Machine Learning and Computational Statistics
Homework 3: SVM and Sentiment Analysis

Due: Monday, February 29, 2016, at 6pm (Submit via NYU Classes)

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g. LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

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 $g \in \mathbf{R}^d$ is a **subgradient** of $f : \mathbf{R}^d \to \mathbf{R}$ 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 **subdifferential** of f at a point x, denoted $\partial f(x)$, is the set of all subgradients of f at x.

Just as there is a calculus for gradients, there is a calculus for subgradients[1]. For our purposes, we can usually get by using the definition of subgradient directly. However, in the first problem below we derive a property that will make our life easier for finding a subgradient of the hinge loss and perceptron loss.

[1] A good reference for subgradients are the course notes on Subgradients by Boyd et al.

1. [Subgradients for pointwise maximum of functions] Suppose $f_1, \ldots, f_m : \mathbf{R}^d \to \mathbf{R}$ 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 $\mathbf{R}^d$ has a non-empty subdifferential at all points.] Show that $g \in \partial f(x)$.

Solution:

Fix any $z \in \mathbf{R}^d$. Then

$$\begin{aligned} f(z) &\geq f_k(z) \quad \text{by definition of } f(x) \\ &\geq f_k(x) + g^T(z-x) \quad \text{since } g \in \partial f_k(x) \\ &= f(x) + g^T(z-x) \quad \text{by choice of } k. \end{aligned}$$

Thus $f(z) \geq f(x) + g^T(z-x)$. Since this is true for all $z \in \mathbf{R}^d$, we have shown $g \in \partial f(x)$.

2. [Subgradient of hinge loss for linear prediction] Give a subgradient of

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

Solution:

By the previous problem, a subgradient is given by

$$g = \begin{cases} -yx & \text{for } yw^T x \leq 1 \\ 0 & \text{for } yw^T x > 1. \end{cases}$$

3 Perceptron

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

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

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

In this problem we will see why this loss function has this name.

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 $w^{(k)}$

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

   Solution:

   If $w^T x = 0$ is a separating hyperplane, then $y_i w^T x_i > 0$ for all i, and $\ell(w^T x_i, y_i) = \max\left\{0, -w^T x_i y_i\right\} = 0$ for all i. Thus the average over i is also 0.

2. Let $\mathcal{H}$ 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 loss is 0, and we make the right choice of subgradient, then we are exactly doing the Perceptron algorithm.
   Solution:

   The empirical risk is given by

   $$J(w) = \frac{1}{n} \sum_{i=1}^{n} \max\left\{0, -y_i w^T x_i\right\}.$$

   Let $f_i = \max\left\{0, -y_i w^T x_i\right\}$. A subgradient of f_i at w is

   $$g = \begin{cases} -y_i x_i & \text{for } y_i w^T x_i \leq 0 \\ 0 & \text{for } y_i w^T x_i > 0. \end{cases}$$

With a step size of 1, the step from $w^{(k)}$ to $w^{(k+1)}$ is given by

$$w^{(k+1)} = \begin{cases} w^{(k)} - (-y_i x_i) & \text{for } y_i w^T x_i \leq 0 \\ w^{(k)} & \text{for } y_i w^T x_i > 0, \end{cases}$$

which is exactly implemented by the perceptron algorithm.

3. 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 $\alpha_1, \ldots, \alpha_n \in \mathbf{R}$. 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.

   Solution:

   Note that initially $w = 0$, and we only ever add to it multiples of x_i for some i. Thus w is a linear combination of x_i's. If a point x_i was never misclassified when it was examined by the algorithm, then it is not a support vector. Otherwise, it is.

4 The Data

We will be using the Polarity Dataset v2.0, constructed by Pang and Lee. It has the full text from 2000 movies reivews: 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.

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: 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 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 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: https://docs.python.org/2/library/collections.html. Note that a Counter is also a dict.

   Solution:

```
class review_instance:
    """ Build the review class """
    def __init__(self):
        self.word_list = []
        self.label = 0
        self.word_dict = Counter()
        self.word_tfidf = Counter()

    def set_label(self, label_val):
        self.label = label_val

    def read_file(self, file_name):
        '''
        Read each file into a list of strings. And set the list to self.word_list
        '''
        f = open(file_name)
        lines = f.read().split(' ')
        symbols = '${}()[].,:;+-*/&|<>=~" '
       words = map(lambda Element: Element.translate(None, symbols).strip(), lines)
        words = filter(None, words)
        self.word_list = words
        self.construct_word_dict(stop_word=stopwords.words('english'))

    def construct_word_dict(self, stop_word=None, count_words=None):
        """
        count the words in word_list, transform to a dict of (word: word_count)
        :param stop_word: a list of stop words, The words you hope to filter, not
    included in dict, default set to None
        :param count_words: a list, the words you hope to keep in the count_dict, if
     set to None, will keep all the words
        :return:
        """
        c = Counter()
        c = Counter(self.word_list)
        if count_words:
            c = Counter({i: c[i] for i in count_words})

        if stop_word:
            for i in stop_word:
                del c[i]
        self.word_dict = c
```

```
    def transform_to_tfidf(self, idf_dict):
        """
        Take a idf dict as input, constrcut the {word: tfidf} vector tfidf = tf*(idf
    +1)
        """
        for word in self.word_dict:
            self.word_tfidf[word] = self.word_dict.get(word) * idf_dict.get(word, 1)
```

2. [Optional] Write a version of `generic_gradient_checker` from Homework 1 that works with sparse vectors represented as dict types. See Homework 1 solutions if you didn't do that part. Since we'll be using it for stochastic methods, it should take a single (x, y) pair, rather than the entire dataset. Be sure to use the dotProduct and increment primitives we provide, or make your own. **Note**: SVM loss is not differentiable everywhere. Yet our method for checking the gradient doesn't extent to subgradients. Thus in certain situations, the gradient checker may indicate that the gradient is not correct, when in fact you have provided a perfectly fine subgradient. This is an interesting opportunity to see how often we actually end up in those places.
   Solution:

```
def gradient_checker_for_pegasos(review_X, review_y, weight, reg_lambda,
    objective_func=pegasos_sgd_loss, gradient_func=pegasos_sgd_gradient, epsilon
    =0.01, tolerance=1e-4):
    true_gradient = gradient_func(review_X, review_y, weight, reg_lambda)
    weight_to_check = weight.copy()
    #To make weight to check have all the words appear in w and review stored in
    dict
    increment(weight_to_check, 0, review_X)
    approx_grad = weight_to_check.copy()

    for word in weight_to_check:
        weight_plus = weight_to_check.copy()
        weight_plus[word] += epsilon
        loss_plus = objective_func(review_X,review_y, weight_plus, reg_lambda)
        weight_minus = weight_to_check.copy()
        weight_minus[word] -= epsilon
        loss_minus = objective_func(review_X,review_y, weight_minus, reg_lambda)
        approx_grad[word] = (loss_plus - loss_minus)/(2*epsilon)
    distance = np.sum([(true_gradient[i] - approx_grad[i])**2 for i in approx_grad])
    return distance < tolerance
```

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 paper[2], we're considering the following formulation of the SVM objective function:

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

[2]Shalev-Shwartz et al.'s "Pegasos: Primal Estimated sub-GrAdient SOlver for SVM".

Note that, for simplicity, we are leaving off the unregularized bias term b, and the expression with "max" is just another way to write $\left(1 - y_i w^T x\right)_+$. 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] Compute a subgradient for the "stochastic" SVM objective, which assumes a single training point. Show that if your step size rule is $\eta_t = 1/(\lambda t)$, then the the corresponding SGD update is the same as given in the pseudocode. [You may use the following facts without proof: 1) If $f_1, \ldots, f_m : \mathbf{R}^d \to \mathbf{R}$ 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 $\alpha \geq 0$, $\partial(\alpha f)(x) = \alpha \partial f(x)$.]

   Solution:

   We need to find a subgradient of $\frac{\lambda}{2}\|w\|^2 + \max\left\{0, 1 - y_i w^T x_i\right\}$. Using the rules provided and the calculation in the first problem, a subgradient at w is given by

   $$g = \begin{cases} \lambda w - y_i x_i & \text{for } y_i w^T x_i < 1 \\ \lambda w & \text{for } y_i w^T x_i \geq 1. \end{cases}$$

   So a stochastic subgradient descent update would be

   $$w \leftarrow w - \eta g.$$

   If $y_I w^T x_i < 1$, the updated w value is

   $$w - \eta(\lambda w - y_i x_i) = (1 - \eta\lambda) w + \eta y_i x_i,$$

   and otherwise is simply

   $$w - \eta\lambda w = w(1 - \eta\lambda),$$

   which are the updates given in the algorithm.

2. 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.

Solution:

```
def pegasos_sgd_loss(review_X, review_y, w, reg_lambda):
    reg_term = np.sum([(w[i])**2 for i in w])*reg_lambda/2.0
    hinge_term = max(0, 1-review_y*dotProduct(w, review_X))
    return reg_term+hinge_term

def pegasos_sgd_gradient(review_X, review_y, w, reg_lambda):
    weight_grad = w.copy()
    for key in weight_grad:
        weight_grad[key] *= reg_lambda
    if review_y*dotProduct(w, review_X)<1:
        increment(weight_grad, -review_y, review_X)
    else:
        pass
    return weight_grad
```

3. 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 relatively slow. We can make things significantly faster by representing w as $w = sW$, where $s \in \mathbf{R}$ and $W \in \mathbf{R}^d$. You can start with $s = 1$ and W all zeros (i.e. an empty dictionary). Note that both updates 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**:

$$\begin{aligned} s_{t+1} &= (1-\eta_t\lambda)\, s_t \\ W_{t+1} &= W_t + \frac{1}{s_{t+1}}\eta_t y_j x_j. \end{aligned}$$

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 this change**. [See section 5.1 of Leon Bottou's Stochastic Gradient Tricks for a more generic version of this technique, and many other useful tricks.]

Solution:

```
def pegasos_fast(review_list, max_epoch, lam, watch_list=None, tfidf= False):
    """
    A faster version of pegasos tfidf: whether use tfidf features
    """
    if lam < 0:
```

```
        sys.exit("Lam must be greater than 0")
    print "lambda = %r, use tfidf = %r" %(lam, tfidf)

    #Initialization
    weight = Counter()
    epoch = 0
    t = 1.
    review_number = len(review_list)
    s = 1.
    while epoch < max_epoch:
        start_time = time.time()
        epoch += 1
        for j in xrange(review_number):
            t += 1
            eta = 1./(t*lam)
            s = (1 -eta*lam)*s
            review = review_list[j]
            label = review.label
            if tfidf:
                feature = review.word_tfidf
            else:
                feature = review.word_dict
            if label*dotProduct(feature, weight) < 1:
                temp = eta*label/s
                increment(weight, temp, feature)
                #print weight
        end_time = time.time()
        epoch_weight = Counter()
        increment(epoch_weight, s, weight)
        print "Epoch %r: in %.3f seconds. Training accuracy: %.3f, Test accuracy:
    %.3f" \ %(epoch, end_time-start_time, accuracy_percent(review_list, epoch_weight
    , tfidf=tfidf) accuracy_percent(watch_list, epoch_weight, tfidf=tfidf))

    return epoch_weight
```

4. Run both implementations of Pegasos to train an SVM on the training data (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.
Solution:

Both versions of the Pegasos algorithm were run for 10 iterations (with λ set to 0.1), and the final weight vectors w and runtimes were compared. First, the runtimes are given below:

Version of Pegasos	**Runtime**
First version (naive representation of w) including check for convergence	`1 loops, best of 3: 9min 4s per loop`
Second version (representing $w = sW$) not including check for convergence	`1 loops, best of 3: 4.12 s per loop`
Second version (representing $w = sW$) including check for convergence	`1 loops, best of 3: 7.18 s per loop`

Obviously the second version of the algorithm is a dramatic improvement over on the first. Next, to confirm the two results return essentially the same weight vector, the relative differences in weights are essentially zero.

5. 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 $x \mapsto w^T x$.

   Solution:

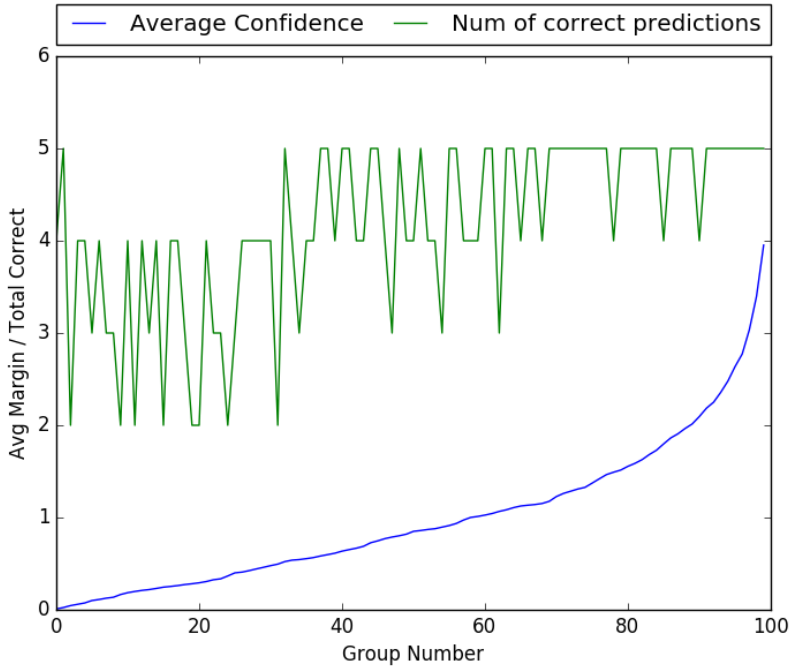
```
def svm_predict(review_X, weight):
    if dotProduct(weight, review_X)>0:
        return 1
    else:
        return -1

def accuracy_percent(review_list, weight, tfidf=False):
    label_list = [i.label for i in review_list]
    if tfidf:
        predict_list = [svm_predict(i.word_tfidf, weight) for i in review_list]
    else:
        predict_list = [svm_predict(i.word_dict, weight) for i in review_list]

    return sum([1 for i in range(len(label_list)) if label_list[i] == predict_list[i
    ]]) /float(len(label_list))
```

6. Using the bag-of-words feature representation described above, search for the regularization parameter that gives the minimal percent error on your test set. 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 new feature).

   Solution:

```
def stepsize_search(X, y, theta, loss_func, grad_func, epsilon=1e-6):
    alpha = 1.0
    gamma = 0.5
    loss = loss_func(X, y, theta)
    gradient = grad_func(X, y, theta)
    while True:
        theta_next = theta - alpha*grad_func(X,y, theta)
        loss_next = loss_func(X, y, theta_next)
        if loss_next > loss-epsilon:
            alpha = alpha*gamma
        else:
            return alpha
```

7. [Optional] Recall that the "score" is the value of the prediction $f(x) = w^T x$. We like to think that the magnitude of the score represents the confidence of the prediction. This is something we can directly verify or refute. Break the predictions into groups based on the score (you can play with the size of the groups to get a result you think is informative). For each group, examine the percentage error. You can make a table or graph. Summarize the results. Is there a correlation between higher magnitude scores and accuracy?
Solution:



The size of the test set is 500. This was sorted in increasing order by $\|w^T x\|$ which is the absolute score. This sorted set was divided into groups of 5 to give a total of 100 groups. For each group the number of correct predictions was measured and plotted as shown.
It is clear that for higher absolute scores the number of correct predictions is either 5 or 4. in very few cases it drops to 3. And for lower scores the number of correct predictions is either 4 or 3 and in few cases drops to 2.

8. [Optional] Our objective is not differentiable when $y_i w^T x_i = 1$. Investigate how often and when we have $y_i w^T x_i = 1$ (or perhaps within a small distance of 1 – this is for you to explore) . Describe your findings. If we didn't know about subgradients, one might suggest just skipping the update when $y w^T x_i = 1$. Does this seem reasonable?
Solution:

Our hinge loss function is not differentiable when $y_i w^T x_i = 1$. However, it is worth asking how often and when we hit this point. In our training data, we find 0 training set points with $y_i w^T x_i = 1$ This makes sense- w is a vector in R^{38793}, with floating point entries. Thus, the probability of constructing a text string such that $y_i w^T x_i = 1$ is extremely low.

Looking at the ten closest margins to one, that we encounter, look like this (Differences).

[1.4820084174083092e-05, 0.00012873878900576674, 0.0001810537327425754, 0.0002983012528537943, 0.000307147767330207, 0.00030998140113724926, 0.0003696983324755206, 0.0004078426633586929, 0.00042164032624636416, 0.0006180296092244131]. Of course we don't expect exact equality with computer arithmetic on doubles, but mathematically, we could definitely have some exactly equal to 1 at the solution.

Simply skipping the update if the margin is one does not seem to be ideal. When the margin is one the hinge loss is zero and so updating w just as we would if the hinge loss is something more than one, when the loss is zero seems to be the ideal thing to do. The difference between skipping and doing this update is that we are performing the update that's coming from the regularization (ie shrinking w towards 0) — the update from the loss part would still be 0.

7 Error Analysis

The natural language processing domain is particularly nice in that often 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 some examples that the model got wrong. List the features that contributed most heavily to the descision (e.g. rank them by $|w_i x_i|$), along with x_i, w_i, xw_i. Do you understand 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.
   Solution:

   Solution: An incorrect prediction on the most incorrect test sample as determined by the score was chosen and examined. the below tables lists the words that contributed most to the score sorted in descending order by their contribution to the score, i.e $w_i x_i$ for two samples from the test set for which the prediction was incorrect. It seems that words that do not have any real sentiment and that should be treated as neutral like "and" "i" and "the" are contributing towards this. Also this was a sample which was actually a negative sample but was predicted as positive. words like special and great are also contributing towards the postive prediction and this is because the context of the words and their usage is not being taken into account and simply the meaning of itself alone.

word	wx	w	No. Of Occurences in sample
and	0.904387941494	0.0215330462261	42
i	0.886254849935	0.0192664097812	46
the	0.613325155665	0.00666657777896	92
effects	0.565592458767	0.0471327048973	12
see	0.484793536086	0.080798922681	6
with	0.425594325409	0.0354661937841	12
is	0.289462807163	0.0111331848909	26
special	0.27959627205	0.03106625245	9
seen	0.277996293383	0.0926654311276	3
one	0.253396621378	0.0361995173398	7
as	0.241996773376	0.0241996773376	10
also	0.234996866708	0.0783322889028	3
in	0.234663537819	0.0106665244463	22
what	0.213863815149	0.0267329768936	8
well	0.212797162704	0.0709323875682	3
most	0.205663924481	0.0411327848962	5
first	0.204863935148	0.0292662764496	7
great	0.17946427381	0.0897321369048	2
i've	0.175464327142	0.0438660817856	4
lucas	0.173331022253	0.0173331022253	10

word	wx	w	No. Of Occurences in sample
and	1.01998640018	0.0509993200091	20
scream	0.989986800176	0.0549992666764	18
the	0.759656537913	0.0143331422248	53
as	0.356661911175	0.0356661911175	10
with	0.28532952894	0.0356661911175	8
your	0.212997160038	0.070999053346	3
will	0.188997480034	0.0629991600112	3
is	0.177664297809	0.0136664844469	13
from	0.173997680031	0.0289996133385	6
you	0.158664551139	0.0793322755697	2
american	0.14899801336	0.14899801336	1
back	0.144331408915	0.144331408915	1
most	0.13933147558	0.0696657377902	2
of	0.128331622245	0.00366661777843	35
he	0.1266649778	0.0633324889001	2
music	0.116665111132	0.0583325555659	2
one	0.10466527113	0.0523326355649	2
world	0.0983320222397	0.0983320222397	1
craven	0.0979986933508	0.0326662311169	3
see	0.0969987066839	0.0969987066839	1

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, 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.

1. Based on your error analysis, or on some idea you have, construct a new feature (or group of features) that you hope will improve your test performance. Describe the features and what kind of improvement they give. At this point, it's important to consider the standard errors $\sqrt{p(1-p)/n}$ (where p is the proportion of the test examples you got correct, and n is the size of the test set) on your performance estimates, to know whether the improvement is statistically significant.
   Solution:

   Ass described above, we will attempted to improve performance using the following feature:

   (a) 1-grams (previously conducted)

   (b) 1-grams and bigrams.

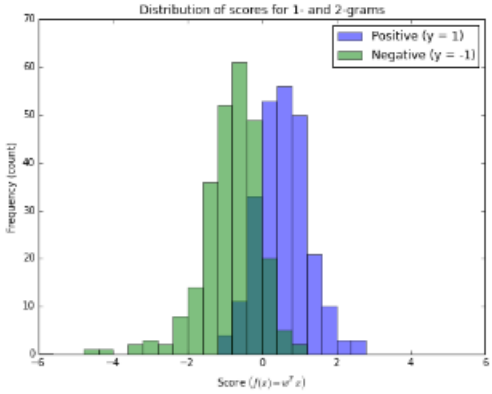
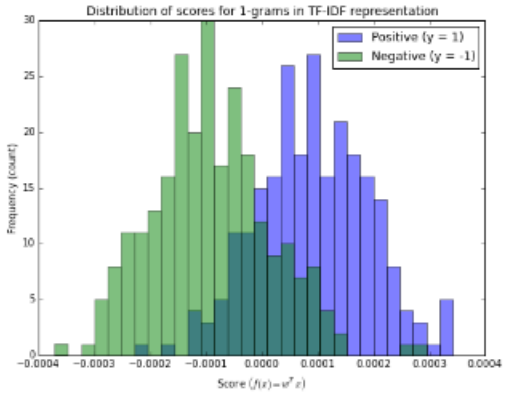
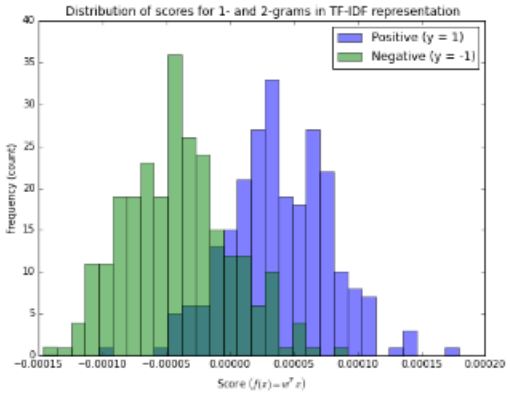
(c) TF-IDF for 1-grams

(d) TF-IDF for 1-grams and bigrams

We present histograms showing the distribution of scores for the true positive and true negative ratings, as well as the estimated probabilities of errors (including 95% confidence intervals on this probability).

Unfortunately, it does not seem that any of these alternate feature sets or representations produced an improved model (i.e. model with lower out-of-sample average 0-1 loss). Note, however, λ was not optimized for these alternate models. If additional time was available, I would have first searched for an optimal lambda value for each of these feature spaces, then determined the validation set 0-1 loss for this optimized model.

Still, given the results available, it appears the best model is the original 1-gram model (both in terms of runtime and average 0-1 loss).

Feature set	**Distribution**	**Average 0-1 loss** $(\hat{p}, \hat{p} \pm 1.96 \cdot SE)$
1- and 2-grams (counts)		0.15 (0.1187, 0.1813)
1-grams (TF-IDF)		0.188 (0.1538, 0.2222)
1- and 2-grams (TF-IDF)		0.166 (0.1334, 0.1986)

2. [Optional] Try to get the best performance possible by generating lots of new features, chang-

ing the pre-processing, or any other method you want, so long as you are using the same core SVM model. Describe what you tried, and how much improvement each thing brought to the model. To get you thinking on features, here are some basic ideas of varying quality: 1) how many words are in the review? 2) How many "negative" words are there? (You'd have to construct or find a list of negative words.) 3) Word n-gram features: Instead of single-word features, you can make every pair of consecutive words a feature. 4) Character n-gram features: Ignore word boundaries and make every sequence of n characters into a feature (this will be a lot). 5) Adding an extra feature whenever a word is preceded by "not". For example "not amazing" becomes its own feature. 6) Do we really need to eliminate those funny characters in the data loading phase? Might there be useful signal there? 7) Use tf-idf instead of raw word counts. The tf-idf is calculated as

$$\text{tfidf}(f_i) = \frac{FF_i}{\log(DF_i)} \tag{1}$$

where FF_i is the feature frequency of feature f_i and DF_i is the number of document containing f_i. In this way we increase the weight of rare words. Sometimes this scheme helps, sometimes it makes things worse. You could try using both! [Extra credit points will be awarded in proportion to how much improvement you achieve.]

Solution:

1. Length of the text
2. N-gram (N>1)
3. Word Vector: See Mikolov's word2vec(https://code.google.com/p/word2vec/) or Stanford's GloVe(http://nlp.stanford.edu/projects/glove/). For sentences/documents, see Quoc Le's paragaph vector(http://cs.stanford.edu/~quocle/paragraph_vector. pdf)
4. Topic-Model features: eg LDA
5. ...