Naive Cartographer: A Markov Decision Process Learner

Brandon Rohrer March 18, 2024

tl;dr: A naive Bayes algorithm for learning Markov decision processes with fuzzy state

Fuzzy Naive Cartographer (FNC) is an online algorithm for model-based reinforcement learning (RL), specificially it learns a Markov decision process and an action-value function.

FNC assumes a fuzzy state vector, namely

$$\mathcal{S} = [s_0, s_1, s_2, \cdots, s_i, \cdots, s_n] \text{ where } s_i \in [0, 1] \quad (1)$$

and makes the unusual implementation decision of treating all elements of the state vector as conditionally independent, as in Naive Bayes classification algorithms.

In FNC all actions are discrete.

$$\mathcal{A} = [a_0, a_1, a_2, \cdots, a_j, \cdots, a_m] \text{ where } a_j \in \{0, 1\} \quad (2)$$

FNC has two internal actions as well, a "do-nothing" action as a_{m+1} and an "average" action as a_{m+2}. The do-nothing action explicity represents refraining from taking an action, and the average action is always 1. It allows FNC to learn typical feature-outcome patterns that may be independent of agent actions.

Rewards are real valued, but FNC also allows them to be absent, and it allows for multiple reward channels.

$$\mathcal{R} = [r_0, r_1, r_2, \cdots, r_k, \cdots, r_l] \text{ where } r_k \in \{\mathbb{R}, \emptyset\} \quad (3)$$

At each time step, t, an agent receives a state s and reward r and takes a set of actions a. The state from the following iteration is represented as s' to show that it is the outcome of previous state and actions. FNC operates on a historical trace of prior states $\overline{s}$ where

$$\overline{s}_{i,t} = \max(s_{i,t}, \gamma_f \overline{s}_{i,t-1}) \quad (4)$$

In this recursive calculation $\gamma_f \in [0, 1]$ is a discount factor determining how quickly the influence of prior states decays.

FNC also operates on a historical trace of state-action pairs, $(\overline{s, a})$, where

$$(\overline{\overline{s_i}, a_j})_t = \max((\overline{s_i}, a_j)_t, \gamma_r (\overline{\overline{s_i}, a_j})_{t-1}) \quad (5)$$

Again here γ_r is a discount factor, and may be different than γ_f. The activity of a state action pair is the product of the individual activities of the feature and the action.

$$(\overline{s_i}, a_j) = \overline{s_i} a_j \quad (6)$$

Transition probabilities are defined in terms of the all-time sums of the transition occurrences and the feature-action trace activities.

$$p(s'_k | (\overline{\overline{s_i}, a_j}) = \frac{\sum_t (\overline{\overline{s_i}, a_j}, s'_k)_t}{\sum_t (\overline{\overline{s_i}, a_j})_t + 1} \quad (7)$$

Conditional transition probabilities at each time step are aggregated over all active features with a weighted $\max()$ operation.

$$p(s'_k | a_j) = \max_i (\overline{s_i}\, p(s'_k | (\overline{\overline{s_i}, a_j})) \quad (8)$$

FNC also generates an uncertainty estimate associated with each feature-action pair.

$$\delta_{ij} = \frac{1}{\sum_t (\overline{\overline{s_i}, a_j}) + 1} \quad (9)$$

The conditional uncertainty is aggregated in the same manner as transition probabilities.

$$\delta_j = \max_i (\overline{s_i}\, \delta_{ij}) \quad (10)$$

The estimate for reward r_k and feature-action pair $(\overline{\overline{s_i}, a_j})$ at time t is $Q_{ijk,t}$ and is calculated incrementally via this update rule

$$Q_{ijk,t} = Q_{ijk,t-1} + \alpha (\overline{\overline{s_i}, a_j})_t (r_{k,t} - Q_{ijk,t-1}) \quad (11)$$

where $\alpha \in [0, 1]$ is a learning rate.

The conditional reward for each action a_j is given by

$$Q_j = \sum_k \left(\frac{\sum_i Q_{ijk} \overline{s_i}}{\sum_i \overline{s_i}} \right) \quad (12)$$

1. Concepts and Related Work

1.1. Reinforcement Learning

To set the stage, the Naive Cartographer falls into a category of machine learning called reinforcement learning (RL). A RL algorithm learns to take actions that lead to rewards and avoid actions that lead to bad experiences, much like a toddler learns to prefer cookies over brussel sprouts through trial and error. This is distinct from the other two main branches of machine learning: supervised learning, such as algorithms that learn to detect fraudulent charges on our credit cards, and unsupervised learning, such as might be used to group tracks into genres in a music catalog. The favored reference in the field by Sutton and Barto is freely available in pdf .[1] I'll be mimicking their notation here.

A single pass of a RL algorithm looks something like

1. See a cookie. (Sense the state of the world, s.)
2. Eat the cookie. (Take an action, a.)
3. Enjoy the cookie. (Experience a reward, r.)

The next pass may be "See a brussel sprout, Eat the brussel sprout, Wrinkle nose at the brussel sprout experience." It will also have a state-action-reward (s, a, r) structure. The algorithm repeats the cycle of s, a, r tuples until the end of time or until someone turns it off.

1.2. Value function

The thing that makes RL algorithms useful is that after a while they get good at knowing which action will bring the most reward. After trying a cookie and a brussel sprout a couple of times, a clear pattern will emerge. Then, when presented with the state s of "See a brussel sprout and a cookie" the algorithm will have the option of two actions: "Eat the cookie" or "Eat the brussel sprout." It will only need to reach back into its accumulated experience to know that eating the cookie will lead to higher (immediate) reward than eating the brussel sprout.

The way this accumulated state-action-reward experience is represented is a value function. For every possible state, and for every possible action in each of those states, the value function is the reward that is expected. It is typically written as $Q(s, a)$ when it is looking ahead to include all anticipated rewards in the next few time steps, and $r(s, a)$ when considering only the immediate result of the current time step. Once a RL algorithm has learned the value function well, it can always make the choice that leads to the highest reward. It need never eat another brussel sprout again.

1.3. Tabular RL

An important distinction among RL algorithms is whether they require their states and actions to be **discrete** (that is, you could write them out in a list if you wanted to) or whether they can handle **continuous** state variables and action outputs. The cookie/brussel sprout example is discrete. It's easy to list out all possible states

- See cookie
- See brussel sprout
- See both
- See neither

and all possible actions

- Eat cookie
- Eat brussel sprout
- Eat both
- Eat neither

Not all actions are possible in all states. You can't eat a cookie if you can't see a cookie. That's OK.

Chess is another example of an environment with listable states (a discrete state space, $\mathcal{S}$) and listable actions (a discrete action space, $\mathcal{A}$). It's true that the lists of possible states and actions are very long, but if you were sufficiently determined you could count them. They don't go on forever.

Physical robots are often examples of continuous state and action spaces. The possible angles of a robot elbow can't be listed without cheating and grouping them together into bins. The amount of torque applied to a robot elbow is similarly impossible to list. For any two listed torques, I can always find another torque that sits between them.

Discrete state and action spaces are interesting because they allow you to (in theory) capture the value function $Q(s, a)$ in a single big table. Each row is a state, each column is an action, and the value of each cell is the expected reward. RL algorithms that work with discrete state and action spaces are called **tabular**, because they can represent their value function in a table, as in Table 1.

FNC is a tabular approach, so it does represent the world as discrete. However it can also be used with continuous state spaces too, thanks to the cheat of binning.

[1] Sutton, R. S., and Barto, A. G. (2020) Reinforcement Learning: An Introduction, second edition, Westchester Publishing Service. http://incompleteideas.net/book/RLbook2020.pdf

	Eat the cookie	Eat the brussel sprout	Eat both	Eat neither
See a cookie	1	NA	NA	0
See a brussel sprout	NA	-1	NA	0
See both	1	-1	0 (+1 and -1)	0
See neither	NA	NA	NA	0

Table 1. Value function for the cookie/brussel sprout world

1.4. Exploration

When a RL algorithm first starts out, it doesn't have a good value function. That only happens through trial and error. After a short time, it learns some of the value function, enough to get by, but most of it is still unknown. You may have experienced this when choosing what to eat at a restaurant.

In this scenario you are the RL algorithm, also known as the **agent**—the entity responsible for deciding what to do next. Your current state s is "staring at a menu". Your function is to choose an item to order and eat. For instance, one of your potential actions, a, might be "cheeseburger". Your action space $\mathcal{A}$ consists of all the items on the menu. Your objective is to choose the most satisfying item, maximizing your reward r.

If you've only tried three things on the menu so far, you're in a tricky place. Sure, the fish and chips was really good. That action had a high reward. But the next time you order, do you get the fish and chips again? Or branch out and try something new? Sticking with what you know is the safest course. It's also called **exploitation** in the sense of exploiting, or making the most of, your past experience. Trying something new, or **exploration**, is more of a gamble. It's possible that the shepherd's pie will be the best thing you've ever eaten but it might also be gross.

Learning your favorite items on a menu is hard enough, but it's even harder when the world changes—when seasonal menu rotations kick in or there is a Portabella mushroom shortage or the management turns over and starts cutting costs with cheap cheese. There is a real danger that the value function you learned becomes outdated and inaccurate. Your favorite dish might get pulled from the menu or re-imagined by an ambitious chef. A world that refuses to stay the same is considered **nonstationary** in RL terms, and dealing with nonstationary worlds is particularly challenging. Nonstationarity means that the need for exploration never goes away. Even after you've explored and learned your entire value function, you have to keep exploring in case it changes.

Striking the balance between exploration and exploitation is a recurring theme in RL. The best trade-off depends entirely on the specifics of the world and the goals of the agent. One of the ways that high performing algorithms distinguish themselves is by cleverly juicing as much information as possible out of their explorations.

1.5. Eligibility traces

Yet another challenge to building a useful value function comes when there is some delay between taking an action and reaping the reward. Imagine a robot whose goal it is to open a door. A successful episode might require a whole sequence of (state, action) pairs before the goal is achieved.

1. (sitting in center of room, drive toward door)
2. (sitting in front of door, reach for handle)
3. (grasping handle, turn handle)
4. (grasping rotated handle, pull rotated handle)

Only at this point is the goal of an open door achieved (Yay!) and a reward is awarded. Now the question arises of how to allocate the reward. Certainly Step 4 deserves credit for the success and is assigned a high value, but Steps 1-3 were necessary preconditions for reaching Step 4, and so it stands to reason that they should be assigned some value too. The next time it is sitting in the center of the room, the robot should know that driving toward the door is a valuable course of action. To establish this pattern, some amount of value should be assigned to the Step 1 state-action pair as well.

There are a variety of strategies for sharing credit across the state-action history, but conceptually most of them resemble a decaying allocation that gets smaller as it reaches further back in time. For example, the value of Step 4 might be allocated half of the reward, one quarter for Step 3, and one eighth each to Step 2 and Step 1. This sequence of weights $[\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}]$ is called an **eligibility trace** and specifies how to spread the reward backward in time to update the value of state-action pairs. RL algorithms differ in how they calculate eligibility traces, and how they update them when the same state-action pair is visited multiple times.

1.6. Model-based RL

Learning a value function for all possible state-action pairs can be an effective approach to solving a problem, but it's not the only way. It's analogous to blindfolding someone

in the street and guiding them on their walk to the grocery store by calling out "hotter" and "colder" as they wander. The signal only helps them determine their next step. It doesn't convey how far away the store is, or whether there might be another path.

Alternatively, we could just hand the person a map with "X" marking the location of their destination and let them plot their own course. This is the concept behind model-based RL.

Typically the RL agent doesn't have it quite this easy. It has to be its own cartographer, keeping track of its past experiences and stitching them together into a picture of the landscape. The person looking for the grocery store would note that starting from the bookstore and going north lands them at the bank. Going east from there lands them at the coffee shop. And then going north again gets them to the grocery store. This can be represented as state-action-state transitions, (s, a, s').

1. (bookstore, go north, bank)
2. (bank, go east, coffee shop)
3. (coffee shop, go north, grocery store)

After doing this for long enough, these transitions can be patched together to figure out where one would end up after taking any number of jumps in any direction.

Because the real world is frightening and unpredictable, taking a particular action while in a particular state doesn't always produce the same result. When flipping a coin or buying a lottery scratcher or asking someone on a date, results can vary. This nuance can be represented mathematically by talking about the probability of landing in state s' when starting from state s and taking action a: $p(s'|s, a)$. In the coin flipping example,

$$\begin{aligned} p(\text{heads}|\text{have coin, do a flip}) &= .5 \\ p(\text{tails}|\text{have coin, do a flip}) &= .5 \end{aligned}$$

This is a collection of the probabilities of transitioning from one state to another, a state transition model. The combination of a state transition model and a value function is a **Markov decision process**. If you drop this name in casual conversation you can fool people into thinking you really know what you're talking about.

If you feel inspired to do more reading on model-based RL, I recommend Chapter 8 of Sutton and Barto, which discusses the Dyna architecture, an early showcase of the power of the method.

1.7. Goal-conditioned RL

Working from a model opens up a whole new world of possibilities. Imagine that our grocery shopper, after making thousands of trips to the grocery store, decides one day to go to the pub. If the agent were relying on a value function they're hosed. All of their accumulated experience only tells them how close each location is to the grocery store and which actions are likely to get them closer. Nothing tells them how to get to a pub. Not only do they have to learn the value function for the pub but they also have to un-learn all of the values that would drive them toward the grocery store. They will not be getting that pint any time soon.

But luckily our grocery shopper was a model-based agent. They recorded all their walks and wanderings and can search through their past experiences to learn that going west and south from the bookstore will get them to the pub. When the goal changed, they were able to adapt right away.

Having a model enables an agent to do more than one task. If the door opening robot learned to do its job using only a value function, then it would truly be a single-purpose robot. However, if it were model-based then it could be quickly repurposed by giving it a new goal, say, of passing the butter at the dinner table. Its state transition model would allow it to chart a path from wherever it is in the state space (perhaps it starts out sitting at the center of the room again) to the end goal state of having the butter be in front of my plate at the table. There may be any number of intermediate steps, such as "move to the table", "grasp the butter dish", "move the butter dish near me", and "release the butter dish", but the composite map of the state-action-state transitions in the model would make those steps knowable.

Rewards are still valuable for elements that don't change. In human bodies, pain is a useful signal that something has damaged our hardware. Avoiding pain helps us avoid situations that might damage our hardware. Assigning a negative reward to pain, or the robot equivalent, is a good measure to have in place. No matter what else it is doing, it serves a robot well to avoid overheating its motor coils or moving its limbs too quickly in an enclosed space.

The process of using a transition model to plan a sequence of actions to reach a goal is prosaically known as planning and the portion of the RL agent that does this is called a **planner**. Notably, FNC does not contain a planner. It is a model and value function learner, but it relies on another component to do the planning and goal setting. The details for what a FNC-complementary planner needs to do, together with some examples, will be given in the Architecture section to follow.

1.8. Intermittent reward

The default assumption of the reinforcement learning problem is that there is a reward signal available at every time step. On each pass through the sense-learn-act cycle the reinforcement learning agent has some basis for updating its value function. This is a reasonable assumption when reward is a function of sensor information. When a child eats an orange slice, the reward is sensed directly through hard-wired circuits in the brain. When a robot reaches its charging station, it can collect the reward that has been explicitly tied to getting recharged. When a child falls down, the negative rewards of pain and frustration are associated with a host of physical experiences that it can sense directly. When a robot has a negative reward associated with exceeding an electrical current limit, it can be confident that it will never miss that punitive reward signal. In all of these cases, the reward is a direct consequence of sensed values.

There are many reinforcement learning problems where the reward only occurs occasionally. For instance, an agent playing tic-tac-toe or solving maze often won't know how well it's doing until the very end. These types of environments are said to have a **sparse reward**, and there are clever approaches to making the most of these. But even in cases of sparse reward, the channel is always active. The agent can operate with confidence that every tic-tac-toe win or loss will come through as a positive or negative reward. It will never have to wonder how it did on a given game.

There is an alternative formulation for the reinforcement learning problem where reward is **intermittent**. Sometimes the reward signal is present, in which case it may be positive or negative or zero. But other times the reward signal is simply absent, and there is no information available about how well the agent is doing. This is a reasonable way to represent reinforcement that comes from human trainers. If there is a human in the loop providing feedback on whether a robot gripper is grasping a part correctly, they can provide explicit reward signals for "do it more like that" (reward of +1), "do it less like that" (reward of -1), or even "meh" (reward of 0). But if they step away from their station and the robot is running autonomously, it doesn't make sense for it to assume that it's receiving a reward of zero for every action from the absent human. Rather, it makes more sense to recognize that the signal is temporarily missing, and not to perform any additional updates on the reward estimates until that changes. This allows past human training to persist, rather than rapidly decay with the passage of time. It makes efficient use of human supervision, that sparsest of resources.

One way to characterize this difference is **internal** versus **external** rewards. (The distinction isn't quite the same as intrinsic versus extrinsic rewards, but close enough.) Internal rewards can be anything that is a direct function of sensor information. A reward for reaching a battery charger or getting a grasp on an object or moving solar cells into a sunbeam. There can also be a cost, a small negative reward or gentle punishment, for high motor currents or excesively fast actions or bumping into things.

An external reward would be something that is not directly a function of sensors—a signal provided by a human, some external process, or agent, or even some odd state that the robot can't sense directly. External rewards can easily be intermittent, and in many cases it may be useful to assume a default of no reward information, rather than a reward of zero, as is customary.

1.9. Naive Bayes

There's a famous gotcha in tabular RL, the ominously named Curse of Dimensionality. It comes from the fact that the state of an agent in their environment includes all of the sensor information, everything that's known about both the robot and its world. In chess if you move one pawn, the board is in an entirely new state. If a robot moves one finger, it's in a new state. If you change one pixel in the camera of a computer vision agent, it's a new state. The list of states becomes unfathomably long for all but very simple environments.

The most common way to handle this is with some type of approximation where the agent extends what it learns about one state to similar states, so it doesn't have to experience every individual variation. It's a form of generalization, and it works well for a lot of applications. If you've watched simulated two-legged robots wobble around, you've seen it in action.

There's another alternative that I haven't seen anyone try yet, which is to handle each of the features separately, instead of combinatorially. It changes the notion of state. If a robot twitches a finger, almost every other aspect of it is still considered to be in the same state. Only the new finger position is a change in state. The whole representation of state is broken up into independent components which are assumed not to interact with each other.

Another term for this is "conditionally independent features," and it is the assumption behind **Naive Bayes** methods. (Confusingly, this has little to do with Bayesian inference or Bayes Law.) It is naive in the sense that, yes of course we know that states aren't independent. 70 degrees west longitude is a very different experience if you're at 20 degrees north latitude or 70. But we are going to pretend that they are independent because it's convenient to do so.

This approach can also be imagined as a whole bunch of tiny, dumb models, each based on only one feature. In machine learning a collection of models like this is called an **ensemble**, and ensembles have proven to be remarkably ef-

fective. Even though each individual model is only partly right, or only right part of the time, when aggregated together they do a good job steering the ship in the right direction. It's a metaphor for a healthy democracy.

For this to work well, features have to contain important information individually, rather than in combination. One way this can break is when individual features are too narrow. For instance, when describing the position of a piece on a chessboard it would be reasonable to have one set of features for the rows 1 through 8 another set of features for representing the columns a through h. This representation is concise, just 16 features, but it puts the model at a disadvantage. Trying to reason about a knight in row 3 without also knowing what column it's in doesn't make much sense. The answer would be very different depending on whether the knight is in column b or column e. A better set of features would be the combination of all rows and columns, a1 through h8. It results in more features, 64 instead of 16, but each one is much more informative.

It's tough to strike the right balance between making features informative and keeping them down to a manageable number. Individual letters aren't very informative, but when they are grouped into words they start to have some interesting semantic content. However there are far more words than letters. Individual pixels don't help discern the content of an image, but once they are combined into edges and textures they start to give clues as to what's going on. However there are many more ways that pixels can be combine than there are individual pixels. Choosing good features is an art and requires thinking carefully about what each one means, especially how it might be useful to the agent trying to navigate its world. This is the case in all machine learning work, but especially so for a Naive Bayes approach.

The particular problem of learning interesting features from low level sensor data is partially addressed by the Ziptie feature learner. Although FNC can work well on its own, it is designed to pair well with Ziptie to remove some of the manual crafting of features.

1.10. Fuzzy variables

In tabular RL, each state and action is categorical, meaning that it is discrete, distinct from the others. A chess piece is on square d6 or it is not. A switch if flipped or it is not. Zero or one. True or False. Tabular RL can still be used with continuous states, but the states have to be binned first. Test scores that can fall anywhere between 0 and 100 would have to be grouped, say into tests with scores between 90-100, 80-90, 70-80, etc. Then each bin could represent a different state.

FNC requires categorical state variables too, but with a twist; it allows them to take on any value between zero and one. For example, the robot might have a state variable of "obstacle nearness", derived from a proximity sensor. A zero might mean there was no obstacle within the range of the sensor, and a one might mean that there was an obstacle almost touching the robot. A .5 might mean that the obstacle was in the middle distance. A .2 might mean that the obstacle was fairly close.

The technical term for variables that represent a single category and can vary between 0 and 1 is **fuzzy**. A more detailed description with examples can be found in Section 2 of the Ziptie paper.

2. Algorithm

The core of the Naive Cartographer is a Markov Decision Process model pair:

A transition probability estimate and

$$p(s'|s,a) \tag{13}$$

A reward estimate

$$Q(s,a) \tag{14}$$

(following the notation of Sutton and Barto). If you already have a mental model of these, that will serve you well, but FNC puts its own spin on them. These quirks, the deviations from the default vanilla interpretations, are what makes FNC unique.

Quirk 1: Fuzzy state varibles

As mentioned above, FNC requires all state values to be between 0 and 1 and interprets them as fuzzy variables. The later implications of this are many, but they don't otherwise look odd as a state representation. I'm not aware of any other RL modeling approaches that take this approach.

Quirk 2: Conditional independence of features

The weirdest thing FNC does is assume that all of its features have independent effects on the outcome of the model. This assumption is patently wrong, but it is so helpful in taming the curse of dimensionality that FNC gets good mileage out of it despite that. This trade-off has been successful in natural language processing (NLP) classification tasks under the name of Naive Bayes. As far as I know it's never been applied to reinforcement learning.

This Naive assumption allows several tricks and shortcuts that will be described in later paragraphs.

Quirk 3: Features instead of states

This quirk is worth a closer look because it breaks some of our hard won intuitions about state spaces. Traditionally Markov Decision Processes (MDPs), describe an agent that was in a state, s, took an action, a, and ended up in a new state, s'. FNC's state representation is not actually a state representation at all. In physics, the state of an object is a specific set of information that lets you know exactly what it's doing. In the case of a bowling ball, if you know where it is, how fast it's moving, and how fast it's spinning, you can predict everything it's going to do next with high accuracy. The bowling ball's state consists of position, orientation, velocity, and angular velocity, all in three dimensions. A physics simulator would need all of these to represent the bowling ball. The notion of state in MDPs is this holistic representation, and a good portion of RL analysis subtly bakes in this assumption.

FNC does not presuppose that all the state information is contained in its inputs. MDP learning problems where what is observed is different than the state of the system can be described as a Partially observable Markov decision process (POMDP). POMDPs posit that there is a concise set of state variables, but assumes that the sensors in use only reflect them indirectly. This would be akin to estimating the state of the bowling ball based only on the audio input from an array of microphones placed along the lane. Unlike POMDPs, FNC takes the additional shortcut of ignoring the underlying state variables entirely. It focuses on using the sensor information to predict future sensor information. It's a pragmatic approach and adroitly avoids any philosophical rabbit holes of what it means to represent the state of something complex in its world, like a human. (Warning: If you drop "POMDP" into a casual conversation it will look like you're trying too hard.)

Most practical RL implementations make this casual sensors-are-good-enough assumption too, but this can create tension between algorithms that assume they are working with complete state and sensor inputs that don't contain it. FNC tries to keep expectations aligned and doesn't assume that inputs contain complete state or even that they are all relevant. For this reason, I've taken to using the term **feature** and feature vector, rather than state, in the code and comments, although to maintain consistency with the mathematical notation of Sutton and Barto, a given set of features is still s, and the full space of possible feature combinations is still $\mathcal{S}$.

$$\mathcal{S} = [s_0, s_1, s_2, \cdots, s_i, \cdots, s_n] \text{ where } s_i \in [0, 1] \quad (15)$$

The outcome features of a feature-action-new-feature transition, (s, a, s'), will also be referred to as **outcome** to differentiate it from the initial features.

Quirk 4: Feature-action pair activities

In traditional tabular RL, a state-action pair can be represented in a two-dimensional array where rows are distinct states and columns are individual actions. In this 2D one-hot state-action array, a single state-action pair might be a cell with a 1, where everything else is a 0, illustrated in the left panel of Figure 1.

In FNC this is a bit different. A feature-action pair is still represented as a two-dimensional array, but with each row representing a single feature, rather than a distinct feature array. This gives the tremendous benefit that the number of rows will stay at a manageable size, rather than try to encompass every conceivable combination of valid feature values. (Thank you Naive Bayes!)

This also means that the 2D array will no longer be one-hot. Many individual feature values may be non-zero at any given moment, so the corresponding number of feature-action cells will also be non-zero. The right panel of Figure 1 shows and example of how this might play out.

Since features can take on any value between zero and one, this also means that feature-action values will not be limited to one and zero. It opens up the notion of a feature-action activity, which FNC defines to be the product of the feature activity and the action activity (which has already been limited to zero and one).

$$(s_i, a_j) = s_i a_j \quad (16)$$

Quirk 5: The condensed trace

It's common in tabular RL algorithms to track a historical trace of recently active state-action pairs in order to appropriately attribute reward when it is encountered. When a reward is finally encountered it gets attributed most strongly to the most recent state-action pair, with a decay as it reaches further back in time. For instance, it may use a decay schedule of attributing the full reward r to $(s_i, a_j)_t$, a discounted reward $\gamma_r r$ to $(s_i, a_j)_{t-1}$, where γ_r is a discount factor between 0 and 1. From there, an additional factor of γ_r is applied at each time step: γ_r^2 to the feature-action pair at time $t-2$, γ_r^3 at $t-3$, and so on with γ_r^n at time $t-n$.

reward	feature-state pair
r	$(s_i, a_j)_t$
$r\gamma_r$	$(s_i, a_j)_{t-1}$
$r\gamma_r^2$	$(s_i, a_j)_{t-2}$
$\vdots$	$\vdots$
$r\gamma_r^n$	$(s_i, a_j)_{t-n}$
$\vdots$	$\vdots$

With a little rearrangent, it is possible to shift the decay from the reward to the feature-action activities. This is

	Keep email	Archive email	Delete email
Known sender, no attachment, no misspellings	0	0	0
Known sender, has attachment, no misspellings	1	0	0
Unknown sender, has attachment, contains misspellings	0	0	0
⋮	⋮	⋮	⋮

	Keep email	Archive email	Delete email
Known sender	.8	0	0
Has attachment	0	0	0
Contains misspellings	.4	0	0

Figure 1. A one-hot state-action array on the left and a FNC feature-action array on the right for an email routing bot.

straightforward, since FNC is already representing the pairs with real values between zero and one.

reward	feature-state pair
r	$(s_i, a_j)_t$
r	$\gamma_r(s_i, a_j)_{t-1}$
r	$\gamma_r^2(s_i, a_j)_{t-2}$
⋮	⋮
r	$\gamma_r^n(s_i, a_j)_{t-n}$
⋮	⋮

In this formulation, the decay is applied to the feature-action activities, rather than to the reward. One way to calulate this is to reach back in time and incrementally decay every feature-action pair by an additional factor of γ_r.

Since FNC is making use of a 2D array to represent feature-action activites, it can cut a small corner and put the entire feature-action history of the agent in one array. The the discounting update at each timestep is a straightforward matter of multiplying every element in the array by γ_r.

Using one array to represent the whole history of feature-action activities means that it will have to account for collisions—times when the same element in the array has a nonzero value associated with multiple time steps. The way that FNC resolves this is with the maximum operator. Whichever of the two values is greater wins.

Putting it all together,

$$\begin{aligned}(\overline{s_i, a_j})_t = \max(\quad & (s_i, a_j)_t, \\ & \gamma_r(s_i, a_j)_{t-1}, \\ & \gamma_r^2(s_i, a_j)_{t-2}, \\ & \cdots, \\ & \gamma_r^n(s_i, a_j)_{t-n}, \\ & \cdots)\end{aligned}$$

where $(\overline{s_i, a_j})_t$ is the condensed, discounted history of the agent's feature-action activities. With a little fancy footwork, this can become an incremental update, well suited to the iterative computation that happens in a RL agent.

$$(\overline{(s_i, a_j)})_t = \max((s_i, a_j)_t, \gamma_r(\overline{s_i, a_j})_{t-1}) \qquad (17)$$

The choice of γ_r determines how rapidly history is forgotten and how far back in time an agent's successes and failures can be attributed to its decisions. An agent playing tic-tac-toe can get by with a γ_r of around .7 because it only needs to reach back 4 of 5 moves to cover the whole game. An agent solving a maze that takes about 100 moves to solve will probably want a γ_r in the neighborhood of .98 so that it can apply its learning far enough back in time to be useful on its next pass through the maze. And an agent learning to classify a string of unrelated objects might be best served by a γ_r of zero. It would be a mistake to apply any of the reward to the agent's prior decisions. Each time step is independent of the last and however it goes, shouldn't affect the agent's perception of its performance on prior objects.

Quirk 6: Condensed feature history

The feature-action trace enables two similar but distinct things. The first is learning multi-step tasks, as illustrated in the examples above. The second is compensating for a slow world. If an agent can sense and act very quickly compared to its world's dynamics, it may take several time steps before it realizes the reward from an action. A fast agent may throw a basketball toward a hoop, then perform several iterations through its RL loop before the ball goes through, earns a point, and earns a reward for the agent. A condensed history allows the critical feature-action pair from several moments prior to get its fair share of reward allocated to it. Alternatively, any time a reward is manually administered by a human it will be subject to humans' cognitive processing times. Even under the best of circumstances, this will not be faster than 150 ms. Typically it will be ten times greater. It's easy for a very chill agent to complete several iterations in this time.

This trick of accounting for fast/slow system mismatches by spreading credit across the history of the fast system can be applied to feature activities on their own too. In an ideal RL system, the sensed state at each time step would be current and complete (observable), but in practice it can be both indirect and delayed (partially observable), as detailed in Quirk No.3 above. When creating feature-action pairs, this means that it may actually be a feature sensed several time steps prior that is most relevant to the current action selection.

A natural way for this to manifest is, once a feature is detected, the underlying state that triggered it remains valid for some time. For instance, consider a robot that is learning to respond to human verbal commands. When the human says "stop", that is sensed by the robot and reported in the feature array at time step t. However, the underlying state of the system includes the intent of the human, which may remain for many time steps thereafter until the robot stops doing whatever it is doing. Having a condensed representation of feature history gives the robot a short-term memory for recent features, and allows it a small time window for experimentation to find the action to pair with the feature to get a reward.

To allow these connections to be learned, FNC also maintains a condensed representation of feature history in the same way it keeps a feature-action history.

$$\overline{s}_{i,t} = \max(s_{i,t}, \gamma_f \overline{s}_{i,t-1}) \tag{18}$$

where γ_f is another discount factor specific to the feature history. It can have a different value than γ_r, allowing for different decay dynamics for the feature history.

In FNC the condensed feature-action trace uses the condensed feature history instead, so its full expression is

$$(\overline{\overline{s_i}, a_j})_t = \max((\overline{s_i}, a_j)_t, \gamma_r(\overline{\overline{s_i}, a_j})_{t-1}) \tag{19}$$

Quirk 7: The do-nothing action

The "do-nothing" action is a special actions included in every FNC model. It lets the model represent what happens if it deliberately refrains from taking any actions. If an agent is already in a high reward state and expects to keep receiving that reward, then its best option is to not take any new actions, just sit and let the reward continue to roll in. This could be a robot vacuum sitting on its charging station, or an interplanetary probe waiting for its next instruction from earth. In some situations, the best result comes when the agent nearly sits on its ass and does nothing. The do-nothing action gives it that option.

Quirk 8: The average action

The second special action in every FNC model is the "average" action. It represents what is likely to happen in a given circumstance, if the model is deliberately blind to which action was taken. It is a way for the agent to learn when its actions don't matter. There are many circumstances where the world is going to do what the world is going to do and there's nothing the agent can do in that moment to change its course. If an agent is watching the clock, no action that it can take will prevent time from advancing from one second to the next. An agent that is observing a ball sail through the air will not be able to alter the ball's course, no matter what it does (unless perhaps it is a robot with very long arms, or is very good at throwing rocks). The average action finds the average outcome for an initial set of features, regardless of the action taken. It helps the agent to learn what patterns depend only on the previous state, and not on any action that was taken. Learning this is useful because it frees the agent up to make decisions based on other costs and benefits that are causal.

The average action highlights a common assumption of RL agents, namely that every action they take impinges on the external world in some way. In adversarial environments, they can spend a lot of resources trying to learn to influence things that they have no control over. FNC won't have any better luck at controlling those things, but it will at least be able to learn to stop worrying about them.

Here is the full representation of actions generated by FNC

$$\mathcal{A} = [a_0, a_1, a_2, \cdots, a_j, \cdots, a_m] \text{ where } a_j \in \{0, 1\} \tag{20}$$

Actions are discrete, on or off. Internally to FNC, and in the predictions it provides to the planner, the do-nothing and average actions are appended to the array as a_{m+1} and a_{m+2} respectively.

Not a quirk: Reward estimation

The way reward is learned is anything but quirky. It's more of a classic. It is the same approach described in Equation 2.5 of Sutton and Barto in the context of multi-armed bandits, a simple RL scenario. For a given feature-action pair

$$Q_t = Q_{t-1} + \alpha(r_t - Q_{t-1}) \tag{21}$$

where is Q_t is the expected value of reward for that feature-action pair as of time t, r_t is the observed value, and α is a constant learning rate between zero and one, usually much closer to zero than one.

This is a recursive filter, using the previous time step's value to estimate the next. It results in a exponentially-weighted moving average of the reward. The entire history of observed rewards contribute to the estimate, but recent observations are weighted much more heavily than older ones. It has the useful properties that it doesn't jump around too much from one time step to the next (as long as α isn't too large), but it will converge on the true average value of r (if the average value of r is constant) and will adapt to any changes in r as well (if its average isn't constant).

The expression above for updating Q only applies when the feature-action pair ij is active, that is, when feature i has occurred and action j has been taken. In traditional tabular RL, this is an all-or-nothing proposition, but in FNC it can by fractionally valued. Here is the more complete and explicit version that includes the condensed feature-action history.

$$Q_{ij,t} = Q_{ij,t-1} + \alpha(\overline{s_i, a_j})_t(r_t - Q_{ij,t-1}) \tag{22}$$

Quirk 9: Reward prediction

The recursive reward estimation method above creates a two-dimensional array of reward estimates where every row represents a feature and every column an action. In this form it becomes a very helpful source of information about what actions the agent should take on the following time step. If it were a traditional tabular learning algorithm, the planning procedure would be to isolate the row associated with the most recent state observation. This row would show, for all the possible actions the agent might take, what the expected reward would be. A greedy agent would pick the action associated with the highest reward and be done with it. A more exploratory agent might choose one of the other actions just to learn more about them. But either way, the action-reward relationship is the foundation on which these decisions are based.

In FNC many individual features can be active at once, and can be active at any value between zero and one. There is not just one row of reward estimates to be pulled out of the array, there are several. The way FNC handles this is to do a weighted average of all the rows, based on their corresponding feature activities. This ensures that rows associated with highly active features are counted strongly, and rows without any significant activity are ignored. Q_j is the predicted reward for action a_j.

$$Q_j = \frac{\sum_i Q_{ij}\overline{s_i}}{\sum_i \overline{s_i}} \tag{23}$$

The act of combining reward across features is an important one. There are many reasonable ways to do it, each with a different outcome. The choice of a weighted average is a deliberate design decision. It would also be possible to take the maximum, the sum, the unweighted average, or to select one of the rewards at random. Since features are assumed to independent, one way to view the feature-action reward estaimtes is as an ensemble model, a collection of simple (and simplistic) models that all cast their votes. How those votes are tallied from an ensemble is a bit of art. In practice, the weighted average ended up making the most sense here because it does a good job of only considering rewards associated with active features, and considering those rewards with the most active features most heavily.

Quirk 10: Intermittent rewards

Intermittent rewards were described in some detail in Section 1.8. FNC allows for rewards to be intermittent, that is, to be absent on any number of time steps. When there is no reward to be reported. FNC just skips the reward update on that time step. The exponentially decayed moving average doesn't incorporate a new value, not even a zero, and it doesn't change the estimate for r.

Quirk 11: Multiple rewards

It's natural to build a system where both internal and external rewards are at play. There may even be multiple sources of external reward. When any of these can be intermittent, this leads to a scenario where some aspects of the reward signal need to be updated on a given time step where others do not.

A really bizzare thing that FNC does is that it allows for multiple reward channels. (As far as I know it is unique in this respect.) This allows it to accommodate any number of reward sources, any one of which may be absent at any given time step.

$$\mathcal{R} = [r_0, r_1, r_2, \cdots, r_k, \cdots, r_l] \text{ where } r_k \in \{\mathbb{R}, \emptyset\} \tag{24}$$

FNC handles this by keeping track of l different feature-

action reward estimates, and only updating estimates on each time step for which it has a non-null reward.

$$Q_{ijk,t} = Q_{ijk,t-1} + \alpha(\overline{s_i, a_j})_t(r_{k,t} - Q_{ijk,t-1}) \quad (25)$$

When it comes time to predict the reward associated with each action for the next time step, it makes a prediction for each reward channel individually, with a weighted average as in the method above, and then it sums the predicted reward across all reward sources.

$$Q_j = \sum_k \left(\frac{\sum_i Q_{ijk}\overline{s_i}}{\sum_i \overline{s_i}} \right) \quad (26)$$

Not so quirky: Transition probability estimation

FNC's transition probability calculations are fairly vanilla. The only quirks it has are spillovers from its other quirks. In Markov decision processes, transition probabilities to an outcome s' are typically defined in terms of the most recent state and action.

$$p(s'|s, a) \quad (27)$$

The probability is based on the crusty old frequentist technique of counting all the times the transition happened and dividing it by all the times it had the opportunity to happen. Considering specfically the transition from feature s_i to outcome s'_k via action a_j.

$$p(s'_k|(s_i, a_j) = \frac{\text{count}(s_i, a_j, s'_k)_t}{\text{count}(s_i, a_j)_t} \quad (28)$$

Due to the fact that feature-action pairs and outcomes can have fractional values, FNC needs to modify this from using count() and use a sum instead. Also, to ensure that the fraction is well behaved, one is added to the denominator. This results in a somewhat lower probability estimate early in FNC's experience, but its influence fades as the number of occurrences grows.

$$p(s'_k|(s_i, a_j) = \frac{\sum_t (s_i, a_j, s'_k)_t}{\sum_t (s_i, a_j)_t + 1} \quad (29)$$

In FNC, transition probabilities are defined in terms of the condensed feature-action activity trace.

$$p(s'|(\overline{s, a})) \quad (30)$$

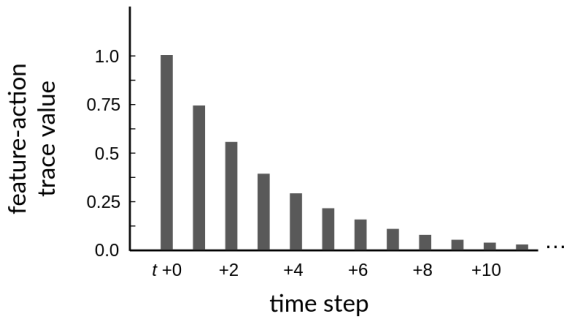
And the transition activity is given by the product of the activity trace and the outcome activity.

$$(\overline{s_i, a_j}, s'_k) = (\overline{s_i, a_j})s'_k \quad (31)$$

All together, this gives FNC's version of transition probabilities.

$$p(s'_k|(\overline{s_i, a_j}) = \frac{\sum_t (\overline{s_i, a_j}, s'_k)_t}{\sum_t (\overline{s_i, a_j})_t + 1} \quad (32)$$

Although the form of this is very similar to canonical frequentist probability estimates, FNC's idiosyncracies lead to a slightly different interpretation. Imagine that feature s_i is fully active when action a_j is taken at time t, so that the trace $(\overline{s_i}, a_j)_t = 1$. Thanks to the trace decay schedule induced by γ_r it will trail off over time, even if no other actions are taken. The value of the trace at $t + \tau$ will be γ_r^τ.



The sum of the trace over time will be the total of all the decayed components.

$$\sum_t (\overline{s_i, a_j})_t = (\overline{s_i}, a_j)\left(1 + \gamma_r + \gamma_r^2 + \gamma_r^3 + \cdots\right) \quad (33)$$

This form has its own name, geometric series, and even better, it has a closed form solution.

$$1 + \gamma_r + \gamma_r^2 + \gamma_r^3 + \cdots = \frac{1}{1 - \gamma_r} \quad (34)$$

The result of this is that even for a feature-action pair that is active on a single timestep, a single occurence, the resulting sum of the trace will not be 1, but will be $1/(1 - \gamma_r)$. The way this impacts the frequency calculation is that it inflates the count of the feature-action pair occurrence. If the outcome s_k occurs with an activity of 1 at time t, but is zero thereafter, then the transition $(\overline{s_i, a_j}, s'_k)$ will only be incremented by 1, even though the feature-action pair will be incremented by $1/(1 - \gamma_r)$. So even if the outcome occurred for a single timestep every time the feature-action pair of s_i and a_j occurred, it would still show up with a transition probability of (possibly much) less than one.

The only way that the sum of transition occurrences can keep pace with the sum of the feture-action occurrences is if the outcome s_k becomes fully active *and stays active*, at least until the feature-action trace has decayed to almost nothing. In that case the calculated transition probability will stay close to one.

All of this gives the method for predicting individual transition probabilities. It still treats every feature as independent. When in operation, each time step will bring FNC

a new combination of features, each with its own activity level. It has to aggregate these in come way to come up with an overall conditional probability estimate $p(s'_k|a_j)$.

FNC does this by multiplying each feature-action-outcome transition probability estimate by the condensed activity history $\overline{s_i}$ of its corresponding feature.

$$p(s'_k|a_j) = \max_i(\overline{s_i}\, p(s'_k|(\overline{\overline{s_i}, a_j})) \tag{35}$$

This allows FNC to generate a set of what if scenarios for each action. If a given action is taken, it tells the expected probability of each outcome feature. The $\max()$ operator works as an aggregator for this, because it doesn't get weighed down by all the features that aren't currently active or aren't particularly predictive of a given outcome. All it needs is one feature-action pair to be confident of a particular outcome for it to embrace it.

Quirk 12: Uncertainty

The last unusual thing that FNC does is to provide an uncertainty estimate on the transition probabilities. This estimate is approximate, but it gives a planner some basis for interpreting the transition probability estimates.

Imagine that the outcome FNC is trying to predict is the result of a coin flip where the coin might be unfair. After two flips, the coin has come up heads twice. Because it is two for two, FNC will predict that the next flip will be heads with a probability of one. The entirety of its (extremely limited) experience leads it to believe this. Of course, we know that even a fair coin will have its first two flips be the same half of the time. A more honest estimate should reflect the fact that after two flips we still don't know very much.

However, if after 1000 flips the coin has come up heads 900 times, it is reasonable to predict that it will come up heads nine times out of ten on average, and to be quite confident in that prediction. It may not be exactly right, but it's probably close.

The pattern here is that with more observations uncertainty decreases. Leaning on statistics helps make this more concrete. In the case of a biased coin where the outcome is always one of two values, it's natural to model the outcome using a Bernoulli probability distribution. In the more general case where an outcome feature can take on any value between zero and one, a beta distribution is a better fit. It's very flexible and can degenerate into a Bernoulli distribution, a uniform distribution, and a rich set of other shapes, all supported on the interval $[0, 1]$.

The good news for us is that we don't have to care about any of that. The central limit theorem lets us sidestep it. As long as we can assume that the outcome is (more or less) independent from one time to the next and distributed in (pretty much) the same way each time, the central limit theorem tells us that the mean of a sample will follow a normal distribution, and that the distribution will have a standard error of $\sigma^2/\sqrt{n}$, where σ^2 is the variance of the sample and n is the number of observations in it. For a distribution supported on $[0, 1]$, σ can ever be more than .5. The range associated with a 95% confidence interval is $\pm 1.96\sigma/\sqrt{n}$ implying that $\pm 1/\sqrt{n}$ is a slightly conservative bound for the mean, a cautious estimate for the uncertainty in FNC's predictions. The the case of FNC, n is the number of times a feature action pair has occurred. (Note that, as was just called out in the previous section, FNC overcounts this by $1/(1-\gamma)$ due to how it handles trace decay.

Another shortcut that FNC takes is reporting $1/n$ instead of $1/\sqrt{n}$. This is pure laziness, just to save FNC some computation. It results in an uncertainty estimate that smaller, reflecting a somewhat lower confidence level. Also, since the number of occurrences of a feature-action pair can be arbitrarily small, $1/n$ can be arbitrarily big. To keep it properly limited to be no greater than one, an extra observation is added to n. The actual uncertainty estimate FNC returns after all these modifications, δ, is given by

$$\delta_{ij} = \frac{1}{1 + \sum_t(\overline{\overline{s_i}, a_j})} \tag{36}$$

A planner that consumes the uncertainty has the option fix the uncertainty estimates, to subtract the extra observation, to take the square root and to apply a $1/(1-\gamma_r)$ correction to get a more accurate confidence interval. Or it can also just accept $1/(1+n)$ as a rough approximation built on a scaffold of assumptions, some generous, some conservative.

As with the transition probabilities, it is also necessary to aggregate the conditional uncertainties, the uncertainties across all the currently active features. And as with the transition probabilities, the $\max()$ operator is the aggregator of choice. Specifically the uncertainty associated with a given action δ_j is the maximum of all the features-action uncertainties δ_{ij}, each multiplied by the recent feature activity history $\overline{s_i}$.

$$\delta_j = \max_i(\overline{s_i}\, \delta_{ij}) \tag{37}$$

There are several other defensible ways to aggregate uncertainties, but the choice of the maximum is deliberate. It ensures that whichever aspect of the current situation is the most uncertain, whichever feature has been paired with this action the fewest times, gets to dominate.

3. Limitations and Extensions

Under construction.

4. Agent architecture and planners

Under construction.

5. Bookkeeping

5.1. Naive Cartographer code

The Naive Cartographer Python package is hosted in a Codeberg repository.

The README of the repo has instructions for installation and examples of how to integrate it into your project.

5.2. Versions

The **latest version** of this document and all the files needed to render it are in **this Codeberg repository**.

I don't expect this doc to ever be done. I'm always learning new things, or thinking of a better way to explain something, or I do a new piece of work I can't help myself from including. And there's always one more bug. Since it's a git repository, you are free to browse past commits to watch the evolution, but I'll try to keep a running record of important updates here.

- **March 17, 2024**. Early draft with a full description of how Naive Cartographer works.

5.3. Citations

If you end up using Naive Cartographer in your work, give it a shout out. Here's an APA example you can copy and paste. (You may have to fiddle with the dates.)

Rohrer, B. (2024). Naive Cartographer: A Markov Decision Process Learner [White Paper]. Retrieved March 18, 2024, from https://brandonrohrer.com/cartographer

5.4. Licensing

5.5. Contact Me

I'm at brohrer@gmail.com. You're welcome to email me at any time for any reason. I don't guarantee I'll respond, but I try to. If you're so inclined, drop me a note. I love to hear about how Naive Cartographer is being used.