# CS 181 Section: Midterm 2 Study Guide

Ankit Gupta, Peter Ku, Vincent Nguyen, and Aidi Zhang

Week of April 24, 2016

## 1 Principal Component Analysis

PCA is a technique widely used for dimensionality reduction and feature extraction. It is essentially the orthogonal projection of data onto a lower dimensional linear space.

### 1.1 Reconstruction Method

Consider a data set of observations $\{x_n\} \in \mathbb{R}$ where $n = 1, 2, ..., N$. We now want to project the data onto space with dimensionality $M < D$ with the goal of **minimizing reconstruction error**. Given $\{u_d\}_{d=1}^{D}$ under the constraints $u_d^T u_d = 1$ and $u_d u_{d'} = 0$ ($u$ are **orthonormal** vectors), we want to reconstruct $x_n$ exactly. Express data with $u$'s as such:

$$x_n = \bar{x} + \sum_{d=1}^{D} \alpha_d^n u_d \Rightarrow \alpha_d^n = (x_n - \bar{x})^T u_d$$

Deriving the reconstruction error for particular $x_n$,

$$\begin{aligned} ||x_n - \hat{x_n}||_2^2 &= ||(\bar{x} + \sum_{d=1}^{D} \alpha_d^n u_d) - (\bar{x} + \sum_{d=1}^{K} \alpha_d^n u_d)||_2^2 \\ &= || \sum_{d=K+1}^{D} \alpha_d^n u_d||_2^2 = \sum_{d=K+1}^{D} \sum_{d'=K+1}^{D} \alpha_d^n \alpha_{d'}^n u_d u_{d'} = \sum_{d=K+1}^{D} (\alpha_d^n)^2 \end{aligned}$$

Generalizing reconstruction error to entire dataset $\{x_n\}_{n=1}^{N}$,

$$\begin{aligned} \sum_{n=1}^{N} \sum_{d=K+1}^{D} (\alpha_d^n)^2 &= \sum_{n=1}^{N} \sum_{d=K+1}^{D} ((x_n - \bar{x})^T u_d)^2 \\ &= \sum_{n=1}^{N} \sum_{d=K+1}^{D} u_d^T (x_n - \bar{x})(x_n - \bar{x})^T u_d \end{aligned}$$

$$= \sum_{d=K+1}^{D} u_d^T \left[ \sum_{n=1}^{N} (x_n - \bar{x})(x_n - \bar{x})^T \right] u_d$$
$$= \sum_{d=K+1}^{D} u_d^T S u_d$$

where $S$ is the **empirical covariance matrix** of the dataset. Taking derivative with respect to $u_d$ (think of this as one additional vector that we want to remove), with the help of a Lagrange multiplier given the orthonormal constraint, we get

$$\frac{\partial \left( u_d^T S u_d + \lambda_1 (1 - u_d^T u_d) \right)}{\partial u_d} = 0 \Rightarrow S u_d = \lambda_1 u_d$$

$u_d$ is hence an eigenvector of $S$. Reconstruction error for entire dataset becomes simply $u_d^T S u_d = \lambda_d$. Hence, if we want $K$ $\{u_d\}$ vectors that minimizes this error, the vectors $K+1, ..., D$ we leave out should have the smallest eigenvalues, in other words the $K$ that we do keep are **eigenvalues of $S$ corresponding to the largest eigenvalues.**

## 1.2 Maximal Variance Formulation

Now we consider another metric that yields similar results: **maximizing variance of projected data**.

First, consider projection onto 1 dimensional subspace the direction of which is defined by D-dimensional unit vector $u_1$ where $u_1^T u_1 = 1$. Each data point is then projected onto scalar value $u_1^T x_n$. The mean and variance of the entire dataset is then

$$\bar{x} = \frac{1}{N} \sum_{n=1}^{N} x_n$$
$$Var(x) = \frac{1}{N} \sum_{n=1}^{N} \left( u_1^T x_n - u_1^T \bar{x} \right)^2 = u_1^T S u_1$$
$$\text{where } S = \frac{1}{N} \sum_{n=1}^{N} (x_n - \bar{x})(x_n - \bar{x})^T$$

where $S$ is again the data covariance matrix. Maximizing the variance given constraint that $u_1$ is a unit vector using Lagrange multipliers, we get

$$\frac{\partial \left( u_1^T S u_1 + \lambda_1 (1 - u_1^T u_1) \right)}{\partial u_1} = 0 \Rightarrow S u_1 = \lambda_1 u_1$$

The first principal component is hence the eigenvector that corresponds to the largest eigenvalue. We define each subsequent principal component in incremental fashion by choosing each new direction to be that which maximizes the

projected variance amongst all possible directions orthogonal to those already considered. If we originally set out to project onto a $M$-dimensional subspace, the optimal linear projection for which variance of the projected data is maximized are exactly the $M$ eigenvectors $u_1, u_2, ..., u_M$ of $S$ corresponding to the $M$ largest eigenvalues $\lambda_1, \lambda_2, ..., \lambda_M$ (shown by proof of induction).

### 1.3 PCA TL;DR

Find directions that minimize reconstruction error/maximize explained variance; these turn out to be eigenvectors of data covariance matrix corresponding to the highest eigenvalues; project high-dimensional data onto these eigenvectors onto a smaller-dimensional subspace while retaining most of the information.

## 2 K-Means Clustering

In K-means clustering, you are trying to find an assignment of classes to a set of datapoints in $\mathbb{R}^n$. In particular, you are given the $\{\mathbf{x_i}\}_{1...N}$, where $\mathbf{x_i} \in \mathbb{R}^n$, and you are trying to find a set of assignments of these $N$ datapoints to $K$ classes, which are represented by their means $\{\mathbf{u_k}\}_{1...K}$, where $\mathbf{u_k} \in \mathbb{R}^n$. These are assignments are defined by $N$ vectors called responsibilities, where $r_{nk} = 1$ if the $n$th datapoint is assigned the $k$th class, or 0 otherwise. Importantly, $\mathbf{r_n}$ is one-hot.

We want to find the clustering that minimizes the sum of squared L2-norms between each point and the class they are assigned to. In other words, we want to minimize

$$J(\{\mathbf{r_n}\}_{1...N}, \{\mathbf{x_n}\}_{1...N}, \{\mathbf{u_k}\}_{1...K}) = \sum_{n=1}^{N}\sum_{k=1}^{K} r_{nk}||\mathbf{x_n} - \mu_\mathbf{k}||^2$$

To minimize this, we note that the data points are fixed, so what we can change are the responsibilities and the cluster centers. To do this, we first hold the cluster centers fixed by updating the resposibilities. Note that if we hold the centers fixed, the entire $||\mathbf{x_n} - \mu_\mathbf{k}||^2$ term in the objective function is held fixed. Thus, for each point, we will set $r_{nk}$ to be 1 such that $||\mathbf{x_n} - \mu_\mathbf{k}||^2$ is minimized. Thus, we will assign each point to the closest mean.

Then, to update the means, we will hold the responsibilities fixed, while updating the $\mu_k$. To do this update, we can simply take the gradient and solve for when it equals 0. Thus, we get

$$2\sum_n \sum_k r_{nk}(x_n - \mu_k) = 0$$

which implies that

$$\mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}}$$

which is just the mean of the points assigned to class $k$. Thus, to update the means, we just iterate through the classes, and take the mean of the point assigned to that class.

We repeatly update the assignments and the means until the convergence. Note that this will always decrease since at each step you are minimizing the objective function with respect to one set of variables while holding the other set fixed. Thus, it can never increase in objective function value. Furthermore, there are only a finite number of responsibilities, and thus this must eventually converge. However, this is not guaranteed to converge to the global optimum.

# 3 Hierarchical Agglomerative Clustering

Hierarchical clustering constructs a tree over the data, where the leaves are individual data items, while the root is a single cluster that contains all of the data. When drawing the dendrogram, for the clustering to be valid, the distances between the two groups being merged should be monotonically increasing.
The main decision is what the distance criterion should be between groups.

## 3.1 The Single-Linkage Criterion

The idea is to merge groups based on the shortest distance over all possible pairs:

$$DIST(\{x_n\}_{n=1}^N, \{y_m\}_{m=1}^M) = min_{n,m}||x_n - y_m||$$

this produces the minimum spanning tree for the data.

## 3.2 The Complete-Linkage Criterion

The idea is to merge groups based on the largest distance over all possible pairs:

$$DIST(\{x_n\}_{n=1}^N, \{y_m\}_{m=1}^M) = max_{n,m}||x_n - y_m||$$

## 3.3 The Average-Linkage Criterion

The idea is to average over all possible pairs between the groups:

$$DIST(\{x_n\}_{n=1}^N, \{y_m\}_{m=1}^M) = \frac{1}{NM}\sum_{n=1}^{N}\sum_{m=1}^{M}||x_n - y_m||$$

### 3.4 The Centroid Criterion

The idea is to look at the difference between the groups' centroids:

$$DIST(\{x_n\}_{n=1}^N, \{y_m\}_{m=1}^M) = || \left( \frac{1}{N} \sum_{n=1}^{N} x_n \right) - \left( \frac{1}{M} \sum_{m=1}^{M} y_m \right) ||$$

## 4 Mixture Models and EM

In section, we looked at a Bernoulli Mixture Model. You should be comfortable with working with mixture models for different distributions. We'll take a look at the Gaussian Mixture Model. Remember, a mixture model for all training examples roughly follows the likelihood function

$$p(X, Z|\theta, \pi) = \prod_{n=1}^{N} \prod_{k=1}^{K} \Big( \pi_k p(x_n|\theta_k) \Big)^{z_{nk}} \tag{1}$$

$$= \prod_{n=1}^{N} \prod_{k=1}^{K} \Big( \pi_k \prod_{d=1}^{D} p(x_{nd}|\theta_{kd}) \Big)^{z_{nk}} \tag{2}$$

Here, $\theta$ is a model parameter(s) for some type of distribution. Since we are working with a Gaussian, we can include $\mu$ and $\sigma$ instead of the general $\theta$:

$$p(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left( -\frac{(x-\mu)^2}{2\sigma^2} \right) \tag{3}$$

Plugging this distribution into Equation 2,

$$p(X, Z|\mu, \sigma, \pi) = \prod_{n=1}^{N} \prod_{k=1}^{K} \Big( \pi_k \prod_{d=1}^{D} \frac{1}{\sigma\sqrt{2\pi}} \exp\left( -\frac{(x-\mu)^2}{2\sigma^2} \right) \Big)^{z_{nk}} \tag{4}$$

We can take the log of the likelihood where I'll just use $L(\theta)$ as short-hand notation:

$$L(\theta) = \sum_{n=1}^{N} \sum_{k=1}^{K} z_{nk} \left( \log(\pi_k) + \sum_{d=1}^{D} \left( \log(-\sigma_{kd}\sqrt{2\pi}) - \frac{(x_{nd} - \mu_{kd})^2}{2\sigma_{kd}^2} \right) \right) \tag{5}$$

**Here, we arrive at a very crucial point in understanding mixture models and EM overall.** Recall that the $z$s are considered latent or hidden variables, that is, we do not observe them. Since we don't observe the hidden variables, then Equation 5 should really be of no use at all to use. This is where the EM algorithm comes in. In the E-Step, we take the expectation with respect to the latent variable $z$ so that

$$p(z_{nk} = 1|x_n) = \frac{p(z_{nk} = 1)p(x_n|z_{nk} = 1)}{\sum_{k'=1}^{K} p(z_{nk'} = 1)p(x_n|z_{nk'} = 1)} \tag{6}$$

$$= \frac{\pi_k p(x_n|\mu_k, \sigma_k)}{\sum_{k'=1}^{K} \pi_k p(x_n|\mu_{k'}, \sigma_{k'})} \tag{7}$$

We will say that $\gamma(z_{nk}) = p(z_{nk} = 1|x_n) = \mathbb{E}[z_{nk}]$. This gives us an expectation for $z_{nk}$. So after the E-Step, we have the following expected log-likelihood that we now want to maximize the model parameters:

$$\mathbb{E}_z[L(\theta)] = \sum_{n=1}^{N} \sum_{k=1}^{K} \gamma(z_{nk}) \left( \log(\pi_k) + \sum_{d=1}^{D} \left( \log(-\sigma_{kd}\sqrt{2\pi}) - \frac{(x_{nd} - \mu_{kd})^2}{2\sigma_{kd}^2} \right) \right) \tag{8}$$

Maximizing for $\pi_k$ is the same as written out in section notes since that term only appears once so we'd just be dealing with the first additive half of Equation 8. Let's maximize the $\sigma_{kd}$ and the $\mu_{kd}$:

$$\frac{\partial \mathbb{E}_z[L(\theta)]}{\partial \mu_{kd}} = \sum_{n=1}^{N} \frac{\gamma(z_{nk})(x_{nd} - \mu_{kd})}{\sigma_{kd}^2} = 0 \tag{9}$$

$$\Rightarrow \hat{\mu}_{kd} = \frac{\sum_{n=1}^{N} \gamma(z_{nk}) x_{nd}}{\sum_{n=1}^{N} \gamma(z_{nk})} = \frac{N_{kd}}{N_k} \tag{10}$$

where I've just used the $N$ for short-hand, e.g. $\sum_{n=1}^{N} \gamma(z_{nk}) = N_k$. Now for $\sigma_{kd}$:

$$\frac{\partial \mathbb{E}_z[L(\theta)]}{\partial \sigma_{kd}} = \sum_{n=1}^{N} \left( -\frac{\gamma(z_{nk})\sqrt{2\pi}}{\sigma_{kd}} + \frac{\gamma(z_{nk})(x_{nd} - \mu_{kd})^2}{\sigma_{kd}^3} \right) = 0 \tag{11}$$

$$= -\frac{N_k}{\sigma_{kd}} + \frac{N_{kd}^2 - 2N_{kd}\mu_{kd} + \mu_{kd}^2}{\sigma_{kd}^3} \tag{12}$$

$$= -\frac{N_k}{\sigma_{kd}} + \frac{N_{kd}^2 - 2N_{kd}^2/N_k + N_{kd}^2/N_k^2}{\sigma_{kd}^3} = 0 \tag{13}$$

$$\Rightarrow \hat{\sigma}_{kd}^2 = \frac{1}{N_k} \left( N_{kd}^2 - 2\frac{N_{kd}^2}{N_k} + \frac{N_{kd}^2}{N_k^2} \right) = \frac{\sum_{n=1}^{N} \gamma(z_{nk})(x_{nd} - \mu_{kd})^2}{\sum_{n=1}^{N} \gamma(z_{nk})} \tag{14}$$

Important takeaways: Know why we want to use EM, how EM works (E-Step and M-Step), working with Bernoulli and Gaussian mixture models, maximizing model parameters, and soft-clustering.

# 5 Hidden Markov Models and EM

In section, we looked at Hidden Markov Models (HMM) as a generalization of mixture models. Most of the in-depth derivations are featured in the corresponding section notes, so this will mostly act as an HMM summarization. In addition, some of the notation will be fleshed out and made more clear including symbols.

The HMM is used on data that can be represented as a sequence. This can include time series data, word character recognition, and part of speech tagging. For sequenced data, we have the inputs, observed states or emissions,

$$X = \{x_1, x_2, \ldots, x_N\} \tag{15}$$

The term input is a bit confusing as the HMM model considers the input as emissions or outputs from the hidden states. It is sometimes called input because it is the only observed state which is fed into the model. Each observed state has a corresponding hidden or latent state,

$$Z = \{z_1, z_2, \ldots, z_N\} \tag{16}$$

We use $N$ to denote the length of the sequence. We can represent the HMM as a directed graphical model in Figure 1 (the HMM can also be undirected, but we are just dealing with directed ones)

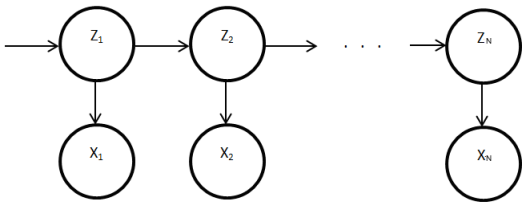


Figure 1: Observed and Hidden States of an HMM

We consider this a directed graph because there is a direction associated with the transition of latent states in the sequence. As such we can represent the transition probabilities of the latent states with a **transition matrix**:

$$T(z'|z) = \begin{pmatrix} p(z_1|z_1) & p(z_1|z_2) & \ldots & p(z_1|z_K) \\ p(z_2|z_1) & p(z_2|z_2) & \ldots & p(z_2|z_K) \\ \vdots & \vdots & \ddots & \vdots \\ p(z_K|z_1) & p(z_K|z_2) & \ldots & p(z_K|z_K) \end{pmatrix} \tag{17}$$

where each entry denotes the probability of transitioning to some state at time $n+1$ given the state you are in at time $n$. Note that this matrix $T \in [0,1]^{|K|\times|K|}$ is square, each row must sum up to 1 and each entry, $p(z'|z)$, is between 0 and 1 where $z'$ is the next state.

Now that we've represented the states, what about the inputs. Each individual $x$ consists of feature(s). They are usually denoted as vectors where each entry represents some weight for a feature. **Alternatively, each emission could just be a single output.** So, we also have latent state-conditional probabilities for the inputs represented by the **emission matrix**:

$$\Omega(x_f|z) = \begin{pmatrix} p(x_1|z_1) & p(x_2|z_1) & \ldots & p(x_F|z_1) \\ \vdots & \vdots & \ddots & \vdots \\ p(x_1|z_K) & p(x_2|z_K) & \ldots & p(x_F|z_K) \end{pmatrix} \tag{18}$$

Important: if there are $|F|$ total features, we are just saying $x_F$ is feature $F$ for a certain $x$ not an $x$ at time step $F$! Note that this matrix $T \in [0,1]^{|Z| \times |F|}$ and **is not** square, each entry, $p(x_f|z)$, is between 0 and 1 where $f$ is some input feature, and the rows do sum to one.

A brief aside that might help you understand the HMM better: think about a Naive Bayes Classifier. This type of classifier is known as a generative classifier. What this means is that instead of modeling the condition probability $p(z|x)$ (think logistic regression), a Naive Bayes operates under the hypothesis that modeling the joint probability $p(x, z)$ describes the full generational process of the data, e.g. sampling from $p(z)$ and $p(x|z)$ generates the full data $p(x, z)$. So an HMM is basically just a bunch of individual Naive Bayes Classifier that have latent state transition probabilities! So know you can see why we call it an emission matrix – our hypothesis is that the latent states generate or emit the features. Other more detailed stuff is in the section notes on HMM and EM.

# 6 Reinforcement Learning

## Markov Decision Process

In a Markov Decision Process (MDP), we have:

- A set of states $\mathcal{S}$ that the agent may observe.
- A set of actions $\mathcal{A}$ that the agent gets to choose.
- A reward function $R(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, which is the reward the agent gets when taking action $a$ under state $s$.
- A transition model $T(s'|s, a) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, which is the probability of a state transition to $s'$ when the agent takes action $a$ under state $s$. This is what the environment does to the agent.

The goal of the agent is therefore to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$, which decides what action to take under state $s$, to interact with its environment such that it maximizes future rewards.

## Finite Horizon

For a world with a finite time horizon $T$, our objective is to maximize

$$\sum_{t=1}^{T} R(s_t, a_t)$$

where $s_t$ is the state that the agent is at and $a_t$ is the action that the agent takes at time $t$. Replacing $a_t$ with our policy $\pi$, we have

$$\sum_{t=1}^{T} R(s_t, \pi(s_t))$$

What would we do in the last step of the game? We simply choose the action that maximizes the reward in the last step before the world ends. Let us index our policy $\pi$ with the number of steps before the game ends. Then

$$\pi_1(s) = \arg\max_{a \in \mathcal{A}} R(s, a)$$

From this, we can define value, which is the reward that we can get (by following an optimal policy) in state $s$: $V_k : \mathcal{S} \to \mathbb{R}$. Again, this is indexed by the number of steps away from the end of the game

$$V_1(s) = \max_{a \in \mathcal{A}} R(s, a)$$

If we are instead two steps away from the game ending, then the policy is trying to maximize is the reward on the second last step, plus the reward on the last step (assuming that we follow the optimal policy on that step)

$$\begin{aligned}
\pi_2(s) &= \arg\max_{a \in \mathcal{A}} \left\{ R(s, a) + \max_{a'} E_{s'} \left[ R(s', a') \right] \right\} \\
&= \arg\max_{a \in \mathcal{A}} \left\{ R(s, a) + \sum_{s'} T(s'|s, a) \max_{a'} R(s', a') \right\} \\
&= \arg\max_{a \in \mathcal{A}} \left\{ R(s, a) + \sum_{s'} T(s'|s, a) V_1(s') \right\}
\end{aligned}$$

We can then use this to define the value two steps away from the game ending

$$V_2(s) = \max_{a \in \mathcal{A}} \left\{ R(s, a) + \sum_{s'} T(s'|s, a) V_1(s') \right\}$$

This generalizes to

$$V_k(s) = \max_{a \in \mathcal{A}} \left\{ R(s, a) + \sum_{s'} T(s'|s, a) V_{k-1}(s') \right\}$$

It is then natural to define a "value" function not only for states, but for state-action pairs. This is known as the "Q" function: $Q_k : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, again indexed by the number of steps away from the end of the game. This function is defined as

$$Q_k(s, a) = R(s, a) + \sum_{s'} T(s'|s, a) V_{k-1}(s')$$

and thus the optimal policy $\pi^*$ is given by

$$\pi_k^*(s) = \arg\max_{a \in \mathcal{A}} Q_k(s, a)$$

With our "Q" function, we can rewrite our value function as

$$V_k(s) = \max_{a \in \mathcal{A}} Q_k(s, a) = Q_k(s, \pi_k^*(s))$$

with initialization $V_0(s) = 0$

## Infinite Horizon

We saw in the previous section how to deal with finite horizon games, but what if the agent lives forever? Our recursion in the previous case did not converge (there is no reason that it would), so we can't simply extend that method to $T \to \infty$. The reason is that the longer we lived the more rewards we would get (simply because we can take more actions). To deal with the infinite horizon case, we have to discount future rewards to make it converge (this may be because future events are not guaranteed to happen, or we are impatient and want rewards now rather than later, etc). We introduce a discount factor $\gamma \in (0, 1)$, and now we want to maximize

$$\sum_{t=1}^{\infty} \gamma^t R(s_t, a_t)$$

Our new "Q", value, and policies are

$$\begin{aligned} Q(s, a) &= R(s, a) + \gamma \sum_{s'} T(s'|s, a) V(s') \\ \pi^*(s) &= \arg\max_{a \in \mathcal{A}} Q(s, a) \\ V(s) &= \max_{a \in \mathcal{A}} Q(s, a) = Q(s, \pi^*(s)) \end{aligned}$$

We can estimate this "Q" function by iteratively updating it to the rewards we get in each step and the expected value going forward. This is known as **value iteration**. It turns out that this "Q" function is guaranteed to converge, but the time it takes to converge will depend on the mixing time of the Markov chain. However, this is out of scope, and we will not be discussing it.

## Policy Iteration

Instead of watching our "Q" function converge, we could instead watch our policy (which is something discrete that tells us which action $a$ to take under

state $s$). Let us define a matrix $\mathbf{T}$ whereby the element at row $s$ and column $s'$ is

$$T_{s,s'} = T(s'|s, \pi(s))$$

In other words, this is the probabily of getting to state $s'$ forom state $s$, given that we are following policy $\pi$. Now, we define vectors

- $\mathbf{V} \in \mathbb{R}^{|\mathcal{S}|}$, where $V_s = V(s)$
- $\mathbf{r} \in \mathbb{R}^{|\mathcal{S}|}$, where $r_s = R(s, \pi(s))$

Now, we can write down our problem as

$$\mathbf{V} = \mathbf{r} + \gamma \mathbf{T}\mathbf{V}$$

This is simply a vectorized form of our previous equations, namely

$$V(s) = Q(s, \pi(s)) = R(s, \pi(s)) + \gamma \sum_{s'} T(s'|s, \pi(s))V(s')$$

This allows us to solve for $\mathbf{V}$ (since this is just a linear equation) and from there, we can optimize our policy $\pi$ until it converges. In reality, the way that this would work is at every epoch, we solve the above linear equation, update the Q function values for every state-action pair, and then update the policy, noting that $\pi^*(s) = \max_a Q(s, a)$.

# Reinforcement Learning

Value iteration and policy iteration allow you to plan for the optimal policy in a known environment, in which both the reward function and the transition probabilities are known. However, in reinforcement learning, we have unknown rewards and unknown transitions, while the states and actions are still known. There are two different approaches: model-based and model-free reinforcement learning. Model-based says that we're going to learn what the model is, and then use value iteration or policy iteration to solve that Markov Decision Process. Essentially, we loop in which we wander around and gather data, using it to estimate our model, and then plan with V.I. or P.I. What are the kinds of statistics we may want to gather as we wander? Our transition model and reward function.

We gather data by tracking counts and modeling averages. We can define $N_{s,a}$ as the number of times we took action $a$ from state $s$. We can also define $N_{s,a,s'}$ as the number of times we landed in $s'$ after taking action $a$ in state $s$. We can then estimate $T(s'|s, a)$ as the ratio $\frac{N_{s,a,s'}}{N_{s,a}}$. The last thing we need to do is track rewards $R_{s,a}^{total}$ as the total reward amount from all of the times we have taken action $a$ in state $s$, and then estimate $R(s, a)$ as $\frac{R_{s,a}^{total}}{N_{s,a}}$.

This model isn't too ideal. We can't generalize to all states, and we may have to

wander a long time to see all our possible states and outcomes. This also uses up a lot of memory in the case we have a lot of states. Planning may also take too long, which leads to our idea of model-free RL. This says that all we really need is the Q Function - we don't need to do planning if we can estimate what the Q Function is, since the Q Function determines everything we need to do.

# Q-Learning

The key idea behind Q-Learning is that the sum in the Bellman equation is an expectation

$$\begin{aligned} Q(s,a) &= R(s,a) + \gamma \sum_{s'} T(s'|s,a) \max_{a'} Q(s',a') \\ &= R(s,a) + \gamma E_{s'} \left[ \max_{a'} Q(s',a') \right] \\ &= E_{s'} \left[ R(s,a) + \gamma \max_{a'} Q(s',a') \right] \end{aligned}$$

Hence, $Q(s,a)$ is an expectation, and moreover, the expectation of something we can compute. For some $s$ and some $a$, we can take that action and go to $s'$, so we can end up with a future estimate of what that reward is based on where we go. There's an error between what we thought would happen when we took an action and a state, and what ended up happening. Using this error, we can perform a stochastic-gradient-descent-like update to learn the optimal solution.
Our algorithm is the following, where $\alpha$ is our learning rate, and $r$ is the reward we actually got when taking action $a$ in state $s$:

$$Q_{s,a}^{new} \leftarrow Q_{s,a}^{old} + \alpha \left[ \left( r + \gamma \max_{a'} Q_{s',a'}^{old} \right) - Q_{s,a}^{old} \right]$$

The bracket following the learning rate is our expected quantity from the equations above subtracted by $Q_{s,a}^{old}$, which represents the error in our estimations. That is, the parenthetical term is like an estimate arising from the world, and we have stochastic gradient descent with this noisy estimate.