CS 181 Final Project Writeup

Team Name: KDA9000
Team Members: Kewei Li, Ding Zhou, Andy Shi

May 9, 2014

In order to write an effective pacman agent, one has to use many techniques from CS 181 to classify the ghosts and capsules and train the pacman agent. We tried a few techniques to accomplish this task, and wrote an agent which combined k-means clustering, SVM classification, and Q-learning to play pacman reasonably effectively.

1 Data Collection

We implemented data collection with the help of the `-d` option for the pacman game. We noticed that ghosts would periodically respawn, but capsules only respawned if pacman ate one of the capsules (then all the capsules would respawn). To get the most capsule and ghost data, we wrote a pacman agent, `DataCollectorAgent`, which would greedily eat the capsule closest to it. Running this agent with logging enabled allowed us to collect data on over 1 million capsules and almost 500,000 ghosts. However, we noticed that the capsule data did not include labels, that is, whether or not the capsule was a placebo. We were very reluctant to just use the `getGoodCapsuleExamples` function to give us labels for the good capsules, since for each random seed, that function only returned 5 capsules. Instead, we tried to collect labeled data on the capsules. To achieve this, we wrote a pacman agent that printed out a capsule's feature vector when it was going to eat the capsule, and then printed information on whether or not there was a scared ghost present. We then piped this output into a file, and we were able to collect around 500,000 data points this way [1].

2 Classifying Capsules

As a first step to classifying capsules, we thought it would be a good idea to plot out capsule data. Plotting was especially convenient because the feature vector of the capsule was three-dimensional. The plot is shown in Figure 1. As we can see, the capsules cluster nicely into three clusters, but as shown by the red and blue dots, the capsules we thought were placebos and the capsules we thought were real seemed to be linearly inseparable. Using a variety of different classification techniques, including logistic regression, naive Bayes, decision trees, and SVMs, we only got an overall accuracy rate of 66% training on 60% of the data and holding out 40% of the data for testing. We tried a few transformations, including squaring each term in the feature vector, but none of these transformations produced linearly inseparable clusters of red and blue. However,

[1]The number of data points was determined using the `wc` command line utility.

when we used `getGoodCapsuleExamples` and plotted the feature vectors of these capsules for a few different random seeds, we found that they all came from one cluster. This led us to think there was a bug in the code that was processing our collected data, so we decided to go for an approach using clustering.

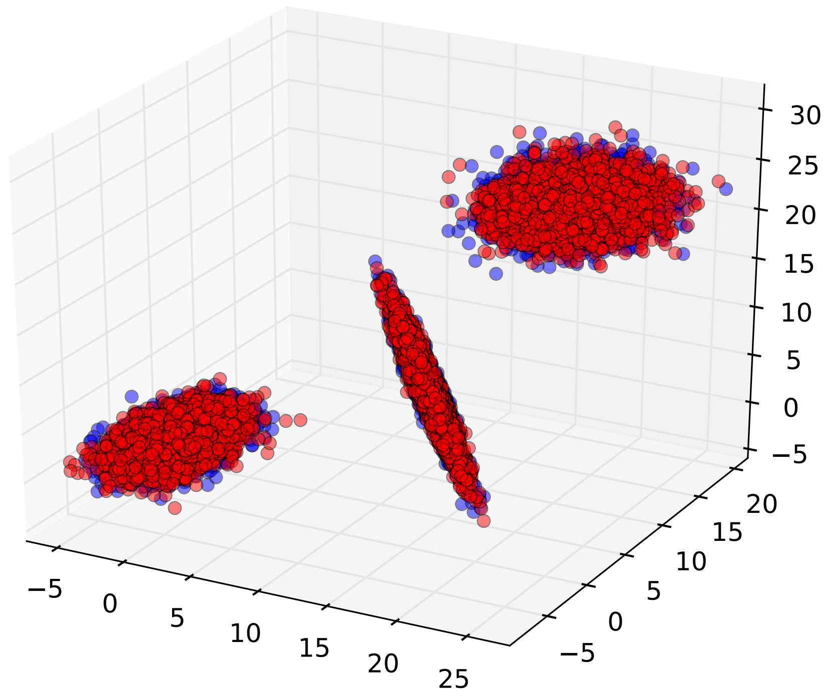

Figure 1: The blue dots in this figure are the placebo capsules, and the red are the real capsules.

We implemented k-means to cluster the data. First, we normalized our data to have mean 0 and variance 1. We initialized k-means with k-means++ and chose $K = 3$, corresponding to the three clusters identified by visualization. A plot of the results is shown in Figure 2.

As we can see, k-means classified most of the points into the three natural clusters determined by visualization, with all of the points from `getGoodCapsuleExamples` calls in the same cluster. We saved the parameters for the normalization and k-means obtained with our training data. In our pacman agent, to classify a new capsule, we first normalize the feature vector $\boldsymbol{x}$ for that capsule using our saved normalization parameters to produce a normalized $\tilde{\boldsymbol{x}}$, where

$$\tilde{\boldsymbol{x}} = \frac{\boldsymbol{x} - \boldsymbol{\mu}}{\boldsymbol{\sigma}} \tag{1}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are the mean and standard deviation of the training data, respectively. We then found which cluster center was closest to $\tilde{\boldsymbol{x}}$, and assigned this new capsule to this cluster. Since

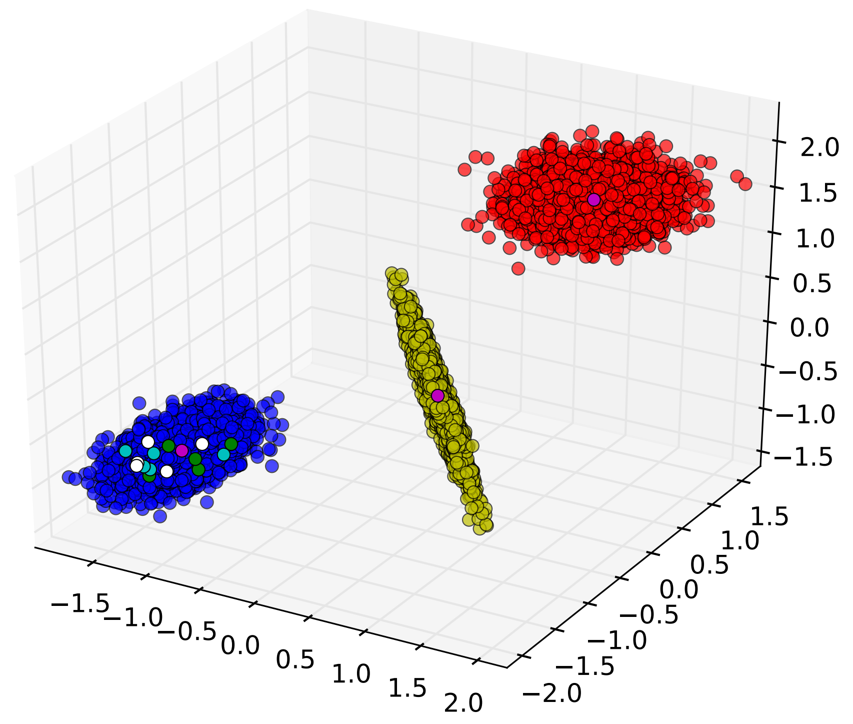

Figure 2: Red, Yellow, Blue: The three clusters of capsule data; Magenta: Cluster centers; White, Green, Cyan: examples of "good" points from `getGoodCapsuleExamples`.

each cluster is arbitrarily labeled, when the pacman agent is initialized, we cluster 10 points generated from `getGoodCapsuleExamples` and take note of their cluster centers. If the new capsule's cluster assignment is the same as the plurality of the results for the 10 points generated from `getGoodCapsuleExamples`, we say that it is a good capsule, else it is a placebo capsule.

3 Classifying Ghosts

As a first step in classification, we first plotted the value of each feature for each class and visually examined the distribution, as shown in Figure 3.

Judging from the plots, we could tell that many of the features were drawn from normal distributions, with different means and variances for each class and each feature. To estimate the mean and variance of each normal distribution for each class and feature, we used the sample means (the MLE for the true mean) and sample standard deviations

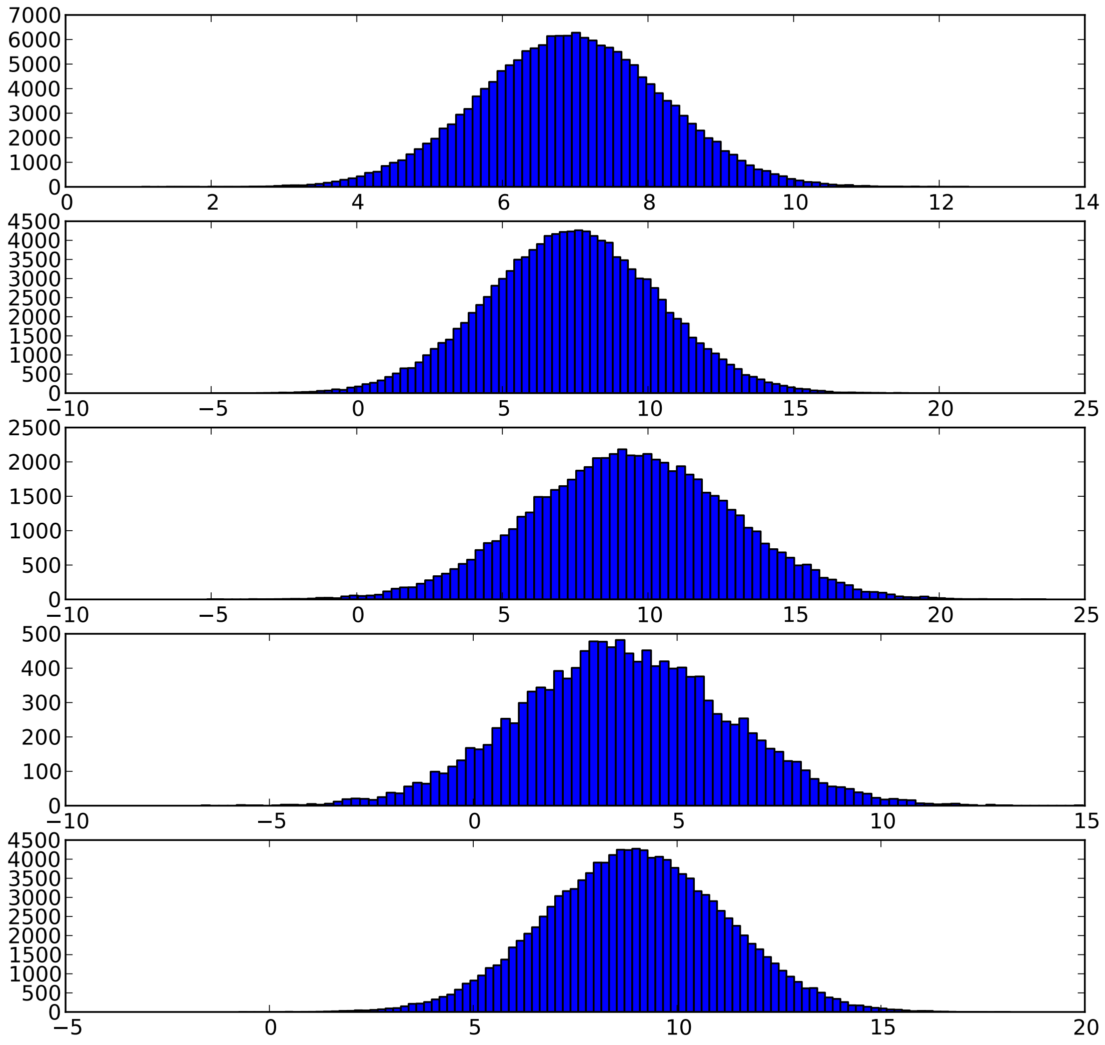
Figure 3: The distribution of first element of the feature vector for each latent class.

$$s^2 = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \bar{x})^2$$

for $\{x_n\}_{n=1}^{N}$ data points and sample mean $\bar{x}$.

3.1 Maximizing the Likelihood

Our first attempt at classification relied on using a maximum likelihood approach directly using the features which were normally distributed. Naively, we assumed that the features were independent within a class and between classes. Then, the likelihood for a feature x given parameters μ and

σ^2 from its corresponding normal distribution was the $\mathcal{N}(\mu, \sigma^2)$ density evaluated at x. By the independence assumption, to compute the likelihood a data point with feature vector $\boldsymbol{x} \in \mathbb{R}^D$ belonging to class k, we evaluated

$$\prod_{d=1}^{D} p(x_d|\mu_{k,d}, \sigma^2_{k,d}) \tag{2}$$

Then, the class that a ghost with feature vector x belongs to is

$$k^* = \text{argmax}_k \prod_{d=1}^{D} p(x_d|\mu_{k,d}, \sigma^2_{k,d}) \tag{3}$$

After using this method to classify all of our collected ghost data, we found we had a 50% overall accuracy rate, and a 79% true-positive and 13% false-positive rate for discerning ghosts from class 5 (bad ghosts). Applying Bayes rule, we found that, given we classified a ghost as class 5, there was a less than 70% chance it would actually be from class 5. We thought we could do better, so we looked at our assumptions again.

The first assumption we questioned was the independence of features within classes. To test our assumption, we calculated the Pearson correlation coefficient (PCC) between all pairwise features for each class. The PCC of two random variables X, Y is a measure of their linear dependence, and if X and Y are independent, the PCC is 0. There is also an associated hypothesis test which tests if X and Y have 0 correlation. When we calculated the PCC for pairwise features, we found that many features were correlated, and some even had PCC of 1! Therefore, our within-class independence assumption was faulty and we considered other classification methods.

3.2 SVMs

Looking at our plots of the distributions of the different features for different classes, we saw that, because they had very different means with moderate variances, we thought the data could be linearly separable. We ran an LDA, which learns a projection that maximizes variance between classes while minimizing variance within classes. Using a one-vs-all classifier for class 5 and not class 5 in Figure 4, we see that class 5 is linearly separable from the others. We also looked at one-vs-all classifiers for the other classes, and they all looked linearly separable except for class 1.

Therefore, we expected SVMs to work well. Though we had many choices of kernel for the SVM, we decided on a linear kernel because the other kernels took too long to train, even on a reduced data set of 10,000 samples. Using cross validation (withholding 40% of our training data for testing), we achieved an overall accuracy rate of 92% using SVMs, and a 100% true-positive rate for class 5 (bad ghosts). We considered other classifiers such as naive Bayes, but did not implement it because naive Bayes assumes independence of features and we discovered earlier that the features were not independent within a class. We were quite satisfied with our ghost classification with SVMs and did not try other classifiers.

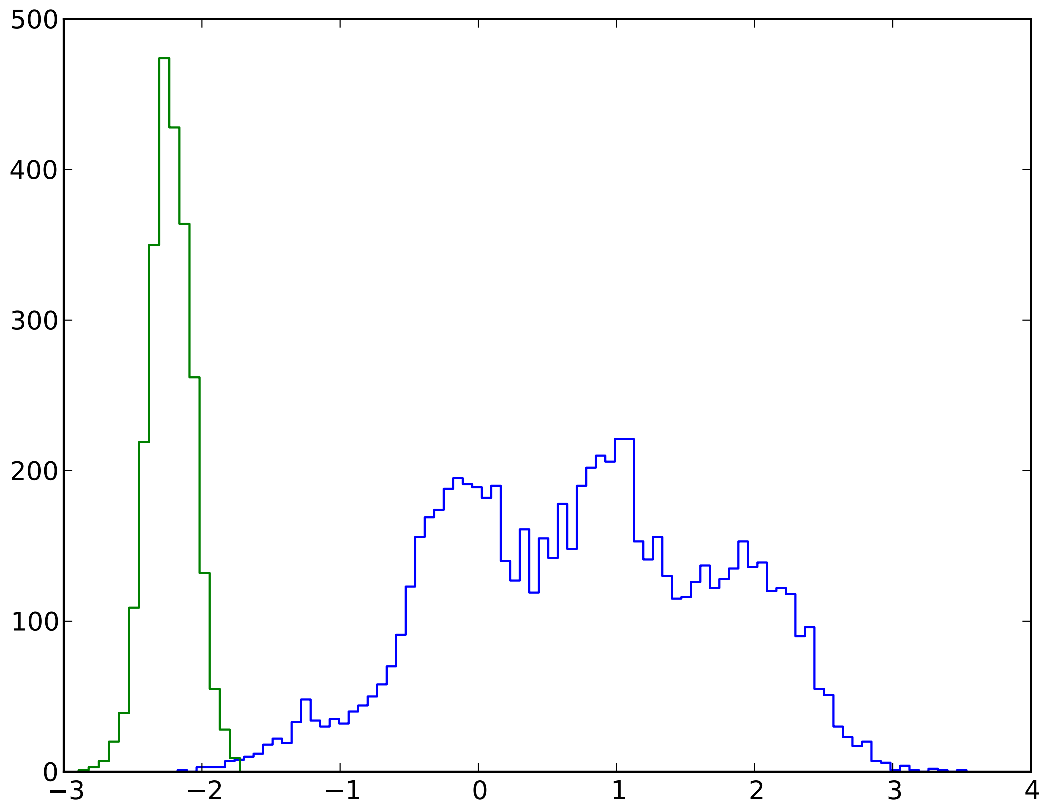

Figure 4: LDA results. Green: class 5; Blue: all other classes; X-axis: Most discriminating axis from LDA; Y-axis: density

4 Reinforcement Learning

We used Q-learning for our reinforcement learning attempt. Our first thought is to do vanilla Q-learning (like SwingyMonkey), but we quickly ran into problems with the sheer size of the state space. The sides of the map are on the order of 10-20 squares, and there are at least 7 objects we need to pay attention to (pacman, 4 ghosts, and 2 good capsules). This gives a state space of size at least $(10 \times 10)^7 = 10^{14}$, which is way too big to be usefully trained using vanilla Q-learning.

Thus we decided to try the approach of approximating the Q function with a function approximation, as suggested by the lecture notes as a way of dealing with large state spaces. This involves defining a parametric approximation $\hat{Q}_{\boldsymbol{\theta}}(s, a)$ where $\boldsymbol{\theta}$ are the parameters to learn. Following the lecture notes, we used a linear combination of feature functions as our approximation function.

$$\hat{Q}_{\boldsymbol{\theta}}(s, a) = \theta_1 f_1(s, a) + ... + \theta_J f_J(s, a) \tag{4}$$

defined on J features $f_j(s, a)$. We can define a squared error

$$\text{Error}(s,a) = \frac{1}{2}\left(\hat{Q}_{\boldsymbol{\theta}}(s,a) - \text{Target}(s,a)\right)^2 \tag{5}$$

We calculate the derivative in order to perform gradient descent.

$$\frac{\partial \text{Error}(s,a)}{\partial \theta_j} = \left(\hat{Q}_{\boldsymbol{\theta}}(s,a) - \text{Target}(s,a)\right) f_j(s,a) \tag{6}$$

Now we can use gradient descent as our Q-update rule

$$\theta_j \leftarrow \theta_j + \alpha \left(\text{Target}(s,a) - \hat{Q}_{\boldsymbol{\theta}}(s,a)\right) f_j(s,a) \tag{7}$$

where $\text{Target}(s,a) = R(s,a) + \gamma \max_{a'} \hat{Q}_{\boldsymbol{\theta}}(s',a')$, where s' is the state reached taking action a from state s.

However, this leaves us with the challenge of picking appropriate feature functions. Not only do these feature functions have to be relevant, they have to be linearly proportional to the Q-values of a certain state-action pair. The first thing we tried was the absolute position of pacman, together with the relative x, y position of every object of interest (all ghosts and capsules) to pacman. The hope was the this contained enough information for the pacman to learn an intelligible approach to tackling the problem. Unfortunately, this did not work very well. We think this is because of our linear functional approximation, which results in objects further away (higher value of relative x, y position) contributing more to the Q value approximation, which should not be the case (we'd like pacman to pay more attention to objects that are near). So the next thing we tried was simply to take the reciprocal of the relative x, y positions. However, this didn't work very well either, because an object that is 1 square away from the pacman in terms of, say, x position could be very far away from the pacman in terms of y position, and thus should be ignored. Instead, the reciprocal of the relative position becomes very large and pacman pays extraordinary attention to such objects.

The final approach which we found to work was a combination of inspiration from the bad ghost code and the heuristic that objects further away should matter less. By examining how the bad ghost chases pacman, we found that it considers the set of all possible actions and then picks the one that minimizes the Manhattan distance between the ghost and the current pacman position. Using this as inspiration, we decided to use the quantities $\delta d_{g_i}(s,a)$, which is the change in Manhattan distance between the pacman position and the position of ghost g_i in state s, as a result of taking action a. To incorporate the notion that objects further away should matter less, we divided $\delta d_{g_i}(s,a)$ by $d_{g_i}(s,a)$, which is just the Manhattan distance between pacman and ghost g_i in state s. So our final $\hat{Q}_{\boldsymbol{\theta}}(s,a)$ looks like

$$\hat{Q}_{\boldsymbol{\theta}}(s,a) = \sum_i \frac{d_{g_i}(s,a)}{d_{g_i}(s,a)} + \sum_j \frac{d_{c_j}(s,a)}{d_{c_j}(s,a)} \tag{8}$$

where the i index runs over the ghosts and the j index runs over the capsules. This was found to work well with a learning rate α that goes as the reciprocal of times past and a discount factor γ that is very close (within a factor of 10^{-4}) to 1.

5 Heuristic Approach

We were curious as to how a heuristic approach would compare to a machine-learned approach. For heuristics, we believed that trying to eat the good capsules in order to eat the bad ghost was the best way to go, since the bad ghost is worth so many more points than the good ghosts. We implemented two agents to test our hypothesis.

We first implemented an agent which greedily ate the nearest good capsule, then greedily tried to eat the bad ghost. This agent would also avoid the bad ghost if pacman had not eaten a good power capsule. This agent did not perform as well as our machine-learned approach. We noticed that this agent would fail to eat the bad ghost when it ate the good capsule because it would spend a lot of time chasing the bad ghost to no avail.

Based on this observation, we then implemented a second agent which tried to bait the bad ghost to come closer to pacman. We noticed that the bad ghost is programmed to generally move closer to pacman, so we programmed our pacman to wait 1 step away from a good power capsule until the bad ghost was 1 step away from pacman. Then, pacman would eat the power capsule and chase the bad ghost. This agent would also avoid the bad ghost if pacman had not eaten a good power capsule. Surprisingly, this agent performed better than our machine-learned agent. However, this agent was highly variable, possibly due to misclassification of the bad ghost, or because it would get trapped by the bad ghost into corners where it was difficult to escape. This agent sometimes would score over 25,000 points in 1000 steps, and other times would have very negative results.

6 Tournament Results

We had some difficulty deciding which agent to submit for the tournament. Ultimately, because we thought the tournament would be in the “best of” format, we decided on our 2nd heuristic agent, based on its higher mean score. However, if we had known the tournament would be single-elimination, we would have picked our machine-learned agent, whose results had smaller variance. Our machine-learned agent is found in the `qLearningAgent` function, and we are curious as to how well it would have performed in the tournament. Even though we did not perform too well in the tournament, we were able to put together many techniques and ideas we learned this year from CS 181 to create a sensible and effective pacman agent.