Anthony Poerio

[adp59@pitt.edu](mailto:adp59@pitt.edu)

University of Pittsburgh

CS1571 – Artificial Intelligence

Homework #04

Linear Regression – Report

**Project Overview**

For our fourth assignment in Dr. Hwa’s AI class, our goal was ….

**Part I – Univariate Regression**

Chose alpha by trial and error. At first, I was using 0.8 and then 0.7… etc… I made it progressively lower because at first I was diverging really quickly. And I suppose that’s because with so many values… we get large ones at first…. And gets us off on the wrong track… Threshold of 0.00001 because if it doesn’t change by more than that we are more or less stable… Highly likely to have found the global minimum or very close to it….

**Part II – Multivariate Regression**

Am doing some randomization, so may get an assertion error. If data sets aren’t divided evenly (expected to happen on occasion). If it does, please re-run.

Also, values may change from run to run.

Probably want to do like 10, and take average

Method 1:

Simply take the data set as is. Don’t do anything special to it.

The AVERAGE Squared Error over the Entire Testing Data Set = 17.8970380912

The AVERAGE Overall Error For \_\_any given prediction\_\_ = 4.2304891078

🡪 not bad

Method 2:

Academic Vectors only 🡪 irrespective of age. Like pure academic talent

The AVERAGE Squared Error over the Entire Testing Data Set = 15.1453370033

The AVERAGE Overall Error For \_\_any given prediction\_\_ = 3.89170104238

🡪 small improvement over the whole set

Method 3:

Personal Life Vectors only

Slightly worse

Method 4:

Parental Vectors only, and student age

Seems to work a little better than the pure academic indicators….

Lots of other stuff is really dictated by these factors 🡪 student’s age and parent’s academic achievement, because the parents dictate how the student behaves and studies, especially when they are younger. So the other variables are affected by these but seem to actually introduce more noise because of added complexity

Method 5:

Health & Relationship vectors 🡪 works arguably the best so far. Which is surprising to some extent, but makes logical sense. Can’t perform well in class if you’re unhealthy and can’t make it to class, have trouble at home, and so on.

Method 6:

Academics, Age, And Time

Not the best, about average

Method 7 (Control) 🡪 Random Guessing To Create Parameters:

Much worse.. Shows that we are converging on something useful.

Method 8 (Control) 🡪 Random Guessing Directly, between 0 and 20

Again, much worse… though a little better than guessing to create the parameters

Conclusion:

We can improve predictions by using fewer data points.. but not able to find a definitive best combination from my testing. There is, however, definitely some noise in this feature set. Goal should be to remove it.

Also likely possible to improve through more complicated means… but wasn’t able to find a good method in my testing.

Overall:

The predictions generally work reasonably well, though they aren’t ideal. However, it is a very strong improvement over random guessing.

The version of **Flappy Bird** that we were tasked with updating a is a Python implementation created by Timo Wilken and available for download directly at: <https://github.com/TimoWilken/flappy-bird-pygame>. This Flappy Bird version is implemented using the **PyGame** library.

More specifically, **Flappy Bird** is simple computer game in which the use controls a small bird, and at any given time is given two choices: Stay, or Jump. The user must select the best option at any given time in order to pass obstacles in the bird’s path, thereby scoring points.

Our AI is tasked with playing Flappy Bird automatically. This write-up documents the decisions I made in programming my Flappy AI.

**Framework Approach**

The first challenge I had in implementing the framework for my Flappy AI was determining exactly how the game worked, in its original state.

By using the debugger and stepping through the game’s code during some trial runs, I was able to figure out where key decisions where made, how data flowed into the game, and exactly where I would need to position my agent.

At its basic level, I created an “Agent” class, and passed that class into the running game code. Then, at each loop of the game, I examined the variables available to me, and then passed a ‘MOUSEBUTTONUP’ command to the PyGame event queue whenever the AI decided to jump. Otherwise, I did nothing.

**State Representations**

From there, the next step was determining a way to model the problem. I decided to use follow the basic guidelines [outlined by Sarvagya Vaish, here](http://sarvagyavaish.github.io/FlappyBirdRL/).

First, I discretized the space in which the bird sat, relative to the next pipe. I was able to get pipe data by accessing the **pipe object** in the original game code. Similarly, I was able to get bird data by accessing the **bird object**.

From there, I could determine the location of the **bird** and the **pipes** relative to each other. I discretized this space as a 16x16 grid, with the following parameters:

# first value in state tuple  
height\_category = 0  
dist\_to\_pipe\_bottom = pipe\_bottom - bird.y  
**if** dist\_to\_pipe\_bottom < 8: # very close  
 height\_category = 0  
**elif** dist\_to\_pipe\_bottom < 20: # close  
 height\_category = 1  
**elif** dist\_to\_pipe\_bottom < 125: #mid  
 height\_category = 2  
**elif** dist\_to\_pipe\_bottom < 250: # far  
 height\_category = 3  
**else**:  
 height\_category = 4  
  
# second value in state tuple  
dist\_category = 0  
dist\_to\_pipe\_horz = pp.x - bird.x  
**if** dist\_to\_pipe\_horz < 8: # very close  
 dist\_category = 0  
**elif** dist\_to\_pipe\_horz < 20: # close  
 dist\_category = 1  
**elif** dist\_to\_pipe\_horz < 125: # mid   
 dist\_category = 2  
**elif** dist\_to\_pipe\_horz < 250: # far  
 dist\_category = 3  
**else**:  
 dist\_category = 4

Using this methodology, I created a **state tuple** that looked like this:

(**height\_category={0,1,2,3,4}, dist\_category={0,1,2,3,4} , collision=True/False)**

Then, each iteration of the game loop, I was able to determine the bird’s relative position, and whether it had made a collision with the pipes or not.

If there was no collision, I issued a reward of +1.

If there was a collision, I issued a reward of -1000.

I tried many different state representations here, but mostly it was matter of determining an optimal number of grid spaces and the right parameters for those spaces.

Initially, I started with a 9x9 grid, but moved to 16x16 because I got to a point in 9x9 where I just couldn’t make any more learning progress.

Very generally, we want to have a ***tighter grid around the pipes***, as this is where most collisions happen. And we want a ***looser grid as we move outwards***. This seemed to give me the best results, as we need different strategies at different locations on the grid.

**Exploration Approaches**

My next task was implementing an exploration approach.

Because we have only two choices at any given state (JUMP—or—STAY), implementing exploration was relatively simple.

I started out with a high **exploration factor** (I used **1/time\_value+1**), and then I generated a random number between [0,1). If the random number was less than the exploration factor, then I explored.

Over time the exploration factor got lower, and therefore the AI explored less frequently.

Exploration essentially consisted of flipping a fair coin (generating a Boolean value randomly).

* If true🡪 then I chose to JUMP.
* If false🡪 I chose to STAY.

The main problem I encountered with this method is that the exploration factor was very at the beginning, and sometimes choices were made that were not representative of actual situations that the bird would encounter in ‘true’ gameplay.

BUT, because these decisions were made earlier, they were weighted more heavily in the overall Q-Learning algorithm.

This isn’t ideal, but exploration is necessary, and overall the algorithm works well. So it wasn’t a large problem, overall.

**Learning Rates & Their Impact**

The first learning rate I tried was **alpha=(1/time+1).** However, this gave very poor results in practice.

This is because time is NOT the most important factor in determining a strategy from any given state. Rather, it is how many times we’ve been to that state.

The problem is that we make extremely poor choices at the beginning of the game (because we simply don’t know any better). But with **alpha=(1/time+1)**, the results of these these poor choices are weighted the most highly.

Once I changed the learning factor to **alpha=1/N(s,a),** I immediately saw *dramatically better* results. (That is, where N(s,a) tracks how many times we’ve been in a given state and performed the same action.)

**Training**

My final, “Smart” bird is the result of about **4 hours of training**.

I don’t actually think there would be a way to make the training more efficient, aside from speeding up the gameplay in some way.

Overall, I the results I received from the investment of time I put it in reasonable.

Given more time, I would probably ***discretize the space even more finely*** (maybe a 25x25 or 36x36 grid) – so that I could find even more optimal strategies from a more fine-tuned set of positions in the game-space.

**How to Use my Smart Bird**

To use my smart bird, simply take the following steps:

1. cd into a directory containing my source code
2. Ensure that this directory includes the file named ‘**qdata.txt**’
3. Run the command:
   1. **python flappybird.py “qdata.txt”**
4. Watch Flappy crush it. (the game will run 10x)

**Citations**

I consulted the following resources to implement my AI:

* **Implementation ideas** based on discussion here: <http://sarvagyavaish.github.io/FlappyBirdRL/>
* **Code** created from Dr. Hwa’s Pseudo-Code here: <https://docs.google.com/document/d/1r1X8IvDLYkk1ztHNbcZsd1LfP2gqP4AZvivKI509G7Q/edit>
* **For understanding what I was doing better:** <http://mnemstudio.org/path-finding-q-learning-tutorial.htm>