Machine Learning	**Assignment 6**
Comp540 Spring 2020	**due April 20 2020 at 8 pm**

This homework is due April 20 at 8 pm on Canvas. The code base `hw6.zip` *for the assignment is an attachment to Assignment 6 on Canvas. You will add your code at the indicated spots in the files there. Place your answers to Problems 1, 2, 3 (typeset) in a file called* `writeup.pdf`. *Problem 6 on reinforcement learning is an extra credit problem. When you submit, please submit the following items as separate attachments before the due date and time:*

- *Your writeup pdf, named* `writeup.pdf`.
- *Your jupyter notebooks, saved in HTML format. Submit notebooks for Problems 4 and 5 separately.*
- *All the* `.py` *files that you modified, each one as a separate upload.*
- *The zip file containing your work (notebooks + .py). Be sure that the datasets you use are NOT part of the zip, as this will increase the size considerably.*
- *For Problem 6, in* `writeup.pdf` *attach screen shots of the output of the autograder on all 8 parts of that problem.*

1 Hidden Markov Models (10 points)

We are tracking the evolution of a patient's health $\{healthy, unhealthy\}$ over time periods $0, 1, \ldots$ by taking measurements of P (a medical test) with values $\{low, medium, high\}$. Measurements are taken at every time step $1, \ldots$ and will be used to infer the patient's health. The value of the observed test at time t is a stochastic function of the patient's health at t. When the patient is healthy, the probability of observing a low test reading is 0.5, a medium test reading is 0.3, and a high test reading is 0.2. When the patient is unhealthy, the probability of observing a low test reading is 0.3, a medium test reading is 0.3 and a high test reading is 0.4. At time 0, the patient's condition is unknown – we will assume that it is equally likely that he is healthy or unhealthy. There is a probability of 0.2 that there is a health state change between time steps t and $t+1$, between values *healthy* and *unhealthy*.

- (3 points) Formulate this problem as a Hidden Markov model identifying the set of observables O, the set of hidden states S and the parameters $\lambda = [\pi, a, b]$, the initial state distribution, the transition matrix and the emission matrix. Draw the corresponding graphical model.
- (3 points) Calculate the probability that the patient is healthy at $t = 2$ given that the test readings at $t = 1$ and $t = 2$ are both low.
- (4 points) What is the most likely state sequence for $t = 0, 1, 2$ given the evidence from the previous subpart?

2 EM for mixtures of Bernoullis (10 points)

- (5 points) Show that the M step for ML estimation of a mixture of Bernoullis is given by

$$\mu_{kj} = \frac{\sum_{i=1}^{m} r_k^{(i)} x_j^{(i)}}{\sum_{i=1}^{m} r_k^{(i)}}$$

- (5 points) Show that the M step for the MAP estimation of a mixture of Bernoullis with a $Beta(\alpha, \beta)$ prior is given by

$$\mu_{kj} = \frac{(\sum_{i=1}^{m} r_k^{(i)} x_j^{(i)}) + \alpha + 1}{(\sum_{i=1}^{m} r_k^{(i)}) + \alpha + \beta - 2}$$

3 Principal Components Analysis (10 points)

In class, we showed that PCA finds the variance maximizing directions onto which to project the data. In this problem, we find another interpretation of PCA. Suppose we are given a set of points $\{x^{(1)}, \ldots, x^{(m)}\}$. Let us assume that we have preprocessed the data to have zero-mean and unit variance in each dimension. For a given unit-length vector u, let $f_u(x)$ be the projection of point x into the direction given by u. Let $V = \{\alpha u : \alpha \in \Re\}$. Then,

$$f_u(v) = argmin_{v \in V} ||x - v||^2$$

Show that the unit length vector u that minimizes the mean squared error between projected points and original points corresponds to the first principal component for the data, i.e., show that

$$argmin_{u:uu^T=1} \sum_{i=1}^{m} ||x^{(i)} - f_u(x^{(i)})||^2$$

gives the first principal component of the data. Hint: First show that $f_u(x) = u^T x u$.

4 K-means clustering (10 points)

In this problem, you will implement the k-means algorithm and use it for image compression. You will first start on an example 2D dataset that will help you gain intuition about how the k-means algorithm works. After that, you will use the k-means algorithm for image compression by reducing the number of colors that occur in an image to only those that are most common in that image. You will be using `kmeans.ipynb` for this part. The relevant files for this part are in the folder `kmeans`.

Name	**Edit?**	**Read?**	**Description**
kmeans.ipynb	No	Yes	Python notebook that will run your functions for k-means clustering
utils_kmeans.py	Yes	Yes	Functions for implementing k-means
kmeansdata2.mat	No	No	Simple 2D Example dataset for k-means
bird_small.png	No	No	Example image for compression by k-means
hw6.pdf	No	Yes	this document

Implementing k-means

The k-means algorithm is a method to automatically cluster similar examples together. That is, given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ (where $x^{(i)} \in \Re^d$), k-means groups the data into a few cohesive clusters. The intuition behind k-means is an iterative procedure that starts by guessing the initial cluster centroids, and then refines this guess by repeatedly assigning examples to their closest centroids and then recomputing the centroids based on the assignments. The k-means algorithm is as follows:

```
# Initialize centroids
centroids = kmeans_init_centroids(X, K)
for iter in range(iterations):
     # Cluster assignment step: Assign each data point to the
     # closest centroid. idx[i]  is the index
     # of the centroid assigned to example i
     idx = find_closest_centroids(X, centroids)
     # Move centroid step: Compute means based on centroid
     # assignments
     centroids = compute_centroids(X, idx, K);
```

The inner-loop of the algorithm repeatedly carries out two steps: (i) Assigning each training example $x^{(i)}$ to its closest centroid, and (ii) Recomputing each centroid using the points assigned to it. The k-means algorithm will always converge to some final set of centroids. Note that the converged solution may not always be ideal and will depend on the initial setting of the centroids. Therefore, in practice the k-means algorithm is usually run a few times with different random initializations. One way to choose between these different solutions from different random initializations is to choose the one with the lowest cost function (distortion). You will implement the two phases of the k-means algorithm separately in the next sections.

Problem 3.1: Finding closest centroids (4 points)

In the cluster assignment phase of the k-means algorithm, the algorithm assigns every training example $x^{(i)}$ to its closest centroid, given the current positions of centroids. Specifically, for every example i we set

$$c^{(i)} = j \quad \text{that minimizes } ||x^{(i)} - \mu_j||^2$$

where $c^{(i)}$ is the index of the centroid that is closest to $x^{(i)}$, and μ_j is the position (value) of the j^{th} centroid. Note that $c^{(i)}$ corresponds to `idx[i]` in our code.

Your task is to complete the function `find_closest_centroids` in `utils_kmeans.py`. This function takes the data matrix `X` and the locations of all centroids inside `centroids` and outputs a one-dimensional array `idx` that holds the index (a value in {0, ...,K-1}, where K is total number of centroids) of the closest centroid to every training example. You can implement this using a loop over every training example and every centroid.

When you run the appropriate cell of the notebook `kmeans.ipynb` you should see the output [0 2 1] corresponding to the centroid assignments for the first 3 examples in our data set.

Problem 3.2: Computing centroid means (3 points)

Given assignments of every point to a centroid, the second phase of the algorithm recomputes, for each centroid, the mean of the points that were assigned to it. Specifically, for every centroid j we set

$$\mu_j = \frac{1}{|C_j|} \sum_{i \in C_j} x^{(i)}$$

where C_j is the set of examples that are assigned to centroid j.

You should now complete the function `compute_centroids` in `utils_kmeans.py`. You can implement this function using a loop over the centroids. You can also use a loop over the examples; but if you can use a vectorized implementation that does not use such a loop, your code should run faster. Once you have completed the function, the appropriate cell in `means.ipynb` will run your function and output the centroids after the first step of k-means.

k-means on example dataset

After you have completed the two functions (`find_closest_centroids` and `compute_centroids`), the next step in `means.ipynb` will run the k-means algorithm on a toy 2D dataset to help you understand how k-means works. Your functions are called from inside the `run_kmeans` function in `utils_kmeans.py`. We encourage you to take a look at the function to understand how it works. Notice that the function calls the two functions you implemented in a loop. When you run the next step, the k-means code will produce a visualization that steps you through the progress of the algorithm at each iteration. At the end, your figure should look as the one displayed in Figure 1.

Problem 3.3: Random initialization (3 points)

The initial assignments of centroids for the example dataset were designed so that you will see the same figure as in Figure 1. In practice, a good strategy for initializing the centroids is to select random examples from the training set. In this part of the exercise, you should complete the function `kmeans_init_centroids` in `utils_kmeans.py`. First, randomly permute the indices of the examples. Then, select the first `K` examples based on the random permutation of the indices. This allows the examples to be selected at random without the risk of selecting the same example twice.

Image compression with k-means

In this exercise, you will apply k-means to image compression. In a straightforward 24-bit color representation of an image, each pixel is represented as three 8-bit unsigned integers (ranging from 0 to 255) that specify the red, green and blue intensity values. This encoding is often refered to as the RGB encoding. Our image contains thousands of colors, and in this part of the exercise, you will reduce the number of colors to 16 colors.

By making this reduction, it is possible to represent (compress) the photo in an efficient way. Specifically, you only need to store the RGB values of the 16 selected colors, and for each pixel in the image you now need to only store the index of the color at that location (where only 4 bits

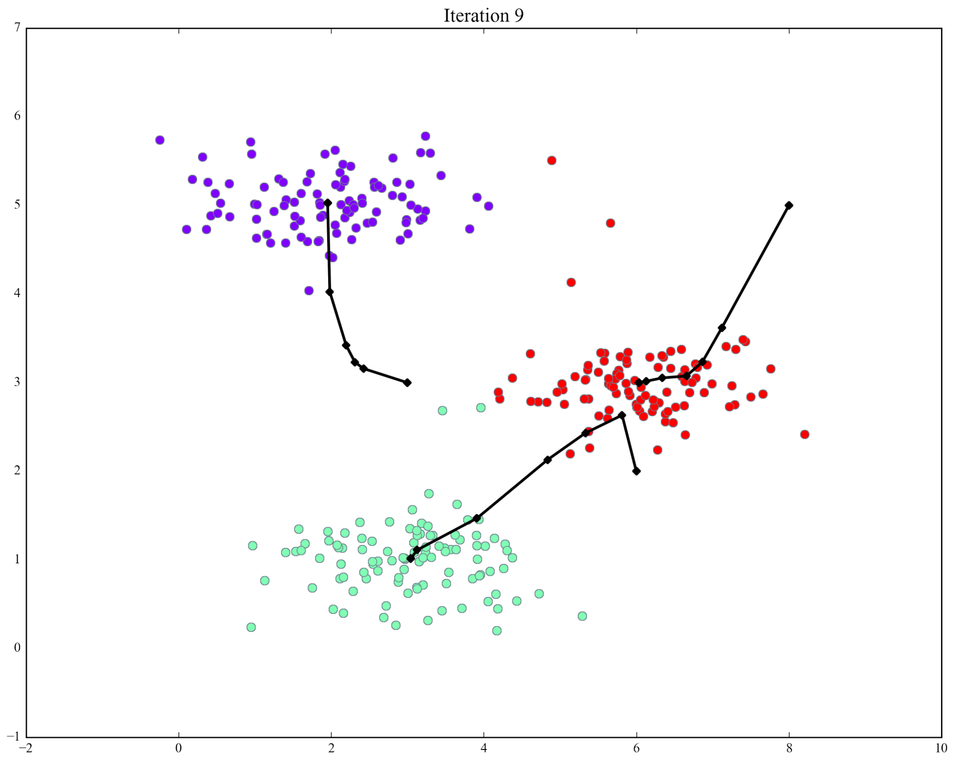

Figure 1: Expected output of k-means

are necessary to represent 16 possibilities). In this exercise, you will use the k-means algorithm to select the 16 colors that will be used to represent the compressed image. In particular, you will treat every pixel in the original image as a data example and use the K-means algorithm to find the 16 colors that best group (cluster) the pixels in the 3- dimensional RGB space. Once you have computed the cluster centroids on the image, you will then use the 16 colors to replace the pixels in the original image.

k-means on pixels

A cell in `kmeans.ipynb` first loads the image, and then reshapes it to create an $m \times 3$ matrix of pixel colors (where $m = 16384 = 128 \times 128$), and calls your k-means function on it. After finding the top K $= 16$ colors to represent the image, you can now assign each pixel position to its closest centroid using the `find_closest_centroids` function. This allows you to represent the original image using the centroid assignments of each pixel. Notice that you have significantly reduced the number of bits that are required to describe the image. The original image required 24 bits for each one of the 128×128 pixel locations, resulting in total size of $128 \times 128 \times 24 = 393,216$ bits. The new representation requires some overhead storage in form of a dictionary of 16 colors, each of which require 24 bits, but the image itself then only requires 4 bits per pixel location. The final number of bits used is therefore $16 \times 24 + 128 \times 128 \times 4 = 65,920$ bits, which corresponds to compressing the original image by about a factor of 6.

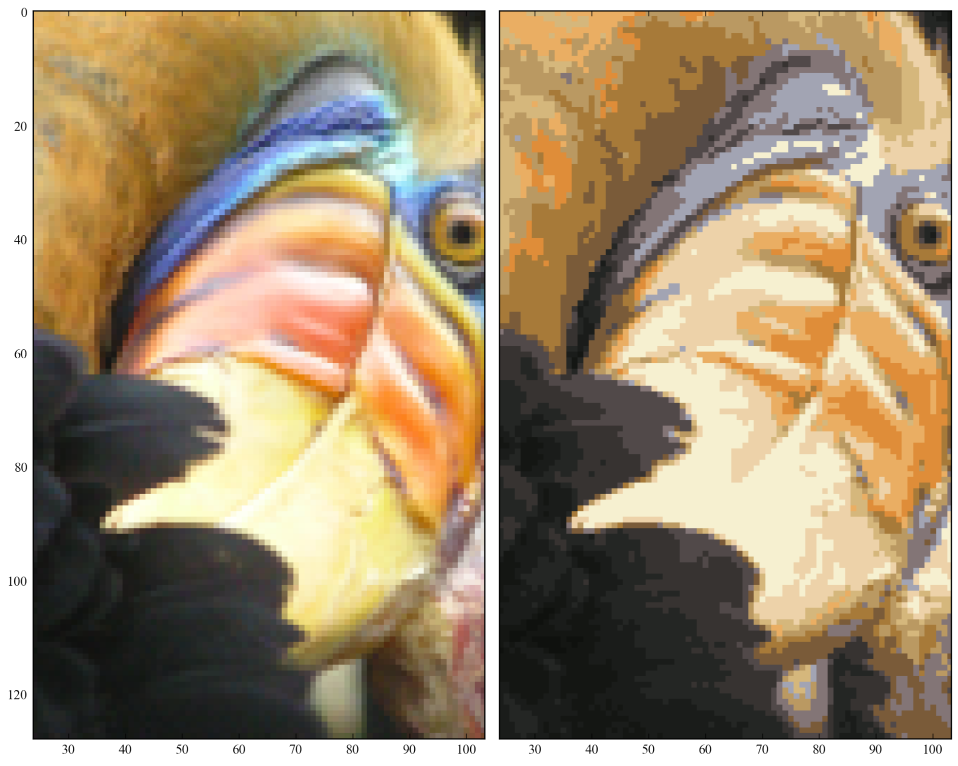

Figure 2: Original and reconstructed image (when using k-means to compress the image).

Finally, you can view the effects of the compression by reconstructing the image based only on the centroid assignments. Specifically, you can replace each pixel location with the mean of the centroid assigned to it. Figure 2 shows the reconstruction we obtained. Even though the resulting image retains most of the characteristics of the original, we also see some compression artifacts.

5 Principal Components Analysis (10 points)

In this exercise, you will use principal component analysis (PCA) to perform dimensionality reduction. You will first experiment with an example 2D dataset to get intuition on how PCA works, and then use it on a bigger dataset of 5000 faces. The notebook `pca.ipynb`, will help you step through the exercise.

Name	**Edit?**	**Read?**	**Description**
pca.ipynb	Yes	Yes	Python notebook for running your functions for principal component analysis
utils_pca.py	Yes	Yes	functions for implementing PCA
pcadata1.mat	No	No	Simple 2D Example dataset for PCA
pcafaces.mat	No	No	faces dataset for PCA
hw6.pdf	No	Yes	this document

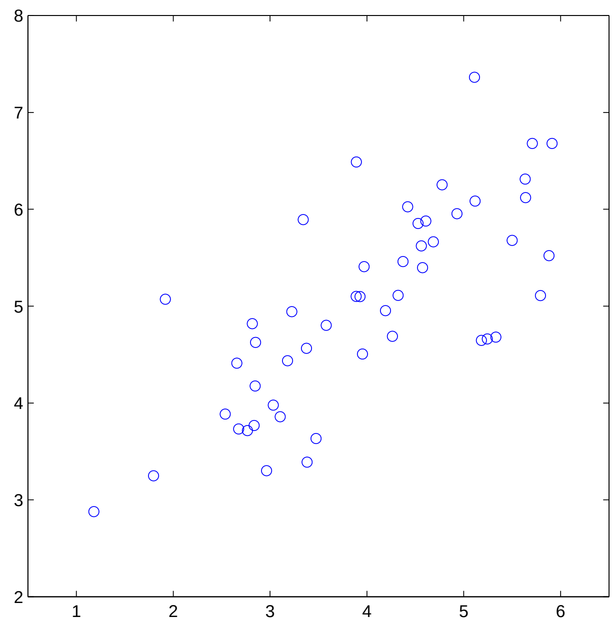

Figure 3: Example Dataset 1

Example Dataset

To help you understand how PCA works, you will first start with a 2D dataset which has one direction of large variation and one of smaller variation. The notebook `pca.ipynb` will plot the training data (Figure 3). In this part of the exercise, you will visualize what happens when you use PCA to reduce the data from 2D to 1D. In practice, you might want to reduce data from 256 to 50 dimensions, say; but using lower dimensional data in this example allows us to visualize the algorithms better.

Problem 4.1: Implementing PCA (4 points)

In this part of the exercise, you will implement PCA. PCA consists of two computational steps: First, you compute the covariance matrix of the data. Then, you use `numpy`'s SVD function to compute the eigenvectors $U_1, U_2, \ldots, U_n$. These will correspond to the principal components of variation in the data.

Before using PCA, it is important to first normalize the data by subtracting the mean value of each feature from the dataset, and scaling each dimension so that they are in the same range. The notebook `pca.ipynb`, does this normalization for you using the `feature_normalize` function. After normalizing the data, you can run PCA to compute the principal components. You task is to complete the function `pca` in `utils_pca.py` to compute the principal components of the dataset.

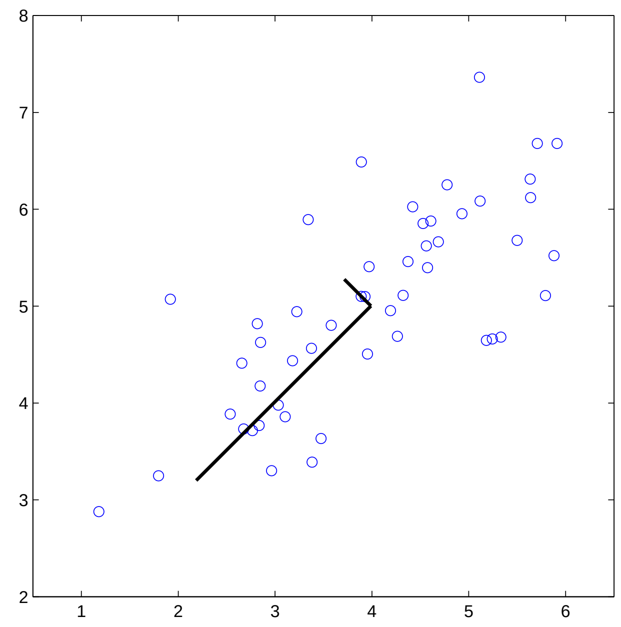

Figure 4: Computed eigenvectors of the dataset

First, you should compute the covariance matrix of the data, which is given by:

$$\Sigma = \frac{1}{m} X^T X$$

where X is the data matrix with examples in rows, and m is the number of examples. Note that Σ is a $d \times d$ matrix and not the summation operator.

After computing the covariance matrix, you can run SVD on it to compute the principal components. In `numpy`, you can run SVD with the following command:

```
U, S, V = np.linalg.svd(Sigma,full_matrices = False)
```

where `U` will contain the principal components and `S` will contain a diagonal matrix.

Once you have completed the function `pcs`, the `pca.ipynb` notebook will run PCA on the example dataset and plot the corresponding principal components found (Figure 4). The script will also output the top principal component (eigenvector) found, and you should expect to see an output of about [-0.707 -0.707]. (It is possible that `numpy` may instead output the negative of this, since U_1 and $-U_1$ are equally valid choices for the first principal component.)

Dimensionality reduction with PCA

After computing the principal components, you can use them to reduce the feature dimension of your dataset by projecting each example onto a lower dimensional space, $x^{(i)} \to z^{(i)}$ (e.g., projecting the data from 2D to 1D). In this part of the exercise, you will use the eigenvectors returned by PCA and project the example dataset into a 1-dimensional space. In practice, if you were using a learning algorithm such as linear regression or perhaps neural networks, you could now use the projected data instead of the original data. By using the projected data, you can train your model faster as there are fewer dimensions in the input.

Problem 4.2: Projecting the data onto the principal components (3 points)

You should now complete the function `project_data` in `utils_pca.py`. Specifically, you are given a dataset `X`, the principal components `U`, and the desired number of dimensions to reduce to `K`. You should project each example in `X` onto the top `K` components in `U`. Note that the top `K` components in `U` are given by the first `K` columns of `U`. Once you have completed `project_data`, the `pica.ipynb` notebook will project the first example onto the first dimension and you should see a value of about 1.481 (or possibly -1.481, if you got $-U_1$ instead of U_1).

Problem 4.3: Reconstructing an approximation of the data (3 points)

After projecting the data onto the lower dimensional space, you can approximately recover the data by projecting them back onto the original high dimensional space. Your task is to complete the function `recover_data` in `utils_pca.py` to project each example in `Z` back onto the original space and return the recovered approximation in `X_rec`. Once you have completed the function `recover_data`, `pca.ipynb` will recover an approximation of the first example and you should see a value of about [-1.047 -1.047].

Visualizing the projections

After completing both `project_data` and `recover_data`, `pca.ipynb` will now perform both the projection and approximate reconstruction to show how the projection affects the data. In Figure 5, the original data points are indicated with the blue circles, while the projected data points are indicated with the red circles. The projection effectively only retains the information in the direction given by U_1.

Face image dataset

In this part of the exercise, you will run PCA on face images to see how it can be used in practice for dimension reduction. The dataset `pcafaces.mat` contains a dataset `X` of face images, each 32×32 in grayscale. Each row of `X` corresponds to one face image (a row vector of length 1024). The next cell in `pcaipynb` will load and visualize the first 100 of these face images (Figure 6).

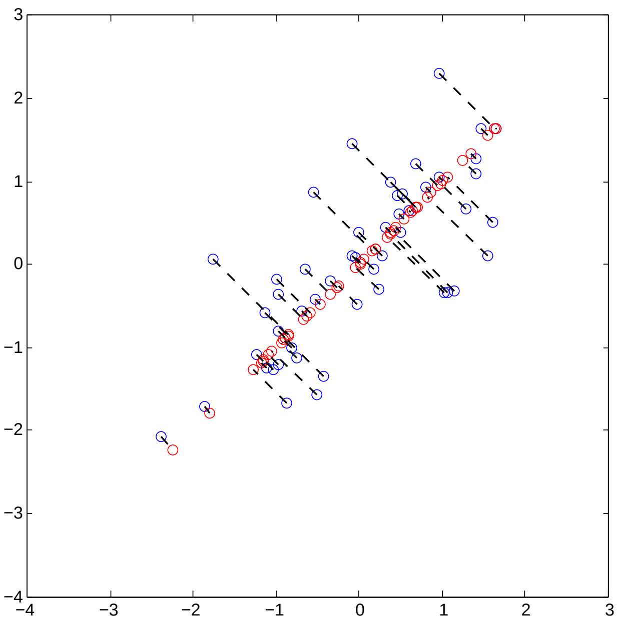

Figure 5: The normalized and projected data after PCA

PCA on faces

To run PCA on the face dataset, we first normalize the dataset by subtracting the mean of each feature from the data matrix `X`. The notebook `pca.ipynb` will do this for you and then run your PCA function. After running PCA, you will obtain the principal components of the dataset. Notice that each principal component in U (each row) is a vector of length d (where for the face dataset, $d = 1024$). It turns out that we can visualize these principal components by reshaping each of them into a 32×32 matrix that corresponds to the pixels in the original dataset. The notebook `pca.ipynb` displays the first 25 principal components that describe the largest variations (Figure 7). If you want, you can also change the code to display more principal components to see how they capture more and more details.

Dimensionality reduction

Now that you have computed the principal components for the face dataset, you can use it to reduce the dimension of the face dataset. This allows you to use your learning algorithm with a smaller input size (e.g., 100 dimensions) instead of the original 1024 dimensions. This can help speed up your learning algorithm.

The next part in `pca.ipynb` will project the face dataset onto only the first 100 principal components. Concretely, each face image is now described by a vector $z^{(i)} \in \Re^{100}$. To understand

Figure 6: Faces dataset

what is lost in the dimension reduction, you can recover the data using only the projected dataset. In `pica.ipynb`, an approximate recovery of the data is performed and the original and projected face images are displayed side by side (Figure 8). From the reconstruction, you can observe that the general structure and appearance of the face are kept while the fine details are lost. This is a remarkable reduction (more than 10×) in the dataset size that can help speed up your learning algorithm significantly. For example, if you were training a neural network to perform person recognition (given a face image, predict the identitfy of the person), you can use the dimension reduced input of only a 100 dimensions instead of the original pixels.

6 Reinforcement Learning (extra credit) (40 points)

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
```

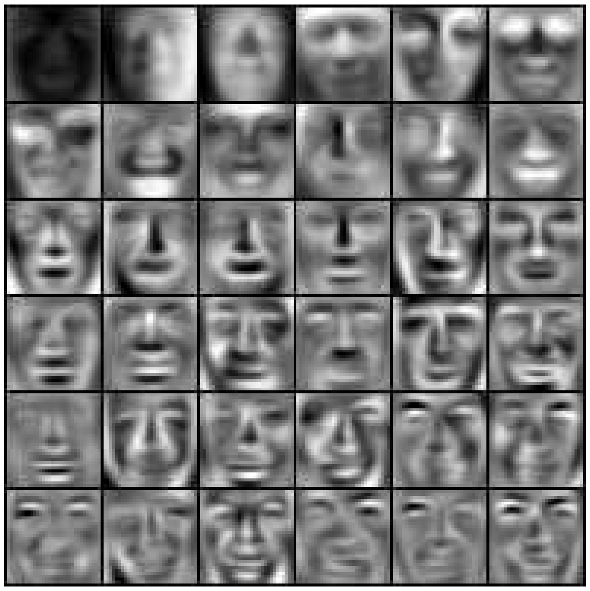

Figure 7: Principal components on the face dataset

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.

Markov Decision Processes and Grid World

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

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

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

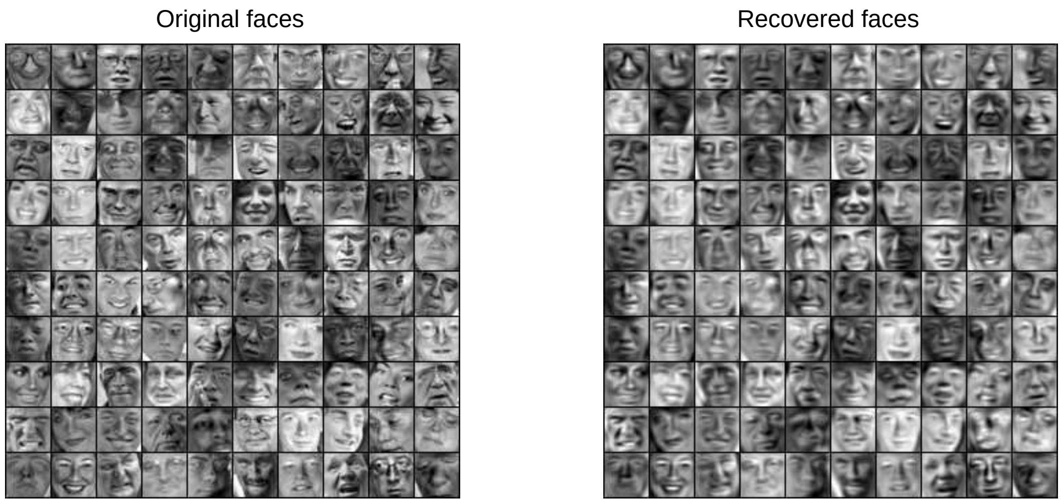

Figure 8: Original and reconstructed face dataset reconstructed from only the top 100 principal components

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 7.1 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
```

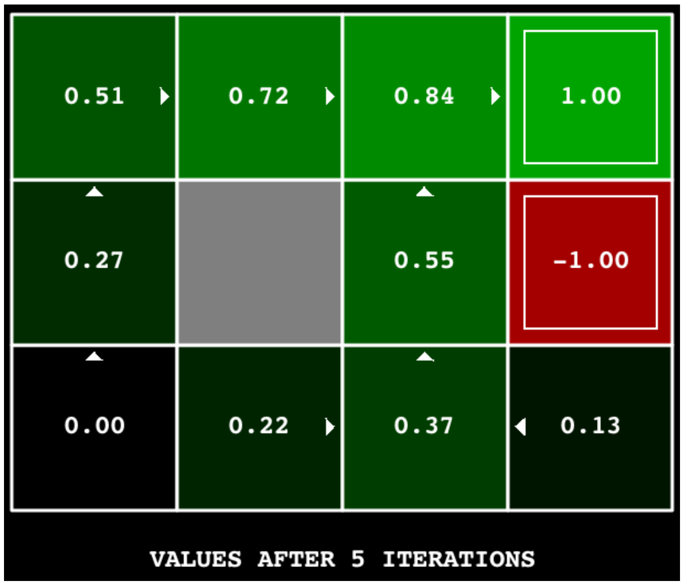

Figure 9: Results of value iteration on BookGrid

Problem 7.2 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:

```
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 7.3 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

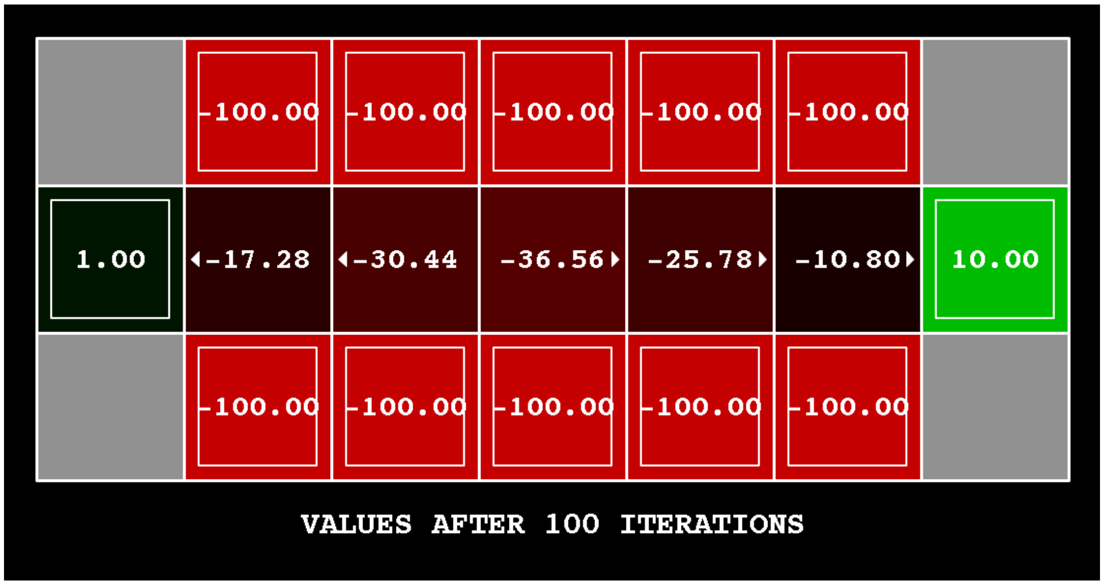

Figure 10: Bridge Crossing Analysis

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`.

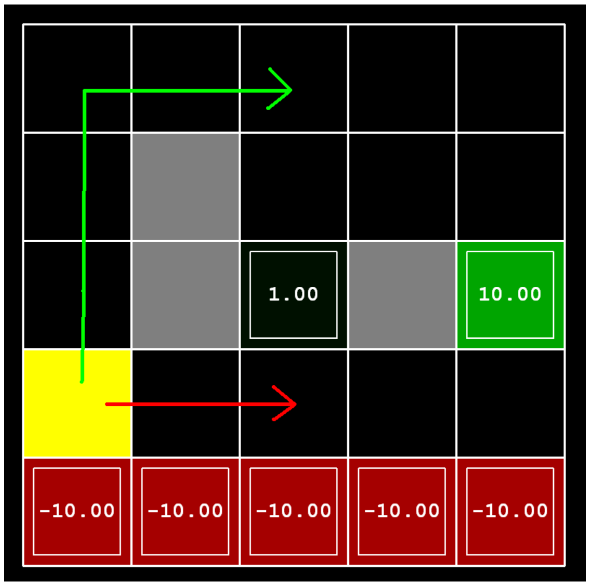

Figure 11: DiscountGrid Problem

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.

Problem 7.4: 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.

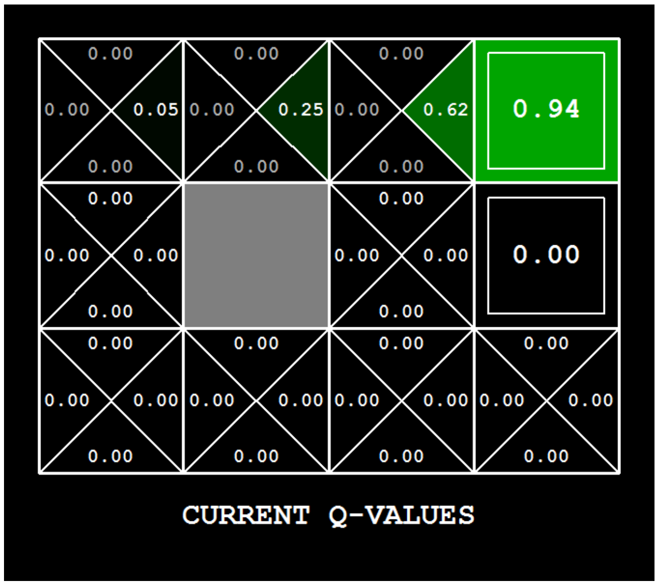

Figure 12: Q-learning

Important: Make sure that in your `computeValueFromQValues` and `computeActionFromQValues` functions, you only access Q values by calling `getQValue`. This abstraction will be useful for 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 7.5: 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 7.6: 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 7.7: 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 7.8: 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
```