Machine Learning	**Assignment 4**
Comp540 Spring 2017	**due 6 March 2017 at 8 pm**

This homework is due March 6 at 8 pm on Canvas. The code base `hw4.zip` *for the assignment is an attachment to* Assignment 4 *on Canvas. You will add your code at the indicated spots in the files there. Place your answers to Problems 1, 2 and 3 (typeset) in a file called* `writeup.pdf` *and add it to the zip archive. Upload the code archive back to Canvas before the due date and time. DO NOT upload the image data sets onto Canvas – remove them before you zip up your archive for upload.*

1 Kernelizing k-nearest neighbors (5 points)

The nearest neighbor classifier assigns a new vector $x \in \Re^d$ to the same class as that of the nearest neighbor $x^{(i)}$ in the training set $\mathcal{D} = \{(x^{(i)}, y^{(i)}) \mid x^{(i)} \in Re^d, y^{(i)} \in \{-1, 1\}\}$ where the distance between two points in $\Re^d$ is the Euclidean distance. Re-express this classification rule in terms of dot products and then make use of the kernel trick to formulate the k-nearest neighbor algorithm for a general non-linear kernel function K satisfying the Mercer condition.

2 Constructing kernels (10 points)

Show that given valid kernels $k_1(x, x^{'})$ and $k_2(x, x^{'})$ defined over $x, x^{'} \in \Re^d$, the following new kernels will also be valid.

- $k(x, x^{'}) = ck_1(x, x^{'})$
- $k(x, x^{'}) = f(x)k_1(x, x^{'})f(x^{'})$ where $f(x)$ is any function on $x \in \Re^d$.
- $k(x, x^{'}) = k_1(x, x^{'}) + k_2(x, x^{'})$

3 Fitting an SVM classifier by hand (15 points)

Consider a dataset with two points $\mathcal{D} = \{(0, -1), (\sqrt{2}, +1)\}$. We will map each of these points to three dimensions using the feature vector $\phi(x) = (1, \sqrt{2}x, x^2)$. Recall that the maximum margin classifier has the form:

$$min \ \frac{1}{2}||\theta||^2 \text{ such that}$$

$$\begin{aligned} y^{(1)}(\theta^T \phi(x^{(1)}) + \theta_0) &\geq 1 \\ y^{(2)}(\theta^T \phi(x^{(2)}) + \theta_0) &\geq 1 \end{aligned}$$

for the points $(x^{(1)}, y^{(1)})$ and $(x^{(2)}, y^{(2)})$ in $\mathcal{D}$.

- (5 points) Write down a vector that is parallel to the optimal vector θ. Hint: θ is perpendicular to the decision boundary between the two points in the three-dimensional space.
- (3 points) What is the value of the margin that is achieved by this θ? Hint: the margin is the distance from each support vector to the decision boundary.
- (3 points) Solve for the θ given that the margin is equal to $\frac{1}{||\theta||}$.
- (3 points) Solve for the intercept θ_0 using your value for θ and the inequalities above. Since both points are support vectors, the inequalities will be tight.
- (3 points) Write down the equation for the decision boundary in terms of θ, θ_0 and x.

4 Support vector machines for binary classification (25 points)

In this exercise, you will be building support vector machines for binary classification problems. To get started, please download the code base `hw4.zip` from Canvas. The files relevant to this problem are shown below.

Name	Edit?	Read?	Description
binary_svm.ipynb	Yes	Yes	Python notebook that will run your functions for the two class SVM
svm_spam.ipynb	Yes	Yes	Python notebook that will run your functions for spam classification using the two class SVM
linear_svm.py	Yes	Yes	loss functions for the SVM
linear_classifier.py	No	Yes	SVM training and prediction functions
utils.py	Yes	Yes	plot utility functions, kernels
data/ex4data1.mat	No	No	Example dataset 1
data/ex4data2.mat	No	No	Example dataset 2
data/ex4data3.mat	No	No	Example dataset 3
data/spamTrain.mat	No	No	Spam training set
data/spamTest.mat	No	No	Spam test set
data/vocab.txt	No	Yes	vocabulary list
hw4.pdf	No	Yes	this document

4.1: Support vector machines

In this exercise, you will build support vector machines (SVMs) for solving binary classification problems. You will experiment with your classifier on three example 2D datasets. Experimenting with these datasets will help you gain intuition into how SVMs work and how to use a Gaussian kernel with SVMs. The provided notebook `binary_svm.ipynb`, will help

you step through these parts of the exercise. In the final part of the exercise, you will be using your SVM implementation to build a spam classifier. You will complete the notebook `svm_spam.ipynb` and use your SVM classifier to classify a given spam data set.

4.1A: The hinge loss function and gradient (5 points)

Now you will implement the hinge loss cost function and its gradient for support vector machines. Complete the `binary_svm_loss` function in `linear_svm.py` to return the cost and gradient for the hinge loss function. Recall that the hinge loss function is

$$J(\theta) = \frac{1}{2m}\sum_{j=0}^{d}{\theta_j}^2 + \frac{C}{m}\sum_{i=1}^{m} max(0, 1 - y^{(i)}h_\theta(x^{(i)}))$$

where $h_\theta(x) = \theta^T x$ with $x_0 = 1$. C is the penalty factor which measures how much misclassifications are penalized. If $y^{(i)}h_\theta(x^{(i)})) \geq 1$, then $x^{(i)}$ is correctly classified and the loss associated with that example is zero. If $y^{(i)}h_\theta(x^{(i)})) < 1$, then $x^{(i)}$ is not within the appropriate margin (positive or negative) and the loss associated with that example is greater than zero. The gradient of the hinge loss function is a vector of the same length as θ where the j^{th} element, $j = 0, 1, \ldots, d$ is defined as follows:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \begin{cases} \frac{1}{m}\theta_j + \frac{C}{m}\sum_{i=1}^{m} -y^{(i)}x_j^{(i)} & \text{if } y^{(i)}h_\theta(x^{(i)})) < 1 \\ \frac{1}{m}\theta_j & \text{if } y^{(i)}h_\theta(x^{(i)})) \geq 1 \end{cases}$$

Once you are done, a cell in `binary_svm.ipynb` will call your `binary_svm_loss` function with a zero vector θ. You should see that the cost J is 1.0. The gradient of the loss function with respect to an all-zeros θ vector is also computed and should be $[-0.12956186 - 0.00167647]^T$.

4.1B Example dataset 1: impact of varying C (2 points)

We will begin with a 2D example dataset which can be separated by a linear boundary (shown in Figure 1). In this dataset, the positions of the positive examples (green circles) and the negative examples (indicated with red circles) suggest a natural separation indicated by the gap. However, notice that there is an outlier positive example on the far left at about (0.1, 4.1). As part of this exercise, you will also see how this outlier affects the SVM decision boundary.

In this part of the exercise, you will try using different values of the C parameter with SVMs. Informally, the C parameter is a positive value that controls the penalty for misclassified training examples. A large C parameter tells the SVM to try to classify all the examples correctly. C plays a role similar to $\frac{1}{\lambda}$, where λ is the regularization parameter that we were using previously for logistic regression.

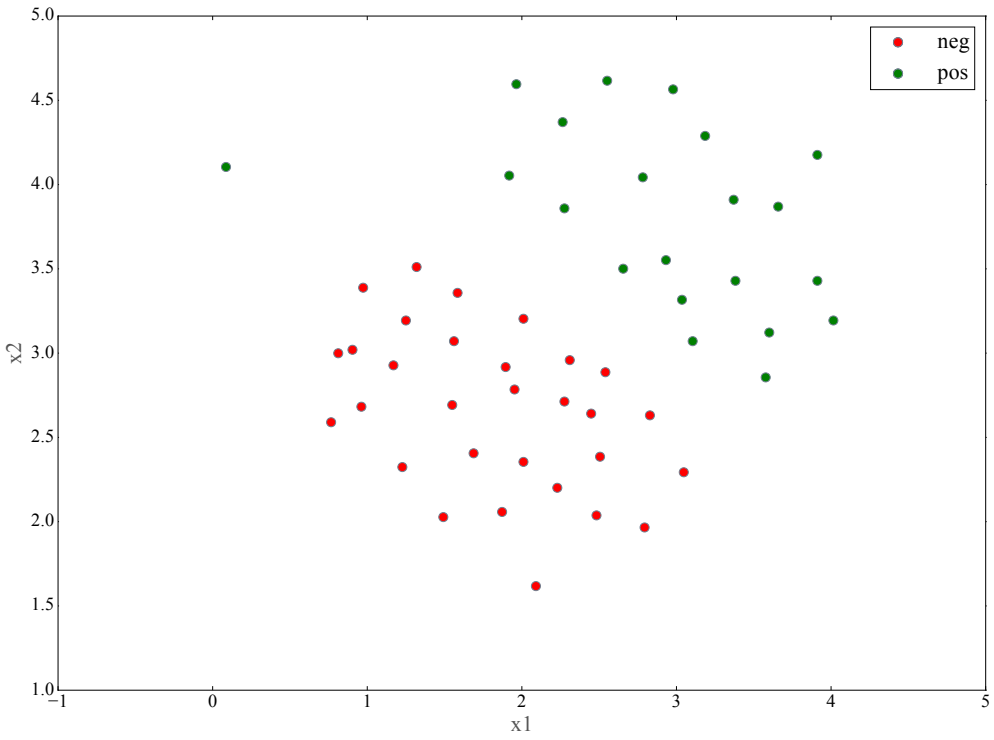

Figure 1: Example dataset 1

The SVM training function is in `linear_classifier.py` – this is a gradient descent algorithm that uses your loss and gradient functions. The next cell in `ex4.ipynb` will train an SVM on the example data set 1 with C = 1. It first scales the data to have zero mean and unit variance, and adds the intercept term to the data matrix. When C = 1, you should find that the SVM puts the decision boundary in the gap between the two datasets and misclassifies the data point on the far left (Figure 2).

Your task is to try different values of C on this dataset. Specifically, you should change the value of C in the script to C = 100 and run the SVM training again. When C = 100, you should find that the SVM now classifies every single example correctly, but has a decision boundary that does not appear to be a natural fit for the data (Figure 3).

SVM with Gaussian kernels

In this part of the exercise, you will be using SVMs to do non-linear classification. In particular, you will be using SVMs with Gaussian kernels on datasets that are not linearly separable.

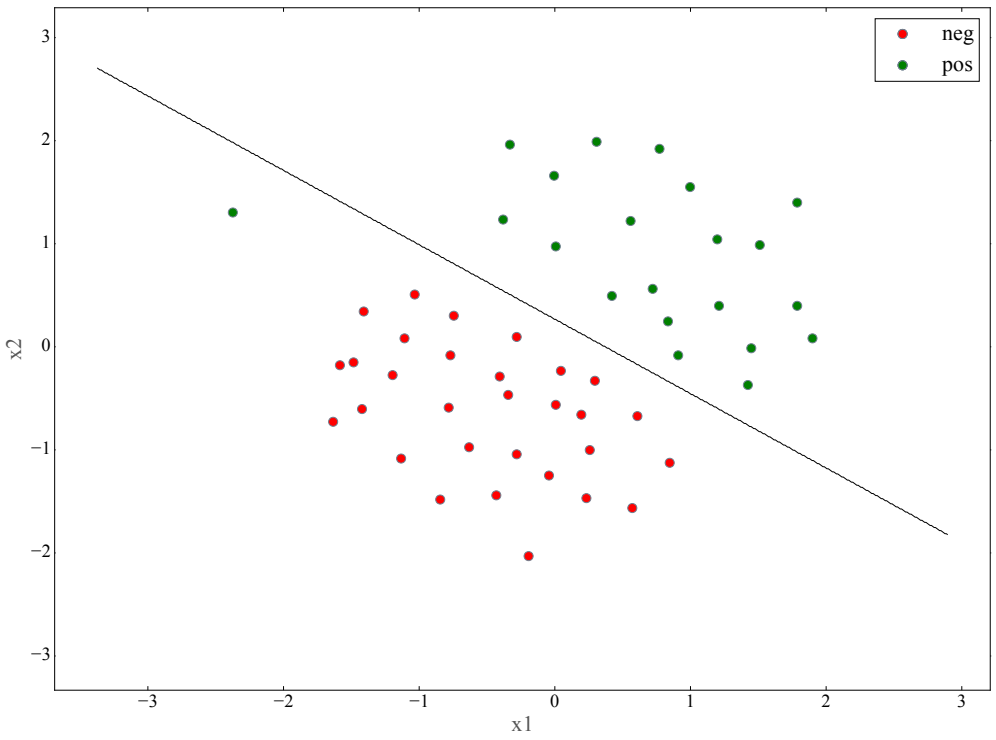

Figure 2: SVM decision boundary with C = 1 for Example dataset 1

4.1C Gaussian kernel (3 points)

To find non-linear decision boundaries with the SVM, we need to first implement a Gaussian kernel. You can think of the Gaussian kernel as a similarity function that measures the distance between a pair of examples, $(x^{(i)}, x^{(j)})$. The Gaussian kernel is also parameterized by a bandwidth parameter, σ, which determines how fast the similarity metric decreases (to 0) as the examples are further apart. You should now complete the function `gaussian_kernel` in `utils.py` to compute the Gaussian kernel between two examples. The Gaussian kernel function is defined as:

$$k(x^{(i)}, x^{(j)}) = exp\left(-\frac{||x^{(i)} - x^{(j)}||^2}{2\sigma^2}\right)$$

When you have completed the function, a cell in the notebook `binary_svm.ipynb` will test your kernel function on two provided examples and you should expect to see a value of 0.324652.

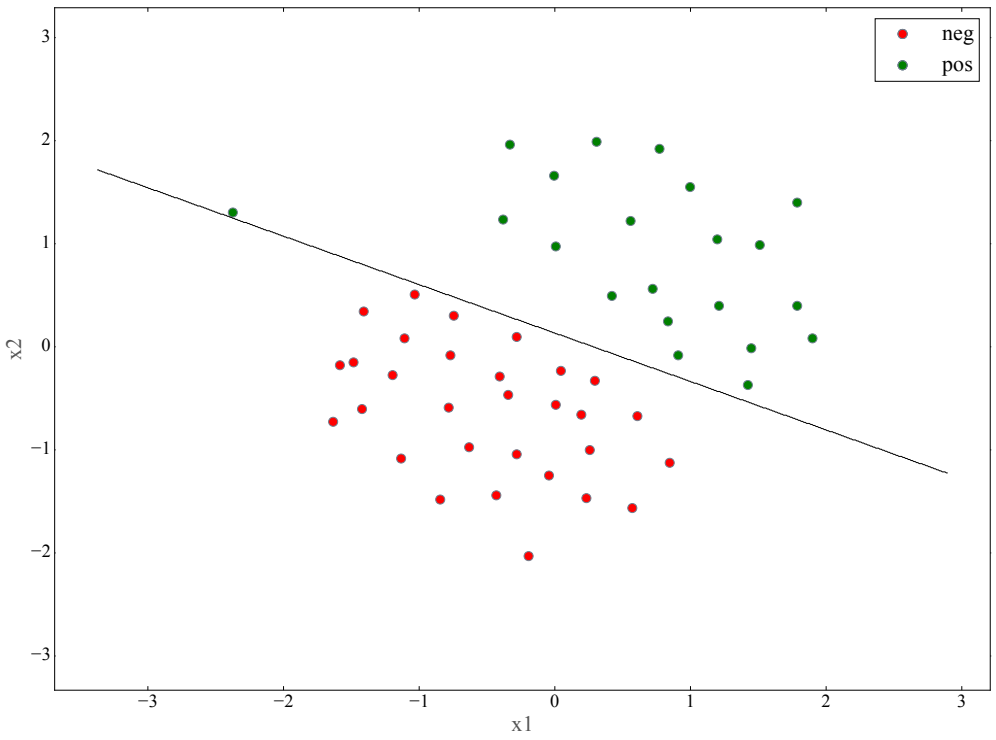

Figure 3: SVM decision boundary with C = 100 for Example dataset 1

Example dataset 2: learning non-linear boundaries

The next cell in `binary_svm.ipynb` will load and plot dataset 2 (Figure 4). From the figure, you can observe that there is no linear decision boundary that separates the positive and negative examples for this dataset. However, by using the Gaussian kernel with the SVM, you will be able to learn a non-linear decision boundary that can perform reasonably well for the dataset. If you have correctly implemented the Gaussian kernel function, a cell in the `binary_svm.ipynb` notebook will proceed to train the SVM with the Gaussian kernel on this dataset.

Figure 5 shows the decision boundary found by the SVM with C = 1 and a Gaussian kernel with $\sigma = 0.01$. The decision boundary is able to separate most of the positive and negative examples correctly and follows the contours of the dataset well.

4.2 Example dataset 3: selecting hyper parameters for SVMs (5 points)

In this part of the exercise, you will gain more practical skills on how to use a SVM with a Gaussian kernel. The next part of `binary_svm.ipynb` will load and display a third dataset (Figure 6). In the provided dataset, `ex4data3.mat`, you are given the variables `X`, `y`, `Xval`, `yval`. You will be using the SVM with the Gaussian kernel with this dataset. Your task is

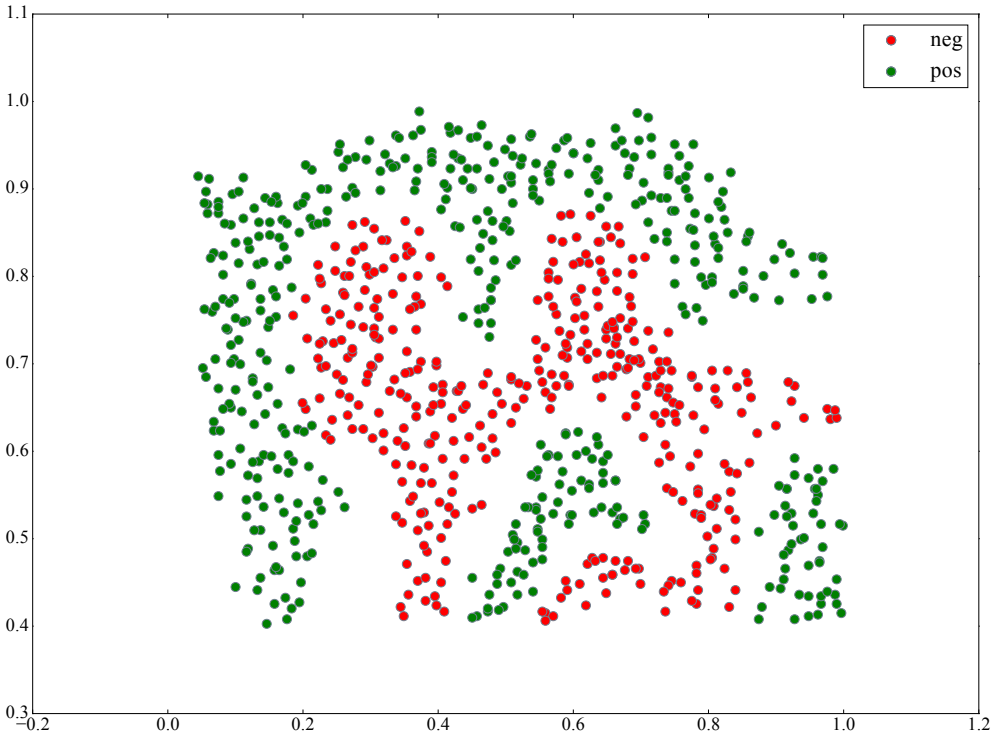

Figure 4: Example dataset 2

to use the validation set `Xval, yval` to determine the best C and σ parameter to use. You should write any additional code necessary to help you search over the parameters C and σ. For both C and σ, we suggest trying values in multiplicative steps (e.g., 0.01, 0.03, 0.1, 0.3, 1, 3, 10, 30). Note that you should try all possible pairs of values for C and σ (e.g., C = 0.3 and $\sigma = 0.1$). For example, if you try each of the 8 values listed above for C and for σ, you would end up training and evaluating (on the validation set) a total of $8^2 = 64$ different models.

When selecting the best C and σ parameter to use, you train on `X,y` with a given C and σ, and then evaluate the error of the model on the validation set. Recall that for classification, the error is defined as the fraction of the validation examples that were classified incorrectly. You can use the `predict` method of the SVM classifier to generate the predictions for the validation set.

After you have determined the best C and σ parameters to use, you should replace the assignments to `best_C` and `best_sigma` in `binary_svm.ipynb` with the best values you find. For our best parameters, the SVM returned a decision boundary shown in Figure 7.

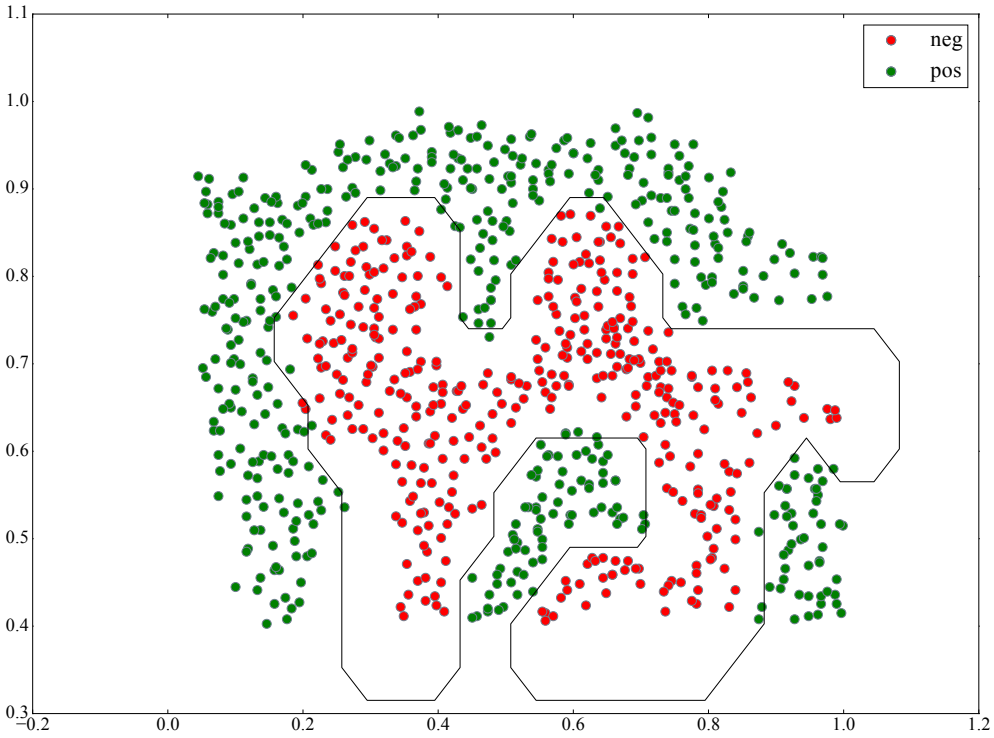

Figure 5: SVM gaussian kernel decision boundary for Example dataset 2, C = 1 and $\sigma = 0.02$

4.3 Spam Classification with SVMs (10 points)

Many email services today provide spam filters that are able to classify emails into spam and non-spam email with high accuracy. In this part of the exercise, you will use SVMs to build your own spam filter. You will be training a classifier to classify whether a given email, x, is spam or non-spam. Throughout the rest of this exercise, you will be using the notebook `svm_spam.ipynb`. The dataset included for this exercise is based on a subset of the SpamAssassin Public Corpus. For the purpose of this exercise, you will only be using the body of the email (excluding the email headers).

Preprocessing email

Before starting on a machine learning task, it is usually insightful to take a look at examples from the dataset. Figure 8 shows a sample email that contains a URL, an email address (at the end), numbers, and dollar amounts. While many emails would contain similar types of entities (e.g., numbers, other URLs, or other email addresses), the specific entities (e.g., the specific URL or specific dollar amount) will be different in almost every email. Therefore, one method often employed in processing emails is to normalize these values, so that all URLs are treated the same, all numbers are treated the same, etc. For example, we could

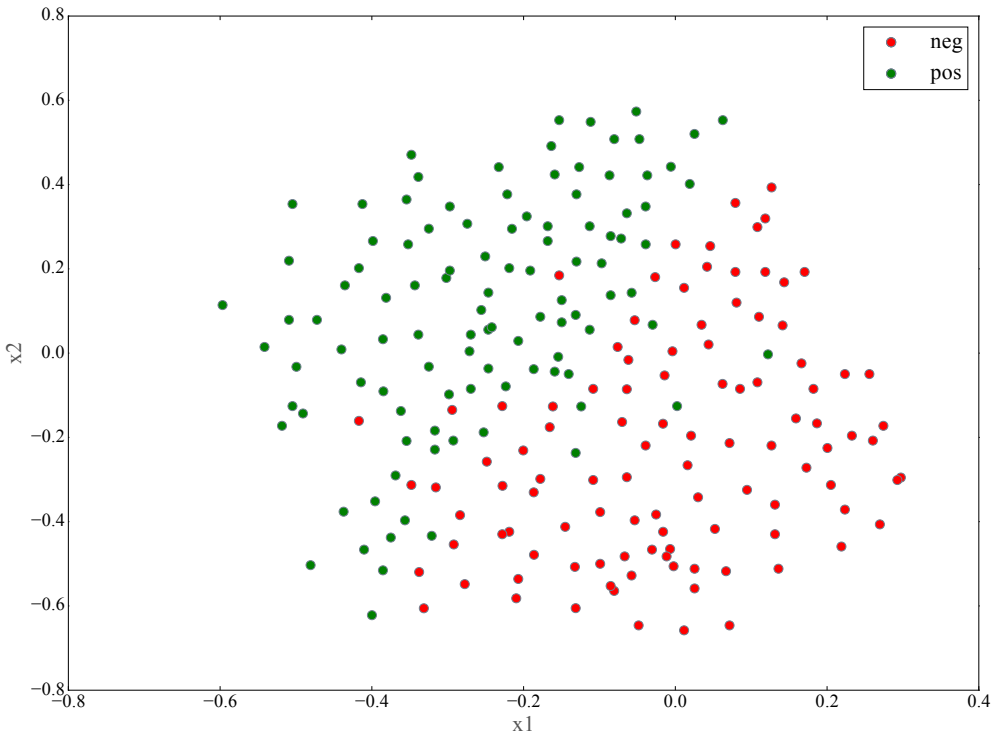

Figure 6: Example dataset 3

replace each URL in the email with the unique string "httpaddr" to indicate that a URL was present.

This has the effect of letting the spam classifier make a classification decision based on whether any URL was present, rather than whether a specific URL was present. This typically improves the performance of a spam classifier, since spammers often randomize the URLs, and thus the odds of seeing any particular URL again in a new piece of spam is very small.

Here are typical email preprocessing and normalization steps:

- **Lower-casing:** The entire email is converted into lower case, so that capitalization is ignored (e.g., IndIcaTE is treated the same as Indicate).
- **Stripping HTML:** All HTML tags are removed from the emails. Many emails often come with HTML formatting; we remove all the HTML tags, so that only the content remains.
- **Normalizing URLs:** All URLs are replaced with the text `httpaddr`.
- **Normalizing Email addresses:** All email addresses are replaced with the text `emailaddr`.

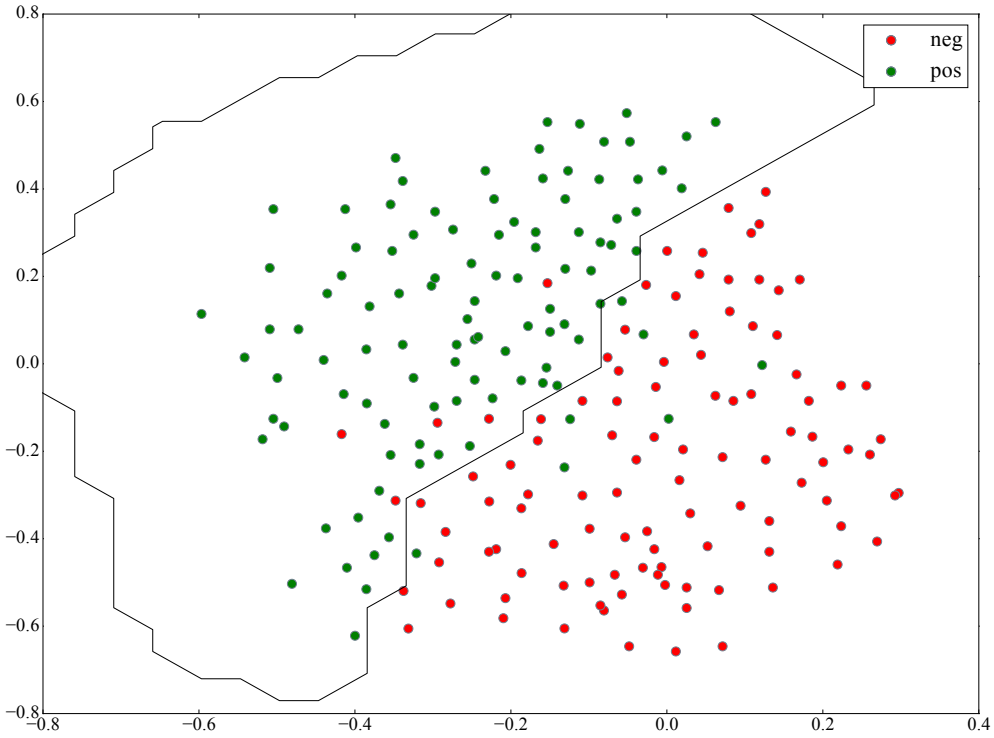

Figure 7: SVM gaussian kernel decision boundary for Example dataset 3 with the best hyper parameters.

- **Normalizing numbers:** All numbers are replaced with the text `number`.
- **Normalizing dollars:** All dollar signs ($) are replaced with the text `dollar`.
- **Word stemming:** Words are reduced to their stemmed form. For example, discount, discounts, discounted and discounting are all replaced with discount. Sometimes, the stemmer actually strips off additional characters from the end, so include, includes, included, and including are all replaced with includ.
- **Removal of non-words:** Non-words and punctuation have been removed. All white spaces (tabs, newlines, spaces) have all been trimmed to a single space character.

The result of these preprocessing steps is shown in Figure 9. While preprocessing has left word fragments and non-words, this form turns out to be much easier to work with for performing feature extraction.

Feature representation

After preprocessing the emails, we have a list of words (e.g., Figure 9) for each email. The next step is to choose which words we would like to use in our classifier and which we would

```
> Anyone knows how much it costs to host a web portal ?
> Well, it depends on how many visitors youre expecting. This can be
anywhere from less than 10 bucks a month to a couple of $100. You
should checkout http://www.rackspace.com/ or perhaps Amazon EC2 if
youre running something big..
To unsubscribe yourself from this mailing list, send an email to:
groupname-unsubscribe@egroups.com
```

Figure 8: Sample email

```
anyon know how much it cost to host a web portal well it depend on how
mani visitor your expect thi can be anywher from less than number buck
a month to a coupl of dollarnumb you should checkout httpaddr or perhap
amazon ecnumb if your run someth big to unsubscrib yourself from thi
mail list send an email to emailaddr
```

Figure 9: Sample email - preprocessed

want to leave out.

For this exercise, we have chosen only the most frequently occurring words as our set of words considered (the vocabulary list). Since words that occur rarely in the training set are only in a few emails, they might cause the model to overfit our training set. The complete vocabulary list is in the file `vocab.txt`. Our vocabulary list was selected by choosing all words which occur at least a 100 times in the spam corpus, resulting in a list of 1899 words. In practice, a vocabulary list with about 10,000 to 50,000 words is often used.

Given the vocabulary list, we then map each word in the preprocessed emails (e.g., Figure 9) into a list of word indices that contains the index of the word in the vocabulary list. Figure 10 shows the mapping for the sample email. Specifically, in the sample email, the word anyone was first normalized to anyon and then mapped onto the index 86 in the vocabulary list.

Training SVMs for spam classification (10 points)

`svm_spam.ipynb` will load a training dataset `spamTrain.mat` which has been pre-processed according to the scheme shown above. `spamTrain.mat` contains 4000 training examples of spam and non-spam email, while `spamTest.mat` contains 1000 test examples. That is, each original email was processed and converted into a vector $x \in \Re^{1899}$. Your task is to design a protocol for training an SVM classifier for this data set. Issues you need to consider are: should you scale the data matrix and if so how, how do you set the learning rate, the number of iterations and the penalty parameter C? What kind of kernel is best for this problem, and how do you select the corresponding kernel hyper parameters? Use the SVM with the

```
86 916 794 1077 883 370 1699 790 1822
1831 883 431 1171 794 1002 1893 1364
592 1676 238 162 89 688 945 1663 1120
1062 1699 375 1162 479 1893 1510 799
1182 1237 810 1895 1440 1547 181 1699
1758 1896 688 1676 992 961 1477 71 530
1699 531
```

Figure 10: Word indices for Sample email. This email will be represented by a feature vector of length 1899 where the values at indices in this list are set to 1 and the rest are 0.

loss function `binary_svm_loss` and the training algorithm in `linear_classifier.py`. In `writeup.pdf` explain how you chose the parameters for training the SVM, providing graphs and tables to support your choices. Give us your best values for all of the chosen parameters and hyper parameters. Make sure that you do not use the test set to choose these parameters. Finally, evaluate the accuracy of your model on the test set and report it. You can use the provided vocabulary list (and the function `get_vocab_dict` in `utils.py`) to interpret the θ associated with the model – find the words most indicative of spam or ham using the parameters of your best model.

Our best classifier gets a training accuracy of about 99.8% and a test accuracy of about 98.5%.

5 Support vector machines for multi-class classification (35 points)

In this exercise, you will be building support vector machines for multi-class classification problems. To get started, please download the code base `hw4.zip` from Canvas. The files relevant to this problem are shown below.

Name	**Edit?**	**Read?**	**Description**
multiclass_svm.ipynb	Yes	Yes	Python notebook to run your functions for learning an SVM classifier for CIFAR-10 data
linear_svm.py	Yes	Yes	loss functions for the SVM
linear_classifier.py	Yes	Yes	SVM training and prediction functions
data_utils.py	No	Yes	Functions for reading CIFAR-10 data
utils.py	No	Yes	plot utility functions, kernels
hw4.pdf	No	Yes	this document

In this exercise you will: implement

- a fully-vectorized loss function for the multi-class SVM classifier,
- a fully-vectorized expression for its analytic gradient,

- check your implementation using numerical gradient,
- use a validation set to tune the learning rate and regularization strength,
- optimize the loss function with stochastic gradient descent and
- visualize the final learned weights on the CIFAR-10 dataset.

Download the data

Open up a terminal window and navigate to the `datasets` directory of `hw4`. Run the `get_datasets.sh` script. On my Mac, I just type in `./get_datasets.sh` at the shell prompt. A new folder called `cifar_10_batches_py` will be created and it will contain 60,000 labeled images for training and 10,000 labeled images for testing. If you have already downloaded this data set before, then go into `data_utils.py` and change the path to the `datasets` folder there.

We have provided a function to read this data in `data_utils.py` – this function also partitions the training data into a training set of 49,000 images and a validation set of 1,000 images used for selecting hyperparameters for softmax regression. Each image is a 32×32 array of RGB triples. It is preprocessed by subtracting the mean image from all images, and flatted into a 1-dimensional array of size 3072. Then a 1 is appended to the front of that vector to handle the intercept term. So the training set is a `numpy` matrix of size 49000×3073, the validation set is a matrix of size 1000×3073 and the set-aside test set is of size 10000×3073. We also have a random sample of 500 images from the training data to serve as a development set or dev set to test our gradient and loss function implementations. Make sure you use the version of `data_utils.py` included with this assignment, and not a previous one.

5A: Loss and gradient function for multi-class SVM – naive version (5 points)

Your code for this section will all be written inside `linear_svm.py`. As you can see, we have written the function `svm_loss_naive` which uses `for` loops to evaluate the multiclass SVM loss function. The multi-class SVM loss is set up so that the score associated with the correct class for each example is higher than the score for the other (incorrect classes) by some fixed margin Δ. The Multiclass SVM loss for the i^{th} example is then formalized as follows:

$$L^{(i)}((x^{(i)}, y^{(i)}), \Delta) = \sum_{j \neq y^{(i)}} max(0, {\theta^{(j)}}^T x^{(i)} - {\theta^{y^{(i)}}}^T x^{(i)} + \Delta)$$

where $\theta^{(j)}$ is the j^{th} column of the θ matrix, corresponding to the j^{th} class.

Suppose that we have three classes and let $\theta * x^{(i)}$ be the vector $[13, -7, 11]$. Let $\Delta = 10$ and the true class $y^{(i)} = 0$. The expression above sums over all incorrect classes ($j \neq y^{(i)}$), so we get two terms in our loss expression for this example $x^{(i)}$.

$$L^{(i)} = max(0, -7 - 13 + 10) + max(0, 11 - 13 + 10) = 0 + 8 = 8$$

The first term evaluates to zero since $-7 - 13 + 10$ is a negative number, which is then thresholded to zero with the $max(0, -)$ function. Intuitively, we get zero for the first term because the score associated with the correct class (13) is greater than the score for class 1 (-7) by at least the margin 10. In fact, the difference is 20, which is much greater than 10, but the SVM only cares that the difference is at least 10; any additional difference above the margin is clamped at zero with the max operation. The second term evaluates to $11 - 13 + 10 = 8$. That is, even though the correct class had a higher score than the incorrect class ($13 > 11$), it was not greater by the desired margin of 10. The difference was only 2, which is why the loss comes out to 8 (i.e. how much higher the difference would have to be to meet the margin). In summary, the SVM loss function wants the score of the correct class $y^{(i)}$ to be larger than the incorrect class scores by at least by Δ. If this is not the case, it accumulates loss. `svm_naive_loss` sums up loss over all the examples $i = 1, \ldots, m$.

The gradient returned from the function is right now all zero. Derive and implement the gradient for the multi-class SVM cost function and implement it in the function `svm_loss_naive` in `linear_svm.py`. To check that you have correctly implemented the gradient, we will numerically estimate the gradient of the loss function and compare the numeric estimate to the gradient that you computed. We have provided code that does this for you in `multiclass_svm.ipynb`. It is possible that once in a while a dimension in the gradient check will not match exactly. What could such a discrepancy be caused by? Is it a reason for concern? Hint: the SVM loss function is not, strictly speaking, differentiable.

5B: Loss and gradient function for multi-class SVM – vectorized version (10 points)

Now complete the function `svm_loss_vectorized` in `linear_svm.py` to implement the loss function $J(\theta)$ and the gradient of the loss function without using any `for` loops. Re-express the computations in terms of matrix operations on X, y and θ.

Implementing mini-batch gradient descent

In large-scale applications, the training data can have millions of examples. Hence, it seems wasteful to compute the loss function over the entire training set in order to perform only a single parameter update. A very common approach to addressing this challenge is to compute the gradient over batches of the training data. For example, a typical batch contains 256 examples from a training set of over 1.2 million. This batch is then used to perform a parameter update:

$$\theta^{(k)} \rightarrow \theta^{(k)} - \alpha \nabla_{\theta^{(k)}} J(\theta)$$

where α is the step size or learning rate for gradient descent.

We have implemented mini-batch gradient descent in the method `train` in `linear_classifier.py`. You can set the `verbose` argument of `train` to be `True` and observe how the loss function varies with iteration number. A cell in the `multiclass_svm.ipynb` notebook sets up a multiclass_svm instance and trains it with a specifid learning rate and regularization strength for 1500 iterations. This will give you an idea of how long it takes to train an SVM on this data set.

5C: Prediction function for multi-class SVM (5 points)

Write the predict function in `linear_classifier.py` for the multiclass SVM. Then, a cell in the `multiclass_svm.ipynb` notebook will evaluate the performance of your SVM learned in the previous step on both the training and validation set. You should expect to see about 37-39% accuracy here.

5D: Tuning hyper parameters for training a multi-class SVM (10 points)

Use the validation set to tune hyperparameters (regularization strength and learning rate). You should experiment with 4 learning rates and 4 regularization strengths; if you are careful you should be able to get a classification accuracy of about 0.4 on the validation set. Suggested parameter values are provided in the Python notebook – feel free to experiment outside that range.

Write code that chooses the best hyperparameters by tuning on the validation set. For each combination of hyperparameters, train a linear SVM on the training set, compute its accuracy on the training and validation sets, and store these numbers in the `results` dictionary. In addition, store the best validation accuracy in best_val and the LinearSVM object that achieves this accuracy in best_svm. Hint: You should use a small value for num_iters as you develop your validation code so that the SVMs don't take much time to train; once you are confident that your validation code works, you should rerun the validation code with a larger value for num_iters.

We have provided visualization code in the succeeding cells for you to see the variation in training and validation set accuracy with changes in regularization strength and learning rate. Then, evaluate the test set accuracy on the best_svm learned. The final cell visualizes the θ associated with each of the ten class. You can save the figure in a PDF file and attach it to your `writeup.pdf`, or leave the best result in your notebook when you upload it on Canvas.

5E: Comparing the performance of multi-class SVM and softmax regression (5 points)

Compare the performance of the multi-class SVM you have built in this assignment with the softmax regression that you built for the previous one. Which approach takes longer

to train, which approach achieves higher performance? Compare the visualizations of the θ parameters learned by both methods – do you see any differences? Comment on hyper parameter selection for both methods. Place your discussion with supporting graphs and plots in `writeup.pdf`.