Homework 4

Support Vector Machines

Issue Date: April 18, 2019
Due Date: May 4, 2019

Introduction

In this exercise, you will be using support vector machines (SVMs) to build a spam classifier. Before starting on the programming exercise, we strongly recommend revising the lectures on SVMs and coming for office hours if needed.

The exercise is based on Octave/MATLAB. If you do not have any of these sofware installed, refer to Homework2 for instructions on installing them.

Files included in this exercise

ex6.m - Octave/MATLAB script for the first half of the exercise

ex6data1.mat - Example Dataset 1

ex6data2.mat - Example Dataset 2

svmTrain.m - SVM training function

svmPredict.m - SVM prediction function

plotData.m - Plot 2D data

visualizeBoundaryLinear.m - Plot linear boundary

visualizeBoundary.m - Plot non-linear boundary

linearKernel.m - Linear kernel for SVM

[⋆] gaussianKernel.m - Gaussian kernel for SVM

ex6_spam.m - Octave/MATLAB script for the second half of the exercise

spamTrain.mat - Spam training set

spamTest.mat - Spam test set

emailSample1.txt - Sample email 1

emailSample2.txt - Sample email 2

spamSample1.txt - Sample spam 1

spamSample2.txt - Sample spam 2

vocab.txt - Vocabulary list

getVocabList.m - Load vocabulary list

porterStemmer.m - Stemming function

readFile.m - Reads a file into a character string

[⋆] processEmail.m - Email preprocessing

[⋆] emailFeatures.m - Feature extraction from emails
⋆ indicates files you will need to complete

Throughout the exercise, you will be using the script ex6.m. These scripts set up the dataset for the problems and make calls to functions that you will write. You are only required to modify functions in other files, by following the instructions in this assignment.

Where to get help

The exercise uses Octave1 or MATLAB, a high-level programming language well-suited for numerical computations. At the Octave/MATLAB command line, typing **help** followed by a function name displays documentation for a built-in function. For example, *help plot* will bring up help information for plotting. Further documentation for Octave functions can be found at the Octave documentation pages. MATLAB documentation can be found at the MATLAB documentation pages.

1 Support Vector Machines

In the first half of this exercise, you will be using support vector machines (SVMs) with various example 2D datasets. Experimenting with these datasets will help you gain an intuition of how SVMs work and how to use a Gaussian kernel with SVMs. In the next half of the exercise, you will be using support vector machines to build a spam classifier. The provided script, ex6.m, will help you step through the first half of the exercise.

1.1 Example Dataset 1

We will begin by with a 2D example dataset which can be separated by a linear boundary. The script ex6.m will plot the training data (Figure 1). In this dataset, the positions of the positive examples (indicated with +) and the negative examples (indicated with o) 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.

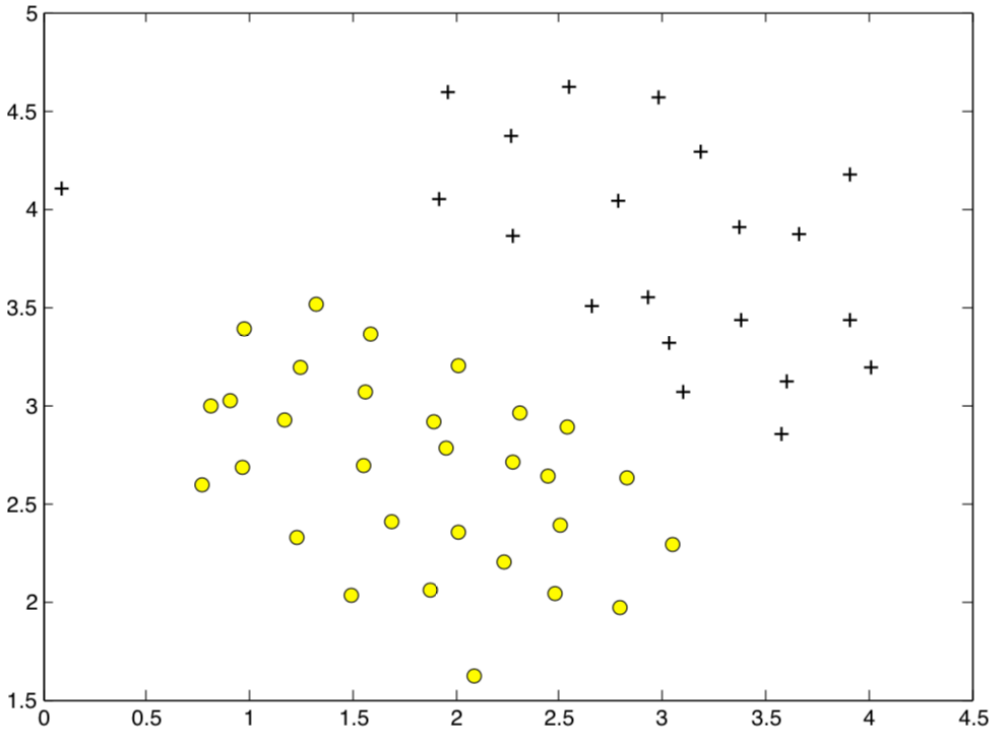

Figure 1: Example Dataset 1

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

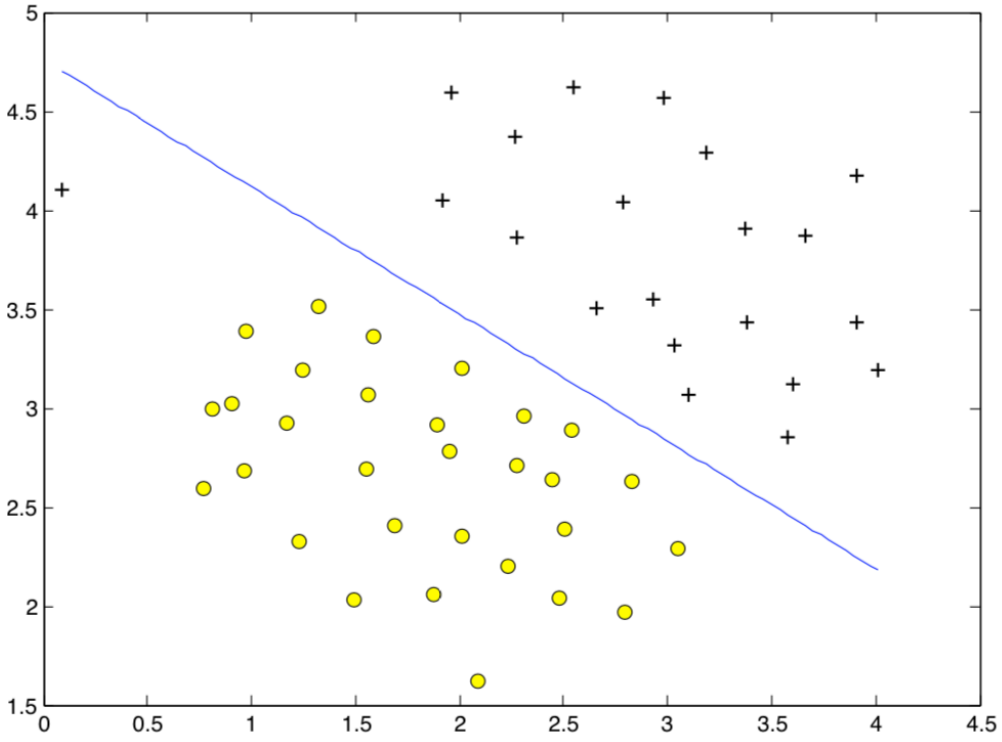

Figure 2: SVM Decision Boundary with $\mathcal{C} = 1$ (Example Dataset 1)

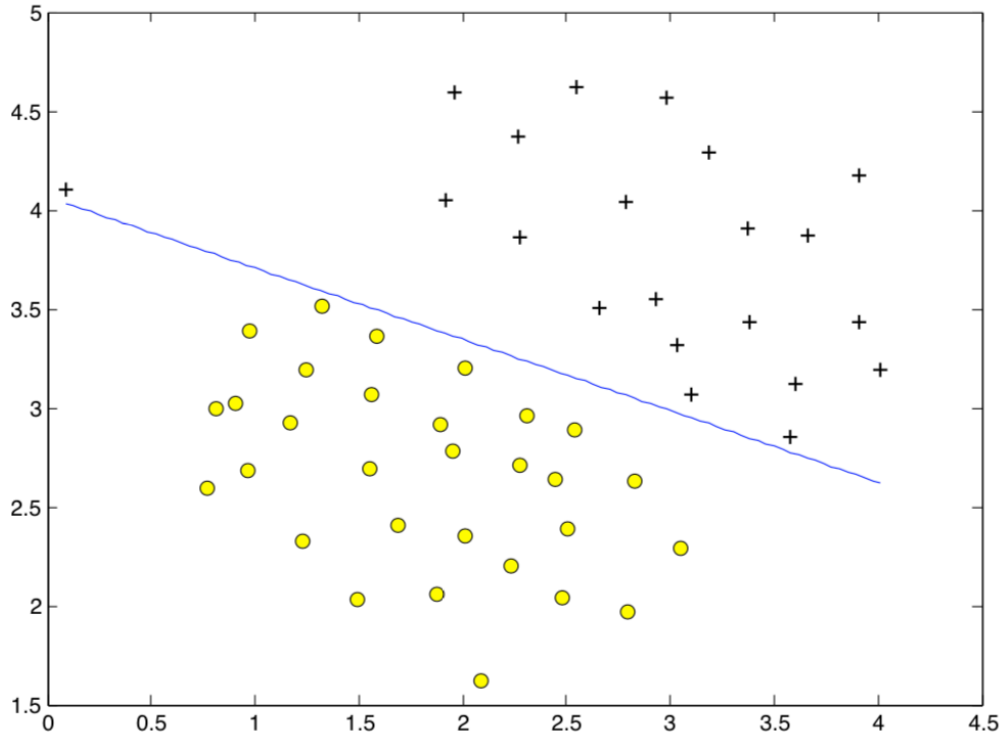

Figure 3: SVM Decision Boundary with $\mathcal{C} = \infty$// (Example Dataset 1)

The next part in ex6.m will run the SVM training (with $\mathcal{C} = 1$) using SVM software that we have included with the starter code, svmTrain.m[1]. When $\mathcal{C} = 1$, you should find that the SVM puts the decision boundary

[1]In order to ensure compatibility with Octave/MATLAB, we have included this imple- mentation of an SVM learning

in the gap between the two datasets and *misclassifies* the data point on the far left (Figure 2)

> **Implementation Note:** Most SVM software packages (including svmTrain.m) automatically add the extra feature $x_0 = 1$ for you and automatically take care of learning the intercept term θ_0. So when passing your training data to the SVM software, there is no need to add this extra feature $x_0 = 1$ yourself. In particular, in Octave/MATLAB your code should be working with training examples $x \in R^n$ (rather than $x \in R^{n+1}$); for example, in the first example dataset $x \in R^2$

Your task is to try different values of $\mathcal{C}$ on this dataset. Specifically, you should change the value of $\mathcal{C}$ in the script to $\mathcal{C} = 100$ and run the SVM training again. When $\mathcal{C} = 100$, 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).

1.2 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.

1.2.1 Gaussian Kernel

To find non-linear decision boundaries with the SVM, we need to first im- plement a Gaussian kernel. You can think of the Gaussian kernel as a sim- ilarity function that measures the "distance" between a pair of examples, $\left(x^{(i)}, x^{(j)}\right)$. 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 code in gaussianKernel.m to compute the Gaussian kernel between two examples, $\left(x^{(i)}, x^{(j)}\right)$. The Gaussian kernel function is defined as:

$$K_{gaussian}\left(x^{(i)}, x^{(j)}\right) = \exp\left(-\frac{||x^{(i)}, x^{(j)}||^2}{2\sigma^2}\right) = \exp\left(-\frac{\sum_{k=1}^{n}(x_k^{(i)} - x_k^{(j)})^2}{2\sigma^2}\right) \quad (1)$$

Once you've completed the function gaussianKernel.m, the script ex6.m will test your kernel function on two provided examples and you should expect to see a value of 0.324652.

algorithm. However, this particular implementation was chosen to maximize compatibility, and is not very efficient. If you are training an SVM on a real problem, especially if you need to scale to a larger dataset, we strongly recommend instead using a highly optimized SVM toolbox such as LIBSVM.

1.2.2 Example Dataset 2

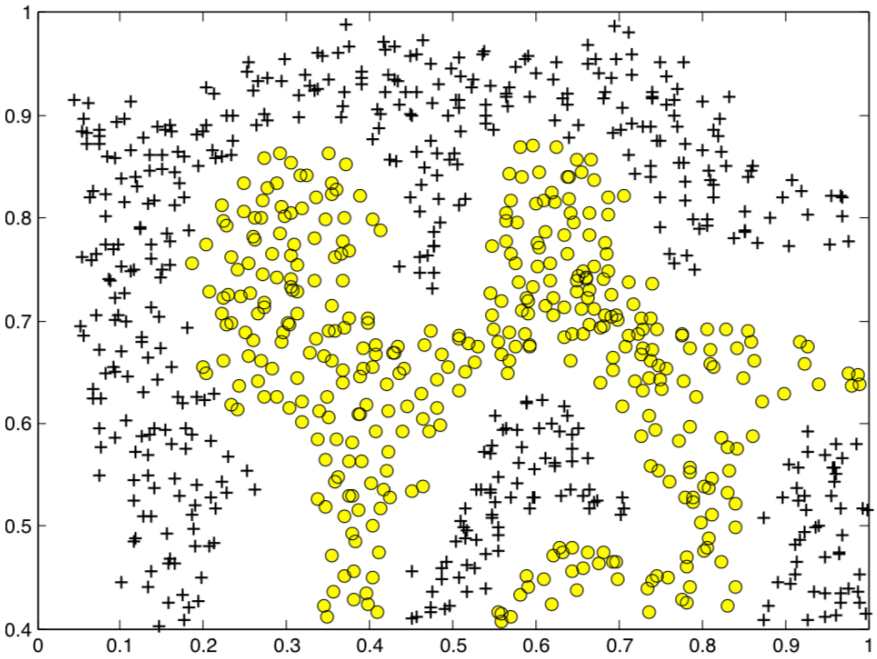

Figure 4: Example Dataset 2

The next part in ex6.m will load and plot dataset 2 (Figure 3). From the figure, you can obserse 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, ex6.m will proceed to train the SVM with the Gaussian kernel on this dataset.

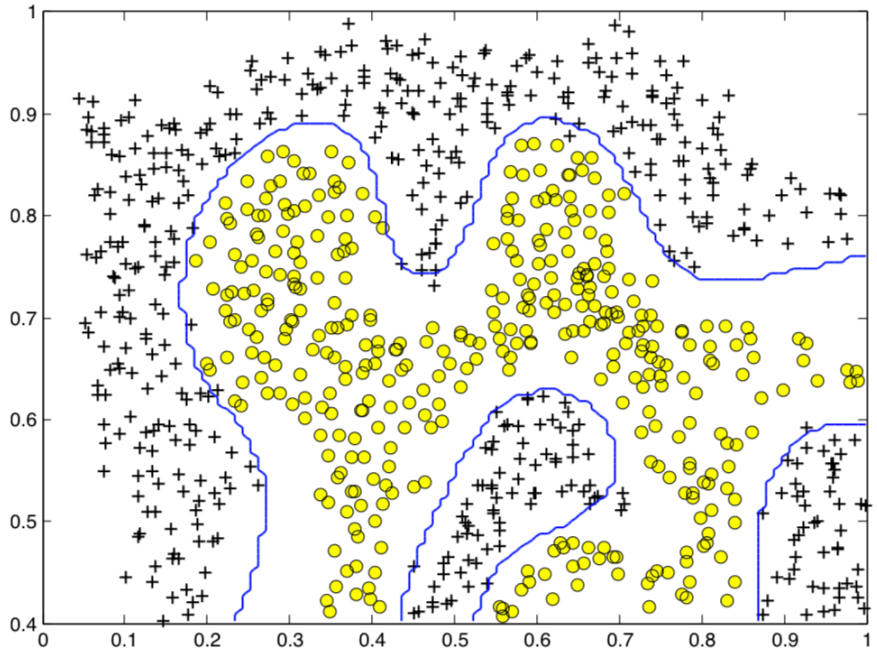

Figure 5: SVM (Gaussian Kernel) Decision Boundary (Example Dataset 2)

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

2 Spam Classification

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 ($y = 1$) or non-spam ($y = 0$). In particular, you need to convert each email into a feature vector $x \in R^n$. The following parts of the exercise will walk you through how such a feature vector can be constructed from an email. Throughout the rest of this exercise, you will be using the the script ex6spam.m The dataset included for this exercise is based on a a subset of the SpamAssassin Public Corpus[2]. For the purpose of this exercise, you will only be using the body of the email (excluding the email headers).

2.1 Preprocessing Emails

```
> 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 6: Sample Email

Before starting on a machine learning task, it is usually insightful to take a look at examples from the dataset. Figure 6 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 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.

In processEmail.m, we have implemented the following email prepro- cessing and normalization steps:

- **Lower-casing:** The entire email is converted into lower case, so that captialization is ignored (e.g., IndIcaTE is treated the same as Indicate).
- **Stripping HTML:** 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".
- **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".

[2] http://spamassassin.apache.org/old/publiccorpus/

- **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 7 While preprocessing has left word fragments and non-words, this form turns out to be much easier to work with for performing feature extraction.

```
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 7: Preprocessed Sample Email

```
1 aa
2 ab
3 abil
...
86 anyon
...
916 know
...
1898 zero
1899 zip
```

Figure 8: Vocabulary List

```
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 9: Word Indices for Sample Email

2.1.1 Vocabulary List

After preprocessing the emails, we have a list of words (e.g., Figure 7) for each email. The next step is to choose which words we would like to use in our classifier and which we would 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 and also shown in Figure 8. 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 can now map each word in the preprocessed emails (e.g., Figure 7) into a list of word indices that contains the index of the word in the vocabulary list. Figure 9 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.

Your task now is to complete the code in processEmail.m to perform this mapping. In the code, you are given a string **str** which is a single word from the processed email. You should look up the word in the vocabulary list vocabList and find if the word exists in the vocabulary list. If the word exists, you should add the index of the word into the word indices variable. If the word does not exist, and is therefore not in the vocabulary, you can skip the word.

once you have implemented processEmail.m, the script ex6 spam.m will run your code on the email sample and you should see an output similar to Figures 7 & 9.

Octave/MATLAB Tip: In Octave/MATLAB, you can compare two strings with the strcmp function. For example, strcmp(str1, str2) will return 1 only when both strings are equal. In the provided starter code, vocabList is a "cell-array" containing the words in the vocabulary. In Octave/MATLAB, a cell-array is just like a normal array (i.e., a vector), except that its elements can also be strings (which they can't in a normal Octave/MATLAB matrix/vector), and you index into them using curly braces instead of square brackets. Specifically, to get the word at index i, you can use vocabList{i}. You can also use length(vocabList) to get the number of words in the vocabulary.

2.2 Extracting Features from Emails

You will now implement the feature extraction that converts each email into a vector in R^n. For this exercise, you will be using $n = \#$ words in vocabulary list. Specifically, the feature $x_i \in \{0, 1\}$ for an email corresponds to whether the i-th word in the dictionary occurs in the email. That is, $x_i = 1$ if the i-th word is in the email and $x_i = 0$ if the i-th word is not present in the email.

Thus, for a typical email, this feature would look like:

$$x = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 1 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \in R^n$$

You should now complete the code in emailFeatures.m to generate a feature vector for an email, given the word indices.

Once you have implemented emailFeatures.m, the next part of ex6 spam.m will run your code on the email sample. You should see that the feature vector had length 1899 and 45 non-zero entries.

2.3 Training SVM for Spam Classification

After you have completed the feature extraction functions, the next step of ex6_spam.m will load a preprocessed training dataset that will be used to train a SVM classifier. spamTrain.mat contains 4000 training examples of spam and non-spam email, while spamTest.mat contains 1000 test examples. Each original email was processed using the processEmail and emailFeatures functions and converted into a vector $x^{(i)} \in R^{1899}$.

After loading the dataset, ex6_spam.m will proceed to train a SVM to classify between spam ($y = 1$) and non-spam ($y = 0$) emails. Once the training completes, you should see that the classifier gets a training accuracy of about 99.8% and a test accuracy of about 98.5%.

2.4 Top Predictors for Spam

```
our click remov guarante visit basenumb dollar will price pleas nbsp
most lo ga dollarnumb
```

Figure 10: Top predictors for spam email

To better understand how the spam classifier works, we can inspect the parameters to see which words the classifier thinks are the most predictive of spam. The next step of ex6_spam.m finds the parameters with the largest positive values in the classifier and displays the corresponding words (Figure 10). Thus, if an email contains words such as "guarantee", "remove", "dollar", and "price" (the top predictors shown in Figure 10), it is likely to be classified as spam.