CS273P Homework #3

Machine Learning & Data Mining: Spring 2018

Due: Thursday May 10th, 2018

Write neatly (or type) and show all your work!

Download the provided Homework 3 code, and replace last week's code (several new elements have been added, and others modified).

Problem 1: Perceptrons and Logistic Regression (80 pts)

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 = np.genfromtxt("data/iris.txt",delimiter=None)
X, Y = iris[:,0:2], iris[:,-1]   # get first two features & target
X,Y  = ml.shuffleData(X,Y)       # reorder randomly (important later)
X,_  = rescale(X)                # works much better on rescaled data

XA, YA = X[Y<2,:], Y[Y<2]        # get class 0 vs 1
XB, YB = X[Y>0,:], Y[Y>0]        # get class 1 vs 2
```

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: The code uses numpy's `permute` to iterate over data randomly; should avoid issues due to the default order of the data (by class). Similarly, rescaling and centering the data may help speed up convergence as well.

(a) (5pts) Show the two classes in a scatter plot (one for each data set) and verify that one data set is linearly separable while the other is not.

(b) (10pts) Write (fill in) the function `plotBoundary(...)` in `logisticClassify2.py` to compute the points on the decision boundary. This will plot the data & boundary quickly, which is useful for visualizing the model during training. To demo your function plot the decision boundary corresponding to the classifier

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

along with the A data, and again with the B data. (These fixed parameters will look like an OK classifier on one data set, but a poor classifier on the other.) You can create a "blank" learner and set the weights by:

```
import mltools as ml
from logisiticClassify2 import *

learner = logisticClassify2();          # create "blank" learner
learner.classes = np.unique(YA)         # define class labels using YA or YB
wts = np.array([theta0,theta1,theta2]); # TODO: fill in values
learner.theta = wts;                    # set the learner's parameters
```

(c) (10pts) Complete the `logisticClassify2.predict` function to make predictions for your linear classifier. Note that, in my code, the two classes are stored in the variable `self.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, and higher on set B.)

(d) (5pts) Verify that your predict code matches your boundary plot by using `plotClassify2D` with your manually constructed learner on the two data sets. This will call "predict" on a dense grid of points, and you should find that the resulting decision boundary matches the one you computed analytically.

(e) (10pts) In my provided code, I first transform the classes in the data Y into $Y01$, with canonical labels for the two classes: "class 0" (negative) and "class 1" (positive). In our notation, let $z = x^{(j)} \cdot \theta^T$ is the linear response of the perceptron, and σ is the standard logistic function

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

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

$$J_j(\theta) = -y^{(j)} \log \sigma(x^{(j)}\theta^T) \ - \ (1 - y^{(j)}) \log(1 - \sigma(x^{(j)}\theta^T))$$

where $y^{(j)}$ is either 0 or 1. Derive the gradient of the 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.)

(f) (15pts) Complete your `train(...)` 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 epoch ($J = \frac{1}{m}\sum J_j$, from the previous part);
(2) computing the prediction and gradient associated with each data point $x^{(j)}, y^{(j)}$;
(3) a stopping criterion (usually either `stopEpochs` epochs or that J has not changed by more than `stopTol` since the last epoch (here meaning, pass through all the data).
Note on plotting: The code generates plots as the algorithm runs, so you can see its behavior over time; this is done with `pyplot.draw()`. Run your code either interactively or as a script to see these display over time; unfortunately it does not work easily in Jupyter (you will only see a plot at the end, which is difficult to use for diagnostics.

(g) (10pts) Run your logistic regression classifier on both data sets (A and B). Describe your parameter choices (stepsize, etc.) and show a plot showing the convergence of the surrogate loss and error rate (e.g., the loss values as a function of epoch during gradient descent), and a plot showing the final converged classifier with the data (using e.g. `plotClassify2D`). In your report, please also include a listing of any functions that you wrote (at minimum, `train()`, but possibly a few small helper functions as well).

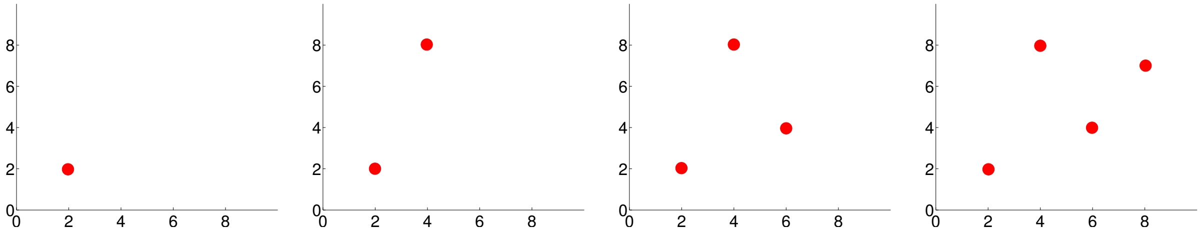
(h) (15pts) Add an L2 regularization term ($+\alpha \sum_i \theta_i^2$) to your surrogate loss function, and update the gradient and your code to reflect this addition. Try re-running your learner with some regularization (e.g. $\alpha = 2$) and see how different the resulting parameters are. Find a value of α that gives noticeably different results & explain them.

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 manually step through the code – for example by pausing after each parameter update using `raw_input()` (Python 2.7) or `input()` (Python 3) – so that I can examine its behavior. You can also (of course) use a more sophisticated debugger.

Problem 2: Shattering and VC Dimension (20 pts)

Consider the following learners and data points, which have two real-valued features x_1, x_2. Which of the following four examples can be shattered by each learner? Give a brief explanation / justification and use your results to guess the VC dimension of the classifier. (You do not have to give a formal proof, just your reasoning.)

Data points:



(a) (b) (c) (d)

For the two learners, $T[z]$ is the sign threshold function, $T[z] = +1$ for $z \geq 0$ and $T[z] = -1$ for $z < 0$. The learner parameters a, b, c are real-valued scalars, and each data point has two real-valued input features x_1, x_2.

(a) $T(\, a + bx_1 \,)$

(b) $T(\, (x_1 - a)^2 + (x_2 - b)^2 + c \,)$

(c) $T(\, (a * b)x_1 + (c/a)x_2 \,)$

(d) $T(\, a + b * x_1 + c * x_2 \,) \cdot T(\, d + b * x_1 + c * x_2 \,)$