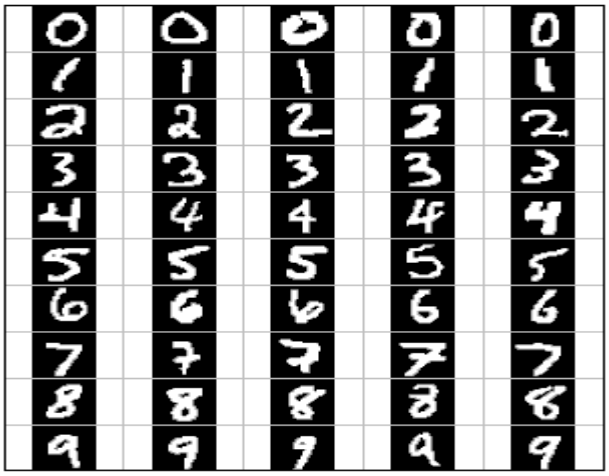
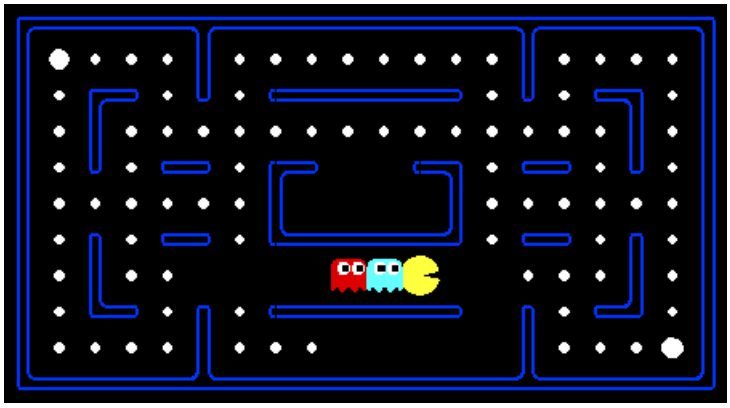
Project 2: Classification



Classification of handwritten digits: Which digit?



Classification of actions: Which action?

Introduction

In this project, you will design three classifiers: a perceptron classifier, a large-margin (MIRA) classifier, and a slightly modified perceptron classifier for behavioral cloning. You will test the first two classifiers on a set of scanned handwritten digit images, and the last on sets of recorded pacman games from various agents. Even with simple features, your classifiers will be able to do quite well on these tasks when given enough training data.

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore.

Files you'll edit:

perceptron.py	The location where you will write your perceptron classifier. (Q1)
answers.py	Answer to Question 2 goes here. (Q2)
dataClassifier.py	The wrapper code that will call your classifiers. You will also write your enhanced feature extractor here. You will also use this code to analyze the behavior of your classifier. (Q6, Q4)
mira.py	The location where you will write your MIRA classifier. (Q3)
perceptron_pacman.py	The location where you will write your perceptron classifier for behavioral cloning (Q5)

Files you might want to look at:

classificationMethod.py	Abstract super class for the classifiers you will write.
samples.py	I/O code to read in the classification data.
pacman.py	Pacman GameState object from the first project is defined here.
util.py	Code defining some useful tools. You may be familiar with some of these by now, and they will save you a lot of time.

Question 1 (4 points): Perceptron

A skeleton implementation of a perceptron classifier is provided for you in perceptron.py. In this part, you will fill in the train function.

Unlike the naive Bayes classifier, a perceptron does not use probabilities to make its decisions. Instead, it keeps a weight vector w^y of each class y (y is an identifier, not an exponent). Given a feature list f, the perceptron compute the class y whose weight vector is most similar to the input vector f. Formally, given a feature vector f (in our case, a map from pixel locations to indicators of whether they are on), we score each class with:

$$\text{score}(f, y) = \sum_i f_i w_i^y$$

Then we choose the class with highest score as the predicted label for that data instance. In the code, we will represent w^y as a **Counter**.

Learning weights

In the basic multi-class perceptron, we scan over the data, one instance at a time. When we come to an instance *(f, y)*, we find the label with highest score:

$$y' = \arg\max_{y''} \text{score}\,(f, y'')$$

We compare y' to the true label y. If $y'=y$, we've gotten the instance correct, and we do nothing. Otherwise, we guessed y' but we should have guessed y. That means that w^y should have scored f higher, and $w^{y'}$ should have scored f lower, in order to prevent this error in the future. We update these two weight vectors accordingly:

$$w^y = w^y + f$$

$$w^{y'} = w^{y'} - f$$

Using the addition, subtraction, and multiplication functionality of the Counter class in util.py, the perceptron updates should be relatively easy to code. Certain implementation issues have been taken care of for you in perceptron.py, such as handling iterations over the training data and ordering the update trials. Furthermore, the code sets up the weights data structure for you. Each legal label needs its own Counter full of weights.

Question

Fill in the train method in perceptron.py. Run your code with:

```
python dataClassifier.py -c perceptron
```

Hints and observations

- The classify method of perceptron takes **list** of features to be predicted; not just a single feature vector.
- The command above should yield validation accuracies in the range between 40% to 70% and test accuracy between 40% and 70% (with the default 3 iterations). These ranges are wide because the perceptron is a lot more sensitive to the specific choice of tie-breaking than naive Bayes.

- One of the problems with the perceptron is that its performance is sensitive to several practical details, such as how many iterations you train it for, and the order you use for the training examples (in practice, using a randomized order works better than a fixed order). The current code uses a default value of 3 training iterations. You can change the number of iterations for the perceptron with the `-i iterations` option. Try different numbers of iterations and see how it influences the performance. In practice, you would use the performance on the validation set to figure out when to stop training, but you don't need to implement this stopping criterion for this assignment.

Question 2 (1 point): Perceptron Analysis

Visualizing weights

Perceptron classifiers, and other discriminative methods, are often criticized because the parameters they learn are hard to interpret. To see a demonstration of this issue, we can write a function to find features that are characteristic of one class. (Note that, because of the way perceptrons are trained, it is not as crucial to find odds ratios.)

Question

Fill in findHighWeightFeatures(self, label) in perceptron.py. It should return a list of the 100 features with highest weight for that label. You can display the 100 pixels with the largest weights using the command:

```
python dataClassifier.py -c perceptron -w
```

Use this command to look at the weights, and answer the following question. Which of the following sequence of weights is most representative of the perceptron?

Answer the question answers.py in the method q2, returning either 'a' or 'b'.

Question 3 (6 points): MIRA

A skeleton implementation of the MIRA classifier is provided for you in mira.py. MIRA is an online learner which is closely related to both the support vector machine and perceptron classifiers. You will fill in the trainAndTune function.

Theory

Similar to a multi-class perceptron classifier, multi-class MIRA classifier also keeps a weight vector w^y of each label *y*. We also scan over the data, one instance at a time. When we come to an instance *(f, y)*, we find the label with highest score:

$$y' = \arg\max_{y''} \text{score}\,(f, y'')$$

We compare *y'* to the true label *y*. If *y'=y*, we've gotten the instance correct, and we do nothing. Otherwise, we guessed *y'* but we should have guessed *y*. Unlike the perceptron, we update the weight vectors of these labels with a variable step size:

$$w^y = w^y + \tau f$$

$$w^{y'} = w^{y'} - \tau f,$$

where

$$\tau = \frac{\left(w^{y'} - w^y\right) f + 1}{2\|f\|_2^2},$$

where $||f||_2$ is the euclidean (L^2) norm. However, we would like to cap the maximum possible value of τ by a positive constant C, which leads us to

$$\tau = \min\left(C, \frac{\left(w^{y'} - w^y\right) f + 1}{2\|f\|_2^2}\right)$$

Question

Implement trainAndTune in mira.py. This method should train a MIRA classifier using each value of C in Cgrid. Evaluate accuracy on the held-out validation set for each C and choose the C with the highest validation accuracy. In case of ties, prefer the lowest value of C. Test your MIRA implementation with:

```
python dataClassifier.py -c mira --autotune
```

Hints and observations

- The L^2 norm can be calculated as dot product of coordinate vectors: $||f||_2 = \sqrt{f \cdot f}$

- Pass through the data self.max_iterations times during training.
- To use a fixed value of *C=0.001*, remove the --autotune option from the command above.
- Validation and test accuracy when using --autotune should be around the 60 – 70 %.
- The same code for returning high odds features in your perceptron implementation should also work for MIRA if you're curious what your classifier is learning.

Question 4 (6 points): Digit Feature Design

Building classifiers is only a small part of getting a good system working for a task. Indeed, the main difference between a good classification system and a bad one is usually not the classifier itself (e.g. perceptron vs. naive Bayes), but rather the quality of the features used. So far, we have used the simplest possible features: the identity of each pixel (being on/off).

To increase your classifier's accuracy further, you will need to extract more useful features from the data. The EnhancedFeatureExtractorDigit in dataClassifier.py is your new playground. When analyzing your classifiers' results, you should look at some of your errors and look for characteristics of the input that would give the classifier useful information about the label. You can add code to the analysis function in dataClassifier.py to inspect what your classifier is doing. For instance in the digit data, consider the number of separate, connected regions of black pixels, which varies by digit type. 1, 2, 3, 5, 7 tend to have one contiguous region of black space while the loops in 6, 8, 9 create more. The number of black regions in a 4 depends on the writer. This is an example of a feature that is not directly available to the classifier from the per-pixel information. If your feature extractor adds new features that encode these properties, the classifier will be able exploit them. Note that some features may require non-trivial computation to extract, so write efficient and correct code.

Question

Add new binary features for the digit dataset in the EnhancedFeatureExtractorDigit function. Note that you can encode a feature which takes 3 values [1,2,3] by using 3 binary features, of which only one is on at the time, to indicate which of the three possibilities you have (i.e. extra feature vectors of (0, 0, 1), (0, 1, 0) and (1, 0, 0)). In theory, features aren't conditionally independent as naive Bayes requires, but your classifier can still work well in practice. We will test your classifier with the following command:

```
python dataClassifier.py -d digits -c naiveBayes -f -a -t 1000
```

With the basic features (without the -f option), your optimal choice of smoothing parameter should yield 82% on the validation set with a test performance of 78%. You will receive 3 points for implementing new feature(s) which yield any improvement at all. You will receive 3 additional points if your new feature(s) give you a test performance greater than or equal to 84% with the above command.

Question 5 (4 points): Behavioral Cloning

You have built two different types of classifiers, a perceptron classifier and mira. You will now use a modified version of perceptron in order to learn from pacman agents. In this question, you will fill in the train method in perceptron_pacman.py. This code should be similar to the methods you've written in perceptron.py.

For this application of classifiers, the data will be states, and the labels for a state will be all legal actions possible from that state. Unlike perceptron for digits, all of the labels share a single weight vector w, and the features extracted are a function of both the state and possible label. For each action, the classify method in perceptron_pacman.py calculates the score as follows:

$$\text{score}(s, \text{a}) = w * f(s, a)$$

Then the classifier assigns whichever label receives the highest score:

$$a' = \arg\max_{a''} \text{score}\,(s, a'')$$

Training updates occur in much the same way that they do for the standard classifiers. Instead of modifying two separate weight vectors on each update, the weights for the actual and predicted labels, both updates occur on the shared weights as follows:

$$w = w + f(s, a) \# \text{ Correct action}$$
$$w = w - f\,(s, a') \,\# \text{ Guessed action}$$

Question

Fill in the train method in perceptron_pacman.py. Run your code with:

```
python dataClassifier.py -c perceptron -d pacman
```

This command should yield validation and test accuracy of over 70%.

Question 6 (4 points): Pacman Feature Design

In this part you will write your own features in order to allow the classifier agent to clone the behavior of observed agents. We have provided several agents for you to try to copy behavior from:

- **StopAgent**: An agent that only stops
- **FoodAgent**: An agent that only aims to eat the food, not caring about anything else in the environment.
- **SuicideAgent**: An agent that only moves towards the closest ghost, regardless of whether it is scared or not scared.
- **ContestAgent**: Agent from the first project that smartly avoids ghosts, eats power capsules and food.

We've placed files containing multiple recorded games for each agent in the data/pacmandata directory. Each agent has 15 games recorded and saved for training data, and 10 games for both validation and testing.

Question

Add new features for behavioral cloning in the enhancedPacmanFeatures function in dataClassifier.py.

Upon completing your features, you should get at least 90% accuracy on the ContestAgent, and 80% on each of the other 3 provided agents with 4 training iterations. You can directly test this using the --agentToClone <Agent name>, -g <Agent name> option for dataClassifier.py:

```
python dataClassifier.py -c perceptron -d pacman -f -g ContestAgent -t 1000 -s 100 -i 4
```

Other helpful options

We have also provided a new ClassifierAgent in pacmanAgents.py for you that uses your implementation of perceptron_pacman. You can run this agent to see how your implementation of features works with the following command:

```
python pacman.py -p ClassifierAgent --agentArgs agentToClone=<Agent Name>
```

Where <Agent Name> is one of the following: StopAgent, FoodAgent, SuicideAgent or ContestAgent. Note that the command is case sensitive, and will default to ContestAgent (with no warning), if you provide some other agent name.

Hints

- Check the GameState class from pacman.py to see what information is available for features.

Check Your Implementations and Grading

This project includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

You may also check the results of a particular question by running (for example for question 1):

```
python autograder.py -q q1
```

Files to Edit: You will fill in portions of perceptron.py, answers.py, dataClassifier.py, mira.py and perceptron_pacman.py during the assignment. Once you have completed the assignment, you should put all the files into one single ZIP file for submission.

Evaluation: 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. The final judgement about your grade is based on the results from autograder.py. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Project tutorials are there for your support; please use them. If you can't make our tutorial sessions, let us know and we can try to arrange private Zoom session. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But we don't know when or how to help unless you ask.

Submission Policy: Please submit your implementations as one zip file named as "AI_P2_Student ID_FullName", which contains all the python codes to the related page in Moodle no later than **11.3.2024, 23:59**.

You may also do the project as a team of two students at most. In this case, please include a text file in your submission, which contains name and student ID of team members. Note, both members must submit the files in the Moodle.