Andy Taylor

Software - Part B

SYSTEM SPECS

Software

- MacOS 10.13.6+
- Python3.6

Hardware

- 1440x900 Minimum Resolution
- 4GB RAM Recommended
- 465 MB of Storage Space
- Basic input devices
    - Mouse
    - Keyboard

BUILD LOG

See Here

USER MANUAL

Installation Instructions

Easy Install

~~Simply double-click on the provided application~~

~~*Tested on my brother's computer, though I know my track record here isn't great :)*~~

Estimated 10 hrs to upload this, sorry cmd line will have to do

Alternative Install

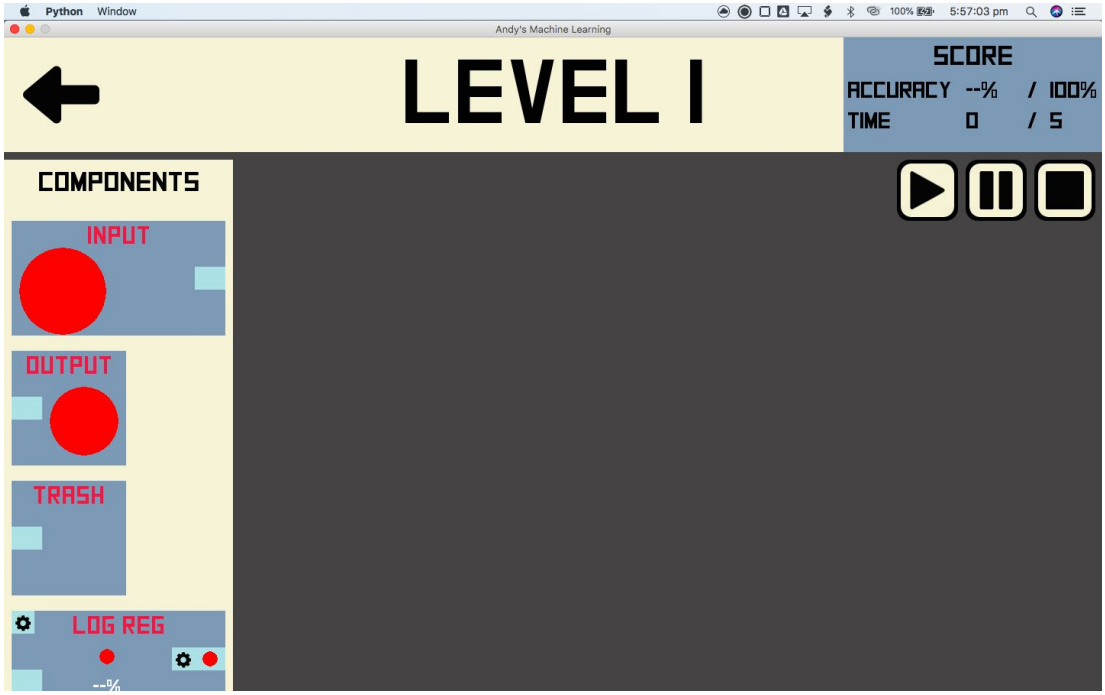
Navigate to the project's directory in the command line

Enter into the command line

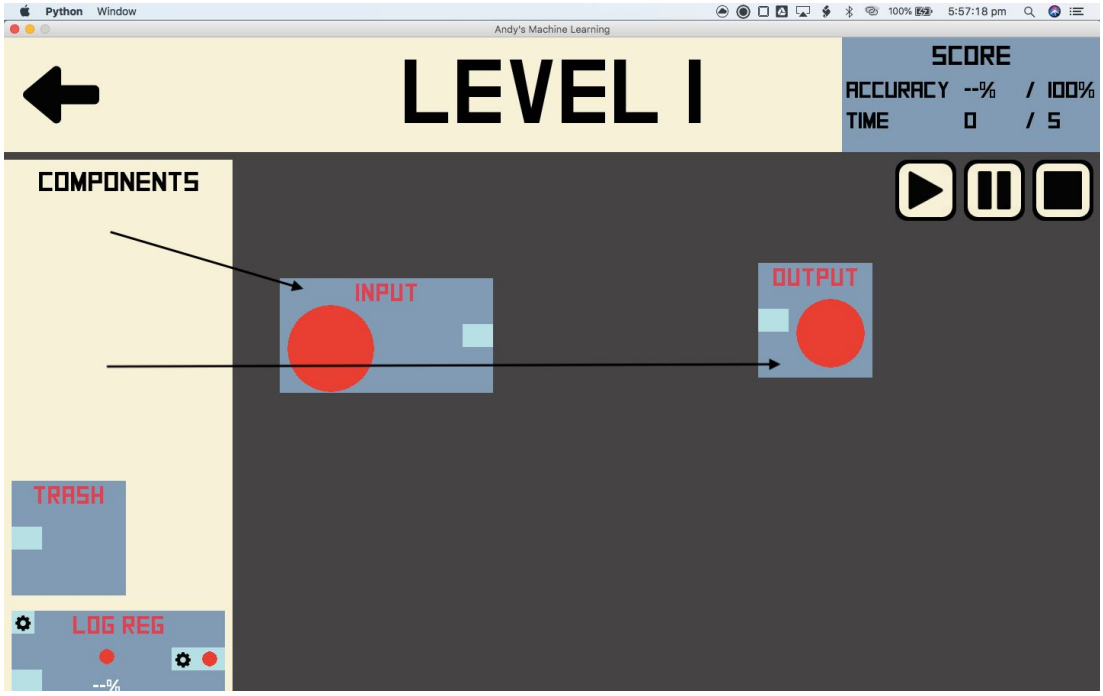
python3 main.py

Basic Gameplay

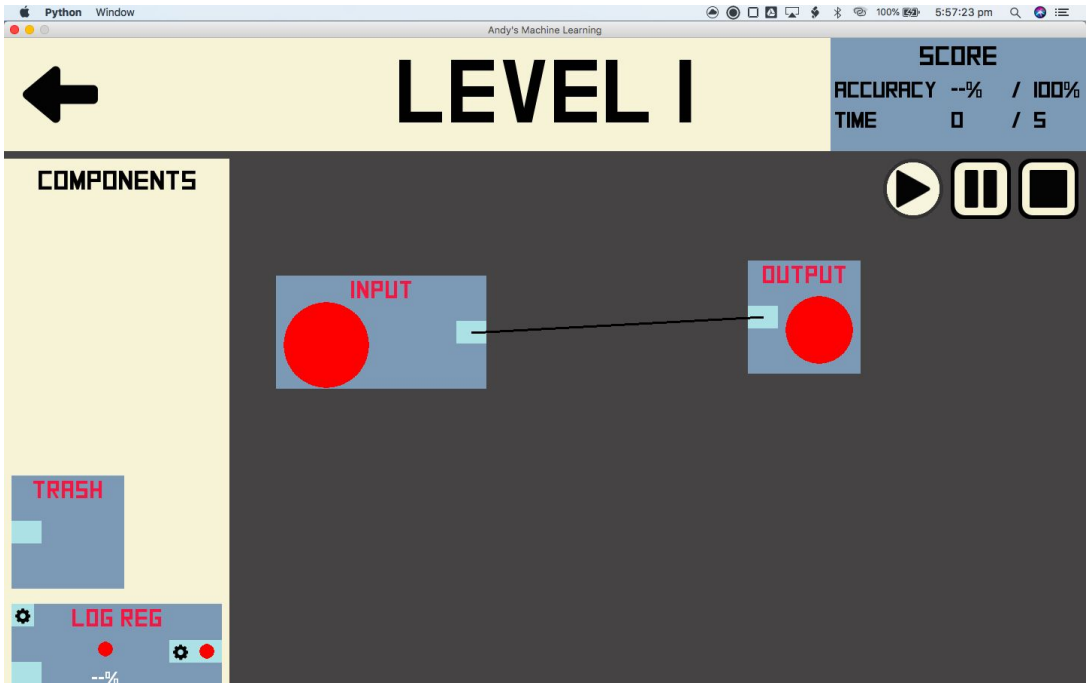
Upon entering a level, you will be greeted by a screen similar to this one



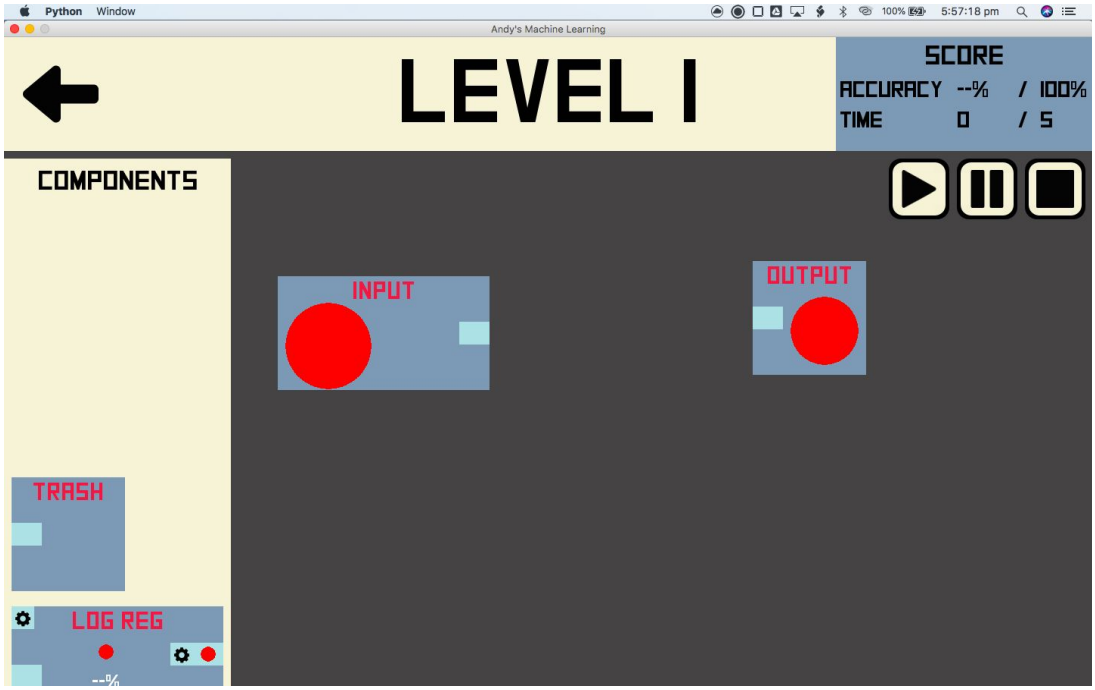
You can drag components into the workspace with the mouse



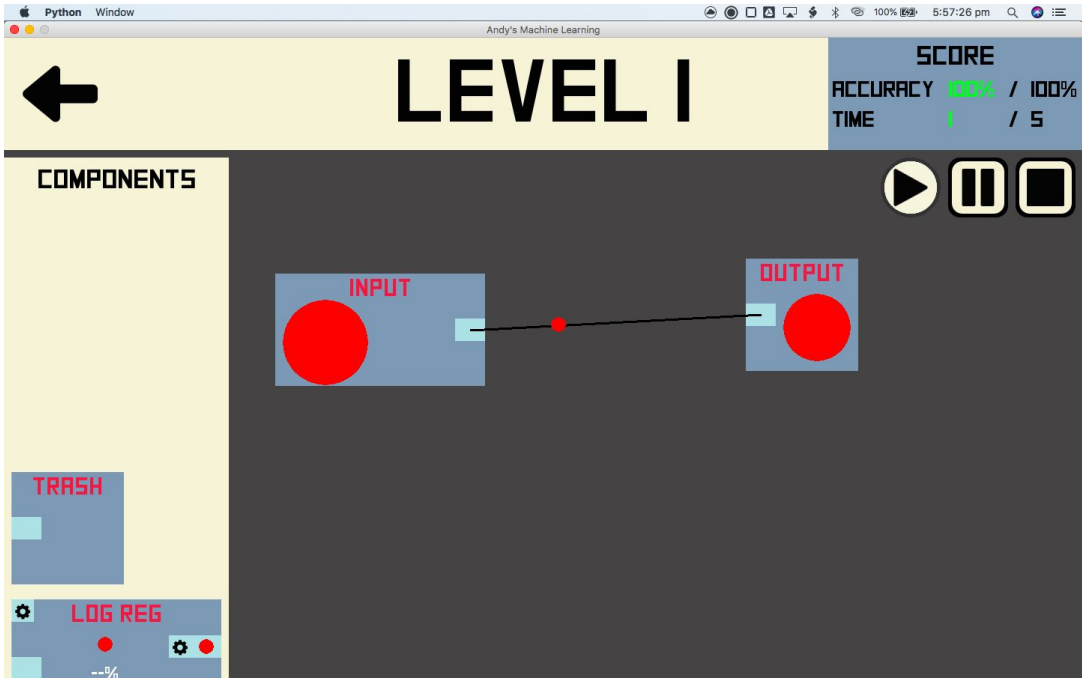
By dragging the mouse between holders, you can create connections



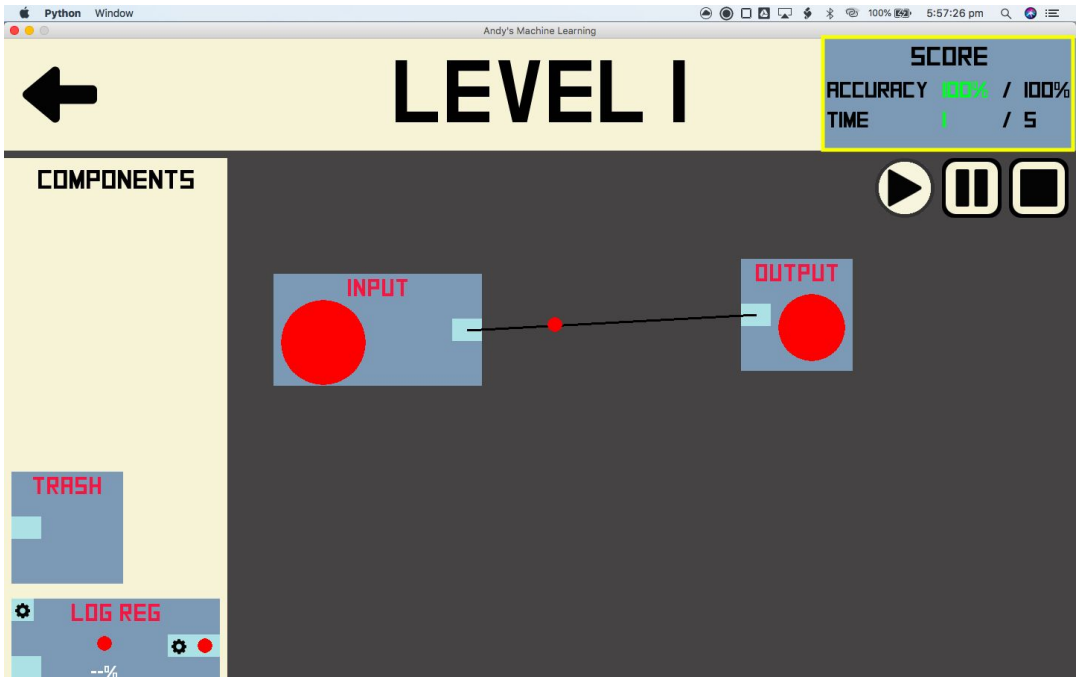
To delete a connection, simply right click on the holder that the connection is connected to
Note: this will disconnect everything from that holder



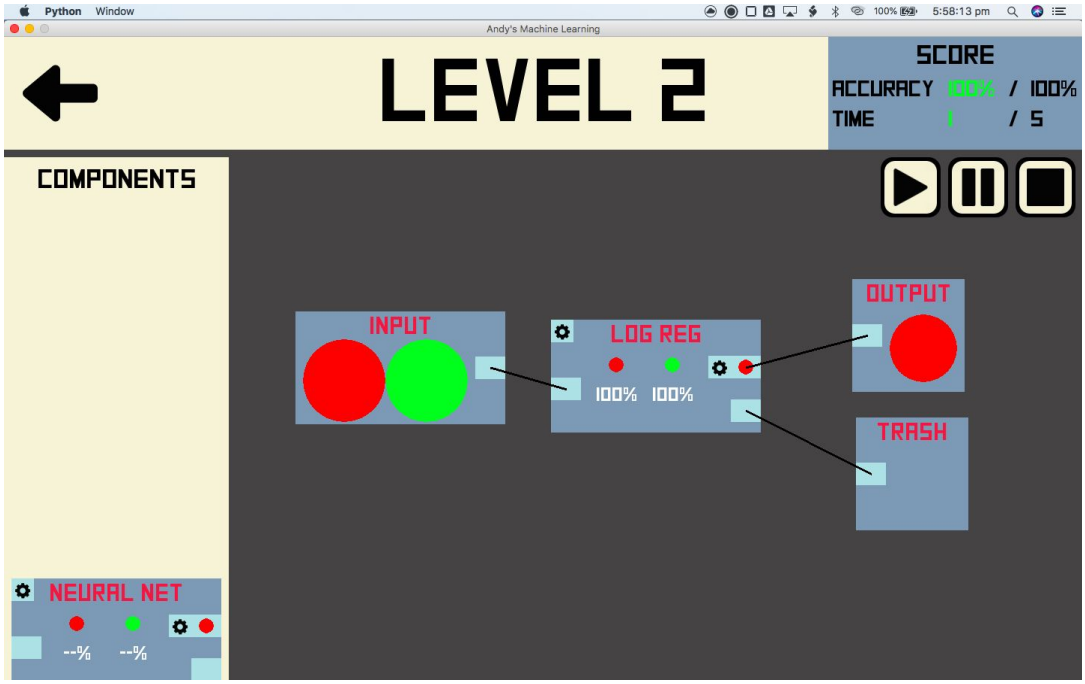
When you are satisfied with your pipeline, you can select the play button and data will begin to flow



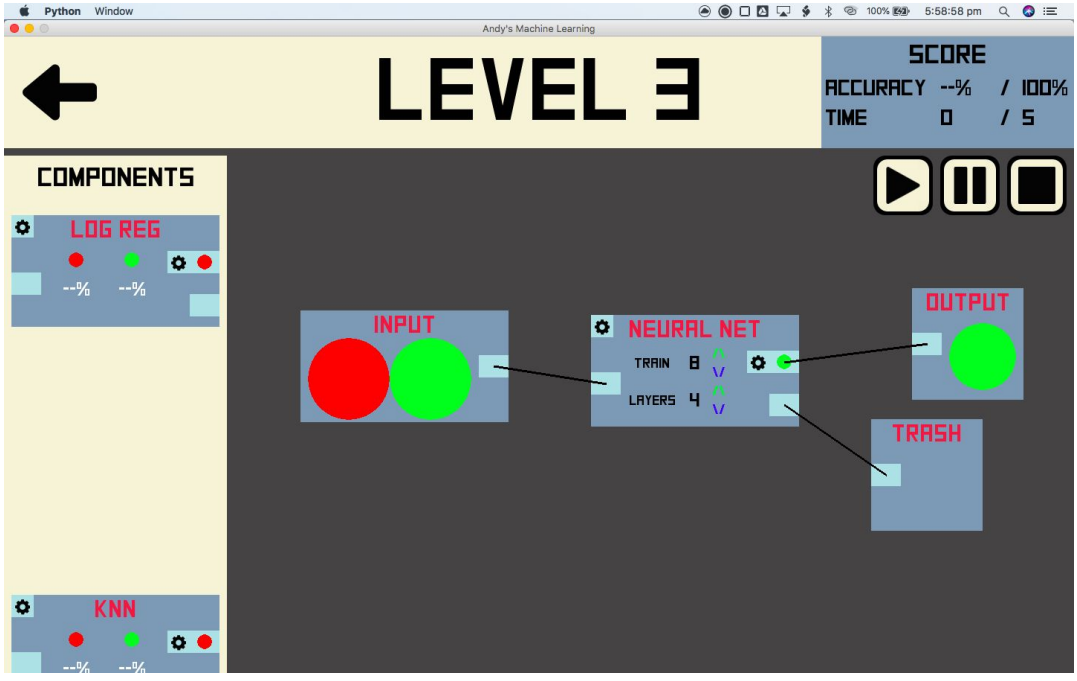
To pass a level, you must achieve a certain accuracy within a certain timeframe



In further levels, you will have to use algorithms to sort out the data



These algorithms can be fine tuned by pressing the config button at the top left of the algorithm



The Shop

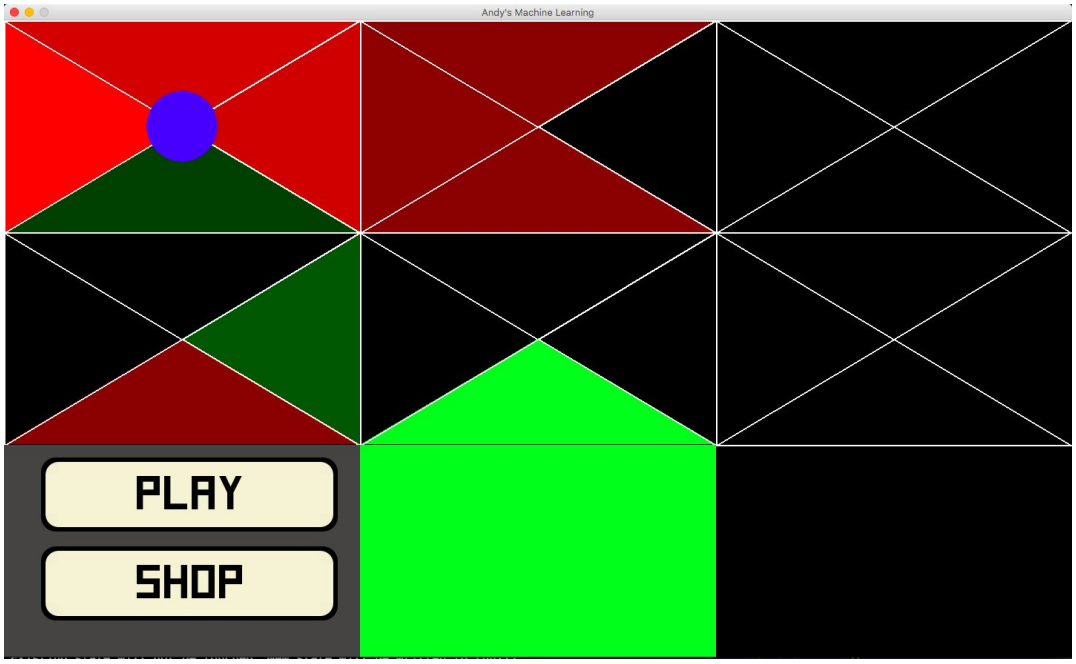
Algorithms can be purchased once and will remain unlocked

Upgrades can be purchased numerous times and will stack with each other

Main menu additions are purely cosmetic, and offer you another avenue to spend your hard earned cash

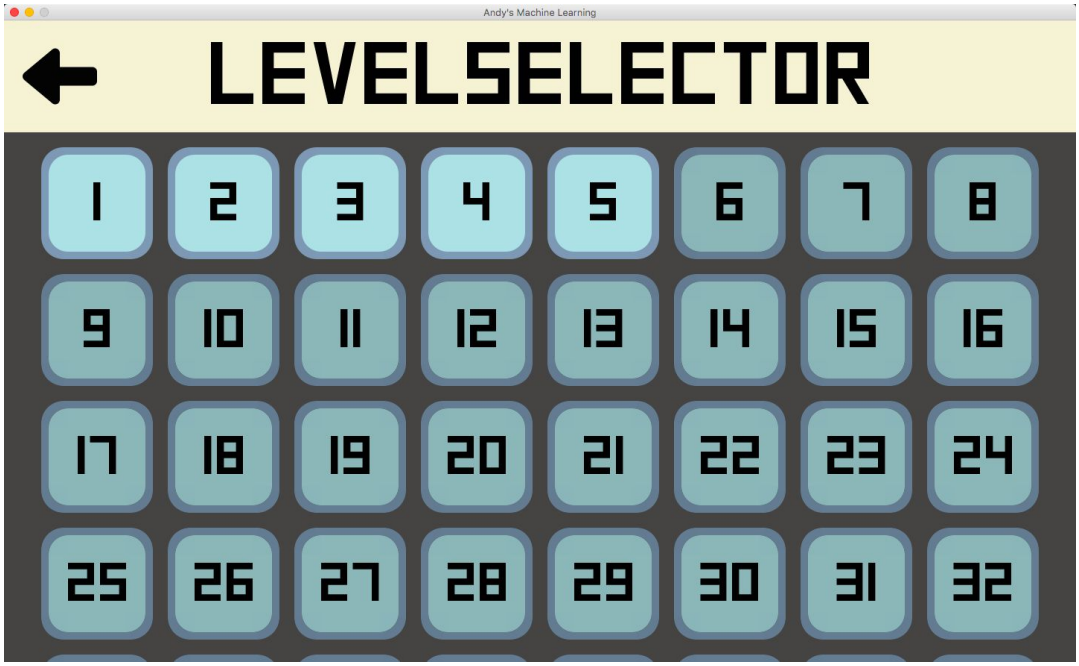
Main Menu

This is *view only* (excluding the 2 buttons) and illustrates some basic reinforcement learning



Level Selector

Here you can either select a level or navigate back to the main menu



Troubleshooting

I can't complete a level and don't have enough money to purchase upgrades

You can replay levels to earn a small amount of money

I want to delete a connection

Simply right-click on the holder to delete all connections to that holder

Why can't I configure this algorithm?

Configurations must be unlocked through the upgrades section in the shop

The level won't finish

Make sure that any empty holders are connected to the trash, all algorithms must be empty for the level to finish!

FEATURES OF THE APPLICATION

Plug 'n' Play Algorithms

Each of the 4 algorithms can be interchanged simply by dragging and dropping them into the workspace and connecting them to other components.

This ease of use is extremely important in making ML accessible to users, but has to be managed carefully when developing code. In particular each should have common methods, such as
__init__(self, num_features, ...)
train(self, X, y)
predict(self, X)
which all take the same parameters, and return similar values

This means that KNN has a constructor with a blank parameter, as it doesn't need to know the number of features.
It also means that all *predict* functions will take and return matrices*, meaning that KNN and bayes require extra loops since they usually predict for single samples at a time.

* This is mostly because I'd implemented it that way first in the logistic regression

A Shop and Currency System

The shop is important for two reasons, 1) it adds to the depth of the game and 2) it allows the user to progressively gain access to more complex features (such as neural network layers).

The shop is accessible from the main menu and exists in its own screen. Within the shop, there are 3 sections for ‘algorithms’, ‘upgrades’ and ‘main menu’ additions.

Algorithms are one time purchases that can then be used to help solve levels, with each algorithm having certain strengths and weaknesses.
Upgrades can be purchased multiple times and affect the configuration options of different algorithms.
Main Menu additions pretty much just look cool and offer the user another way to spend their hard earned cash.

Currency in the game is gained by completing levels. $500 is gained for completing a level for the first time and $250 is earned for replaying a level.

Active Main Menu

The main menu is a demonstration of a tabular Q Learning algorithm on a small gridded world. This serves a few key purposes.

1. It helps expand the user’s understanding of what ML can do further than just classification algorithms, which are what the levels focus on as they’re most easily ‘gamified’
    a. Concretely, turning that demonstration into a game would be very ineffective as it requires almost no interaction. More in reflection.
2. It adds another way for the user to spend in game currency which motivates them to continue playing, and hence educating themselves
3. It adds to the general appeal of the game by replacing an otherwise extremely bland background

PEER REPORT

Report on my code - Chester

Report on Chester’s code

TEST DATA

Machine Learning Algorithms

These algorithms were tested primarily on 2 different data sets, each for a different purpose.

The Donut Data Set
They were tested on the donut data set, which generates a series of points each with an x and y between 0 and 1. These were plotted, then those that fell within a circle were given a label of 0, and those that fell outside the circle were labelled 1.

This dataset allows for clear visualisations to ensure the algorithm was learning properly. This is important because sometimes the accuracy of the algorithm drops, even though it is actually improving.
Red dots have a label of 0
Blue dots have a label of 1
Black dots are dots the algorithm incorrectly classified

The MNIST Handwritten Digit Dataset
This dataset consists of 28x28 pixel grayscale images of handwritten digits. The algorithm has to try to correctly correlate each drawing with the digit it depicts.

This dataset is much harder to visualise, but tests that the algorithm is able to learn more complex functions. They are assessed by measuring their cost function*, if applicable, and their accuracy on the data.

*Cross entropy loss function for neural nets and logistic regression

Score Display

The score consists of a play time and an accuracy component. Each of these was tested using stubs and flags to ensure that the algorithm correctly determines whether the play time is acceptable and whether the accuracy is acceptable.

This ensures that the colour of the score displays correctly and that the user is correctly declared as a winner / loser.

JUSTIFICATION OF CODING

There's a lot I want to justify, and about as much that I couldn't justify no matter how hard I tried, but I'll stick to looking at code style and some of my modular structure decisions.

Code Style

The code style is almost entirely consistent with the flake8 standard, with a few exceptions.

1. I've used wildcard imports in __init__ files as, in this project, they always import everything from their directory
2. I've allowed long strings to exceed the character limit, which are soft wrapped

```
1    from .supervised import *  # noqa
2    from .reinforcement import *  # noqa
3
4    from .reinforcement.QLearn import QLearn  # noqa
```

```
37                  {
38                      'name': 'Naive Bayes',
39                      'cost': 0,
40                      'blurb': 'Fast statistical classifier that models data using gaussian
                            distributions'  # noqa
41                  }, {
42                      'name': 'Logistic Regression',
43                      'cost': 1000,
44                      'blurb': 'Creates approximate functions to create estimates'
45                  }, {
46                      'name': 'KNN',
47                      'cost': 1500,
48                      'blurb': 'Lazy classifier that finds similar examples that it was shown during
                            training'  # noqa
```

Both variables and functions are named in snake_case. I came to this decision when I couldn't find a definitive answer on what the accepted practice was for *Python* specifically, so I decided to try it this way for a change.

Classes are titled PascalCase which is a common standard.

```
def on_init(self):
    for widget in self.widgets:
        widget.on_init()

def on_shutdown(self):
    for widget in self.widgets:
        widget.on_shutdown()
```

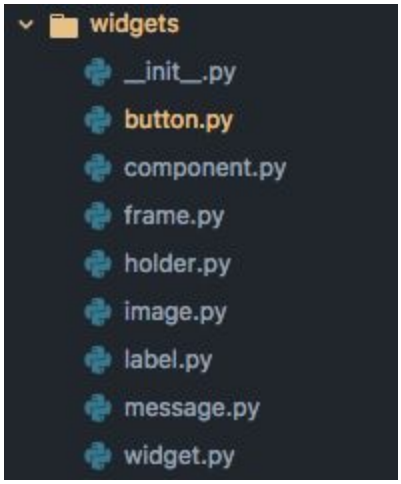
Modular Structure

Widgets

This wide hierarchical structure allows each widget to be entirely different to others and encourages the building of more complex ‘one-size-fits-all’ solutions, as opposed to having a large amount of slightly different classes.

The Frame widget is special as it allows widgets to be contained recursively, and hence very easily managed.

This is appropriate for a UI as the small amount of classes are more easily managed and the Frames allow for very versatile screen designs.



Screens

Screens act as an interface between the UI, provided by widgets, and the flow of execution, provided by the StateAppRunner. They also implement any special logic required for that screen, which is particularly important for the Level.

REFLECTION

The User Interface System

I'm not afraid to admit I severely underestimated how much time the user interface would take to put together. The biggest issue was that I went into it with literally no planning and just tried to get results on the screen quickly. While this method has stung me before, I can usually recover enough to get the final 'working' product down. This time, I ended up re-writing the entire UI System twice because the first 2 attempts simply wouldn't have been able to produce a good quality interface. My biggest issue is I was thinking too Object Oriented, I kept trying to inherit from class after class until I lost track of what was happening. As mentioned in the section above, this system has a very different approach which I found more difficult to implement but much more effective.

Reinforcement Learning

I'd written in part A that I really wanted to include some reinforcement learning algorithms in this, but I've been unable to for 2 reasons.

1. I actually found it quite difficult to 'gamify' ML algorithms, which I'll discuss more below. While I think I came up with something for the classification algorithms present in the game, I still don't have a way to incorporate these RL algorithms in the game effectively.
2. I ran out of time to find a solution to the above

The key difficulty with RL algorithms is that they're even less interactive than classification algorithms. As such, I'm pretty happy with my compromise of including a 'demonstration' of some reinforcement learning as the background for the main menu.

Gamifying Machine Learning

When I started this project, the number one goal was to "make machine learning simple".

On the bright side, I think I've certainly achieved that. Hundreds of lines of code with maths I hardly get is hidden behind a small blue box, with a single input and 2 outputs. There are a couple of configuration settings, but even they hide a certain level of complexity[1].

The issue is I think I've oversimplified it. This has two key impacts.

1. By so greatly abstracting the proper hyperparameters and mechanics of each algorithm, I impede the user from ever being able to use the full capabilities of that algorithm
    a. This in turn limits the datasets I can include in the program

2. This abstraction also prevents the user from seeing the training process in action, which seriously impacts their appreciation for what these algorithms are doing.
    a. For example, the perfect classification algorithm would look exactly like a CASEWHERE, which is not very impressive. It's only by appreciating how the algorithm does what it does that you can appreciate what it's capable of.

One other challenge I came across while gamifying these algorithms is that it's literally never necessary to use more than 1 algorithm. This confused me, because the similar game I referenced, While True: learn(), has pipelines with > 10 algorithms. What I hadn't fully considered was that their algorithms are actually just IF/ELSE statements, and they purposely limit them to only 2 different outputs. In reality, these algorithms are capable of filtering between many more classes. To combat this, I attempted to focus on the nature of the data and the speed of the algorithm to distinguish them between levels and algorithms, as opposed to the amount of different types of data.

The final and probably biggest limitation was due to training times. Training a neural network on an environment as simple as the Donut takes 2-3 minutes, which is an unrealistic amount of time to make the user wait. This also makes more complex tasks such as digit recognition impossible within the space of the game. I was unable to come up with a solution overcome this issue.

[1]
1 'Train' actually corresponds to 200 training iterations for that algorithm
1 'Layer' automatically generates nodes and a 'relu' activation to create non-linearities in the neural net