COMS 4771 Machine Learning *Fall 2023*

Homework 3

Due October 20, 2023 at 11:59 PM

Instructions for preparing and submitting your homework write-up are available here:
`https://www.cs.columbia.edu/~djhsu/coms4771-f23/policies-homework.html`

Please submit your Python code for Problems 1–4 on Gradescope.

Linear classifiers for text

The Yelp restaurant reviews dataset is comprised of user-contributed reviews posted to Yelp for restaurants in Pittsburgh collected several years ago. Each review is accompanied by a binary indicator of whether or not the reviewer-assigned rating is at least four (on a five-point scale). The text of the reviews has been processed to replace all non-alphanumeric symbols with whitespace, and all letters have been changed to lowercase. The prediction problem we consider here is predicting the binary indicator from the review text.

The training data is contained in the file `reviews_tr.csv`, and the test data is contained in the file `reviews_te.csv`. The following code can be used to load the training data.

```python
from csv import DictReader

vocab = {}
vocab_size = 0
examples = []
with open('reviews_tr.csv', 'r') as f:
    reader = DictReader(f)
    for row in reader:
        label = row['rating'] == '1'
        words = row['text'].split(' ')
        for word in words:
            if word not in vocab:
                vocab[word] = vocab_size
                vocab_size += 1
        examples.append((label, [vocab[word] for word in words]))
```

This code assigns every word that appears in training data a unique non-negative integer, and the mapping is provided in the dictionary `vocab`. Each training example is represented as a binary label paired with a list of non-negative integers (which we'll call "word IDs") corresponding to the words that appear in the review. Note that the list may contain repetitions of any given word ID (since a review may contain repetitions of a word). The examples are collected in the array `examples`. Throughout the rest of this section, we'll refer to the variables defined by this code.

Of course, the test data can be loaded in a similar way. Note that the test data may contain words that do not appear in the training data.

Different examples may have different number of words, so the lists of word IDs may have different lengths. It may not be obvious how they can be used with common machine learning algorithms. However, it is relatively easy to transform a list of word IDs into a "bag-of-words" vector (also called "term frequency" vector), which is a vector $x = (x_0, \ldots, x_{d-1})$ where x_j is the number of times word ID j appears in the list. **This is the representation you should use for the feature vectors throughout this section.** The dimension d of the vector is equal to the vocabulary size `vocab_size` (and the vector indices, like the word IDs, start from 0). Such a vector representation is readily usable by many machine learning algorithms.

The following code applies this transformation to the first training example.

```python
from numpy import zeros

def bag_of_words_rep(word_ids, dim):
    bow_vector = zeros(dim) # creates a numpy.ndarray of shape (dim,)
    for word_id in word_ids:
        bow_vector[word_id] += 1
    return bow_vector

first_bow_vector = bag_of_words_rep(examples[0][1], vocab_size)
```

Note that applying this transformation to all training examples would result in a large collection of high-dimensional vectors (each of dimension `vocab_size`), ultimately consuming a lot of memory:

```python
print('memory required for examples:', getsizeof(examples))
print('(estimate of) memory required for bag-of-words representation:',
↪  len(examples) * getsizeof(first_bow_vector))
```

The difference is quite a few orders-of-magnitude. Therefore, function `bag_of_words_rep` should not be used explicitly; its purpose here is just to explain the semantics of the "bag-of-words" representation. A memory-efficient learning algorithm should use `examples` directly as input and avoid explicitly forming the "bag-of-words" vectors for these examples.

Online Perceptron

The Online Perceptron algorithm is a variant of the standard Perceptron algorithm. Recall that the standard Perceptron algorithm maintains a weight vector w, and repeatedly updates the weight vector with any training example that is misclassified by the (homogeneous) linear classifier with parameter w. This continues until there are no misclassified training examples. Of course, if the training data is not linearly separable, this will go on forever—some examples will be used to update the weight vector infinitely often. The Online Perceptron, shown below, rectifies this by only considering each training example exactly once, in some fixed ordering over the training examples, and then halting.

- Input: training examples $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})$

```
• Initialize: $w^{(0)} = (0, \dots, 0)$
• For $i = 1, 2, \dots, n$:
    – If $(x^{(i)}, y^{(i)})$ is misclassified by the linear classifier with parameter $w^{(i-1)}$, then
        $w^{(i)} \leftarrow \begin{cases} w^{(i-1)} + x^{(i)} & \text{if } y^{(i)} = \text{true} \\ w^{(i-1)} - x^{(i)} & \text{if } y^{(i)} = \text{false} \end{cases}$
    – Else $w^{(i)} \leftarrow w^{(i-1)}$
• Return: $w^{(n)}$
```

Here, we have assumed that each label is either "true" or "false", and that a linear classifier with parameter w predicts "true" on an input feature vector x if and only if $w^{\intercal}x > 0$.

Problem 1. Implement the Online Perceptron algorithm and apply it to the training data (in the order that they appear in `reviews_tr.csv`). Briefly explain how you avoid explicitly forming the bag-of-words feature vectors in your implementation. What is the training error rate of the linear classifier returned by Online Perceptron? And what is the test error rate (based on the test data from `reviews_te.csv`)? Please report three significant digits for all computed error rates.

Recall, that the test data may contain words that do not appear in the training data. It may seem that this could be cumbersome to deal with. However, it turns out to be easy to handle because the initial weight vector in Online Perceptron is the zero vector.

Problem 2. To get a sense of the behavior of the linear classifier that you obtained above, determine the 10 words that have the highest (i.e., most positive) weights in the weight vector, and also determine the 10 words that have the lowest (i.e., most negative) weights in the weight vector. Report the actual words, not the word IDs. You may find it helpful to invert the mapping given in `vocab`.

Upgrades

There are many ways to upgrade Online Perceptron. Below, we discuss two possibilities.

The first is the "Averaged Perceptron" variant, which is exactly the same as Online Perceptron, except that the weight vector returned is $w^{\text{avg}} := \frac{1}{n}\sum_{i=1}^{n} w^{(i)}$ (instead of just $w^{(n)}$). This is more "stable" than Online Perceptron, since the average of the weight vectors is much less sensitive to a single training example than the current weight vector maintained by Online Perceptron. (Of course, $w^{\text{sum}} := \sum_{i=1}^{n} w^{(i)}$ works just as well.)

The second is to improve the feature representation. So far, we have only used the bag-of-words representation of the reviews. It is called bag-of-words because it only depends on the number of times words appear in the review; the order of the words doesn't matter at all. One way to take the order of words into account (in a very limited way) is to also consider the number of times "bigrams" appear in the review. A bigram is a pair of words that appear consecutively. For example, in "a rose is a rose", the bigrams that appear are: (a, rose), which appears twice; (rose, is); and (is, a). If there are d words in a vocabulary, then there are d^2 possible bigrams, which can be enormous when d is even just moderately large. This makes it even more important to avoid explicitly forming the bag-of-words-and-bigrams feature vectors.

Problem 3 (Optional). Implement one of the above suggested improvements, or any other improvement you can think of. (If you come up with your own improvement, give a high-level description of it and explain its motivation.) Apply the new algorithm to the training data. What are the training and test error rates of the classifier you obtain? Which 10 words (or bigrams) have the highest weights in the weight vector (à la Problem 2), and which have the lowest weights?

Freedman's paradox

Load the training dataset `freedman.pkl` as follows:

```python
import pickle
freedman = pickle.load(open('freedman.pkl', 'rb'))
```

In `freedman['data']`, there is an $n \times d$ matrix whose rows correspond to n feature vectors $x^{(1)}, \dots, x^{(n)} \in \mathbb{R}^d$. In `freedman['labels']`, we have the corresponding labels $y^{(1)}, \dots, y^{(n)} \in \mathbb{R}$. Features and labels are *approximately* standardized (i.e., mean zero and variance one).

Suppose we are interested in predicting the label from the feature vectors using a (homogeneous) linear function. Note that $n = 100$ and $d = 1000$, so we don't expect ordinary least squares to work well on this dataset. But we might hope that ordinary least squares will work well if we only consider a small number of features. So consider the following three-step procedure:

- (Step 1.) Estimate the correlations between the features and the label:

$$\hat{\rho}_j := \frac{1}{n} \sum_{i=1}^{n} x_j^{(i)} y^{(i)} \quad \text{for } j = 1, \dots, d.$$

  (This is a reasonable estimate since the features and labels are approximately standardized.)

- (Step 2.) Let $\widehat{J}$ be the set of features (well, technically the indices of these features) for which the estimated correlation is larger than $2/\sqrt{n}$:

$$\widehat{J} := \{ j \in \{1, \dots, d\} : |\hat{\rho}_j| > 2/\sqrt{n} \}.$$

- (Step 3.) Now construct the ordinary least squares estimate $\hat{w} \in \mathbb{R}^d$ using only features in $\widehat{J}$. In other words,

$$\hat{w} \in \underset{\substack{w \in \mathbb{R}^d \text{ such that} \\ w_j = 0 \text{ for all } j \notin \widehat{J}}}{\arg\min} \frac{1}{n} \sum_{i=1}^{n} (w^\top x^{(i)} - y^{(i)})^2.$$

  This is equivalent to removing the columns of `freedman['data']` that are *not* in $\widehat{J}$, and then applying ordinary least squares with the resulting dataset.

The first two steps comprise a "screening procedure" whose purpose is to remove irrelevant features. If the number of features $|\widehat{J}|$ remaining after the screening is much smaller than n, then one might think that ordinary least squares (using only features in $\widehat{J}$) will work well; this is the motivation for Step 3.

Problem 4.

(a) Carry-out the procedure described above on the given training data. How many features are in $\widehat{J}$ after Step 2? It should be much smaller than n.

(b) What is the empirical risk of $\hat{w}$ after Step 3, i.e.,

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{w}^{\mathsf{T}}x^{(i)} - y^{(i)})^2?$$

It should be much smaller than the variance of the labels on this dataset (which is close to 1).

(c) A separate test set is available in `freedman['testdata']`, `freedman['testlabels']`. What is the risk of $\hat{w}$ on this test set? (This can be regarded as "test risk" of $\hat{w}$.)

(d) It turns out that all features and labels are drawn independently from $\mathrm{N}(0,1)$—i.e., there is no real correlation between the features and labels. Yet, based on the relatively small empirical risk of $\hat{w}$, together with the small number of features used in $\hat{w}$ relative to the sample size n, one might have thought that there *is* a relationship between the features and labels! This apparent paradox is called "Freedman's paradox". Briefly explain why this happens.

Optional: What is the "true" risk of $\hat{w}$? The answer can be expressed as a certain function of $\hat{w}$.

(e) Suppose Step 3 was modified to use an independent and identically distributed dataset (instead of the same dataset used in Steps 1 and 2). In fact, `freedman['data2']` and `freedman['labels2']` can be used for this purpose. So $\widehat{J}$ is determined using the first dataset, but $\hat{w}$ is now obtained using the second dataset. Do you think the empirical risk (on the second dataset) of the resulting weight vector $\hat{w}$ will be higher or lower than the answer from Part (b)? Please explain your answer. (Try to come up with an answer and explanation before trying it out using the data.)

Please report three significant digits for all computed risks.

Linear algebra, linear classifiers, and feature maps

Problem 5.

(a) Explain why, for every $n \times d$ matrix A, we have $\mathsf{NS}(A) = \mathsf{NS}(A^\intercal A)$.

(b) Explain why, for every $n \times d$ matrix A, we have $\mathsf{CS}(A) = \mathsf{CS}(AA^\intercal)$.

(Here, $\mathsf{CS}(M)$ denotes the column space of M, and $\mathsf{NS}(M)$ denotes the nullspace of M.)

Problem 6. Let $\mathrm{LL}\colon \mathbb{R}^d \to \mathbb{R}$ be the log-likelihood function based on training data $(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})$ from $\mathbb{R}^d \times \{0,1\}$ under the logistic regression model:

$$\mathrm{LL}(w) = \sum_{i=1}^{n} \Big(y^{(i)} w^\intercal x^{(i)} - \ln(1 + e^{w^\intercal x^{(i)}}) \Big).$$

Suppose you have a weight vector $\hat{w} \in \mathbb{R}^d$ such that

$$\hat{w}^\intercal x^{(i)} > 0 \quad \text{if and only if} \quad y^{(i)} = 1, \quad \text{for all } i = 1, \dots, n.$$

A statistician friend asks you if $\hat{w}$ (approximately) maximizes the log-likelihood function. However, in a computing accident, you lose access to the training data and cannot compute $\mathrm{LL}(\hat{w})$. The system administrator is only able to recover two pieces of information about the training data for you: the number of training examples n, and $\gamma = \min\{|\hat{w}^\intercal x^{(1)}|, \dots, |\hat{w}^\intercal x^{(n)}|\}$. Write a procedure that, given $\hat{w}$, n, γ, and an arbitrary positive number $\epsilon > 0$, returns another weight vector $v \in \mathbb{R}^d$ such that $\mathrm{LL}(v) \geq -\epsilon$. Explain why your procedure works.

Problem 7. Specify a feature map $\varphi\colon \mathbb{R}^d \to \mathbb{R}^p$, with p as small as possible, so that homogeneous linear classifiers in the feature space are able to represent all binary classifiers $f_{c,r}\colon \mathbb{R}^d \to \{0,1\}$ of the following form:

$$f_{c,r}(x) = \begin{cases} 1 & \text{if } \|x - c\| < r \\ 0 & \text{otherwise.} \end{cases}$$

Here, $c \in \mathbb{R}^d$ and $r > 0$ are parameters of the classifier. In other words, for every $c \in \mathbb{R}^d$ and $r > 0$, there should exist $w \in \mathbb{R}^p$ such that

$$\varphi(x)^\intercal w > 0 \quad \text{if and only if} \quad f_{c,r}(x) = 1, \quad \text{for all } x \in \mathbb{R}^d.$$

Explain why your feature map works.