Machine Learning — **Homework 2**
Comp540 Spring 2018 — **due 2 February 2018 at 8 pm**

This homework is due February 2 at 8 pm on Canvas. The code base `hw2.zip` *for the assignment is an attachment to* Assignment 2 *on Canvas. You will add your code at the indicated spots in the files there. Place your answers to Problems 1 and 2 (typeset) in a file called* `writeup.pdf` *and add it to the zip archive. Upload the entire archive back to Canvas before the due date and time.*

1 Gradient and Hessian of $NLL(\theta)$ for logistic regression (10 points)

- (2 points) Let $g(z) = \frac{1}{1+e^{-z}}$. Show that $\frac{\partial g(z)}{\partial z} = g(z)(1 - g(z))$.
- (4 points) Using the previous result and the chain rule of calculus, derive the following expression for the gradient of the negative log likelihood function $NLL(\theta)$ for logistic regression.

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

- (4 points) The Hessian or second derivative of the $NLL(\theta)$ can be written as $H = X^T SX$ where

$$S = diag(h_\theta(x^{(1)})(1 - h_\theta(x^{(1)})), ..., h_\theta(x^{(m)})(1 - h_\theta(x^{(m)})))$$

  Show that H is positive definite. You may assume that $0 < h_\theta(x^{(i)}) < 1$, so the elements of S are strictly positive and that X is full rank.

2 Properties of L2 regularized logistic regression (20 points)

Consider minimizing the L2 penalized logistic regression cost function:

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

for a data set $\mathcal{D} = \{(x^{(i)}, y^{(i)}) | 1 \leq i \leq m; x^{(i)} \in \Re^d; y^{(i)} \in \{0, 1\}\}$, regularization constant $\lambda \geq 0$.

Answer the following true/false questions with respect to minimizing the regularized cost function for logistic regression. Provide a brief justification for your answer.

- (True or False) $J(\theta)$ has multiple locally optimal solutions.

- (True or False) Let $\theta^* = argmin_\theta J(\theta)$ be a global optimum. θ^* is sparse (has many zero entries).
- (True or False) If the training data is linearly separable, then some coefficients θ_j might become infinite if $\lambda = 0$.
- (True or False) The first term of $J(\theta)$ always increases as we increase λ.

3 Implementing a k-nearest-neighbor classifier (25 points)

To get started, please download the code base `hw2.zip` from Canvas. When you unzip the archive, you will see two folders: `knn` and `logreg` and a pdf file `hw2.pdf` which contains this document.

Navigate to the `knn` folder. You will see the following files.

Name	**Edit?**	**Read?**	**Description**
k_nearest_regressor.py	Yes	Yes	contains functions that implement k-nearest-neighbor classifier
knn.ipynb	Yes	Yes	Python notebook that will run the k-nearest-neighbor classifier
data_utils.py	No	No	utilities for loading the CIFAR-10 dataset
datasets	No	No	folder for holding CIFAR-10 dataset

Table 1: Contents of folder `knn`

During training, the k-nearest-neighbor classifier takes the training data and simply remembers it. During testing, it classifies a test image by comparing to all training images and selecting the majority label among the k most similar training examples The value of k is computed by cross-validation. In this exercise you will implement these steps and gain proficiency in writing efficient, vectorized code for distance computations. You will also select the best value of k by cross-validation. You will be using a version of the CIFAR-10 object recognition dataset for this exercise.

Download the data

Open up a terminal window and navigate to the `datasets` folder inside the `knn` folder. Run the `get_datasets.sh` script. On my Mac, I just type in `./get_datasets.sh` at the shell prompt. A new folder called `cifar_10_batches_py` will be created and it will contain 50,000 labeled images for training and 10,000 labeled images for testing. We have provided a function to read this data in `data_utils.py`. Each image is a 32×32 array of RGB triples. It is preprocessed by subtracting the mean image from all images.

Set up training and test sets

To make the nearest neighbors computations efficient, we subsample 5000 images from the training set and 500 images from the test set, and flatten each image into a 1-dimensional array of size 3072 (i.e., $32 \times 32 \times 3$). So the training set is a `numpy` matrix of size 5000×3072, and the set-aside test set is of size 500×3072.

Create a k-nearest-neighbor classifier

Open up the file `k_nearest_neighbor.py` and read the `init` and `train` methods of the class `KNearestNeighbor` defined there. Remember that training a k-nearest-neighbor classifier is a no-op. The classifier simply remembers the data and does no further processing

Classify test data with a k-nearest-neighbor classifier

We would now like to classify test data with the k-nearest-neighbor classifier. Recall that we can break down this process into two steps.

- First we compute the distances between all test examples and all training examples.
- Given these distances, for each test example we find the k nearest training examples and then compute a majority vote on the labels.

Distance matrix computation with two loops (5 points)

We compute the distance matrix between all training and test examples. For example, if there are M training examples and N test examples, this stage should result in a N x M matrix where each element (i, j) is the distance between the i^{th} test set example and j^{th} training set example. First, open `k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time (4 points).

In `knn.ipynb`, we visualize the distance matrix. Each row is a single test example and its distances to training examples. Notice the structured patterns in the distance matrix, where some rows or columns are visible brighter. Note that with the default color scheme black indicates low distances while white indicates high distances. What in the data is the cause behind the distinctly bright rows? What causes the columns? Fill your answer in the indicated cell in `knn.ipynb` (1 point).

Compute majority label (5 points)

Given the distance matrix computed above, the labels for all test examples can be computed for a specified value of k by selecting the closest k training examples and finding the majority label in that set. Complete the implementation of the method `predict_labels` in `k_nearest_neighbor.py` and then run the cell in `knn.ipynb` that tests your implementation for $k = 1$. You should expect to see approximately 27% accuracy. Run the following cell (in which $k = 5$) and you should see slightly higher accuracies than for $k = 1$.

Distance matrix computation with one loop (5 points)

Now lets speed up distance matrix computation by using partial vectorization with one loop. Implement the function `compute_distances_one_loop` in `k_nearest_neighbor.py` and run the cell in `knn.ipynb` under the heading: Speeding up distance computations. You should learn to use *broadcasting* operations in `numpy`. See `http://eli.thegreenplace.net/2015/broadcasting-arrays-in-numpy/` and `https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html` for more on broadcasting.

Distance matrix computation with no loops (5 points)

You should implement this function using only basic array operations; in particular you should not use functions from `scipy`. Try to formulate L2 distance between points in two sets in terms of matrix multiplication and two broadcast sums.

Now `knn.ipynb` will time the three versions of the distance computation. You should see that the no loop version is significantly faster (about x100) than the two loop version.

Choosing k by cross validation (5 points)

Split up the training data into `num_folds` folds. After splitting, the array `X_train_folds` and `y_train_folds` should each be lists of length `num_folds`, where `y_train_folds[i]` is the label vector for the points in `X_train_folds[i]`. You will use cross-validation to select k from the set $\{1, 3, 5, 8, 10, 12, 15, 20, 50, 100\}$.

For each value of k in the set, run the k-nearest-neighbor algorithm `num_folds` times, where in each case you use all but one of the folds as training data and the last fold as a 'test' set. Store the accuracies for all folds and all values of k in the `k_to_accuracies` dictionary.

Next, `knn.ipynb` will plot the values in the `k_to_accuracies` dictionary, yielding a figure which should like Figure 1.

Based on the cross-validation results above, choose the best value for k. Then, retrain the classifier using all the training data, and test it on the test data. You should be able to get above 28% accuracy on the test data.

4 Implementing logistic regression (40 points)

In this problemt, you will implement logistic regression and regularized logistic regression and use it to analyze two different data sets. When you unzip the homework archive, you

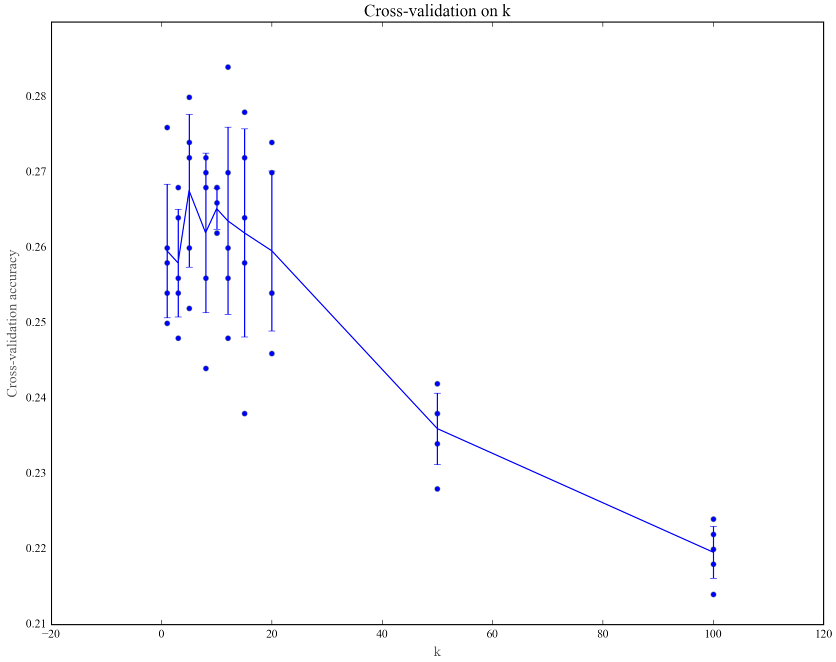

Figure 1: Choosing k by crossvalidation on the CIFAR-10 dataset

will see the `logreg` folder which contains the code base for this problem. The contents of this folder are in Table 2.

Problem 3, Part A: Logistic regression (15 points)

In this problem, you will implement logistic regression to predict whether a student gets admitted into a university. Suppose you are an admissions officer and you want to determine an applicant's chance of admission based on the scores on two exams. You have historical data from previous applicants consisting of their scores on the two exams and the admission decision made. Your task is to build a classifier that estimates the probability of admission based on the two scores.

Visualizing the dataset

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. In the first part of `ex1.ipynb`, we load the data and display it on a 2-dimensional plot as shown in Figure 2, where the axes are the two exam scores, and the positive and negative examples are shown with different colors.

Name	**Edit?**	**Read?**	**Description**
logistic_regressor.py	Yes	Yes	contains unregularized and regularized logistic regressor classes
utils.py	Yes	Yes	contains the sigmoid function, functions to standardize features,and for selecting lambda by crossvalidation
plot_utils.py	No	Yes	Functions to plot 2D classification data, and classifier decision boundaries
logreg.ipynb	No	Yes	Python notebook that will run your functions for unregularized logistic regression
logreg_reg.ipynb	No	Yes	Python notebook that will run your functions for regularized logistic regression
logreg_spam.ipynb	No	Yes	Python notebook that will run your functions for spam data classification
ex2data1.txt	No	No	Linearly separable sataset for use by `ex1.ipynb`
ex2data2.txt	No	No	Nonlinearly separable dataset for use by `ex1_reg.ipynb`
spamData.mat	No	No	Dataset for the second part of the assignment

Table 2: Contents of folder `logreg`

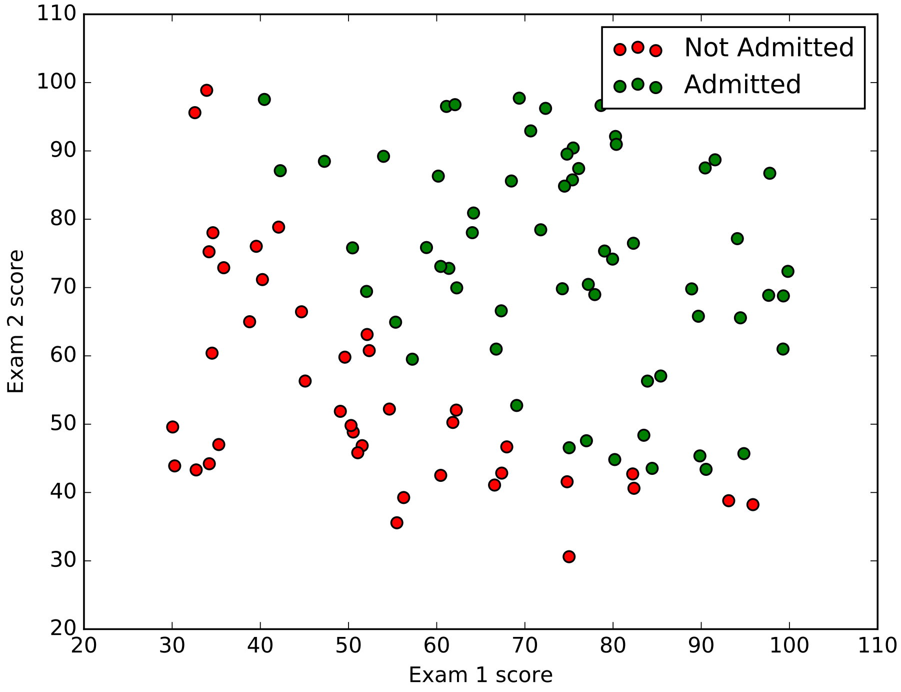

Figure 2: The training data

Problem 3A1: Implementing logistic regression : the sigmoid function (5 points)

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

$$h_\theta(x) = g(\theta^T x)$$

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

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

Your first step is to implement the `sigmoid` function in `utils.py`. When you are finished, try testing a few values by calling `sigmoid(x)`. For large positive values of `x`, the sigmoid should be close to 1, while for large negative values, the sigmoid should be close to 0. Evaluating `sigmoid(0)` should give you exactly 0.5. Your code should also work with vectors and matrices. For a matrix, your function should calculate the sigmoid function on every element.

Problem 3A2: Cost function and gradient of logistic regression (5 points)

Now you will implement the cost function and gradient for logistic regression. Complete the `loss` and `grad_loss` methods in the `LogisticRegressor` class in `logistic_regressor.py` to return the cost and gradient for logistic regression. The cost function in logistic regression is

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

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

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

Note that while this gradient looks identical to the linear regression gradient, the formula is actually different because linear and logistic regression have different definitions of $h_\theta(x)$. Once you are done, `logreg.ipynb` will call your `loss` method with a zero vector θ. You should see that the cost is about 0.693. The gradient of the loss function with respect to an all-zeros θ vector is also computed and should be $[-0.1, -12.01, -11.26]^T$.

Learning parameters using fmin_bfgs

`scipy.optimize`'s `fmin_bfgs` is an optimization solver that finds the minimum of a function. For logistic regression, you want to optimize the cost function $J(\theta)$ with parameters θ. We have provided the `train` method in `logistic_regressor.py` which uses `min_bfgs` to find the best parameters for the logistic regression cost function, given a labeled data set `X` and `y`.

If you have completed the `loss` and `grad_loss` methods correctly, `fmin_bfgs` will converge on the right optimization parameters and return the final values of the parameter vector θ. Notice that by using `fmin_bfgs`, you did not have to write any loops yourself, or set a learning rate like you did for gradient descent. This is all done by `fmin_bfgs`: you only needed to provide a function calculating the cost and the gradient. Once `fmin_bfgs` completes, `logreg.ipynb` will call your cost method using the optimal θ. You should see that the cost is about 0.203. This final θ value will then be used to plot the decision boundary on the training data, resulting in a figure similar to Figure 3. We also encourage you to look at the code in `plot_utils.py` to see how to plot such a boundary using the θ values.

Problem 3A3: Prediction using a logistic regression model (5 points)

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

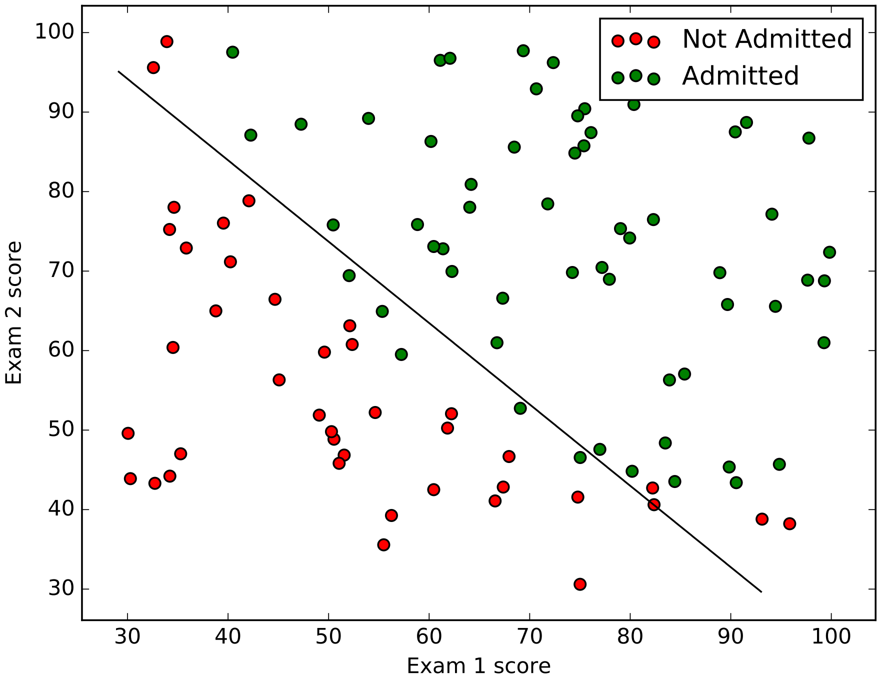

Figure 3: The decision boundary

you should expect to see an admission probability of about 0.774. Another way to evaluate the quality of the parameters we have found is to see how well the learned model predicts on our training set. In this part, your task is to complete the `predict` method for the `LogisticRegressor` class in `logistic_regressor.py`. The `predict` method will produce 1 or 0 predictions given an X matrix and a learned parameter vector θ. The `logreg.ipynb` script will now proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct. You should expect to see 89% accuracy on the training data.

Problem 3, Part B: Regularized logistic regression (15 points)

In this part of the exercise, you will implement regularized logistic regression to predict whether microchips from a fabrication plant pass quality assurance (QA). During QA, each microchip goes through various tests to ensure it is functioning correctly. Suppose you are the product manager of the factory and you have the test results for some microchips on two different tests. From these two tests, you would like to determine whether the microchips should be accepted or rejected. To help you make the decision, you have a dataset of test results on past microchips, from which you can build a logistic regression model. You will use another notebook, `logreg_reg.ipynb` to complete this portion of the exercise.

Visualizing the data

The plotting function in `plot_utils.py` is used to generate the plot in Figure 4, where the axes are the two test scores, and the positive (y = 1, accepted) and negative (y = 0, rejected) examples are shown with different colored markers. Figure 4 shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straightforward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

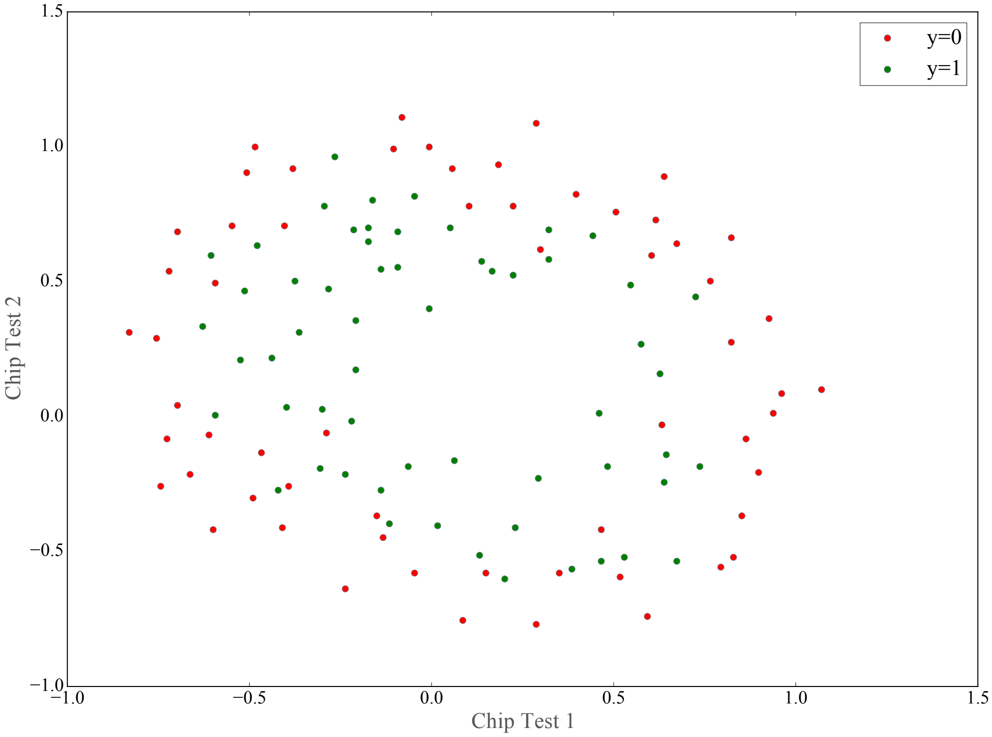

Figure 4: Plot of training data

Feature mapping

One way to fit the data better is to use basis function expansion – i.e., we create more features from each data point. In particular, we use `sklearn`'s `preprocessing` module to map the features into all polynomial terms of x_1 and x_2 up to the sixth power.

As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 28-dimensional vector. A logistic regression classifier trained on this higher-dimension feature vector will have a more complex decision boundary which will appear nonlinear when drawn in our 2-dimensional plot. While the feature mapping allows us to build a more expressive classifier, it also more susceptible to overfitting. In the next parts of the exercise, you will implement regularized logistic regression to fit the data and also see for yourself how regularization can help combat the overfitting problem.

Problem 3B1: Cost function and gradient for regularized logistic regression (10 points)

Now you will implement code to compute the cost function and gradient for regularized logistic regression. Complete the methods `loss` and `grad_loss` for the class `RegLogisticRegressor` in `logistic_regression.py` to return the cost and gradient. The regularized cost function is:

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

Note that we do not regularize θ_0. The gradient of the cost function is a vector where the j^{th} element is defined as follows:

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

Similar to the previous parts, you will use `fmin_bgfs` to learn the optimal parameters θ. If you have completed the cost and gradient for regularized logistic regressioncorrectly, you should be able to step through the next part of `logreg_reg.ipynb` to learn the parameter θ.

Plotting the decision boundary

To help you visualize the model learned by this classifier, we have provided a function in `plot_utils.py` which plots the (non-linear) decision boundary that separates the positive and negative examples. In this function, we plot the non-linear decision boundary by computing the classifiers predictions on an evenly spaced grid and then draw a a contour plot of where the predictions change from y = 0 to y = 1. After learning the parameters θ, the next step in `logreg_reg.m` will plot a decision boundary shown in Figure 5.

Problem 3B2: Prediction using the model (2 points)

One way to evaluate the quality of the parameters we have found is to see how well the learned model predicts on our training set. In this part, your task is to complete the `predict` method for the `RegLogisticRegressor` class in `logistic_regressor.py`. The `predict` method will produce 1 or 0 predictions given an X matrix and a learned parameter vector θ.

The logreg_reg.ipynb notebook will now proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct. You should expect to see 83.05% accuracy on the training data.

Problem 3B3: Varying λ (3 points)

In this part of the exercise, you will get to try out different regularization parameters for the dataset to understand how regularization prevents overfitting. Notice the changes in the decision boundary as you vary λ. With a small λ (say 0), you should find that the classifier gets almost every training example correct, but draws a very complicated boundary, thus overfitting the data. With a larger λ, you should get a simpler decision boundary which still separates the positives and negatives fairly well. However, if λ is set to too high a value (say 100), you will not get a good fit and the decision boundary will not follow the data so well, thus under-fitting the data. Show plots of the decision boundary for two lambdas showing overfitting and under-fitting on this data set. Include them in your writeup.

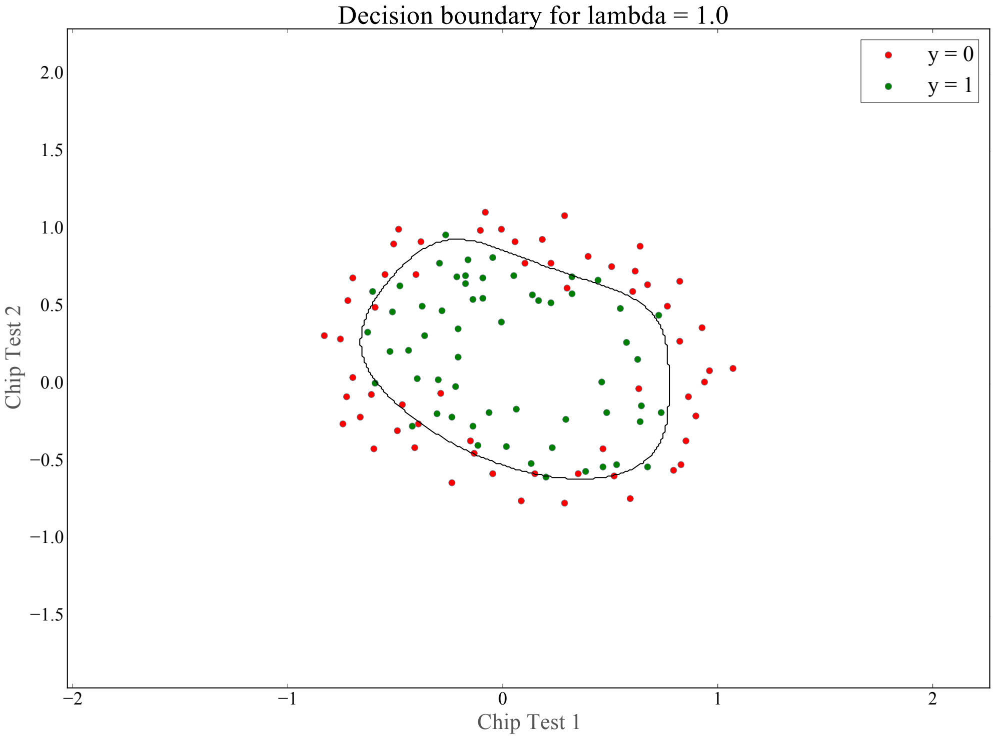

Figure 5: Training data with decision boundary for lambda = 1

Problem 3B4: Exploring L1 and L2 penalized logistic regression

In this sub problem, you will work with `sklearn`'s logistic regression model and in particular, compare L1 and L2 regularization. In the notebook `logreg_reg.ipynb`, we first set up a logistic regression model with L2 regularization and then another model with L1 regularization using `sklearn`'s LogisticRegression. It is useful to compare the coefficients of the learned models under these penalization schemes. Try varying λ (variable `reg`) in the notebook and see how the two models behave in terms of loss as well as the number of non-zero coefficients.

Problem 3 Part C: Logistic regression for spam classification (10 points)

(Source: Kevin Murphy) Consider the email spam data set developed by Hastie et. al. It has 4601 email messages, from which 57 features have been extracted. This data is in `spamData.mat` which has a training set of size 3065 and a test set of size 1536. The features are as follows:

- 48 features in [0,100], giving the percentage of words in a given email which match a given word on the list. The list contains words such as "business", "free", "george", etc. The data was collected by George Forman, so his name occurs quite a lot.
- 6 features in [0,100], giving the percentage of characters in the email that match a given character on the list. The characters are ;, (, [, !, $, #.
- Feature 55: the average length of an uninterrupted sequence of capital letters (max is 40.3, min is 4.9).
- Feature 56: the length of the longest uninterrupted sequence of capital letters (max is 45.0, mean is 52.6).
- Feature 57: the sum of the lengths of uninterrupted sequence of capital letters (max is 25.6, mean is 282.2).

Feature transformation (2 points)

Scaling features is important in logistic regression. Here you will implement three methods for transforming features in the spam data set: `stdFeatures`, `logTransformFeatures` and `binarizeFeatures`. Here are the descriptions of the transformations.

- (0 points) Standardize features: transform each column of the data set by subtracting the mean of the column and dividing by the standard deviation. Thus each column has a mean of zero and unit variance.
- (1 point) Log transform features: replace every $x_j^{(i)}$ by $log(1 + x_j^{(i)})$.
- (1 point) Binarize features: replace every $x_j^{(i)}$ by 1 if $x_j^{(i)} > 0$ or 0 otherwise.

Fitting regularized logistic regression models (L2 and L1) (8 points)

For each representation of the features, we will fit L1 and L2 regularized logistic regression models. Your task is to complete the function `select_lambda_crossval` in `utils.py` to select the best regularization parameter λ by 10-fold cross-validation on the training data. This function takes a training set `X` and `y` and sweeps a range of λ's from `lambda_low` to `lambda_high` in steps of `lambda_step`. Default values for these parameters are in `logreg_spam.ipynb`. For each λ, divide the training data into ten equal portions using `sklearn.cross_validation`'s function `KFold`. Train a regularized `sklearn` logistic regression model (of the L1 and L2 variety) on nine of those parts and test its accuracy on the left out portion. The accuracy of a model trained with that λ is the average of the ten test errors you obtain. Do this for every λ in the swept range and return the lambda that yields the highest accuracy .

`logreg_spam.ipynb` will then build the regularized model with the best lambda for both L1 and L2 regularization you calculate and then determine the training and test set accuracies of the model. You should see test set accuracies between 91% and 94% with the different feature transforms and the two regularization schemes. Comment on the model sparsities with L1 and L2 regularization. Which class of models will you recommend for this data set and why?

What to turn in

Please zip up all the files in the archive (including files that you did not modify) and submit it as `hw2_netid`.zip on Canvas before the deadline, where `netid` is to be replaced with your netid. Include a PDF file in the archive that presents your plots and discussion of results from the programming component of the assignment. Also include typeset solutions to the written problems 1 and 2 of this assignment in your writeup. Only one submission per group of two, please.