Ryerson University

Department of Electrical and Computer Engineering

ELE 888 / EE 8209 - Intelligent Systems (Machine Learning)

Winter 2017 – Lab # 2

Logistic Regression and Bayesian Decision Theory

Author: Raymond Phan

Edited by: Md Moinuddin Bhuiyan

Objectives

In this lab assignment, you will implement logistic regression for binary and multi-class classification and see how it works with both synthetic and real data. You will also explore anonymous functions and how they can be used for minimizing cost functions with MATLAB's Optimization Toolbox.

For the binary case, you will explore how decision boundaries are used to separate synthetic data into their positive and negative classes for both linearly separable data (i.e. the decision boundary is a line) and linearly non-separable data (i.e. the decision boundary is complex). You will also explore training your model to get the model parameters by adding regularization and seeing how the decision boundary is affected for linearly non-separable data. Analyzing the accuracy of these classifiers will also be explored.

For multi-class classification, you will perform the task of classifying hand-written digits where the training set consists of digitzed images of hand-written digits created by various people. This is to be implemented in a One-Vs-All framework and you will determine the accuracy of your multi-class classification framework by classifying hand-written digits in a test dataset where these images were not part of the training.

Finally, you will also explore aspects of Bayesian Decision Theory for the binary case as an alternative method to performing classification. Specifically, you will consider the features collected for the training dataset to fall under a Gaussian distribution and using this knowledge to apply Bayes decision rule for binary classification.

List of Files Included

In this lab assignment, you are given some files with code already completed to help you complete this lab assignment. These are primarily for setup so you can concentrate on actually implementing machine learning algorithms rather than being bogged down with minor minutiae. There are additionally other files that are skeletons where it is your task to complete these files or functions and submit them with your lab report.

Demonstration Scripts

These are scripts that help familiarize yourself with the lab and may help you in completing the required portions of this lab.

- `univariate_reg_cost.m` : This is a MATLAB function script file that demonstrates what is expected as input into any of the functions found in MATLAB's Optimization Toolbox. This function computes the cost and derivative / gradient terms of the parameters for univariate linear regression. These are computed with respect to an input set of parameters.
- `univariate_reg_demo.m` : This is a MATLAB script file that demonstrates how to use MATLAB's Optimization Toolbox so that the right parameters for univariate linear regression are found that minimize the

cost function for this particular problem.

Dataset Files

These are MATLAB MAT files that contain datasets that will be used for this lab. These are stored in the `data` directory of the lab and these files are:

- `lab2dataq1a.mat` : A synthetic dataset consisting of two features and associated binary labels: 0 (negative) or 1 (positive). This data is (more or less) linearly separable. This was taken from Andrew Ng's online Machine Learning course through `coursera.org` .
- `lab2dataq1b.mat` : A synthetic dataset much like `lab2dataq1a.mat` , except that the data is non-linearly separable. This was also taken from Andrew Ng's `coursera.org` course.
- `lab2digits.mat` : A real dataset from the Mixed National Institute of Standards and Technology (MNIST) database that contains digitized hand-written digits. This was created by Yann LeCun who was formerly from New York University and is now director of Facebook AI Research. The link to obtain the dataset can be found here: http://yann.lecun.com/exdb/mnist/.
- `lab2flowers.mat` : A real dataset created by Ronald A. Fisher which is called the Iris Dataset. The original source can be found here on the UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Iris. This dataset consists of sepal lengths and widths as well as the petal lengths and widths for a sampling of three different flowers: Iris Setosa, Iris Versicolour and Iris Viriginica.

Helper Functions

These are scripts that are designed to help you complete the lab. Some other functions are used to help you give a sense of the problems you are attempting to solve in this lab. These are stored in the `helper` directory of the lab.

The following is a list of functions for each part of this lab.

For Part 1

- `create_polynomial_features` : A function that returns a data matrix of non-linear features given two feature vectors. This matrix returns combinations of products of powers of these features as well as taking the features by themselves raised to some powers. This is to be used in the first part of the lab.

For Part 2

- `fmincg` : This is a function written by Carl Edward Rasmussen from the University of Cambridge. Some modifications were made by Andrew Ng and further modifications were made by Raymond Phan. Essentially, this finds the minimum of a cost function very much like `fminunc` in MATLAB but uses the Conjugate Gradient method for finding the minimum. This function is included because it is especially optimized for handling data with many features and it is also more memory efficient. This will especially be useful when you perform multi-class classification with the hand-written digit dataset included with this lab.
- `show_digits.m` : This is a function that randomly shows `n` hand-written digitized images given a 3D matrix of digit images. Each slice in this matrix is one image. This is to give a sense of what the digits looks like as a whole.
- `show_random_digits.m` : This is a function that shows 100 random digits for each digit from 0 to 9. This is to give a sense of what each digit looks like.
- `show_misclassified_digits.m` : This is a function that takes in a 3D matrix of digit images with its corresponding **predicted** labels of each digit created by you as well as the **true labels** of each digit, and for a certain number of digits `K` and digit in question, we examine `K` images that were misclassified for a

particular digit. This is primarily to be used for discussion purposes.

For Part 3

- `calculate_decision_boundary` : This function calculates the decision boundary for the binary classification problem using Bayesian Decision Theory provided that we are considering only a single feature and it is Gaussian distributed. The output is a prediction function that decides which class an input belongs to.

To complete

You are required to complete an informal lab report regarding the completed tasks of this lab. All questions that pertain to results in the lab, specifically any answers to observational questions as well as calculations and figures to be included are expected to be placed in this report. Please consult the Lab Guidelines document on our course website for more details.

In addition, these are the scripts you need to complete for this lab. Each script will either be a skeleton that you need to complete in functionality or is a function that you need to write with a few lines of code to get you started.

- `sigmoid.m` : A function that computes the sigmoid term for every element in an input array or matrix.
- `binary_predictor.m` : Given a data matrix and a vector of parameters, this function predicts whether an input from the data matrix belongs to the positive or negative class.
- `lr_cost_function.m` : A cost function compatible for use with MATLAB's Optimization Toolbox that implements logistic regression without regularization
- `lr_cost_function_reg.m` : The same as previously but this also incorporates regularization to prevent overfitting.
- `one_vs_all` : Given a data matrix and a vector of labels that determine which class each training example belongs to, the goal is to return a **matrix** of parameters where each column is the parameters trained to best classify a particular class.
- `multiclass_predict.m` : Given a data matrix and a **matrix** of parameters where each column is the parameters for a particular classifier, this function predicts which class each input belongs to using the One-Vs-All approach.
- `part1a.m` : The first half of Part 1 of this lab for you to complete.
- `part1b.m` : The second half of Part 1 of this lab for you to complete.
- `part2.m` : Part 2 of the lab for you to complete.
- `part3.m` : Part 3 of the lab for you to complete.

Warmup

This section is strictly for your benefit. There is nothing to write up about in your report here. Please read to get the fullest and rewarding experience in your lab.

Optimization Toolbox

In this section, we will talk about how to use MATLAB's Optimization Toolbox for minimizing cost functions. In the previous lab, you used Gradient Descent as a means of finding the minimum of convex cost functions as well as the associated parameters required to provide this minimum. However, the disadvantage of Gradient Descent is that you are required to set the appropriate learning rate so that the algorithm converges. In addition, should your

features have different dynamic ranges, you are required to normalize each feature so that all features have relatively the same dynamic range to ensure convergence.

MATLAB is equipped with high-performance and efficient optimization algorithms from the Optimization Toolbox that allow you to do the equivalent of Gradient Descent and can achieve results in a quicker and more efficient way. If you recall from the lectures, you can use the function `fminunc`, which stands for **F**unction **Min**imization **Unc**onstrained. It is an iterative algorithm where given some initial parameters, it finds the optimum parameters to best minimize a cost function much like Gradient Descent.

Inputs

What you provide to `fminunc` are three input parameters:

- `fun` : A function to minimize. For our purposes, this is our cost function $J(\theta)$ and we seek to find the parameters $\theta = (\theta_0, \theta_1, \dots, \theta_n)^T$ that best minimize this function. n is the total number of features for our machine learning problem and there is an additional parameter for the bias θ_0.

  This function has the following prototype:

  1. Inputs: There is only one input into this function
     - `theta` : An input vector of parameters θ `theta = [theta_0; theta_1; ...; theta_n];` where the vector is `n + 1` elements long and is a column vector. `n` is the total number of features for your problem. This input represents the current state of `fminunc` while trying to find the optimal parameters and you are to calculate what the cost is using these parameters $J(\theta)$ as well as the derivative term for each parameter $\partial J/\partial\theta_j$ for $j = 0, 1, \dots, n$.
  2. Outputs: The output consists of two elements:
     - `cost` : The cost $J(\theta)$ using the input parameters *theta*
     - `grad` : A `n + 1` column vector that represents the derivative term for each parameter $\partial J/\partial\theta_j$ for $j = 0, 1, \dots, n$.

  Therefore the expected MATLAB prototype would look something like this:

  ```
  function [cost, grad] = costFunction(theta)
  ```

  `costFunction` is the name of the file that contains the implementation of the above behaviour (i.e. it would be a file called `costFunction.m`). You can change the name `costFunction` to whatever it is that is most convenient for you so long as the file name that is associated with this function is the same as the name of the function itself.

- `init_theta` : The initial set of parameters used to serve as a starting point (i.e. `init_theta = zeros(n+1,1);`).

- `options` : This is an input structure that defines options on how `fminunc` will work. You can create the structure using the `optimset`. Using `optimset` by itself without any input parameters gives you default options but you can override the options by specifying inputs into `optimset`. These are key/value pairs where the key is the parameter you want to modify and the value is what you want to modify the parameter to. The most usual setup for our purposes is the following:

  ```
  options = optimset('GradObj', 'on', 'MaxIter', N);
  ```

  The `GradObj` flag tells `fminunc` that you are going to provide the derivative of each parameter of the cost function manually. Not doing this will approximate this using discrete differences which will be much slower. It is better if you can manually specify the derivative term of each parameter (i.e. the gradient) of your cost

function whenever possible. `MaxIter` specifies the **maximum** number of iterations to be performed in `fminunc`, which is `N` in the above code. It is very possible that the algorithm will not take `N` iterations, especially if convergence has been detected, so you can play around with this value to ensure convergence of your machine learning problem if you have a setup that doesn't converge quite easily. However, the default value for `MaxIter` is 400.

Outputs

The outputs of `fminunc` consists of three parameters:

- `opt_theta` : The optimal parameters that minimize the cost function $J(\theta)$.
- `cost_val` : The cost to assign the parameters given by `opt_theta` with $J(\theta)$.
- `exit_flag` : An indicator of what happened in `fminunc` when the algorithm stops. You can read the documentation of `fminunc` for more details, but what you want to avoid is seeing `exit_flag` being 0 or negative. `exit_flag = 0` means that the algorithm exceeded the maximum number of iterations so the algorithm may or may not have converged and negative values means that it simply couldn't find a solution. All positive values indicate convergence.

Feature Normalization

Because we are using a highly optimized minimization algorithm, there is **no need** for you to normalize the features. The algorithms are intelligent enough to deduce this and so there is even less work that you have to perform before hand. This makes finding the optimal parameters much more easier to find. As such, when you have access to the data matrix, don't worry about normalizing the features and just use the data as is.

Sidenote: `fmincg` and its use in this lab

`fminunc` is very good to use provided that you don't have as many features in your machine learning problem (i.e. between 10 - 50 features). However, the drawback with `fminunc` is that it requires a lot of memory / RAM to be able to compute the optimal parameters. In addition, should you have many features (i.e. 100s or even 1000s), `fminunc` is known to be quite slow and can sometimes either run out of memory when attempting to find the optimal parameters or not even converge.

Therefore, for the second part of this lab we will be using a variant of `fminunc` called **`fmincg`** : **F**unction **Min**imization **C**onjugate **G**radient. This function is **not** part of MATLAB's Optimization Toolbox and is a file that is included with your lab for your use. This was created by Carl Edward Rasmussen from the University of Cambridge. This variant **does not** achieve a more accurate result in comparison to `fminunc` . It is simply more efficient, gets to the optimum much more quickly and performs a variant to Gradient Descent where the cost function is assumed to adopt certain properties. What's good about `fmincg` is that you call it **exactly the same way as you'd call `fminunc`** . Therefore, all you would need is to replace your call of `fminunc` with `fmincg` and you're good to go.

> *You can see this discussion on Stack Overflow with regards to the differences between `fminunc` and `fmincg` here: http://stackoverflow.com/questions/12115087/octave-logistic-regression-difference-between-fmincg-and-fminunc*

However, you can specify an additional fourth input called `debug` where you set this to `true` or `false` if you want to see the cost function at each iteration until the algorithm converges. This may be useful if you want to double check if the cost function and derivative vector you specified is correct. You can omit this parameter and

this will assume that `debug` is `false`, meaning this verbose output won't be seen. However, your instructor highly encourages you to leave this flag on so you can see how the cost is behaving over the iterations.

Optimization Demo

In class, we briefly went over how you'd use the Optimization Toolbox to get the correct parameters for univariate linear regression via Gradient Descent. However, the first input into `fminunc/fmincg` is a function that relies only on the input parameters at a particular iteration in the algorithm. Because we want to use the input and output data to compute the cost function and the associated derivative terms, you **must** modify this behaviour by creating an **anonymous function** that accepts only the input parameters but also captures the input and expected output data. In addition, your cost function file (i.e. `costFunction`) must now accommodate the inputs and outputs.

As such, your new cost function file would now look something like this:

```
function [cost, grad] = costFunction(X, y, theta)
```

`X` is a data matrix of compatible dimensions where each row is an input sample and each column is a feature. You'll want to make sure that the total number of columns matches the size of the expected parameter vector. `y` is a column vector of expected outputs that spans the same number of rows that `X` has. Finally, `theta` is the parameter vector as we have seen before.

Once you modify your cost function, assuming you have already set up your cost function and it's in a file called `costFunction.m` and assuming that `X` and `y` are already defined in your MATLAB code, you would do something like the following:

```
func = @(theta) costFunction(X, y, theta);
```

The `@(theta)` means that you want to create an anonymous function that will be stored in the variable `func` where the function only takes in one parameter: `theta`. This effectively wraps the cost function so that you're only using **a single variable**: `theta`. `X` and `y` were previously defined and are **captured** when creating the anonymous function. As such, any time you use `func` it remembers what state `X` and `y` were in at the point of creation and it uses that when calling your cost function stored in `costFunction.m`. What's important is that if you decide to change `X` and `y` after you create this anonymous function, when calling `func`, `X` and `y` are what they were **before you created `func`** so the changes are not affected. This is what is known as lexical scope#Lexical_scope_vs._dynamic_scope) in programming parlance.

As such, if you did `func(theta)` now, this would equate to doing `costFunction(X, y, theta)` which completes our workaround for the Optimization Toolbox.

Now let's go through how we'd use the Optimization Toolbox to get the right parameters for univariate linear regression.

Setting up the cost function file

If you recall, the hypothesis we use for univariate linear regression is:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Given that there are `m` training examples where each is a tuple of $(x^{(i)}, y^{(i)})$ where $x^{(i)}$ and $y^{(i)}$ are the single input features and expected outputs for the i^{th} training example, the associated cost function is:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2$$

The derivative term of the cost function with respect to each parameter is also:

$$\frac{\partial J}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})$$
$$\frac{\partial J}{\partial \theta_1} = \frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})x^{(i)}$$

In class, the code to create the cost function file which outputs the cost to use a set of parameters that fit our input and expected output data `cost` as well as the derivative vector for each parameter `grad` is as follows:

```
function [cost, grad] = univariate_reg_cost(X, y, theta)
% Compute cost
m = numel(X);
cost = (0.5/m)*sum((theta(1) + theta(2)*X - y).^2);

% Compute derivative terms
grad = zeros(2,1); % Initialize derivative vector
grad(1) = (1/m)*sum(theta(1) + theta(2)*X - y);
grad(2) = (1/m)*sum((theta(1) + theta(2)*X - y).*X);
end
```

Figure 1: Cost function for univariate linear regression

`X` is a column vector of single features and `y` is the expected outputs. `theta` would be an input parameter column vector that is two elements long.

This file is available to you and is called `univariate_reg_cost.m` and can be found in the `demo` directory that is part of this lab.

The function prototype follows the behaviour that we are expecting. The behaviour of the function lays out what is to be expected when implementing a cost function for use in `fminunc`. First we compute the cost required to use the parameters `theta` for the training set for univariate linear regression. Next we initialize a `2 x 1` vector of zeroes then populate each entry with its respective derivative quantities. The first entry is for $\partial J/\partial\theta_0$ and the second entry is $\partial J/\partial\theta_1$.

Now that we have the cost function file set up, this is how we would go about using `fminunc` to find our parameters:

```
% Create data
X = [1;2;3];
y = [2;4;5];

% Create optimization options
options = optimset('GradObj', 'on', 'MaxIter', 100);

% Create an anonymous function
func = @(theta) univariate_reg_cost(X, y, theta);

init_theta = [0;0]; % Create initial parameters

% Find optimal parameters
[opt_theta, cost_val, exit_flag] = fminunc(func, init_theta, options);

fprintf('The final parameters are:\n');
disp(opt_theta);

fprintf('The final cost is: %f \n', cost_val);
fprintf('The exit flag is: %d \n', exit_flag);
```

Figure 2: Demonstration of MATLAB's Optimization Toolbox for univariate linear regression

This file is available to you and is called `univariate_reg_demo.m` and can be found in the `demo` directory that is part of this lab. This is a MATLAB script file as opposed to `univariate_reg_cost.m` which is a MATLAB function script file. You can simply run `univariate_reg_demo.m` as it without any inputs.

The code is quite self-explanatory. First define a bunch of single features and associated expected outputs (you'll recognize this example from the warmup section of the first lab). Next, we create options to be used with `fminunc` where we are manually going to specify what the gradient vector is as well as the maximum number of potential iterations to be 100. Next we create an anonymous function that is to be used for `fminunc` that takes advantage of using the inputs to compute the univariate linear regression cost function. After, we create a set of initial parameters, finally use `fminunc` to get the optimal parameters, the cost function value with these optimal parameters and the exit status flag. We finally show this to the user.

We get the following verbose output when running this code:

```
Local minimum found.
Optimization completed because the size of the gradient is less than
the default value of the function tolerance.
<stopping criteria details>

The final parameters are:
    0.6667
    1.5000

The final cost is: 0.027778
The exit flag is: 1
```

We can see that the final slope is `1.5`, the final intercept is `0.6667` and the cost associated with these parameters is `0.027778`. The exit flag is positive, meaning that the algorithm converged and didn't exceed the maximum number of iterations.

1 - Binary Classification

1.1 - Introduction

In this section, you will implement logistic regression for the purposes of binary classification. You will investigate the cases where the data are both linearly and non-linearly separable. With regards to non-linear separability, a standard practice is to introduce non-linear features into your data. Adding polynomial features and combinations of products of features for different polynomial powers is performed to find a decision boundary with good performance.

Before we do that, let's have a brief review on what logistic regression is. Recall that for logistic regression, the hypothesis uses the sigmoid function. The sigmoid function $g(z)$ is defined as the following:

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

The hypothesis that we will ultimately use for logistic regression is still a sum of weighted features. Given the parameter vector $\theta = (\theta_0, \theta_1, \dots, \theta_n)^T$ which is a `n x 1` column vector and an input set of features $x = (x_0, x_1, \dots, x_n)^T$, $x_0 = 1$, the hypothesis we will ultimately use for logistic regression is the dot product between θ and x and applying the sigmoid function to this result. The dot product can eloquently be described by a matrix multiplication as well. Concretely, this is equal to $\theta^T x$.

Therefore, the hypothesis $h_\theta(x)$ is:

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + \exp(-\theta^T x)}$$

$\exp(x) = e^x$ here and it is simply notation and to allow for easier writing. Consulting the class notes, given m training examples where each training example is a tuple of $(x^{(i)}, y^{(i)})$ as mentioned before, the cost function for determining the best decision boundary to separate our data is the following:

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

Remember that $\log(x)$ is $\ln(x)$, which is $\log_e(x)$. This convention is adopted because it will avoid confusion when implementing this as MATLAB uses the functoin `log` to represent $\log_e(x)$. The derivative of the cost function with respect to a parameter θ_j is defined as:

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

x_j^i is the j$^{\text{th}}$ feature for the i$^{\text{th}}$ training example. Recall that the hypothesis $h_\theta(x)$ is now non-linear and that the sigmoid function is applied to the weighted sum of features. Also recall that $x_0^{(i)} = 1$ for all training examples $i = 1, 2, \dots, m$.

From the lectures that the **decision boundary** that separates between the two classes is defined as:

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = 0$$

This denotes the point where choosing between the two classes is equiprobable and so this line of separation should naturally split between the two classes.

1.2 - Examining the Data

The data we will be using for this part of the lab is taken from Andrew Ng's online `coursera.org` course which contain training examples consisting of **two** features and an associated label - either positive `y = 1` or negative `y = 0`. Two features were chosen on purpose to allow easier visualization of the data as well as the decision boundary that separates between the two classes.

The file `lab2dataq1a.mat` is a MATLAB data file that contains training data that is linearly separable and the file `lab2dataq1b.mat` is a MATLAB data file that contains training data that is non-linearly separable. You can use `load lab2dataq1a.mat` or `load lab2dataq1b.mat` to load this data into your workspace. They both contain the following variables:

- `X` - A `N x 2` matrix where `N` is the total number of training examples and each column represents a different feature.
- `y` - A `N x 1` column vector which contains either 0 (negative class) or 1 (positive class).

It may be fruitful to actually look at the data visually and see what the decision boundary should more or less be before we get into implementing logistic regression. Below is code that loads in a MATLAB data file given for this section and plotting the data using different markers. For each training example, we can consider this to be a point in 2D space and a different marker can be used for distinguishing between different classes.

```
% 1. Load the data for the first part
load lab2dataq1a.mat;
% or
% load lab2dataq1b.mat;

% 2. Get the positions that have labels 0 and 1
ind_label0 = y == 0;
ind_label1 = y == 1;

xlabel0 = X(ind_label0,:);
xlabel1 = X(ind_label1,:);

% 3. Plot the points
plot(xlabel0(:,1), xlabel0(:,2), 'bo', xlabel1(:,1), xlabel1(:,2), ...
    'rx', 'MarkerSize', 12);
xlabel('x_1'); ylabel('x_2');
legend('Negative Class - y = 0', 'Positive Class - y = 1');
```

The first part loads in the data from file to the MATLAB workspace. The second part creates two binary / `logical` vectors. These are both `N x 1` column vectors. The first binary vector `ind_label0` denotes all training examples whose label belongs to `y = 0`; specifically `ind_label0(i)` is `true` if the i^{th} example belongs to label `y = 0` and `false` otherwise. Similarly, `ind_label1(i)` is `true` if the i^{th} example belongs to label `y = 1` and `false` otherwise. This is what is known as creating **a `logical` condition vector** and their usefulness will be apparant when we plot the points.

The last section of the code finally plots the points. We use a single `plot` command where we plot the points with the label `y = 0` are visualized as blue circles and the points with the label `y = 1` are visualized as red crosses. Doing `X(ind_label0,:)` **slices** into the data matrix and extracts only the row positions that correspond to where `ind_label0` is equal to `true` are what is returned. The end product would be selecting all of the training

examples that belong to the label `y = 0` and plotting those points with a blue circle. A similar observation can be made with the label `y = 1` and its associated binary vector `ind_label1`. We also ensure that the `MarkerSize` attribute is `12`, meaning that the marker is 12 pixels wide and high. We finally add in axes labels and give a legend to ensure no ambiguities in terms of what points belong to what class.

Linearly Separable Data

Using the code above, this is what the linearly separable data looks like referencing `lab2dataq1a.mat`:

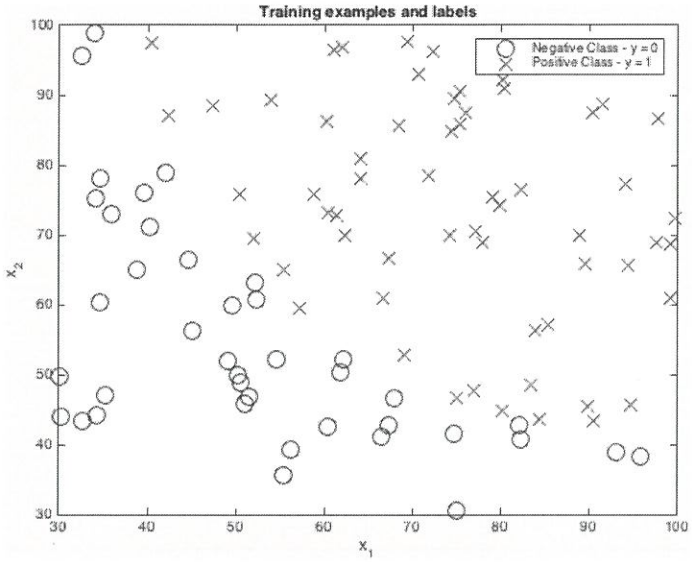


Figure 3: Linearly Separable Data

You can clearly see that the data is linearly separable by a decision boundary that is simply a straight line, though there are some potential mismatches.

Non-linearly Separable Data

Using the code above, this is what the non-linearly separable data looks like referencing `lab2dataq1b.mat`:

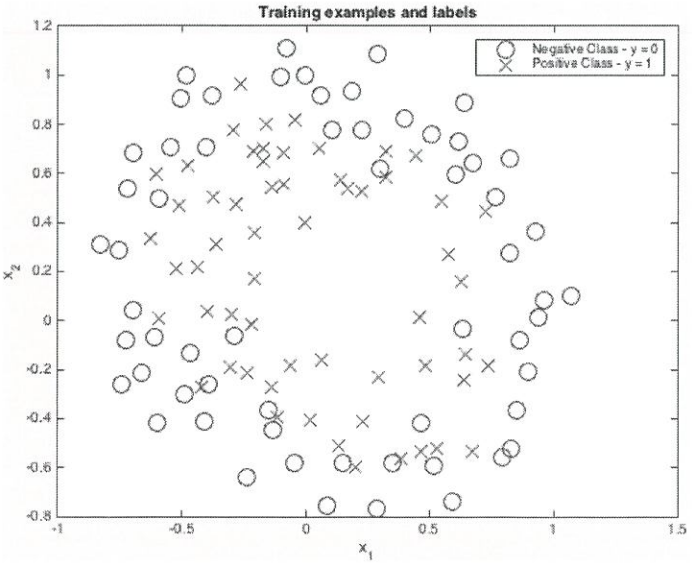


Figure 4: Non-linearly Separable Data

You can clearly see that the data is more or less separable by a decision boundary, but it will be highly non-linear and also there is a potential for some mismatches.

1.3 - Implement Logistic Regression without Regularization

Let's start by looking at the linearly separable data. As noted earlier, the decision boundary is simply a straight line and so we can get away with performing logistic regression without regularization. The features we will be using are simply the features themselves without introducing any polynomial terms for example. There is no possible way to perform any kind of "overfitting" if we are not introducing any of these non-linear terms as new features into our data. As such, your first task is to implement logistic regression without regularization.

Section 1.3.1 - Implement the sigmoid function

As the first step, you are given a skeleton file called `sigmoid.m` where the input is a matrix of any size and the output is a matrix that is the same size as the input where the sigmoid function is applied to every value seen in the input. This will help you implement logistic regression more easily. The function has the following input and output specifications.

Inputs

- `z` : A matrix or vector of any size that contains real numbers.

Outputs

- `g` : A matrix or vector that is the same size as the input where the sigmoid function is applied to every element in this input individually.

*Tip: Consider using **element-wise** operators to do this efficiently (i.e. `.*`, `./`, `.^` etc.) instead of using a `for` loop. Most of the time, MATLAB works best if you avoid using `for` loops.*

Section 1.3.2 - Implement the predictor

If you recall from the lectures, we classify an instance x to belong to the positive class if $h_\theta(x) \geq 0.5$. If $h_\theta(x) < 0.5$, this belongs to the negative class. In this next step, you are given a skeleton file called `binary_predictor.m` and your task is to modify this file so that it implements this decision logic for input instances given the parameters θ. The goal is to output a binary vector that is the same size as the total number of inputs where each element `i` of the output is `0` or `1` denoting that the input $x^{(i)}$ belongs to the negative or positive class respectively.

Inputs

- `X` : The data matrix of size `m x n` , where `m` is the number of examples and `n` is the number of features. Each row is an example and each column is a feature. If you are to compute the hypothesis values by a linear algebra approach (consult the class notes and the first lab), don't forget to prepend a column of ones to the matrix before this function is called.
- `theta` : The prediction model parameters as a `(n + 1) x 1` column vector. There is an addition of a term in the parameter vector due to the bias term θ_0.

Outputs

- `out` : An output binary vector of size `m x 1` where each element `out(i)` denotes the class that the i^{th} training example $x^{(i)}$ belongs to, so either the negative class `out(i) = 0` or the positive class `out(i) = 1`.

Section 1.3.3 - Back to Logistic Regression

Now you are to implement logistic regression without regularization. A skeleton file called `lr_cost_function.m` is available for you and you are to modify the file so that the function returns the cost associated with a given input vector of parameters `theta` as well as the derivative (gradient) vector of the parameters. As for which approach you wish to take, you may take the linear algebra approach or implement this using a nested `for` loops. The function should have the following input and output specifications:

Inputs

1. `X` : A data matrix where each column is a feature and each row is a training example. If you decide to pursue the linear algebra approach, you **must** ensure that the first column of this matrix will append a column of ones necessary for regression to work in the data matrix. This will be a `m x (n + 1)` matrix where `m` is the number of training examples and `n` is the number of features. You can leave this as `m x n` if you don't adopt the linear algebra approach. Assuming that you have already created the matrix `X`, this column of ones can be appended by doing:

   ```
   X = [ones(m,1) X];
   ```

   `ones(m,1)` creates a `m x 1` column vector of all ones.
2. `y` : A column vector of `m x 1` that denotes the expected class (so either 0 or 1) of each training example.
3. `theta` : An input set of parameters that is either `n + 1` or `(n + 1) x 1` column vector depending on whether you are using the linear algebra or nested `for` loop approach. It will be your responsibility to ensure that you use the right size depending on which approach you are to use.

Outputs

1. `cost_val` : The cost function evaluated at the given input set of parameters `theta` given the input data `X` and output labels `y`.
2. `grad` : The derivative (gradient) vector for each parameter evaluated at the input set of parameters `theta`. Concretely this is a `(n + 1) x 1` vector where each element is $\partial J/\partial \theta_j$ for $j = 0, 1, \ldots, m$.

Tip: Use the `sigmoid.m` function you created in Section 1.3.1 to give you an easier time creating the cost function to be placed in `lr_cost_function`.

Tip #2: If you implemented the first lab correctly, the only difference in implementation between multivariate linear regression and logistic regression is that you are applying the sigmoid function to the hypothesis before computing the update.

Tip #3: Using the linear algebra approach, the hypothesis for multivariate linear regression is simply $X\theta$ assuming that you have prepended a column of ones in the beginning of the matrix. This will produce a column vector of hypotheses for each training example. As such, you'd simply apply the sigmoid function to every element of this resulting vector and the rest of the logic remains the same.

Tip #4: If you decide to use the nested `for` loop approach, when computing the hypothesis for each training example when updating a particular parameter θ_j, simply apply the sigmoid function after computing the sum of weights: $\theta^T x^{(i)} = \sum_{j=1}^{m} \theta_j x_j^{(i)}$. Therefore, the new hypothesis is just $g(\theta^T x^{(i)})$.

1.4 - Create a Prediction Model using Linearly Separable Data

Now that you've implemented logistic regression without regularization, your task is to use the input data from `lab2dataq1a.mat` and combined with `fminunc` you are to compute the optimal parameters to find the correct decision boundary that separates between the two classes. The file available to you for completing this task is called `part1a.m` and is in the `tocomplete` directory of the lab. This file is partially complete. This includes the code that loads in the data and plots the data with their associated labels as well as code to show the decision boundary once you compute the optimal parameters. What is left for you is to compute the optimal parameters as well as the classification accuracy of your trained model so the code can successfully run.

Below is the code given in `part1a.m`:

```
% Clear all variables in the workspace and close all figures
clearvars;
close all;
addpath('../helper');
addpath('../data');

% Load the data for the first part
load('../data/lab2dataq1a.mat');

%%% Part 1 - Plotting the data
% Get the positions that have labels 0 and 1
ind_label0 = y == 0;
ind_label1 = y == 1;
```

```
xlabel0 = X(ind_label0,:);
xlabel1 = X(ind_label1,:);

% Plot the points
plot(xlabel0(:,1), xlabel0(:,2), 'bo', xlabel1(:,1), xlabel1(:,2), ...
    'rx', 'MarkerSize', 12);
xlabel('x_1'); ylabel('x_2');
legend('Negative Class - y = 0', 'Positive Class - y = 1');

%%% To be used for later - DON'T MODIFY
xmin = min(X(:,1));
xmax = max(X(:,1));

%%% Part 2 -  Get the logistic regression parameters
%%% FILL IN YOUR CODE HERE
%%% ENSURE THAT THE OUTPUT PARAMETERS ARE STORED IN A VARIABLE CALLED theta

%%%% Part 3 - Plot the line
% theta_0 + theta_1 x_1 + theta_2 x_2
% x_2 = -theta_0/theta_2 - theta_1/theta_2
xx = linspace(xmin, xmax);
yy = -theta(1)/theta(3) - theta(2)*xx/theta(3);
hold on;
plot(xx, yy, 'k');

%%% Part 4 - Calculate the classification accuracy
%%% PLACE CODE HERE
```

The first part clears all variables and closes all figures to ensure we don't get confused. We also add the helper and data directories as part of MATLAB's system path so you don't have to copy those files into the `tocomplete` directory to allow your work to be tidy. This is assuming that you are running the code in the `tocomplete` directory. You can simply call the functions and it will work. The rest of the code for the first part is essentially the same as what you saw in Section 1.2 of this lab. After the code for plotting the points, we find the smallest and largest feature value for the first feature x_1. Leave those variables in as the code later will reference them.

The next part (i.e. the second part) is up to you and you are to find the optimal parameters. However to ensure consistency when marking, **please use the maximum number of iterations to be 400**. The third part plots the decicion boundary **on top of the figure with the points and their corresponding classes** and the last part is where you are to fill in the code required to compute the classification accuracy.

> *Tip: Remember that because we are using a highly optimized there is* ***no need*** *for you to normalize the features. This makes finding the optimal parameters much more easier to find when using a high performance and efficient algorithm.*

Deliverables

Complete all of the functions that have been described previously. Next, complete the implementation of `part1a.m`. After, include what the final output parameters are as well as the plot of the points with the decision boundary. Also include the classification accuracy.

Here are some hints on how to get these quantities.

- Use the demonstration code for setting up the optimization options and using `fminunc`, find the optimal parameters and **ensure they are stored in an output variable called `theta`**. `theta` should be a `3 x 1` vector $\theta = (\theta_0, \theta_1, \theta_2)^T$ which represents the following decision boundary with two features:

  $$\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$$

  Therefore, rearranging for x_2, the equation of the line representing the decision boundary is:

  $$x_2 = -\frac{\theta_0}{\theta_2} - \frac{\theta_1}{\theta_2} x_1$$

  This is what is precisely being plotted in the third part of the code which plots the above decision boundary on top of the plot of features. We generate a span of linearly spaced points from the smallest to largest value in x_1 (remember those variables you're supposed to leave in?), compute the associated value for x_2 and plot these points on top of the points in the already opened figure driven by the first part of this code.

- The classification accuracy of a classifier is determined by taking all of the training examples you used in training and to predict their classes given the parameters θ after you find the prediction model. Once that is determined, the accuracy is found by calculating the fraction of labels the prediction model accurately predicted when comparing with the true labels.

  Recalling from class, given the true label of example i, $y^{(i)}$ and the corresponding predicted label $\tilde{y^{(i)}}$ using parameters θ, the classification accuracy in percentage is defined as:

  $$Accuracy = \frac{100}{m} \sum_{i=1}^{m} \mathbf{1}(y^{(i)} = \tilde{y^{(i)}})$$

  $\mathbf{1}(x)$ is what is known as an indicator function where:

  $$\mathbf{1}(x) = \begin{cases} 1, & \text{if } x = true \\ 0, & \text{if } x = false \end{cases}$$

- Comment on how well this linear decision boundary separates the data. Do you believe a more complicated decision boundary is required for better performance?

1.5 - Implement Logistic Regression With Regularization

We now turn to finding decision boundaries with that are highly non-linear and so a simple straight line is insufficient. The features we will be using are different combinations of powers of products between the features over various powers - more on this later. Because of the introduction of these features, the decision boundary will inevitably be subject to overfitting and so it is natural to introduce regularization when computing the decision boundary and ultimately the parameters that define this boundary. As such, your next task is to implement logistic regression with regularization.

The sigmoid function and predictor function from Sections 1.3.1 and 1.3.2 will be useful in this part of the lab.

1.5.1 - Review

To ensure we're on the same track, let's review what we covered in the lectures on this topic. The cost function $J(\theta)$ for logistic regression with regularization is the following:

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

All of the parameters remain the same as the variant without regularization except for the new parameter λ and is known as the regularization parameter. λ gives a sense of control with regards to preventing overfitting of the decision boundary. Take note that we don't perform regularization on the bias term θ_0 and hence the second summation starts from $j = 1$ and not $j = 0$.

The derivative (gradient) terms $\partial J/\partial\theta_j$ for each parameter is as follows:

$$\frac{\partial J}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)}$$

$$\frac{\partial J}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}\Big) + \frac{\lambda}{m}\theta_j, \text{ for } j = 1, 2, \ldots, m$$

Remember that $x_0^{(i)} = 1, \text{ for } i = 1, 2, \ldots, m.$

1.5.2 - Back to Logistic Regression

A skeleton file called `lr_cost_function_reg.m` is available for you and you are to modify the file so that the function returns the cost associated with a given input vector of parameters `theta` as well as the derivative (gradient) vector of the parameters. As for which approach you wish to take, you may take the linear algebra approach or implement this using a nested `for` loops. The function should have the following input and output specifications:

Inputs

1. `X` : A data matrix where each column is a feature and each row is a training example. If you decide to pursue the linear algebra approach, you **must** ensure that the first column of this matrix will append a column of ones necessary for regression to work in the data matrix. This will be a `m x (n + 1)` matrix where `m` is the number of training examples and `n` is the number of features. You can leave this as `m x n` if you don't adopt the linear algebra approach. Assuming that you have already created the matrix `X`, this column of ones can be appended by doing:

   ```
   X = [ones(m,1) X];
   ```

   `ones(m,1)` creates a `m x 1` column vector of all ones.
2. `y` : A column vector of `m x 1` that denotes the expected output of each training example.
3. `lambda` : The regularization parameter to control the reduction of overfitting for the decision boundary.
4. `theta` : An input set of parameters that is either `n + 1` or `(n + 1) x 1` column vector depending on whether you are using the linear algebra or nested `for` loop approach. It will be your responsibility to ensure that you use the right size depending on which approach you are to use.

Outputs

1. `cost_val` : The cost function evaluated at the given input set of parameters `theta` given the input data `X` and output labels `y`.

2. `grad` : The derivative (gradient) vector for each parameter evaluated at the input set of parameters `theta` . Concretely this is a `(n + 1) x 1` vector where each element is $\partial J / \partial \theta_j$ for $j = 0, 1, \ldots, m$.

1.6 - Create a Prediction Model using Non-Linearly Separable Data

Building on Section 1.5, your task is to use the input data from `lab2dataq1b.mat` and combined with `fminunc` you are to compute the optimal parameters to find the correct decision boundary that separates the two classes. In addition, you are to compute the classification accuracy of your trained model.

1.6.1 - Create non-linear features

Because the data is highly non-linear in nature, the decision boundary will also be non-linear. Simply just using the features themselves are not enough. Before we get into the task of creating this non-linear decision boundary, we will need to develop a mechanism that will introduce non-linear features to your training examples. Given data that uses two features, one common method is to create additional polynomial and products of features taken at different powers.

Concretely, given an example $x = (x_1, x_2)$ that consists of features x_1 and x_2, a new training example x_p is created that consists of the following features:

$$x_p = (1, x_1, x_2, x_1 x_2, x_1^2, x_2^2, \ldots, x_1^i x_2^j, \ldots x_1^n x_2^n)^T, \text{ for } i = 0, 1, \ldots, n, \text{ for } j = 0, 1, \ldots, n$$

Therefore, each element in x_p is generated by taking a unique pair of i and j for $i = 0, 1, \ldots, n$ and $j = 0, 1, \ldots, n$, taking x_1^i and x_2^j and multiplying the two values together. There are $n + 1$ possible values for i and j which in turn produces the vector x_p that is $(n + 1) \times (n + 1)$ large. n is the largest degree we will consider. ***For this lab, we will be using $n = 6$ to model the highly non-linear decision boundary.***

Thankfully you don't have to write this code and it has been made available to you under the `helper` directory for ease in running the code. This function is called `create_polynomial_features` and you are more than welcome to examine the code so you can see for yourself how it works. The understanding of how the function is coded is not essential to completing the lab, but understanding what the expected inputs and outputs are is and they are the following.

Inputs

- `X1` : A `m x 1` column vector of features where each feature belongs to x_1.
- `X2` : A `m x 1` column vector of features where each feature belongs to x_2.
- `n` : The largest degree we are considering.

Outputs

- `X` : A `m x [(n x 1) x (n x 1)]` matrix where each row of this matrix creates the vector x_p from products of powers of x_1 and x_2 as stated above. Take note that this matrix will prepend a column of ones to this matrix so this is very nicely formatted for the linear algebra approach of logistic regression.

1.6.2 - Back to Logistic Regression

Now that you've implemented logistic regression with regularization, your task is to use the input data from `lab2dataq1b.mat` and combined with `fminunc` you are to compute the optimal parameters to find the correct decision boundary that separates between the two classes. The file available to you for completing this task is called `part1b.m` and is found in the `tocomplete` directory of the lab. The file is partially complete. This includes the code that loads in the data and plots the data with their associated labels as well as code to show the decision

boundary once you compute the optimal parameters. What is left for you is to compute the optimal parameters and to determine the classification accuracy of your trained model so the code can successfully run.

Below is the code given in `part1b.m`:

```
% Clear all variables in the workspace and close all figures
clearvars;
close all;

% Load the data for the first part
load('../data/lab2dataq1b.mat');

%%% Part 1 - Plotting the data
% Get the positions that have labels 0 and 1
ind_label0 = y == 0;
ind_label1 = y == 1;

xlabel0 = X(ind_label0,:);
xlabel1 = X(ind_label1,:);

% Plot the points
plot(xlabel0(:,1), xlabel0(:,2), 'bo', xlabel1(:,1), xlabel1(:,2), ...
    'rx', 'MarkerSize', 12);
xlabel('x_1'); ylabel('x_2');
legend('Negative Class - y = 0', 'Positive Class - y = 1');

%%% To be used for later - DON'T MODIFY
xmin = min(X(:,1));
xmax = max(X(:,1));
ymin = min(X(:,2));
ymax = max(X(:,2));

%%% Part 2 -  Get the logistic regression parameters
% Define regularization parameter
lambda = ...; % Fill in here yourself

% Introduce polynomial features
degree = 6;
%%% PLACE CODE FOR CREATING POLYNOMIAL FEATURES HERE

%%% FILL IN YOUR CODE HERE TO FIND PARAMETERS
%%% ENSURE THAT THE OUTPUT PARAMETERS ARE STORED IN A VARIABLE CALLED theta

%%%% Part 3 - Plot decision boundary
[xx,yy] = meshgrid(linspace(xmin, xmax), linspace(ymin, ymax));
XX = create_polynomial_features(xx(:), yy(:), degree);
zz = reshape(XX*theta, size(xx));
hold on;
contour(xx, yy, zz, [0, 0], 'LineWidth', 2);
title(['Training examples and labels - \lambda = ' num2str(lambda)]);

%%% Part 4 - Calculate the classification accuracy
%%% PLACE CODE HERE
```

Part 1 is exactly the same as in `part1a.m`. We also extract the minimum and maximum feature values for x_1 and

x_2 so that we can draw the decision boundary once we find the optimal parameters. This defines a span of values for computing the decision boundary of the input data.

The next part will be your task in filling out what is required to find the right parameters for the decision boundary separating the two classes. The regularization parameter `lambda` as well as the largest degree of polynomial we are considering when adding additional non-linear features `degree` is defined. **However to ensure consistency when marking, please use the maximum number of iterations to be 400**.

The last part finally plots the decision boundary on top of the figure that contains the points separated into their classes. This situation is a bit more complex given the non-linear features that were introduced and it isn't simply a straight line as we have seen with the linearly separable case. This part has been completed for you so you can immediately assess whether your decision boundary is correct. Finally, you are to calculate the classification accuracy after you find the parameters for the decision boundary.

Optional - How is the decision boundary drawn?

This section discusses how the decision boundary is drawn for the case of non-linearly separable data. If you don't care about this you are more than welcome to skip this paragraph. Recall that the decision boundary is equal to $\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n = 0$. We define a 2D grid of values that span the minimum and maximum values of x_1 and x_2 using the `meshgrid` command then take these values and substitute them into the decision boundary equation. We use `create_polynomial_features` to help facilitate this for us. Because `create_polynomial_features` expects two inputs that are vectors, the outputs from `meshgrid` are converted into vectors and then reshaped back into a 2D grid using `reshape` once the decision boundary equation is computed for these points. We finally use `contour` and specify only the height at `z = 0` corresponding to the condition that satisfies the decision boundary which thus plots it.

Deliverables

- Complete all of the functions that have been described previously. Next, complete the implementation of `part1b.m`.
- Set `lambda = 0` (i.e. no regularization) and use the `lr_cost_function_reg` function that you created in conjunction with `fminunc` to find the parameters for the decision boundary separating the two classes. Include what the final output parameters are as well as the plot of the points with the decision boundary. Also include the classification accuracy. Comment on the classification accuracy and compare this with the decision boundary. Do you suspect underfitting, overfitting or is this decision boundary "just right"?
- Repeat the previous task but setting `lambda = 1`. Once you change the regularization parameter, do you suspect underfitting, overfitting or is this "just right"?
- Repeat the previous task but setting `lambda = 100`. Once you change the regularization parameter, do you suspect underfitting, overfitting or is this "just right"?
- Comment on how well the decision boundary separates the two classes for all of $\lambda = 0, 1, 100$. Do you believe that a simple linear line is suitable to separate the two classes?
- Comment on the classification accuracy taking the regularization parameter into account. Specifically, does the classification accuracy increase or decrease as the regularization parameter increases? Is the classification accuracy alone a good indication as to how suitable the decision boundary is for this data?

Tip: Because we are using a degree of 6 for the polynomial features, this will produce a `m x 49` data matrix because with `n = 6`, there will be `(n x 1) x (n x 1) = 49` possible features including the column of ones that is seen in the first column. This will help if you decide to use the linear algebra approach.

Tip #2: Because there are 49 parameters for each value of λ, it may be fruitful to place these in a separate table for ease of presentation.

2 - Multi-Class Classification

2.1 - Introduction

Now that you've implemented logistic regression both with and without regularization and have used it on some synthetic data, it's now time to apply this on a real dataset. The task of this section of the lab is to use logistic regression in a multi-class or One-Vs-All setting to classify hand-written digitized images. Specifically, you will train a multi-class classification system and given a new image that the system hasn't seen before, your task will be to determine what digit this image corresponds to.

Interesting Fact: *One of the most prominent applications for digit recognition is automatically determining the correct postal codes on letters (i.e. snail mail) without human intervention to increase the throughput of letters being processed per day. In fact, the approach we will be taking in this part of the lab was a preliminary approach to the way the United States Postal Service (USPS) automatically recognized US Zip Codes. If time permits for you, see an interesting discussion about the topic here: http://www.livescience.com/32290-how-do-post-office-machines-read-addresses.html.*

The dataset to be used is from the Mixed National Institute of Standards and Technology (MNIST) and was curated by Yann LeCun, a leading machine learning researcher from New York University who is now director of AI Research at Facebook.

The link to this dataset can be found at the beginning under the **List of Files Included** section where the Dataset Files section is. In this dataset, there are handwritten digits created by several people that have been digitized into images. Each image only contains **one** digit from 0 to 9 and there is an associated true label with each digit. Your instructor has post-processed the raw data provided in the dataset and has created separate MATLAB compatible matrices that can simply be loaded into MATLAB. This dataset is stored in the `data` directory and is called `lab2digits.mat`.

For our purposes, we will be using the 10000 image dataset where the dataset is split up into 80% training and 20% testing. Specifically, your instructor randomly chose 8000 images and their associated labels to serve as the training dataset and the remaining 2000 images and their asociated labels will serve as the test dataset. Each image is 28 x 28 pixels large and each pixel value in an image ranges from 0 to 1 where 0 is a black pixel and 1 is a white pixel and therefore each image is grayscale. When you load in the dataset into MATLAB using `load lab2digits.mat`, you will see four separate variables loaded into your workspace:

- `trainImages` : This is a `28 x 28 x 8000` 3D matrix where each 2D slice is an image of a digit. Therefore, `trainImages(:,:,i)` corresponds to the i^{th} image in the training set. This is the data you will be using for training your multi-class classification system.

- `trainLabels` : This is a `8000 x 1` vector that denotes what the digit is supposed to be for the corresponding image in `trainImages` . Specifically, `trainLabels(i)` gives you what the true label of the digit is supposed to be that corresponds to the image `trainImages(:,:,i)` . This is the data you will be using for training your multi-class classification system.
- `testImages` : This is a `28 x 28 x 2000` 3D matrix where each 2D slice is an image of a digit. Therefore, `testImages(:,:,i)` corresponds to the i^{th} image in the test set. This is the data you will be using for assessing how accurate your multi-class classification system is on new inputs the classification system hasn't seen before.
- `trainLabels` : This is a `2000 x 1` vector that denotes what the digit is supposed to be for the corresponding image in `testImages` . Specifically, `testLabels(i)` gives you what the true label of the digit is supposed to be that corresponds to `testImages(:,:,i)` . This is the data you will be using for assessing how accurate your multi-class classification system is on new inputs the classification system hasn't seen before.

2.2 - Examining the Data

To have a sense of what the images look like, the figure below shows 144 randomly selected images from the 10000 image dataset. This was created using the `show_digits` function that is included in the `demo` directory of this lab. Explaining how this script works internally is beyond the scope of what we're talking here. If you're curious you can take a look at the function yourself as it is well documented. However, all you need to know is how to use the function. It takes in a 3D matrix of images (so either `trainImages` or `testImages`) and the total number of digits you want to show `n` . The output is a MATLAB figure that shows those number of digits directly in the figure.

To reproduce the figure shown below, the training set images were used and 144 of them were selected. Simply do `show_digits(trainImages, 144);` in MATLAB once you load in the data. However, because of the random selection of the digits, you will not get the same digits every time you run this function, but the total number of digits will remain the same. Therefore the image below is one such possibility.

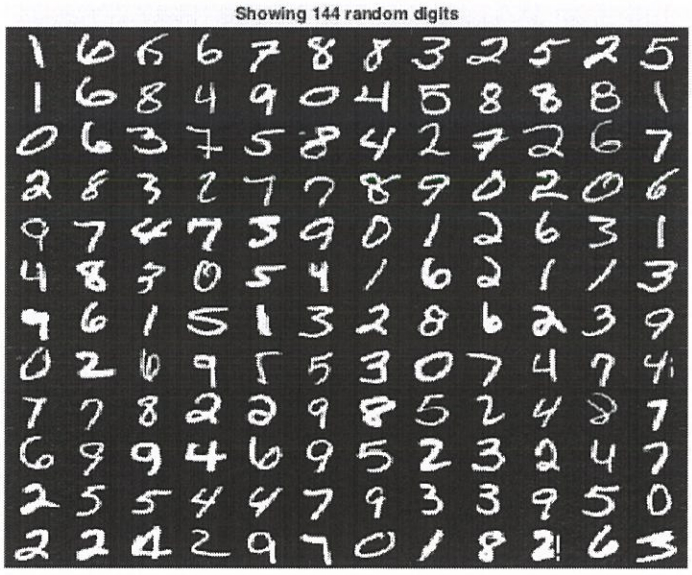


Figure 5: 144 randomly selected digits from the training set

It may also be fruitful to show several of each digit separately. The function `show_random_digits` will show you 100 randomly selected digits for each class (i.e. 0 to 9) and it will pause at each digit. Simply push any key or click your mouse to advance to the next class. The inputs into `show_random_digits` is a 3D matrix of images (so either `trainImages` or `testimages`) and their associated true labels (`trainLabels` or `testLabels`). A couple of examples of the output that `show_random_digits` produces are shown below. This was done using the training set and so doing `show_random_digits(trainImages, trainLabels);` gives you this output.

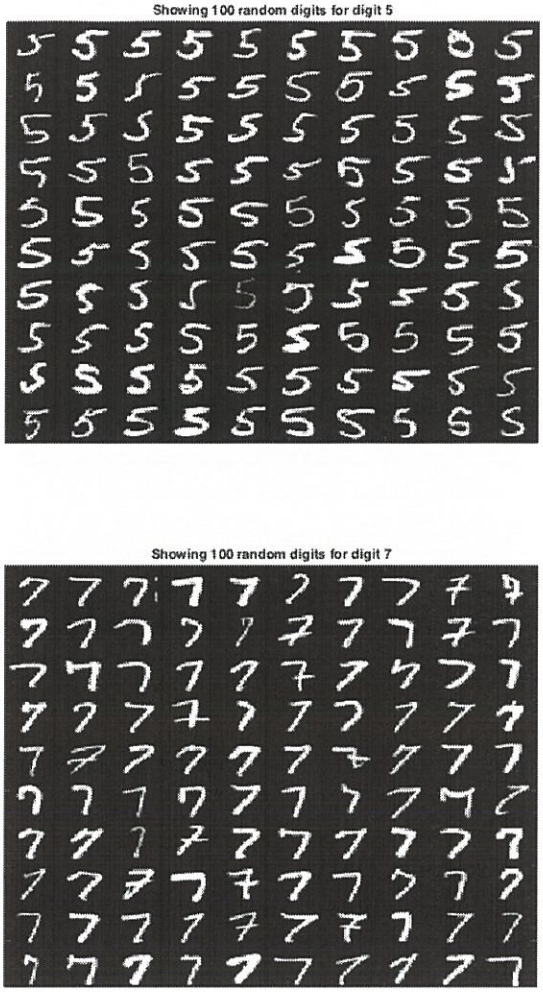


Figure 6: 100 randomly selected digits for digits 5 and 7

2.2 - Implement One-Vs-All Classification

Now that the preliminaries are out of the way, your task now is to implement One-Vs-All Classification. Because we are now dealing with images, it isn't quite clear on what we need to choose as features. Should we represent the entire 28 x 28 image as a single feature? What about a set of features like the mean intensity or the variance? We're only given pixel data for each digit in our datasets.

However, one common method is to represent each **pixel as a feature** in each digit image. This is the approach we will take for this part. Specifically, if each digit is an image that is 28 x 28 large, we can represent each image as an instance in our training sets by stacking all of the rows of pixels together to create one long 784 (`28 x 28 = 784`) element row vector or stacking all of the columns of pixels together to create one long 784 element column vector then transpose this so it becomes a row vector. Therefore, we would transform our training and test datasets into a `8000 x 784` and `2000 x 784` 2D matrices respectively so that each **row** represents a single image that underwent the stacking that we talked about earlier. If you decide to use the linear algebra approach, don't forget to append a column of ones at the beginning of these matrices so their sizes are now `8000 x 785` and `2000 x 785` respectively.

It's important to note that our classes are within the range of $[0, 9]$. In class, we assumed that the labels start at 1 but for our case of digit recognition, 0 is a valid class. For the purposes of this lab, we will assume that the classes that each input belongs can belong to the range of $[0, N-1]$ where N is the total number of classes possible. So in our case, $N = 10$ corresponding to 10 digits.

2.2.1 - Important Note - use `fmincg`

For this part of the lab, **remember to use `fmincg` instead of `fminunc`**. `fmincg` will be faster at converging for a larger amount of features. Here we will be using 785 features (784 + 1 for the bias term) and so `fmincg` is more suited here. You're more than welcome to try using `fminunc` but you'll notice that the algorithm for training will be severely slow to the point where the algorithm does not converge to find the optimal parameters. As such, your instructor implores you to use `fmincg` instead.

2.2.2 - Implement One-Vs-All Training

Before we can continue on with this topic, it's important that you implement the training system for multi-class classification using the One-Vs-All strategy. As discussed in class (and earlier), you find parameters by logistic regression for each class invidually. These parameters are found by making the labels for a particular class positive while the rest of the labels are negative and you repeat this for every class $0, 1, \ldots, N-1$ that you have in your dataset. You will then have a set of N parameters once you complete this approach. Your task in this part is to implement this behaviour in the file `one_vs_all.m` that is found in the `tocomplete` directory of the lab. The goal of this function is given a data matrix of training examples $\$X$ and an associated column vector y that tells you the expected class of each input example, we create a **matrix** of parameters Θ where each **column** Θ_i is a column vector that represents the parameters θ learned for ith class, the output of the function is a vector where each element describes which class a particular instance belongs to. Concretely, if there are N total possible classes seen in the dataset, the matrix Θ is arranged in the following way:

$$\Theta = \begin{bmatrix} \Theta_1 & \Theta_2 & \cdots & \cdots & \Theta_N \end{bmatrix}$$

$$\Theta = \begin{bmatrix} \theta_{00} & \theta_{10} & \theta_{20} & \cdots & \theta_{(N-1)0} & \theta_{N0} \\ \theta_{01} & \theta_{11} & \theta_{21} & \cdots & \theta_{(N-1)1} & \theta_{N1} \\ & & \cdots & \cdots & & \\ & & \cdots & \cdots & & \\ \theta_{0n} & \theta_{1n} & \theta_{2n} & \cdots & \theta_{(N-1)n} & \theta_{Nn} \end{bmatrix}$$

Each column i of this matrix has parameters $\theta = (\theta_0, \theta_1, \ldots, \theta_n)^T$ where these denote the binary classifier parameters learned by setting the labels for ith class. Specifically, the labels for the examples that belong to class i are positive and the rest are negative. As such, the notation of the above matrix has elements θ_{ij} where i denotes the ith class and j denote the jth parameter of the binary classifier learned for the ith class. Therefore, $\Theta_i = (\theta_{i0}, \theta_{i1}, \ldots, \theta_{in})^T$ are the parameters for the class i.

Going with this, your task is to implement this behaviour in the MATLAB function `one_vs_all` where the expected inputs and outputs are the following:

Inputs

1. `X` : A data matrix where each column is a feature and each row is a training example. If you decide to pursue the linear algebra approach, you **must** ensure that the first column of this matrix will append a column of ones necessary for regression to work in the data matrix. This will be a `m x (n + 1)` matrix where `m` is the number of training examples and `n` is the number of features. You can leave this as `m x n` if you don't adopt the linear algebra approach. Assuming that you have already created the matrix `X`, this column of ones can be appended by doing:

```
X = [ones(m,1) X];
```

`ones(m,1)` creates a `m x 1` column vector of all ones.

2. `y` : A column vector of `m x 1` that denotes the expected class of each training example. The dynamic range of `y` is expected to lie in the range $[0, N-1]$ where n is the total number of possible classes.
3. `num_classes` : The total number of classes we are to expect in our data. For the digit recognition task, this is equal to 10.
4. `lambda` : The regularization parameter to prevent overfitting for logistic regression.

Outputs

1. `Theta` : A `(n + 1) x N` matrix where each column `i` describes the parameters θ where class `i` was assigned the positive class and the other classes are negative. Remember that for the purposes of digit recognition, the classes are enumerated from 0 to 9 and since MATLAB starts indexing at 1, `Theta(:,1)` are the parameters for class 0 and `Theta(:,10)` are the parmeters for class 9.

Important Note

To ensure the results are achieved in a relatively short amount of time, we will be setting the ***maximum number of iterations to be 100***. Keep this in mind when creating the options structure for use with `fmincg` when implementing this function.

Tip: The easiest way to do this would be to use a `for` loop for each class `i` you want to find the parameters for, make the right setup with `optimset` and your initial parameters and call `fmincg` to return a column vector of parameters. You'd then place this column vector in the right column of the output `Theta`.

---

*Tip #2: Remember that for finding the parameters for class `i`, the mechanics for logistic regression remain the same. The **only** thing that changes is the expected labels. The expected labels `y` for logistic regression when finding the parameters for class `i` can be found by creating a `logical` condition vector `y` where each element `y(j)` is 1 if the j^{th} training exampling belongs to class `i` and `y(j)` is 0 if it doesn't. You'd then use exactly the same data matrix but a different expected labels vector `y` to compute the parameters. You'd then place these vectors in the right column of the output matrix `Theta`.*

Tip #3: You are essentially finding the parameters for 10 classifiers on an 8000 sample dataset. This

*will **most definitely** take some time to compute all of the parameters. Give allowance to MATLAB to allow for the computation to complete. As an example, your instructor is using a Late 2013 model MacBook Pro with 16 GB of DDR3 1.6 GHz RAM and a 2.3 GHz Intel Core i7 processor. Using the linear algebra approach for specifying the cost function values and gradient vector in `lr_cost_function_reg` for use in finding the parameters for each class took approximately 70 seconds. As such, give MATLAB a few minutes to compute the parameters.*

---

*Tip #4: Because finding the matrix of parameters does tend to take a long time, it is highly advisable that you **save** this matrix somewhere to disk so that if you need the parameters for a later time, you're not re-running the program and you just simply load the parameters from disk. Take a look at the `save` command in MATLAB to help you facilitate this.*

2.2.3 - Implement the Multi-Class Prediction System

Now that you've completed the code that creates a matrix of parameters where each column is designed to detect a particular class, your task now is given a data matrix of **new** instances that the training step did not see as well as this matrix of parameters, we now want to determine which class a particular input instance from this new data matrix belongs to.

The reason why this matrix has special exaggeration is because you can very eloquently find the best class each instance belongs to in a linear algebra approach. If you examine pages 15 and 16 of the Lecture 2 notes, you can create a **prediction** matrix by performing matrix multiplication of the data matrix and parameter matrix Θ where each column i of the output gives you the predictions for the classifier i. To facilitate this for logistic regression, you simply have to use the sigmoid function and apply it to every single element in this matrix. Therefore, each **row** j of this output matrix would be the predictions of the example j for each of the classifiers. You'd then choose whichever class gave the highest response.

You certainly don't have to do this via a linear algebra approach but it would make things easier. Your task is to implement this behaviour in the MATLAB file called `multiclass_predict.m` that is included in the `tocomplete` directory that is part of this lab. This file has the following input and output parameters:

Inputs

1. `X` : A data matrix where each column is a feature and each row is an example. If you decide to pursue the lastinear algebra approach, you **must** ensure that the first column of this matrix will append a column of ones necessary for regression to work in the data matrix. This will be a `m x (n + 1)` matrix where `m` is the number of training examples and `n` is the number of features. You can leave this as `m x n` if you don't adopt the linear algebra approach. Assuming that you have already created the matrix `X` , this column of ones can be appended by doing:

   ```
   X = [ones(m,1) X];
   ```

   `ones(m,1)` creates a `m x 1` column vector of all ones.

2. `Theta` : The matrix of parameters Θ discussed earlier where there are `n + 1` rows and `N` columns. `N` is

the total number of classes you would like to consider and `n` is the total number of features. The addition of 1 is due to the bias term.

Outputs

1. `classes` : A `m + 1` column vector where each element `classes(i)` describes which class the i^{th} example from the i^{th} row of the input `X` belongs to. Remember that for the purposes of digit recognition, the classes are enumerated from 0 to 9 and so the expected outputs seen in this vector should range from 0 to 9.

Tip: Have a look at the `max` command in MATLAB. If you use this command right, you can efficiently do One-Vs-All prediction in one line of code (tentatively... see the next tip.)

---

*Tip #2: If you end up using the `max` command, the **second** output argument of `max` may prove very useful. Take note that you'll have to **subtract the result by 1** to ensure that your output class predictions range from 0 to 9.*

2.2.4 - Create the Digit Recognition Pipeline

Now that we finally have that all set up, our task now is to create the digit recognition pipeline. This consists of the following steps:

1. Load in the digit data.
2. Reshape the 3D digit matrix of the training set so that each row is a training example.
3. Find the matrix of parameters Θ for the classes $0, 1, \ldots, 9$ using a One-Vs-All approach. To prevent overfitting, set the regularization parameter $\lambda = 1$.
4. Using Θ found in step #3 and the training data reshaped in step #2, determine what the digits are recognized as for each digit in the training data set.
5. Reshape the 3D digit matrix of the test set so that each row is a training example.
6. Using Θ found in step #2 and the test data reshaped in step #4, determine what the digits are recognized as for each digit in the test data set.
7. Compute the classification accuracy for both the training data and the test data.
8. Show examples of misclassified digits. For each digit, you are to show 9 misclassified digits. If there are less than 9 misclassified digits, only show up to as many digits that were misclassified instead of the 9.

You are to implement this pipeline in the file `part2.m` found in the `tocomplete` directory of the lab. In this file, steps #1, #2 and #5 have been completed for you. Your task is to complete steps #3, #4, #6, #7 and #8. This skeleton file is shown below:

```
% Clear all variables, close all figures and add the helper directory to
% MATLAB's system path
clearvars;
close all
addpath('../helper');

%%% Part 1
% Load in digits
load('../data/lab2digits.mat');

%%% Part 2
% Reshape the 3D matrix of training digits so that each row is a training
% examples
[rows,cols,numImages] = size(trainImages);
Xtrain = reshape(trainImages, rows*cols, numImages).';
Xtrain = [ones(numImages,1) Xtrain];

%%% Part 3
% Perform training for each digit
lambda = 1; % Define regularization parameter
% FILL IN YOUR CODE HERE

%%% Part 4
% Find predicted labels for training set
% FILL IN YOUR CODE HERE

%%% Part 5
% Reshape the 3D matrix of test digits so that each row is a test
% example
numTestImages = size(testImages, 3);
Xtest = reshape(testImages, rows*cols, numTestImages).';
Xtest = [ones(numTestImages,1) Xtest];

%%% Part 6
% Find predicted labels for the test set
% FILL IN YOUR CODE HERE

%%% Part 7
% Find the classification accuracy for the training and test data set
% FILL IN YOUR CODE HERE

%%% Part 8
% Show some misclassified images - 9 for each digit
% FILL IN YOUR CODE HERE
```

The beginning of this code clears all of the variables in the workspace so we start clean, closes all of the figures that may be open and adds the `helper` directory to MATLAB's system path so that you don't have to unnecessarily copy the files in the `helper` directory to where you are completing the lab work. Part 1 loads in the digits into MATLAB's workspace as we expect. Part 2 **reshapes** the 3D matrix of training digits so that each row is represented as a single digit. The function `reshape` reshapes a matrix so that it conforms to specified dimensions. The way `reshape` works is that in a **column-major** order. This means that columns of a matrix are used to create the output matrix of the desired dimensions. The initial output of `reshape` is that each **column** represents a single image where each row is a feature. We need to transpose this matrix so that each row is a sample and each column is a feature which is why the transposition operator is applied after `reshape` is called. We finally add a

column of ones to complete the data reshaping. Part 5 performs the same style of reshaping as in Part 2 but we are doing this for the test data as opposed to the training data.

Tip: When your instructor implemented this, the classification accuracies for the training and test data are above 90% accurate. If you have anything less than this, there is something wrong in your implementation.

2.2.5 - Showing Misclassified Digits - Part 8

With the current system, it is inevitable that there will be some misclassifications. Due to the highly non-predictable behaviour of individuals writing digits (i.e. people write digits in different ways), there will most certainly be digits that are misclassified. When evaluating the accuracy of your digit recognition system, it may also be fruitful to show examples of where your system misclassified digits. There is another function available to you for use called `show_misclassified_digits.m` that is in the `helper` directory of the lab where its purpose is to show you what instances your system misclassified for a particular digit. The function takes in a 3D digit matrix, a column vector consisting of how your system **predicted** each digit and a column vector of the **true labels** for each digit. Combined with the maximum number of digits you want to show and the desired digit you want to display, the function creates a figure of digits that the system misclassified.

The way you would run the function is in the following fashion:

```
show_misclassified_digits(X, ypredict, ytrue, digit, K);
```

The list of input parameters are:

- `X` is a 3D matrix of digits (so either `trainImages` or `testImages`)
- `ypredict` is a column vector that matches the total number of digits seen in `X` which is the **output of your multi-class prediction scheme** (i.e. the output of `multiclass_predict`),
- `ytrue` is a column vector is the same size as `ypredict` but it contains the **true class for each digit** (i.e. `trainLabels` or `testLabels`)
- `digit`: Is the desired digit we want to examine misclassifications for (i.e. `digit` is between 0 and 9)
- `K`: The maximum number of digits you want to display. This will be 9 for our purposes.

The function returns no outputs but it will spawn a new `figure` every time this is called. This figure contains the digits that were misclassified for a particular digit of interest and for each digit, a title is shown that tells you what the system predicted the digit to be. If you want to show misclassified digits for every class, this can very easily be put in a `for` loop. An example of this figure is shown below for illustrative purposes. This is an example of what happens if `K = 9` when examining the misclassifications for digit 0.

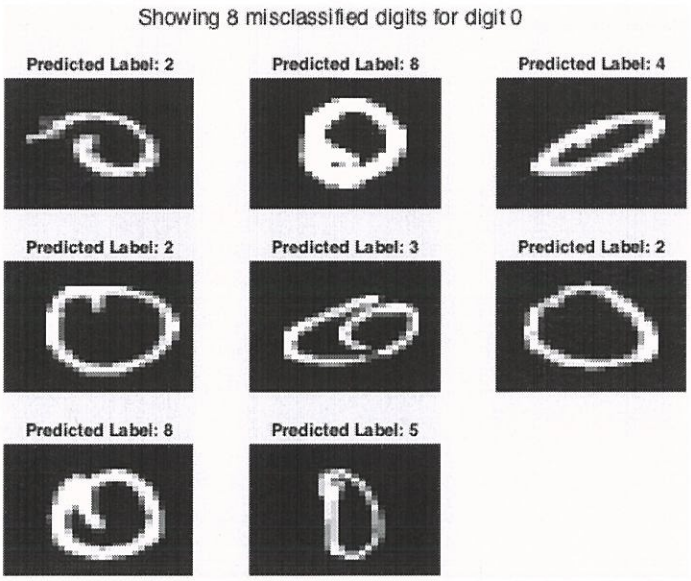


Figure 7: Showing 8 misclassified digits from class 0

The top of each image shows you what the system misclassified the digit as for the class specified in `digit`. For each image underneath each title, you see the digit for the class `digit` itself. Take note that even though `K = 9`, the figure tells us that only 8 images of class / digit 0 were misclassified and these were the digits from class 0 that led to the misclassification given our trained parameters. As such, this function automatically adjusts itself so that it only shows up to as many misclassifications there are in total for a digit if this does not equal `K`.

2.2.6 - Deliverables

- Complete Parts 3, 4, 6, 7 and 8 in `part2.m`. Use all of the functions that were completed by you up to this point in the lab to faciltate the completion of each step. Pay special attention to each step as it will give you insight on what functions you are to use to complete these steps. For part 8, you can get away with using a `for` loop and looping over values from 0 to 9 when setting the `digit` variable. Ensure that you use `K = 9`.
- Because of the larger number of features used in this part of the lab, **you are not required to report the parameters stored in the matrix Θ**. What you simply need to include are the accuracies for both the training data and test data as well 10 figures that illustrate the digits that were misclassified (i.e. from Part 8).
- Examining the misclassified digit figures that are produced from Part 8, comment on a few of the digits that were misclassified and explain why you believe misclassifications took place.
- Speculate what would happen if you changed the regularization parameter when performing the One-Vs-All training of our parameters. Specifically, what would happen if we implemented no regularization $\lambda = 0$? What would happen if we had high regularization (i.e. $\lambda = 10000$)? Please note that there is no need to retrain your classification models to obtain the parameters under new regularization parameters. However, if you find it a fruitful exercise, you're more than welcome to do so.
- Do you believe that classification accuracy is an acceptable measure of performance? Why or why not?

3 - Bayesian Decision Theory

3.1 - Introduction

In this last section, we will investigate how we can use Bayesian Decision Theory for the binary classification case as an alternative method for performing classification. Bayesian Decision Theory is a fundamental statistical approach to classification where the decision problem (i.e. deciding on what class an input example belongs to) is posed in probabilistic terms. For this lab, the following assumptions are made:

- We will concentrate on only the binary classification problem.
- We are assuming that all of the relevant probability values are known.
- We will only concentrate on using a single feature for analysis.
- We are also assuming that the features analyzed are **Gaussian** distributed.

The main driving engine is Bayes Rule. Given two classes in our training data $\omega_j, j = 1, 2$, Bayes Rule states that the posterior probability, or the probability that a particular input is assigned to a class ω_j can be found by the examining the input's features x, the prior probability of the class j, $P(\omega_j)$ and the class-conditional probability or likelihood of the input's features x given the class j, ω_j.

Concretely, Bayes Rule is defined as:

$$P(\omega_j|x) = \frac{P(x|\omega_j)P(\omega_j)}{P(x)}$$

Each of the terms above are defined below:

- $P(\omega_j|x)$ is the posterior probability or the probability that an input defined by a set of features belongs to class ω_j.
- $P(x|\omega_j)$ is the likelihood or class-conditional probability for the class ω_j. This is computed by examining the features for the training set for each class ω_1 and ω_2 and determining the mean μ_j and variance σ_j^2 for $j = 1, 2$. The class-conditional probability is thus defined as a Gaussian Probability Distribution Function (PDF) of the following form:

$$P(x|\omega_j) = \frac{1}{\sqrt{2\pi\sigma_j^2}} \exp\left(-\frac{(x - \mu_j)^2}{2\sigma_j^2} \right)$$

- $P(\omega_j)$ is the prior probability for class ω_j. This can be found by finding the fraction of instances in your training set that belong to class ω_j.
- $P(x)$ is the evidence. Referring to the law of total probability, this can be found by:

  $P(x) = \sum_{j=1}^{c} P(x|\omega_j)P(\omega_j)$

  Note that `c = 2` for the binary classification case.

Bayesian Decision Theory is concerned with examining Bayes Rule between the two classes and given a new input x, the class that the input belongs to is the one that has the larger posterior probability. Concretely, we decide that an input x belongs to class ω_1 if $P(\omega_1|x) > P(\omega_2|x)$, else we decide class ω_2. Substituting Bayes Rule into the aforementioned relationship states that:

$$P(\omega_1|x) > P(\omega_2|x)$$

$$\frac{P(x|\omega_1)P(\omega_1)}{P(x)} > \frac{P(x|\omega_2)P(\omega_2)}{P(x)}$$

$$P(x|\omega_1)P(\omega_1) > P(x|\omega_2)P(\omega_2)$$

Note that th denominator has disappeared as it appears on both sides of the inequality. If we define a discriminant function for each class $g_i(x) = P(x|\omega_i)P(\omega_i)$, Bayesian Decision Theory can concretely be stated as choosing class ω_1 if $g_1(x) > g_2(x)$ or $g_1(x) - g_2(x) > 0$, else we choose class ω_2.

Referring to the class lecture notes, the decision boundary is when the choice is equiprobable, or when

$g_1(x) = g_2(x)$. We derived the following relationship to find the decision boundary given a single input feature x.

$$g_1(x) = g_2(x)$$

$$\frac{(x-\mu_1)^2}{2\sigma_1^2} - \frac{1}{2}\log\sigma_1^2 + \log(P(\omega_1)) = \frac{(x-\mu_2)^2}{2\sigma_2^2} - \frac{1}{2}\log\sigma_2^2 + \log(P(\omega_2))$$

$\log$ is $\log_e$ or $\ln$ as the convention adopted in our course. Solving for x determines where the decision boundary is and then using the rule of $g_1(x) - g_2(x) > 0$, we decide which class we classify the input x into. This is now a quadratic function and if we did a bit of rearranging, $g_1(x) - g_2(x)$ can be re-expressed in the following way:

$$\left(\frac{1}{2\sigma_2^2} - \frac{1}{2\sigma_1^2}\right)x^2 + \left(\frac{\mu_1}{\sigma_2^2} - \frac{\mu_2}{\sigma_1^2}\right)x + \left(\frac{1}{2}\log\left(\frac{\sigma_2^2}{\sigma_1^2}\right) + \log\left(\frac{P(\omega_1)}{P(\omega_2)}\right) - \frac{\mu_1^2}{2\sigma_1^2} + \frac{\mu_2^2}{2\sigma_2^2}\right) = 0$$

Now if we can take the above equation and factor it into a product of two terms: $(x-a_1)(x-a_2) = 0$, we simply substitute the value `x` into the factored form and determine if the $(x-a_1)(x-a_2)$ is positive. If it is, classify `x` to be part of ω_1, else it would be in ω_2. The helper function `calculate_decision_boundary` found in the `helper` directory creates a function that helps you decide whether the input x belongs to class 1 or class 2. To call this method, it requires six inputs: The mean, variance and prior probabilities for class ω_1 and ω_2 respectively.

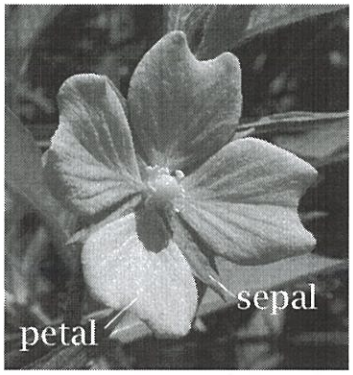
Specifically, you would call it this way:

```
p = calculate_decision_boundary(mu1, var1, pw1, mu2, var2, pw2)
```

`mu1`, `var1` and `pw1` are the mean, variance and prior probabilities for class ω_1 and `mu2`, `var2` and `pw2` are the mean, variance and prior probabilities for class ω_2. The output `p` will be an anonymous function that you can use to input in your predictions. Therefore, you can specify features as a single value, vector or matrix and the output will be the same size as the input where each value will either be 1 or 2 denoting which class each input belongs to. Concretely, if we were given a vector of input features `X`, doing `out = p(X);` would give you a vector `out` the same size as `X` which will contain the values 1 or 2 that denote which class each input feature got classified into.

3.2 - Examining the Data

The data set we will be using was curated by Ronald Fisher who was a prominent British statistician and biologist in the early 20th century. This is the Iris dataset and it consists of sepal and pedal lengths and widths all measured in centimetres (cm) for a sampling of three different flowers: Iris Setosa, Iris Veriscolour and Iris Viriginica. The sepal and petal portions of a flower is illustrated below: (Source: https://en.wikipedia.org/wiki/Sepal):



The sepals function as protection for the flower in bud and often as support for petals when flowers are in bloom. The petals are modified leaves that surround the reproductive parts of flowers.

This dataset is provided in the `data` directory and is called `lab2flowers.mat`. The link to this dataset can be found at the beginning under the **List of Files Included** section where the **Dataset Files** section is. Performing `load lab2flowers.mat` will load in two variables into your workspace. These variables are as follows:

- `X`: A `150 x 4` matrix where there are 150 training examples and 4 features. The 4 features are arranged in the columns of the matrix as follows:
    1. Sepal length (cm)
    2. Sepal width (cm)
    3. Petal length (cm)
    4. Petal width (cm)
- `y`: A `150 x 1` column vector that contains labels 1, 2 or 3 corresponding to each training example where each label corresponds to the following flower:
    1. Iris Setosa
    2. Iris Versicolour
    3. Iris Viriginica

  As such, `X(i,:)` is the i^{th} row or input training example and the corresponding label is found in `y(i)` and is either 1, 2 or 3.

3.3 - Investigate Binary Classification with Bayesian Decision Theory

Your task is to complete the skeleton file `part3.m` in the `tocomplete` directory for the lab. This part is to demonstrate how to design a classifier using Bayesian Decision Theory using one feature. The pipeline to design a classifier using Bayesian Decision Theory usually consists of the following steps. We are assuming that the feature is Gaussian distributed.

1. Compute the prior probabilities for the class ω_1 and ω_2.
2. Compute the mean and variance (or standard deviation) of the single feature for class ω_1 and ω_2
3. Determine the decision boundary that separates between the two classes using the class-conditional probability distributions and prior probabilities.
4. Using the decision boundary, use this to classify between the two classes for new inputs.
5. Calculate the classification accuracy.

For this lab, we will use attempt to create a binary classifier that distinguishes between the **Iris Setosa** ω_1 and the **Iris Versicolour** flowers ω_2. In addition, we will use a single feature which will be the **Sepal Width** or the second column of the matrix `X` when loaded into MATLAB.

The above steps are what you are to implement in `part3.m`. The skeleton for this file is shown below:

```
% Clear all variables, close all figures and add the helper directory to
% MATLAB's system path
clearvars;
close all
addpath('../helper');

% Load in data
load('../data/lab2flowers.mat');
```

```
% Preliminaries - Extract out the Sepal Width then separate into the
% different classes
% Class 1 is the Iris Setosa and Class 2 is the Iris Versicolour
%%% FILL IN YOUR CODE HERE

% 1. Compute the prior probabilities for the class 1 and class 2
%%% FILL IN YOUR CODE HERE

% 2. Compute the mean and variance for the Sepal Width for each class
%%% FILL IN YOUR CODE HERE

% 3. Determine the decision boundary that separates between the two classes
%%% FILL IN YOUR CODE HERE

% 4. Using the decision boundary, use this to classify between the two
% classes for new inputs.
x = [1, 2, 3, 4, 5]; % Define new inputs (in cm)
%%% FILL IN YOUR CODE HERE

% 5. Calculate the classification accuracy.
%%% FILL IN YOUR CODE HERE
```

The beginning of the code is standard. We clear all variables, close all figures and add the helper directory to MATLAB's system path before we proceed. We load in the data for this part and the rest of the code is up to you to complete. We will now go into each step in further detail.

Preliminaries

The first thing you'll need to do is split up the training data so that you extract the Sepal Width between the Iris Setosa and Iris Versicolour classes. Using a `logical` condition vector to check for class 1 and class 2 may be fruitful here. You will need to do this before the next steps that follow.

3.3.1 - Prior Probabilities

Once you do the above, write code to compute the prior probabilities for the Iris Setosa and Iris Versicolour classes. Simply count how many training examples belong to each class and divide by the total number of examples overall for class 1 and 2. This counting will be up to you of course.

Deliverables

State what the prior probabilities for each class is in your report.

3.3.2 - Mean and Variance

Determine the mean and variance of the Iris Setosa and Iris Versicolour classes. To compute the mean, use the `mean` function in MATLAB and to compute the variance, use the `var` function. Save both the means and variances for the next part.

Deliverables

State what the mean and variance is for each class in your report.

3.3.3 - Determine the Decision Boundary

Deliverables

Use the `calculate_decision_boundary` function to determine what the decision boundary is using the quantities found in Sections 3.3.1 and 3.3.2 for each class.

3.3.4 - Make Predictions

Deliverables

Using the Sepal Widths: $x = (1, 2, 3, 4, 5)$, use the decision boundary defined from the previous step to determine which class each input belongs to.

3.3.5 - Determine Classification Accuracy

Deliverables

- Now that you have deterimined the decision boundary, use the function produced in Section 3.3.3 to predict which class each point in your training set belongs to. Once you determine this, calculate what the classification accuracy is when classifying between the two classes.
- Comment on the classification accuracy. How does this compare with the accuracy that you've seen with logistic regression?

Discussion

These are open ended discussion questions designed to assess whether you have learned all you needed to learn in this lab. There are no right or wrong answers. Answer whatever comes off the top of your head and which makes sense!

1. In Part 1 of this lab, we investigated finding the decision boundary for both linearly separable and non-separable data. Is there a way to automatically determine when to use linear and non-linear decision boundaries?
2. In Part 1 of this lab when we consider the non-linear decision boundary, we only considered using the highest degree polynomial to be of order 6. Is there a way to automatically determine what the highest degree that could be used to achieve good accuracy in classification?
3. In the hand-written digit recognition system (or Part 2), we used the simple mechanism of considering each pixel to be a feature in the machine learning problem. Do you consider using the pixels themselves as reliable features?
4. Leading from Question #3, what would happen to the classification accuracy if some noise got introduced into each digit?
5. Continuing from Question #4, what features would you consider to use instead of simply the pixel intensities themselves to achieve a higher classification accuracy?
6. In Part 3 of this lab, we investigated the use of Bayesian Decision Theory combined with a single feature to classify an input into two classes. Do you believe that adding more features would help in classification accuracy? If so, what would you change in the Bayesian Decision Theory pipeline to facilitate this? If not, why do you believe this is the case?