Project 2: Recycling Used Parts for Fun and Profit

Foundations of Intelligent Systems, Spring 2014
Prof. Zanibbi
Due Date: Friday May 2nd, 2014 (11:59pm)

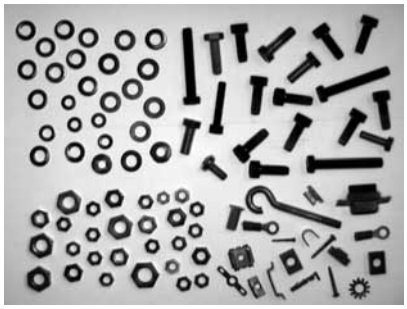
Submission Instructions

Through MyCourses, submit a .zip archive containing 1) your program files as .py files, 2) additional data files needed to run your trained classifiers, 3) a README file explaining how to run your programs and interpret the output, and 4) a .pdf file containing the project write-up.

Problem Description

You work for a company that sells used metal parts for profit. You have been asked to sort recycled metal parts collected from old machines into four categories: nuts, bolts, rings, and scrap. The customer provides the following table, identifying the profit or cost for each correctly/incorrectly identified part once they have received them.

Profit function ($)				
	Correct Class			
Assigned Class	bolt	nut	ring	scrap
bolt	0.20	-0.07	-0.07	-0.07
nut	-0.07	0.15	-0.07	-0.07
ring	-0.07	-0.07	0.05	-0.07
scrap	-0.03	-0.03	-0.03	-0.03



Your system consists of a conveyer belt on which parts are carried, a camera and computer vision system that measure visual features for each part, and a sorter that places each part in a given bin. Your task is to create a classifier that identifies the type of each part as it arrives on the conveyor belt. The vision system provides two features for use in your classifier:

1. **Six-fold rotational symmetry:** a measure of the average symmetry in the shape of the object, obtained by folding a black-and-white image in half in six different directions, each time counting the percentage of disagreeing pixels between the two sides of the fold. This helps to discriminate rings and nuts.
2. **Eccentricity:** a measure of variation in the distance to the center of the object as we trace the object's outer contour (roughly, a measure of how 'uncircular' an object is). This helps discriminate bolts from nuts and rings.

Data: training and testing datasets are provided. Both files contain one example per line, with rotational symmetry in the first column, eccentricity in the second column, and the correct class **(given as 1 for bolt, 2 for nut, 3 ring and 4 scrap)** in the third column.

You need to implement **four programs**, `trainMLP.py`, `executeMLP.py`, `trainDT.py` and `executeDT.py`.

Multi-Layer Perceptron

The first two programs are for a multi-layer perceptron with a single hidden layer, as discussed in class.

1. `trainMLP.py` takes a file containing training data as input and produces as output:
   (a) Five files containing the trained neural network (for use with `executeMLP.py`) after 0 (for initial weights), 10, 100, 1000, and 10,000 epochs.[1] Use the sigmoid (logistic) function as the activation function, and include bias weights in your linear models. Network weights should initially be set randomly in the interval [-1,1], with a learning rate (α) of 0.1. Run over samples in-order, updating weights after each sample is run.
   (b) An image containing a plot of the *learning curve*. The learning curve represents the total sum of squared error (SSE) over all training samples after each *epoch* (i.e. one complete pass over all training samples). Use the python matplotlib library (see http://matplotlib.org/users/index.html) to produce the plots. On the CS computer systems, you may use matplotlib by including '`from pylab import *`' at the top of your program.

2. `executeMLP.py` takes a file defining a trained network and a data file, and runs the network on the data file. The program should produce:
   (a) **(standard output)** The number of classification errors, recognition rate (% correct) and profit obtained.
   (b) **(standard output)** A *confusion matrix* that gives a histogram of the output class (rows) vs. correct class (columns) results.
   (c) **(image file)** containing a plot of the *classification regions*. The program runs the current network over a grid of equally spaced points in the region (0,0) to (1,1) (the limits of the feature space), using a different color to represent the classification decision at each point. Use matplotlib to produce this image.

Decision Tree Ensemble

The third and fourth programs are for training and executing a decision tree ensemble (i.e. AdaBoost or a Random Forest) that uses *continuous* features with binary splits. As discussed in class, to find the best split for a continuous feature we can 1) sort the samples by their values for an attribute, and then 2) compute the information gain at each split point between adjacent samples (using their midpoint), from the smallest to the largest feature value. **Note that we will need to re-use features**. You are welcome to use stumps (i.e. only one feature in each tree) or depth-limited trees (e.g. a maximum depth of 2 or 3 from the root of the tree to each leaf).

1. `trainDT.py` takes a training data file and produces:
   (a) Four files, for the ensemble after 1, 10, 100, and a maximum (e.g. 200) number of trees have been added (for use with `executeDT.py`).
   (b) A learning curve plot, showing the SSE between the classifier outputs and true classes after each classifier is added to the ensemble, showing the sum of squared differences between the weighted vote output of the ensemble vs. the true class.
2. `executeDT.py` takes a file defining a decision tree ensemble and a data set, and produces the same output as `executeMLP` for the ensemble in the passed file.

Experiment

Use `trainNN.py` and the train_data.csv data file to train a multi-layer perceptron with two input nodes, five hidden nodes, and four output nodes (for four classes), plus bias nodes (a

1. It is strongly recommended that you use the Python 'pickle' module, which allows you to save and recover object states from files easily.

2-5-4 architecture). The train program should save the neural network to a file (using pickle) after 0, 10, 100, 1000, and 10,000 training epochs. Similarly, train a decision tree ensemble using `trainDT.py` and train_data.csv.

Then, use `executeNN.py` to run each network (all five versions) on train_data.csv, and then test_data.csv. Collect the performance metrics and classification region images for each classifier. Do the same for the decision tree ensemble, running all four ensembles on the training and test sets.

Repeat the experiment with the MLP a number of times (**remembering to save your previous results!**), and then report the results for what you consider to be the best run as described below. For the decision trees, make sure to run training and testing multiple times if you use a random forest (AdaBoost will be deterministic).

Write-Up

1. **Classifier Designs:** Provide a pair of diagrams illustrating the architecture of your neural network and decision tree ensemble, and briefly discuss their similarities and differences. Which classifier did you expect to perform better, and why?
2. **Data Sets:** Provide a separate plot for the training and test data sets, so that the class for each sample can be seen using color (you can use matplotlib for this). Comment on the distribution of the two data sets.
3. **Results:** Provide the following.
   (a) A table showing for each number of saved epochs for the MLP: recognition rate, profit, and confusion matrix produced for the test data.
   (b) A second table showing the same information for each of the different ensembles (with 1, 10, 100, etc. trees).
   (c) A figure showing plots of the classification regions produced for 1) the different numbers of epochs in training the MLP, and 2) for the different numbers of decision trees in the ensemble. Arrange these in two tables, so that the evolution of the class regions is easy to see.
   (d) The learning curve images for the MLP and decision tree ensemble.
4. **Discussion:** Comment on the results. How do the hypotheses (i.e. class boundaries) and performance metrics change for the MLP and decision tree over time? Which versions of the classifiers performed best, did this match your expectation, and why do you think this was the case? **Also provide** any additional comments on the problem, classifiers, results or process that you feel is informative or interesting.

Grading

- 40% Correctness of implementation
- 10% Test set accuracy for the best classifiers
- 10% Coding style (use a research programming style)
- 40% Write-up

Acknowledgement

This dataset and part image is taken from the introductory pattern recognition textbook, “Classification, Parameter Estimation and State Estimation" by van der Heijden, Duin, Ridder and Tax (Wiley, 2004).