Project 2: Multiclass and Linear Models

There are two parts to this project. The first is to play around with multiclass reductions. The second is to play around with linear models. The project files are available for download in the p2 folder on the course Files page.

Files To Turn In

`multiclass.py`: This contains the multiclass reduction implementations.

`gd.py`: This contains the gradient descent implementation.

`linear.py`: This contains the linear classifier implementation.

`partners.txt`: Lists the **names** and **last four digits of the UID** of all members in your team.

`writeup.pdf`: Answers all the written questions in this assignment (denoted by **WU#** in this file).

Please **do not** change the file names listed above. Submit partners.txt even if you do the project alone. **Only one member** in a team is supposed to submit the project. Submit all the files together in a zipped folder.

Autograding

Your code will be autograded for technical correctness. Please **do not** change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's output -- will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work. Please **do not** import (potentially unsafe) system-related modules such as sys, exec, eval... Otherwise the autograder will assign a zero point without grading.

Multiclass Classification *[30% impl, 20% writeup]*

In this section, you will explore the differences between three multiclass-to-binary reductions: one-against-all (OAA), all-versus-all (AVA) and a tree-based reduction (TREE). We are evaluating on a text classification task: wine classification (sorry if you're not a wine snob and this doesn't mean much to you). Your job is, given the description of the wine, predict the type of wine. There are two tasks: WineData has 20 different wines, WineDataSmall is just the first five of those (sorted roughly by frequency).

You can find the names of the wines both in WineData.labels as well as the file wines.names.

To start out, be sure to import everything:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> import multiclass
>>> import util
>>> from datasets import *
```

To get you started, here's how we can train decision "stumps" (aka depth=1 decision trees) on the large data set:

```
>>> h = multiclass.OAA(20, lambda: DecisionTreeClassifier(max_depth=1))
>>> h.train(WineData.X, WineData.Y)
>>> P = h.predictAll(WineData.Xte)
>>> mean(P == WineData.Yte)
0.29499072356215211
```

That means 29% accuracy on this task. (Note that you might get a slightly different value for accuracy due to randomization in the SKLearn classifier implementations.) The most frequent class is:

```
>>> mode(WineData.Y)
1
>>> WineData.labels[1]
'Cabernet-Sauvignon'
```

And if you were to always predict label 1, you would get the following accuracy:

```
>>> mean(WineData.Yte == 1)
0.17254174397031541
```

So we're doing a bit (12%) better than that using decision stumps.

The default implementation of OAA uses decision tree confidence (probability of prediction) to weigh the votes. You can switch to zero/one predictions to see the effect:

```
>>> P = h.predictAll(WineData.Xte, useZeroOne=True)
>>> mean(P == WineData.Yte)
0.19109461966604824
```

As you can see, this is **markedly worse**.

Switching to the smaller data set for a minute, we can train, say, depth 3 decision trees:

```
>>> h = multiclass.OAA(5, lambda: DecisionTreeClassifier(max_depth=3))
>>> h.train(WineDataSmall.X, WineDataSmall.Y)
>>> P = h.predictAll(WineDataSmall.Xte)
>>> mean(P == WineDataSmall.Yte)
0.60393873085339167
>>> mean(WineDataSmall.Yte == 1)
0.40700218818380746
```

So using depth 3 trees we get an accuracy of about 60% (this number varies a bit), versus a baseline of 41%. That's not too terrible, but not great.

We can look at what this classifier is doing.

```
>>> WineDataSmall.labels[0]
'Sauvignon-Blanc'
>>> util.showTree(h.f[0], WineDataSmall.words)
```

This should show the tree that's associated with predicting label 0 (which is stored in h.f[0]). The 1s mean "likely to be Sauvignon-Blanc" and the 0s mean "likely not to be".

WU1 (10%): Answer questions A, B and C for both OAA and AVA. (Questions about “indicative” are open-ended. Any reasonable answers with analysis will be credited.)

(A) What words are most indicative of *being* Sauvignon-Blanc? Which words are most indicative of not being Sauvignon-Blanc? What about Pinot-Noir (label==2)?

(B) Train depth 3 decision trees on the full WineData task (with 20 labels). What accuracy do you get? How long does this take (in seconds)? One of my least favorite wines is Viognier -- what words are indicative of this?

(C) Compare the accuracy using zero-one predictions versus using confidence. How much

difference does it make?

WU2 (10%):

Now, you must implement a tree-based reduction. Most of train is given to you, but predict you must do all on your own. A tree class is provided to help you:

```
>>> t = multiclass.makeBalancedTree(range(6))
>>> t
[[0 [1 2]] [3 [4 5]]]
>>> t.isLeaf
False
>>> t.getLeft()
[0 [1 2]]
>>> t.getLeft().getLeft()
0
>>> t.getLeft().getLeft().isLeaf
True
```

Here is an example using WineDataSmall.

```
>>> t = multiclass.makeBalancedTree(range(5))
>>> h = multiclass.MCTree(t, lambda: DecisionTreeClassifier(max_depth=3))
>>> h.train(WineDataSmall.X, WineDataSmall.Y)
>>> P = h.predictAll(WineDataSmall.Xte)
>>> mean(P == WineDataSmall.Yte)
0.56017505470459517
```

Show the test accuracy you get with a balanced tree on the WineData using a DecisionTreeClassifier with max depth 3.

Gradient Descent and Linear Classification *[30% impl, 20% writeup]*

Gradient descent

To get started with linear models, we will implement a generic gradient descent methods. This should go in `gd.py`, which contains a single (short) function: `gd` This takes five parameters: the function we're optimizing, it's gradient, an initial position, a number of iterations to run, and an initial step size.

In each iteration of gradient descent, we will compute the gradient and take a step in that

direction, with step size `eta`. We will have an *adaptive* step size, where `eta` is computed as `stepSize` divided by the square root of the iteration number (counting from one).

Once you have an implementation running, we can check it on a simple example of minimizing the function `x^2`:

```
>>> import gd
>>> gd.gd(lambda x: x**2, lambda x: 2*x, 10, 10, 0.2)
(1.0034641051795872, array([ 100.        ,   36.        ,   18.5153247 ,   10.950
          7.00860578,    4.72540613,    3.30810578,    2.38344246,
          1.75697198,    1.31968118,    1.00694021]))
```

You can see that the "solution" found is about 1, which is not great (it should be zero!), but it's better than the initial value of ten! If yours is going up rather than going down, you probably have a sign error somewhere!

We can let it run longer and plot the trajectory:

```
>>> x, trajectory = gd.gd(lambda x: x**2, lambda x: 2*x, 10, 100, 0.2)
>>> x
0.003645900464603937
>>> plot(trajectory)
>>> show(False)
```

It's now found a value close to zero and you can see that the objective is decreasing by looking at the plot.

WU3 (5%): What is the impact of the step size on convergence? Find values of the step size where the algorithm diverges and converges.

WU4 (10%): Come up with a *non-convex* univariate optimization problem. Plot the function you're trying to minimize and show two runs of `gd`, one where it gets caught in a local minimum and one where it manages to make it to a global minimum. (Use different starting points to accomplish this.)

If you implemented it well, this should work in multiple dimensions, too:

```
>>> x, trajectory = gd.gd(lambda x: linalg.norm(x)**2, lambda x: 2*x, array([10,5
>>> x
array([ 0.0036459 ,  0.00182295])
>>> plot(trajectory)
```

Linear Classifiers

Our generic linear classifier implementation is in `linear.py`. The way this works is as follows. We have an interface `LossFunction` that we want to minimize. This must be able to compute the loss for a pair `Y` and `Yhat` where, the former is the truth and the latter are the predictions. It must also be able to compute a gradient when additionally given the data `X`. This should be all you need for these.

There are three loss function stubs: `SquaredLoss` (which is implemented for you!), `LogisticLoss` and `HingeLoss` (both of which you'll have to implement. We suggest holding off implementing the other two until you have the linear classifier working.

The `LinearClassifier` class is a stub implemention of a generic linear classifier with an l2 regularizer. It is *unbiased* so all you have to take care of are the weights. Your implementation should go in `train`, which has a handful of stubs. The idea is to just pass appropriate functions to `gd` and have it do all the work. See the comments inline in the code for more information.

Once you've implemented the function evaluation and gradient, you can test this. Let's begin with a very simple 2D example data set so that solutions can be plotted. Let's also start with *no regularizer* to help you figure out where errors might be if you have them. (You'll have to import `mlGraphics` to make this work.)

```
>>> import runClassifier.py
>>> import linear
>>> f = linear.LinearClassifier({'lossFunction': linear.SquaredLoss(), 'lambda':
>>> runClassifier.trainTestSet(f, datasets.TwoDAxisAligned)
Training accuracy 0.91, test accuracy 0.86
>>> f
w=array([ 2.73466371, -0.29563932])
>>> mlGraphics.plotLinearClassifier(f, datasets.TwoDAxisAligned.X, datasets.TwoDA
>>> show(False)
```

Note that even though this data is clearly linearly separable, the *unbiased* classifier is unable to perfectly separate it.

If we change the regularizer, we'll get a slightly different solution:

```
>>> f = linear.LinearClassifier({'lossFunction': linear.SquaredLoss(), 'lambda':
>>> runClassifier.trainTestSet(f, datasets.TwoDAxisAligned)
Training accuracy 0.9, test accuracy 0.86
>>> f
```

```
w=array([ 1.30221546, -0.06764756])
```

As expected, the weights are *smaller*.

Now, we can try different loss functions. Implement logistic loss and hinge loss. Here are some simple test cases:

```
>>> f = linear.LinearClassifier({'lossFunction': linear.LogisticLoss(), 'lambda':
>>> runClassifier.trainTestSet(f, datasets.TwoDDiagonal)
Training accuracy 0.99, test accuracy 0.86
>>> f
w=array([ 0.29809083,  1.01287561])

>>> f = linear.LinearClassifier({'lossFunction': linear.HingeLoss(), 'lambda': 1,
>>> runClassifier.trainTestSet(f, datasets.TwoDDiagonal)
Training accuracy 0.98, test accuracy 0.86
>>> f
w=array([ 1.17110065,  4.67288657])
```

WU5 (5%): For each of the loss functions, train a model on the binary version of the wine data (called WineDataBinary) and evaluate it on the test data. You should use lambda=1 in all cases. Which works best? For that best model, look at the learned weights. Find the *words* corresponding to the weights with the greatest positive value and those with the greatest negative value. Hint: look at WineDataBinary.words to get the id-to-word mapping. List the top 5 positive and top 5 negative and explain.