7CCSMML1 (Machine Learning)

Coursework 1

(Version 1.1)

1 Overview

For this coursework, you will have to implement a classifier. You will use this classifier in some code that has to make a decision. The code will be controlling Pacman, in the classic game, and the decision will be about how Pacman chooses to move. Your classifier probably won't help Pacman to make particularly good decisions (I will be surprised if it helps Pacman win games, my version certainly didn't), but that is not the point. The point is to write a classifier and use it.

No previous experience with Pacman (either in general, or with the specific UC Berkely AI implementation that we will use) is required.

This coursework is worth 10% of the marks for the module.

Note: Failure to follow submission instructions will result in a deduction of 10% of the marks you earn for this coursework.

2 Getting started

2.1 Start with Pacman

The Pacman code that we will be using for the coursework was developed at UC Berkeley for their AI course. The folk who developed this code then kindly made it available to everyone. The homepage for the Berkeley AI Pacman projects is here:

`http://ai.berkeley.edu/`

Note that we will **not** be doing any of their projects. Note also that the code only supports Python 2.7, so that is what we will use[1].

You should:

(a) Download:

pacman-cw1.zip

from KEATS.

(b) Save that file to your account at KCL (or to your own computer).

(c) Unzip the archive.

This will create a folder `pacman`

[1] If you insist on using Python 3, you are on your own in terms of support, and if the code you sbmit does not work (which is likely) you will lose marks.

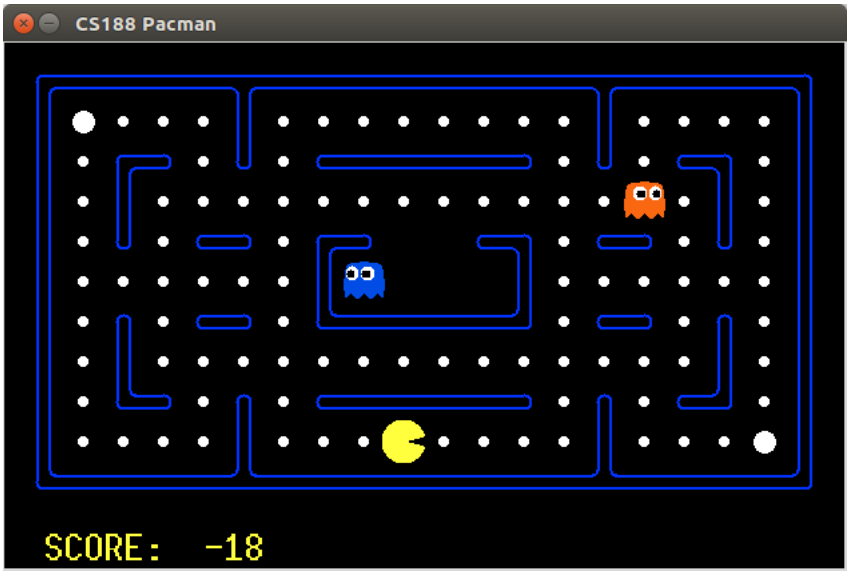

Figure 1: Pacman

(d) From the command line (you will need to use the command line in order to use the various options), switch to the folder `pacman`.

(e) Now type:

```
python pacman.py
```

This will open up a window that looks like that in Figure 1

(f) The default mode is for keyboard control, so you should be able to play this game out using the arrow keys.

Playing Pacman is not the object here — don't worry if there is an issue with controlling Pacman using the keys, that can happen on some platforms — but you will need to run this code to do the coursework. So, if the code causes an error, get help.

When you are tired of running Pacman, move on to the next section.

2.2 Code to control Pacman

Now we work towards controlling Pacman by writing code. The file `sampleAgents.py` contains several simple pieces of code for controlling Pacman. You can see one of these run by executing:

```
python pacman.py --pacman RandomAgent
```

This is not a good player (it is just picking from the available actions at random), but it shows you a couple of things.

First, you execute an agent that you write by using the `--pacman` option, followed by the name of a Python class. The Pacman code looks for this class in files called:

```
<something>Agents.py
```

and, when it finds the class, will compile the relevant class. If the class isn't in an appropriately named file, you will get the error:

```
Traceback (most recent call last):
  File "pacman.py", line 679, in <module>
    args = readCommand( sys.argv[1:] ) # Get game components based on input
  File "pacman.py", line 541, in readCommand
    pacmanType = loadAgent(options.pacman, noKeyboard)
  File "pacman.py", line 608, in loadAgent
    raise Exception('The agent ' + pacman + ' is not specified in any *Agents.py.')
```

Now open your favorite editor and look at `sampleAgents.py`. If you look at `RandomAgent` you will see that all it does is to define a function `getAction()` This function is the only thing that is required to control Pacman[2]. The function is called by the game every time that it needs to know what Pacman does — at every "tick" of the game clock — and what it needs to return is an action. That means returning expressions that the rest of the code can interpret to tell Pacman what to do.

In the basic Pacman game, `getAction()` returns commands like:

```
Directions.STOP
```

which tells Pacman to not move, or:

```
Directions.WEST
```

which tells Pacman to move towards the left side of its grid (North is up the grid).

However, for your coursework, you have to pass this direction to the function `api.makeMove()` first, just as the classes in `sampleAgents.py` do.

`sampleAgents.py` contains a second agent, `RandomishAgent`. Try running it. `RandomsishAgent` picks a random action and then keeps doing that as long as it can.

2.3 Towards a classifier

For this coursework you'll work from some skeleton code that is in the folder `pacman-cw1`. The file to look for is `classifierAgents.py` Two things to note about this:

(a) This defines a class `ClassifierAgent`. When we mark your coursework, we will do so by running this class. If it doesn't exist (because you decided to rename things), we will mark your code as if it doesn't work. So make life easy for yourself, and use the class provided as the basis for your code.

(b) This class provides some simple data handling. It reads data from a file called `good-moves.txt` and turns it into arrays `target` and `data` which are similar to the ones you have used with `scikit-learn`. When we test your code, it will have to be able to read data in the same format as `good-moves.txt`, from a file called `good-moves.txt`. If it doesn't, we will mark your code as not working. So make life easy for yourself and stick to the (admittedly, but intentionally, limited) data format that we have provided.

[2]Of course, controlling Pacman to do something well may require a number of functions in addition to `getAction()`.

To run the code in `classifierAgents.py`, you use:

```
python pacman.py --pacman ClassifierAgent
```

Note the difference in capitalisation between file name and class name.

Now open your editor and take a look at the code for ClassifierAgent. There are six functions in it (in order):

(a) `__init__()`

The constructor. Run when an instance of the class is created. Because the game doesn't exist at this point, it is of limited use.

(b) `convertToArray()`

This is a simple utility. The data in `good-moves.txt` is stored as a string. We need it as an array of integers. This does the conversion.

(c) `registerInitialState()`

This function gets run once the game has started up. Unlike `__init()__`, because the game has started, there is game state information available. Thus it is possible for Pacman to "look" at the world around it.

Right now this is the only function that is doing any real work. It opens the file `good-moves.txt`, and extracts the data from it. It then parses the data into the arrays `data` and `target`. These arrays are accessible from any function. (They are data members of the class `ClassifierAgent`.)

(d) `final()`

This function is run at the end of a game, when Pacman has either won or lost.

(e) `convertNumberToMove()`

Another simple utility. The data in `good-moves.txt` encodes moves that Pacman made in the past using integers. What you need to do is to produce moves of the form:

```
Directions.NORTH
```

since that is the format which the game engine requires. This function converts from one to the other in a way that respects the original conversion from moves to integers.

(f) `getAction()`

This function is called by the game engine every time step. What it returns controls what Pacman does. Right now it just returns `Directions.STOP`, so Pacman doesn't do anything. (The function also does some other stuff, but we will get to that later).

3 What you have to do (and what you aren't allowed to do)

3.1 Write some code

Your task in this coursework is to write a classifier which uses the data in `good-moves.txt` to control Pacman. By "control Pacman" we mean "select an action and return it in the function `getAction`. However, because this is a module on machine learning, not a module on game programming, we are quite prescriptive about how you go about doing this:

(a) Your code is only allowed limited access to information about the state of the game. What you are allowed to access is the information provided by:

```
api.getFeatureVector(state)
```

This returns a feature vector in the form of an array of 1s and 0s like this:

```
[1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

This records details of whether there are walls, food, and ghosts in the squares around Pacman. You don't need to know what each number means (though if you want to know, look in `api.py`). What you do need to know is that if your code uses any other information about the game to decide what to do, you won't get any marks for the coursework[3].

(b) Your code should use a classifier to make a decision, based on the information in `features`, to decide what to do. Thus the classifier should be trained using the information in `self.data` and `self.target`, and should predict an action when passed the data in `features`.

(c) You are allowed to use a classifier from an external library such as `scikit-learn`. However, if you use a classifier from an external library, you will not as many marks as if you write a classifier yourself. (For details on exactly how we will mark your code, see the marksheet on KEATS.)

(d) If you do code your own classifier, it does not have to be complicated. It could be as simple as a 1-nearest neighbour classifier. However, the more sophisticated the classifier, the more marks you will get. (Again, for details you should see the marksheet on KEATS.)

(e) To get full marks, your code has to run until either Pacman wins a game, or until Pacman gets eaten by a ghost (and loses a game). In other words, your code should not crash or otherwise fail while we are running it. Losing a game is not failing. In fact, from the point of view of marking, we don't care if your Pacman wins, loses, gets a high score or a low score. We only care that your code successfully use a classifier to decide what to do.

3.2 Things to know

If you look in `good-moves.txt`, you will see that each line contains a feature vector like the one above, plus a final digit. (There are no brackets or commas, that is because `good-moves.txt` holds strings not arrays.) The first digits are indeed a feature vector, and the last digit encodes an action. When the data is read in by `registerInitialState`, the feature vector part is loaded into `data`, and the "action" is loaded into `target` such that the ith elements of `data` and `target` go together. The data was collected from code that played Pacman. (Indeed, from some code that won games of Pacman.) At each step, the feature vector and move were stored in `good-moves.txt`. And that is exactly why you can create a classifier from it. If you train a classifier on the `good-moves` data, then that classifier should be able to predict a sensible move given a new feature vector.

Note that while the `good-moves` data is what we will test your code with (or rather it is one of the things we will test your code with), you may want to create some custom training data. To make that easy, we have provided `TraceAgent` (in the file `traceagents.py`). If you run this using:

[3] Ok, that is not quite right. The code in `getAction` in the skeleton `classifierAgent` uses `legal = api.legalActions(state)` to get a set of the legal moves at every step. That is techncially information about the game state, and it is both allowed, and sensible, since if you return an illegal action to the game engine, the game crashes. Using any other information is, however, forbidden.

```
python pacman.py --p TraceAgent
```

you will get the same keyboard controlled Pacman as you saw before, BUT one which outputs data on your move and the corresponding feature vector. This data is written to `moves.txt`. (If a file already exists with that name, it is over-written, so be careful.)

3.3 Limitations

There are some limitations on what you can submit.

(a) Your code should be in Python 2.7.

Code written in a language other than Python will not be marked.

Code written in Python 3.X is unlikely to run with the clean copy of `pacman-cw2` that we will test it against. If it doesn't run, you will lose marks.

The reason for this is that we do not have the resources to deal with code written in multiple languages, and to ensure that we can run code written in Python 3.X.

(b) Your code will be tested in the same environment as we have been using in the lab. (this is the same environment for which the installation instructions are given in the instructions for the first two practicals.) That is the standard Ananconda Python distribution, with `scikit-learn` also installed (and the `scikit-learn` distribution includes `numpy`). Code using libraries that are not in this collection *may* not run when we test it. If you choose to use such libraries and your code does not run when we test it, you will lose marks.

The reason for this is that we do not have the resources to deal with setting up arbitrarily complex environments (with the possibility of libraries with arcane interactions) for every submission.

(c) Your code must only interact with the Pacman environment by making calls through the version of `api.py` supplied in `pacman-cw1.zip` (Version 6). Code that finds other ways to access information about the environment will lose marks.

The idea here is to have everyone solve the same task.

(d) You are not allowed to modify any of the files in `pacman-cw1.zip` except `classifierAgents.py`.

Similar to the previous point, the idea is that everyone solves the same problem — you can't change the problem by modifying the base code that runs the Pacman environment. Also, your code will ahve to run against a clean version of the code in `pacman-cw1` so you'll just be making trouble for yourself.

(e) You are not allowed to copy, without credit, code that you might get from other students or find lying around on the Internet. (This includes the use of code that was distributed as part of the module — if you use code from files other than `classifierAgent.py` without attribution, we will consider that to be plagiarism.) We will be checking.

This is the usual plagiarism statement. When you submit work to be marked, you should only seek to get credit for work you have done yourself. When the work you are submitting is code, you can use code that other people wrote, but you have to say clearly that the other person wrote it — you do that by putting in a comment that says who wrote it. That way we can adjust your mark to take account of the work that you didn't do. Please add any citations, descriptions, or whatever you want us to know in the python file.

(f) Your code must be based on using a classifier on the data in `good-moves.txt`. If you don't submit a program that contains a recognisable classifier, you will lose marks.

4 What you have to hand in

Your submission should consist of a single ZIP file. (KEATS will be configured to only accept a single file.) This ZIP file must include a single Python file (your code).

The ZIP file must be named:

`cw1-<lastname>-<firstname>.zip`

so my ZIP file would be named `cw1-parsons-simon.zip`.

Remember that we are going to evaluate your code by running your code by using variations on

```
python pacman.py -p ClassifierAgent
```

and we will do this in a vanilla copy of the `pacman-cw1` folder, so the base class for your agent must be called `ClassifierAgent`.

To streamline the marking of the coursework, you must put all your code in one file, and this file must be called `classifierAgents.py`,

Do not just include the whole `pacman-cw1` folder. You should only include the one file that includes the code you have written.

Submissions that do not follow these instructions will lose marks.

5 How your work will be marked

There will be three main components of the mark for your work:

(a) Functionality

We will test your code by running your `.py` file against a clean copy of `pacman-cw1`.

As discussed above, for full marks for functionality, your code is required to run when we invoke the command:

```
python pacman.py --p ClassifierAgent
```

and run until the game is won or lost. Code that fails to meet these requirements will lose marks.

We will also look at your code for evidence of the use of a classifier. Code that does not use a classifier will lose marks. Code that does not implement a classifier (that is, uses one from an external library like `scikit-learn`) will lose marks.

Code that implements more sophisticated classifiers will get more marks. So, my example (above) of using a 1-NN classifier, which is about the simplest possible classifier, would not get as many marks as the implementation of a more sophisticated classifier.

(b) Style

There are no particular requirements on the way that your code is structured, but it should follow standard good practice in software development and will be marked accordingly.

Remember that your code is only allowed to interact with the Pacman environment through Version 6 of `api.py` (the version in `pacman-cw1`), and is only allowed to use the environment information provided by the fucntions `getFeatureVector()` and +legalActions()+ Code that does not follow this rule will lose marks.

(c) Documentation

All good code is well documented, and your work will be partly assessed by the comments you provide in your code. If we cannot understand from the comments what your code does, then you will lose marks.

A copy of the marksheet, which shows the distribution of marks across the different elements of the coursework, will be available from KEATS.

Version list

- Version 1.0, January 28th 2018
- Version 1.1, January 11th 2021