

2019 - CAB420 - Machine Learning

Group (of 2/1) Assignment 1

Theory (10 Marks)

Logistic regression is a method of fitting a probabilistic classifier that gives soft linear thresholds. It is common to use logistic regression with an objective/loss function consisting of the negative log probability of the data plus a L_2 regularizer:

$$L(w) = - \sum_{i=1}^N \log \left(\frac{1}{1 + e^{-y_i(w^T x_i + b)}} \right) + \lambda \|w\|_2^2$$

(Here w does not include the “extra” weight w_0 .)

- (a) Find the partial derivatives $\frac{\partial L}{\partial w_j}$.
- (b) Find the partial second derivatives $\frac{\partial^2 L}{\partial w_j \partial w_k}$.
- (c) From these results, show that $L(w)$ is a convex function.

Hint: A function L is convex if its Hessian (the matrix \mathbf{H} of second derivatives with elements

$H_{j,k} = \frac{\partial^2 L}{\partial w_j \partial w_k}$) is positive semi-definite (PSD). A matrix \mathbf{H} is PSD if and only if

$$a^T H a \equiv \sum_{j,k} a_j a_k H_{j,k} \geq 0$$

for all real vectors a .

Practice

1. Features, Classes, and Linear Regression (10 Marks)

Assume that we have collected a dataset and stored the data in the data directory. The dataset has already been splitted into train and test subsets as stored in two files: `mTrainData.txt` and `mTestData.txt`. Our task is to develop a regression learner, trained on the train data, to be used in the test data.

In this problem we'll explore construction of a feature matrix, and the use of Matlab classes for our learners. Load the dataset (in the data directory) into Matlab,

```
mTrain=load('data\mTrainData.txt');  
whos
```

(“whos” shows you what variables are in memory).

Features We will separate it into a single feature x and a target value y ,

```
Xtr=mTrain(:,1); Ytr=mTrain(:,2);  
whos  
plot(Xtr,Ytr,'bo');  
title('Plotting all training data points');
```

Often we will only want to use a few of the data points (to see how things change with fewer or more data); you can access the first few entries as e.g.

```
plot(Xtr(1:20),Ytr(1:20),'bo');  
title('Plotting 20 first training data points');
```

Sometimes we will want to add features to improve the complexity of the prediction. For example, you can create a “constant” feature and a quadratic feature of x using

```
Xtr_2 = [ ones(size(Xtr,1) ,1) , Xtr , Xtr.^2];  
whos
```

Xtr_2 is now a feature matrix filled with an all-ones feature, the original features (x -values), and the squares of the original x values. There is a function called `polyx` in the directory that will create these functions for you.

Class Objects We will use Matlab classes to implement our learner methods. An old-style class is created using a directory preceded by `@`. For example, included in your directory is a linear regression learner, `linearReg`. The methods associated with this class are the Matlab `.m` files located within it. The constructor is `linearReg`; all the other functions are called by providing a `linearReg` object as the first argument. (That tells Matlab where to look for the function.) So, for example,

```
learner = linearReg(Xtr_2 ,Ytr); % train a linear regression learner  
yhat = predict( learner , Xtr_2 ); % use it for prediction
```

will create and learn a linear regression predictor from the data Xtr_2, Ytr, and then use it to predict the y -values at the original training data points.

In regression, we will often want to plot a “line”. You can do this by creating new x points at a close spacing, and evaluating the predictor on them:

```
xline = [0:.01:2] ' ; % transpose : make a column vector , like training x  
yline = predict( learner , polyx (xline ,2) ); % assuming quadratic features
```

To turn in:

- (a) Plot the training data in a scatter plot.
- (b) Create a linear regression learner using the above functions. Plot it on the same plot as the training data.
- (c) Create plots with the data and a higher-order polynomial (3, 5, 7, 9, 11, 13).
- (d) Calculate the mean squared error (MSE) associated with each of your learned models on the training data.
- (e) Calculate the MSE for each model on the test data (in `mTestData.txt`).
- (f) Calculate the MAE for each model on the test data. Compare the obtained MAE values with the MSE values obtained in above (e).
- (g) Don't forget to label your plots; see `help legend`.

Notice: for better visualisation, use only 20 data points from the training data.


2. kNN Regression (15 Marks)

In class, we talked about *k-nearest-neighbor* methods, which predict the target value of a new example using the values of the k nearest training examples. Here we will create a kNN regression learner and use it to predict the underlying function.

- (a) Using the `knnRegress` class, implement (add code to) the `predict` function to make it functional. Comment the line `error('You should write prediction code here');`, otherwise you will get an error.

Hint: You can compare to the `knnClassify` class as well; compared to that class, you need to replace the “count neighbors in each class” code with something that computes the average value of those neighbors.

Notes: (1) The `knnRegress` and `knnClassify` classes are constructed as `knnRegress(K,X,Y)` where K is the number of nearest neighbors, and X and Y are the training data to store for the look-up process. (2) In the code, the line with `bsxfun` just computes the distances to the training data points quickly; an equivalent but slightly slower line is commented out below it that may be more interpretable, and an even more interpretable version would use a for-loop over training data points, but this would be slow.

(b) Using the same technique as in Problem 1a, plot the predicted function for several values of k : 1, 2, 3, 5, 10, 50. (You can just use a for-loop to do this.) How does the choice of k relate to the “complexity” of the regression function? 

(c) We discussed in class that the k -nearest-neighbor classifier’s decision boundary can be shown to be piecewise linear. What kind of functions can be output by a nearest neighbor regression function? Briefly justify your conclusion. (You do not need to discuss the general case – just the 1-dimensional regression picture such as your plots.)

3. Hold-out and Cross-validation (15 Marks)

In this problem we study the use of hold-out test data and cross-validation methods to estimate the desired complexity of a model. We will continue to use the data from Problem 1 and the k -nearest-neighbor regression class you created in the previous problem.

(a) Similarly to Problem 1 and 2, compute the MSE of the test data on a model trained on only the first 20 training data examples for $k = 1, 2, 3, \dots, 140$. Plot both train and test MSE versus k on a log-log scale (see `help loglog`). Assign title to your figure (ie. 20 data) and legends to your curves (ie. test, train). **Discuss what you observed from the figure.**

(b) Repeat, but use all the training data. **What happened? Contrast with your results from Problem 1** (hint: which direction is “complexity” in this picture?).

(c) Using *only the training data*, estimate the curve using 4-fold cross-validation. Split the training data into two parts, indices 1:20 and 21:140; use the larger of the two as training data and the smaller as testing data, then repeat three more times with different sets of 20 and average the MSE. Plot this together with (a) and (b). Use different colors or marks to differentiate three scenarios. **Discuss why might we need to use this technique via comparing curves of three scenario?**

(Hint: how many data points did we use in this part versus the previous part?) Again, you may want to use a for-loop:

```
for k =1:140,
    % 140 is number of train data, you might
    % you might need to change it.
    for cv =1:4 ,
        % cross validation
        iTest = ...
        % choose 20 indices for testing
        iTrain = setdiff (1:140 , iTest ); % rest for testing
        learner = ...
        % train on X(iTrain ,:)
        mseTrain (k, cv) = ...
        % train loss
        mseTest (k, cv) = ...
        % test loss
    end ;
end;
figure ; plot (...
    % average and plot results
```

4. Nearest Neighbor Classifiers (15 Marks)

Load the `iris.txt` data into Matlab, and select the first two data features only for the moment. You should first permute the data so that it is not in sorted order.

```
iris = load('data\iris .txt');
pi = randperm( size (iris ,1));
Y = iris(pi ,5); X = iris(pi ,1:2);
```

You can read about these data at <http://archive.ics.uci.edu/ml/datasets/Iris>.

(a) Plot the data by their **feature values**, using the class value to select the color. The easiest way to do this is to use `find` to identify indices for which `Y` takes on each of its possible values. You can use `unique` to see what values `Y` contains. You can then plot each class in turn, using `hold on` to plot on top of the previous plot. (When you don't want this anymore, use `hold off`).

(b) Use the provided `knnClassify` class to learn a 1-nearest-neighbor predictor. Use the function `class2DPlot(learner,X,Y)` to plot the decision regions and training data together.

(c) Do the same thing for several values of `k` (say, [1, 3, 10, 30]) and **comment on their appearance**.

(d) Now split the data into an 80/20 training/validation split. For `k` = [1, 2, 5, 10, 50, 100, 200], learn a model on the 80% and calculate its performance (# of data classified incorrectly) on the validation data. What value of `k` appears to generalize best given your training data? Comment on the performance at the two endpoints, in terms of over- or under-fitting.

```
% first split data into Xtrain , Ytrain and Xvalid , Yvalid
kvalues =[1 ,2 ,5 ,10 ,50 ,100 ,200];
for i=1: length ( kvalues )
    learner = knnClassify (... % train model on X/Ytrain
    Yhat = predict ( learner ,... % predict results on X/Yvalid
    err (i) = ... % count what fraction of predictions are wrong
end ;
figure ; plot (... % plot results as a function of k
```

5. Perceptrons and Logistic Regression (25 Marks)

Note: Debugging machine learning algorithms can be quite challenging, since the results of the algorithm are highly data-dependent, and often somewhat randomized (initialization, etc.). I suggest starting with an extremely small step size and verifying both that the learner's prediction evolves slowly in the correct direction, and that the objective function J decreases monotonically. If that works, go to larger step sizes to observe the behavior. I often use the pause command to slow down execution so that I can examine my code's behavior; you can also step through the code using Matlab's debugger.

In this problem, we'll build a logistic regression classifier and train it on separable and non-separable data. Since it will be specialized to binary classification, I've named the class `logisticClassify2`.

We'll start by building two binary classification problems, one separable and the other not:

```
iris=load('data/iris.txt'); % load the text file
X = iris(:,1:2); Y=iris(:,end); % get first two features
[X Y] = shuffleData(X,Y); % reorder randomly
X = rescale(X); % works much better for rescaled data
XA = X(Y<2, :); YA=Y(Y<2); % get class 0 vs 1
XB = X(Y>0, :); YB=Y(Y>0);
```

For this problem, we are focused on the learning algorithm, rather than performance so, we will not bother creating training and validation splits; just use all your data for training.

Note: Be sure to shuffle your data before doing SGD in part (f) - otherwise, if the data are in a pathological ordering (e.g., ordered by class), you may experience strange behavior and slow convergence during the optimization.

- (a) Show the two classes in a scatter plot and verify that one is linearly separable while the other is not.
- (b) Write (fill in) the function `@logisticClassify2/plot2DLinear.m` so that it plots the two classes of data in different colors, along with the decision boundary (a line). Include the listing of your code in your report. To demo your function plot the decision boundary corresponding to the classifier

$$\text{sign}(.5 + 1x_1 - .25x_2)$$

along with the A data, and again with the B data. You can create a blank learner and set the weights by:

```
learner = logisticClassify2(); % create "blank" learner
learner = setClasses(learner, unique(YA)); % define class labels using YA or YB
wts = [theta0 theta1 theta2]; % TODO: fill in values
learner = setWeights(learner, wts); % set the learner's parameters
```

- (c) Complete the `predict.m` function to make predictions for your linear classifier. Note that, in my code, the two classes are stored in the variable `obj.classes`, with the first entry being the “negative” class (or class 0), and the second entry being the “positive” class. Again, verify that your function works by computing & reporting the error rate of the classifier in the previous part on both data sets A and B. (The error rate on data set A should be ≈ 0.0505 .)

You can also test this and your previous function by comparing your `plot2DLinear` output with the generic `plotClassify2D` function, which shows the decision boundary “manually” by calling `predict` on a dense grid of locations, rather than analytically as your `plot2DLinear` function should do.

- (d) In my provided code, I first transform the classes in the data Y into “class 0” (negative) and “class 1” (positive). In our notation, let $z = \theta x^{(i)T}$ is the linear response of the perceptron, and σ is the standard logistic function

$$\sigma(z) = (1 + \exp(-z))^{-1}$$

The (regularized) logistic negative log likelihood loss for a single data point j is then

$$J_j(\theta) = -y^{(j)} \log \sigma(\theta x^{(j)T}) - (1 - y^{(j)}) \log (1 - \sigma(\theta x^{(j)T})) + \alpha \sum_i \theta_i^2$$

where $y^{(j)}$ is either 0 or 1. Derive the gradient of the regularized negative log likelihood J_j for logistic regression, and give it in your report. (You will need this in your gradient descent code for the next part.)

- (e) Complete your `train.m` function to perform stochastic gradient descent on the logistic loss function. This will require that you fill in:

- (1) computing the surrogate loss function at each iteration ($J = \frac{1}{m} \sum J_i$, from the previous part);
- (2) computing the prediction and gradient associated with each data point $x^{(i)}, y^{(i)}$;
- (3) a gradient step on the parameters θ ;
- (4) a stopping criterion (usually either `stopIter` iterations or that J has not changed by more than `stopTol` since the last iteration through all the data).

(f) Run your logistic regression classifier on both data sets (A and B); for this problem, use no regularization ($\alpha = 0$). Describe your parameter choices (stepsize, etc.) and show a plot of both the convergence of the surrogate loss and error rate, and a plot of the final converged classifier with the data (using e.g. `plotClassify2D`). In your report, please also include the functions that you wrote (at minimum, `train.m`, but possibly a few small helper functions as well).

- (g) To implement the mini batch gradient descent on the logistic function complete your `train_in_batches.m` function. This will require that you :

- (1) fill in `create_mini_batches.m` function, which generates the mini batches of data. shuffle your data inside this function and set the batch size to 11;
- (2) update the training iterations in `train_in_batches.m`, in contrast to the training in section (e), in this function training will be performed on each of these data batches;
- (3) change the training method in `logisticClassify2` (set it into "train_in_batches") and run your mini batch logistic regression classifier. In your report please include the functions that you wrote `create_mini_batches.m` and `train_in_batches.m`)