Homework 4: Decison Trees and Neural Nets Due Date: March 3rd, 11:59PM

Programming: Decision Trees and Neural Networks (25 points)

Similar to the previous two assignments, in this assignment you will program a robot agent to search for victims in a building that has been hit by some natural disaster. The main difference is that in this assignment you will train two classifiers to help the robot do its job. Specifically, you will train a decision tree to determine the obstacle cost of each cell, given various features of the cell (e.g. `fireOpt, liveWireOpt`). In addition, you will train a neural network to distinguish victims, based on 5x5 images of real and fake victims. You will use this because the robot only knows the locations of possible victims, and will have to distinguish between real and fake victims when it reaches them.

Note that this assignment **needs python3 to run**. Also, you will need to install `numpy` (instructions here). The TA's can help you with problems installing or using `numpy`.

Introduction

There are several files needed for this assignment.

- `decisionTree.py`: Includes the `DecisionTree` class that implements a decision tree. **You will need to modify this file.**
- `neuralNet.py`: Includes the `NeuralNet` class that implements a feedforward network. **You will need to modify this file.**
- `classificiation.py`: Includes functions for training and using a neural network to detect victims, as well as functions for creating and using a decision tree to classify the obstruction cost of map cells. **You will need to modify this file.**
- `SAR.py`: You need to replace this file with your code from HW2 (updated with the new `__eq__` function found in the handout version).
- `map.py`: Map representation for the search and rescue domain.
- `params.py`: Defines constants and other parameters used in the code.
- `search.py`: Framework for uninformed and heuristic search.
- `simulate.py`: Simulates your agent to determine failure or success

The first three files (the ones you will modify) will be described in more detail in the three problem descriptions below. Note that each problem corresponds to modifying one of those three files.

Testing

We have provided `autograder.py` to help you debug and test your code, prior to submission. You invoke it using `python autograder.py <options>`. The options are:

- `-p or --problem`: which problem to test (1, 2, 3);
- `-s or --step`: which step to test, if a problem has multiple steps;
- `--debugging`: prints out information about the status of the MDP (states, rewards, transitions generated). May be useful in tracing down bugs in your code.

The following options are relevant only for problem 3:

- `-m or --map`: which map file to use. The choices are in the `maps` subdirectory;
- `-g or --goal`: which mission to perform. The choices are VISIT and RETRIEVE (defined in `params.py`; relevant only for problem 3);
- `-t or --test`: if included, overrides the `-p` and `-s` options and tests the whole MDP implementation, using the simulator and (optionally) the graphics;

- `--no-graphics`: by default, a graphics simulation of the agent performing the mission is shown. This option turns off the graphics. Useful only with the `-t` option;

If the `-p` option is not provided, if will default to testing all the problems.

Important Note: We have designed `autograder.py` so that if your code passes all the autograder tests, it should also pass the GradeScope autograder. Since this year involves significant changes to `autograder.py`, let us know if you do find that not to be the case, and we will try to fix it.

Problem 1 (15 points, total)

In this problem, you will be implementing parts of the algorithm to train decision trees. You will need to modify `decisionTree.py`, which defines two classes: `Node` and `DecisionTree`, where a `Node` is either a leaf node or an interior node in the decision tree. Specifically, the `Node` class supports the following functions:

- `isLeaf`: Whether the node is a leaf node. Leaf nodes have no children and have their `class` attribute set.
- `getClass`: If it is a leaf node, returns the class value associated with that node.
- `setClass`: Sets the class value associated with that node, which by definition makes it a leaf node.
- `getFeature`: Returns the feature attribute associated with the non-leaf node.
- `getChildren`: For non-leaf nodes, a list of `Node` instances, which may be leaf or non-leaf nodes.

The `DecisionTree` class supports the following functions:

- `getRoot`: Returns the root (top-most) node of the decision tree.
- `getClasses`: Returns the list of class values that can exist in the leaf nodes.
- `getFeatures`: Returns the list of features that can be used to branch on.
- `getFeatureValues`: Returns the list of values that a feature can take on. For instance, the features of a decision tree could be `['size', 'color']` and `featureValues('size')` might return `['small', 'medium', 'large']` while `featureValues('color')` might return `['red', 'blue', 'green']`.
- `createNode`: Given a `feature`, create a non-leaf `Node` with that feature as what it uses to branch on.
- `createLeaf`: Given a `class` value, create a leaf node with that classification.
- `addChild`: Given a `parent` node, a `feature_value` that is one of the values for the feature of the parent node, and a `child` node, add the child to the children of the parent.
- `mode`: Given a `dataset` (defined below), return the class value that appears most frequently in the dataset.
- `uniformClass`: If all the class values in the `dataset` are the same, return that value, otherwise return `None`.
- `subDataset`: Given a `feature`, a `value` of that feature, and a `dataset`, return the subset of the data that contain only the value of that feature.
- `infoGain`: Given a `feature` and a `dataset`, return the information gain from choosing that feature. **You will implement this function.**
- `createTree`: Given a `dataset` and an optional list of `features`, construct a decision tree that uses information gain to determine which features to branch on. If `features` is not provided, it defaults to the full set of features of the decision tree. **You will implement this function.**
- `classify`: Given a list of `feature_values`, return the class value that the tree gives for those values.

You will also need to use the function `validateDT` in `utils.py`. Given a `decisionTree` and a `dataset`, this function computes and returns the decision tree's classification **accuracy**. In addition, if an optional `pclass` argument is provide(i.e., the "positive" class you are looking for), it also prints the **precision** and **recall** statistics (the Decision Trees lecture slides has definitions for all these terms).

Many of the functions above take a `dataset` as an argument. A `dataset` is a list of tuples, where each tuple consists of a class value and a list of feature values. The list of feature values is ordered the same as the features used to define the decision tree. For instance, if the class values of a decision tree are `['heavy', 'light']` and the features are `['color', 'size']`, then a dataset might be:

```
[('heavy', ['large','blue']),('light', ['small', 'red']),('light', ['medium', 'red'])]
```

indicating that large blue things are heavy, while small and medium red things are light.

In addition, `decisionTree.py` defines several utility functions that you may find useful:

- `prob`: Given a `class` value and a `dataset`, return the probability that the class value appears in the dataset (i.e., how many times it appears).
- `fprob`: Given a `feature`, a `value` of that feature, the list of all `features`, and a `dataset`, return the probability that that feature's value appears in the dataset.
- `condProb`: Given a `class` value, a `feature`, a `value` of that feature, the list of all `features`, and a `dataset`, return the *conditional probability* that the class value appears in the dataset, given the feature's value.
- `entropy`: Given a list of `probabilities`, return the entropy of those probabilities.

Step 1 (3 points): Implement the `infoGain` function in the `DecisionTree` class in `decisionTree.py`. Given a `feature` and a `dataset` (as defined above), `infoGain` computes and returns the information gain that would occur by selecting that feature. Make use of the utility functions in `decisionTree.py` (described above) to implement this function. Use slide 20 from the Decision Trees lecture (February 15), and the example that follows that slide, as a guideline.

To test your `infoGain` function, run `autograder.py -p 1 -s 1`.

Step 2 (7 points): For this step, modify the function `createTree` in the `DecisionTree` class in `decisionTree.py`. Given a `dataset` and an optional list of `features`, the function should return a `Node` that represents a decision-tree classifier for those features. If `features` is `None` (the default), it uses all of the tree's features (i.e., `self.getFeatures()`), otherwise, you should pass in a subset of the features, based on what has already been branched on. If the class values for the `dataset` are all the same, then a leaf node is created and returned, with the appropriate class value; if there are no more features left to choose, then a leaf node is created and returned, with the appropriate class value. Otherwise, a feature is chosen from the given features, based on information gain, and children nodes are added for each possible value of that feature (i.e., `self.getFeatureValues(chosenFeature)`).

Use the decision tree algorithm presented on slide 16 of the Decision Trees lecture (February 15) as a guideline. Where the algorithm says "pick a feature F", pick the feature, from those that have not already been used, that maximizes information gain for that subset of data. Note that it may be beneficial to implement this function using recursion.

For the sake of consistency, we have provided a `featureGreaterHandlingTies` function in `decisionTree.py` to enforce a way of choosing different features. It can be seen as a compare function on the features. It takes in two feature and their respective infoGain, and returns true if the first one is better and false if the second one is better. The name of the feature field can be `None`, but they will be treated with the lowest priority when tie-breaking. You should use this function to find the best feature to split on.

To test your `createTree` function, run `autograder.py -p 1 -s 2`. Note that since `createTree` relies on `infoGain`, you need to make sure that function works correctly. You will be graded as to how accurate your solution is compared with the reference solution, but note that the accuracy may not have to

be 100% for full score on the problem – decision trees are not always 100% accurate. If you want to see which points were misclassified, run `autograder` with the `--debugging` flag.

As a debugging aid, you can use the function `display` in the `DecisionTree` class to view the created decision tree. `dt.display()` prints out a hierarchical representation of the decision tree (see Figure 1), where each line starts with a feature value and shows either a leaf node, with its class value, or an interior node, showing the feature it splits on.

```
Root:  split:  A
    false:  split:  H
         aLot:  class:  yes
         none:  class:  no
         some:  class:  unknown
    true:  split:  B
         good:  class:  no
         bad:  class:  yes
```

Figure 1: Example tree produced by `dt.display()`

Step 3 (5 points): From the results of testing in Step 2, you have seen that the accuracy of decision trees may not be 100%, with both false positive and false negative classifications. This typically happens due to overfitting, where the training data may either be not reliable, for instance because of a miscalibrated sensor, or may not be chosen from the same distribution as the testing data. To correct for this, it is common to *prune* the tree down to some maximum depth. The idea is that features lower in the tree, that have very low information gain, may not be very reliable in terms of classification.

For this step, you will implement the `prune` function in the `DecisionTree` class in `decisionTree.py`. Given a `dataset`, a maximum `depth` of the tree rooted at that node, and a `treeNode` at which to start pruning, prune the tree to that depth. Keep any nodes that are at, or above, that depth; for non-leaf nodes that are at the given depth, convert them to leaf nodes by setting the node's class value (using `setClass`) to the most likely (mode) value of the `dataset` (note – a leaf node is just one that has a class value specified; you don't need to create any new nodes when doing the pruning).

Note that it may be beneficial to implement this function using recursion. Make sure that each time you recurse, you pass the `dataset` that is the subset for that node's feature value.

To test your `prune` function, run `autograder.py -p 1 -s 3`. The autograder will create a reference tree, have your function prune it to some depth, and compare your pruned tree with the pruned reference tree.

Problem 2 (7 points):

In this problem, you will be writing the function for doing feed forward and training a neural network.

`neuralNet.py` contains the `NeuralNet` class, which supports the following functions that may be of use to you:

- `feed_forward`: Given an `input` vector, computes and return the output vector. **You will implement this function.**
- `classify`: Given an `input` vector, returns the class value (label) associated with the most likely output.
- `train`: Given a `trainingSet`, consisting of an array of tuples of the form `(label, input)`, where `label` is a class value and `input` is a vector of feature values, trains the network for 1 epoch and returns the training error (i.e., fraction of wrong classifications).

You will also need to use the function `validateNN` in `utils.py`. Given a `neuralNet` and a `testingSet`, consisting of an array of tuples of the form `(label, input)`, where `label` is a class value and `input` is

a vector of feature values, this function computes evaluation metrics for the neural network and returns the accuracy (i.e., fraction of correct classifications).

Step 1 (3 points): Extend the `feed_forward` function of the `NeuralNet` class to perform feed forward. The function takes an `input`, which is an array of feature values, and propagates them through the network. For each layer, including the final output layer, feed forward computes the **input** to each node using the dot product of the outputs of the previous layer and adds in the bias term for that node. It then passes the that input through the node's activation function to produce the node's **output**. The outputs are then used to compute the inputs to the nodes of the next layer.

Specifically, let's say that $in_{i,j}$ is the the input to the j-th node at the i-th layer and $out_{i-1,k}$ is the output of the k-th node from the (i-1)-th layer. Then:

$$in_{i,j} = \sum (out_{i-1,k} * weights_{i,k,j}) + bias_{i,j}$$
$$out_{i,j} = activation_j(in_{i,j})$$

Note that the `feed_forward` function maintains the intermediate results (`ins` and `outs`) to be used by the back-propagation function. Don't worry about that part, though, it's already been implemented for you.

To test your `feed_forward` function, run `autograder.py -p 2 -s 1`. The autograder will create a simple network, supply an input, and check the output of the function, along with the intermediate results, against the reference solution.

Step 2 (4 points): Implement the `trainNetwork` function in file `neuralNetwork.py`. The function is given a `dataset`, which is a list of `(label, input)` tuples, the number of hidden layers of the network, the number of epochs with which to train the network, the learning rate, and the number of folds (for doing cross validation).

The function should do N-fold cross validation to find a network that will likely perform best on unseen testing data and returns that best network. To do N-fold validation, split the dataset into N chunks (of approximately equal size) and create N neural networks, using the parameters passed to the function. For each network, save one of the chunks for testing and train on the other N-1 chunks. Test the trained network and return the network that you deem will likely work best on unseen test data (it may not necessarily the network that performed best on N-th chunk, since that might have overfit the data). You may have to experiment with which network to return to get best performance on unseen test data.

To test your `trainNetwork` function, run `autograder.py -p 2 -s 2`. The autograder will provide your function with test data and parameters and then validate the network returned against unseen test data.

Problem 3 (3 points)

In this part, you will put everything together to detect victims using a neural network and classify obstacle obstructions using a decision tree, and then have the agent plan to visit and retrieve victims. For this part, you will be modifying functions in `classification.py`.

Step 1 (1.5 points) Implement the `trainObstacleCost` function to train a decision tree to classify the cost of obstructions (1-4). The input is a list of `feature_values` and `classes`, which are needed to create the decision tree, and a `trainingSet` consisting of `(label, fvalues)` tuples, where the `label` is a string representing the obstruction of a map cell, and `fvalues` is a list of values of the features, in the same order as `feature_values`. In addition to training the decision tree, you may want to consider pruning it, to avoid overfitting. Return the trained decision tree.

You can use `python3 autograder.py -p 3 -s 1` to test your function. It will supply a training set and you can use the results to tune the network parameters to maximize performance. The goal is to match, or better, the refsol's accuracy on a test data set.

Step 2 (1.5 points) Implement the `trainIsVictim` function to train a neural network to detect victims. The input is a `trainingSet` consisting of `(label, image)` tuples, where the `label` is 1 if the image represents a victim and 0 otherwise, and the `image` is a flattened numpy array, representing a 5x5 image. You are free to use the `trainNetwork` function that you implemented in Problem 2, however here you need to choose the parameters of the neural network to get the best overall performance. Return the trained neural network.

You can use `python3 autograder.py -p 3 -s 2` to test your function. It will supply a training set and you can use the results to tune the network parameters to maximize performance. The goal is to match, or better, the refsol's accuracy on a test data set.

One thing to note is that, in this application, false positives (i.e., thinking a cell contains a victim when it does not) is actually worse than false negatives, because it may have the robot plan to go to a place where there is actually no victim, and the way the search-and-rescue simulator is set up, the agent gets only one chance to visit or retrieve a victim. So, pay attention to the false positive (FP) and false negative (FN) rates that you get.

Step 3 (0 points) Now, you get to see how this all works in the search-and-rescue (SAR) domain. You will use the `SAR.py` code that you developed for HW2 to plan a sequence of actions to achieve either the `Visit` or `Retrieve` goal. The way it works is that the autograder will call `trainObstacleCost` and `trainIsVictim` to get the appropriate classifiers, and then will use them to alter the map. For instance, if your neural net predicts that there is a victim where there, in fact, is none, it will augment the map to add an additional victim, and vice versa. Similarly, it will update all the obstruction costs based on your decision tree's classification of the inputs. It will then run A* search on the updated map, but test it against the original map. Sometimes, misclassifications will not affect the optimal plan (autograder gives you 1 point), sometimes you achieve the goal but suboptimally (0.5 points) and sometimes the plan fails when it goes to a cell where there really is no victim (0 points).

Note: The points are only for your own satisfaction – the Gradescope autograder will not be checking this part.

To run this step, use `python3 autograder.py -t`, along with the options that have been defined in previous problems: `--map`, `--goal`, and `--no-graphics`. Note that the test is stochastic, in that it provides different test inputs each time it is run, so that your results will likely vary from run to run – sometimes, all the tests (maps and goal-types) might succeed optimally, and sometimes not. Just like in real life!

Submission

You will need to submit your `decisionTree.py`, `neuralNet.py` and `classification.py` to Gradescope. You have an unlimited number of submissions, and your grade on the programming portion will be indicated by the Gradescope autograder output. You will not be graded on code style; however, we may manually grade portions of your code (e.g., questions that are more open-ended or have multiple implementations).