Machine Learning with Python: From Linear Models to Deep Learning (6.86x) review notes.

David G. Khachatrian

October 18, 2019

1 Preamble

This was made a while after having taken the course. It will likely not be exhaustive. It may also include some editorializing: bits of what I believe are relevant observations and/or information I have come across.

2 Motivation

2.1 Prediction vs. Estimation

What goal does machine learning have, and how does that differ from the goal of econometrics? Machine learning's goal (in supervised learning) is *prediction* ($\hat{y}$) of as-yet-unobserved outputs, while econometrics's goal is *estimation* ($\hat{\theta}$) of the underlying mechanisms of a process (which then produce outputs y). In prediction, we are not concerned with accurately representing the "black box" that is the world; we only care whether we can the outputs of the black box. Meanwhile, estimation attempts to describe the components of the black box (using knowledge of the process and the black box's outputs as a guide).

2.2 Setup

We have a space/class of functions $\mathcal{H}$, individual functions/models $h \in \mathcal{H}$, some inputs X_i, some objective function J. The goal is to find an h that minimizes the objective function, given the data $J(h, X_i)$. In most cases, there is no closed-form solution for h. Instead, we choose a suitably flexible/complex form for h_θ, then iteratively update the parameters θ that comprise h_θ to move toward a minimizer of J with function/model h_{θ^*}.

In *supervised learning*, the individual "unit of input" to h are tuples (X_i, Y_i). In *unsupervised learning*, the unit of input is simply a singleton X_i.

Usually (particularly when considering supervised models), J contains a term that penalizes the model from not predicting the target values for each X_i correctly, and a term to punish choosing a model that's "overly complex":

$$\begin{aligned} J(\theta, X_i) &= L(\theta, X_i) + \lambda R(\theta) \\ &= (\text{loss; penality for inaccurate predictions given the data}) + \lambda \times (\text{regularization; penalty for a complex model}) \end{aligned}$$

This is meant to strike a balance in the *bias/variance tradeoff*. Complex models can fit a given sample X_i perfectly, but will change a huge amount if it were given a different sample X_j. The complex model shows high variance. Meanwhile, an overly simple model will have predictions that are not useful (either far from the target variable in supervised learning, or unactionable results in unsupervised learning). Meaning,

overly simple models have high bias. The regularization parameter λ controls the balances between loss and regularization. The regularization term can be thought of as representing a prior belief (in the Bayesian sense) on a useful form for h (see the "Fundamentals of Statistics" notes for more).

For much of this course, we focus on *supervised learning*. There are then two main subtypes: *classification* and *regression*. In classification, we map inputs to a *discrete* output space of finite cardinality (isomorphic to a subset of the whole numbers): $h : X \to Y; Y \subseteq \mathbb{W}, |Y| < \infty$. In *regression*, we map inputs to a *continuous* output space (isomorphic to the real numbers): $h : X \to Y; Y \subseteq \mathbb{R}$. (If Y is n-dimensional, the output spaces are subsets of $\mathbb{W}^n$ and $\mathbb{R}^n$.)

3 (Linear) classifiers.

A way of describing *linear classifiers* is where we choose a *decision boundary*,

$$\theta \cdot x + \theta_0 = 0$$

Then we can form a binary classifier by taking:

$$h(x, \theta, \theta_0) = sign(\theta \cdot x + \theta_0)$$

(Note that $\theta \cdot x + \theta_0$ is the same as $\theta_{aug} \cdot x_{aug}$ where $\theta_{aug} = \begin{bmatrix}\theta_0 & - & \theta & -\end{bmatrix}$ and $x_{aug} = \begin{bmatrix}1 & - & x & -\end{bmatrix}$.)

3.1 Perceptron algorithm

The *perceptron* algorithm is a way of finding a linear classifier *if the data are linearly separable* – if they aren't separable, the algorithm never converges! It works according to the following logic: (1) See whether my current prediction for X matches its actual label Y. (2) If they don't match, then add (a signed version of) X to my parameter θ. (3) Repeat until convergence/no more mistakes are made throughout the entire dataset.

In pseudocode:
TODO

3.2 Maximum margin classifiers.

Geometrically, the decision boundary is a hyperplane that cuts the domain of X into two halves: the parts that will be classified as $+1$ and -1. θ itself is a vector normal to the hyperplane: the "quickest" way to go from one half of the plane to the other would be to change X along the direction of θ. The magnitude of θ, $\|\theta\|$, describes the "sensitivity" of the classifier to changes in x: the smaller $\|\theta\|$ is, the less sensitive the classifier is to small changes in x. The "actual" distance between a point and the decision boundary is directly proportional to its "score" and inversely proportional to $\|\theta\|$:

$$d = \frac{|\theta \cdot x + \theta_0|}{\|\theta\|}$$

One can consider a classifier that attempts to maximize d for the provided data (with the data being put on the right side of the decision boundary). This is the same as finding a θ with minimal norm $\|\theta\|$.

In cases where a point could be misclassified, we can consider a *hinge loss*:

$$Loss_{hinge}(y_i(\theta \cdot x_i + \theta_0)) = Loss_{hinge}(z) = \min(0, 1 - z)$$

In this case, we "push" our θ so that $y_i(\theta \cdot x_i + \theta) \geq 1$. The hyperplanes corresponding to $\theta \cdot x_i + \theta = \pm 1$ are the *margin boundaries*; we try to separate the margin boundaries as much as possible.

4 Optimization method: (Stochastic) Gradient Descent.

How do we actually update the parameters of our model when it's necessary? We use the *gradient*. The gradient captures the "direction of steepest ascent" of a function. So, more specifically, $\nabla_\theta J(\theta_{cur})$ will say "at the input θ_{cur}, which way should I go to maximally increase J?" We want to *minimize* J, so we step in the *opposite* direction that the gradient shows:

$$\theta \leftarrow \theta - \eta \frac{\partial J}{\partial \theta'}(\theta)$$

$$\theta \leftarrow \theta - \eta \nabla_{\theta'} J(\theta)$$

where η is the *step size* (so we don't jump too much – linear approximations of functions only work well very close to where the approximation is centered).

To be clear, each coordinate is updated "independently" of the others, in the sense that the update for θ_j depends on the partial derivative $\frac{\partial J}{\partial \theta_j}$.

In regular gradient descent, you run through your entire training set, average the computed gradient per sample, and update the parameters with this averaged gradient. This is guaranteed to monotonically improve the objective function, but it is *slow* (n calculations for 1 update).

Often you instead perform *stochastic gradient descent*, or more specifically *minibatch gradient descent*, where you update after only averaging the gradient across $k \ll n$ samples (a "minibatch"). This doesn't always necessarily improve the objective function; we're choosing a subset of our full dataset, so there's randomness built in ($k \to n$ decreases stochasticity but increases time cost; $k \to 1$ increases stochasticity but decreases time cost). But we gain a great deal of speed, and empirically, the minibatch method of optimization tends toward a minimizing point using fewer runs through a dataset than the "non-stochastic" method. (When $k = 1$, our "minibatch" isn't really a batch, so we just call it "stochastic gradient descent".)

For theoretical guarantees, we usually want the following properties for η (where, in the following, i represents the i'th parameter update):

1. $\sum_{i=1}^{\infty} \eta_i = \infty$. There is always enough "push" behind huge accumulations of updates that you could technically reach any point in function space.
2. $\sum_{i=1}^{\infty} \eta_i^2 < \infty$. The variance of our parameters will decrease over time. (So while it could technically reach anywhere, practically it will be strongly localized after a large enough i.)
3. (Square-summability implies that $\lim_{i \to \infty} \eta_i = 0$. This reinforces that idea that eventually, the model is localized near a single point.)

$\eta_t = \frac{1}{1+t}$ satisfies these requirements.

5 Nonlinear classifiers and the kernel trick.

Our earlier linear classifier looked like the following:

$$h(x) = sign(\theta \cdot x + \theta_0)$$

Some relevant observations:

1. Our decision boundary is currently linear. What if the "correct" decision boundary is nonlinear?
2. The dot-product $\theta \cdot x$ is a type of *inner product* ($\langle \theta, x \rangle$), which can generally be interpreted as a *measure of similarity* between the two arguments.

5.1 Feature representations.

What if we want nonlinear decision boundaries? We can "augment" our data samples into richer *feature vectors*: $x \rightarrow \phi(x)$. For example, we can transform a pair of values (x_1, x_2) to the following: $(x_1, x_2) \mapsto [1, x_1, x_2, x_1x_2, x_1^2, x_2^2, tanh(x_1 + x_2)]$. We can now train a linear classifier on this augmented feature vector, and perhaps we find that the decision boundary is very simple in this space, just $x_1x_2 = c$. This is a simple hyperplane in our augmented space (which is *linear*); however, in the original (x_1, x_2) space, the decision boundary is *nonlinear*.

This seems like magic, but remember: we've actually still just trained a linear classifier, just in a larger input space. It just "looks" nonlinear when we project the boundary back down to a smaller subspace. (In neural networks, we basically stack a bunch of these classifiers/filters on top of each other.)

5.2 The kernel trick.

Explicitly computing these feature representations $\phi(x)$ can be costly! (Try calculating explicitly the infinite sequence: $x \mapsto [1, x, x^2, x^3, \cdots]$.) Conveniently, in many algorithms, they don't show up on their own; they only show up as part of a dot product (e.g., $\theta \cdot x$). They only show up in the context of *measuring its similarity with some other object*. So we don't need $\phi(x)$ explicitly, we just need a function

$$K_\phi : (X_1, X_2) \rightarrow \mathbb{R}; \quad K_\phi(x_1, x_2) = \langle \phi(x_1), \phi(x_2) \rangle$$

where K_ϕ is symmetric $K(x_1, x_2) = K(x_2, x_1)$ and $K(x, x) \geq 0$. Such a function, which maps the similarities between two points according to a specific representation of the two points, is called a *kernel function*, and sneaking in K_ϕ to avoid costly computations is called the *kernel trick*.

Every kernel function is implicitly a dot product between a specific representation of its inputs. For example, for $K(x_1, x_2) = x_1^2x_2^2 + cos(x_1)cos(x_2) = \phi(x_1) \cdot \phi(x_2)$, where $\phi(x) = [x^2, cos(x)]$. Because of their structure, you can add, multiply, divide, etc different kernel functions and still obtain valid kernel functions! (Kernel functions are very closely related to covariance matrices; look into the Mercer Theorem for more. They are also essentially the idea behind Multidimensional Scaling (MDS).)

5.2.1 Kernel perceptron algorithm.

Kernel functions can immediately find use in the perceptron algorithm. The parameter θ is always literally just a linear combination of the data points (add 1 to a datapoint's weight whenever it's misclassified), so you can rewrite the classification comparison as an inner product between two datapoints – which you can then rewrite using a kernel function of your choosing!

Pseudocode:
TODO

Though it isn't as obvious, the maximum margin classifier turns out to be determined by only a specific subset of points (those which touch the margin boundaries), and so can similarly be rewritten in terms of kernel functions.

Pseudocode:
TODO

6 Low-rank matrix factorization

Given a matrix $(n \times p)$with missing entries X, you want to impute the missing values in a "principled" manner. An idea is to assume that there are actual k latent factors that determine the values of X.

Let's use the idea of n users having rated some of p movies. We assume that $X = UV^T$ where each row of $U(n \times k)$ represents each user's "affinity" toward some intrinsic factors in movies, and each column of $V^T(k \times p)$ represents the amount each factor is present per movie. One could think of the latent factors as

something like "level of humor", "does it contain famous actors?" (though even these categorical factors will be "soft") – in reality, it will likely be the case that many human-interpretable concepts will be mixed into one "factor".

So our assumption is that

$$X = UV^T$$

and the corresponding objective function is

$$J = \sum_{(a,i)\in D} \frac{(Y_{ai} - (UV^T)_{ai})^2}{2} + \frac{\lambda}{2}\left(\sum_{a,k} U_{a,k}^2 + \sum_{a,k} V_{a,k}^2\right)$$

where D is the set of coordinates in the matrix X that we actually know, and Y_{ai} is that value.

How do we actually optimize this function? By performing *alternating coordinate descent*. First optimize with respect to the parameters of U while holding the parameters of V fixed, i.e., $U \leftarrow U - \eta \nabla_U J(U; V)$. Then hold U fixed and optimize with respect to V. Repeat ~~ad nauseum~~ until convergence.

7 Neural networks.

Neural networks are, in a way, stacks of nonlinear "classifier" units built on top of each other.

Each "neuron"/unit of a neural network takes as input a linear combination of the outputs of the layer behind it, performs some nonlinearity on this linear combination, and emits the result as an output. In a way, each layer forms a new feature mapping (of possibly different dimension) based on the previous layer, $\vec{x} \to \phi(\vec{x})$.

The "tricky" part when you want to train an entire neural network architecture, you're changing the weights of both the final layer that actually emits the final output (e.g., the label if the network is a classifier) *and* the weights for the "hidden" layers (everything between the input and the output). So you're optimizing the final classifier and the representation ϕ that classifier is acting on at the same time. How do you manage to do both?

An important fact is that we assume that the nonlinearities applied at each unit do not ever change.[1] So if we describe a single unit of a layer as $h_i(z_i), z_i = W_{i-1,i}h_{i-1}(z_{i-1}) + b$, we assume the form of h_i is fixed, so only $W_{i-1,i}$ and b need updating. (If all the units in a layer perform the same nonlinearity, which many models adhere to, then W is a matrix, while h, z, b are vectors.)

To emphasize, the "flow" of an input according to our conventions is $h_{i-1} \to_{linearcombination:(W,b)} z_i \to_{nonlinearity:"f"} h_i$.

7.1 Update method: Backpropagation (and SGD).

When it comes to updating individual values, we technically do the same thing as usual: use the partial derivative and provide an update via, e.g., stochastic gradient descent.

$$w_j \leftarrow w_j - \eta \frac{\partial J}{\partial w_j}(\theta)$$

The real question is "How do you get the form of $\frac{\partial J}{\partial w_j}$?" The answer: a whole lot of chain rule. You "propagate" the error from the final output "back" to the parameter of interest – something like:

[1] Differentiable architecture search now exists, but is quite computationally intensive, hard to train, and probably overkill for whatever you're considering.

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_k} \times \frac{\partial z_k}{\partial h_{k-1}} \times \frac{\partial h_{k-1}}{\partial z_{k-1}} \cdots \times \frac{\partial h_j}{\partial w_j}$$

This looks intimidating, but it makes a lot of sense if you look at the network pictorially. You take a path from the output node to the target edge/node, and compute a partial derivative for every traversal you make. More than one path that takes you there? Add together each path's individual partial-derivative-product-train. This is called *backpropagation*.

7.2 Model complexity/choice/convergence considerations.

It might seem like we can go overboard in having a super complex architecture – super deep (many layers), super wide (many neurons per layer). This is true in the sense that a neural network could end up just memorizing the data ("variance" in the bias-variance tradeoff). With proper regularization/a good objective function, there is actually a sense in which it's "easier" to train a "good" model when it has enough flexibility, in the sense that technically you would *eventually* get a model that reaches a minimizer. Consider the objective function and imagine it visually – it's sort of a bumpy mountain range, and we're looking for the deepest value. When you only have one axis you can move along ("left or right"; this would be a 1-D parameter), then you can easily get "stuck" in a localized dip that isn't the deepest valley. But if you had a bunch of different ways to move "away" from a particular point, there is a higher chance that you can escape the valley by following *some* axis, which means you can eventually maneuver your way to the true deepest valley.

This analogy is not perfect, and it takes a well-formed objective function for an overly complex neural network to train as we have described. Also, this does not consider the *computational* costs of training to a minimizer – you'll need more memory to hold the model, you'll have to update many more weights per parameter, etc. There are many practical considerations to consider besides eventual ability to reach the objective function's minimum; if you can get close enough in $1/100$ the training time with a model that's $1/1,000,000$ the size, you'd probably go with that over the monolith.

7.3 Recurrent Neural Networks.

In recurrent neural networks (RNNs), we feed in a sequence of inputs, passing in the next entry in the input sequence *and* an output/structure of the RNN (usually a *hidden state*) into the RNN. So we partially feed the output of RNN back into itself for the next prediction. It is in this way that the RNN is "recurrent". Backpropagation updates the internals of the RNN to maximize the separate components: hidden state, output, etc. (Different architectures have different components. One of the better "classic" models is the *long short-term memory (LSTM)* cell.)

7.4 Convolutional neural networks.

In convolutional neural networks (CNNs), we feed in information where there is some semblance of "proximity" between parts of the input, and we expect that only "nearby" portions of a particular point in the input are relevant for that point. We then have "filters" which slide along the input/layer, and each point gets as its output a dot product with the filter (what is called a convolution, but is actually an *cross-correlation* of the layer with the filter). Nonlinearities are applied in between "convolutions". Some layers include *pooling*, where adjacent patches are summarized in some manner (usually max-pooling). Using backpropgation, the weights on the filters are trained. (The number of filters is a pre-fixed architecture choice.)

The classic use-case for CNNs is just about anything to do with images.

8 Generative vs Discriminative Models

One can consider developing two different types of models:

1. "Given an example x, what is its likely target variable y?" This is something like $f(y \mid x)$ and is called a *discriminative model* – it "discriminates" whether an individual point is associated with any of the possible target values y.

2. "Given a target variable y, what is the distribution of possible inputs x?" This is something like $f(x \mid y)$ and is called a *generative model* because you can generate "examples" of inputs that (according to the model) are associated with y.

Depending on whether you're more interested in "Which among the possible target values is my actual input X most like?" (discriminative) or "What are other examples that seem to belong a target value Y?" (generative), one type of model may be more useful than the other.

9 Unsupervised learning and Clustering

In unsupervised learning, we only have set of datapoints X_i and no "targets". Often, the goal is to find some latent structure in the distribution of the dataset X_i. We have to decide what sort of structure to impose – for clustering algorithms, how many clusters K should we have? It's ideal if there is reason to know what a reasonable value of K would be; one can also programatically decide K by imposing a regularization term to the objective function of the algorithm for more complex models (e.g., the Bayesian Information Criterion).

9.1 K-means, K-medoids

In K-medoids and K-means, we define a distance measure between points $dist(x_1, x_2)$. The algorithm involves the following two steps (after randomly initially $z_1, \cdots, z_k$): (1) Assign each x_i to the closest cluster C_j represented by z_j. (2) Reassign a "best representative" such that $\sum_{x \in C_j} dist(x, z_j)$ is minimal.

The idiosyncrasies to each (according to this course):

- In K-means, $dist(x, y) = \|x - y\|_2^2$, and z_j needn't be an element of the dataset, so the optimal choice of z_j would be $\frac{\sum_{x \in C_j} x}{|C_j|}$.
- In K-medoids, $dist(x, y)$ is free to vary, but z_j must be an element of the dataset.

9.2 Gaussian Mixture Models and the Expectation-Maximization Algorithm

A Gaussian mixture model assumes that a datapoint X is generated in the following two-step process:

1. From a Mulinoulli distribution with parameters $\pi_1, \cdots, \pi_K$, draw the output k.
2. Now based on the output k, draw from $N(\mu_k, \sigma_k^2)$. This is X.

It turns out that optimizing this function by maximizing log-likelihood is not as straightforward as one would hope: you have a $\sum_i (\log \left(\sum_j \cdots \right))$, which cannot be spooled out into a closed-form solution.

What we do is a bit of well-justified fudging-about: We change our objective function to $\sum_i \sum_j (\log \cdots)$. It turns out that this provides an upper-bound for our actual objective function while also having a closed-form solution. What we can then do is update our parameters to the optimum of our approximation, then approximate again, etc until convergence. This will allow us to derive the update steps below.

How do we actually get to our optimum? We first fix the probabilities that each datapoint belongs to a particular cluster (E-step), then we update our π_j, μ_j, σ_j^2 (M-step), and alternate. More specifically, for the E-step:

$$p(j \mid i) = \frac{\pi_j N(x_i; \mu_j, \sigma_j^2)}{\sum_j \pi_j N(x_i; \mu_j, \sigma_j^2)}$$

Then for the M-step (assuming d-dimensional input x and that $\Sigma = \sigma^2 I_d$):

$$n_j \leftarrow \sum_i p(j \mid i), \quad \pi_j \leftarrow \frac{n_j}{n}, \quad \mu_j \leftarrow \frac{\sum_i x_i p(j \mid i)}{\sum_i p(j \mid i)}, \quad \sigma_j^2 \leftarrow \frac{\sum_i p(j \mid i) \|x_i - \mu_j\|_2^2}{d \sum_i p(j \mid i)}$$

Note that Gaussian mixture models are a sort of "soft" version of K-means/K-medoids clustering.

10 Reinforcement Learning

In reinforcement learning, we reformulate our problem. Now, our algorithm will act as an *agent* acting in a world with states s (where all relevant information is captured in s), potential actions a, and transition state probabilities $T(s, a, s') = Pr[\text{end up in state } s' \mid s, a)$. The agent gets "rewards" for performing certain actions in certain states $R(s, a, s')$, and the objective is to maximize the cumulative reward $\sum_t R_t$. To incentive the agent to prefer acting sooner rather than later, we can impose one (or both) of the following adjustments to R:

- Finite Time-Horizon. After T steps, "the world ends"/the agent is no longer able to perform actions.
- Temporal discounting of rewards. A discount factor of γ^k is applied to a reward if it takes k time-steps to perform the requisite state-action pair: $R_{t+k}(s, a) = \gamma^k R_t(s, a)$.

For ease of analysis, we only impose temporal discounting (and technically allow an infinite time horizon – assuming R does not grow geometrically over time, the cumulative reward will remain finite because the discount factor γ will send "distant" rewards to zero fast enough).

Technically, everything we have described so far can be modeled as a *Markov decision process (MDP)* and solved exactly. In most reinforcement learning situations, a further issue is that the agent does not know R and T ahead of time – it has to "explore" the world an perform different actions to determine R and T, while at the same time "exploiting" the knowledge it has gained to try and maximize its cumulative reward. It's worth discussing the MDP a bit more, as it will help in the "unobserved" case.

10.1 Markov decision processes and the Bellman equations.

In the case of observed R and temporal discounting, we can define a *value function* associated with each state $V(s)$. V is a measure of "how good" a specific state is; we want to get to states with higher V. We can also define a *Q-value* dependent on state and action $Q(s, a)$, which measures "how good" a specific action is at a particular state; we want to perform actions with the highest Q-value.

In general, if we knew the final values of Q, we could write:

$$V^*(s) = \max_s Q^*(s, a)$$

$$Q^*(s, a) = \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma \max_{a'} Q^*(s', a'))$$

That is, "my utility if I could do if I'm at s and perform a = performing a, and depending on the state s' I end up, add the discounted utility I would get from performing the optimal action a'." These are called the *Bellman equations.*

(This sort of "one-step-unrolled equivalence" is a type of convergence you can often find in Markov chains. One can find the long-term visitation frequency of states in a Markov chain using the *balance equations*.)

This wording might suggest something like a *policy* $\pi(s)$ – a function that says, given a state s, what the optimal action a is:

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a)$$

The great thing is, similar to Markov chain calculations, we can "unroll" these equations all the way to $t = 0$. That is to say, we can iterate on our current best estimates for V or Q to get the next time-step's estimates, and we can repeat until convergence (at which point, we can determine an optimal policy π^*):

$$V^*(s) = \max_s Q^*(s,a)$$

$$Q_{k+1}(s,a) = \sum_{s'} T(s,a,s')(R(s,a,s') + \gamma \max_{a'} Q_k(s',a'))$$

10.2 RL: The unobserved case.

We know how to maximize cumulative reward in a fully observed case. What if we don't know T and R? We need to estimate these functions. More specifically, we're interested in its expectation: $E_X[f(X)] = \sum_{x \in X} p(x)f(x)$. One could try two approaches to estimation:

1. Model-based approach. We try to figure out the distribution of X and get $\hat{p}(x)$, then calculate the expectation $E[f(X)] \approx \sum_{x \in X} \hat{p}(x)f(x)$. (The main drawback here is having to store and update all of the information to compute $\hat{p}$ for every state-action pair for R/T.)
2. Model-free approach. We use the Law of Large Numbers to our advantage and estimate $E[f(X)] \approx \frac{1}{n}\sum_i f(x_i)$ (where you have performed the state-action pair n times).

The model-free approach is often preferred for lower overhead costs.

How do you update? It could make sense to give more recent samples higher weight than older samples, in which case you could use an *exponentially weighted moving average (EWMA)*. The EWMA, in recurrence relation form, is

$$\bar{x}_n = \alpha x_n + (1-\alpha)\bar{x}_{n-1}$$

Applied to Q-value iteration, we have

$$Q_{k+1}(s,a) = \alpha \times (R(s,a,s') + \gamma \max_{a'} Q_k(s',a')) + (1-\alpha)Q_k(s,a)$$

where the $s, a, s', R(s,a,s')$ were sampled and Q_k is the previous estimate.

How do you determine when to exploit the information you have and when to explore more to see if you find something useful? You can "hardcode" exploration into the agent's behavior. A simple approach is the ϵ-greedy algorithm: At any given state s,

1. choose a completely randomly with probability ϵ
2. choose $a := \pi_k(s)$ ("greedily" based on gathered information) with probability 1 - ϵ