feature vector with that would fix this problem. (Hint: think about the linear effect that each feature has on the classification score.)

1. (-1) bad
2. ($+1$) good
3. ($+1$) not bad

4. (-1) not good

Proof: to classify "bad" and "good" correctly, they must have negative and positive weights, respectively. But if "not" has a positive weight then "not good" is classified incorrectly, and if "not" has a negative weight then "not bad" is classified incorrectly. A single redeeming feature would be (for example) an indicator feature for the phrase "not good". Then the following weight vector classifies all the examples correctly:

$$\mathbf{w} = \{\text{"bad"} : -1,\ \text{"good"} : 1,\ \text{"not"} : 2,\ \text{"not good"} : -4\}$$

Problem 2: Predicting Movie Ratings

Suppose that we are now interested in predicting a numeric rating for each movie review. We will use a non-linear predictor that takes a movie review x and returns $\sigma(\mathbf{w} \cdot \phi(x))$, where $\sigma(z) = (1 + e^{-z})^{-1}$ is the logistic function that squashes a real number to the range $(0, 1)$. Suppose that we wish to use the squared loss. For this problem, assume that the movie rating y is a real-valued variable in the range $[0, 1]$.

a. [2 points] Write out the expression for $\text{Loss}(x, y, \mathbf{w})$.

$$\text{Loss}(x, y, \mathbf{w}) = (\sigma(\mathbf{w} \cdot \phi(x)) - y)^2.$$

b. [3 points] Compute the gradient of the loss with respect to $\mathbf{w}$.
Hint: you can write the answer in terms of the predicted value $p = \sigma(\mathbf{w} \cdot \phi(x))$.

$$\nabla \text{Loss}(x, y, \mathbf{w}) = 2(p - y)p(1 - p)\phi(x),$$

where $p = \sigma(\mathbf{w} \cdot \phi(x))$ is the predicted value. Importantly, $p \in (0, 1)$.

c. [3 points] Suppose there is one datapoint (x, y) with some given $\phi(x)$ and y = 1. Can you choose a $\mathbf{w}$ to make the magnitude of the gradient of the loss with respect to $\mathbf{w}$ arbitrarily small (i.e., minimize the magnitude of the gradient and make it asymptotically approach some value)? If so, how small? Can the magnitude of the gradient ever be exactly zero? You are allowed to make the magnitude of $\mathbf{w}$ arbitrarily large.

Hint: try to understand intuitively what is going on and the contribution of each part of the expression. If you find yourself doing too much algebra, you're probably doing something suboptimal.

Motivation: the reason why we're interested in the magnitude of the gradients is because it governs how far gradient descent will step. For example, if the gradient is close to zero when $\mathbf{w}$ is very far from the optimum, then it could take a long time for gradient descent to reach the optimum (if at all). This is known as the *vanishing gradient problem* when training neural networks.

To make the magnitude of the gradient arbitrarily small, we need to to make p infinitely close to 1 or 0. For 1, we could select a $\mathbf{w}$ of great magnitude such that $\mathbf{w} \cdot \phi(x)$ approaches positive infinity, causing the sigmoid to approach 1. For 0, select a $\mathbf{w}$ of great magnitude such that $\mathbf{w} \cdot \phi(x)$ approaches negative infinity, causing the sigmoid to approach 0. In either case, the expression $(p - 1)p(1 - p)$ gets arbitrarily close to 0, but never exactly hits 0 because $p \in (0, 1)$. This causes the magnitude of the gradient to get arbitrarily close to 0 but never exactly reach it.

d. [3 points] Assuming the same data point as above, what is the largest magnitude that the gradient can take? Leave your answer in terms of $\|\phi(x)\|$.

To make the magnitude of the gradient large, we want to maximize or minimize $(p - 1)p(1 - p)$ (we are interested in any extreme value, whether positive or negative), where $p \in (0, 1)$. An extreme value will occur at the endpoints or where the derivative is 0. At the endpoints (0 and 1), we already know the gradient approaches 0. We can now set the derivative to 0 and solve.

$$\nabla(p - 1)p(1 - p) = \nabla(-p^3 + 2p^2 - p) = -3p^2 + 4p - 1 = 0$$

The solutions are $p = \frac{-4 \pm \sqrt{16-12}}{-6} = \{\frac{1}{3}, 1\}$. We've already checked $p = 1$, so we only need to check $p = \frac{1}{3}$. At this point, the gradient is

$$2 \cdot (\frac{1}{3} - 1) \cdot \frac{1}{3} \cdot (1 - \frac{1}{3}) \cdot \|\phi(x)\| = -\frac{8}{27}\|\phi(x)\|$$

. The largest magnitude possible is the absolute value of this, which is

$$\frac{8}{27}\|\phi(x)\|$$

. Notice that is a minimum because the second derivative is positive:

$$\nabla(-3p^2 + 4p - 1) = -6p + 4 = -6 \cdot \frac{1}{3} + 4 = 2 > 0.$$

The previous problem and this problem hint at the vanishing gradient problem in training neural networks, where the gradient is (close to) zero even when the weights $\mathbf{w}$ are far away from the global optimum (p close to 0 or 1). However, the gradient tends to be large when p is closer to the center.

e. [3 points] The problem with the loss function we have defined so far is that is it is non-convex, which means that gradient descent is not guaranteed to find the global minimum, and in general these types of problems can be difficult to solve. So let us try to reformulate the problem as plain old linear regression. Suppose you have a dataset $\mathbf{D}$ consisting of (x, y) pairs, and that there exists a weight vector $\mathbf{w}$ that yields zero loss on this dataset. Show that there is an easy transformation to a modified dataset $\mathbf{D}'$ of (x, y') pairs such that performing least squares regression (using a linear predictor and the squared loss) on $\mathbf{D}'$ converges to a vector $\mathbf{w}^*$ that yields zero loss on $\mathbf{D}'$. Concretely, write an expression for y' in terms of y and justify this choice. This expression should not be a function of $\mathbf{w}$.

Transform each training point to (x, y) to (x, y'), where

$$y' = \sigma^{-1}(y) = \log\left(\frac{y}{1-y}\right)$$

inverts the transformation σ (which is monotonic). Now suppose $\mathbf{w}$ obtains zero loss on (x, y'), which means that $(\mathbf{w} \cdot \phi(x) - y')^2 = 0$, or equivalently, $\mathbf{w} \cdot \phi(x) = y' = \sigma^{-1}(y)$. Applying σ to both sides yields $\sigma(\mathbf{w} \cdot \phi(x)) = y$, which is saying that $\mathbf{w}$ also obtains zero loss on the original example (x, y).

Problem 3: Sentiment Classification

In this problem, we will build a binary linear classifier that reads movie reviews and guesses whether they are "positive" or "negative." In this problem, you must implement the functions without using libraries like Scikit-learn.

a. [2 points] 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. [4 points] Implement the function `learnPredictor` using stochastic gradient descent and minimize the hinge loss. Print 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. [2 points] 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!

d. [2 points] When you run the grader.py on test case `3b-2`, it should output a `weights` file and a `error-analysis` file. Look through some example incorrect predictions and for five of them, give a one-sentence explanation of why the classification was incorrect. What information would the classifier need to get these correct? In some sense, there's not one correct answer, so don't overthink this problem. The main point is to convey intuition about the problem.

> Here's an example of a positive review that was misclassified as negative:
>
> *wickedly funny, visually engrossing, never boring, this movie challenges us to think about the ways we consume pop culture.*
>
> After looking at the weights in the error-analysis, we find that positive words such as "engrossing" and "funny" are outweighed by negative words such as "never" and "boring" although "never boring" together represent a positive sentiment. Additional information such as which words appear together would help the classifier assign a positive weight to "never boring" and handle negations somewhat more robustly. The idea here is similar to `extractCharacterFeatures` but instead of extracting n-grams of characters, we would add features that are n-grams of words. For example, counts of bigrams (2 consecutive words) could be added to the feature set.

e. [2 points] 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).

f. [3 points] 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. How do you explain this?

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

> The test error is minimized when n is about 5 to 7. Because words in English are roughly this length, character grams can still mostly pick up on words. In addition, character grams can capture the context of adjacent words. A simple example of a review that character grams perform well on is "not good". Word features do not detect that "not" inverts "good", whereas character grams will generate features such as "notgo", which can be detected as negative.

Problem 4: 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

1. $\phi(x_1) = [1, 0]$
2. $\phi(x_2) = [1, 2]$
3. $\phi(x_3) = [3, 0]$
4. $\phi(x_4) = [2, 2]$

a. [2 points] 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:

1. $\mu_1 = [2, 3]$ and $\mu_2 = [2, -1]$
2. $\mu_1 = [0, 1]$ and $\mu_2 = [3, 2]$

> 1. Initial clusterings: μ_1 has $\{x_2, x_4\}$ and μ_2 has $\{x_1, x_3\}$. Then we set $\mu_1 = [\frac{3}{2}, 2]$, and $\mu_2 = [2, 0]$. This is stable.
> 2. Initial clusterings: μ_1 has $\{x_1, x_2\}$ and μ_2 has $\{x_3, x_4\}$. Then we set $\mu_1 = [1, 1]$ and $\mu_2 = [\frac{5}{2}, 1]$. This is stable.

b. [5 points] 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 quantities. 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. In this problem, you are not allowed to use libraries like Scikit-learn.

c. [5 points] Sometimes, we have prior knowledge about which points should belong in the same cluster. Suppose we are given a set S of example pairs (i, j) which must be assigned to the same cluster. For example, suppose we have 5 examples; then $S = \{(1,5), (2,3), (3,4)\}$ says that examples 2, 3, and 4 must be in the same cluster and that examples 1 and 5 must be in the same cluster. Provide the modified k-means algorithm that performs alternating minimization on the reconstruction loss:

$$\sum_{i=1}^{n} \|\mu_{z_i} - \phi(x_i)\|^2$$

where μ_{z_i} is the assigned centroid for the feature vector $\phi(x_i)$.

Recall that alternating minimization is when we are optimizing two variables jointly by alternating which variable we keep constant.

When z is fixed and we are optimizing μ, we get that each μ should be assigned to the average of the points currently assigned to it (this is no different from the usual k-means update step). Proof: the derivative of the reconstruction loss with respect to any one centroid μ_j is

$$2 \sum_{x \text{ is in } \mu_j\text{'s cluster}} (\mu_j - \phi(x))$$

To minimize, we set this equal to zero, and we get

$$\mu_j = \frac{\sum_{x \text{ is in } \mu_j\text{'s cluster}} \phi(x)}{|\{x \mid x \text{ is in } \mu_j\text{'s cluster}\}|}$$

When μ is fixed and we are optimizing over z, we need to do something new. There are two steps to this process. First, note that if $(i, j) \in S$ and $(j, k) \in S$, then i and k must also be in the same cluster (by transitivity). Therefore, S really partitions the data points into disjoint groups. Let g_i denote the group that i is assigned to; this can be computed using the Union-Find algorithm.

Now, the cluster assignments for each group must be done jointly. Specifically, we have

$$z_i = \arg \min_{z \in \{1,..,k\}} \sum_{j: g_i = g_j} \|\phi(x_j) - \mu_z\|^2$$

d. [1 point] What is the advantage of running k-means multiple times on the same dataset with the same K, but different random initializations?

K-means only converges to a local minimum. By running k-means multiple times, we can choose the best local minimum that we reached as our final answer.

e. [1 point] If we scale all dimensions in our initial centroids and data points by some factor, are we guaranteed to retrieve the same clusters after running k-means (i.e. will the same data points belong to the same cluster before and after scaling)? What if we scale only certain dimensions? If your answer is yes, provide a short explanation. If it is no, provide a counterexample.

True if we scale all dimensions since the scaling factor can be factored out of the argmin equation and disregarded since it is applied equally to all possible centroids.

$$z_i = \arg \min_{z \in \{1,..,k\}} \|\,\alpha\phi(x_i) - \alpha\mu_z\|^2 = \arg \min_{z \in \{1,..,k\}} \alpha^2 \|\,\phi(x_i) - \mu_z\|^2 = \arg \min_{z \in \{1,..,k\}} \|\,\phi(x_i) - \mu_z\|^2$$

False if we scale only certain dimensions. Here is a counterexample:

1. $\phi(x_1) = [1, 0]$
2. $\phi(x_2) = [0, 1]$
3. $\phi(x_3) = [100, 0]$
4. $\phi(x_4) = [100, 100]$

With initial centroids $\mu_1 = [1, 1]$ and $\mu_2 = [100, 50]$, x_1 and x_2 belong to μ_1 while x_3 and x_4 belong to μ_2. If we divide the first dimension of the data points and centroids by 100, x_1, x_2, and x_3 now belong to μ_1 while x_4 now belongs to μ_4.