Artificial Intelligence **Assignment 6**
comp440/comp557 Fall 2018 **14 November 2018**

This assignment is due November 30 at 8 pm on Canvas. Download `assignment6.zip` *from Canvas. There are five problems worth a total of 105 regular points + 40 extra credit points for comp440 and 145 regular points for comp557. Problems 1 and 4 require written work only, Problems 2, 3 and 5 require Python code and a writeup. All written work should be placed in a file called* `writeup.pdf` *with problem numbers clearly identified. All code should be included at the labeled points in* `submission.py` *in the folders* `text_classification` *and* `image_classification`*. You will fill in portions of* `valueIterationAgents.py`*,* `qlearningAgents.py`*, and* `analysis.py`*. You should submit these files separately on Canvas and include in* `writeup.pdf` *the results of your autograder runs. For Problems 2, 3 and 5 please run the autograder using the command line* `python grader.py` *and report the results in your writeup. Upload all code directories , code files named above, and* `writeup.pdf` *on Canvas by the due date/time.*

1 Decision networks (10 points)

Consider a student who has the choice to buy or not buy a textbook for a course. We will model this decision problem with one boolean decision node B, indicating whether the student chooses to buy the book, and two Boolean nodes M, indicating whether the student has mastered the material in the book, and P indicating whether the student passes the course. There is a utility node U in the network. A certain student, Sam, has an additive utility function: 0 for not buying the book and -\$100 for buying it; and \$2000 for passing the course and 0 for not passing it. Sam's conditional probability estimates are:

$$\begin{array}{ll} P(p|b,m) = 0.9 & P(p|b,\neg m) = 0.5 \\ P(p|\neg b,m) = 0.8 & P(p|\neg b,\neg m) = 0.3 \\ P(m|b) = 0.9 & P(m|\neg b) = 0.7 \end{array}$$

You might think that P would be independent of B given M. But, this course has an open-book final – so having the book helps.

- (4 points) Draw the decision network for this problem.
- (5 points) Compute the expected utility of buying the book and not buying the book.
- (1 points) What is the optimal decision for Sam?

2 Text classification (40 points)

According to Microsoft, circa 2007, 97% of all email is unwanted spam. Fortunately, most of us avoid the majority of these emails because of well-engineered spam filters. In this assignment, we will build a text classification system that, despite its simplicity, can identify spurious email with impressive accuracy. You will then apply the same techniques to identify positive and negative product reviews and to classify email posts into topical categories. All the code for this problem resides in folder `text_classification`

Problem 2.1: Spam classification (20 points)

Let us start by building a simple rule-based system to classify email as spam or ham. To test our system, we will be using a corpus of email made publicly available after the legal proceedings of the Enron collapse; within the `data/spam-classification/train` directory, you will find two folders `spam` and `ham`. Each folder contains a number of full text files that contain the whole email without headers and have been classified as spam and ham respectively.

In order to implement your own spam classifier, you will subclass `Classifier` and implement the `classify` method appropriately in `submission.py`. As usual, `grader.py` contains a number of simple test cases for your code. To run, type `python grader.py` on the command line.

Additionally, we have provided a script `main.py` to help you interactively run your code with different parameters; be ready to change this script to use alternate features, print different statistics, etc. It is really just meant to help you get started. To use this script, type `python main.py part<part>`, using the section number (2.1, 2.2, etc.). `python main.py part<part> -h` lists additional parameters you might find useful. The script will output the classification error rate as well as a confusion matrix for the training and development set. Each cell of the confusion matrix, (row, column), of the matrix contains the number of elements with the true label row that have been classified as column.

Problem 2.1.1: Rule-based system (3 points)

- (2 points) `data/spam-classification/blacklist.txt` contains a list of words sorted by descending spam correlation. Implement `RuleBasedClassifier` by discarding any email containing at least one word from this list. It is recommended you store the blacklist as a set to reduce lookup time. For comparison, our system achieved a dev error rate of about 48% on the training set with this heuristic. You can test this by running `python main.py part3.1.1` at the command line. If you run `python grader.py` you should get train error of 0.474 and a dev error of .48

- (1 point) Relax this heuristic by only discarding email that contains at least n or more of the first k listed words. You can test this by running `python main.py part3.1.1 -n 1 -k 10000` at the command line for $n = 1$ and $k = 10000$. Report your dev error results on the training set in a 3x3 table with $n = 1, 2, 3$ and $k = 10000, 20000, 30000$.

Problem 2.1.2: Linear classifiers (4 points)

As you have observed, this naive rule-based system is quite ineffective. A reasonable diagnosis is that each word, whether it is viagra or sale, contributes equally to the 'spamminess' of the email. Let us relax this assumption by weighing each word separately.

Consider producing a spamminess score, $f_w(x)$ for each email document x which sums the 'spamminess' scores for each word in that document;

$$f_w(x) = \sum_{i=1}^{L} w(word_i)$$

where L is the length of the document x, and w is the 'spamminess' score for word $word_i$. We can use a linear classifier that will classify the email as spam if $f_w(x) \geq 0$, and as ham, otherwise.

Note that the order of words does not matter when computing the sum $f_w(x)$. Thus, it is convenient to represent a document as a sparse collection of words. An ideal data structure for this is a hash map or dictionary. For example, the document text `The quick dog chased the lazy fox over the brown fence`, would be represented using the dictionary, { `brown`: 1, `lazy`: 1, `fence`: 1, `fox`: 1, `over`: 1, `chased`: 1, `dog`: 1, `quick`: 1, `the`: 2, `The`: 1}.

Let the above vector representation of a document be $\phi(x)$; in this context, the vector representation is called a feature. The use of individual words or unigrams is a common choice in many language modeling tasks. With this vector or feature representation, the spamminess score for a document is simply the inner product of the weights and the document vector $f_w(x) = w.\phi(x)$.

Let the positive label be represented as a 1. Then, we can write the predicted output $\hat{y}$ of the classifier as

$$\hat{y} \quad = \begin{cases} +1 & \text{if } \ w.\phi(x) \geq 0 \\ -1 & \text{if } \ w.\phi(x) < 0 \end{cases}$$

- (2 points) Implement a function `extractUnigramFeatures` that reads a document and returns the sparse vector $\phi(x)$ of unigram features. Run `python grader.py` to check your implementation.
- (2 points) Implement the `classify` function in `WeightedClassifier`. You can test whether you have implemented this correctly by running `python grader.py`.

Problem 2.1.3: Learning to distinguish spam (13 points)

The next question we need to address is where the vector of spamminess scores, w, comes from. Our prediction function is a simple linear function, so we will use the perceptron algorithm to learn weights. The perceptron algorithm visits each training example and incrementally adjusts weights to improve the classification of the current labelled example (x, y). If $\hat{y}$ is the prediction the algorithm makes with the current set of weights $w^{(t)}$, i.e., $\hat{y} = I(w^{(t)}.\phi(x) \geq 0)$, then if $\hat{y} \neq y$, increment $w^{(t)}$ by $y \times \phi(x)$.

- (6 points) Implement `learnWeightsFromPerceptron` that takes as input a corpus of training examples, and returns the w learned by the perceptron algorithm. Initialize your weights uniformly with 0.0. If you run `python grader.py` you should see a train error of 0.008968 and a dev error of 0.04868.
- (2 points) So far, we have looked at scores for single words or unigrams. We will now consider using two adjacent words, or bigrams as features by implementing `extractBigramFeatures`. To handle the edge case of a word at the beginning of a sentence (i.e. after a punctuation like '.', '!' or '?'), use the token `-BEGIN-`. On the previous example, `The quick dog chased the lazy fox over the brown fence`, `extractBigramFeatures` would return, {`the brown`: 1, `brown`: 1, `lazy`: 1, `fence`: 1, `brown fence`: 1, `fox`: 1, `over`: 1, `fox over`: 1, `chased`: 1, `dog`:

1, `lazy fox`: 1, `quick dog`: 1, `The quick`: 1, `the lazy`: 1, `chased the`: 1, `quick`: 1, `the`: 2, `over the`: 1, `-BEGIN- The`: 1, `dog chased`: 1}.

Run `python grader.py` to check your implementation.

- (5 points) Vary the number of examples given to the training classifier in steps of 500 from 500 to 5,000. Provide a table of results showing the training and development set classification error when using bigram features. How did the additional features help the training and development set error? You can run `python main.py part3.1.3` to get the results for producing the table.

Problem 2.2: Sentiment classification (8 points)

You have just constructed a spam classifier that can identify spam with a 96% accuracy. While this in itself is great, what's really impressive is that the same system can easily be used to learn how to tackle other text classification tasks. Let us look at something that is completely different; identifying positive and negative movie reviews. We have provided a dataset in `data/sentiment/train`, with labels `pos` and `neg`.

- (3 points) Use the perceptron learning algorithm you wrote in the previous section to classify positive and negative reviews. Report the training and development set error rate for unigram features and bigram features. You can run `python main.py part3.2` to get these results. You can modify the function `part2` in main.py if you want to change parameters.
- (5 points) Make a table of the training and development set classification errors as you vary the number of iterations of the perceptron algorithm from 1 to 20. Use bigram features for this part. Use `main.py` for this part as before. Does development set error rate monotonically decrease with iteration number? Why or why not? You might find it useful to write another version of `learnWeightsFromPerceptron` that prints training and development error in each iteration. Optionally, you might try plotting the errors to visually see how the two errors behave. We recommend the `matplotlib` library. Include these plots in your writeup if you make them.

Problem 2.3: Document categorization (12 points)

Finally, let's see how this system can be generalized to tasks with multiple labels. We will apply our knowledge to the task of categorizing emails based on topics. Our dataset in `data/topics/train` contains a number of emails from 20 popular USENET groups that have been segregated into 5 broader categories.

- (9 points) A common approach to multi-class classification is to train one binary classifier for each of the categories in a "one-vs-all" scheme. Namely, for the binary classifier i, all examples labelled i are positive examples, and all other examples are negative. When using this classifier for prediction, one uses the class label with the largest score, $f_i(x)$. Implement the `OneVsAllClassifier` classifier (and `MultiClassClassifier`). In order to train this classifier, you will also need to implement `learnOneVsAllClassifiers` that will appropriately train binary classifiers on the provided input data.

- (4 points) Report the train and development set error rate for unigram features and bigram features in your writeup. You can generate these by modifying function `part3` in `main.py`. If you run `python main.py part3.3` with our default `part3`, you should see a development error rate of about 12%.

3 Image classification (40 points)

In this assignment, you will implement an image classifier that distinguishes birds and airplanes. We will use the perceptron algorithm as a starting point but what should the features (arguably the most important part of machine learning) be? We will see that rather than specifying them by hand, as we did for spam filtering, we can actually learn the features automatically using the K-means clustering algorithm. We will be working with the CIFAR-10 dataset, one of the standard benchmarks for image classification. We assume that your version of Python has `numpy` and `PIL` packages already installed. If you do not have these packages, you can install them fairly easily from distributions downloadable from the Web. If you use Anaconda Python, these packages come pre-installed.

Warmup

Now we will get you familiar with the classification task. You will be classifying a 32×32 image as either a bird ($y = 1$) or a plane ($y = 0$). One way to do this is to train a classifier on the raw pixels, that is, $\phi(x) \in \Re^{32\times 32}$ vector where each component of the vector represents the intensity of a particular pixel. Try running this classifier with

```
python run.py --pixels
```

As you can see, these features do not work very well, the classifier drastically overfits the training set and as a result barely generalizes better than chance.

The problem here is that pixel values themselves are not really meaningful. We need a higher-level representation of images. So we resort to the following strategy: Each image is a 32×32 grid of pixels. We will divide the image into sixteen 8×8 "patches". Now comes the key part: we will use K-means to cluster all the patches into centroids. These centroids will then allow us to use a better feature representation of the image which will make the classification task much easier.

Implementing the K-means algorithm (10 points)

In this part of the assignment you will implement K-means clustering to learn centroids for a given training set. Specifically you should fill out the method `runKMeans` in the file `submission.py`.

Let $D = \{x_1, \ldots, x_n\}$ be a set of data, where $x_i \in \Re^d$. We will construct K clusters with means $\mu_1, \ldots, \mu_K$ where $\mu_j \in \Re^d$.

- Initialize a set of means $\mu_1, \mu_2, \ldots, \mu_K$.
- for i in 1..maxIter

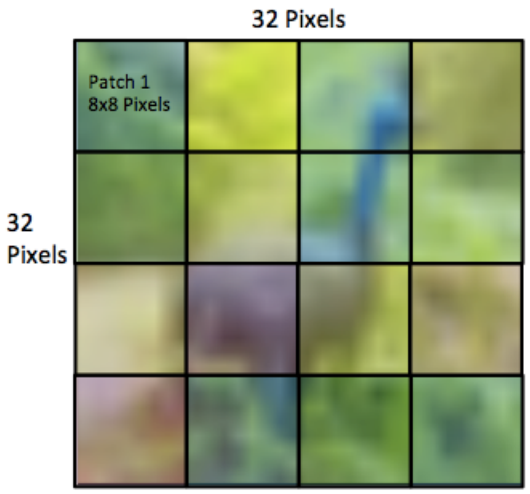

Figure 1: The sixteen patches of size 8x8 pixels corresponding to an image of size 32x32 pixels.

- Assignment step: assign each x_i to the closest cluster mean $z_i \in \{\mu_1, \ldots, \mu_K\}$.

$$z_i = argmin_k ||x_i - \mu_k||_2$$

- Update step: given the cluster assignments z_i, recalculate the cluster means μ's.

$$mu_k = \frac{1}{\sum_{z_i=k} 1} \sum_{z_i=k} x_i$$

We start you off by initializing K-means with random centroids where each floating point number is chosen from a normal distribution with mean 0 and standard deviation 1. Test the K-means code with the provided test utility in `grader.py` by running:

```
python grader.py
```

Optional: One way to determine if your K-means algorithm is learning sensible features is to view the learned centroids using our provided utility function. To view the first 25 learned centroids, run

```
python run.py --view
```

Your centroids should look similar to Figure 2. Notice how the centroids look like edges. Important note on patches: Images are composed of patches which have been converted to gray-scale followed by standard image preprocessing. This includes normalizing them for luminance and contrast as well as further whitening.

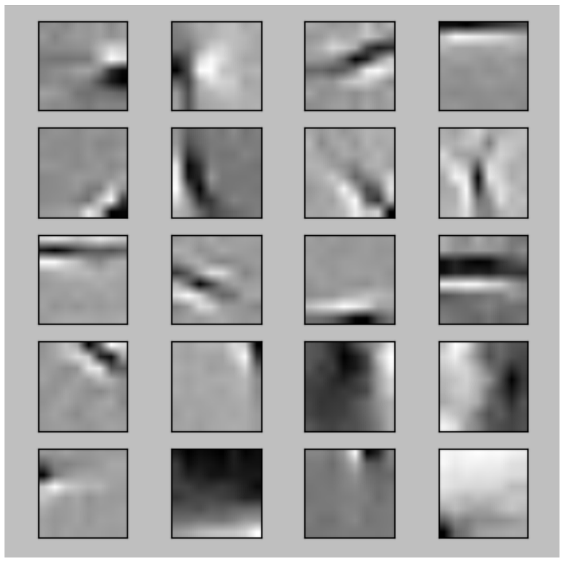

Figure 2: 20 centroids learned from K-means on patches from the first 1000 training images..

Feature Extraction (10 points)

The next step in the pipeline is to use the centroids to generate features that will be more useful for the classification task then the raw pixel intensities. One way to do this is to represent each patch by the distance to the centroids that are closest to it, the intuition being that we can encode a patch by the centroids to which it is most similar. We will map each image x to its new feature vector $\phi(x) \in \Re^{16k}$, where there is a real value for each patch, centroid combination.

Let $p_{ij} \in \Re^{64}$ be the ij-th patch of x where $i, j = 1, \ldots, 4$. The relative activation, a_{ijk}, of centroid μ_k by patch p_{ij} is defined to be the average Euclidean distance from p_{ij} to all centroids minus the Euclidean distance from p_{ij} to μ_k. The Euclidean distance is the L_2 norm, $||v||_2 = \sqrt{v^T v}$.

$$a_{ijk} = \frac{1}{K}\left(\sum_{k'=1}^{K} ||p_{ij} - \mu_{k'}||_2\right) - ||p_{ij} - \mu_k||_2$$

The feature value for patch p_{ij} and centroid μ_k is the max of the relative activation and zero.

$$\phi_{ijk}(x) = max(a_{ijk}, 0)$$

Implement the function `extractFeatures` in `submission.py`. We will use these features in the linear classifier below.

Supervised Training (20 points)

The final task is to use our better feature representation to classify images as birds or planes. We have provided you with a working Perceptron classifier which does fairly well. You will implement the logistic and hinge loss updates to learn the parameters and see if either of these improves classification accuracy and which does better. First you can run the Perceptron classifier to see how it performs on the test set:

```
python run.py --gradient=perceptron
```

You should see a test accuracy result between 60%-65% - we can do better!

- (10 points) Implement the `logisticGradient` method in `submission.py`. You can test this method with `python grader.py`. Given example $(\phi(x), y)$ where $\phi(x) \in \Re^d$ and $y \in \{0, 1\}$, and parameter vector w, define $yy = 2 * y - 1$ to map $\{0, 1\}$ to $\{-1, 1\}$. The logistic loss function is

  $$Loss_{logistic}(w, \phi(x), yy) = log(1 + e^{-w^T\phi(x)*yy})$$

  Compute the gradient of this function with respect to w and complete the `logisticGradient` method.

- (10 points) Implement the `hingeLossGradient` method in `submission.py`. The hinge loss function is

  $$Loss_{hinge}(w, \phi(x), yy) = max(1 - w^T\phi(x) * yy, 0)$$

  Compute the gradient of this function with respect to w and complete the `hingeLossGradient` method. You can test this method with `python grader.py`.

Now you are ready to run the entire pipeline and evaluate performance. Run the full pipeline with:

```
python run.py --gradient=<loss function>
```

The loss function can be any of {"hinge","logistic","perceptron"}. The output of this should be the test accuracy on 1000 images. One benefit of using an unsupervised algorithm to learn features is that we can train the algorithm with unlabeled data which is much more easily obtained than labeled data. We have included an option to train the K-means algorithm using 500 extra unlabeled images. You can run this with

```
python run.py --gradient=hinge -m
```

The performance of the supervised classifier goes up even though we are not using labeled data – a major benefit to using an unsupervised algorithm to learn a feature representation!

Numpy cheat sheet

This is a quick overview of the functions you will need to complete this assignment. A really thorough introduction is at `http://wiki.scipy.org/Tentative_NumPy_Tutorial`.

All standard operations on vectors, matrices and n-dimensional arrays are element-wise in `numpy`. For example

```
A = numpy.random.randn(5,5)
B = numpy.random.randn(5,5)
A+B # element-wise addition
A*B # element-wise multiplication
```

You can also access individual elements, columns and rows of A using the following notation,

```
A[0,0] # first element of the first row of A
A[:,1] # second column of A
A[3:5,:] # fourth and fifth rows of A
```

In order to take the matrix product of A and B use the `numpy.dot` function.

```
numpy.dot(A,B) # matrix multiplication A*B
A.dot(B) # same as above
```

To find the minimum or maximum element in an array or along a specific axis of an array (rows or columns for 2D), use `numpy.minimum` or `numpy.maximum`

```
numpy.minimum(A) # min of A
numpy.minimum(A, axis=0) # min of each column of A
numpy.maximum(A, axis=1) # max of each row of A
```

To take the indices of the minimum element in each row or column of a 2D array use the `numpy.argmin` function and specify the axis

```
numpy.argmin(A,axis=0) # argmin of each column
```

Similarly you can take the mean along the columns or rows of a 2D array using `numpy.mean` and the sum along a specific axis using `numpy.sum`

```
numpy.mean(A,axis=0) # mean of each column of A
numpy.sum(A,axis=1) # sum of each row of A
```

4 Perceptrons, Decision Trees, Neural Networks (15 points)

The data set below is a subset of a census database (the full data is available at `ftp://ftp.ics.uci.edu/pub/machine-learning-databases/adult`). Your goal is to predict whether an individual earns more or less than 50K a year based on education, gender and citizenship. Education is a discrete-valued attribute which can take one of three values; BS, MS, PhD; gender takes one of two values: male, female; and citizenship has two values: US and nonUS. You will use four different algorithms to learn models over this training data set.

Education	Gender	Citizenship	Income
BS	male	US	$\leq$50K
MS	male	nonUS	$>$50K
BS	female	US	$\leq$50K
PhD	male	nonUS	$\leq$50K
MS	female	US	$>$50K
PhD	female	nonUS	$\leq$50K
BS	male	US	$\leq$50K
PhD	male	US	$>$50K
BS	female	nonUS	$\leq$50K
PhD	female	US	$>$50K

Test your models on the following test set.

Education	Gender	Citizenship
PhD	male	US
PhD	male	nonUS
MS	female	nonUS

a. The perceptron algorithm (5 points)

- (1 point) How would you encode the training data as inputs to a perceptron?
- (3 points) Initialize all weights to zero and perform one pass of perceptron training on the training set, doing updates in the order in which the observations are presented. Present your result in a table, showing the weight vector after processing each example.
- (1 point) Will the perceptron converge if you run it long enough? Justify your answer.

b. Decision Trees (5 points)

- (2 points) Calculate the information gain of each of the three features. Which feature should be chosen as the root of the decision tree?
- (2 points) Now continue with the decision tree construction process with the root attribute chosen above. Determine the attribute to split on for the next level of nodes in the decision tree. Show your information gain calculations. Continue the construction process till all the leaf nodes have instances belonging to the same `Income` class. Show the final (unpruned) tree that you have constructed.
- (1 point) What does your decision tree predict on the three examples in the test set? That is, fill out the following table.

Education	Gender	Citizenship	Income
PhD	male	US	
PhD	male	nonUS	
MS	female	nonUS	

c. Neural networks (5 points)

- (1 point) How would you encode the data as inputs to a feedforward neural network?
- (3 points) Code this example with sklearn's MLP implementation, and train the network on the given training set. Documentation is available at `http://scikit-learn.org/stable/modules/neural_networks_supervised.html` You will need to set up two numpy arrays for training `trainX` and `trainy` corresponding to the 10 training examples. You will also need to set up the test set as the array `testX`. You will have to make a choice of the number of hidden units. Try the range $2, \ldots, 5$ and report the cross-validated training error.
- (1 point) What are the predictions made on the test set? Compare the classifications of the test cases made by the four classifiers and comment on their similarities and differences.

5 Reinforcement Learning (40 points) (required for comp557 and optional for comp440)

This problem uses the UC Berkeley pacman project code base. In this project, you will implement value iteration and Q-learning. You will test your agents first on Gridworld (from class), then apply them to a simulated robot controller (Crawler) and finally to the game of Pacman. This project includes an autograder for you to grade your solutions on your machine. This can be run on all questions with the command:

```
python autograder.py
```

It can be run for one particular question, such as q2, by:

```
python autograder.py -q q2
```

It can be run for one particular test by commands of the form:

```
python autograder.py -t test_cases/q2/1-bridge-grid
```

The code for this project is organized as follows:

What to submit

You will fill in portions of `valueIterationAgents.py`, `qlearningAgents.py`, and `analysis.py`. You should submit these files separately on Canvas and include in `writeup.pdf` the results of your autograder runs (screenshots). Also submit a zipped version of the code base with these files included – so we can run your solution if needed. Please do not change the other files in this distribution or submit any of our original files other than these files.

Filename	Edit?	Read?	Description
valueIterationAgents.py	Yes	Yes	A value iteration agent for solving known MDPs.
qlearningAgents.py	Yes	Yes	Q-learning agents for Gridworld, Crawler and Pacman.
analysis.py	Yes	Yes	A file to put your answers to questions given in the project.
mdp.py	No	Yes	Defines methods on general MDPs.
learningAgents.py	No	Yes	Defines the base classes ValueEstimationAgent and QLearningAgent, which your agents will extend.
util.py	No	Yes	Utilities, including util.Counter, which is particularly useful for Q-learners.
gridworld.py	No	Yes	The Gridworld implementation.
featureExtractors.py	No	Yes	Classes for extracting features on (state,action) pairs. Used for the approximate Q-learning agent (in qlearningAgents.py).
environment.py	No	No	Abstract class for general reinforcement learning environments. Used by gridworld.py.
graphicsGridworldDisplay.py	No	No	Gridworld graphical display.
graphicsUtils.py	No	No	Graphics utilities.
textGridworldDisplay.py	No	No	Plug-in for the Gridworld text interface.
crawler.py	No	No	The crawler code and test harness. You will run this but not edit it.
graphicsCrawlerDisplay.py	No	No	GUI for the crawler robot.
autograder.py	No	No	Project autograder
testParser.py	No	No	Parses autograder test and solution files
testClasses.py	No	No	General autograding test classes
test_cases/	No	No	Directory containing the test cases for each question
reinforcementTestClasses.py	No	No	Project 3 specific autograding test classes

Markov Decision Processes and Grid World

To get started, run Gridworld in manual control mode, which uses the arrow keys:

```
python gridworld.py -m
```

You will see the two-exit layout from class. The blue dot is the agent. Note that when you press up, the agent only actually moves north 80% of the time. Such is the life of a Gridworld agent!

You can control many aspects of the simulation. A full list of options is available by running:

```
python gridworld.py -h
```

The default agent moves randomly

```
python gridworld.py -g MazeGrid
```

You should see the random agent bounce around the grid until it happens upon an exit. Not the finest hour for an AI agent.

Note: The Gridworld MDP is such that you first must enter a pre-terminal state (the double boxes shown in the GUI) and then take the special 'exit' action before the episode actually ends (in the true terminal state called TERMINAL_STATE, which is not shown in the GUI). If you run an episode manually, your total return may be less than you expected, due to the discount rate (`-d` to change; 0.9 by default).

Look at the console output that accompanies the graphical output (or use `-t` for all text). You will be told about each transition the agent experiences (to turn this off, use `-q`).

As in Pacman, positions are represented by (x,y) Cartesian coordinates and any arrays are indexed by [x][y], with 'north' being the direction of increasing y, etc. By default, most transitions will receive a reward of zero, though you can change this with the living reward option (`-r`).

Problem 5.1 Value Iteration (10 points)

Write a value iteration agent in ValueIterationAgent, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run (option `-i`) in its initial planning phase. ValueIterationAgent takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.

Value iteration computes k-step estimates of the optimal values, V_k. In addition to running value iteration, implement the following methods for ValueIterationAgent using V_k.

- `computeActionFromValues(state)` computes the best action according to the value function given by `self.values`.
- `computeQValueFromValues(state, action)` returns the Q-value of the `(state, action)` pair given by the value function given by `self.values`.

These quantities are all displayed in the GUI: values are numbers in squares, Q-values are numbers in square quarters, and policies are arrows out from each square.

Important: Use the "batch" version of value iteration where each vector V_k is computed from a fixed vector V_{k-1} , not the "online" version where one single weight vector is updated in place. This means that when a state's value is updated in iteration k based on the values of its successor states, the successor state values used in the value update computation should be those from iteration $k-1$ (even if some of the successor states had already been updated in iteration k). The difference is discussed in Sutton & Barto in the 6th paragraph of chapter 4.1.

Note: A policy synthesized from values of depth k (which reflect the next k rewards) will actually reflect the next $k+1$ rewards (i.e. you return π_{k+1}). Similarly, the Q-values will also reflect one more reward than the values (i.e. you return Q_{k+1}). You should return the synthesized policy π_{k+1}.

Hint: Use the `util.Counter class` in `util.py`, which is a dictionary with a default value of zero. Methods such as `totalCount` should simplify your code. However, be careful with `argMax`: the actual argmax you want may be a key not in the counter!

Hint: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

To test your implementation, run the autograder:

```
python autograder.py -q q1
```

The following command loads your ValueIterationAgent, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (V(start), which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

```
python gridworld.py -a value -i 100 -k 10
```

Hint: On the default BookGrid, running value iteration for 5 iterations should give you the output in Figure 12.

```
python gridworld.py -a value -i 5
```

Problem 5.2 Bridge Crossing Analysis (2 points)

BridgeGrid is a grid world map with the a low-reward terminal state and a high-reward terminal state separated by a narrow "bridge", on either side of which is a chasm of high negative reward. The agent starts near the low-reward state. With the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. Change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in question2() of `analysis.py`. Noise refers to how often an agent ends up in an unintended successor state when they perform an action. The default corresponds to:

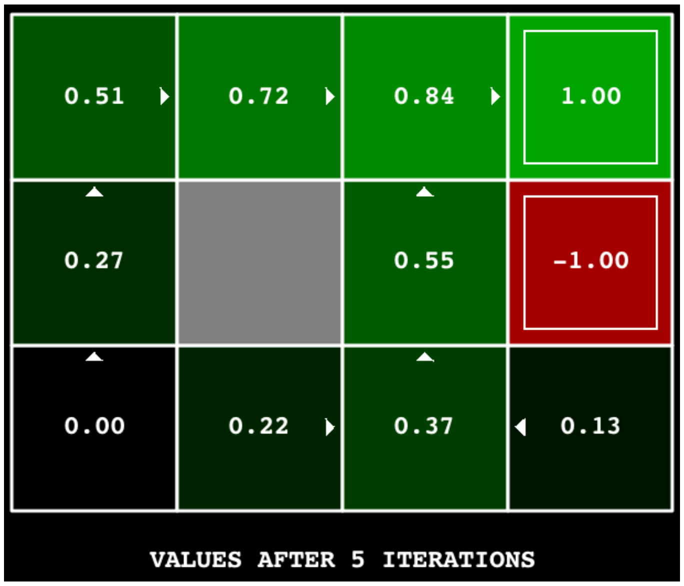

Figure 3: Results of value iteration on BookGrid

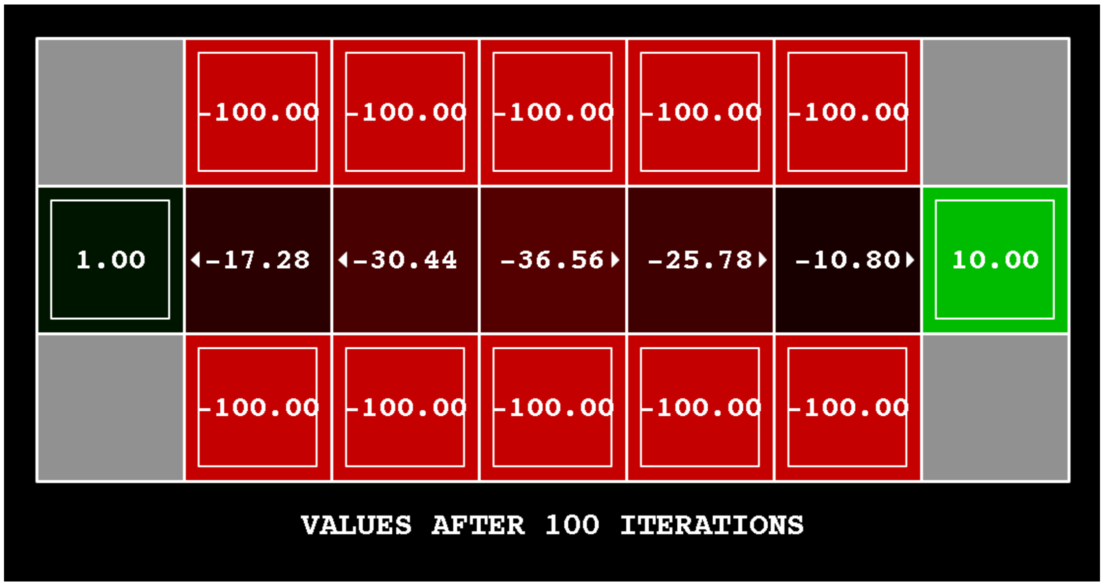

Figure 4: Bridge Crossing Analysis

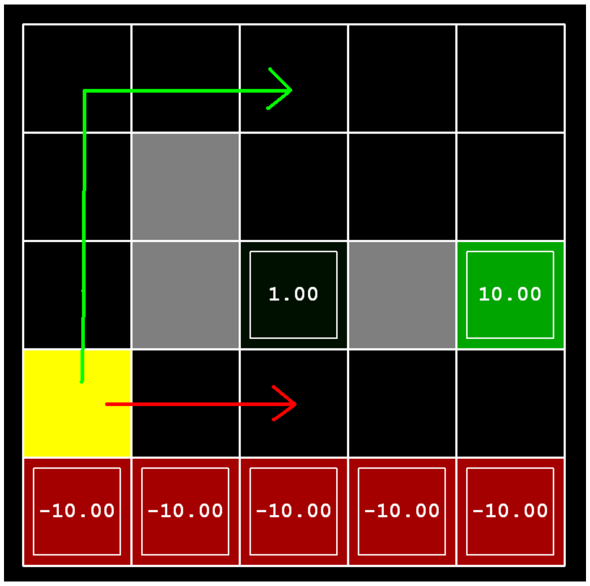

Figure 5: DiscountGrid Problem

```
python gridworld.py -a value -i 100 -g BridgeGrid --discount 0.9 --noise 0.2
```

Grading: We will check that you only changed one of the given parameters, and that with this change, a correct value iteration agent should cross the bridge. To check your answer, run the autograder:

```
python autograder.py -q q2
```

Problem 5.3 Policies (5 points)

Consider the DiscountGrid layout, shown below. This grid has two terminal states with positive payoff (in the middle row), a close exit with payoff +1 and a distant exit with payoff +10. The bottom row of the grid consists of terminal states with negative payoff (shown in red); each state in this "cliff" region has payoff -10. The starting state is the yellow square. We distinguish between two types of paths: (1) paths that "risk the cliff" and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in the figure below. (2) paths that "avoid the cliff" and travel along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. These paths are represented by the green arrow in the figure below.

In this question, you will choose settings of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several different types. Your setting of the parameter values for each part should have the property that, if your agent followed its optimal policy without being subject to any noise, it would exhibit the given behavior. If a particular behavior is not achieved for any setting of the parameters, assert that the policy is impossible by returning the string 'NOT POSSIBLE'.

Here are the optimal policy types you should attempt to produce:

- Prefer the close exit (+1), risking the cliff (-10)
- Prefer the close exit (+1), but avoiding the cliff (-10)
- Prefer the distant exit (+10), risking the cliff (-10)
- Prefer the distant exit (+10), avoiding the cliff (-10)
- Avoid both exits and the cliff (so an episode should never terminate)

To check your answers, run the autograder:

```
python autograder.py -q q3
```

question3a() through question3e() should each return a 3-item tuple of (discount, noise, living reward) in `analysis.py`.

Note: You can check your policies in the GUI. For example, using a correct answer to 3(a), the arrow in (0,1) should point east, the arrow in (1,1) should also point east, and the arrow in (2,1) should point north.

Note: On some machines you may not see an arrow. In this case, press a button on the keyboard to switch to qValue display, and mentally calculate the policy by taking the arg max of the available qValues for each state.

Grading: We will check that the desired policy is returned in each case.

5.1 Problem 5.4: Q-learning (10 points)

Note that your value iteration agent does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridword, but it's very important in the real world, where the real MDP is not available.

You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its `update(state, action, nextState, reward)` method. A stub of a Q-learner is specified in QLearningAgent in `qlearningAgents.py`, and you can select it with the option `-a q`. For this question, you must implement the update, `computeValueFromQValues`, `getQValue`, and `computeActionFromQValues` methods.

Note: For `computeActionFromQValues`, you should break ties randomly for better behavior. The `random.choice()` function will help. In a particular state, actions that your agent hasn't seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent has seen before have a negative Q-value, an unseen action may be optimal.

Important: Make sure that in your `computeValueFromQValues` and `computeActionFromQValues` functions, you only access Q values by calling `getQValue`. This abstraction will be useful for

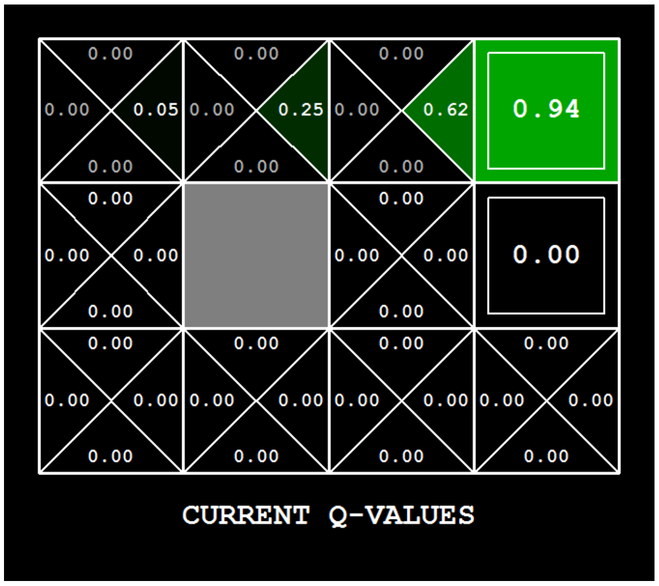

Figure 6: Q-learning

question 8 when you override `getQValue` to use features of state-action pairs rather than state-action pairs directly.

With the Q-learning update in place, you can watch your Q-learner learn under manual control, using the keyboard:

```
python gridworld.py -a q -k 5 -m
```

Recall that `-k` will control the number of episodes your agent gets to learn. Watch how the agent learns about the state it was just in, not the one it moves to, and "leaves learning in its wake." Hint: to help with debugging, you can turn off noise by using the `--noise 0.0` parameter (though this obviously makes Q-learning less interesting). If you manually steer Pacman north and then east along the optimal path for four episodes, you should see the following Q-values:

Grading: We will run your Q-learning agent and check that it learns the same Q-values and policy as our reference implementation when each is presented with the same set of examples. To grade your implementation, run the autograder:

```
python autograder.py -q q4
```

Problem 5.5: Epsilon-greedy policies (5 points)

Complete your Q-learning agent by implementing epsilon-greedy action selection in `getAction`, meaning it chooses random actions an epsilon fraction of the time, and follows its current best Q-values otherwise. Note that choosing a random action may result in choosing the best action - that is, you should not choose a random sub-optimal action, but rather any random legal action.

```
python gridworld.py -a q -k 100
```

Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than the Q-values predict because of the random actions and the initial learning phase.

You can choose an element from a list uniformly at random by calling the random.choice function. You can simulate a binary variable with probability p of success by using `util.flipCoin(p)`, which returns True with probability `p` and False with probability `1-p`.

To test your implementation, run the autograder:

```
python autograder.py -q q5
```

With no additional code, you should now be able to run a Q-learning crawler robot:

```
python crawler.py
```

If this doesn't work, you've probably written some code too specific to the GridWorld problem and you should make it more general to all MDPs.

This will invoke the crawling robot from class using your Q-learner. Play around with the various learning parameters to see how they affect the agent's policies and actions. Note that the step delay is a parameter of the simulation, whereas the learning rate and epsilon are parameters of your learning algorithm, and the discount factor is a property of the environment.

Problem 5.6: Bridge Crossing Revisited (1 point)

First, train a completely random Q-learner with the default learning rate on the noiseless BridgeGrid for 50 episodes and observe whether it finds the optimal policy.

```
python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e 1
```

Now try the same experiment with an epsilon of 0. Is there an epsilon and a learning rate for which it is highly likely (greater than 99%) that the optimal policy will be learned after 50 iterations? `question6()` in `analysis.py` should return EITHER a 2-item tuple of (epsilon, learning rate) OR the string 'NOT POSSIBLE' if there is none. Epsilon is controlled by `-e`, learning rate by `-l`.

Note: Your response should be not depend on the exact tie-breaking mechanism used to choose actions. This means your answer should be correct even if for instance we rotated the entire bridge grid world 90 degrees.

To grade your answer, run the autograder:

```
python autograder.py -q q6
```

Problem 5.7: Q-learning and Pacman (2 points)

Time to play some Pacman! Pacman will play games in two phases. In the first phase, training, Pacman will begin to learn about the values of positions and actions. Because it takes a very long time to learn accurate Q-values even for tiny grids, Pacman's training games run in quiet mode by default, with no GUI (or console) display. Once Pacman's training is complete, he will enter testing mode. When testing, Pacman's `self.epsilon` and `self.alpha` will be set to 0.0, effectively stopping Q-learning and disabling exploration, in order to allow Pacman to exploit his learned policy. Test games are shown in the GUI by default. Without any code changes you should be able to run Q-learning Pacman for very tiny grids as follows:

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

Note that `PacmanQAgent` is already defined for you in terms of the `QLearningAgent` you've already written. `PacmanQAgent` is only different in that it has default learning parameters that are more effective for the Pacman problem (epsilon=0.05, alpha=0.2, gamma=0.8). You will receive full credit for this question if the command above works without exceptions and your agent wins at least 80% of the time. The autograder will run 100 test games after the 2000 training games.

Hint: If your `QLearningAgent` works for `gridworld.py` and `crawler.py` but does not seem to be learning a good policy for Pacman on smallGrid, it may be because your `getAction` and/or `computeActionFromQValues` methods do not in some cases properly consider unseen actions. In particular, because unseen actions have by definition a Q-value of zero, if all of the actions that have been seen have negative Q-values, an unseen action may be optimal. Beware of the `argmax` function from `util.Counter`!

Note: To grade your answer, run:

```
python autograder.py -q q7
```

Note: If you want to experiment with learning parameters, you can use the option `-a`, for example `-a epsilon=0.1,alpha=0.3,gamma=0.7`. These values will then be accessible as `self.epsilon`, `self.gamma` and `self.alpha` inside the agent.

Note: While a total of 2010 games will be played, the first 2000 games will not be displayed because of the option `-x 2000`, which designates the first 2000 games for training (no output). Thus, you will only see Pacman play the last 10 of these games. The number of training games is also passed to your agent as the option `numTraining`.

Note: If you want to watch 10 training games to see what's going on, use the command:

```
python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10
```

During training, you will see output every 100 games with statistics about how Pacman is faring. Epsilon is positive during training, so Pacman will play poorly even after having learned a good policy: this is because he occasionally makes a random exploratory move into a ghost. As a benchmark, it should take between 1000 and 1400 games before Pacman's rewards for a 100 episode segment becomes positive, reflecting that he's started winning more than losing. By the end of training, it should remain positive and be fairly high (between 100 and 350).

Make sure you understand what is happening here: the MDP state is the exact board configuration facing Pacman, with the now complex transitions describing an entire ply of change to that state. The intermediate game configurations in which Pacman has moved but the ghosts have not replied are not MDP states, but are bundled in to the transitions. Once Pacman is done training, he should win very reliably in test games (at least 90% of the time), since now he is exploiting his learned policy. However, you will find that training the same agent on the seemingly simple mediumGrid does not work well. In our implementation, Pacman's average training rewards remain negative throughout training. At test time, he plays badly, probably losing all of his test games. Training will also take a long time, despite its ineffectiveness. Pacman fails to win on larger layouts because each board configuration is a separate state with separate Q-values. He has no way to generalize that running into a ghost is bad for all positions. Obviously, this approach will not scale.

Problem 5.8: Approximate Q-learning (5 points)

Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation in `ApproximateQAgent` class in `qlearningAgents.py`, which is a subclass of `PacmanQAgent`.

Note: Approximate Q-learning assumes the existence of a feature function $f(s, a)$ over state and action pairs, which yields a vector $f_1(s, a), \ldots, f_i(s, a), \ldots, f_n(s, a)$ of feature values. We provide feature functions for you in `featureExtractors.py`. Feature vectors are `util.Counter` (like a dictionary) objects containing the non-zero pairs of features and values; all omitted features have value zero.

The approximate Q-function takes the following form

$$Q(s, a) = \sum_{i=1}^{i=n} w_i f_i(s, a)$$

where each weight w_i is associated with a particular feature $f_i(s, a)$. In your code, you should implement the weight vector as a dictionary mapping features (which the feature extractors will return) to weight values. You will update your weight vectors similarly to how you updated Q-values:

$$\begin{aligned} w_i &\leftarrow w_i + \alpha * difference * f_i(s, a) \\ difference &= [r + \gamma max_{a'} Q(s', a^{'})] - Q(s, a) \end{aligned}$$

Note that *difference* is the same as for the Q-learning updating with the $(s, a, r, s^{'})$ tuple.

By default, `ApproximateQAgent` uses the `IdentityExtractor`, which assigns a single feature to every (state,action) pair. With this feature extractor, your approximate Q-learning agent should work identically to `PacmanQAgent`. You can test this with the following command:

```
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid
```

Important: `ApproximateQAgent` is a subclass of `QLearningAgent`, and it therefore shares several methods like `getAction`. Make sure that your methods in `QLearningAgent` call `getQValue` instead of accessing Q-values directly, so that when you override `getQValue` in your approximate agent, the new approximate q-values are used to compute actions.

Once you're confident that your approximate learner works correctly with the identity features, run your approximate Q-learning agent with our custom feature extractor, which can learn to win with ease:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid
```

Even much larger layouts should be no problem for your ApproximateQAgent. (warning: this may take a few minutes to train)

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassi
```

If you have no errors, your approximate Q-learning agent should win almost every time with these simple features, even with only 50 training games.

Grading: We will run your approximate Q-learning agent and check that it learns the same Q-values and feature weights as our reference implementation when each is presented with the same set of examples. To grade your implementation, run the autograder:

```
python autograder.py -q q8
```