Perceptron Learning – Path Finding – Is there a path?

Traditionally when we are faced with the task of determining whether a given maze has a path or not we resort to various search or path finding algorithms which are able to answer this question with absolute certainty. In this assignment, however, we will take a different approach to solve this problem by building a classifier that determines (to a certain degree) whether a given maze has a path or not.

Files included in this assignment

perceptron_classifier.py -- implementation of the perceptron classifier (from the textbook).

feature_extraction.py -- script for extracting features from any given maze.

training_phase.py -- script for training your model.

model_evaluation.py – script for evaluating how well your model did.

training_set_positives.p -- contains 150 mazes that have a path. Each training example in this file has a label of +1.

training_set_negatives.p -- contains 150 mazes that do not have a path. Each training example in this file has a label of -1.

test_set_negatives.p -- contains mazes that do not have a path. You will use these to evaluate your model.

test_set_positives.p -- contains mazes that do have a path. You will also use these to evaluate your model.

Note

All files with a (.p) extension are in pickle format. Details on how to open these files will be provided below.

The setup for this problem is as follows:

- Your training set has 300 mazes in total
- Each training example is a 2 dimensional array where the green squares are represented as zeros and the black squares are represented as ones (see the figures below). For instance, the first three rows of the maze in Figure 1.1 are represented by the following array, [[0,0,0,1,0,1,0,0], [0,1,0,0,0,1,0,0], [0,0,1,1,0,0,1,1], …]
- Your training set consists of two dictionaries (one for the positive examples and the other for the negative examples) that have these 2 dimensional arrays as their values.

In order to open the files which are stored in pickle format, you will use the pickle module as follows:

```
import cPickle as pickle

train_positives = pickle.load(open('training_set_positives.p', 'rb'))
```

train_positives is a dictionary with the training examples that have a path

The same logic can be applied to the negative examples as well.

Below are two visualized instances of your training set.

Figure 1.1

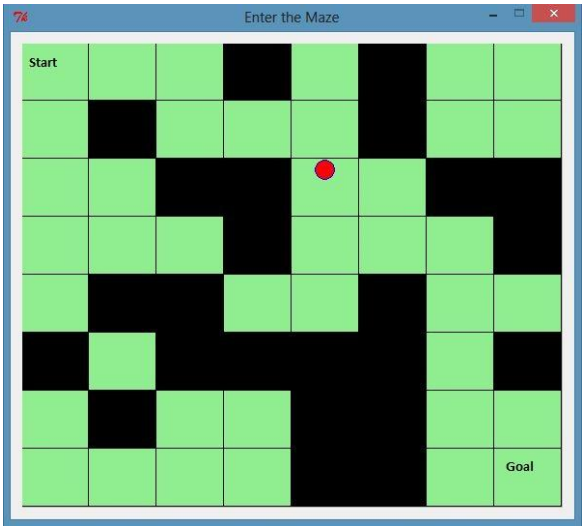
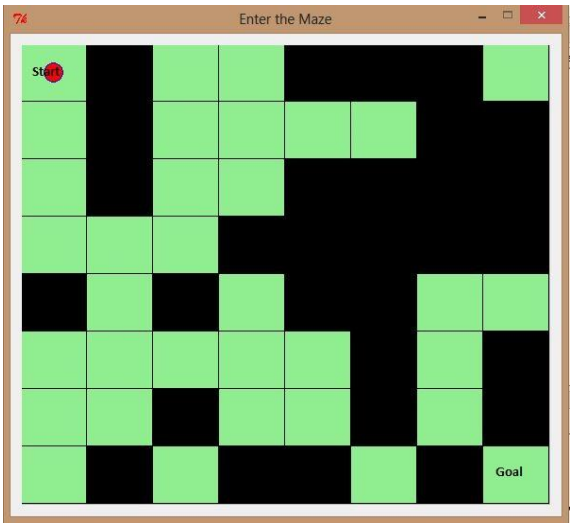


Figure 1.2



(Figure 1.1 shows an instance where there is a path from start to goal. Figure 1.2 shows an instance where a path from start to goal does not exist. The red circle denotes where you are in the maze.)

(a) <u>Feature Extraction – 25 points</u>

Your first task is to extract two features from the given dataset. You have been provided with the python script *feature_extraction.py* and within this script you have two skeleton functions defined for you.

The first feature you will compute is the proportion of black squares to the total number of squares in the grid. You will write your code for extracting this feature within the function *feature1(x)*. If the input to your function is the maze in Figure 1.1 then the feature value that should be returned for that maze is 24/64 = 0.375.

The second feature you will compute is the sum over all the rows of the maximum number of continuous black squares in each row. You will write your code for extracting this feature within the function *feature2(x)*. In Figure 1.1 the maximum number of continuous black squares in the first row (from the top) is 1, in the second row 1, in the third row 2, and in the sixth row 4, etc. The value of this feature for this example is therefore the sum of these values, i.e., 1+1+2+1+2+4+2+2 = 15.

(b) <u>Visualising Features – 15 points</u>

After writing your functions for extracting features, you will now visualize your training set using those features. This means you should extract these features from every training example. You should write your code in in the script *training_phase.py.* If we make a call to the function *visualise_features* we should be able to see a plot similar to Figure 1.3 below.

Below is a scatter plot for our training set.

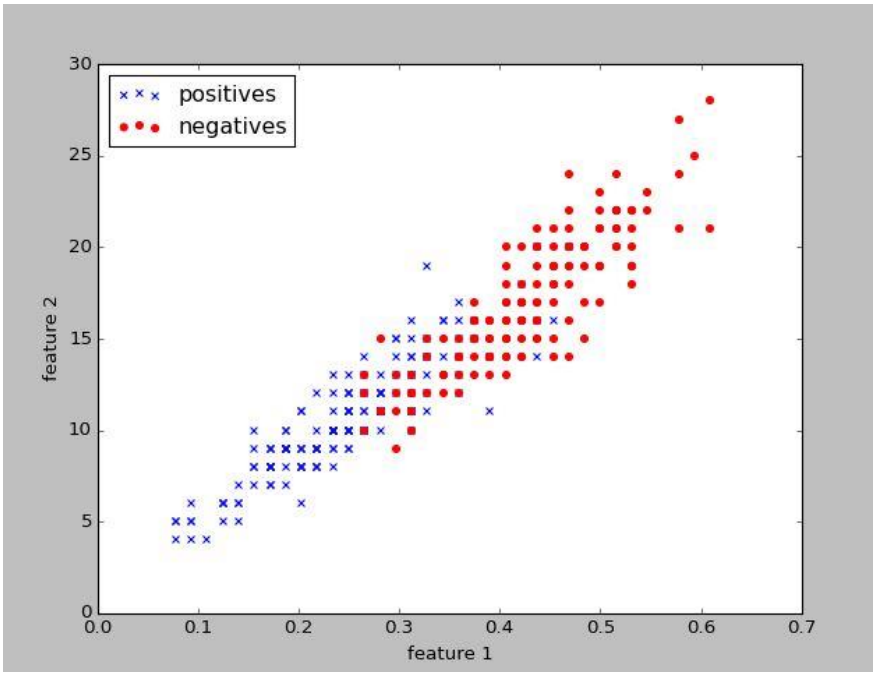


Figure 1.3 Distribution of the features for every maze in the training set

(c) <u>Perceptron Classifier - Training – 30 points</u>

As you can see from Figure 1.3, the data is not linearly separable. Hence, simply using the vanilla perceptron algorithm to classify this data will lead to unpredictable behaviour (Try it!). This is due to the fact that the perceptron returns the most recent version of the weight vector which is well adapted to the latest instances it encountered but may not generalize well to other (earlier) instances. You have been provided with an implementation of the perceptron classifier from your textbook. You are going to modify this algorithm, and implement the averaged perceptron classifier. The motivation behind implementing this classifier is to average all the weights seen so far and generate weight vectors that are well adapted to the training set as a whole. The way it works is as follows:

Assume we have a training set of size *m* and that we are running our algorithm for *n* iterations. We can view each version of the weight vector as a separate classifier, i.e., we have *mn* classifiers. The most intuitive way of implementing this algorithm would be to store each weight vector W_i and average them to get our averaged weight vector W_{avg} as follows:

$$W_{avg} = \frac{(W_0 + W_1 + \ldots + W_{mn})}{mn}$$

However, this approach is too impractical because it requires storing *mn* weight vectors which is a waste of memory.

A better approach would be to update our weight vector by using a running average where the vector W_i is the sum of all updates so far. For instance, if we wanted to compute the average of the vectors W_1, W_2, and W_3 we would do it as shown below:

$$W_0 = (0, 0, \dots, 0)$$

$$W_1 = W_0 + \Delta_1 = \Delta_1$$

$$W_2 = W_1 + \Delta_2 = \Delta_1 + \Delta_2$$

$$W_3 = W_2 + \Delta_3 = \Delta_1 + \Delta_2 + \Delta_3$$

$$W_{avg} = (W_1 + W_2 + W_3)/3 = (3/3)\,\Delta_1 + (2/3)\,\Delta_2 + (1/3)\,\Delta_3$$

Here Δ_j is defined as $\eta * (y^{(i)} - \bar{y}^{(i)}) x_j^{(i)}$ (the perceptron update rule from your book) where $y^{(i)}$ is the actual label of the i^{th} training example, $\bar{y}^{(i)}$ is the predicted label of the i^{th} training example and $x_j^{(i)}$ is the j^{th} feature of the i^{th} training example. W_1 corresponds to the weight learned after the first iteration of the first training example, W_2 corresponds to the weight learned after the first iteration of the second training example, and so forth. Modify the perceptron algorithm and implement the averaged perceptron classifier using the weight update rule above. Make sure to shuffle the training samples at every iteration.

Using the classifier that you built, you are now going to train your data for 1000 iterations with a learning rate $\eta = 0.1$. Please make sure that the function *train_classifier* returns the averaged weights learned by your perceptron classifier and these weights should have the same format as *self.w_* in *perceptron_classifier.py*. Also make sure to save your weights before returning them (you may write them to a text file), as you will need them later on.

(d) <u>Visualising the decision boundary – 15 points</u>

Now using the weights that you learned, you will plot a decision boundary on your dataset, and generate a plot like the one shown in Figure 1.4 (The colouring scheme is not necessary, we should just be able to distinguish the positive examples from the negative ones). You should write the code for generating this plot in the same script and within the function *visualise_decision_boundary*. You can load the weights you had saved earlier on for plotting this boundary.

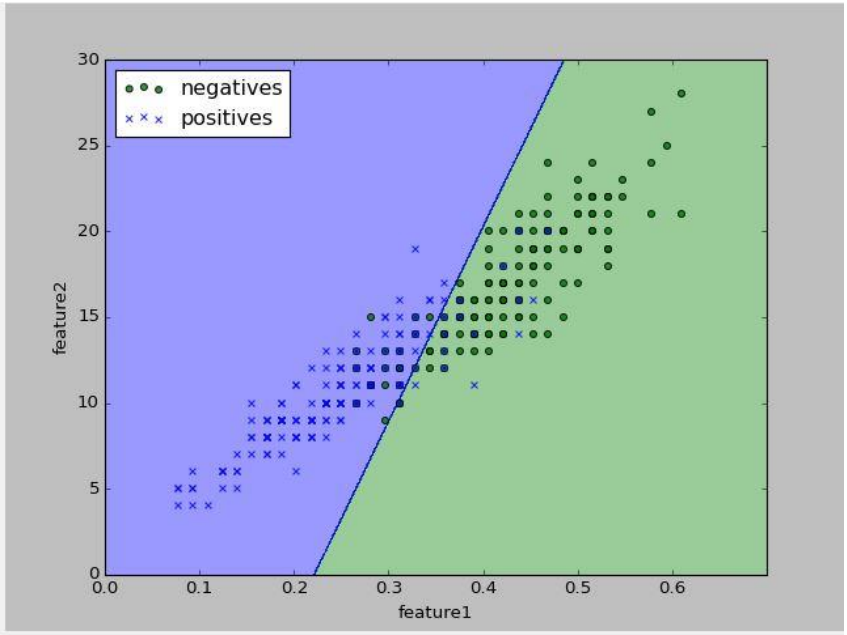


Figure 1.4 Decision boundary learned using the averaged perceptron classifier.

(e) Model Evaluation – 15 points

Your task is to compute the accuracy of your model on the training and test set we gave you and return an accuracy score. You should write your code in the function *evaluate_training_and_test_set*. You should return a tuple as follows:

```
return (accuracy_on_training_set, accuracy_on_test_set)
```

Prepare and upload one zip file which you will name as *<your first name>_<your last name>_ assignment1.* This zip file should contain all of the materials used in this assignment.