Final Project: Pac-Man Ghost Hunting

Code due 23:59 on Sunday 4 May 2014
Tournament 1-3pm Thursday 8 May 2014, in Room 330 of 60 Oxford St.
Writeup due 23:59 on Thursday 8 May 2014

CS 181 Students: You will do this project in groups of three. You can seek partners via Piazza. Course staff can also help you find partners. You will submit both your code and your writeup via the iSites dropbox. Please note that the code and writeup have different deadlines and different iSites folders.

CSCI E-181 Students: You will do this assignment on your own. You will submit both your code and your writeup via the Extension School iSites dropbox. Please note that the code and writeup have different deadlines and different iSites folders.

Introduction

In this project, you will use machine learning techniques to design Pac-Man agents to navigate a maze and eat ghosts. There are two types of ghosts: *good ghosts* and *bad ghosts*. Good ghosts have a positive reward, depending on how juicy they are, and Pac-Man has a limited number of moves to try to eat the juiciest ghosts. On the other hand, bad ghosts have a very negative reward — unless Pac-Man can find the correct power capsule that allows him to eat the bad ghost. These power capsules are only active for a limited period of time after being eaten. There are a variety of challenges in this environment that might be met with machine learning techniques. For example, Pac-Man only gets noisy observations of the ghost types and capsule types. Also, the ghosts appear according to stochastic dynamics that can be modeled.

Your objective is to build a Pac-Man agent that can do well in this complicated environment. Like the practicals, this project is open-ended and there are many different ways you might tackle the problem. There are opportunities to use classification, regression, clustering, and reinforcement learning techniques. How exactly you tackle the problem and what you apply is up to you. Our only requirement for you is that you try at least three distinct machine learning techniques in solving the problem.

Most likely, you will go through the following stages in developing your agent:

Stage 1: Data collection

Stage 2: Training to recognize ghosts and capsules.

Stage 3: Reinforcement learning

To build a successful agent, you'll need to learn to identify the various ghosts in the Pac-Man world. To learn how to do this, you'll need to gather data about the environment. Then, in the second stage you need to create features and build models that will classify

Figure 1: Screenshot of the Pac-Man ghost hunting world. Here we can see three classes of ghosts, where the red and orange ghosts represent *good* ghosts and the purple ghost is a *bad* ghost. Pac-Man, however, cannot see what type of those they are! The white capsules represent the power capsules that let Pac-Man eat the *bad* ghosts for a short time. The green capsules are placebos that don't help Pac-Man with the bad ghosts.

the ghosts and identify how juicy they are. Then you'll want to use this information to build a smart Pac-Man agent to hunt the ghosts.

Rules of the Game

Goal Maximize your score by having Pac-Man gather the juiciest ghosts. The score is displayed at the bottom left hand corner of the screen.

Time Limit There is a time limit! The game runs for 1000 time steps.

The Game World Layout The world is randomly generated with walls, ghosts, and power capsules. Walls are simple: you cannot move onto those locations.

What Pac-Man Can Hunt Pac-Man can hunt good ghosts and power capsules. Pac-Man can hunt the bad ghost when the (non-placebo) power capsules are eaten.

Penalty Without food, Pac-Man gets hungry! At each time step, your score **decreases** by half a point.

Property Name	Observed	Length	Support
Quadrant	Yes	1	$\{1, 2, 3, 4\}$
Class	No	1	$\{0, 1, 2, 3, 5\}$
Score	No	1	$\mathbf{R}^+$
Feature vector	Yes	13	$\mathbf{R}^{13}$

Table 1: Summary of ghost properties.

If Pac-Man is Eaten If Pac-Man is eaten by the bad ghost, you lose 1000 points. Pac-Man is then regenerated randomly.

Ghost Hunting Each ghost has a latent class, which determines its juiciness level. The Pac-Man game graphics allow you to see the different ghost classes, encoded by the color of the ghost. However, Pac-Man does **not** know this information! Thus, you will need to gather data on the ghosts to infer which ghosts are the juiciest.

Properties of Ghosts There are two types of ghosts: good ghosts and bad ghosts. All ghosts come from a latent class, taking on values in $\{0, 1, 2, 3, 5\}$. After Pac-Man eats a good ghost, a new ghost of the one of classes $0 - 3$ is generated. You can assume the bad ghost always regenerates from Class 5. Ghosts also regenerate after every 100 steps.

Good Ghosts Each good ghost has the following properties:

1. Class (unknown): each ghost is generated from a latent class. Every ghost from that class will have features drawn from the same mean.
2. Feature vector (known): a thirteen-dimensional vector of values, which help determine the juiciness (score) of the ghost. See data collection section for details.
3. Score (unknown): this represents the juiciness of the ghost and is determined by a function of unknown weights and the features of the ghost. **Note**: the score for the bad ghost is not determined by this value and is fixed.
4. Quadrant (known): each ghost is generated from a specific quadrant of the map, labeled $1, 2, 3, 4$ (see Figure 2). Certain classes of ghosts may come from a specific quadrant more than other ghosts.

We summarize these properties in Table 1.

Bad Ghosts Bad ghosts also have a feature vector, which distinguishes them from good ghosts. The bad ghosts chase Pac-Man. If Pac-Man is eaten by a bad ghost, Pac-Man **loses 1000 points**. However, Pac-Man can eat the bad ghost for a **reward of 1200 points** if it

Figure 2: Each ghost is generated in a particular quadrant. Certain classes of ghosts are more likely to gather in a particular quadrant than others.

finds one of the right types of capsules to eat. In this game, you can assume there is **only one bad ghost at any given time**. The bad ghost is not unique: it regenerates with a new set of features.

Power Capsules Each power capsule has a feature vector associated with it, but some of them are placebos and don't help Pac-Man. If Pac-Man eats a non-placebo capsule, the bad ghost turns white and runs from Pac-Man for a limited amount of time, allowing Pac-Man to eat the ghost. The amount of time the ghost is scared is randomly drawn from some distribution and thus changes every time a new power capsule is consumed. The non-placebo power capsules are marked as white in the game graphics, and the placebos are marked in green. Pac-Man does not know what capsules are placebos and needs to learn this information.

Pac-Man does not have any labels for these capsules. Instead, Pac-Man has access to a sample of correct capsules, which he can study to figure out which capsules in the game world are the ones he wants to eat.

Summary of Pac-Man's Observations Below we summarize what Pac-Man knows and doesn't know:

- What Pac-Man knows:
  - The positions of all ghosts and power capsules
  - The features of all ghosts and power capsules
  - The quadrant the ghost is generated from
  - Samples containing features of the "correct" power capsule class
  - The bad ghost is worth -1000 points *without* the correct power capsule

  - The bad ghost is worth +1200 points *with* the correct power capsule

- What Pac-Man doesn't know[1]:
  - The class of each ghost, including whether it's a good or bad ghost
  - The juiciness (score) of the ghost
  - Whether or not each power capsule is a placebo

High-Level Code Overview

To run the game, type:

```
python pacman.py
```

By default, this loads the game without an agent, which allows you to control Pac-Man with your arrow keys. Give the keyboard inference a try first to get a feel for the game world.

The gzipped tarball's contents are shown in Figure 3. You will implement your agent in **studentAgents.py**. We have written a sample agent for you called `ExampleTeamAgent` in the file **studentAgents.py**. To load the agent, type:

```
python pacman.py -T ExampleTeam
```

The `-T` option specifies your team name and by default loads the agent `<TeamName>Agent`. Here `ExampleTeamAgent` is the name of the agent class. This will be the agent we will grade for your final submission. Your team name needs to be a valid Python module name (e.g., no dashes).

You may also wish to test other agents (that won't be graded). For instance, if you have a class named `RandomAgent` in **studentAgents.py**, you can load it using the `-p` option:

```
python pacman.py -T ExampleTeam -p RandomAgent
```

Note: to run other agents, you must also specify your team name using the `-T` option.

Sometimes (e.g., data collection or while reinforcement learning) you will want to run the game without graphics. To do so, use the option `-q` to turn on quiet graphics:

```
python pacman.py -q
```

[1]Keep in mind that some aspects you, the user, can see through the game graphics, e.g. colors of ghosts and capsules. However, Pac-Man cannot see this information.

```
final-project/
    __init__.py
    pacman.py
    observedState.py
    ...
    data/
        (training data will go here)
    ExampleTeam/ (submit this folder)
        README.txt
        __init__.py
        studentAgents.py
        config.py
        (include any pickle files, etc.
            DO NOT include your training data)
```

Figure 3: Directory structure of code.

Other Command-Line Options As mentioned above, we will assess your agent's performance on a game of 1000 steps. Although by default the game will run for infinitely many steps, you can specify the number of steps until the game terminates with the `-m` flag. Thus, running the game as

```
python pacman.py -m 1000
```

will let you see how well your agent does after just 1000 steps.

Another command-line option you may find useful is the `-s` option, which allows you to run the game using a particular random-seed. By default, the game will use a random seed of 3, but you can change this by passing in any integer, as follows:

```
python pacman.py -s 10
```

To see all the available command-line options, type

```
python pacman.py -h
```

Files you will edit and submit

- **ExampleTeam/studentAgents.py**: implement your agent here. Name your final agent's class `<TeamName>Agent`.
- **ExampleTeam/config.py**: fill in details about your team here.

Files you will use but not edit

- **observedState.py**: call functions regarding what Pac-Man knows
- **pacman.py**: run the game and data collection mode

Files you will not use or edit

- All other files provided by us

You will fill in portions of **studentAgents.py** during the assignment. You should submit this file with your code and comments. Please do not change the other files in this distribution or submit any of our original files other than these files. To make things easier, we have included a class called **observedState.py**, which encapsulates what Pac-Man knows about the state of the game.

Code Details

Two tarballs are available with code, one for Mac and one for Linux, named `final_project_code_ma` and `final_project_code_linux.tar.gz`, respectively. These tarballs contain Python files and shared objects for running the project. **There is no Windows tarball.** This is because of the need for compiled shared libraries. If you're stuck on Windows, you should SSH to a Linux machine such as `ice6.fas.harvard.edu` or use a virtual machine via, e.g., VirtalBox and Vagrant, using the files provided for practical 4. If you have trouble setting up an environment, contact the course staff.

Observed State

In the file **observedState.py**, we have included functions which give access to Pac-Man knows about the state of the game. **Read through this file carefully**, as you will call functions from this file for your Pac-Man agent. To help you get a sense of the contents of this file, we list a few important functions in this file:

getNumMovesLeft(): returns the number of moves remaining in the game.

getGhostQuadrant(ghostState): returns the quadrant the given ghost is currently in.

getLegalPacmanActions(): returns actions that Pac-Man can take from the current state.

pacmanFuturePosition(actions): returns the (x, y) position of Pac-Man after making the sequence of moves in the actions variable from the current state.

ghostFuturePosition(ghost_idx, actions): gets the (x, y) position of ghost_idx after making the number of moves in the actions variable.

getPacmanState(): gets the `AgentState` for Pac-Man. `AgentState` contains information such as Pac-Man's current (x, y) position (`AgentState.getPosition()`) and the direction Pac-Man was most recently taking (`AgentState.getDirection()`).

getGhostStates(): gets the `AgentState` for each ghost on the board.

getCapsuleData(): returns a list of tuples containing the (x, y) position of the capsule and the feature vector, corresponding to power capsules currently on the board.

getGoodCapsuleExamples(): returns a list of feature vectors from capsules that are known to be *good* capsules (i.e. that make the bad ghost scared and turn white). These capsule examples can be used to figure out which capsules on the board Pac-Man wants to eat.

scaredGhostPresent(): returns True if the attacking ghost is currently scared (i.e. Pac-Man ate the correct power capsule), else False.

hasWall(x,y): returns True if there is a wall at position (x, y), else False.

Project Stages

Stage 1: Data collection mode

First, you will need to collect some data about the world and the distribution of the ghosts. To enter data collection mode, type

```
python pacman.py -d
```

This will start the game with some set options that will help gather data about the ghosts. The data collected will be output to two files, located in the data directory:

- data/ghost_train.csv - contains training information for ghosts
- data/capsule_train.csv - contains training information for power capsules

The file **ghost_train.csv** contains the information for each training example, separated by spaces (see Table 1 for details):

```
<quadrant-id> <latent-class> <score> <feature-vector>
```

Each line of the file **capsule_train.csv** contains the feature vector of the power capsule.

Note: if you run the data collection mode without an agent (i.e. keyboard mode), you have to move Pac-Man manually to gather data on the power capsules. Otherwise, you can write a simple agent for Pac-Man to gather data on the power capsules or load the `ExampleTeamAgent` we have written for you.

Stage 2: Training

Given the data you collected from Stage 1, you want to be able to identify which ghosts are the juiciest by learning the class and score of each ghost you encounter in the world, given its feature vector. In addition, you want to identify which power capsules most closely resemble the samples you are presented with to help you hunt the bad ghost. Try out several methods and evaluate them using methods we've discussed in class. Include your results with corresponding plots and tables in your writeup.

Stage 3: Reinforcement Learning

Now that you have learned about the state of the world, write a Pac-Man agent to hunt the juiciest ghosts! Implement your Pac-Man agent in the file **studentAgents.py** with a class named after your team. To get a sense of how the code works, look at the example provided in the `ExampleTeamAgent`.

Note: you should not be re-training your parameters regarding the ghosts and capsules during each time step. Make sure you have saved your learned parameter values in a separate file that your agent can accesses that **does not** include all of the training data. Only submit this file with your agent.

Questions and Answers

What should I turn in? There are two deliverables for this project: code that can run in the competition, and a writeup. These have different deadlines, as the code is due earlier for the tournament. The writeup should be a four-to-five page typewritten document in PDF format that describes the work you did. This may include figures, tables, math, references, or whatever else is necessary for you to communicate to us how you worked through the problem. Concretely, you should turn in via the iSites dropbox: a 4-5 page PDF writeup that describes how you solved the problem. Make sure to include the name of the team and the names of all partners.

You will also turn in a zipped folder folder named after your team's name `<TeamName>` that contains the files shown in Figure 3. Keep in mind the following things:

- Fill in the README file explaining any unusual details about running your code.
- Fill in information about your team in the file **config.py**.
- Make sure your the folder name `<TeamName>` matches the name of your agent, `<TeamName>Agent` in **studentAgents.py**. We will use this agent when we run your code.
- Include any pickle, csv, etc. files your agent depends on. Do **not** include any training data.
- Make sure your code does not depend on any external dependencies beyond either standard modules or a few modules we'll make sure to have:
  - numpy
  - scipy
  - scikit-learn
  - pybrain

In particular, you'll want to be careful that any pickle files you depend on do not need to load in non-standard modules.

Submit your code via the iSites dropbox.

How will my work be assessed? The final project is intended to be a fun but realistic representation of what it is like to tackle a multi-faceted problem with machine learning. As such, there is no single correct answer and you will be expected to think critically about how to solve it, execute and iterate your approach, and describe your solution. The upshot of this open-endedness is that you will have a lot of flexibility in how you build the Pac-Man agent. You can focus on methods that we discussed in class, or you can use this as an opportunity to learn about approaches for which we did not have time or scope. You are welcome to use whatever tools and implementations help you get the job done, although if you need them at test-time you will need to include them with your code and they should not depend significantly on external libraries. Note also that you will be expected to *understand* everything you do, even if you do not implement the low-level code yourself. It is your responsibility to make it clear in your writeup that you did not simply download and run code that you found somewhere online.

You will be assessed on a scale of 25 points, divided evenly into five categories:

1. **Three Techniques:** Did you implement three distinct ideas from the course in solving the problem? For example, you could use a classification procedure for the ghosts, a clustering technique for the capsules, and a reinforcement learning method for the Pac-Man control.

2. **Effort:** Did you thoughtfully tackle the problem? Did you iterate through methods and ideas to find a solution? Did you explore several methods, perhaps going beyond those we discussed in class? Did you think hard about your approach, or just try random things?

3. **Technical Approach:** Did you make tuning and configuration decisions using quantitative assessment? Did you compare your approach to reasonable baselines? Did you dive deeply into the methods or just try off-the-shelf tools with default settings?

4. **Explanation:** Do you explain not just what you did, but your thought process for your approach? Do you present evidence for your conclusions in the form of figures and tables? Do you provide references to resources your used? Do you clearly explain and label the figures in your report?

5. **Execution:** Does your agent run? Did your methods give reasonable performance? How did you do in the tournament? Don't worry, you will not be graded in proportion to your ranking; we'll be using the ranking to help calibrate how difficult the task was and to award bonus points to those who go above and beyond.

Bonus Points for CS 181 Students: The top three teams among CS 181 (Harvard College and local cross-registrations) students will be eligible for extra credit. The first place team will receive an extra five points on the final project. The second and third place teams will each receive three extra points. All of these points are conditioned on the teams making a post on Piazza about their approach.

Bonus Points for CSCI E-181 Students: The top three individuals among CSCI E-181 (Extension School) students will be eligible for extra credit. The first place team will receive an extra five points on the final project and the second and third place teams will each receive three extra points, all conditioned on posting an explanation of their approach on Piazza. Extension school students who choose to form teams will be pooled with the Harvard College teams for purposes of awarding bonus points.

What language should I code in? Your agent will ultimately need to be in Python. In principle, you could do the learning in other languages and dump data to be read by Python.

Can I use {**scikit-learn** | **pylearn** | **torch** | **shogun** | **other ML library**}**?** You can use these tools, but not blindly. You are expected to show a deep understanding of the methods we study in the course, and your writeup will be where you demonstrate this.

The graphics are very slow on my Mac! For reasons that are a mystery to everyone, Mavericks causes the Tk graphics toolkit (which is what the Pac-Man interfaces uses) to run super slow. Many hours have been spent by the course staff trying to figure out how to resolve this, to no avail. However, this should not be a big deal in practice, as you don't need the graphics to do the project. If you really need fast graphics for debugging and testing, there are a couple of different options: 1) use the VirtualBox/Vagrant combo with a Ubuntu Linux VM that was posted with practical 4, or 2) run the code on one of the FAS Unix machines (on which you'll need to submit the code anyway) such as `ice6.fas.harvard.edu`. SSH into the machine with X11 forwarding and the game seems to run reasonably fast even with the graphics going over the wire.

Can I have an extension? There are no extensions for the final project.

Changelog

This format for assignments is somewhat experimental and so we may need to tweak things slightly over time. In order to be transparent about this, a changelog is provided below.

- **v1.5** – 1 May 2014 at 20:30pm (added information regarding code submission)

- **v1.4** – 27 April 2014 at 16:00pm (fixes inconsistency in Table 1)
- **v1.3** – 23 April 2014 at 19:00pm (notes command-line options explicitly)
- **v1.2** – 22 April 2014 at 11:00pm (updated due dates)
- **v1.1** – 22 April 2014 at 11:00am (fixed indexing issue)
- **v1.0** – 11 April 2014 at 23:59pm