Homework 3: SVM and Sentiment Analysis

1 Introduction

In this assignment, we'll be working with natural language data. In particular, we'll be doing sentiment analysis on movie reviews. This problem will give you the opportunity to try your hand at feature engineering, which is one of the most important parts of many data science problems. From a technical standpoint, this homework has two new pieces. First, you'll be implementing Pegasos. Pegasos is essentially stochastic subgradient descent for the SVM with a particular schedule for the step-size. Second, because in natural langauge domains we typically have huge feature spaces, we work with sparse representations of feature vectors, where only the non-zero entries are explicitly recorded. This will require coding your gradient and SGD code using hash tables (dictionaries in Python), rather than numpy arrays. We begin with some practice with subgradients and an easy problem that introduces the Perceptron algorithm.

2 Calculating Subgradients

Recall that a vector $g \in \mathfrak{R}^d$ is a $\textbf{subgradient}$ of $f : \mathfrak{R}^d \to \mathfrak{R}$ at x if for all z,

$$f(z) \geq f(x) + g^T(z - x).$$

As we noted in lecture, there may be 0, 1, or infinitely many subgradients at any point. The $\textbf{subdifferential}$ of f at a point x, denoted $\partial f(x)$, is the set of all subgradients of f at x.

1. Subgradients for pointwise maximum of functions. Suppose $f_1, \ldots, f_m : \mathfrak{R}^d \to \mathfrak{R}$ are convex functions, and

$$f(x) = \max_{i=1,\ldots,m} f_i(x).$$

Let k be any index for which $f_k(x) = f(x)$, and choose $g \in \partial f_k(x)$. We are using the fact that a convex function on $\mathfrak{R}^d$ has a non-empty subdifferential at all points. Show that $g \in \partial f(x)$.

Answer: We can find $f(x) = f_k(x)$ for any exsiting k, let $g \in \partial f_k(x)$ at point x, and any $f_k(z) \geq f_k(x) + g^T(z - x)$, so $f(z) \geq f_k(z) \geq f_k(x) + g^T(z - x)$, so g must be $g \in \partial f(x)$.

1. Subgradient of hinge loss for linear prediction. Give a subgradient of

$$J(w) = \max\left\{0, 1 - yw^Tx\right\}.$$

Answer: compute the subgradient of hinge loss function

$$\partial J(w) = \begin{cases} -1 & \text{if } yw^T x < 1 \\ 0 & \text{if } yw^T x > 1 \\ [-1, 0] & \text{if } yw^T x = 1 \end{cases} .$$

3 Perceptron

The perceptron algorithm is often the first classification algorithm taught in machine learning classes. Suppose we have a labeled training set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathfrak{R}^d \times \{-1, 1\}$ In the perceptron algorithm, we are looking for a hyperplane that perfectly separates the classes. That is, we're looking for $w \in \mathfrak{R}^d$ such that

$$y_i w^T x_i > 0 \; \forall i \in \{1, \ldots, n\} .$$

Visually, this would mean that all the x's with label $y = 1$ are on one side of the hyperplane $\left\{x \mid w^T x = 0\right\}$, and all the $x's$ with label $y = -1$ are on the other side. When such a hyperplane exists, we say that the data are **linearly separable**. The perceptron algorithm is given in Algorithm 1

Algorithm 1: Perceptron Algorithm

```
input: Training set $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbf{R}^d \times \{-1, 1\}$
$w^{(0)} = (0, \ldots, 0) \in \mathbf{R}^d$
$k = 0$ # step number
repeat
  all_correct = TRUE
  for $i = 1, 2, \ldots, n$ # loop through data
    if ($y_i x_i^T w^{(k)} \leq 0$)
      $w^{(k+1)} = w^{(k)} + y_i x_i$
      all_correct = FALSE
    else
      $w^{(k+1)} = w^{(k)}$
    end if
    $k = k + 1$
  end for
until (all_correct == TRUE)
return $w^{(k)}$
```

.

There is also something called the **perceptron loss,** given by

$$\ell(\hat{y}, y) = \max\left\{0, -\hat{y}y\right\} .$$

1. Show that if $\left\{x \mid w^T x = 0\right\}$ is a separating hyperplane for a training set $D = ((x_1, y_1), \ldots, (x_n, y_n))$ then the average perceptron loss on D is 0. Thus any separating hyperplane of D is an empirical risk minimizer for perceptron loss.

Answer: if $\left\{x \mid w^T x = 0\right\}$ is a separating hyperplane for a training set $D = ((x_1, y_1), \ldots, (x_n, y_n))$ we always have $y_i w^T x_i > 0 \ \forall i \in \{1, \ldots, n\}$, which means that $\hat{y}y > 0$ always. Then the perceptron loss for each training example is 0.

1. Let H be the linear hypothesis space consisting of functions $x \mapsto w^T x$. Consider running stochastic subgradient descent (SSGD) to minimize the empirical risk with the perceptron loss. We'll use the version of SSGD in which we cycle through the data points in each epoch. Show that if we use a fixed step size 1, we terminate when our training data are separated, and we make the right choice of subgradient, then we are exactly doing the Perceptron algorithm.

Answer: The loss function

$$J(w) = \max\left\{0, -\hat{y}y\right\} \begin{cases} 0 & \hat{y}y > 0 \\ -\hat{y}y & \hat{y}y < 0 \end{cases}$$

When trying to run stochatic subgradient descent, update gradient for each example, where $\alpha = 1$ as step size.

$$w = w - \alpha * \partial J(w) = \begin{cases} w & \hat{y}y > 0 \\ w + \alpha y_i x_i & \hat{y}y < 0 \end{cases}$$

Then we can see that it is the same as perception algorithm.

1. Suppose the perceptron algorithm returns w. Show that w is a linear combination of the input points. That is, we can write $w = \sum_{i=1}^{n} \alpha_i x_i$ for some $\alpha_1, \ldots, \alpha_n \in \mathfrak{R}$. The x_i for which $\alpha_i \neq 0$ are called support vectors. Give a characterization of points that are support vectors and not support vectors.

Answer: In the training, if current $w^T x$ cannot classify correctly for one example, we will add $y_i x_i$ to the w, until we classify all the examples correctly. Thus w will the linear combination of x_i. During training, the training examples that classify incorrectly are considered as support vectors, and the ones that classify correctly are consider as non support vectors.

4 The Data

We will be using the Polarity Dataset v2.0 (https://www.cs.cornell.edu/people/pabo/movie-review-data), constructed by Pang and Lee. It has the full text from 2000 movies reviews: 1000 reviews are classified as `positive` and 1000 as `negative.` Our goal is to predict whether a review has positive or negative sentiment from the text of the review. Each review is stored in a separate file: the positive reviews are in a folder called `pos`, and the negative reviews are in `neg`. We have provided some code in `load.py` to assist with reading these files. You can use the code, or write your own version. The code removes some special symbols from the reviews. Later you can check if this helps or hurts your results.

1. Load all the data and randomly split it into 1500 training examples and 500 validation examples.

In [93]:
```
# from load import shuffle_data
# # Save data into 'save.p' file
# shuffle_data()
```

In [94]:
```
import pickle

dataset = pickle.load(open("save.p", "rb"))

num_train = 1500
train_examples = dataset[:num_train]
valid_examples = dataset[num_train:]
# print(len(train_examples))
# print(len(valid_examples))
# print(train_examples[0])
```

5 Sparse Representations

The most basic way to represent text documents for machine learning is with a `bag-of-words` representation. Here every possible word is a feature, and the value of a word feature is the number of times that word appears in the document. Of course, most words will not appear in any particular document, and those counts will be zero. Rather than store a huge number of zeros, we use a sparse representation, in which we only store the counts that are nonzero. The counts are stored in a key/value store (such as a dictionary in Python). For example, `Harry Potter and Harry Potter II` would be represented as the following Python dict: $\text{x=\{'Harry':2, 'Potter':2, 'and':1, 'II':1\}}$. We will be using linear classifiers of the form $f(x) = w^T x$, and we can store the w vector in a sparse format as well, such as $\text{w=\{'minimal':1.3, 'Harry':-1.1, 'viable':-4.2, 'and':2.2, 'product':9.1\}}$. The inner product between w and x would only involve the features that appear in both x and w, since whatever doesn't appear is assumed to be zero. For this example, the inner product would be $\text{x[Harry] * w[Harry] + x[and] * w[and] = 2 * (-1.1) + 1 * (2.2)}$. To help you along, we've included two functions for working with sparse vectors: 1) a dot product between two vectors represented as dict's and 2) a function that increments one sparse vector by a scaled multiple of another vector, which is a very common operation. These functions are located in util.py. It is worth reading the code, even if you intend to implement it yourself. You may get some ideas on how to make things faster.

1. Write a function that converts an example (e.g. a list of words) into a sparse bag-of-words representation. You may find Python's Counter class to be useful here: url (https://docs.python.org/2/library/collections.html). Note that a Counter is also a dict.

In [2]:
```
from collections import Counter

def convertDoc2Sparse(doc):
    cnt = Counter()
    for w in doc:
        cnt[w] += 1
    return cnt

# Test doc sparse representation
docSparseRep = convertDoc2Sparse(train_examples[0][:-1])
print(docSparseRep)
```

```
Counter({'the': 21, 'a': 17, 'and': 15, 'for': 11, 'of': 10, 'but': 10,
'to': 9, 'some': 8, 'have': 8, 'in': 8, 'is': 8, 'compensate': 6, 'yo
u': 6, 'who': 5, 'this': 5, 'like': 5, 'an': 5, 'thornton': 4, 'cusac
k': 4, 'it': 4, 'that': 4, 'cast': 3, 'can': 3, 'pushing': 3, 'tin': 3,
'blanchett': 3, 'jolie': 3, 'has': 3, 'hip': 3, 'be': 3, 'film': 3, 'ai
r': 3, 'with': 3, '!': 3, 'sometimes': 2, 'things': 2, 'are': 2, 'oh':
2, 'yes': 2, 'might': 2, 'not': 2, 'at': 2, 'people': 2, 'terrific': 2,
'score': 2, '?': 2, 'planes': 2, 'us': 2, 'how': 2, 'best': 2, 'i': 2,
'one': 2, 'so': 2, 'traffic': 2, 'controllers': 2, 'these': 2, 'falzon
e': 2, 'up': 2, 'boys': 2, 'will': 2, 'then': 2, "doesn't": 2, 'there':
2, 'wife': 2, 'last': 2, 'newell': 2, 'make': 2, "they'll": 2, 'minute
s': 2, 'herself': 2, 'by': 2, 'fine': 2, 'actress': 2, 'his': 2, 'too':
2, 'stellar': 1, 'lot': 1, 'certainly': 1, 'features': 1, 'name': 1, 's
tars': 1, 'going': 1, 'places': 1, 'billy': 1, 'bob': 1, 'cate': 1, 'an
gelina': 1, 'john': 1, 'realize': 1, 'first': 1, "he's": 1, 'actually':
1, 'veteran': 1, 'among': 1, 'quartet': 1, 'finelooking': 1, 'lackluste
r': 1, 'screen': 1, 'treatment': 1, 'idea': 1, 'comedy': 1, 'written':
1, 'all': 1, 'over': 1, 'workmanlike': 1, 'uninspired': 1, 'direction':
1, 'obnoxious': 1, 'would': 1, 'anyone': 1, 'tone': 1, 'deaf': 1, 'scre
aming': 1, 'exits': 1, 'clich': 1, 'd': 1, 'characterizations': 1, 'emb
arrassing': 1, 'joking': 1, 'situations': 1, 'etc': 1, "don't": 1, 'ear
thly': 1, 'from': 1, 'opening': 1, 'sequence': 1, 'big': 1, 'trouble':
1, 'squiggly': 1, 'quirky': 1, 'credits': 1, 'fakelooking': 1, 'passeng
er': 1, 'circling': 1, 'new': 1, 'york': 1, 'anne': 1, "dudley's": 1,
'inyourear': 1, 'music': 1, 'making': 1, 'wonder': 1, 'she': 1, 'ever':
1, 'got': 1, 'original': 1, 'nomination': 1, 'full': 1, 'monty': 1, 'le
t': 1, 'alone': 1, 'won': 1, "wasn't": 1, 'ready': 1, 'walk': 1, 'jus
t': 1, 'yet': 1, 'quickly': 1, 'we': 1, 'descend': 1, 'into': 1, 'tight
lyedited': 1, 'montage': 1, 'which': 1, 'screams': 1, 'large': 1, 'capi
tal': 1, 'letters': 1, 'difficult': 1, 'job': 1, 'what': 1, 'their': 1,
'frantic': 1, 'mileaminute': 1, 'instructional': 1, 'personas': 1, 'jug
gling': 1, "passenger's": 1, 'lives': 1, 'huge': 1, 'real': 1, 'midai
r': 1, 'video': 1, 'game': 1, 'cool': 1, 'demonic': 1, 'auctioneer': 1,
'nick': 1, 'zone': 1, 'biz': 1, 'course': 1, 'until': 1, 'hipper': 1,
'cooler': 1, 'leatherclad': 1, 'flyboy': 1, 'assist': 1, 'guise': 1, 'r
ussell': 1, 'bell': 1, 'shows': 1, 'challenge': 1, "falzone's": 1, 'fin
ite': 1, 'space': 1, 'heavy': 1, 'duty': 1, 'testosterone': 1, 'start
s': 1, 'exuding': 1, 'macho': 1, 'oneupmanship': 1, 'begins': 1, 'sto
p': 1, 'seeing': 1, 'juggle': 1, 'three': 1, '747s': 1, 'within': 1, "c
at's": 1, 'whisker': 1, 'each': 1, 'other': 1, 'no': 1, 'broken': 1, 'h
oop': 1, 'dreams': 1, 'wannaseehowfasticandrives': 1, 'ultimate': 1, 's
howdown': 1, 'was': 1, 'my': 1, 'saw': 1, 'night': 1, 'director': 1, 'm
ike': 1, 'four': 1, 'weddings': 1, 'funeral': 1, 'must': 1, 'read': 1,
'different': 1, 'draft': 1, 'script': 1, 'because': 1, "that's": 1, 'be
ing': 1, 'acted': 1, 'out': 1, 'between': 1, 'newark': 1, 'jfk': 1, 'l
a': 1, 'guardia': 1, 'ounce': 1, 'subtlety': 1, 'made': 1, 'awfully':
1, 'goodand': 1, 'funnymovies': 1, 'before': 1, 'antics': 1, 'cringe':
1, 'frown': 1, 'disbelief': 1, 'constantly': 1, 'looking': 1, 'your':
1, 'watch': 1, 'wait': 1, "there's": 1, 'still': 1, '100': 1, 'go': 1,
"film's": 1, 'only': 1, 'saving': 1, 'grace': 1, 'whose': 1, 'connie':
1, 'spunky': 1, 'brash': 1, 'long': 1, 'island': 1, 'housewife': 1, 'wa
nts': 1, 'better': 1, 'taking': 1, 'art': 1, 'classes': 1, 'wonderful':
1, 'accomplishment': 1, 'previously': 1, 'played': 1, 'redheaded': 1,
'australian': 1, 'gambler': 1, 'oscar': 1, 'lucinda': 1, 'tempestuous':
1, 'british': 1, 'monarch': 1, 'elizabeth': 1, "she's": 1, 'enough': 1,
'save': 1, 'picture': 1, 'looks': 1, 'performs': 1, 'solidly': 1, 'char
acter': 1, 'joke': 1, 'as': 1, "russell's": 1, 'knock': 1, "'em": 1, 'd
ead': 1, "isn't": 1, 'bad': 1, 'upandcoming': 1, 'disappoints': 1, 'all
owing': 1, 'displayed': 1, 'plaything': 1, 'cracks': 1, 'gum': 1, 'don
s': 1, 'shades': 1, 'acts': 1, 'throughout': 1, 'everything': 1, 'els
e': 1, 'performance': 1, 'forced': 1, 'ten': 1, 'or': 1, 'inexplicabl
e': 1, 'reason': 1, 'start': 1, 'coming': 1, 'together': 1, 'begin': 1,
'get': 1, 'sense': 1, 'been': 1, 'trailer': 1, 'teases': 1, "it's": 1,
'little': 1, 'late': 1, 'aside': 1, 'nothing': 1, 'more': 1, 'than': 1,
'embarrassment': 1})
```

6 Support Vector Machine via Pegasos

In this question you will build an SVM using the Pegasos algorithm. To align with the notation used in the Pegasos Pegasos: Primal Estimated sub-GrAdient SOlver for SVM (http://ttic.uchicago.edu/~nati/Publications/PegasosMPB.pdf), we're considering the following formulation of the SVM objective function:

$$\min_{w\in\Re^d} \frac{\lambda}{2}\|w\|^2 + \frac{1}{m}\sum_{i=1}^{m} \max\left\{0, 1 - y_i w^T x_i\right\}.$$

Note that, for simplicity, we are leaving off the unregularized bias term b. Pegasos is stochastic subgradient descent using a step size rule $\eta_t = 1/(\lambda t)$. The pseudocode is given below:

```
Input: $\lambda > 0$. Choose $w_1 = 0, t = 0$
While termination condition not met
  For $j = 1, \ldots, m$ (assumes data is randomly permuted)
    $t = t + 1$
    $\eta_t = 1/(t\lambda)$;
    If $y_j w_t^T x_j < 1$
      $w_{t+1} = (1 - \eta_t\lambda)w_t + \eta_t y_j x_j$
    Else
      $w_{t+1} = (1 - \eta_t\lambda)w_t$
```

1. (Written). Consider the `stochastic` SVM objective function, which is the SVM objective function with a single training point (Recall that if i is selected uniformly from the set $\{1, \ldots, m\}$, then this stochastic objective function has the same expected value as the full SVM objective function.): $J_i(w) = \frac{\lambda}{2}\|w\|^2 + \max\left\{0, 1 - y_i w^T x_i\right\}$. The function $J_i(\theta)$ is not differentiable everywhere. Give an expression for the gradient of $J_i(w)$ where it's defined, and specify where it is not defined.

Answer:

$$\nabla J_i(w) = \begin{cases} \lambda w - y_i x_i & \text{for } y_i w^T x_i < 1 \\ \lambda w & \text{for } y_i w^T x_i > 1. \\ \text{undefined} & \text{for } y_i w^T x_i = 1 \end{cases}$$

1. (Written) Show that a subgradient of $J_i(w)$ is given by
$$g = \begin{cases} \lambda w - y_i x_i & \text{for } y_i w^T x_i < 1 \\ \lambda w & \text{for } y_i w^T x_i \geq 1. \end{cases}$$

You may use the following facts without proof: 1) If $f_1, \dots, f_m : \mathfrak{R}^d \to \mathfrak{R}$ are convex functions and $f = f_1 + \cdots + f_m$, then $\partial f(x) = \partial f_1(x) + \cdots + \partial f_m(x)$. 2) For $\alpha \geq 0$, $\partial(\alpha f)(x) = \alpha \partial f(x)$. (Hint: Use the rules provided and the calculation in the first problem.)

Answer: Since we have gradient at all points except $y_i w^T x_i = 1$, so we only consider the subgradient at this point. The subgradient at $m = 1$ in the hinge loss function $max\{0, 1 - m\}$ is $[-1, 0]$, thus the subgradient for $\partial J_i(w)$ can be $\lambda w + k$ at $y_i w^T x_i = 1$, where k is in $[-1, 0]$.

1. (Written). Show that if your step size rule is $\eta_t = 1/(\lambda t)$, then doing SGD with the subgradient direction from the previous problem is the same as given in the pseudocode.

Answer: If we use SGD with the subgradient direction, for each example (x_i, y_i), the subgradient descent is $w = w - \alpha * g$, where g is subgradient is question 2, α is step size as $\eta_t = 1/(\lambda t)$. Thus the whole algorithm is the same as the given pseudocode.

1. Implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector w. Note that our Pegasos algorithm starts at $w = 0$. In a sparse representation, this corresponds to an empty dictionary. **Note:** With this problem, you will need to take some care to code things efficiently. In particular, be aware that making copies of the weight dictionary can slow down your code significantly. If you want to make a copy of your weights (e.g. for checking for convergence), make sure you don't do this more than once per epoch. **Also**: If you normalize your data in some way, be sure not to destroy the sparsity of your data. Anything that starts as 0 should stay at 0.

In [21]:
```
# 4. Implement the Pegasos algorithm
import numpy as np
from util import dotProduct, increment, scaleProduct

def svmPegasos(train_examples, regularized_term = 1.0, num_epoch = 10):
    num_train = len(train_examples)
    w = Counter()
    t = 0

    for epoch in range(num_epoch):
        # shuffle training data
        np.random.shuffle(train_examples)
        for idx in range(num_train):
            t = t + 1
            eta = 1.0/(t*regularized_term)
            xi = convertDoc2Sparse(train_examples[idx][:-1])
            yi = train_examples[idx][-1]
            margin = yi * dotProduct(w, xi)
            scaleProduct(w, 1-eta*regularized_term)
            if margin < 1:
                increment(w, eta*yi, xi)
        print("Running epoch # {} in Pegasos algorithm".format(epoch))
    return w
```

1. Note that in every step of the Pegasos algorithm, we rescale every entry of w_t by the factor $(1 - \eta_t \lambda)$. Implementing this directly with dictionaries is very slow. We can make things significantly faster by representing w as $w = sW$, where $s \in \mathfrak{R}$ and $W \in \mathfrak{R}^d$. You can start with $s = 1$ and W all zeros (i.e. an empty dictionary). Note that both updates (i.e. whether or not we have a margin error) start with rescaling w_t, which we can do simply by setting $s_{t+1} = (1 - \eta_t \lambda)\, s_t$. If the update is $w_{t+1} = (1 - \eta_t \lambda) w_t + \eta_t y_j x_j$, then **verify that the Pegasos update step is equivalent to**:
$$s_{t+1} = (1 - \eta_t \lambda)\, s_t$$
$$W_{t+1} = W_t + \frac{1}{s_{t+1}} \eta_t y_j x_j.$$
There is one subtle issue with the approach described above: if we ever have $1 - \eta_t \lambda = 0$, then $s_{t+1} = 0$, and we'll have a divide by 0 in the calculation for W_{t+1}. This only happens when $\eta_t = 1/\lambda$. With our step-size rule of $\eta_t = 1/(\lambda t)$, it happens exactly when $t = 1$. So one approach is to just start at $t = 2$. More generically, note that if $s_{t+1} = 0$, then $w_{t+1} = 0$. Thus an equivalent representation is $s_{t+1} = 1$ and $W = 0$. Thus if we ever get $s_{t+1} = 0$, simply set it back to 1 and reset W_{t+1} to zero, which is an empty dictionary in a sparse representation.
Implement the Pegasos algorithm with the (s, W) representation described above. (See section 5.1 of Leon Bottou's Stochastic Gradient Tricks (http://leon.bottou.org/papers/bottou-tricks-2012) for a more generic version of this technique, and many other useful tricks.)

In [95]:
```
# Implement the Pegasos algorithm with the (s,W) representation.
import numpy as np
from util import dotProduct, increment, scaleProduct

def svmPegasosV2(train_examples, regularized_term = 1.0, num_epoch = 10
):
    num_train = len(train_examples)
    W = Counter()
    t, s = 0, 1

    for epoch in range(num_epoch):
        # shuffle training data
        np.random.shuffle(train_examples)
        for idx in range(num_train):
            t = t + 1
            eta = 1.0/(t*regularized_term)
            s_next = (1-eta*regularized_term)* s
            if s_next == 0:
                s = 1
                W = Counter()
                continue
            xi = convertDoc2Sparse(train_examples[idx][:-1])
            yi = train_examples[idx][-1]
            margin = yi * s * dotProduct(W, xi)
            if margin < 1:
                increment(W, 1/s*eta*yi, xi)
            s = s_next
        if epoch % 50 == 0:
            print("Run epoch# {}/{} in fast Pegasos algorithm with regul
arized term {}".format(
                epoch, num_epoch, regularized_term))
    scaleProduct(W, s) # w = sW
    return W
```

1. Run both implementations of Pegasos on the training data for a couple epochs (using the bag-of-words feature representation described above). Make sure your implementations are correct by verifying that the two approaches give essentially the same result. Report on the time taken to run each approach.

In [96]:
```
import time

# Run the first Pegasos algorithm approach
# start_time = time.time()
# w1 = svmPegasos(train_examples, regularized_term = 1.0, num_epoch = 1)
# print("Total Pegasos algorithm execution time: {}s".format(time.time()
 - start_time))
# print(len(w1.keys()))

# Run the second Pegassos algorithm approach with the (s, W ) representa
tion
start_time = time.time()
w2 = svmPegasosV2(train_examples, regularized_term = 0.2, num_epoch = 30
0)
print("Total Pegasos algorithm V2 execution time: {}s".format(time.time
() - start_time))
# print(len(w2.keys()))

# print(w1)
# print(w2)
```

```
Run epoch# 0/300 in fast Pegasos algorithm with regularized term 0.2
Run epoch# 50/300 in fast Pegasos algorithm with regularized term 0.2
Run epoch# 100/300 in fast Pegasos algorithm with regularized term 0.2
Run epoch# 150/300 in fast Pegasos algorithm with regularized term 0.2
Run epoch# 200/300 in fast Pegasos algorithm with regularized term 0.2
Run epoch# 250/300 in fast Pegasos algorithm with regularized term 0.2
Total Pegasos algorithm V2 execution time: 205.28203797340393s
```

1. Write a function that takes a sparse weight vector w and a collection of (x, y) pairs, and returns the percent error when predicting y using $\text{sign}(w^T x)$. In other words, the function reports the 0-1 loss of the linear predictor $x \mapsto w^T x$.

In [54]:
```
# Predict the percent error
from util import dotProduct

def predictPercentError(data, w):
    error_count, total_count = 0, len(data)
    for idx in range(total_count):
        xi = convertDoc2Sparse(data[idx][:-1])
        yi = data[idx][-1]
        if yi * dotProduct(w, xi) < 0: # make wrong prediction
            error_count += 1
    return error_count/total_count

# Verify on validation set
errorRate = predictPercentError(valid_examples, w2)
print(errorRate)
```

```
0.152
```

1. Using the bag-of-words feature representation described above, search for the regularization parameter that gives the minimal percent error on your test set. (You should now use your faster Pegasos implementation, and run it to convergence.) A good search strategy is to start with a set of regularization parameters spanning a broad range of orders of magnitude. Then, continue to zoom in until you're convinced that additional search will not significantly improve your test performance. Once you have a sense of the general range of regularization parameters that give good results, you do not have to search over orders of magnitude every time you change something (such as adding a new feature).

In [52]:
```
# Hyper-parameter search

regularized_term_list = [10**2, 10, 1, 0.5, 0.2, 0.1, 0.05, 10**-2, 10**
-3]
# regularized_term_list = [1]
error_rate_list = []

for r in regularized_term_list:
    w = svmPegasosV2(train_examples, regularized_term = r, num_epoch = 2
00)
    error_rate = predictPercentError(valid_examples, w)
    error_rate_list.append(error_rate)

# Display result
for i in range(len(regularized_term_list)):
    print("regularized paramter {}: error rate is {}".format(
        regularized_term_list[i], error_rate_list[i]))
```

```
Run epoch# 0/200 in fast Pegasos algorithm with regularized term 100
Run epoch# 50/200 in fast Pegasos algorithm with regularized term 100
Run epoch# 100/200 in fast Pegasos algorithm with regularized term 100
Run epoch# 150/200 in fast Pegasos algorithm with regularized term 100
Run epoch# 0/200 in fast Pegasos algorithm with regularized term 10
Run epoch# 50/200 in fast Pegasos algorithm with regularized term 10
Run epoch# 100/200 in fast Pegasos algorithm with regularized term 10
Run epoch# 150/200 in fast Pegasos algorithm with regularized term 10
Run epoch# 0/200 in fast Pegasos algorithm with regularized term 1
Run epoch# 50/200 in fast Pegasos algorithm with regularized term 1
Run epoch# 100/200 in fast Pegasos algorithm with regularized term 1
Run epoch# 150/200 in fast Pegasos algorithm with regularized term 1
Run epoch# 0/200 in fast Pegasos algorithm with regularized term 0.5
Run epoch# 50/200 in fast Pegasos algorithm with regularized term 0.5
Run epoch# 100/200 in fast Pegasos algorithm with regularized term 0.5
Run epoch# 150/200 in fast Pegasos algorithm with regularized term 0.5
Run epoch# 0/200 in fast Pegasos algorithm with regularized term 0.2
Run epoch# 50/200 in fast Pegasos algorithm with regularized term 0.2
Run epoch# 100/200 in fast Pegasos algorithm with regularized term 0.2
Run epoch# 150/200 in fast Pegasos algorithm with regularized term 0.2
Run epoch# 0/200 in fast Pegasos algorithm with regularized term 0.1
Run epoch# 50/200 in fast Pegasos algorithm with regularized term 0.1
Run epoch# 100/200 in fast Pegasos algorithm with regularized term 0.1
Run epoch# 150/200 in fast Pegasos algorithm with regularized term 0.1
Run epoch# 0/200 in fast Pegasos algorithm with regularized term 0.05
Run epoch# 50/200 in fast Pegasos algorithm with regularized term 0.05
Run epoch# 100/200 in fast Pegasos algorithm with regularized term 0.05
Run epoch# 150/200 in fast Pegasos algorithm with regularized term 0.05
Run epoch# 0/200 in fast Pegasos algorithm with regularized term 0.01
Run epoch# 50/200 in fast Pegasos algorithm with regularized term 0.01
Run epoch# 100/200 in fast Pegasos algorithm with regularized term 0.01
Run epoch# 150/200 in fast Pegasos algorithm with regularized term 0.01
Run epoch# 0/200 in fast Pegasos algorithm with regularized term 0.001
Run epoch# 50/200 in fast Pegasos algorithm with regularized term 0.001
Run epoch# 100/200 in fast Pegasos algorithm with regularized term 0.00
1
Run epoch# 150/200 in fast Pegasos algorithm with regularized term 0.00
1
regularized paramter 100: error rate is 0.478
regularized paramter 10: error rate is 0.304
regularized paramter 1: error rate is 0.166
regularized paramter 0.5: error rate is 0.168
regularized paramter 0.2: error rate is 0.156
regularized paramter 0.1: error rate is 0.158
regularized paramter 0.05: error rate is 0.17
regularized paramter 0.01: error rate is 0.166
regularized paramter 0.001: error rate is 0.176
```

7 Error Analysis

The natural language processing domain is particularly nice in that one can often interpret why a model has performed well or poorly on a specific example, and sometimes it is not very difficult to come up with ideas for new features that might help fix a problem. The first step in this process is to look closely at the errors that our model makes.

1. Choose an input example $x = (x_1, \ldots, x_d) \in \mathfrak{R}^d$ that the model got wrong. We want to investigate what features contributed to this incorrect prediction. One way to rank the importance of the features to the decision is to sort them by the size of their contributions to the score. That is, for each feature we compute $|w_i x_i|$, where w_i is the weight of the ith feature in the prediction function, and x_i is the value of the ith feature in the input x. Create a table of the most important features, sorted by $|w_i x_i|$, including the feature name, the feature value x_i, the feature weight w_i, and the product $w_i x_i$. Attempt to explain why the model was incorrect. Can you think of a new feature that might be able to fix the issue? Include a short analysis for at least 2 incorrect examples.

In [91]:
```
import pandas as pd

# Create feature table
def createFeatTable(x, w):
    table = []
    for k,v in x.items():
        wi = w.get(k,0)
        table.append({'name': k, 'xi': v, 'wi': wi, 'wixi': wi*v, 'wixi_
abs': abs(wi*v)})
    return table

# Choose hyperparameter regularized term 0.2 as to analyze the error by
 running `svmPegasosV2` above.
# Find top 2 index in valid examples having prediction error
result = []
for i in range(len(valid_examples)):
    xi = convertDoc2Sparse(valid_examples[i][:-1])
    yi = valid_examples[i][-1]
    result.append({'val': yi * dotProduct(w2, xi), 'index': i})
result.sort(key=lambda x: x['val'])
# print(result)

# Find top two errors
# The first error
idx1 = result[0]['index']
x1 = convertDoc2Sparse(valid_examples[idx1][:-1])
y1 = valid_examples[idx1][-1]
print('The first example is {} label, predicted as {}'.format(y1, dotPro
duct(w2, x1)))
t1 = createFeatTable(x1, w2)
data1 = pd.DataFrame(t1)
data1 = data1.sort_values(by='wixi_abs', ascending=False)
data1.head(10)
```

```
The first example is -1 label, predicted as 2.0514856620204314
```

Out[91]:

	name	wi	wixi	wixi_abs	xi
135	and	0.025845	1.085489	1.085489	42
5	the	0.009806	0.902166	0.902166	92
4	i	0.018439	0.848181	0.848181	46
145	have	-0.055104	-0.440833	0.440833	8
168	why	-0.053836	-0.430686	0.430686	8
102	see	0.065249	0.391496	0.391496	6
82	to	-0.012099	-0.362983	0.362983	30
96	is	0.013218	0.343666	0.343666	26
119	as	0.033636	0.336362	0.336362	10
90	so	-0.031281	-0.312813	0.312813	10

Analysis: We only consider word count as features, the word 'and' appears two many times and the learned w_i is positive which pulls $w_i x_i$ to a positive large number. We can also see similar results on word 'the'. We might try to normalize the work count for each word to fix this issue or use tf-idf as feature instead.

In [92]:
```
# The second error
idx2 = result[1]['index']
x2 = convertDoc2Sparse(valid_examples[idx2][:-1])
y2 = valid_examples[idx2][-1]
print('The second example is {} label, predicted as {}'.format(y2, dotPr
oduct(w2, x2)))
t2 = createFeatTable(x2, w2)
data2 = pd.DataFrame(t2)
data2 = data2.sort_values(by='wixi_abs', ascending=False)
data2.head(10)
```

```
The second example is 1 label, predicted as -1.4796341005594358
```

Out[92]:

	name	wi	wixi	wixi_abs	xi
33	as	0.033636	0.403635	0.403635	12
70	the	0.009806	0.343215	0.343215	35
48	even	-0.051443	-0.308656	0.308656	6
36	and	0.025845	0.284295	0.284295	11
91	!	-0.030849	-0.277642	0.277642	9
38	any	-0.070071	-0.210212	0.210212	3
81	worst	-0.104383	-0.208765	0.208765	2
148	into	-0.042721	-0.170884	0.170884	4
51	i	0.018439	0.165949	0.165949	9
63	are	-0.031585	-0.157923	0.157923	5

Analysis: We can see the word 'worst' with a high weight learned from SVM to heavily pull down the predicted value. This is expected since we usually think that the movie will be negative if we see word 'worst'. But we have not analyzed the semantics and do not know if the word indeed is 'not worst'. To fix this issue, we can try to add bi-gram model feature, i.e., we not only analyze word itself, but also we should analyze two continous words in setences.

8 Features

For a problem like this, the features you use are far more important than the learning model you choose. Whenever you enter a new problem domain, one of your first orders of business is to beg, borrow, or steal the best features you can find. This means looking at any relevant published work and seeing what they've used. Maybe it means asking a colleague what features they use. But eventually you'll need to engineer new features that help in your particular situation. To get ideas for this dataset, you might check the discussion board on this Kaggle competition (https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews), which is using a very similar dataset. There are also a very large number of academic research papers on sentiment analysis that you can look at for ideas.