

Machine Learning

(Programming Exercise)

Andrew Ng

Stanford University

Contents

Programming Exercise 1: Linear Regression	1
Programming Exercise 2: Logistic Regression	16
Programming Exercise 3: Multi-class Classification and Neural Networks	29
Programming Exercise 4: Neural Networks Learning	41
Programming Exercise 5: Regularized Linear Regression and Bias v.s. Variance	56
Programming Exercise 6: Support Vector Machines	70
Programming Exercise 7: K-means Clustering and Principal Component Analysis	86
Programming Exercise 8: Anomaly Detection and Recommender Systems	102

Programming Exercise 1: Linear Regression

Machine Learning

Introduction

In this exercise, you will implement linear regression and get to see it work on data. Before starting on this programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics.

To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise. If needed, use the `cd` command in Octave to change to this directory before starting this exercise.

You can also find instructions for installing Octave on the “Octave Installation” page on the course website.

Files included in this exercise

- `ex1.m` - Octave script that will help step you through the exercise
- `ex1_multi.m` - Octave script for the later parts of the exercise
- `ex1data1.txt` - Dataset for linear regression with one variable
- `ex1data2.txt` - Dataset for linear regression with multiple variables
- `submit.m` - Submission script that sends your solutions to our servers
- [*] `warmUpExercise.m` - Simple example function in Octave
- [*] `plotData.m` - Function to display the dataset
- [*] `computeCost.m` - Function to compute the cost of linear regression
- [*] `gradientDescent.m` - Function to run gradient descent
- [†] `computeCostMulti.m` - Cost function for multiple variables
- [†] `gradientDescentMulti.m` - Gradient descent for multiple variables
- [†] `featureNormalize.m` - Function to normalize features
- [†] `normalEqn.m` - Function to compute the normal equations

* indicates files you will need to complete

† indicates extra credit exercises

Throughout the exercise, you will be using the scripts `ex1.m` and `ex1_multi.m`. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify either of them. You are only required to modify functions in other files, by following the instructions in this assignment.

For this programming exercise, you are only required to complete the first part of the exercise to implement linear regression with one variable. The second part of the exercise, which you may complete for extra credit, covers linear regression with multiple variables.

Where to get help

The exercises in this course use Octave,¹ a high-level programming language well-suited for numerical computations. If you do not have Octave installed, please refer to the installation instructions at the “Octave Installation” page on the course website.

At the Octave 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](#).

We also strongly encourage using the online **Q&A Forum** to discuss exercises with other students. However, do not look at any source code written by others or share your source code with others.

1 Simple octave function

The first part of `ex1.m` gives you practice with Octave syntax and the homework submission process. In the file `warmUpExercise.m`, you will find the outline of an Octave function. Modify it to return a 5 x 5 identity matrix by filling in the following code:

```
A = eye(5);
```

When you are finished, run `ex1.m` (assuming you are in the correct directory, type “`ex1`” at the Octave prompt) and you should see output similar to the following:

¹Octave is a free alternative to MATLAB. For the programming exercises, you are free to use either Octave or MATLAB.

```
ans =

Diagonal Matrix

   1   0   0   0   0
   0   1   0   0   0
   0   0   1   0   0
   0   0   0   1   0
   0   0   0   0   1
```

Now `ex1.m` will pause until you press any key, and then will run the code for the next part of the assignment. If you wish to quit, typing `ctrl-c` will stop the program in the middle of its run.

1.1 Submitting Solutions

After completing a part of the exercise, you can submit your solutions for grading by typing `submit` at the Octave command line. The submission script will prompt you for your username and password and ask you which files you want to submit. You can obtain a submission password from the website’s “Programming Exercises” page.

You should now submit the warm up exercise.

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration. To prevent rapid-fire guessing, the system enforces a minimum of 5 minutes between submissions.

2 Linear regression with one variable

In this part of this exercise, you will implement linear regression with one variable to predict profits for a food truck. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities and you have data for profits and populations from the cities.

You would like to use this data to help you select which city to expand to next.

The file `ex1data1.txt` contains the dataset for our linear regression problem. The first column is the population of a city and the second column is the profit of a food truck in that city. A negative value for profit indicates a loss.

The `ex1.m` script has already been set up to load this data for you.

2.1 Plotting the Data

Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population). (Many other problems that you will encounter in real life are multi-dimensional and can't be plotted on a 2-d plot.)

In `ex1.m`, the dataset is loaded from the data file into the variables X and y :

```
data = load('ex1data1.txt');           % read comma separated data
X = data(:, 1); y = data(:, 2);
m = length(y);                         % number of training examples
```

Next, the script calls the `plotData` function to create a scatter plot of the data. Your job is to complete `plotData.m` to draw the plot; modify the file and fill in the following code:

```
plot(x, y, 'rx', 'MarkerSize', 10);    % Plot the data
ylabel('Profit in $10,000s');           % Set the y-axis label
xlabel('Population of City in 10,000s'); % Set the x-axis label
```

Now, when you continue to run `ex1.m`, our end result should look like Figure 1, with the same red “x” markers and axis labels.

To learn more about the plot command, you can type `help plot` at the Octave command prompt or to search online for plotting documentation. (To change the markers to red “x”, we used the option ‘rx’ together with the plot command, i.e., `plot(...,[your options here],..., 'rx');`)

2.2 Gradient Descent

In this part, you will fit the linear regression parameters θ to our dataset using gradient descent.

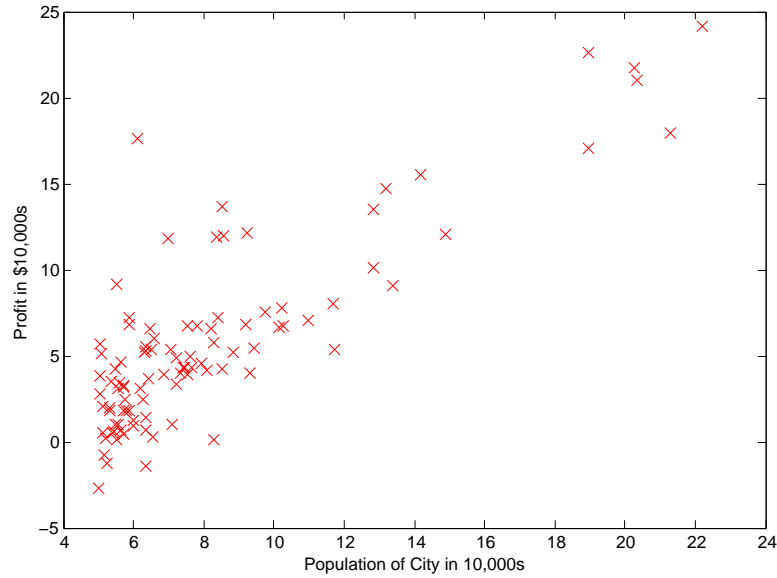


Figure 1: Scatter plot of training data

2.2.1 Update Equations

The objective of linear regression is to minimize the cost function

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

where the hypothesis $h_{\theta}(x)$ is given by the linear model

$$h_{\theta}(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

Recall that the parameters of your model are the θ_j values. These are the values you will adjust to minimize cost $J(\theta)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad (\text{simultaneously update } \theta_j \text{ for all } j).$$

With each step of gradient descent, your parameters θ_j come closer to the

optimal values that will achieve the lowest cost $J(\theta)$.

Implementation Note: We store each example as a row in the the X matrix in Octave. To take into account the intercept term (θ_0), we add an additional first column to X and set it to all ones. This allows us to treat θ_0 as simply another ‘feature’.

2.2.2 Implementation

In `ex1.m`, we have already set up the data for linear regression. In the following lines, we add another dimension to our data to accommodate the θ_0 intercept term. We also initialize the initial parameters to 0 and the learning rate `alpha` to 0.01.

```
X = [ones(m, 1), data(:,1)]; % Add a column of ones to x
theta = zeros(2, 1); % initialize fitting parameters

iterations = 1500;
alpha = 0.01;
```

2.2.3 Computing the cost $J(\theta)$

As you perform gradient descent to learn minimize the cost function $J(\theta)$, it is helpful to monitor the convergence by computing the cost. In this section, you will implement a function to calculate $J(\theta)$ so you can check the convergence of your gradient descent implementation.

Your next task is to complete the code in the file `computeCost.m`, which is a function that computes $J(\theta)$. As you are doing this, remember that the variables X and y are not scalar values, but matrices whose rows represent the examples from the training set.

Once you have completed the function, the next step in `ex1.m` will run `computeCost` once using θ initialized to zeros, and you will see the cost printed to the screen.

You should expect to see a cost of 32.07.

You should now submit “compute cost” for linear regression with one variable.

2.2.4 Gradient descent

Next, you will implement gradient descent in the file `gradientDescent.m`. The loop structure has been written for you, and you only need to supply the updates to θ within each iteration.

As you program, make sure you understand what you are trying to optimize and what is being updated. Keep in mind that the cost $J(\theta)$ is parameterized by the vector θ , not X and y . That is, we minimize the value of $J(\theta)$ by changing the values of the vector θ , not by changing X or y . Refer to the equations in this handout and to the video lectures if you are uncertain.

A good way to verify that gradient descent is working correctly is to look at the value of $J(\theta)$ and check that it is decreasing with each step. The starter code for `gradientDescent.m` calls `computeCost` on every iteration and prints the cost. Assuming you have implemented gradient descent and `computeCost` correctly, your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm.

After you are finished, `ex1.m` will use your final parameters to plot the linear fit. The result should look something like Figure 2:

Your final values for θ will also be used to make predictions on profits in areas of 35,000 and 70,000 people. Note the way that the following lines in `ex1.m` uses matrix multiplication, rather than explicit summation or looping, to calculate the predictions. This is an example of code vectorization in Octave.

You should now submit gradient descent for linear regression with one variable.

```
predict1 = [1, 3.5] * theta;  
predict2 = [1, 7] * theta;
```

2.3 Debugging

Here are some things to keep in mind as you implement gradient descent:

- Octave array indices start from one, not zero. If you're storing θ_0 and θ_1 in a vector called `theta`, the values will be `theta(1)` and `theta(2)`.
- If you are seeing many errors at runtime, inspect your matrix operations to make sure that you're adding and multiplying matrices of compatible dimensions. Printing the dimensions of variables with the `size` command will help you debug.

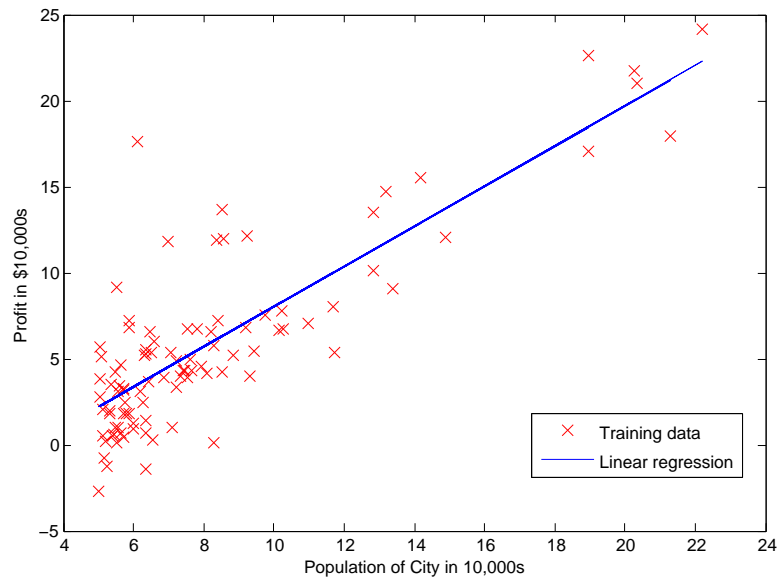


Figure 2: Training data with linear regression fit

- By default, Octave interprets math operators to be matrix operators. This is a common source of size incompatibility errors. If you don't want matrix multiplication, you need to add the “dot” notation to specify this to Octave. For example, `A*B` does a matrix multiply, while `A.*B` does an element-wise multiplication.

2.4 Visualizing $J(\theta)$

To understand the cost function $J(\theta)$ better, you will now plot the cost over a 2-dimensional grid of θ_0 and θ_1 values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images.

In the next step of `ex1.m`, there is code set up to calculate $J(\theta)$ over a grid of values using the `computeCost` function that you wrote.

```

% initialize J_vals to a matrix of 0's
J_vals = zeros(length(theta0_vals), length(theta1_vals));

% Fill out J_vals
for i = 1:length(theta0_vals)
    for j = 1:length(theta1_vals)
        t = [theta0_vals(i); theta1_vals(j)];
        J_vals(i,j) = computeCost(x, y, t);
    end
end
end

```

After these lines are executed, you will have a 2-D array of $J(\theta)$ values. The script `ex1.m` will then use these values to produce surface and contour plots of $J(\theta)$ using the `surf` and `contour` commands. The plots should look something like Figure 3:

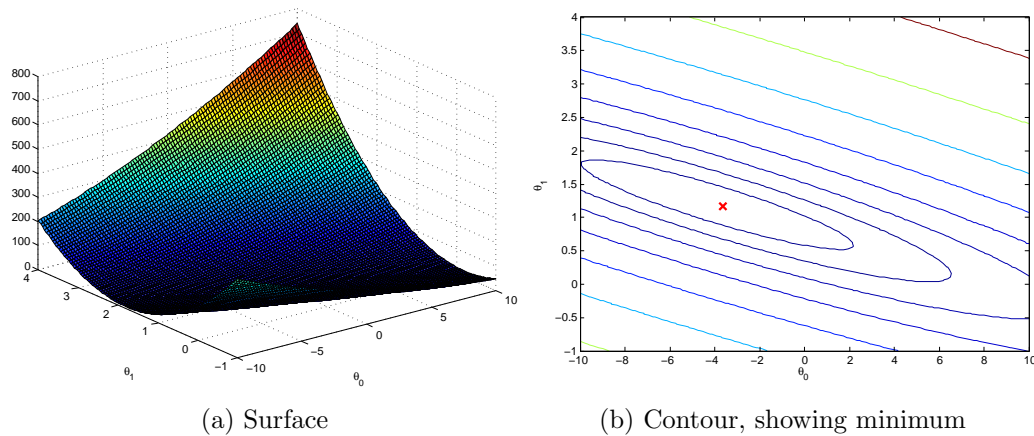


Figure 3: Cost function $J(\theta)$

The purpose of these graphs is to show you that how $J(\theta)$ varies with changes in θ_0 and θ_1 . The cost function $J(\theta)$ is bowl-shaped and has a global minimum. (This is easier to see in the contour plot than in the 3D surface plot). This minimum is the optimal point for θ_0 and θ_1 , and each step of gradient descent moves closer to this point.

Extra Credit Exercises (optional)

If you have successfully completed the material above, congratulations! You now understand linear regression and should be able to start using it on your own datasets.

For the rest of this programming exercise, we have included the following optional extra credit exercises. These exercises will help you gain a deeper understanding of the material, and if you are able to do so, we encourage you to complete them as well.

3 Linear regression with multiple variables

In this part, you will implement linear regression with multiple variables to predict the prices of houses. Suppose you are selling your house and you want to know what a good market price would be. One way to do this is to first collect information on recent houses sold and make a model of housing prices.

The file `ex1data2.txt` contains a training set of housing prices in Portland, Oregon. The first column is the size of the house (in square feet), the second column is the number of bedrooms, and the third column is the price of the house.

The `ex1_multi.m` script has been set up to help you step through this exercise.

3.1 Feature Normalization

The `ex1_multi.m` script will start by loading and displaying some values from this dataset. By looking at the values, note that house sizes are about 1000 times the number of bedrooms. When features differ by orders of magnitude, first performing feature scaling can make gradient descent converge much more quickly.

Your task here is to complete the code in `featureNormalize.m` to

- Subtract the mean value of each feature from the dataset.
- After subtracting the mean, additionally scale (divide) the feature values by their respective “standard deviations.”

The standard deviation is a way of measuring how much variation there is in the range of values of a particular feature (most data points will lie within ± 2 standard deviations of the mean); this is an alternative to taking the range of values (max-min). In Octave, you can use the “`std`” function to compute the standard deviation. For example, inside `featureNormalize.m`, the quantity `X(:,1)` contains all the values of x_1 (house sizes) in the training set, so `std(X(:,1))` computes the standard deviation of the house sizes. At the time that `featureNormalize.m` is called, the extra column of 1’s corresponding to $x_0 = 1$ has not yet been added to `X` (see `ex1_multi.m` for details).

You will do this for all the features and your code should work with datasets of all sizes (any number of features / examples). Note that each column of the matrix `X` corresponds to one feature.

You should now submit feature normalization.

Implementation Note: When normalizing the features, it is important to store the values used for normalization - the *mean value* and the *standard deviation* used for the computations. After learning the parameters from the model, we often want to predict the prices of houses we have not seen before. Given a new `x` value (living room area and number of bedrooms), we must first normalize `x` using the mean and standard deviation that we had previously computed from the training set.

3.2 Gradient Descent

Previously, you implemented gradient descent on a univariate regression problem. The only difference now is that there is one more feature in the matrix `X`. The hypothesis function and the batch gradient descent update rule remain unchanged.

You should complete the code in `computeCostMulti.m` and `gradientDescentMulti.m` to implement the cost function and gradient descent for linear regression with multiple variables. If your code in the previous part (single variable) already supports multiple variables, you can use it here too.

Make sure your code supports any number of features and is well-vectorized. You can use `size(X, 2)` to find out how many features are present in the dataset.

You should now submit compute cost and gradient descent for linear regression with multiple variables.

Implementation Note: In the multivariate case, the cost function can also be written in the following vectorized form:

$$J(\theta) = \frac{1}{2m} (X\theta - \vec{y})^T (X\theta - \vec{y})$$

where

$$X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} \quad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

The vectorized version is efficient when you're working with numerical computing tools like Octave. If you are an expert with matrix operations, you can prove to yourself that the two forms are equivalent.

3.2.1 Optional (ungraded) exercise: Selecting learning rates

In this part of the exercise, you will get to try out different learning rates for the dataset and find a learning rate that converges quickly. You can change the learning rate by modifying `ex1_multi.m` and changing the part of the code that sets the learning rate.

The next phase in `ex1_multi.m` will call your `gradientDescent.m` function and run gradient descent for about 50 iterations at the chosen learning rate. The function should also return the history of $J(\theta)$ values in a vector `J`. After the last iteration, the `ex1_multi.m` script plots the `J` values against the number of the iterations.

If you picked a learning rate within a good range, your plot look similar Figure 4. If your graph looks very different, especially if your value of $J(\theta)$ increases or even blows up, adjust your learning rate and try again. We recommend trying values of the learning rate α on a log-scale, at multiplicative steps of about 3 times the previous value (i.e., 0.3, 0.1, 0.03, 0.01 and so on). You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve.

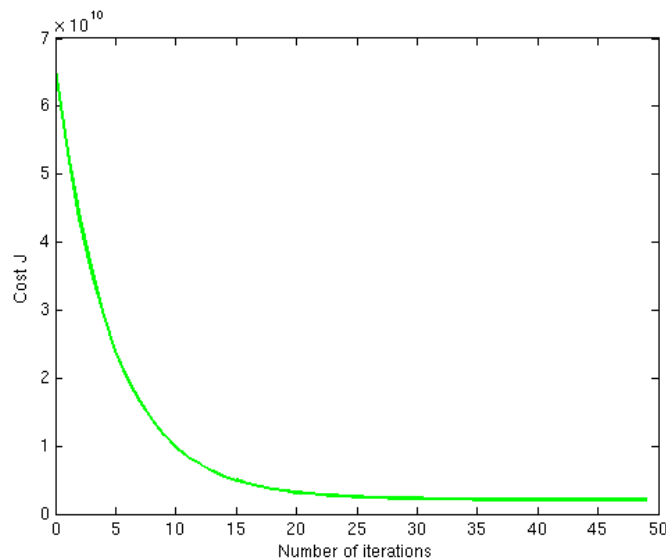


Figure 4: Convergence of gradient descent with an appropriate learning rate

Implementation Note: If your learning rate is too large, $J(\theta)$ can diverge and ‘blow up’, resulting in values which are too large for computer calculations. In these situations, Octave will tend to return NaNs. NaN stands for ‘not a number’ and is often caused by undefined operations that involve $-\infty$ and $+\infty$.

Octave Tip: To compare how different learning rates affect convergence, it’s helpful to plot J for several learning rates on the same figure. In Octave, this can be done by performing gradient descent multiple times with a ‘hold on’ command between plots. Concretely, if you’ve tried three different values of alpha (you should probably try more values than this) and stored the costs in J1, J2 and J3, you can use the following commands to plot them on the same figure:

```
plot(1:50, J1(1:50), 'b');
hold on;
plot(1:50, J2(1:50), 'r');
plot(1:50, J3(1:50), 'k');
```

The final arguments ‘b’, ‘r’, and ‘k’ specify different colors for the plots.

Notice the changes in the convergence curves as the learning rate changes. With a small learning rate, you should find that gradient descent takes a very long time to converge to the optimal value. Conversely, with a large learning rate, gradient descent might not converge or might even diverge!

Using the best learning rate that you found, run the `ex1_multi.m` script to run gradient descent until convergence to find the final values of θ . Next, use this value of θ to predict the price of a house with 1650 square feet and 3 bedrooms. You will use value later to check your implementation of the normal equations. Don't forget to normalize your features when you make this prediction!

You do not need to submit any solutions for these optional (ungraded) exercises.

3.3 Normal Equations

In the lecture videos, you learned that the closed-form solution to linear regression is

$$\theta = (X^T X)^{-1} X^T \vec{y}.$$

Using this formula does not require any feature scaling, and you will get an exact solution in one calculation: there is no “loop until convergence” like in gradient descent.

Complete the code in `normalEqn.m` to use the formula above to calculate θ . Remember that while you don't need to scale your features, we still need to add a column of 1's to the X matrix to have an intercept term (θ_0). The code in `ex1.m` will add the column of 1's to X for you.

You should now submit the normal equations function.

Optional (ungraded) exercise: Now, once you have found θ using this method, use it to make a price prediction for a 1650-square-foot house with 3 bedrooms. You should find that gives the same predicted price as the value you obtained using the model fit with gradient descent (in Section 3.2.1).

Submission and Grading

After completing various parts of the assignment, be sure to use the `submit` function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Warm up exercise	<code>warmUpExercise.m</code>	10 points
Compute cost for one variable	<code>computeCost.m</code>	40 points
Gradient descent for one variable	<code>gradientDescent.m</code>	50 points
Total Points		100 points

Extra Credit Exercises (optional)

Feature normalization	<code>featureNormalize.m</code>	10 points
Compute cost for multiple variables	<code>computeCostMulti.m</code>	15 points
Gradient descent for multiple variables	<code>gradientDescentMulti.m</code>	15 points
Normal Equations	<code>normalEqn.m</code>	10 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration. To prevent rapid-fire guessing, the system enforces a minimum of 5 minutes between submissions.

Programming Exercise 2: Logistic Regression

Machine Learning

Introduction

In this exercise, you will implement logistic regression and apply it to two different datasets. Before starting on the programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics.

To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise. If needed, use the `cd` command in Octave to change to this directory before starting this exercise.

You can also find instructions for installing Octave on the “Octave Installation” page on the course website.

Files included in this exercise

- `ex2.m` - Octave script that will help step you through the exercise
- `ex2_reg.m` - Octave script for the later parts of the exercise
- `ex2data1.txt` - Training set for the first half of the exercise
- `ex2data2.txt` - Training set for the second half of the exercise
- `submit.m` - Submission script that sends your solutions to our servers
- `mapFeature.m` - Function to generate polynomial features
- `plotDecisionBoundary.m` - Function to plot classifier’s decision boundary
- [*] `plotData.m` - Function to plot 2D classification data
- [*] `sigmoid.m` - Sigmoid Function
- [*] `costFunction.m` - Logistic Regression Cost Function
- [*] `predict.m` - Logistic Regression Prediction Function
- [*] `costFunctionReg.m` - Regularized Logistic Regression Cost

* indicates files you will need to complete

Throughout the exercise, you will be using the scripts `ex2.m` and `ex2_reg.m`. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify either of them. You are only required to modify functions in other files, by following the instructions in this assignment.

Where to get help

The exercises in this course use Octave,¹ a high-level programming language well-suited for numerical computations. If you do not have Octave installed, please refer to the installation instructions at the “Octave Installation” page on the course website.

At the Octave 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](#).

We also strongly encourage using the online **Q&A Forum** to discuss exercises with other students. However, do not look at any source code written by others or share your source code with others.

If you run into network errors using the `submit` script, you can also use an online form for submitting your solutions. To use this *alternative* submission interface, run the `submitWeb` script to generate a submission file (e.g., `submit_ex2_part1.txt`). You can then submit this file through the web submission form in the programming exercises page (go to the programming exercises page, then select the exercise you are submitting for). If you are having no problems submitting through the standard submission system using the `submit` script, you do *not* need to use this alternative submission interface.

1 Logistic Regression

In this part of the exercise, you will build a logistic regression model to predict whether a **student gets admitted into a university**.

Suppose that you are the administrator of a university department and you want to determine each applicant’s chance of admission based on their

¹Octave is a free alternative to MATLAB. For the programming exercises, you are free to use either Octave or MATLAB.

results on two exams. You have historical data from previous applicants that you can use as a training set for **logistic regression**. For each training example, you have the applicant's scores on **two exams** and the admissions decision.

Your task is to build a classification model that estimates an applicant's probability of admission **based the scores from those two exams**. This outline and the framework code in `ex2.m` will guide you through the exercise.

1.1 Visualizing the data

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. In the first part of `ex2.m`, the code will load the data and display it on a 2-dimensional plot by calling the function `plotData`.

You will now complete the code in `plotData` so that it displays a figure like Figure 1, where the axes are the two exam scores, and the positive and negative examples are shown with different markers.

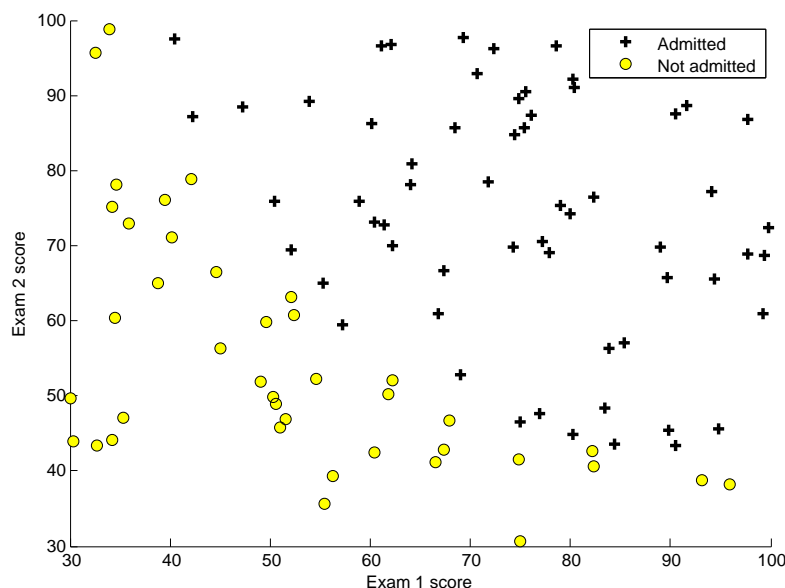


Figure 1: Scatter plot of training data

To help you get more familiar with plotting, we have left `plotData.m` empty so you can try to implement it yourself. However, this is an *optional (ungraded) exercise*. We also provide our implementation below so you can copy it or refer to it. If you choose to copy our example, make sure you learn what each of its commands is doing by consulting the Octave documentation.

```
% Find Indices of Positive and Negative Examples
pos = find(y==1); neg = find(y == 0);

% Plot Examples
plot(X(pos, 1), X(pos, 2), 'k+', 'LineWidth', 2, ...
     'MarkerSize', 7);
plot(X(neg, 1), X(neg, 2), 'ko', 'MarkerFaceColor', 'y', ...
     'MarkerSize', 7);
```

1.2 Implementation

1.2.1 Warmup exercise: sigmoid function

Before you start with the actual cost function, recall that the logistic regression hypothesis is defined as:

$$h_{\theta}(x) = g(\theta^T x),$$

where function g is the sigmoid function. The sigmoid function is defined as:

$$g(z) = \frac{1}{1 + e^{-z}}.$$

Your first step is to implement this function in `sigmoid.m` so it can be called by the rest of your program. When you are finished, try testing a few values by calling `sigmoid(x)` at the octave command line. For large positive values of x , the sigmoid should be close to 1, while for large negative values, the sigmoid should be close to 0. Evaluating `sigmoid(0)` should give you exactly 0.5. Your code should also work with vectors and matrices. **For a matrix, your function should perform the sigmoid function on every element.**

You can submit your solution for grading by typing `submit` at the Octave command line. The submission script will prompt you for your username and password and ask you which files you want to submit. You can obtain a submission password from the website.

You should now submit the warm up exercise.

1.2.2 Cost function and gradient

Now you will implement the cost function and gradient for logistic regression. Complete the code in `costFunction.m` to return the cost and gradient.

Recall that the cost function in logistic regression is

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m [-y^{(i)} \log(h_{\theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))] ,$$

and the gradient of the cost is a vector of the same length as θ where the j^{th} element (for $j = 0, 1, \dots, n$) is defined as follows:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Note that while this gradient looks identical to the linear regression gradient, the formula is actually different because linear and logistic regression have different definitions of $h_{\theta}(x)$.

Once you are done, `ex2.m` will call your `costFunction` using the initial parameters of θ . You should see that the cost is about 0.693.

You should now submit the cost function and gradient for logistic regression. Make two submissions: one for the cost function and one for the gradient.

1.2.3 Learning parameters using `fminunc`

In the previous assignment, you found the optimal parameters of a linear regression model by implementing gradient descent. You wrote a cost function and calculated its gradient, then took a gradient descent step accordingly. This time, instead of taking gradient descent steps, you will use an Octave built-in function called `fminunc`.

Octave's `fminunc` is an optimization solver that finds the minimum of an unconstrained² function. For logistic regression, you want to optimize the cost function $J(\theta)$ with parameters θ .

Concretely, you are going to use `fminunc` to find the best parameters θ for the logistic regression cost function, given a fixed dataset (of X and y values). You will pass to `fminunc` the following inputs:

- The initial values of the parameters we are trying to optimize.

²Constraints in optimization often refer to constraints on the parameters, for example, constraints that bound the possible values θ can take (e.g., $\theta \leq 1$). Logistic regression does not have such constraints since θ is allowed to take any real value.

- A function that, when given the training set and a particular θ , computes the logistic regression cost and gradient with respect to θ for the dataset (X, y)

In `ex2.m`, we already have code written to call `fminunc` with the correct arguments.

```
% Set options for fminunc
options = optimset('GradObj', 'on', 'MaxIter', 400);

% Run fminunc to obtain the optimal theta
% This function will return theta and the cost
[theta, cost] = ...
    fminunc(@(t)(costFunction(t, X, y)), initial_theta, options);
```

In this code snippet, we first defined the options to be used with `fminunc`. Specifically, we set the `GradObj` option to `on`, which tells `fminunc` that our function returns both the cost and the gradient. This allows `fminunc` to use the gradient when minimizing the function. Furthermore, we set the `MaxIter` option to 400, so that `fminunc` will run for at most 400 steps before it terminates.

To specify the actual function we are minimizing, we use a “short-hand” for specifying functions with the `@(t) (costFunction(t, X, y))`. This creates a function, with argument `t`, which calls your `costFunction`. This allows us to wrap the `costFunction` for use with `fminunc`.

If you have completed the `costFunction` correctly, `fminunc` will converge on the right optimization parameters and return the final values of the cost and θ . Notice that by using `fminunc`, you did not have to write any loops yourself, or set a learning rate like you did for gradient descent. **This is all done by `fminunc`: you only needed to provide a function calculating the cost and the gradient.**

Once `fminunc` completes, `ex2.m` will call your `costFunction` function using the optimal parameters of θ . You should see that the cost is about 0.203.

This final θ value will then be used to plot the decision boundary on the training data, resulting in a figure similar to Figure 2. We also encourage you to look at the code in `plotDecisionBoundary.m` to see how to plot such a boundary using the θ values.

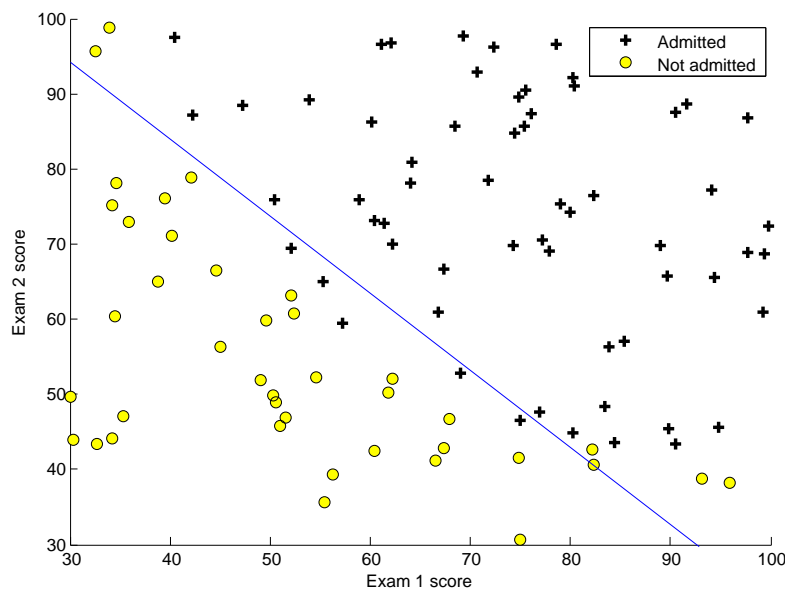


Figure 2: Training data with decision boundary

1.2.4 Evaluating logistic regression

After learning the parameters, you can use the model to predict whether a particular student will be admitted. For a student with an Exam 1 score of 45 and an Exam 2 score of 85, you should expect to see an admission probability of 0.776.

Another way to evaluate the quality of the parameters we have found is to see how well the learned model predicts on our training set. In this part, your task is to complete the code in `predict.m`. The `predict` function will produce “1” or “0” predictions given a dataset and a learned parameter vector θ .

After you have completed the code in `predict.m`, the `ex2.m` script will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct.

You should now submit the prediction function for logistic regression.

2 Regularized logistic regression

In this part of the exercise, you will implement regularized logistic regression to predict whether microchips from a fabrication plant passes quality assurance (QA). During QA, each microchip goes through various tests to ensure it is functioning correctly.

Suppose you are the product manager of the factory and you have the test results for some microchips on two different tests. From these two tests, you would like to determine whether the microchips should be accepted or rejected. To help you make the decision, you have a dataset of test results on past microchips, from which you can build a logistic regression model.

You will use another script, `ex2_reg.m` to complete this portion of the exercise.

2.1 Visualizing the data

Similar to the previous parts of this exercise, `plotData` is used to generate a figure like Figure 3, where the axes are the two test scores, and the positive ($y = 1$, accepted) and negative ($y = 0$, rejected) examples are shown with different markers.

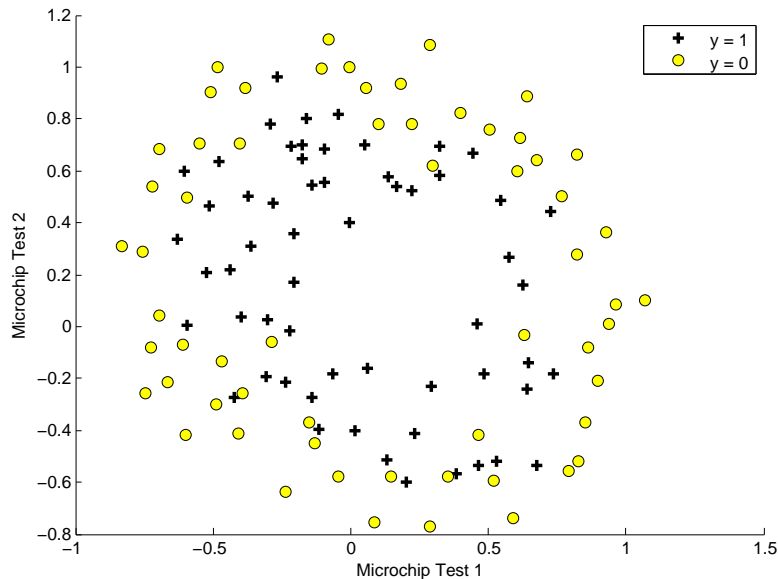


Figure 3: Plot of training data

Figure 3 shows that our dataset cannot be separated into positive and

negative examples by a straight-line through the plot. Therefore, a straightforward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

2.2 Feature mapping

One way to fit the data better is to create more features from each data point. In the provided function `mapFeature.m`, we will map the features into all polynomial terms of x_1 and x_2 up to the sixth power.

$$\text{mapFeature}(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1x_2^5 \\ x_2^6 \end{bmatrix}$$

As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 28-dimensional vector. A logistic regression classifier trained on this higher-dimension feature vector will have a more complex decision boundary and will appear nonlinear when drawn in our 2-dimensional plot.

While the feature mapping allows us to build a more expressive classifier, it also more susceptible to overfitting. In the next parts of the exercise, you will implement regularized logistic regression to fit the data and also see for yourself how regularization can help combat the overfitting problem.

2.3 Cost function and gradient

Now you will implement code to compute the cost function and gradient for regularized logistic regression. Complete the code in `costFunctionReg.m` to return the cost and gradient.

Recall that the regularized cost function in logistic regression is

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m [-y^{(i)} \log(h_{\theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2.$$

Note that you should not regularize the parameter θ_0 . In **Octave**, recall that indexing starts from 1, hence, you should not be regularizing the `theta(1)` parameter (which corresponds to θ_0) in the code. The gradient of the cost function is a vector where the j^{th} element is defined as follows:

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad \text{for } j = 0$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \quad \text{for } j \geq 1$$

Once you are done, `ex2_reg.m` will call your `costFunctionReg` function using the initial value of θ (initialized to all zeros). You should see that the cost is about 0.693.

You should now submit the cost function and gradient for regularized logistic regression. Make two submissions, one for the cost function and one for the gradient.

2.3.1 Learning parameters using `fminunc`

Similar to the previous parts, you will use `fminunc` to learn the optimal parameters θ . If you have completed the cost and gradient for regularized logistic regression (`costFunctionReg.m`) correctly, you should be able to step through the next part of `ex2_reg.m` to learn the parameters θ using `fminunc`.

2.4 Plotting the decision boundary

To help you visualize the model learned by this classifier, we have provided the function `plotDecisionBoundary.m` which plots the (non-linear) decision boundary that separates the positive and negative examples. In `plotDecisionBoundary.m`, we plot the non-linear decision boundary by computing the classifier's predictions on an evenly spaced grid and then and drew a contour plot of where the predictions change from $y = 0$ to $y = 1$.

After learning the parameters θ , the next step in `ex_reg.m` will plot a decision boundary similar to Figure 4.

2.5 Optional (ungraded) exercises

In this part of the exercise, you will get to try out different regularization parameters for the dataset to understand how regularization prevents overfitting.

Notice the changes in the decision boundary as you vary λ . With a small λ , you should find that the classifier gets almost every training example correct, but draws a very complicated boundary, thus overfitting the data (Figure 5). This is not a good decision boundary: for example, it predicts that a point at $x = (-0.25, 1.5)$ is accepted ($y = 1$), which seems to be an incorrect decision given the training set.

With a larger λ , you should see a plot that shows a simpler decision boundary which still separates the positives and negatives fairly well. However, if λ is set to too high a value, you will not get a good fit and the decision boundary will not follow the data so well, thus underfitting the data (Figure 6).

You do not need to submit any solutions for these optional (ungraded) exercises.

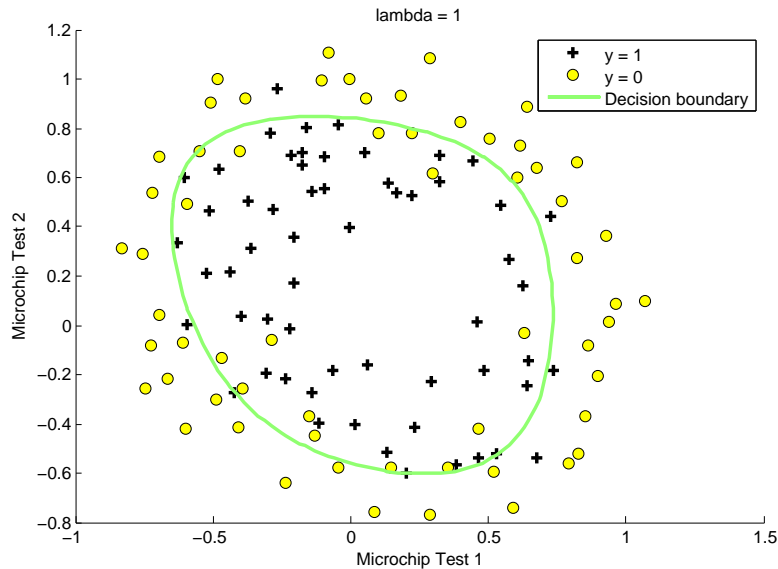


Figure 4: Training data with decision boundary ($\lambda = 1$)

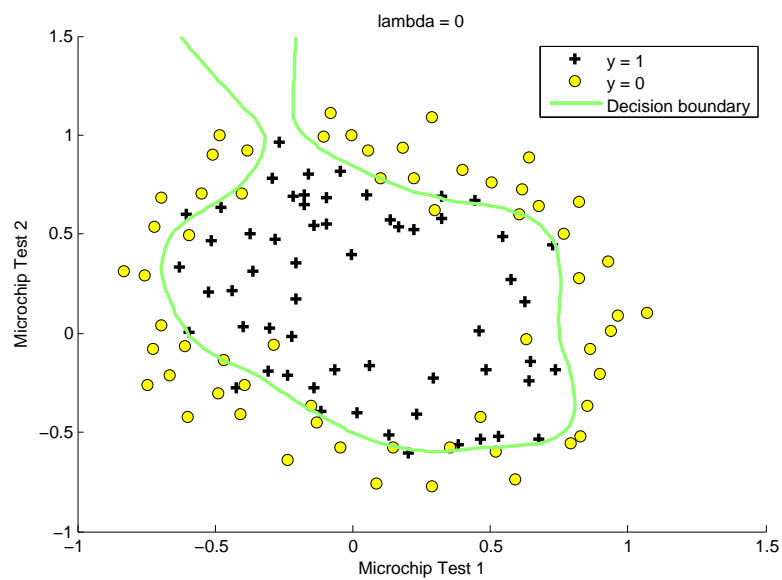


Figure 5: No regularization (Overfitting) ($\lambda = 0$)

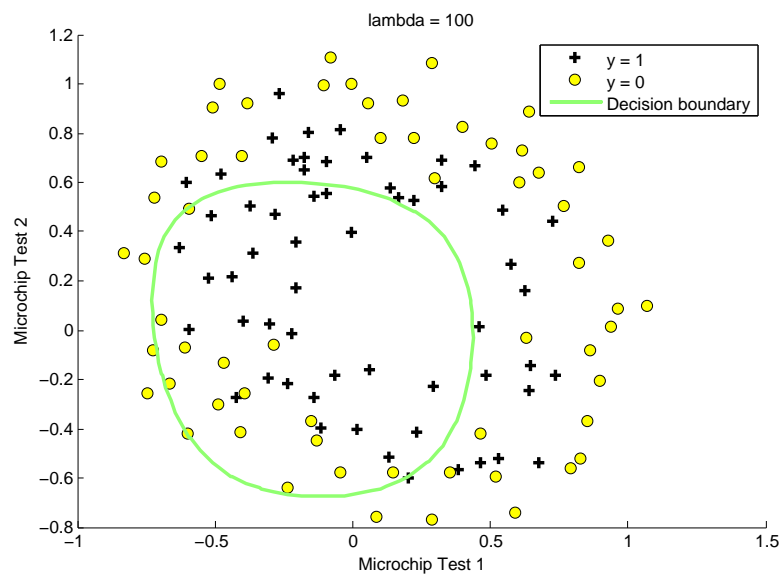


Figure 6: Too much regularization (Underfitting) ($\lambda = 100$)

Submission and Grading

After completing various parts of the assignment, be sure to use the `submit` function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Sigmoid Function	<code>sigmoid.m</code>	5 points
Compute cost for logistic regression	<code>costFunction.m</code>	30 points
Gradient for logistic regression	<code>costFunction.m</code>	30 points
Predict Function	<code>predict.m</code>	5 points
Compute cost for regularized LR	<code>costFunctionReg.m</code>	15 points
Gradient for regularized LR	<code>costFunctionReg.m</code>	15 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration. To prevent rapid-fire guessing, the system enforces a minimum of 5 minutes between submissions.

Programming Exercise 3: Multi-class Classification and Neural Networks

Machine Learning

Introduction

In this exercise, you will implement one-vs-all logistic regression and neural networks to recognize hand-written digits. Before starting the programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics.

To get started with the exercise, download the starter code and unzip its contents to the directory where you wish to complete the exercise. If needed, use the `cd` command in Octave to change to this directory before starting this exercise.

Files included in this exercise

- `ex3.m` - Octave script that will help step you through part 1
- `ex3_nn.m` - Octave script that will help step you through part 2
- `ex3data1.mat` - Training set of hand-written digits
- `ex3weights.mat` - Initial weights for the neural network exercise
- `submitWeb.m` - Alternative submission script
- `submit.m` - Submission script that sends your solutions to our servers
- `displayData.m` - Function to help visualize the dataset
- `fmincg.m` - Function minimization routine (similar to `fminunc`)
- `sigmoid.m` - Sigmoid function
- [*] `lrCostFunction.m` - Logistic regression cost function
- [*] `oneVsAll.m` - Train a one-vs-all multi-class classifier
- [*] `predictOneVsAll.m` - Predict using a one-vs-all multi-class classifier
- [*] `predict.m` - Neural network prediction function

* indicates files you will need to complete

Throughout the exercise, you will be using the scripts `ex3.m` and `ex3_nn.m`. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify these scripts. You are only required to modify functions in other files, by following the instructions in this assignment.

Where to get help

We also strongly encourage using the online **Q&A Forum** to discuss exercises with other students. However, do not look at any source code written by others or share your source code with others.

If you run into network errors using the `submit` script, you can also use an online form for submitting your solutions. To use this *alternative* submission interface, run the `submitWeb` script to generate a submission file (e.g., `submit_ex2_part1.txt`). You can then submit this file through the web submission form in the programming exercises page (go to the programming exercises page, then select the exercise you are submitting for). If you are having no problems submitting through the standard submission system using the `submit` script, you do *not* need to use this alternative submission interface.

1 Multi-class Classification

For this exercise, you will use logistic regression and neural networks to recognize handwritten digits (from 0 to 9). Automated handwritten digit recognition is widely used today - from recognizing zip codes (postal codes) on mail envelopes to recognizing amounts written on bank checks. This exercise will show you how the methods you've learned can be used for this classification task.

In the first part of the exercise, you will extend your previous implementation of logistic regression and apply it to one-vs-all classification.

1.1 Dataset

You are given a data set in `ex3data1.mat` that contains 5000 training examples of handwritten digits.¹ The `.mat` format means that the data

¹This is a subset of the MNIST handwritten digit dataset (<http://yann.lecun.com/exdb/mnist/>).

has been saved in a native Octave/Matlab matrix format, instead of a text (ASCII) format like a csv-file. These matrices can be read directly into your program by using the `load` command. After loading, matrices of the correct dimensions and values will appear in your program’s memory. The matrix will already be named, so you do not need to assign names to them.

```
% Load saved matrices from file
load('ex3data1.mat');
% The matrices X and y will now be in your Octave environment
```

There are 5000 training examples in `ex3data1.mat`, where each training example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is “unrolled” into a 400-dimensional vector. Each of these training examples becomes a single row in our data matrix `X`. This gives us a 5000 by 400 matrix `X` where every row is a training example for a handwritten digit image.

$$X = \begin{bmatrix} \text{---} (x^{(1)})^T \text{---} \\ \text{---} (x^{(2)})^T \text{---} \\ \vdots \\ \text{---} (x^{(m)})^T \text{---} \end{bmatrix}$$

The second part of the training set is a 5000-dimensional vector `y` that contains labels for the training set. To make things more compatible with Octave/Matlab indexing, where there is no zero index, we have mapped the digit zero to the value ten. Therefore, a “0” digit is labeled as “10”, while the digits “1” to “9” are labeled as “1” to “9” in their natural order.

1.2 Visualizing the data

You will begin by visualizing a subset of the training set. In Part 1 of `ex3.m`, the code randomly selects selects 100 rows from `X` and passes those rows to the `displayData` function. This function maps each row to a 20 pixel by 20 pixel grayscale image and displays the images together. We have provided the `displayData` function, and you are encouraged to examine the code to see how it works. After you run this step, you should see an image like Figure 1.



Figure 1: Examples from the dataset

1.3 Vectorizing Logistic Regression

You will be using multiple one-vs-all logistic regression models to build a multi-class classifier. Since there are 10 classes, you will need to train 10 separate logistic regression classifiers. To make this training efficient, it is important to ensure that your code is well vectorized. In this section, you will implement a vectorized version of logistic regression that does not employ any `for` loops. You can use your code in the last exercise as a starting point for this exercise.

1.3.1 Vectorizing the cost function

We will begin by writing a vectorized version of the cost function. Recall that in (unregularized) logistic regression, the cost function is

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m [-y^{(i)} \log(h_{\theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))].$$

To compute each element in the summation, we have to compute $h_{\theta}(x^{(i)})$ for every example i , where $h_{\theta}(x^{(i)}) = g(\theta^T x^{(i)})$ and $g(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function. It turns out that we can compute this quickly for all our examples by using matrix multiplication. Let us define X and θ as

$$X = \begin{bmatrix} - & (x^{(1)})^T & - \\ - & (x^{(2)})^T & - \\ & \vdots & \\ - & (x^{(m)})^T & - \end{bmatrix} \quad \text{and} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}.$$

Then, by computing the matrix product $X\theta$, we have

$$X\theta = \begin{bmatrix} - & (x^{(1)})^T \theta & - \\ - & (x^{(2)})^T \theta & - \\ & \vdots & \\ - & (x^{(m)})^T \theta & - \end{bmatrix} = \begin{bmatrix} - & \theta^T(x^{(1)}) & - \\ - & \theta^T(x^{(2)}) & - \\ & \vdots & \\ - & \theta^T(x^{(m)}) & - \end{bmatrix}.$$

In the last equality, we used the fact that $a^T b = b^T a$ if a and b are vectors. This allows us to compute the products $\theta^T x^{(i)}$ for all our examples i in one line of code.

Your job is to write the unregularized cost function in the file `lrCostFunction.m`. Your implementation should use the strategy we presented above to calculate $\theta^T x^{(i)}$. You should also use a vectorized approach for the rest of the cost function. A fully vectorized version of `lrCostFunction.m` should not contain any loops.

(Hint: You might want to use the element-wise multiplication operation `(.*)` and the sum operation `sum` when writing this function)

1.3.2 Vectorizing the gradient

Recall that the gradient of the (unregularized) logistic regression cost is a vector where the j^{th} element is defined as

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^m \left((h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \right).$$

To vectorize this operation over the dataset, we start by writing out all

the partial derivatives explicitly for all θ_j ,

$$\begin{aligned}
\begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{bmatrix} &= \frac{1}{m} \begin{bmatrix} \sum_{i=1}^m \left((h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)} \right) \\ \sum_{i=1}^m \left((h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)} \right) \\ \sum_{i=1}^m \left((h_\theta(x^{(i)}) - y^{(i)})x_2^{(i)} \right) \\ \vdots \\ \sum_{i=1}^m \left((h_\theta(x^{(i)}) - y^{(i)})x_n^{(i)} \right) \end{bmatrix} \\
&= \frac{1}{m} \sum_{i=1}^m ((h_\theta(x^{(i)}) - y^{(i)})x^{(i)}) \\
&= \frac{1}{m} X^T (h_\theta(x) - y). \tag{1}
\end{aligned}$$

where

$$h_\theta(x) - y = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}.$$

Note that $x^{(i)}$ is a vector, while $(h_\theta(x^{(i)}) - y^{(i)})$ is a scalar (single number). To understand the last step of the derivation, let $\beta_i = (h_\theta(x^{(i)}) - y^{(i)})$ and observe that:

$$\sum_i \beta_i x^{(i)} = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} = X^T \beta,$$

where the values $\beta_i = (h_\theta(x^{(i)}) - y^{(i)})$.

The expression above allows us to compute all the partial derivatives without any loops. If you are comfortable with linear algebra, we encourage you to work through the matrix multiplications above to convince yourself that the vectorized version does the same computations. You should now implement Equation 1 to compute the correct vectorized gradient. Once you are done, complete the function `lrCostFunction.m` by implementing the gradient.

Debugging Tip: Vectorizing code can sometimes be tricky. One common strategy for debugging is to print out the sizes of the matrices you are working with using the `size` function. For example, given a data matrix X of size 100×20 (100 examples, 20 features) and θ , a vector with dimensions 20×1 , you can observe that $X\theta$ is a valid multiplication operation, while θX is not. Furthermore, if you have a non-vectorized version of your code, you can compare the output of your vectorized code and non-vectorized code to make sure that they produce the same outputs.

1.3.3 Vectorizing regularized logistic regression

After you have implemented vectorization for logistic regression, you will now add regularization to the cost function. Recall that for regularized logistic regression, the cost function is defined as

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m [-y^{(i)} \log(h_{\theta}(x^{(i)})) - (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2.$$

Note that you should *not* be regularizing θ_0 which is used for the bias term.

Correspondingly, the partial derivative of regularized logistic regression cost for θ_j is defined as

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta_0} &= \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} && \text{for } j = 0 \\ \frac{\partial J(\theta)}{\partial \theta_j} &= \left(\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j && \text{for } j \geq 1 \end{aligned}$$

Now modify your code in `lrCostFunction` to account for regularization. Once again, you should not put any loops into your code.

Octave Tip: When implementing the vectorization for regularized logistic regression, you might often want to only sum and update certain elements of θ . In Octave, **you can index into the matrices to access and update only certain elements**. For example, `A(:, 3:5) = B(:, 1:3)` will replace the columns 3 to 5 of A with the columns 1 to 3 from B . One special keyword you can use in indexing is the `end` keyword in indexing. This allows us to select columns (or rows) until the end of the matrix. For example, `A(:, 2:end)` will only return elements from the 2nd to last column of A . Thus, you could use this together with the `sum` and `.^` operations to compute the sum of only the elements you are interested in (e.g., `sum(z(2:end).^2)`). In the starter code, `lrCostFunction.m`, we have also provided hints on yet *another* possible method computing the regularized gradient.

You should now submit your vectorized logistic regression cost function.

1.4 One-vs-all Classification

In this part of the exercise, you will implement one-vs-all classification by training multiple regularized logistic regression classifiers, one for each of the K classes in our dataset (Figure 1). In the handwritten digits dataset, $K = 10$, but your code should work for any value of K .

You should now complete the code in `oneVsAll.m` to train one classifier for each class. In particular, your code should return all the classifier parameters in a matrix $\Theta \in \mathbb{R}^{K \times (N+1)}$, where each row of Θ corresponds to the learned logistic regression parameters for one class. You can do this with a “for”-loop from 1 to K , training each classifier independently.

Note that the `y` argument to this function is a vector of labels from 1 to 10, where we have mapped the digit “0” to the label 10 (to avoid confusions with indexing).

When training the classifier for class $k \in \{1, \dots, K\}$, you will want a m -dimensional vector of labels y , where $y_j \in \{0, 1\}$ indicates whether the j -th training instance belongs to class k ($y_j = 1$), or if it belongs to a different class ($y_j = 0$). You may find logical arrays helpful for this task.

Octave Tip: Logical arrays in Octave are arrays which contain binary (0 or 1) elements. In Octave, evaluating the expression `a == b` for a vector `a` (of size $m \times 1$) and scalar `b` will return a vector of the same size as `a` with ones at positions where the elements of `a` are equal to `b` and zeroes where they are different. To see how this works for yourself, try the following code in Octave:

```
a = 1:10; % Create a and b
b = 3;
a == b    % You should try different values of b here
```

Furthermore, you will be using `fmincg` for this exercise (instead of `fminunc`). `fmincg` works similarly to `fminunc`, but is more more efficient for dealing with a large number of parameters.

After you have correctly completed the code for `oneVsAll.m`, the script `ex3.m` will continue to use your `oneVsAll` function to train a multi-class classifier.

You should now submit the training function for one-vs-all classification.

1.4.1 One-vs-all Prediction

After training your one-vs-all classifier, you can now use it to predict the digit contained in a given image. For each input, you should compute the “probability” that it belongs to each class using the trained logistic regression classifiers. Your one-vs-all prediction function will pick the class for which the corresponding logistic regression classifier outputs the highest probability and return the class label (1, 2,..., or K) as the prediction for the input example.

You should now complete the code in `predictOneVsAll.m` to use the one-vs-all classifier to make predictions.

Once you are done, `ex3.m` will call your `predictOneVsAll` function using the learned value of Θ . You should see that the training set accuracy is about 94.9% (i.e., it classifies 94.9% of the examples in the training set correctly).

You should now submit the prediction function for one-vs-all classification.

2 Neural Networks

In the previous part of this exercise, you implemented multi-class logistic regression to recognize handwritten digits. However, logistic regression cannot form more complex hypotheses as it is only a linear classifier.²

In this part of the exercise, you will implement a neural network to recognize handwritten digits using the same training set as before. The neural network will be able to represent complex models that form non-linear hypotheses. For this week, you will be using parameters from a neural network that we have already trained. Your goal is to implement the feedforward propagation algorithm to use our weights for prediction. In next week's exercise, you will write the backpropagation algorithm for learning the neural network parameters.

The provided script, `ex3_nn.m`, will help you step through this exercise.

2.1 Model representation

Our neural network is shown in Figure 2. It has 3 layers – an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20×20 , this gives us 400 input layer units (excluding the extra bias unit which always outputs +1). As before, the training data will be loaded into the variables `X` and `y`.

You have been provided with a set of network parameters ($\Theta^{(1)}, \Theta^{(2)}$) already trained by us. These are stored in `ex3weights.mat` and will be loaded by `ex3_nn.m` into `Theta1` and `Theta2`. The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes).

```
% Load saved matrices from file
load('ex3weights.mat');

% The matrices Theta1 and Theta2 will now be in your Octave
% environment
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```

²You could add more features (such as polynomial features) to logistic regression, but that can be very expensive to train.

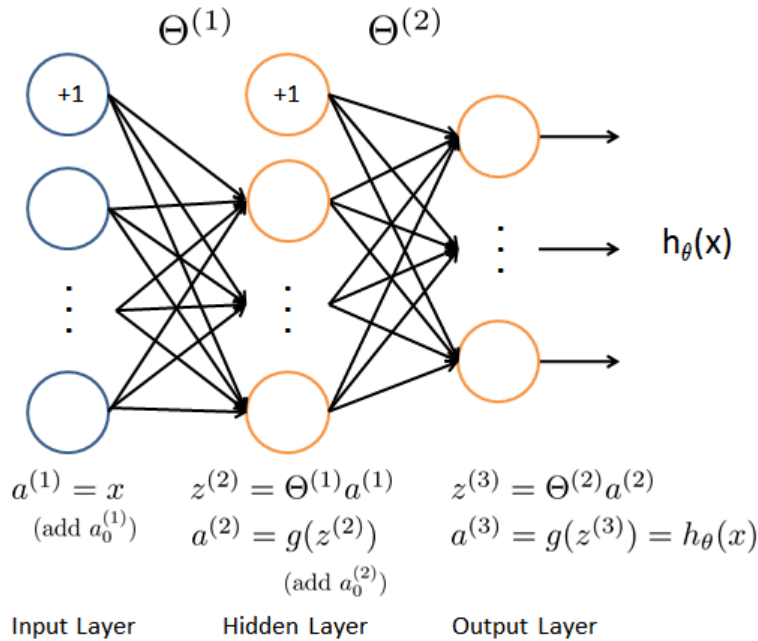


Figure 2: Neural network model.

2.2 Feedforward Propagation and Prediction

Now you will implement feedforward propagation for the neural network. You will need to complete the code in `predict.m` to return the neural network's prediction.

You should implement the feedforward computation that computes $h_{\theta}(x^{(i)})$ for every example i and returns the associated predictions. Similar to the one-vs-all classification strategy, the prediction from the neural network will be the label that has the largest output $(h_{\theta}(x))_k$.

Implementation Note: The matrix `X` contains the examples in rows. When you complete the code in `predict.m`, you will need to add the column of 1's to the matrix. The matrices `Theta1` and `Theta2` contain the parameters for each unit in rows. Specifically, the first row of `Theta1` corresponds to the first hidden unit in the second layer. In Octave, when you compute $z^{(2)} = \Theta^{(1)} a^{(1)}$, be sure that you index (and if necessary, transpose) `X` correctly so that you get $a^{(l)}$ as a column vector.

Once you are done, `ex3_nn.m` will call your `predict` function using the loaded set of parameters for `Theta1` and `Theta2`. You should see that the

accuracy is about 97.5%. After that, an interactive sequence will launch displaying images from the training set one at a time, while the console prints out the predicted label for the displayed image. To stop the image sequence, press **Ctrl-C**.

You should now submit the neural network prediction function.

Submission and Grading

After completing this assignment, be sure to use the `submit` function to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Regularized Logistic Regression	<code>lrCostFunction.m</code>	30 points
One-vs-all classifier training	<code>oneVsAll.m</code>	20 points
One-vs-all classifier prediction	<code>predictOneVsAll.m</code>	20 points
Neural Network Prediction Function	<code>predict.m</code>	30 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration. To prevent rapid-fire guessing, the system enforces a minimum of 5 minutes between submissions.

Programming Exercise 4: Neural Networks Learning

Machine Learning

Introduction

In this exercise, you will implement the backpropagation algorithm for neural networks and apply it to the task of hand-written digit recognition. Before starting on the programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics.

To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise. If needed, use the `cd` command in Octave to change to this directory before starting this exercise.

Files included in this exercise

- `ex4.m` - Octave script that will help step you through the exercise
- `ex4data1.mat` - Training set of hand-written digits
- `ex4weights.mat` - Neural network parameters for exercise 4
- `submit.m` - Submission script that sends your solutions to our servers
- `submitWeb.m` - Alternative submission script
- `displayData.m` - Function to help visualize the dataset
- `fmincg.m` - Function minimization routine (similar to `fminunc`)
- `sigmoid.m` - Sigmoid function
- `computeNumericalGradient.m` - Numerically compute gradients
- `checkNNGradients.m` - Function to help check your gradients
- `debugInitializeWeights.m` - Function for initializing weights
- `predict.m` - Neural network prediction function
- [*] `sigmoidGradient.m` - Compute the gradient of the sigmoid function
- [*] `randInitializeWeights.m` - Randomly initialize weights
- [*] `nnCostFunction.m` - Neural network cost function

★ indicates files you will need to complete

Throughout the exercise, you will be using the script `ex4.m`. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify the script. You are only required to modify functions in other files, by following the instructions in this assignment.

Where to get help

We also strongly encourage using the online **Q&A Forum** to discuss exercises with other students. However, do not look at any source code written by others or share your source code with others.

If you run into network errors using the `submit` script, you can also use an online form for submitting your solutions. To use this *alternative* submission interface, run the `submitWeb` script to generate a submission file (e.g., `submit_ex2_part1.txt`). You can then submit this file through the web submission form in the programming exercises page (go to the programming exercises page, then select the exercise you are submitting for). If you are having no problems submitting through the standard submission system using the `submit` script, you do *not* need to use this alternative submission interface.

1 Neural Networks

In the previous exercise, you implemented feedforward propagation for neural networks and used it to predict handwritten digits with the weights we provided. In this exercise, you will implement the backpropagation algorithm to *learn* the parameters for the neural network.

The provided script, `ex4.m`, will help you step through this exercise.

1.1 Visualizing the data

In the first part of `ex4.m`, the code will load the data and display it on a 2-dimensional plot (Figure 1) by calling the function `displayData`.

This is the same dataset that you used in the previous exercise. There are 5000 training examples in `ex3data1.mat`, where each training example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by



Figure 1: Examples from the dataset

a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is “unrolled” into a 400-dimensional vector. Each of these training examples becomes a single row in our data matrix X . This gives us a 5000 by 400 matrix X where every row is a training example for a handwritten digit image.

$$X = \begin{bmatrix} \text{---} (x^{(1)})^T \text{---} \\ \text{---} (x^{(2)})^T \text{---} \\ \vdots \\ \text{---} (x^{(m)})^T \text{---} \end{bmatrix}$$

The second part of the training set is a 5000-dimensional vector y that contains labels for the training set. To make things more compatible with Octave/Matlab indexing, where there is no zero index, we have mapped the digit zero to the value ten. Therefore, a “0” digit is labeled as “10”, while the digits “1” to “9” are labeled as “1” to “9” in their natural order.

1.2 Model representation

Our neural network is shown in Figure 2. It has 3 layers – an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20×20 , this gives us 400 input layer units (not counting the extra bias unit which always outputs +1). The training data will be loaded into the variables X and y by the `ex4.m` script.

```
% Load saved matrices from file
load('ex4weights.mat');

% The matrices Theta1 and Theta2 will now be in your workspace
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```



1.3 Feedforward and cost function

Recall that the cost function for the neural network (without regulariza-

tion) is

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K \left[-y_k^{(i)} \log((h_{\theta}(x^{(i)}))_k) - (1 - y_k^{(i)}) \log(1 - (h_{\theta}(x^{(i)}))_k) \right],$$

where $h_{\theta}(x^{(i)})$ is computed as shown in the Figure 2 and $K = 10$ is the total number of possible labels. Note that $h_{\theta}(x^{(i)})_k = a_k^{(3)}$ is the activation (output value) of the k -th output unit. Also, recall that whereas the original labels (in the variable y) were 1, 2, ..., 10, for the purpose of training a neural network, we need to recode the labels as vectors containing only values 0 or 1, so that

$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \dots \quad \text{or} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}.$$

For example, if $x^{(i)}$ is an image of the digit 5, then the corresponding $y^{(i)}$ (that you should use with the cost function) should be a 10-dimensional vector with $y_5 = 1$, and the other elements equal to 0.

You should implement the feedforward computation that computes $h_{\theta}(x^{(i)})$ for every example i and sum the cost over all examples. **Your code should also work for a dataset of any size, with any number of labels** (you can assume that there are always at least $K \geq 3$ labels).

Implementation Note: The matrix X contains the examples in rows (i.e., $X(i,:)$ is the i -th training example $x^{(i)}$, expressed as a $n \times 1$ vector.) When you complete the code in `nnCostFunction.m`, you will need to add the column of 1's to the X matrix. The parameters for each unit in the neural network is represented in **Theta1** and **Theta2** as one row. Specifically, the first row of **Theta1** corresponds to the first hidden unit in the second layer. You can use a `for`-loop over the examples to compute the cost.

Once you are done, `ex4.m` will call your `nnCostFunction` using the loaded set of parameters for **Theta1** and **Theta2**. You should see that the cost is about 0.287629.

You should now submit the neural network cost function (feedforward).

1.4 Regularized cost function

The cost function for neural networks with regularization is given by

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K \left[-y_k^{(i)} \log((h_{\theta}(x^{(i)}))_k) - (1 - y_k^{(i)}) \log(1 - (h_{\theta}(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \left[\sum_{j=1}^{25} \sum_{k=1}^{400} (\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10} \sum_{k=1}^{25} (\Theta_{j,k}^{(2)})^2 \right].$$

You can assume that the neural network will only have 3 layers – an input layer, a hidden layer and an output layer. However, your code should work for any number of input units, hidden units and outputs units. While we have explicitly listed the indices above for $\Theta^{(1)}$ and $\Theta^{(2)}$ for clarity, do note that **your code should in general work with $\Theta^{(1)}$ and $\Theta^{(2)}$ of any size.**

Note that you should not be regularizing the terms that correspond to the bias. For the matrices `Theta1` and `Theta2`, this corresponds to the first column of each matrix. You should now add regularization to your cost function. Notice that you can first compute the unregularized cost function J using your existing `nnCostFunction.m` and then later add the cost for the regularization terms.

Once you are done, `ex4.m` will call your `nnCostFunction` using the loaded set of parameters for `Theta1` and `Theta2`, and $\lambda = 1$. You should see that the cost is about 0.383770.

You should now submit the regularized neural network cost function (feed-forward).

2 Backpropagation

In this part of the exercise, you will implement the backpropagation algorithm to compute the gradient for the neural network cost function. You will need to complete the `nnCostFunction.m` so that it returns an appropriate value for `grad`. Once you have computed the gradient, you will be able

to train the neural network by minimizing the cost function $J(\Theta)$ using an advanced optimizer such as `fmincg`.

You will first implement the backpropagation algorithm to compute the gradients for the parameters for the (unregularized) neural network. After you have verified that your gradient computation for the unregularized case is correct, you will implement the gradient for the regularized neural network.

2.1 Sigmoid gradient

To help you get started with this part of the exercise, you will first implement the sigmoid gradient function. The gradient for the sigmoid function can be computed as

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1 - g(z))$$

where

$$\text{sigmoid}(z) = g(z) = \frac{1}{1 + e^{-z}}.$$

When you are done, try testing a few values by calling `sigmoidGradient(z)` at the Octave command line. For large values (both positive and negative) of \mathbf{z} , the gradient should be close to 0. When $\mathbf{z} = 0$, the gradient should be exactly 0.25. Your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid gradient function on every element.

You should now submit the sigmoid gradient function.

2.2 Random initialization

When training neural networks, it is important to randomly initialize the parameters for **symmetry breaking**. One effective strategy for random initialization is to randomly select values for $\Theta^{(l)}$ uniformly in the range $[-\epsilon_{init}, \epsilon_{init}]$. You should use $\epsilon_{init} = 0.12$.¹ This range of values ensures that the parameters are kept small and makes the learning more efficient.

Your job is to complete `randInitializeWeights.m` to initialize the weights for Θ ; modify the file and fill in the following code:

¹One effective strategy for choosing ϵ_{init} is to base it on the number of units in the network. A good choice of ϵ_{init} is $\epsilon_{init} = \frac{\sqrt{6}}{\sqrt{L_{in} + L_{out}}}$, where $L_{in} = s_l$ and $L_{out} = s_{l+1}$ are the number of units in the layers adjacent to $\Theta^{(l)}$.

```
% Randomly initialize the weights to small values
epsilon_init = 0.12;
W = rand(Lout, 1 + Llin) * 2 * epsilon_init - epsilon_init;
```

You do not need to submit any code for this part of the exercise.

2.3 Backpropagation

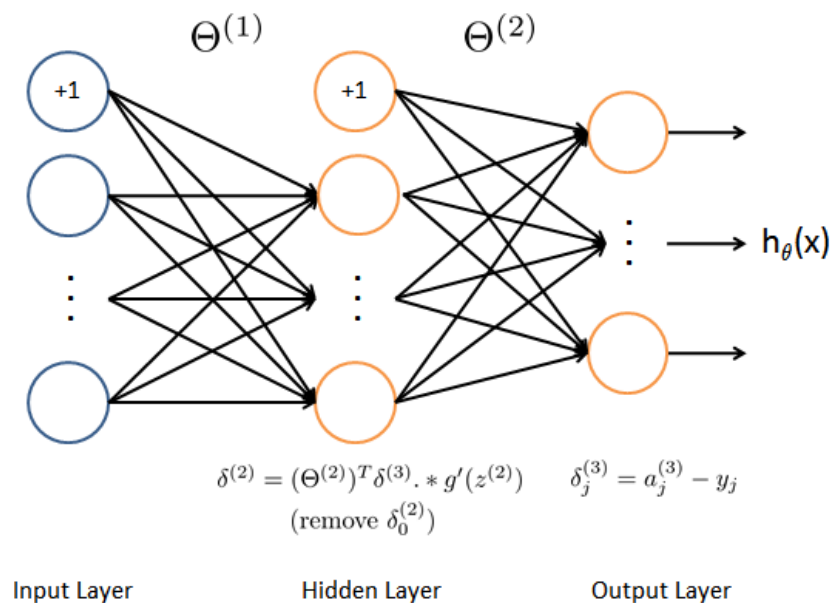


Figure 3: Backpropagation Updates.

Now, you will implement the backpropagation algorithm. Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example $(x^{(t)}, y^{(t)})$, we will first run a “forward pass” to compute all the activations throughout the network, including the output value of the hypothesis $h_{\Theta}(x)$. Then, for each node j in layer l , we would like to compute an “error term” $\delta_j^{(l)}$ that measures how much that node was “responsible” for any errors in our output.

For an output node, we can directly measure the difference between the network’s activation and the true target value, and use that to define $\delta_j^{(3)}$ (since layer 3 is the output layer). For the hidden units, you will compute $\delta_j^{(l)}$ based on a weighted average of the error terms of the nodes in layer $(l + 1)$.

In detail, here is the backpropagation algorithm (also depicted in Figure 3). You should implement steps 1 to 4 in a loop that processes one example at a time. Concretely, you should implement a for-loop `for t = 1:m` and place steps 1-4 below inside the for-loop, with the t^{th} iteration performing the calculation on the t^{th} training example $(x^{(t)}, y^{(t)})$. Step 5 will divide the accumulated gradients by m to obtain the gradients for the neural network cost function.

1. Set the input layer's values $(a^{(1)})$ to the t -th training example $x^{(t)}$. Perform a feedforward pass (Figure 2), computing the activations $(z^{(2)}, a^{(2)}, z^{(3)}, a^{(3)})$ for layers 2 and 3. Note that you need to add a $+1$ term to ensure that the vectors of activations for layers $a^{(1)}$ and $a^{(2)}$ also include the bias unit. In Octave, if `a_1` is a column vector, adding one corresponds to `a_1 = [1 ; a_1]`.

2. For each output unit k in layer 3 (the output layer), set

$$\delta_k^{(3)} = (a_k^{(3)} - y_k),$$

where $y_k \in \{0, 1\}$ indicates whether the current training example belongs to class k ($y_k = 1$), or if it belongs to a different class ($y_k = 0$). You may find logical arrays helpful for this task (explained in the previous programming exercise).

3. For the hidden layer $l = 2$, set

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

维数不对，前面的有26行，g'(z)却只有25行。

4. Accumulate the gradient from this example using the following formula. Note that you should skip or remove $\delta_0^{(2)}$. In Octave, removing $\delta_0^{(2)}$ corresponds to `delta_2 = delta_2(2:end)`.

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$$

5. Obtain the (unregularized) gradient for the neural network cost function by dividing the accumulated gradients by $\frac{1}{m}$:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

Octave Tip: You should implement the backpropagation algorithm only after you have successfully completed the feedforward and cost functions. While implementing the backpropagation algorithm, it is often useful to use the `size` function to print out the sizes of the variables you are working with if you run into dimension mismatch errors (“nonconformant arguments” errors in Octave).

After you have implemented the backpropagation algorithm, the script `ex4.m` will proceed to run gradient checking on your implementation. The gradient check will allow you to increase your confidence that your code is computing the gradients correctly.

2.4 Gradient checking

In your neural network, you are minimizing the cost function $J(\Theta)$. To perform gradient checking on your parameters, you can imagine “unrolling” the parameters $\Theta^{(1)}, \Theta^{(2)}$ into a long vector θ . By doing so, you can think of the cost function being $J(\theta)$ instead and use the following gradient checking procedure.

Suppose you have a function $f_i(\theta)$ that purportedly computes $\frac{\partial}{\partial \theta_i} J(\theta)$; you’d like to check if f_i is outputting correct derivative values.

$$\text{Let } \theta^{(i+)} = \theta + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix} \quad \text{and} \quad \theta^{(i-)} = \theta - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}$$

So, $\theta^{(i+)}$ is the same as θ , except its i -th element has been incremented by ϵ . Similarly, $\theta^{(i-)}$ is the corresponding vector with the i -th element decreased by ϵ . You can now numerically verify $f_i(\theta)$ ’s correctness by checking, for each i , that:

$$f_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon}.$$

The degree to which these two values should approximate each other will depend on the details of J . But assuming $\epsilon = 10^{-4}$, you’ll usually find that the left- and right-hand sides of the above will agree to at least 4 significant digits (and often many more).

We have implemented the function to compute the numerical gradient for you in `computeNumericalGradient.m`. While you are not required to modify the file, we highly encourage you to take a look at the code to understand how it works.

In the next step of `ex4.m`, it will run the provided function `checkNNGradients.m` which will create a small neural network and dataset that will be used for checking your gradients. If your backpropagation implementation is correct, you should see a relative difference that is less than $1e-9$.

Practical Tip: When performing gradient checking, it is much more efficient to use a small neural network with a relatively small number of input units and hidden units, thus having a relatively small number of parameters. Each dimension of θ requires two evaluations of the cost function and this can be expensive. In the function `checkNNGradients`, our code creates a small random model and dataset which is used with `computeNumericalGradient` for gradient checking. Furthermore, after you are confident that your gradient computations are correct, you should turn off gradient checking before running your learning algorithm.

Practical Tip: Gradient checking works for any function where you are computing the cost and the gradient. Concretely, you can use the same `computeNumericalGradient.m` function to check if your gradient implementations for the other exercises are correct too (e.g., logistic regression's cost function).

Once your cost function passes the gradient check for the (unregularized) neural network cost function, you should submit the neural network gradient function (backpropagation).

2.5 Regularized Neural Networks

After you have successfully implemented the backpropagation algorithm, you will add regularization to the gradient. To account for regularization, it turns out that you can add this as an additional term *after* computing the gradients using backpropagation.

Specifically, after you have computed $\Delta_{ij}^{(l)}$ using backpropagation, you should add regularization using

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} \quad \text{for } j = 0$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \frac{\lambda}{m} \Theta_{ij}^{(l)} \quad \text{for } j \geq 1$$

Note that you should *not* be regularizing the first column of $\Theta^{(l)}$ which is used for the bias term. Furthermore, in the parameters $\Theta_{ij}^{(l)}$, i is indexed starting from 1, and j is indexed starting from 0. Thus,

$$\Theta^{(l)} = \begin{bmatrix} \Theta_{1,0}^{(l)} & \Theta_{1,1}^{(l)} & \cdots \\ \Theta_{2,0}^{(l)} & \Theta_{2,1}^{(l)} & \\ \vdots & & \ddots \end{bmatrix}.$$

Somewhat confusingly, indexing in Octave starts from 1 (for both i and j), thus `Theta1(2, 1)` actually corresponds to $\Theta_{2,0}^{(1)}$ (i.e., the entry in the second row, first column of the matrix $\Theta^{(1)}$ shown above)

Now modify your code that computes `grad` in `nnCostFunction` to account for regularization. After you are done, the `ex4.m` script will proceed to run gradient checking on your implementation. If your code is correct, you should expect to see a relative difference that is less than 1e-9.

You should now submit your regularized neural network gradient.

2.6 Learning parameters using `fmincg`

After you have successfully implemented the neural network cost function and gradient computation, the next step of the `ex4.m` script will use `fmincg` to learn a good set parameters.

After the training completes, the `ex4.m` script will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct. If your implementation is correct, you should see a reported training accuracy of about 95.3% (this may vary by about 1% due to the random initialization). It is possible to get higher training accuracies by training the neural network for more iterations. We encourage you to try training the neural network for more iterations (e.g., set `MaxIter` to 400) and also vary the regularization parameter λ . With the right learning settings, it is possible to get the neural network to perfectly fit the training set.

3 Visualizing the hidden layer

One way to understand what your neural network is learning is to visualize what the representations captured by the hidden units. Informally, given a

particular hidden unit, one way to visualize what it computes is to find an input x that will cause it to activate (that is, to have an activation value $(a_i^{(l)})$ close to 1). For the neural network you trained, notice that the i^{th} row of $\Theta^{(l)}$ is a 401-dimensional vector that represents the parameter for the i^{th} hidden unit. If we discard the bias term, we get a 400 dimensional vector that represents the weights from each input pixel to the hidden unit.

Thus, one way to visualize the “representation” captured by the hidden unit is to reshape this 400 dimensional vector into a 20×20 image and display it.² The next step of `ex4.m` does this by using the `displayData` function and it will show you an image (similar to Figure 4) with 25 units, each corresponding to one hidden unit in the network.

In your trained network, you should find that the hidden units corresponds roughly to detectors that look for strokes and other patterns in the input.

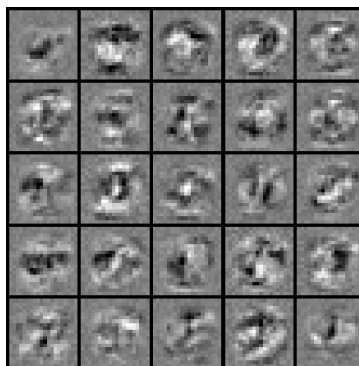


Figure 4: Visualization of Hidden Units.

3.1 Optional (ungraded) exercise

In this part of the exercise, you will get to try out different learning settings for the neural network to see how the performance of the neural network varies with the regularization parameter λ and number of training steps (the `MaxIter` option when using `fmincg`).

Neural networks are very powerful models that can form highly complex decision boundaries. Without regularization, it is possible for a neural network to “overfit” a training set so that it obtains close to 100% accuracy on

²It turns out that this is equivalent to finding the input that gives the highest activation for the hidden unit, given a “norm” constraint on the input (i.e., $\|x\|_2 \leq 1$).

the training set but does not as well on new examples that it has not seen before. You can set the regularization λ to a smaller value and the **MaxIter** parameter to a higher number of iterations to see this for yourself.

You will also be able to see for yourself the changes in the visualizations of the hidden units when you change the learning parameters λ and **MaxIter**.

You do not need to submit any solutions for this optional (ungraded) exercise.

Submission and Grading

After completing various parts of the assignment, be sure to use the `submit` function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Feedforward and Cost Function	<code>nnCostFunction.m</code>	30 points
Regularized Cost Function	<code>nnCostFunction.m</code>	15 points
Sigmoid Gradient	<code>sigmoidGradient.m</code>	5 points
Neural Net Gradient Function (Backpropagation)	<code>nnCostFunction.m</code>	40 points
Regularized Gradient	<code>nnCostFunction.m</code>	10 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration. To prevent rapid-fire guessing, the system enforces a minimum of 5 minutes between submissions.

Programming Exercise 5: Regularized Linear Regression and Bias v.s. Variance

Machine Learning

Introduction

In this exercise, you will implement regularized linear regression and use it to study models with different bias-variance properties. Before starting on the programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics.

To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise. If needed, use the `cd` command in Octave to change to this directory before starting this exercise.

Files included in this exercise

- `ex5.m` - Octave script that will help step you through the exercise
- `ex5data1.mat` - Dataset
- `submit.m` - Submission script that sends your solutions to our servers
- `submitWeb.m` - Alternative submission script
- `featureNormalize.m` - Feature normalization function
- `fmincg.m` - Function minimization routine (similar to `fminunc`)
- `plotFit.m` - Plot a polynomial fit
- `trainLinearReg.m` - Trains linear regression using your cost function
- [*] `linearRegCostFunction.m` - Regularized linear regression cost function
- [*] `learningCurve.m` - Generates a learning curve
- [*] `polyFeatures.m` - Maps data into polynomial feature space
- [*] `validationCurve.m` - Generates a cross validation curve

* indicates files you will need to complete

Throughout the exercise, you will be using the script `ex5.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

We also strongly encourage using the online **Q&A Forum** to discuss exercises with other students. However, do not look at any source code written by others or share your source code with others.

If you run into network errors using the `submit` script, you can also use an online form for submitting your solutions. To use this *alternative* submission interface, run the `submitWeb` script to generate a submission file (e.g., `submit_ex5_part2.txt`). You can then submit this file through the web submission form in the programming exercises page (go to the programming exercises page, then select the exercise you are submitting for). If you are having no problems submitting through the standard submission system using the `submit` script, you do *not* need to use this alternative submission interface.

1 Regularized Linear Regression

In the first half of the exercise, you will implement regularized linear regression to predict the amount of water flowing out of a dam using the change of water level in a reservoir. In the next half, you will go through some diagnostics of debugging learning algorithms and examine the effects of bias v.s. variance.

The provided script, `ex5.m`, will help you step through this exercise.

1.1 Visualizing the dataset

We will begin by visualizing the dataset containing historical records on the change in the water level, x , and the amount of water flowing out of the dam, y .

This dataset is divided into three parts:

- A **training** set that your model will learn on: X , y

- A **cross validation** set for determining the regularization parameter: `Xval`, `yval`
- A **test** set for evaluating performance. These are “unseen” examples which your model did not see during training: `Xtest`, `ytest`

The next step of `ex5.m` will plot the training data (Figure 1). In the following parts, you will implement linear regression and use that to fit a straight line to the data and plot learning curves. Following that, you will implement polynomial regression to find a better fit to the data.

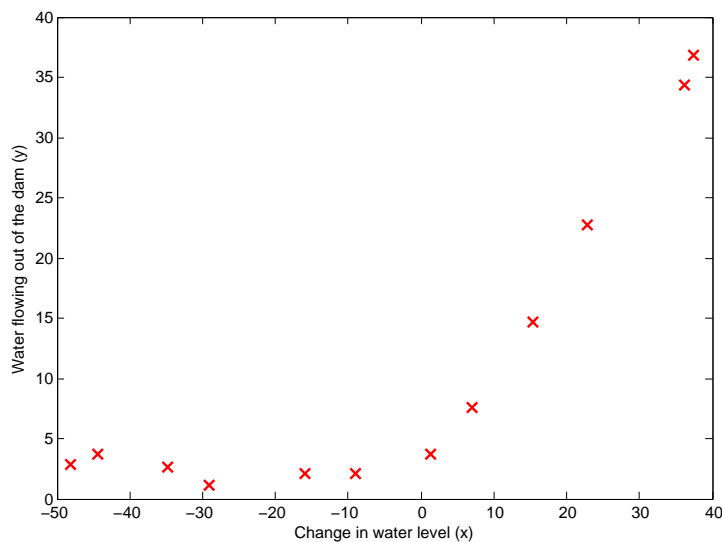


Figure 1: Data

1.2 Regularized linear regression cost function

Recall that regularized linear regression has the following cost function:

$$J(\theta) = \frac{1}{2m} \left(\sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 \right) + \frac{\lambda}{2m} \left(\sum_{j=1}^n \theta_j^2 \right),$$

where λ is a regularization parameter which controls the degree of regularization (thus, help preventing overfitting). The regularization term puts a penalty on the overall cost J . As the magnitudes of the model parameters θ_j increase, the penalty increases as well. Note that you should not regularize the θ_0 term. (In Octave, the θ_0 term is represented as `theta(1)` since indexing in Octave starts from 1).

You should now complete the code in the file `linearRegCostFunction.m`. Your task is to write a function to calculate the regularized linear regression cost function. If possible, try to vectorize your code and avoid writing loops. When you are finished, the next part of `ex5.m` will run your cost function using `theta` initialized at `[1; 1]`. You should expect to see an output of 303.993.

You should now submit your regularized linear regression cost function.

1.3 Regularized linear regression gradient

Correspondingly, the partial derivative of regularized linear regression's cost for θ_j is defined as

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad \text{for } j = 0$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \quad \text{for } j \geq 1$$

In `linearRegCostFunction.m`, add code to calculate the gradient, returning it in the variable `grad`. When you are finished, the next part of `ex5.m` will run your gradient function using `theta` initialized at `[1; 1]`. You should expect to see a gradient of `[-15.30; 598.250]`.

You should now submit your regularized linear regression gradient function.

1.4 Fitting linear regression

Once your cost function and gradient are working correctly, the next part of `ex5.m` will run the code in `trainLinearReg.m` to compute the optimal values of θ . This training function uses `fmincg` to optimize the cost function.

In this part, we set regularization parameter λ to zero. Because our current implementation of linear regression is trying to fit a 2-dimensional θ , regularization will not be incredibly helpful for a θ of such low dimension. In the later parts of the exercise, you will be using polynomial regression with regularization.

Finally, the `ex5.m` script should also plot the best fit line, resulting in an image similar to Figure 2. The best fit line tells us that the model is

not a good fit to the data because the data has a non-linear pattern. While visualizing the best fit as shown is one possible way to debug your learning algorithm, it is not always easy to visualize the data and model. In the next section, you will implement a function to generate learning curves that can help you debug your learning algorithm even if it is not easy to visualize the data.

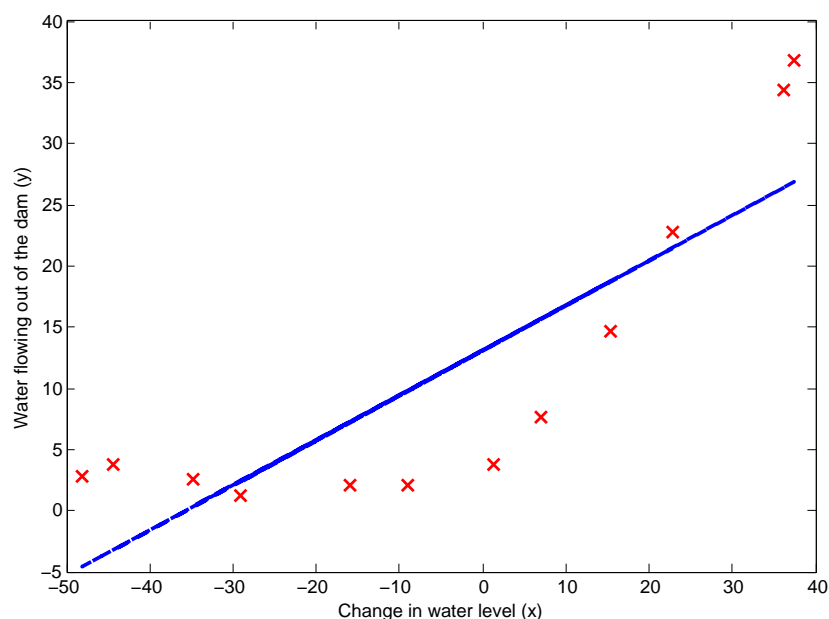


Figure 2: Linear Fit

2 Bias-variance

An important concept in machine learning is the bias-variance tradeoff. Models with high bias are not complex enough for the data and tend to underfit, while models with high variance overfit to the training data.

In this part of the exercise, you will plot training and test errors on a learning curve to diagnose bias-variance problems.

2.1 Learning curves

You will now implement code to generate the learning curves that will be useful in debugging learning algorithms. Recall that a learning curve plots

training and cross validation error as a function of training set size. Your job is to fill in `learningCurve.m` so that it returns a vector of errors for the training set and cross validation set.

To plot the learning curve, we need a training and cross validation set error for different *training* set sizes. To obtain different training set sizes, you should use different subsets of the original training set `X`. Specifically, for a training set size of `i`, you should use the first `i` examples (i.e., `X(1:i,:)` and `y(1:i)`).

You can use the `trainLinearReg` function to find the θ parameters. Note that the `lambda` is passed as a parameter to the `learningCurve` function. After learning the θ parameters, you should compute the **error** on the training and cross validation sets. Recall that the training error for a dataset is defined as

$$J_{\text{train}}(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 \right].$$

In particular, note that the training error does not include the regularization term. One way to compute the training error is to use your existing cost function and set λ to 0 *only* when using it to compute the training error and cross validation error. When you are computing the training set error, make sure you compute it on the training subset (i.e., `X(1:n,:)` and `y(1:n)`) (instead of the entire training set). However, for the cross validation error, you should compute it over the *entire* cross validation set. You should store the computed errors in the vectors `error_train` and `error_val`.

When you are finished, `ex5.m` will print the learning curves and produce a plot similar to Figure 3.

You should now submit your learning curve function.

In Figure 3, you can observe that *both* the train error and cross validation error are high when the number of training examples is increased. This reflects a **high bias** problem in the model – the linear regression model is too simple and is unable to fit our dataset well. In the next section, you will implement polynomial regression to fit a better model for this dataset.

3 Polynomial regression

The problem with our linear model was that it was too simple for the data and resulted in underfitting (high bias). In this part of the exercise, you will

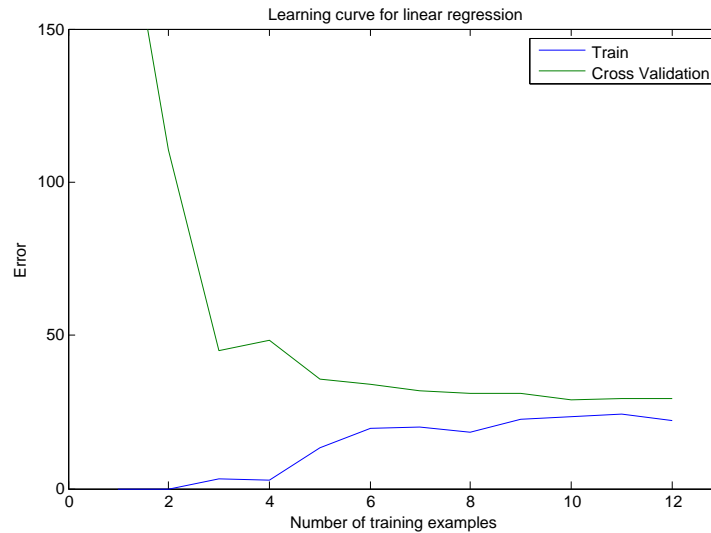


Figure 3: Linear regression learning curve

address this problem by adding more features.

For use polynomial regression, our hypothesis has the form:

$$\begin{aligned}
 h_{\theta}(x) &= \theta_0 + \theta_1 * (\text{waterLevel}) + \theta_2 * (\text{waterLevel})^2 + \dots + \theta_p * (\text{waterLevel})^p \\
 &= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_p x_p.
 \end{aligned}$$

Notice that by defining $x_1 = (\text{waterLevel})$, $x_2 = (\text{waterLevel})^2, \dots, x_p = (\text{waterLevel})^p$, we obtain a linear regression model where the features are the various powers of the original value (waterLevel).

Now, you will add more features using the higher powers of the existing feature x in the dataset. Your task in this part is to complete the code in `polyFeatures.m` so that the function maps the original training set \mathbf{X} of size $m \times 1$ into its higher powers. Specifically, when a training set \mathbf{X} of size $m \times 1$ is passed into the function, the function should return a $m \times p$ matrix \mathbf{X}_{poly} , where column 1 holds the original values of \mathbf{X} , column 2 holds the values of $\mathbf{X}.^2$, column 3 holds the values of $\mathbf{X}.^3$, and so on. Note that you don't have to account for the zero-eth power in this function.

Now you have a function that will map features to a higher dimension, and Part 6 of `ex5.m` will apply it to the training set, the test set, and the cross validation set (which you haven't used yet).

You should now submit your polynomial feature mapping function.

3.1 Learning Polynomial Regression

After you have completed `polyFeatures.m`, the `ex5.m` script will proceed to train polynomial regression using your linear regression cost function.

Keep in mind that even though we have polynomial terms in our feature vector, we are still solving a linear regression optimization problem. The polynomial terms have simply turned into features that we can use for linear regression. We are using the same cost function and gradient that you wrote for the earlier part of this exercise.

For this part of the exercise, you will be using a polynomial of degree 8. It turns out that if we run the training directly on the projected data, will not work well as the features would be badly scaled (e.g., an example with $x = 40$ will now have a feature $x_8 = 40^8 = 6.5 \times 10^{12}$). Therefore, you will need to use feature normalization.

Before learning the parameters θ for the polynomial regression, `ex5.m` will first call `featureNormalize` and normalize the features of the training set, storing the `mu`, `sigma` parameters separately. We have already implemented this function for you and it is the same function from the first exercise.

After learning the parameters θ , you should see two plots (Figure 4,5) generated for polynomial regression with $\lambda = 0$.

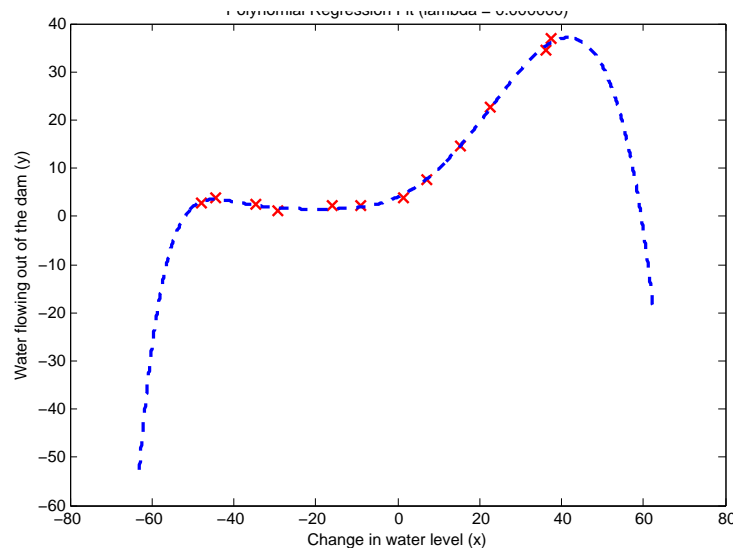


Figure 4: Polynomial fit, $\lambda = 0$

From Figure 4, you should see that the polynomial fit is able to follow the datapoints very well - thus, obtaining a low training error. However, the

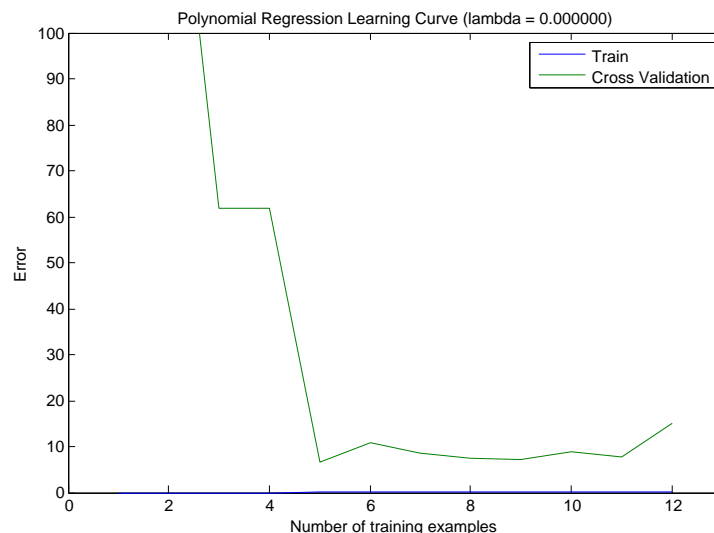


Figure 5: Polynomial learning curve, $\lambda = 0$

polynomial fit is very complex and even drops off at the extremes. This is an indicator that the polynomial regression model is overfitting the training data and will not generalize well.

To better understand the problems with the unregularized ($\lambda = 0$) model, you can see that the learning curve (Figure 5) shows the same effect where the low training error is low, but the cross validation error is high. There is a gap between the training and cross validation errors, indicating a high variance problem.

One way to combat the overfitting (high-variance) problem is to add regularization to the model. In the next section, you will get to try different λ parameters to see how regularization can lead to a better model.

3.2 Optional (ungraded) exercise: Adjusting the regularization parameter

In this section, you will get to observe how the regularization parameter affects the bias-variance of regularized polynomial regression. You should now modify the `lambda` parameter in the `ex5.m` and try $\lambda = 1, 100$. For each of these values, the script should generate a polynomial fit to the data and also a learning curve.

For $\lambda = 1$, you should see a polynomial fit that follows the data trend well (Figure 6) and a learning curve (Figure 7) showing that both the cross

validation and training error converge to a relatively low value. This shows the $\lambda = 1$ regularized polynomial regression model does not have the high-bias or high-variance problems. In effect, it achieves a good trade-off between bias and variance.

For $\lambda = 100$, you should see a polynomial fit (Figure 8) that does not follow the data well. In this case, there is too much regularization and the model is unable to fit the training data.

You do not need to submit any solutions for this optional (ungraded) exercise.

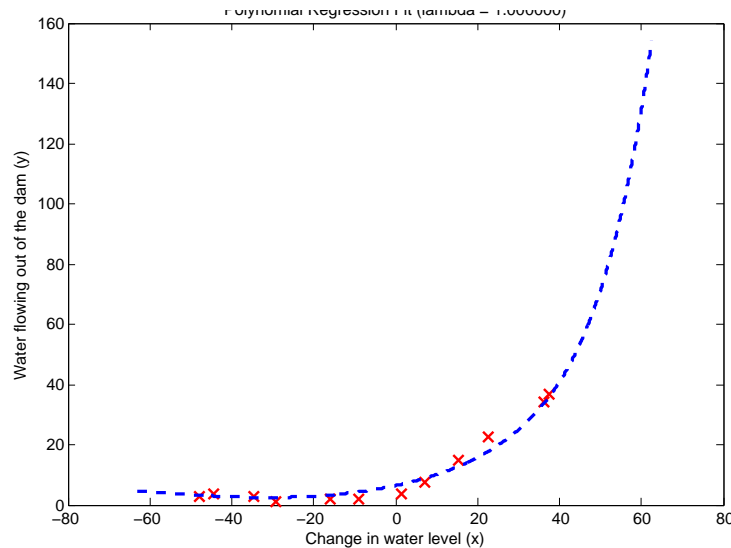


Figure 6: Polynomial fit, $\lambda = 1$

3.3 Selecting λ using a cross validation set

From the previous parts of the exercise, you observed that the value of λ can significantly affect the results of regularized polynomial regression on the training and cross validation set. In particular, a model without regularization ($\lambda = 0$) fits the training set well, but does not generalize. Conversely, a model with too much regularization ($\lambda = 100$) does not fit the training set and testing set well. A good choice of λ (e.g., $\lambda = 1$) can provide a good fit to the data.

In this section, you will implement an automated method to select the λ parameter. Concretely, you will use a cross validation set to evaluate

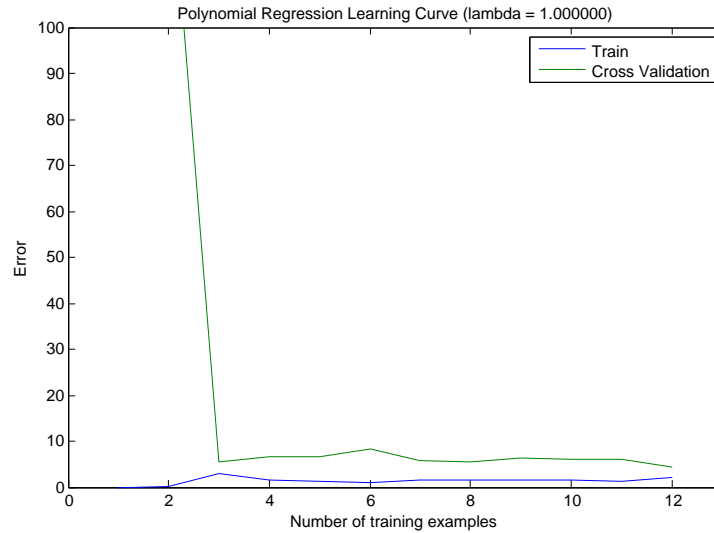


Figure 7: Polynomial learning curve, $\lambda = 1$

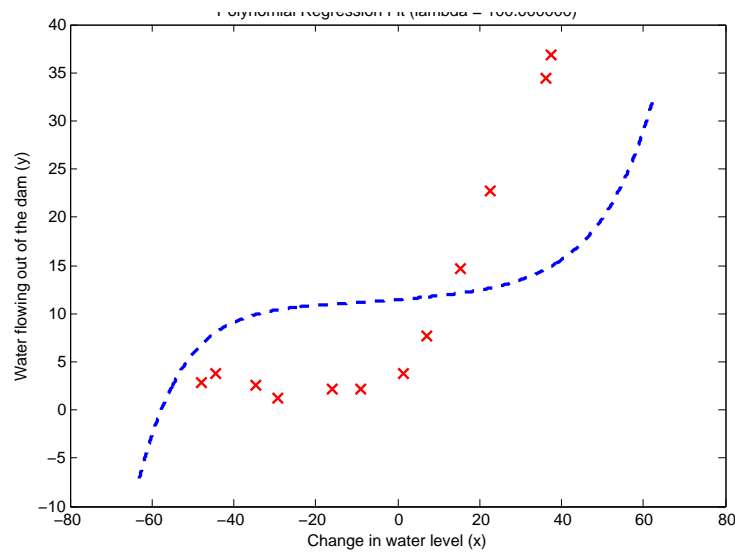


Figure 8: Polynomial fit, $\lambda = 100$

how good each λ value is. After selecting the best λ value using the cross validation set, we can then evaluate the model on the test set to estimate how well the model will perform on actual unseen data.

Your task is to complete the code in `validationCurve.m`. Specifically, you should use the `trainLinearReg` function to train the model using

different values of λ and compute the training error and cross validation error. You should try λ in the following range: $\{0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10\}$.

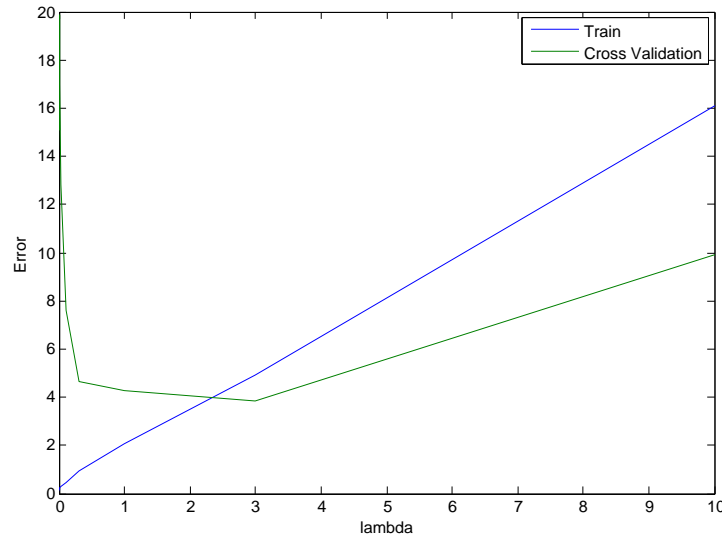


Figure 9: Selecting λ using a cross validation set

After you have completed the code, the next part of `ex5.m` will run your function can plot a cross validation curve of error v.s. λ that allows you select which λ parameter to use. You should see a plot similar to Figure 9. In this figure, we can see that the best value of λ is around 3. Due to randomness in the training and validation splits of the dataset, the cross validation error can sometimes be lower than the training error.

You should now submit your validation curve function.

3.4 Optional (ungraded) exercise: Computing test set error

In the previous part of the exercise, you implemented code to compute the cross validation error for various values of the regularization parameter λ . However, to get a better indication of the model's performance in the real world, it is important to evaluate the “final” model on a test set that was not used in any part of training (that is, it was neither used to select the λ parameters, nor to learn the model parameters θ).

For this optional (ungraded) exercise, you should compute the test error using the best value of λ you found. In our cross validation, we obtained a

test error of 3.8599 for $\lambda = 3$.

You do not need to submit any solutions for this optional (ungraded) exercise.

3.5 Optional (ungraded) exercise: Plotting learning curves with randomly selected examples

In practice, especially for small training sets, when you plot learning curves to debug your algorithms, it is often helpful to average across multiple sets of randomly selected examples to determine the training error and cross validation error.

Concretely, to determine the training error and cross validation error for i examples, you should first randomly select i examples from the training set and i examples from the cross validation set. You will then learn the parameters θ using the randomly chosen training set and evaluate the parameters θ on the randomly chosen training set and cross validation set. The above steps should then be repeated multiple times (say 50) and the averaged error should be used to determine the training error and cross validation error for i examples.

For this optional (ungraded) exercise, you should implement the above strategy for computing the learning curves. For reference, figure 10 shows the learning curve we obtained for polynomial regression with $\lambda = 0.01$. Your figure may differ slightly due to the random selection of examples.

You do not need to submit any solutions for this optional (ungraded) exercise.

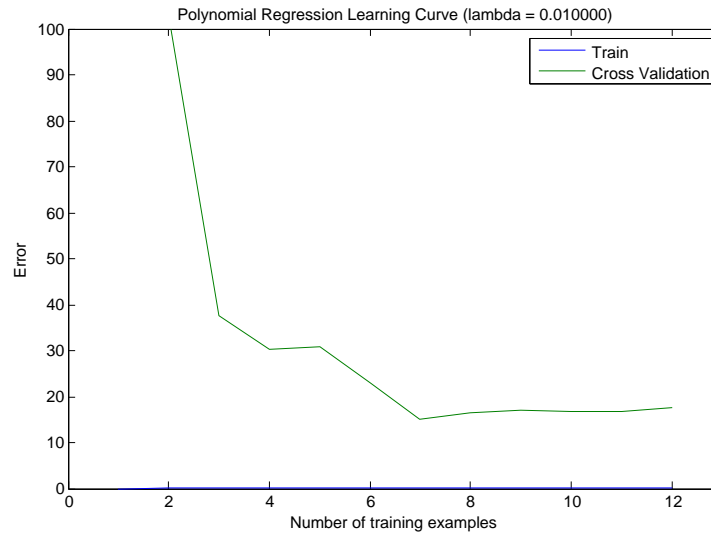


Figure 10: Optional (ungraded) exercise: Learning curve with randomly selected examples

Submission and Grading

After completing various parts of the assignment, be sure to use the `submit` function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Regularized Linear Regression Cost Function	<code>linearRegCostFunction.m</code>	25 points
Regularized Linear Regression Gradient	<code>linearRegCostFunction.m</code>	25 points
Learning Curve	<code>learningCurve.m</code>	20 points
Polynomial Feature Mapping	<code>polyFeatures.m</code>	10 points
Cross Validation Curve	<code>validationCurve.m</code>	20 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration. To prevent rapid-fire guessing, the system enforces a minimum of 5 minutes between submissions.

Programming Exercise 6: Support Vector Machines

Machine Learning

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 watching the video lectures and completing the review questions for the associated topics.

To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise. If needed, use the `cd` command in Octave to change to this directory before starting this exercise.

Files included in this exercise

- `ex6.m` - Octave script for the first half of the exercise
- `ex6data1.mat` - Example Dataset 1
- `ex6data2.mat` - Example Dataset 2
- `ex6data3.mat` - Example Dataset 3
- `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
- [★] `dataset3Params.m` - Parameters to use for Dataset 3

- `ex6_spam.m` - Octave 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
`submit.m` - Submission script that sends your solutions to our servers
`submitWeb.m` - Alternative submission script
[*] `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

We also strongly encourage using the online **Q&A Forum** to discuss exercises with other students. However, do not look at any source code written by others or share your source code with others.

If you run into network errors using the `submit` script, you can also use an online form for submitting your solutions. To use this *alternative* submission interface, run the `submitWeb` script to generate a submission file (e.g., `submit_ex6_part2.txt`). You can then submit this file through the web submission form in the programming exercises page (go to the programming exercises page, then select the exercise you are submitting for). If you are having no problems submitting through the standard submission system using the `submit` script, you do *not* need to use this alternative submission interface.

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.

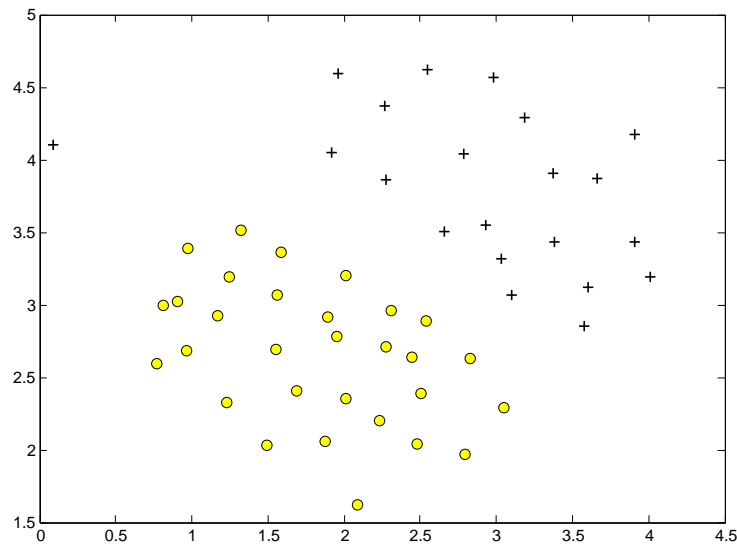


Figure 1: Example Dataset 1

In this part of the exercise, you will try using different values of the 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.

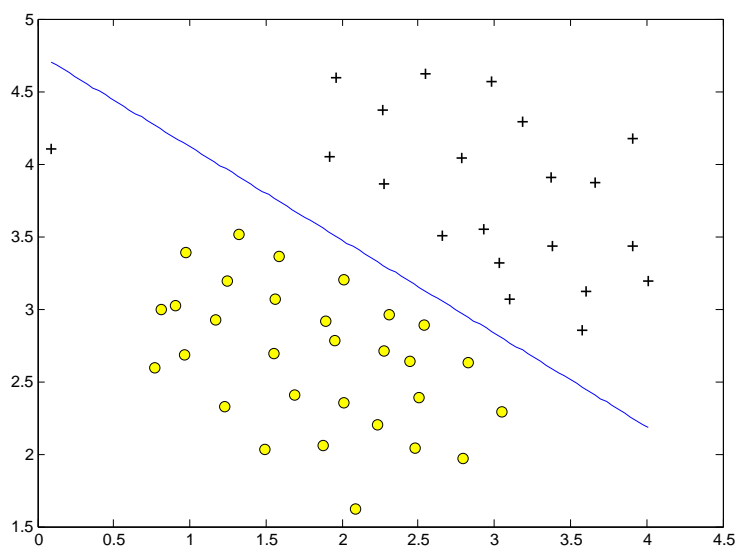


Figure 2: SVM Decision Boundary with $C = 1$ (Example Dataset 1)

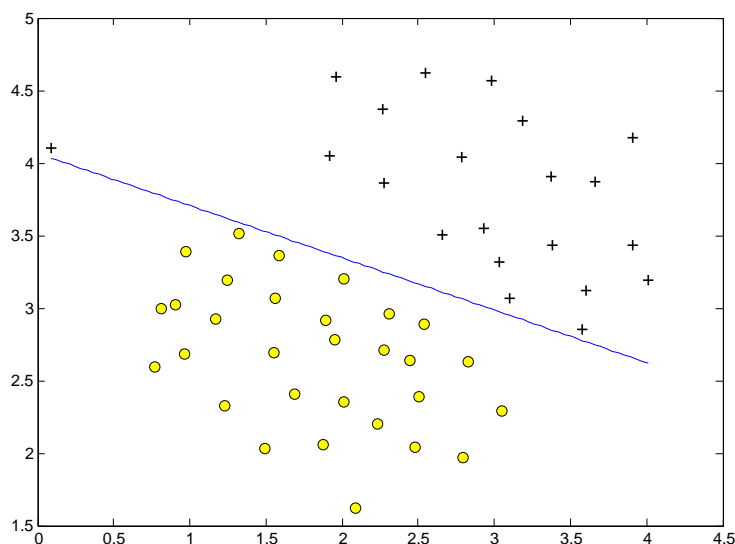


Figure 3: SVM Decision Boundary with $C = 100$ (Example Dataset 1)

The next part in `ex6.m` will run the SVM training (with $C = 1$) using

SVM software that we have included with the starter code, `svmTrain.m`.¹ 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).

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 your code should be working with training examples $x \in \mathbb{R}^n$ (rather than $x \in \mathbb{R}^{n+1}$); for example, in the first example dataset $x \in \mathbb{R}^2$.

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).

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 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 code in `gaussianKernel.m` to compute the Gaussian kernel between two examples, $(x^{(i)}, x^{(j)})$. The Gaussian kernel

¹In order to ensure compatibility with Octave, we have included this implementation of an SVM learning 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](#).

function is defined as:

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

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.

You should now submit your function that computes the Gaussian kernel.

1.2.2 Example Dataset 2

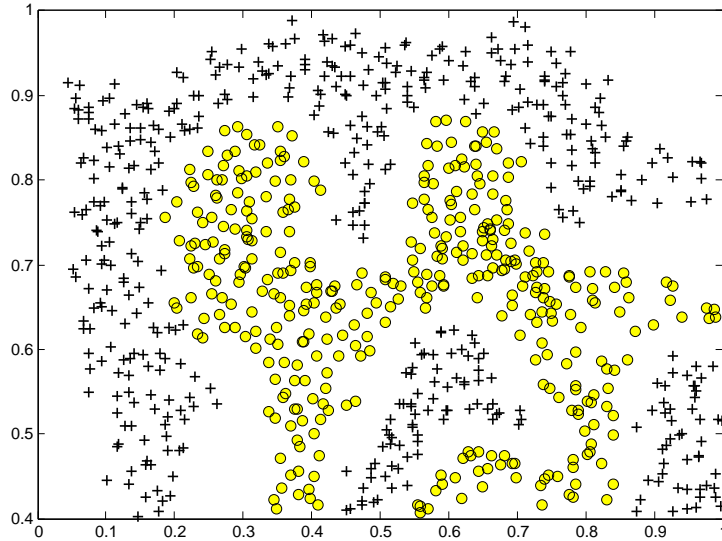


Figure 4: Example Dataset 2

The next part in `ex6.m` 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, `ex6.m` will proceed to train the SVM with the Gaussian kernel on this dataset.

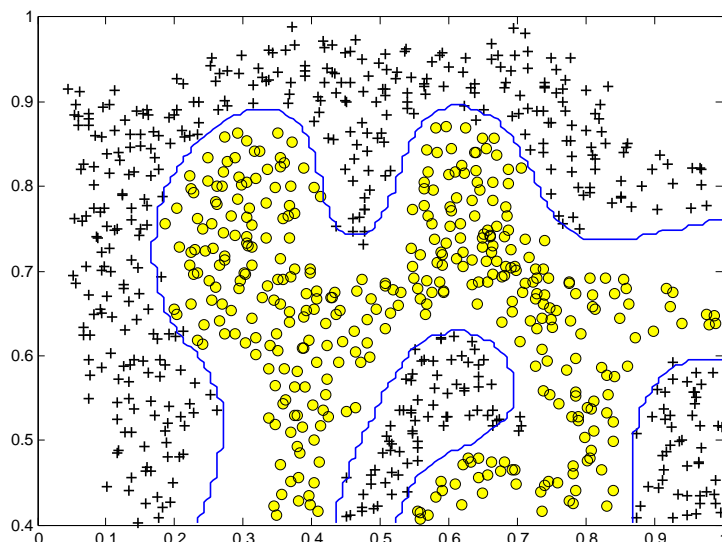


Figure 5: SVM (Gaussian Kernel) Decision Boundary (Example Dataset 2)

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.

1.2.3 Example Dataset 3

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 `ex6.m` will load and display a third dataset (Figure 6). You will be using the SVM with the Gaussian kernel with this dataset.

In the provided dataset, `ex6data3.mat`, you are given the variables `X`, `y`, `Xval`, `yval`. The provided code in `ex6.m` trains the SVM classifier using the training set (`X`, `y`) using parameters loaded from `dataset3Params.m`.

Your task is to use the cross 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^2 , you would end up training and evaluating (on the cross validation set) a total of $8^2 = 64$ different models.

After you have determined the best C and σ parameters to use, you should modify the code in `dataset3Params.m`, filling in the best parameters

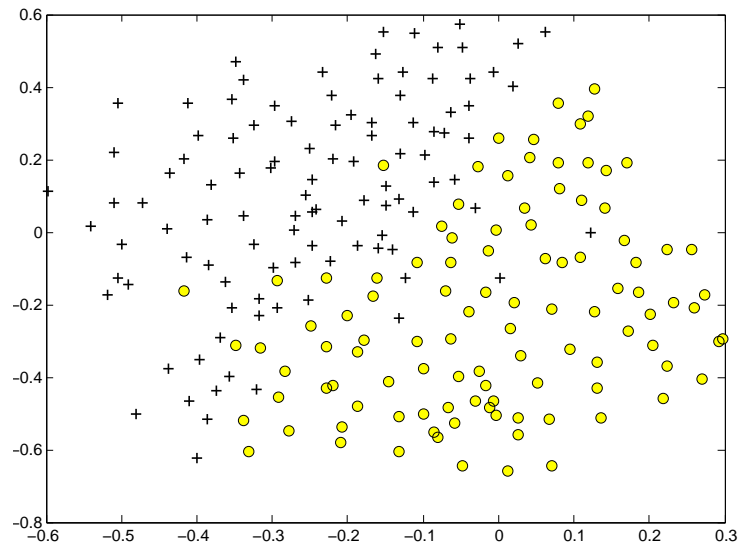


Figure 6: Example Dataset 3

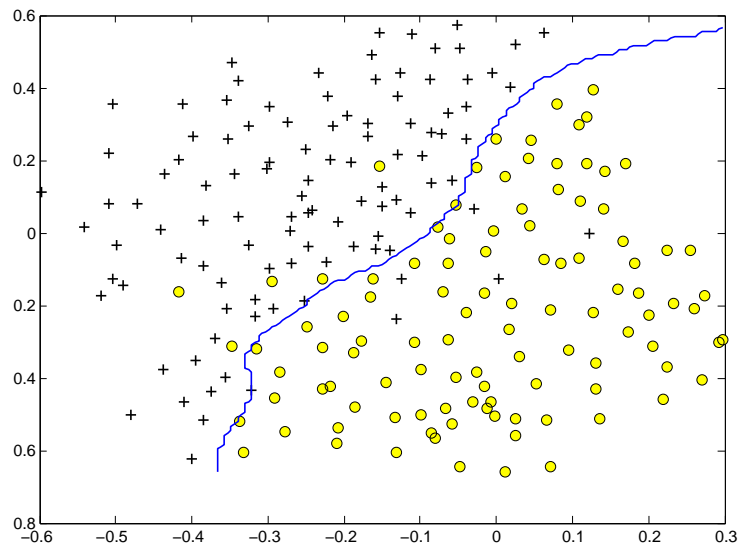


Figure 7: SVM (Gaussian Kernel) Decision Boundary (Example Dataset 3)

you found. For our best parameters, the SVM returned a decision boundary shown in Figure 7.

Implementation Tip: When implementing cross validation to select the best C and σ parameter to use, you need to evaluate the error on the cross validation set. Recall that for classification, the error is defined as the fraction of the cross validation examples that were classified incorrectly. In Octave, you can compute this error using `mean(double(predictions ~= yval))`, where `predictions` is a vector containing all the predictions from the SVM, and `yval` are the true labels from the cross validation set. You can use the `svmPredict` function to generate the predictions for the cross validation set.

You should now submit your best C and σ values.

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 $x \in \mathbb{R}^n$. 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 script `ex6_spam.m`. The dataset included for this exercise is based on 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 8: Sample 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 replace each URL in the email with the unique string “httpaddr” to indicate that a URL was present.

²<http://spamassassin.apache.org/publiccorpus/>

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 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”`.
- **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.

```

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: Preprocessed Sample Email

```

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

```

Figure 10: 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 11: Word Indices for Sample Email

2.1.1 Vocabulary List

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 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 10. 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 9) into a list of word indices that contains the index of the word in the vocabulary list. Figure 11 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 9 & 11.

Octave Tip: In Octave, 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, 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 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.

You should now submit the email preprocessing function.

2.2 Extracting Features from Emails

You will now implement the feature extraction that converts each email into a vector in \mathbb{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:

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

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.

You should now submit the email feature extraction function.

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 $x^{(i)} \in \mathbb{R}^{1899}$.

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 12: 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 12). Thus, if an email contains words such as “guarantee”, “remove”, “dollar”, and “price” (the top predictors shown in Figure 12), it is likely to be classified as spam.

2.5 Optional (ungraded) exercise: Try your own emails

Now that you have trained a spam classifier, you can start trying it out on your own emails. In the starter code, we have included two email examples (`emailSample1.txt` and `emailSample2.txt`) and two spam examples (`spamSample1.txt` and `spamSample2.txt`). The last part of `ex6_spam.m` runs the spam classifier over the first spam example and classifies it using the learned SVM. You should now try the other examples we have provided and see if the classifier gets them right. You can also try your own emails by replacing the examples (plain text files) with your own emails.

You do not need to submit any solutions for this optional (ungraded) exercise.

2.6 Optional (ungraded) exercise: Build your own dataset

In this exercise, we provided a preprocessed training set and test set. These datasets were created using the same functions (`processEmail.m` and `emailFeatures.m`) that you now have completed. For this optional (ungraded) exercise, you will build your own dataset using the original emails from the [SpamAssassin Public Corpus](#).

Your task in this optional (ungraded) exercise is to download the original files from the public corpus and extract them. After extracting them, you should run the `processEmail`³ and `emailFeatures` functions on each email to extract a feature vector from each email. This will allow you to build a dataset X , y of examples. You should then randomly divide up the dataset into a training set, a cross validation set and a test set.

While you are building your own dataset, we also encourage you to try building your own vocabulary list (by selecting the high frequency words

³The original emails will have email headers that you might wish to leave out. We have included code in `processEmail` that will help you remove these headers.

that occur in the dataset) and adding any additional features that you think might be useful.

Finally, we also suggest trying to use highly optimized SVM toolboxes such as [LIBSVM](#).

You do not need to submit any solutions for this optional (ungraded) exercise.

Submission and Grading

After completing various parts of the assignment, be sure to use the `submit` function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Gaussian Kernel	<code>gaussianKernel.m</code>	25 points
Parameters (C , σ) for Dataset 3	<code>dataset3Params.m</code>	25 points
Email Preprocessing	<code>processEmail.m</code>	25 points
Email Feature Extraction	<code>emailFeatures.m</code>	25 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration. To prevent rapid-fire guessing, the system enforces a minimum of 5 minutes between submissions.

Programming Exercise 7: *K*-means Clustering and Principal Component Analysis

Machine Learning

Introduction

In this exercise, you will implement the *K*-means clustering algorithm and apply it to compress an image. In the second part, you will use principal component analysis to find a low-dimensional representation of face images. Before starting on the programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics.

To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise. If needed, use the `cd` command in Octave to change to this directory before starting this exercise.

Files included in this exercise

- `ex7.m` - Octave/Matlab script for the first exercise on *K*-means
- `ex7_pca.m` - Octave/Matlab script for the second exercise on PCA
- `ex7data1.mat` - Example Dataset for PCA
- `ex7data2.mat` - Example Dataset for *K*-means
- `ex7faces.mat` - Faces Dataset
- `bird_small.png` - Example Image
- `displayData.m` - Displays 2D data stored in a matrix
- `drawLine.m` - Draws a line over an existing figure
- `plotDataPoints.m` - Initialization for *K*-means centroids
- `plotProgresskMeans.m` - Plots each step of *K*-means as it proceeds
- `runkMeans.m` - Runs the *K*-means algorithm
- [*] `pca.m` - Perform principal component analysis

- [★] `projectData.m` - Projects a data set into a lower dimensional space
- [★] `recoverData.m` - Recovers the original data from the projection
- [★] `findClosestCentroids.m` - Find closest centroids (used in K -means)
- [★] `computeCentroids.m` - Compute centroid means (used in K -means)
- [★] `kMeansInitCentroids.m` - Initialization for K -means centroids

★ indicates files you will need to complete

Throughout the first part of the exercise, you will be using the script `ex7.m`, for the second part you will use `ex7_pca.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

We also strongly encourage using the online **Q&A Forum** to discuss exercises with other students. However, do not look at any source code written by others or share your source code with others.

If you run into network errors using the `submit` script, you can also use an online form for submitting your solutions. To use this *alternative* submission interface, run the `submitWeb` script to generate a submission file (e.g., `submit_ex7_part2.txt`). You can then submit this file through the web submission form in the programming exercises page (go to the programming exercises page, then select the exercise you are submitting for). If you are having no problems submitting through the standard submission system using the `submit` script, you do *not* need to use this alternative submission interface.

1 K -means Clustering

In this this exercise, you will implement the K -means algorithm and use it for image compression. You will first start on an example 2D dataset that will help you gain an intuition of how the K -means algorithm works. After that, you will use the K -means algorithm for image compression by reducing the number of colors that occur in an image to only those that are most common in that image. You will be using `ex7.m` for this part of the exercise.

1.1 Implementing K -means

The K -means algorithm is a method to automatically cluster similar data examples together. Concretely, you are given a training set $\{x^{(1)}, \dots, x^{(m)}\}$ (where $x^{(i)} \in \mathbb{R}^n$), and want to group the data into a few cohesive “clusters”. The intuition behind K -means is an iterative procedure that starts by guessing the initial centroids, and then refines this guess by repeatedly assigning examples to their closest centroids and then recomputing the centroids based on the assignments.

The K -means algorithm is as follows:

```
% Initialize centroids
centroids = kMeansInitCentroids(X, K);
for iter = 1:iterations
    % Cluster assignment step: Assign each data point to the
    % closest centroid. idx(i) corresponds to  $c^{(i)}$ , the index
    % of the centroid assigned to example i
    idx = findClosestCentroids(X, centroids);

    % Move centroid step: Compute means based on centroid
    % assignments
    centroids = computeMeans(X, idx, K);
end
```

The inner-loop of the algorithm repeatedly carries out two steps: (i) Assigning each training example $x^{(i)}$ to its closest centroid, and (ii) Recomputing the mean of each centroid using the points assigned to it. The K -means algorithm will always converge to some final set of means for the centroids. Note that the converged solution may not always be ideal and depends on the initial setting of the centroids. Therefore, in practice the K -means algorithm is usually run a few times with different random initializations. One way to choose between these different solutions from different random initializations is to choose the one with the lowest cost function value (distortion).

You will implement the two phases of the K -means algorithm separately in the next sections.

1.1.1 Finding closest centroids

In the “cluster assignment” phase of the K -means algorithm, the algorithm assigns every training example $x^{(i)}$ to its closest centroid, given the current positions of centroids. Specifically, for every example i we set

$$c^{(i)} := j \quad \text{that minimizes} \quad \|x^{(i)} - \mu_j\|^2,$$

where $c^{(i)}$ is the index of the centroid that is closest to $x^{(i)}$, and μ_j is the position (value) of the j 'th centroid. Note that $c^{(i)}$ corresponds to `idx(i)` in the starter code.

Your task is to complete the code in `findClosestCentroids.m`. This function takes the data matrix `X` and the locations of all centroids inside `centroids` and should output a one-dimensional array `idx` that holds the index (a value in $\{1, \dots, K\}$, where K is total number of centroids) of the closest centroid to every training example.

You can implement this using a loop over every training example and every centroid.

Once you have completed the code in `findClosestCentroids.m`, the script `ex7.m` will run your code and you should see the output `[1 3 2]` corresponding to the centroid assignments for the first 3 examples.

You should now submit your “finding closest centroids” function.

1.1.2 Computing centroid means

Given assignments of every point to a centroid, the second phase of the algorithm recomputes, for each centroid, the mean of the points that were assigned to it. Specifically, for every centroid k we set

$$\mu_k := \frac{1}{|C_k|} \sum_{i \in C_k} x^{(i)}$$

where C_k is the set of examples that are assigned to centroid k . Concretely, if two examples say $x^{(3)}$ and $x^{(5)}$ are assigned to centroid $k = 2$, then you should update $\mu_2 = \frac{1}{2}(x^{(3)} + x^{(5)})$.

You should now complete the code in `computeCentroids.m`. You can implement this function using a loop over the centroids. You can also use a loop over the examples; but if you can use a vectorized implementation that does not use such a loop, your code may run faster.

Once you have completed the code in `computeCentroids.m`, the script `ex7.m` will run your code and output the centroids after the first step of K -means.

You should now submit your compute centroids function.

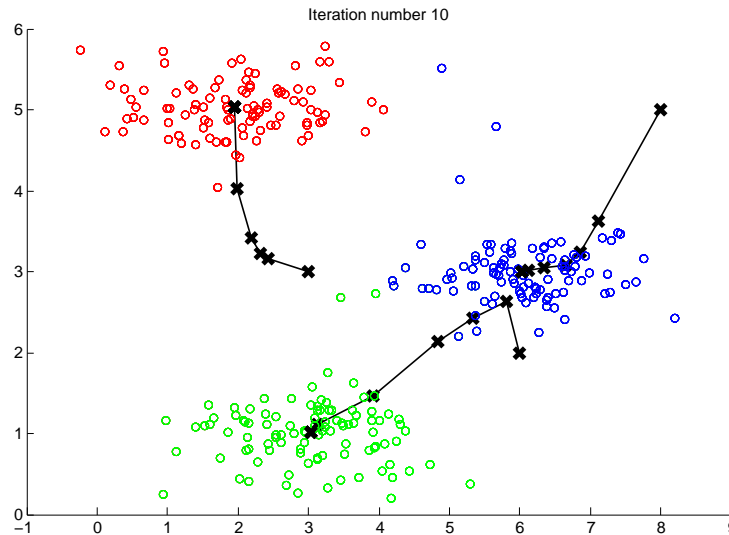


Figure 1: The expected output.

1.2 K -means on example dataset

After you have completed the two functions (`findClosestCentroids` and `computeCentroids`), the next step in `ex7.m` will run the K -means algorithm on a toy 2D dataset to help you understand how K -means works. Your functions are called from inside the `runKmeans.m` script. We encourage you to take a look at the function to understand how it works. Notice that the code calls the two functions you implemented in a loop.

When you run the next step, the K -means code will produce a visualization that steps you through the progress of the algorithm at each iteration. Press *enter* multiple times to see how each step of the K -means algorithm changes the centroids and cluster assignments. At the end, your figure should look as the one displayed in Figure 1.

1.3 Random initialization

The initial assignments of centroids for the example dataset in `ex7.m` were designed so that you will see the same figure as in Figure 1. In practice, a good strategy for initializing the centroids is to select random examples from the training set.

In this part of the exercise, you should complete the function `kMeansInitCentroids.m` with the following code:

```
% Initialize the centroids to be random examples

% Randomly reorder the indices of examples
randidx = randperm(size(X, 1));
% Take the first K examples as centroids
centroids = X(randidx(1:K), :);
```

The code above first randomly permutes the indices of the examples (using `randperm`). Then, it selects the first K examples based on the random permutation of the indices. This allows the examples to be selected at random without the risk of selecting the same example twice.

You do not need to make any submissions for this part of the exercise.

1.4 Image compression with K -means

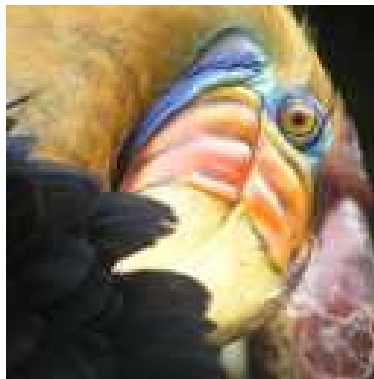


Figure 2: The original 128x128 image.

In this exercise, you will apply K -means to image compression. In a straightforward 24-bit color representation of an image,¹ each pixel is represented as three 8-bit unsigned integers (ranging from 0 to 255) that specify the red, green and blue intensity values. This encoding is often referred to as the RGB encoding. Our image contains thousands of colors, and in this part of the exercise, you will reduce the number of colors to 16 colors.

By making this reduction, it is possible to represent (compress) the photo in an efficient way. Specifically, you only need to store the RGB values of

¹The provided photo used in this exercise belongs to Frank Wouters and is used with his permission.

the 16 selected colors, and for each pixel in the image you now need to only store the index of the color at that location (where only 4 bits are necessary to represent 16 possibilities).

In this exercise, you will use the K -means algorithm to select the 16 colors that will be used to represent the compressed image. Concretely, you will treat every pixel in the original image as a data example and use the K -means algorithm to find the 16 colors that best group (cluster) the pixels in the 3-dimensional RGB space. Once you have computed the cluster centroids on the image, you will then use the 16 colors to replace the pixels in the original image.

1.4.1 K -means on pixels

In Matlab and Octave, images can be read in as follows:

```
% Load 128x128 color image (bird_small.png)
A = imread('bird_small.png');

% You will need to have installed the image package to use
% imread. If you do not have the image package installed, you
% should instead change the following line to
%
%   load('bird_small.mat'); % Loads the image into the variable A
```

This creates a three-dimensional matrix A whose first two indices identify a pixel position and whose last index represents red, green, or blue. For example, $A(50, 33, 3)$ gives the blue intensity of the pixel at row 50 and column 33.

The code inside `ex7.m` first loads the image, and then reshapes it to create an $m \times 3$ matrix of pixel colors (where $m = 16384 = 128 \times 128$), and calls your K -means function on it.

After finding the top $K = 16$ colors to represent the image, you can now assign each pixel position to its closest centroid using the `findClosestCentroids` function. This allows you to represent the original image using the centroid assignments of each pixel. Notice that you have significantly reduced the number of bits that are required to describe the image. The original image required 24 bits for each one of the 128×128 pixel locations, resulting in total size of $128 \times 128 \times 24 = 393,216$ bits. The new representation requires some overhead storage in form of a dictionary of 16 colors, each of which require 24 bits, but the image itself then only requires 4 bits per pixel location. The final number of bits used is therefore $16 \times 24 + 128 \times 128 \times 4 = 65,920$ bits, which corresponds to compressing the original image by about a factor of 6.

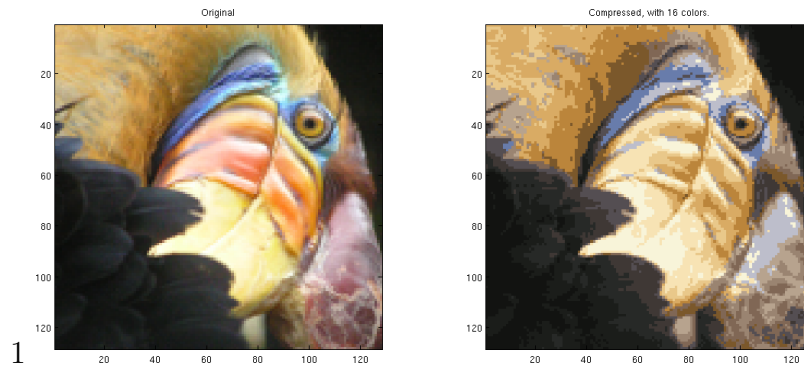


Figure 3: Original and reconstructed image (when using K -means to compress the image).

Finally, you can view the effects of the compression by reconstructing the image based only on the centroid assignments. Specifically, you can replace each pixel location with the mean of the centroid assigned to it. Figure 3 shows the reconstruction we obtained. Even though the resulting image retains most of the characteristics of the original, we also see some compression artifacts.

You do not need to make any submissions for this part of the exercise.

1.5 Optional (ungraded) exercise: Use your own image

In this exercise, modify the code we have supplied to run on one of your own images. Note that if your image is very large, then K -means can take a long time to run. Therefore, we recommend that you resize your images to manageable sizes before running the code. You can also try to vary K to see the effects on the compression.

2 Principal Component Analysis

In this exercise, you will use principal component analysis (PCA) to perform dimensionality reduction. You will first experiment with an example 2D dataset to get intuition on how PCA works, and then use it on a bigger dataset of 5000 face image dataset.

The provided script, `ex7_pca.m`, will help you step through the first half of the exercise.

2.1 Example Dataset

To help you understand how PCA works, you will first start with a 2D dataset which has one direction of large variation and one of smaller variation. The script `ex7_pca.m` will plot the training data (Figure 4). In this part of the exercise, you will visualize what happens when you use PCA to reduce the data from 2D to 1D. In practice, you might want to reduce data from 256 to 50 dimensions, say; but using lower dimensional data in this example allows us to visualize the algorithms better.

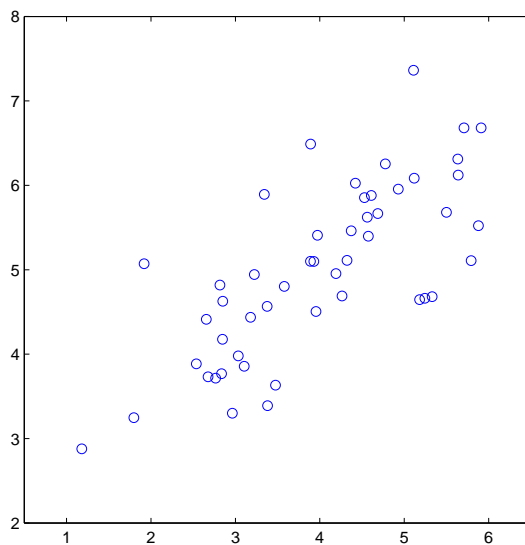


Figure 4: Example Dataset 1

2.2 Implementing PCA

In this part of the exercise, you will implement PCA. PCA consists of two computational steps: First, you **compute the covariance matrix** of the

data. Then, you use Octave's **SVD** function to compute the eigenvectors U_1, U_2, \dots, U_n . These will correspond to the principal components of variation in the data.

Before using PCA, it is important to first normalize the data by subtracting the mean value of each feature from the dataset, and scaling each dimension so that they are in the same range. In the provided script `ex7_pca.m`, this normalization has been performed for you using the `featureNormalize` function.

After normalizing the data, you can run PCA to compute the principal components. Your task is to complete the code in `pca.m` to compute the principal components of the dataset. First, you should compute the covariance matrix of the data, which is given by:

$$\Sigma = \frac{1}{m} X^T X$$

where X is the data matrix with examples in rows, and m is the number of examples. Note that Σ is a $n \times n$ matrix and not the summation operator.

After computing the covariance matrix, you can run SVD on it to compute the principal components. In Octave, you can run SVD with the following command: `[U, S, V] = svd(Sigma)`, where U will contain the principal components and S will contain a diagonal matrix.

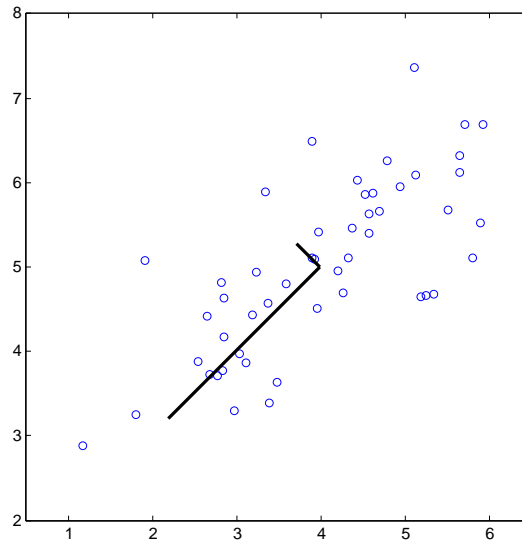


Figure 5: Computed eigenvectors of the dataset

Once you have completed `pca.m`, the `ex7_pca.m` script will run PCA on the example dataset and plot the corresponding principal components found

(Figure 5). The script will also output the top principal component (eigenvector) found, and you should expect to see an output of about $[-0.707 \ -0.707]$. (It is possible that Octave may instead output the negative of this, since U_1 and $-U_1$ are equally valid choices for the first principal component.)

You should now submit your PCA function.

2.3 Dimensionality Reduction with PCA

After computing the principal components, you can use them to reduce the feature dimension of your dataset by projecting each example onto a lower dimensional space, $x^{(i)} \rightarrow z^{(i)}$ (e.g., projecting the data from 2D to 1D). In this part of the exercise, you will use the eigenvectors returned by PCA and project the example dataset into a 1-dimensional space.

In practice, if you were using a learning algorithm such as linear regression or perhaps neural networks, you could now use the projected data instead of the original data. By using the projected data, you can train your model faster as there are less dimensions in the input.

2.3.1 Projecting the data onto the principal components

You should now complete the code in `projectData.m`. Specifically, you are given a dataset X , the principal components U , and the desired number of dimensions to reduce to K . You should project each example in X onto the top K components in U . Note that the top K components in U are given by the first K columns of U , that is `U_reduce = U(:, 1:K)`.

Once you have completed the code in `projectData.m`, `ex7_pca.m` will project the first example onto the first dimension and you should see a value of about 1.481 (or possibly -1.481, if you got $-U_1$ instead of U_1).

You should now submit the project data function.

2.3.2 Reconstructing an approximation of the data

After projecting the data onto the lower dimensional space, you can approximately recover the data by projecting them back onto the original high dimensional space. Your task is to complete `recoverData.m` to project each example in Z back onto the original space and return the recovered approximation in `X_rec`.

Once you have completed the code in `projectData.m`, `ex7_pca.m` will recover an approximation of the first example and you should see a value of about `[-1.047 -1.047]`.

You should now submit the recover data function.

2.3.3 Visualizing the projections

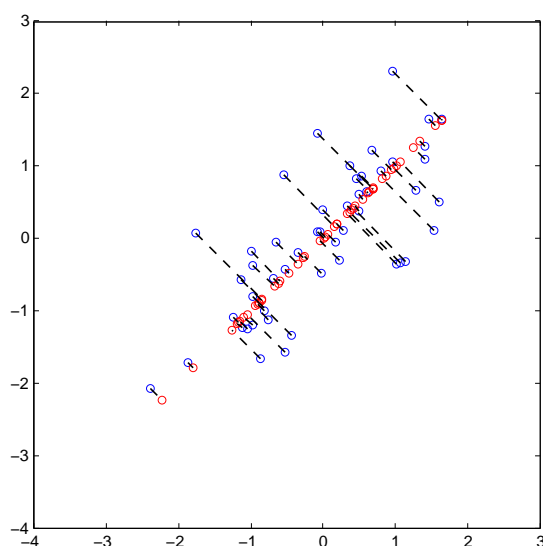


Figure 6: The normalized and projected data after PCA.

After completing both `projectData` and `recoverData`, `ex7_pca.m` will now perform both the projection and approximate reconstruction to show how the projection affects the data. In Figure 6, the original data points are indicated with the blue circles, while the projected data points are indicated with the red circles. The projection effectively only retains the information in the direction given by U_1 .

2.4 Face Image Dataset

In this part of the exercise, you will run PCA on face images to see how it can be used in practice for dimension reduction. The dataset `ex7faces.mat` contains a dataset² X of face images, each 32×32 in grayscale. Each row of X corresponds to one face image (a row vector of length 1024). The next

²This dataset was based on a [cropped version](#) of the [labeled faces in the wild](#) dataset.

step in `ex7_pca.m` will load and visualize the first 100 of these face images (Figure 7).



Figure 7: Faces dataset

2.4.1 PCA on Faces

To run PCA on the face dataset, we first normalize the dataset by subtracting the mean of each feature from the data matrix X . The script `ex7_pca.m` will do this for you and then run your PCA code. After running PCA, you will obtain the principal components of the dataset. Notice that each principal component in U (each row) is a vector of length n (where for the face dataset, $n = 1024$). It turns out that we can visualize these principal components by reshaping each of them into a 32×32 matrix that corresponds to the pixels in the original dataset. The script `ex7_pca.m` displays the first 36 principal components that describe the largest variations (Figure 8). If you want, you can also change the code to display more principal components to see how they capture more and more details.

2.4.2 Dimensionality Reduction

Now that you have computed the principal components for the face dataset, you can use it to reduce the dimension of the face dataset. This allows you to use your learning algorithm with a smaller input size (e.g., 100 dimensions) instead of the original 1024 dimensions. This can help speed up your learning algorithm.



Figure 8: Principal components on the face dataset



Figure 9: Original images of faces and ones reconstructed from only the top 100 principal components.

The next part in `ex7_pca.m` will project the face dataset onto only the first 100 principal components. Concretely, each face image is now described by a vector $z^{(i)} \in \mathbb{R}^{100}$.

To understand what is lost in the dimension reduction, you can recover the data using only the projected dataset. In `ex7_pca.m`, an approximate recovery of the data is performed and the original and projected face images are displayed side by side (Figure 9). From the reconstruction, you can observe that the general structure and appearance of the face are kept while the fine details are lost. This is a remarkable reduction (more than $10\times$) in

the dataset size that can help speed up your learning algorithm significantly. For example, if you were training a neural network to perform person recognition (given a face image, predict the identity of the person), you can use the dimension reduced input of only a 100 dimensions instead of the original pixels.

2.5 Optional (ungraded) exercise: PCA for visualization

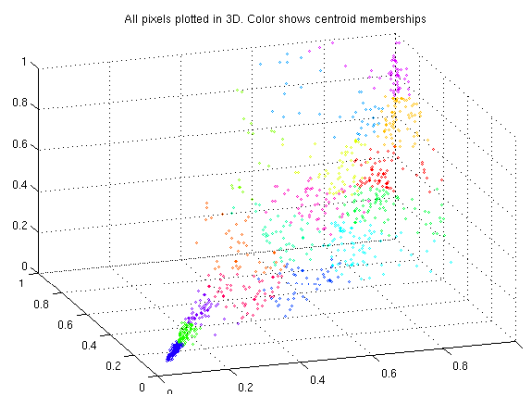


Figure 10: Original data in 3D

In the earlier K -means image compression exercise, you used the K -means algorithm in the 3-dimensional RGB space. In the last part of the `ex7_pca.m` script, we have provided code to visualize the final pixel assignments in this 3D space using the `scatter3` function. Each data point is colored according to the cluster it has been assigned to. You can drag your mouse on the figure to rotate and inspect this data in 3 dimensions.

It turns out that visualizing datasets in 3 dimensions or greater can be cumbersome. Therefore, it is often desirable to only display the data in 2D even at the cost of losing some information. In practice, PCA is often used to reduce the dimensionality of data for visualization purposes. In the next part of `ex7_pca.m`, the script will apply your implementation of PCA to the 3-dimensional data to reduce it to 2 dimensions and visualize the result in a 2D scatter plot. The PCA projection can be thought of as a rotation that selects the view that maximizes the spread of the data, which often corresponds to the “best” view.

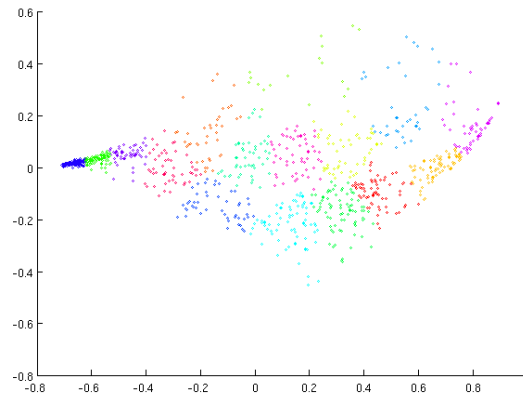


Figure 11: 2D visualization produced using PCA

Submission and Grading

After completing various parts of the assignment, be sure to use the `submit` function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Find Closest Centroids	<code>findClosestCentroids.m</code>	30 points
Compute Centroid Means	<code>computeCentroids.m</code>	30 points
PCA	<code>pca.m</code>	20 points
Project Data	<code>projectData.m</code>	10 points
Recover Data	<code>recoverData.m</code>	10 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration. To prevent rapid-fire guessing, the system enforces a minimum of 5 minutes between submissions.

Programming Exercise 8: Anomaly Detection and Recommender Systems

Machine Learning

Introduction

In this exercise, you will implement the anomaly detection algorithm and apply it to detect failing servers on a network. In the second part, you will use collaborative filtering to build a recommender system for movies. Before starting on the programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics.

To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the exercise. If needed, use the `cd` command in Octave to change to this directory before starting this exercise.

Files included in this exercise

- `ex8.m` - Octave/Matlab script for first part of exercise
- `ex8_cofi.m` - Octave/Matlab script for second part of exercise
- `ex8data1.mat` - First example Dataset for anomaly detection
- `ex8data2.mat` - Second example Dataset for anomaly detection
- `ex8_movies.mat` - Movie Review Dataset
- `ex8_movieParams.mat` - Parameters provided for debugging
- `multivariateGaussian.m` - Computes the probability density function for a Gaussian distribution
- `visualizeFit.m` - 2D plot of a Gaussian distribution and a dataset
- `checkCostFunction.m` - Gradient checking for collaborative filtering
- `computeNumericalGradient.m` - Numerically compute gradients
- `fmincg.m` - Function minimization routine (similar to `fminunc`)
- `loadMovieList.m` - Loads the list of movies into a cell-array

`movie_ids.txt` - List of movies
`normalizeRatings.m` - Mean normalization for collaborative filtering
[*] `estimateGaussian.m` - Estimate the parameters of a Gaussian distribution with a diagonal covariance matrix
[*] `selectThreshold.m` - Find a threshold for anomaly detection
[*] `cofiCostFunc.m` - Implement the cost function for collaborative filtering

★ indicates files you will need to complete

Throughout the first part of the exercise (anomaly detection) you will be using the script `ex8.m`. For the second part of collaborative filtering, you will use `ex8_cofi.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

We also strongly encourage using the online **Q&A Forum** to discuss exercises with other students. However, do not look at any source code written by others or share your source code with others.

If you run into network errors using the `submit` script, you can also use an online form for submitting your solutions. To use this *alternative* submission interface, run the `submitWeb` script to generate a submission file (e.g., `submit_ex8_part2.txt`). You can then submit this file through the web submission form in the programming exercises page (go to the programming exercises page, then select the exercise you are submitting for). If you are having no problems submitting through the standard submission system using the `submit` script, you do *not* need to use this alternative submission interface.

1 Anomaly detection

In this exercise, you will implement an anomaly detection algorithm to detect anomalous behavior in server computers. The features measure the **throughput (mb/s)** and **latency (ms)** of response of each server. While your servers were operating, you collected $m = 307$ examples of how they were behaving,

and thus have an unlabeled dataset $\{x^{(1)}, \dots, x^{(m)}\}$. You suspect that the vast majority of these examples are “normal” (non-anomalous) examples of the servers operating normally, but there might also be some examples of servers acting anomalously within this dataset.

You will use a Gaussian model to detect anomalous examples in your dataset. You will first start on a 2D dataset that will allow you to visualize what the algorithm is doing. On that dataset you will fit a Gaussian distribution and then find values that have very low probability and hence can be considered anomalies. After that, you will apply the anomaly detection algorithm to a larger dataset with many dimensions. You will be using `ex8.m` for this part of the exercise.

The first part of `ex8.m` will visualize the dataset as shown in Figure 1.

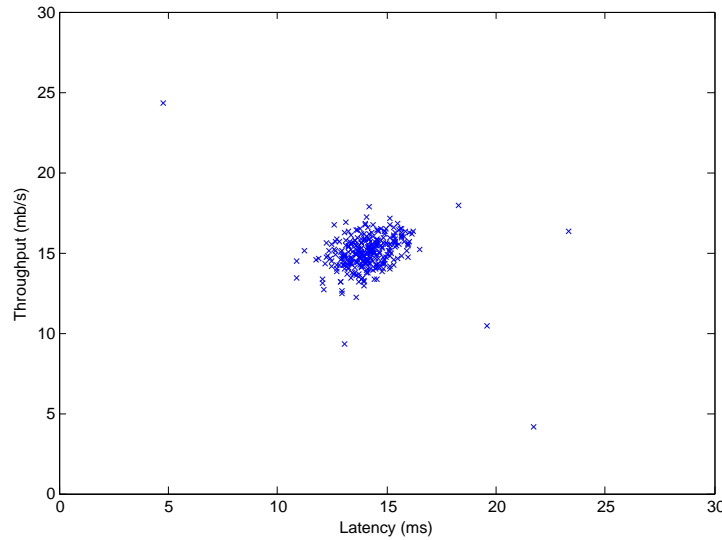


Figure 1: The first dataset.

1.1 Gaussian distribution

To perform anomaly detection, you will first need to fit a model to the data’s distribution.

Given a training set $\{x^{(1)}, \dots, x^{(m)}\}$ (where $x^{(i)} \in \mathbb{R}^n$), you want to estimate the Gaussian distribution for each of the features x_i . For each feature $i = 1 \dots n$, you need to find parameters μ_i and σ_i^2 that fit the data in the i -th dimension $\{x_i^{(1)}, \dots, x_i^{(m)}\}$ (the i -th dimension of each example).

The Gaussian distribution is given by

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ^2 controls the variance.

1.2 Estimating parameters for a Gaussian

You can estimate the parameters, (μ_i, σ_i^2) , of the i -th feature by using the following equations. To estimate the mean, you will use:

$$\mu_i = \frac{1}{m} \sum_{j=1}^m x_i^{(j)}, \quad (1)$$

and for the variance you will use:

$$\sigma_i^2 = \frac{1}{m} \sum_{j=1}^m (x_i^{(j)} - \mu_i)^2. \quad (2)$$

Your task is to complete the code in `estimateGaussian.m`. This function takes as input the data matrix **X** and should output an n -dimension vector **mu** that holds the mean of all the n features and another n -dimension vector **sigma2** that holds the variances of all the features. You can implement this using a for-loop over every feature and every training example (though a vectorized implementation might be more efficient; feel free to use a vectorized implementation if you prefer). Note that in Octave, the `var` function will (by default) use $\frac{1}{m-1}$, instead of $\frac{1}{m}$, when computing σ_i^2 .

Once you have completed the code in `estimateGaussian.m`, the next part of `ex8.m` will visualize the contours of the fitted Gaussian distribution. You should get a plot similar to Figure 2. From your plot, you can see that most of the examples are in the region with the highest probability, while the anomalous examples are in the regions with lower probabilities.

You should now submit your estimate Gaussian parameters function.

1.3 Selecting the threshold, ε

Now that you have estimated the Gaussian parameters, you can investigate which examples have a very high probability given this distribution and which examples have a very low probability. The low probability examples are

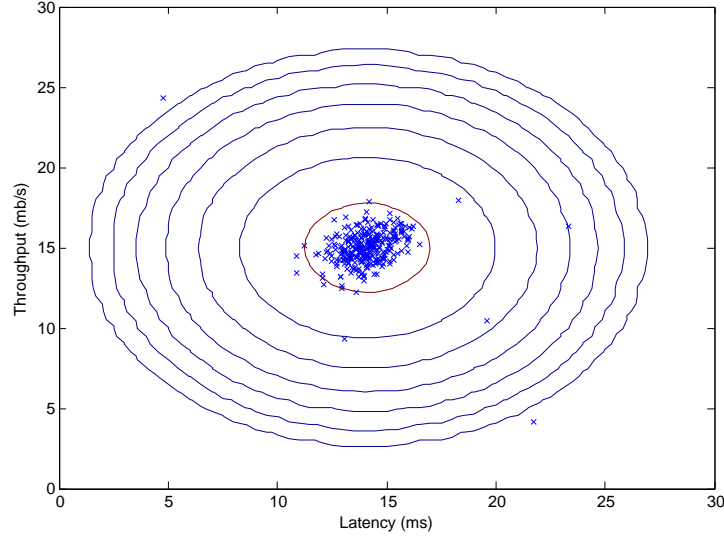


Figure 2: The Gaussian distribution contours of the distribution fit to the dataset.

more likely to be the anomalies in our dataset. One way to determine which examples are anomalies is to select a threshold based on a cross validation set. In this part of the exercise, you will implement an algorithm to select the threshold ε using the F_1 score on a cross validation set.

You should now complete the code in `selectThreshold.m`. For this, we will use a cross validation set $\{(x_{cv}^{(1)}, y_{cv}^{(1)}), \dots, (x_{cv}^{(m_{cv})}, y_{cv}^{(m_{cv})})\}$, where the label $y = 1$ corresponds to an anomalous example, and $y = 0$ corresponds to a normal example. For each cross validation example, we will compute $p(x_{cv}^{(i)})$. The vector of all of these probabilities $p(x_{cv}^{(1)}), \dots, p(x_{cv}^{(m_{cv})})$ is passed to `selectThreshold.m` in the vector `pval`. The corresponding labels $y_{cv}^{(1)}, \dots, y_{cv}^{(m_{cv})}$ is passed to the same function in the vector `yval`.

The function `selectThreshold.m` should return two values; the first is the selected threshold ε . If an example x has a low probability $p(x) < \varepsilon$, then it is considered to be an anomaly. The function should also return the F_1 score, which tells you how well you're doing on finding the ground truth anomalies given a certain threshold. For many different values of ε , you will compute the resulting F_1 score by computing how many examples the current threshold classifies correctly and incorrectly.

The F_1 score is computed using precision (*prec*) and recall (*rec*):

$$F_1 = \frac{2 \cdot \text{prec} \cdot \text{rec}}{\text{prec} + \text{rec}}, \quad (3)$$

You compute precision and recall by:

$$prec = \frac{tp}{tp + fp} \quad (4)$$

$$rec = \frac{tp}{tp + fn}, \quad (5)$$

where

- tp is the number of true positives: the ground truth label says it's an anomaly and our algorithm correctly classified it as an anomaly.
- fp is the number of false positives: the ground truth label says it's not an anomaly, but our algorithm incorrectly classified it as an anomaly.
- fn is the number of false negatives: the ground truth label says it's an anomaly, but our algorithm incorrectly classified it as not being anomalous.

In the provided code `selectThreshold.m`, there is already a loop that will try many different values of ε and select the best ε based on the F_1 score.

You should now complete the code in `selectThreshold.m`. You can implement the computation of the F_1 score using a for-loop over all the cross validation examples (to compute the values tp , fp , fn). You should see a value for epsilon of about 8.99e-05.

Implementation Note: In order to compute tp , fp and fn , you may be able to use a vectorized implementation rather than loop over all the examples. This can be implemented by Octave's equality test between a vector and a single number. If you have several binary values in an n -dimensional binary vector $v \in \{0, 1\}^n$, you can find out how many values in this vector are 0 by using: `sum(v == 0)`. You can also apply a logical **and** operator to such binary vectors. For instance, let `cvPredictions` be a binary vector of the size of your number of cross validation set, where the i -th element is 1 if your algorithm considers $x_{cv}^{(i)}$ an anomaly, and 0 otherwise. You can then, for example, compute the number of false positives using: `fp = sum((cvPredictions == 1) & (yval == 0))`.

Once you have completed the code in `selectThreshold.m`, the next step in `ex8.m` will run your anomaly detection code and circle the anomalies in the plot (Figure 3).

You should now submit your select threshold function.

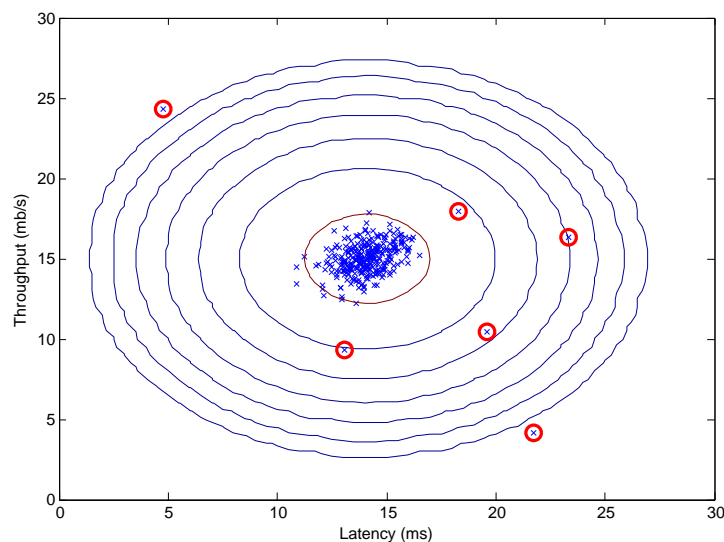


Figure 3: The classified anomalies.

1.4 High dimensional dataset

The last part of the script `ex8.m` will run the anomaly detection algorithm you implemented on a more realistic and much harder dataset. In this dataset, each example is described by 11 features, capturing many more properties of your compute servers.

The script will use your code to estimate the Gaussian parameters (μ_i and σ_i^2), evaluate the probabilities for both the training data \mathbf{X} from which you estimated the Gaussian parameters, and do so for the the cross-validation set \mathbf{Xval} . Finally, it will use `selectThreshold` to find the best threshold ϵ . You should see a value epsilon of about $1.38\text{e-}18$, and 117 anomalies found.

2 Recommender Systems

In this part of the exercise, you will implement the collaborative filtering learning algorithm and apply it to a dataset of movie ratings.¹ This dataset consists of ratings on a scale of 1 to 5. The dataset has $n_u = 943$ users, and $n_m = 1682$ movies. For this part of the exercise, you will be working with the script `ex8_cofi.m`.

¹[MovieLens 100k Dataset](#) from GroupLens Research.

In the next parts of this exercise, you will implement the function `cofiCostFunc.m` that computes the collaborative filtering objective function and gradient. After implementing the cost function and gradient, you will use `fmincg.m` to learn the parameters for collaborative filtering.

2.1 Movie ratings dataset

The first part of the script `ex8_cofi.m` will load the dataset `ex8_movies.mat`, providing the variables `Y` and `R` in your Octave environment.

The matrix Y (a `num_movies` \times `num_users` matrix) stores the ratings $y^{(i,j)}$ (from 1 to 5). The matrix R is a binary-valued indicator matrix, where $R(i, j) = 1$ if user j gave a rating to movie i , and $R(i, j) = 0$ otherwise. The objective of collaborative filtering is to predict movie ratings for the movies that users have not yet rated, that is, the entries with $R(i, j) = 0$. This will allow us to recommend the movies with the highest predicted ratings to the user.

To help you understand the matrix `Y`, the script `ex8_cofi.m` will compute the average movie rating for the first movie (Toy Story) and output the average rating to the screen.

Throughout this part of the exercise, you will also be working with the matrices, `X` and `Theta`:

$$\mathbf{X} = \begin{bmatrix} \text{---} (x^{(1)})^T \text{---} \\ \text{---} (x^{(2)})^T \text{---} \\ \vdots \\ \text{---} (x^{(n_m)})^T \text{---} \end{bmatrix}, \quad \mathbf{\Theta} = \begin{bmatrix} \text{---} (\theta^{(1)})^T \text{---} \\ \text{---} (\theta^{(2)})^T \text{---} \\ \vdots \\ \text{---} (\theta^{(n_u)})^T \text{---} \end{bmatrix}.$$

The i -th row of `X` corresponds to the feature vector $x^{(i)}$ for the i -th movie, and the j -th row of `Theta` corresponds to one parameter vector $\theta^{(j)}$, for the j -th user. Both $x^{(i)}$ and $\theta^{(j)}$ are n -dimensional vectors. For the purposes of this exercise, you will use $n = 100$, and therefore, $x^{(i)} \in \mathbb{R}^{100}$ and $\theta^{(j)} \in \mathbb{R}^{100}$. Correspondingly, `X` is a $n_m \times 100$ matrix and `Theta` is a $n_u \times 100$ matrix.

2.2 Collaborative filtering learning algorithm

Now, you will start implementing the collaborative filtering learning algorithm. You will start by implementing the cost function (without regularization).

The collaborative filtering algorithm in the setting of movie recommendations considers a set of n -dimensional parameter vectors $x^{(1)}, \dots, x^{(n_m)}$ and

$\theta^{(1)}, \dots, \theta^{(n_u)}$, where the model predicts the rating for movie i by user j as $y^{(i,j)} = (\theta^{(j)})^T x^{(i)}$. Given a dataset that consists of a set of ratings produced by some users on some movies, you wish to learn the parameter vectors $x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)}$ that produce the best fit (minimizes the squared error).

You will complete the code in `cofiCostFunc.m` to compute the cost function and gradient for collaborative filtering. Note that the parameters to the function (i.e., the values that you are trying to learn) are **X** and **Theta**. In order to use an off-the-shelf minimizer such as `fmincg`, the cost function has been set up to unroll the parameters into a single vector **params**. You had previously used the same vector unrolling method in the neural networks programming exercise.

2.2.1 Collaborative filtering cost function

The collaborative filtering cost function (without regularization) is given by

$$J(x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)}) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2.$$

You should now modify `cofiCostFunc.m` to return this cost in the variable **J**. Note that you should be accumulating the cost for user j and movie i only if $R(i, j) = 1$.

After you have completed the function, the script `ex8_cofi.m` will run your cost function. You should expect to see an output of 22.22.

You should now submit your cost function.

Implementation Note: We strongly encourage you to use a vectorized implementation to compute J , since it will later be called many times by the optimization package `fmincg`. As usual, it might be easiest to first write a non-vectorized implementation (to make sure you have the right answer), and then modify it to become a vectorized implementation (checking that the vectorization steps don't change your algorithm's output). To come up with a vectorized implementation, the following tip might be helpful: You can use the `R` matrix to set selected entries to 0. For example, `R .* M` will do an element-wise multiplication between `M` and `R`; since `R` only has elements with values either 0 or 1, this has the effect of setting the elements of `M` to 0 only when the corresponding value in `R` is 0. Hence, `sum(sum(R.*M))` is the sum of all the elements of `M` for which the corresponding element in `R` equals 1.

2.2.2 Collaborative filtering gradient

Now, you should implement the gradient (without regularization). Specifically, you should complete the code in `cofiCostFunc.m` to return the variables `X_grad` and `Theta_grad`. Note that `X_grad` should be a matrix of the same size as `X` and similarly, `Theta_grad` is a matrix of the same size as `Theta`. The gradients of the cost function is given by:

$$\frac{\partial J}{\partial x_k^{(i)}} = \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) \theta_k^{(j)}$$

$$\frac{\partial J}{\partial \theta_k^{(j)}} = \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)}.$$

Note that the function returns the gradient for both sets of variables by unrolling them into a single vector. After you have completed the code to compute the gradients, the script `ex8_cofi.m` will run a gradient check (`checkCostFunction`) to numerically check the implementation of your gradients.² If your implementation is correct, you should find that the analytical and numerical gradients match up closely.

You should now submit your collaborative filtering gradient function.

²This is similar to the numerical check that you used in the neural networks exercise.

Implementation Note: You can get full credit for this assignment without using a vectorized implementation, but your code will run much more slowly (a small number of hours), and so we recommend that you try to vectorize your implementation.

To get started, you can implement the gradient with a for-loop over movies (for computing $\frac{\partial J}{\partial x_k^{(i)}}$) and a for-loop over users (for computing $\frac{\partial J}{\partial \theta_k^{(j)}}$). When you first implement the gradient, you might start with an unvectorized version, by implementing another inner for-loop that computes each element in the summation. After you have completed the gradient computation this way, you should try to vectorize your implementation (vectorize the inner for-loops), so that you're left with only two for-loops (one for looping over movies to compute $\frac{\partial J}{\partial x_k^{(i)}}$ for each movie, and one for looping over users to compute $\frac{\partial J}{\partial \theta_k^{(j)}}$ for each user).

Implementation Tip: To perform the vectorization, you might find this helpful: You should come up with a way to compute all the derivatives associated with $x_1^{(i)}, x_2^{(i)}, \dots, x_n^{(i)}$ (i.e., the derivative terms associated with the feature vector $x^{(i)}$) at the same time. Let us define the derivatives for the feature vector of the i -th movie as:

$$(\mathbf{X}_{\text{grad}}(\mathbf{i}, :))^T = \begin{bmatrix} \frac{\partial J}{\partial x_1^{(i)}} \\ \frac{\partial J}{\partial x_2^{(i)}} \\ \vdots \\ \frac{\partial J}{\partial x_n^{(i)}} \end{bmatrix} = \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) \theta^{(j)}$$

To vectorize the above expression, you can start by indexing into **Theta** and **Y** to select only the elements of interests (that is, those with $r(i, j) = 1$). Intuitively, when you consider the features for the i -th movie, you only need to be concern about the users who had given ratings to the movie, and this allows you to remove all the other users from **Theta** and **Y**.

Concretely, you can set `idx = find(R(i, :)==1)` to be a list of all the users that have rated movie i . This will allow you to create the temporary matrices **Theta**_{temp} = **Theta**(idx, :) and **Y**_{temp} = **Y**(i, idx) that index into **Theta** and **Y** to give you only the set of users which have rated the i -th movie. This will allow you to write the derivatives as:

$$\mathbf{X}_{\text{grad}}(\mathbf{i}, :) = (\mathbf{X}(\mathbf{i}, :) * \mathbf{Theta}_{\text{temp}}^T - \mathbf{Y}_{\text{temp}}) * \mathbf{Theta}_{\text{temp}}.$$

(Note: The vectorized computation above returns a row-vector instead.)

After you have vectorized the computations of the derivatives with respect to $x^{(i)}$, you should use a similar method to vectorize the derivatives with respect to $\theta^{(j)}$ as well.

2.2.3 Regularized cost function

The cost function for collaborative filtering with regularization is given by

$$J(x^{(1)}, \dots, x^{(n_m)}, \theta^{(1)}, \dots, \theta^{(n_u)}) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \left(\frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^n (\theta_k^{(j)})^2 \right) + \left(\frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^n (x_k^{(i)})^2 \right).$$

You should now add regularization to your original computations of the cost function, J . After you are done, the script `ex8_cofi.m` will run your regularized cost function, and you should expect to see a cost of about 31.34.

You should now submit your regularized cost function.

2.2.4 Regularized gradient

Now that you have implemented the regularized cost function, you should proceed to implement regularization for the gradient. You should add to your implementation in `cofiCostFunc.m` to return the regularized gradient by adding the contributions from the regularization terms. Note that the gradients for the regularized cost function is given by:

$$\frac{\partial J}{\partial x_k^{(i)}} = \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) \theta_k^{(j)} + \lambda x_k^{(i)}$$

$$\frac{\partial J}{\partial \theta_k^{(j)}} = \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)}.$$

This means that you just need to add $\lambda x^{(i)}$ to the `X_grad(i,:)` variable described earlier, and add $\lambda \theta^{(j)}$ to the `Theta_grad(j,:)` variable described earlier.

After you have completed the code to compute the gradients, the script `ex8_cofi.m` will run another gradient check (`checkCostFunction`) to numerically check the implementation of your gradients.

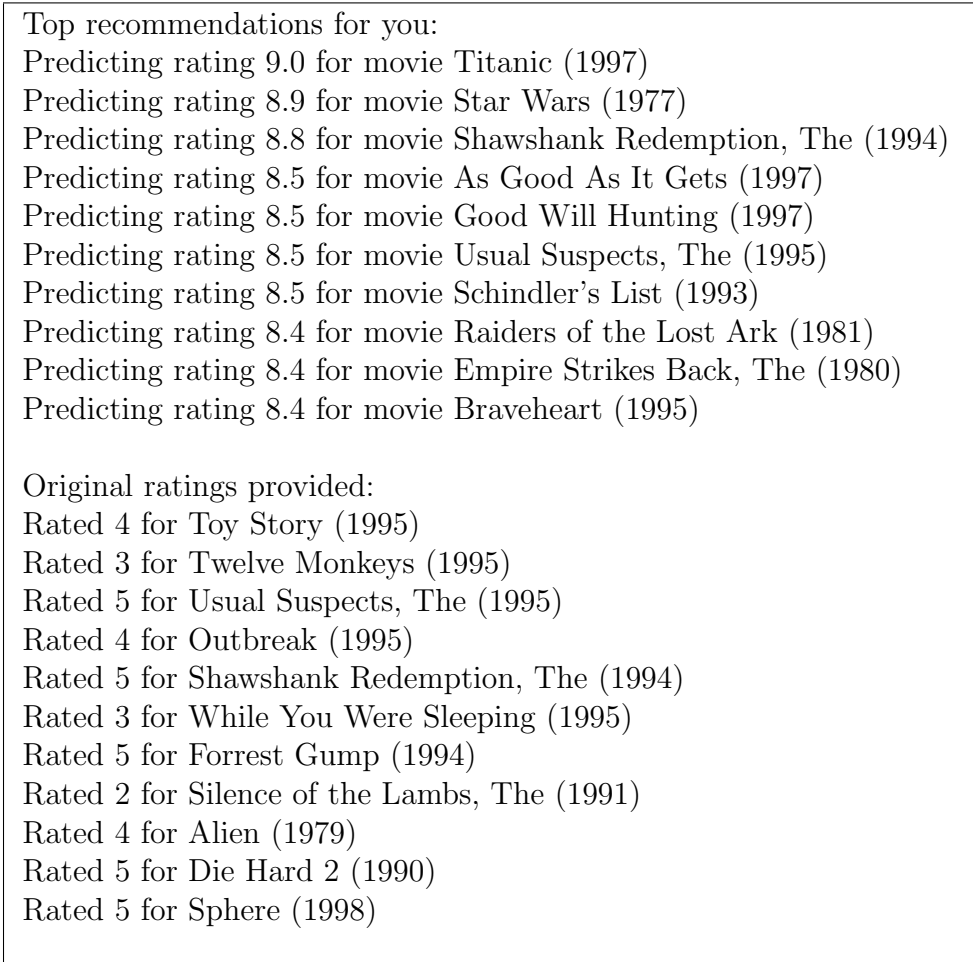
You should now submit the regularized gradient function.

2.3 Learning movie recommendations

After you have finished implementing the collaborative filtering cost function and gradient, you can now start training your algorithm to make movie

recommendations for yourself. In the next part of the `ex8_cofi.m` script, you can enter your own movie preferences, so that later when the algorithm runs, you can get your own movie recommendations! We have filled out some values according to our own preferences, but you should change this according to your own tastes. The list of all movies and their number in the dataset can be found listed in the file `movie_idx.txt`.

2.3.1 Recommendations



Top recommendations for you:

- Predicting rating 9.0 for movie Titanic (1997)
- Predicting rating 8.9 for movie Star Wars (1977)
- Predicting rating 8.8 for movie Shawshank Redemption, The (1994)
- Predicting rating 8.5 for movie As Good As It Gets (1997)
- Predicting rating 8.5 for movie Good Will Hunting (1997)
- Predicting rating 8.5 for movie Usual Suspects, The (1995)
- Predicting rating 8.5 for movie Schindler's List (1993)
- Predicting rating 8.4 for movie Raiders of the Lost Ark (1981)
- Predicting rating 8.4 for movie Empire Strikes Back, The (1980)
- Predicting rating 8.4 for movie Braveheart (1995)

Original ratings provided:

- Rated 4 for Toy Story (1995)
- Rated 3 for Twelve Monkeys (1995)
- Rated 5 for Usual Suspects, The (1995)
- Rated 4 for Outbreak (1995)
- Rated 5 for Shawshank Redemption, The (1994)
- Rated 3 for While You Were Sleeping (1995)
- Rated 5 for Forrest Gump (1994)
- Rated 2 for Silence of the Lambs, The (1991)
- Rated 4 for Alien (1979)
- Rated 5 for Die Hard 2 (1990)
- Rated 5 for Sphere (1998)

Figure 4: Movie recommendations

After the additional ratings have been added to the dataset, the script will proceed to train the collaborative filtering model. This will learn the parameters \mathbf{X} and Θ . To predict the rating of movie i for user j , you need

to compute $(\theta^{(j)})^T x^{(i)}$. The next part of the script computes the ratings for all the movies and users and displays the movies that it recommends (Figure 4), according to ratings that were entered earlier in the script. Note that you might obtain a different set of the predictions due to different random initializations.

Submission and Grading

After completing various parts of the assignment, be sure to use the `submit` function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Estimate Gaussian Parameters	<code>estimateGuassian.m</code>	15 points
Select Threshold	<code>selectThreshold.m</code>	15 points
Collaborative Filtering Cost	<code>cofiCostFunc.m</code>	20 points
Collaborative Filtering Gradient	<code>cofiCostFunc.m</code>	30 points
Regularized Cost	<code>cofiCostFunc.m</code>	10 points
Gradient with regularization	<code>cofiCostFunc.m</code>	10 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration. To prevent rapid-fire guessing, the system enforces a minimum of 5 minutes between submissions.