PREV
Chapter 5. Automatically clustering data

NEXT
Part 3. The neural network paradigm

Chapter 6. Hidden Markov models

This chapter covers

- Defining interpretive models
- Using Markov chains to model data
- Inferring hidden state using a hidden Markov model

If a rocket blows up, someone's probably going to get fired, so rocket scientists and engineers must be able to make confident decisions about all components and configurations. They do so by physical simulations and mathematical deduction from first principles. You, too, have solved science problems with pure logical thinking. Consider Boyle's law: pressure and volume of a gas are inversely related under a fixed temperature. You can make insightful inferences from these simple laws that have been discovered about the world. Recently, machine learning has started to play the role of an important sidekick to deductive reasoning.

Rocket science and *machine learning* aren't phrases that usually appear together. But nowadays, modeling real-world sensor readings by using intelligent data-driven algorithms is more approachable in the aerospace industry. Also, the use of machine-learning techniques is flourishing in the healthcare and automotive industries. But why?

This influx can be partly attributed to better understanding of *interpretable* models, which are machine-learning models in which the learned parameters have clear interpretations. If a rocket blows up, for example, an interpretable model might help trace the root cause.

Exercise 6.1

What makes a model interpretable may be slightly subjective. What's your criteria for an interpretable model?

ANSWER

We like to refer to mathematical proofs as the de facto explanation technique. If one were to convince another about the truth of a mathematical theorem, then a proof that irrefutably traces the steps of reasoning is sufficient.

This chapter is about exposing the hidden explanations behind observations. Consider a puppet master pulling strings to make a puppet appear alive. Analyzing only the motions of the puppet might lead to overly complicated conclusions about how it's possible for an inanimate object to move. After you notice the attached strings, you'll realize that a puppet master is the best explanation for the lifelike motions.

On that note, this chapter introduces *hidden Markov models* (HMMs), which reveal intuitive properties about the problem under study. The HMM is the "puppet master," which explains the observations. You model observations by using Markov chains, which are described in section 6.2.

Before going into detail about Markov chains and HMMs, let's consider alternative models. In the next section, you'll see models that may not be interpretable.

6.1. EXAMPLE OF A NOT-SO-INTERPRETABLE MODEL

One classic example of a black-box machine-learning algorithm that's difficult to interpret is image classification. In an image-classification task, the goal is to assign a label to each input image. More simply, image classification is often posed as a multiple-choice question: which one of the listed categories best describes the image? Machine-learning practitioners have made tremendous advancements in solving this problem, to the point where today's best image classifiers match human-level performance on certain datasets.

You'll learn how to solve the problem of classifying images in chapter 9—convolutional neural networks (CNNs), which are a class of machine-learning models that end up learning a lot of parameters. But those parameters are the problem with CNNs: what

do each of the thousands, if not millions, of parameters mean? It's difficult to ask an image classifier why it made the decision that it did. All we have available are the learned parameters, which may not easily explain the reasoning behind the classification.

Machine learning sometimes gets the notoriety of being a black-box tool that solves a specific problem without revealing how it arrives at its conclusion. The purpose of this chapter is to unveil an area of machine learning with an interpretable model. Specifically, you'll learn about the HMM and use TensorFlow to implement it.

6.2. MARKOV MODEL

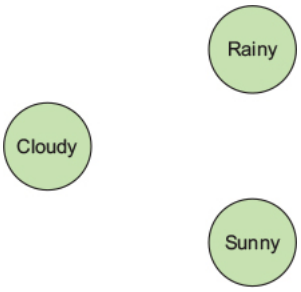
Andrey Markov was a Russian mathematician who studied the ways systems change over time in the presence of randomness. Imagine gas particles bouncing around in the air. Tracking the position of each particle by Newtonian physics can get way too complicated, so introducing randomness helps simplify the physical model a little.

Markov realized that what helps simplify a random system even further is considering only a limited area around the gas particle to model it. For example, maybe a gas particle in Europe has barely any effect on a particle in the United States. So why not ignore it? The mathematics is simplified when you look only at a nearby neighborhood instead of the entire system. This notion is now referred to as the *Markov property*.

Consider modeling the weather. A meteorologist evaluates various conditions with thermometers, barometers, and anemometers to help predict the weather. They draw on brilliant insight and years of experience to do their job.

Let's use the Markov property to help us get started with a simple model. First, you identify the possible situations, or *states*, that you care to study. Figure 6.1 shows three weather states as nodes in a graph: Cloudy, Rainy, and Sunny.

Figure 6.1. Weather conditions (states) represented as nodes in a graph



Now that you have the states, you want to also define how one state transforms into another. Modeling weather as a deterministic system is difficult. It's not an obvious conclusion that if it's sunny today, it'll certainly be sunny again tomorrow. Instead, you can introduce randomness and say that if it's sunny today, there's a 90% chance it'll be sunny again tomorrow, and a 10% chance it'll be cloudy. The Markov property comes into play when you use only today's weather condition to predict tomorrow's (instead of using all previous history).

Exercise 6.2

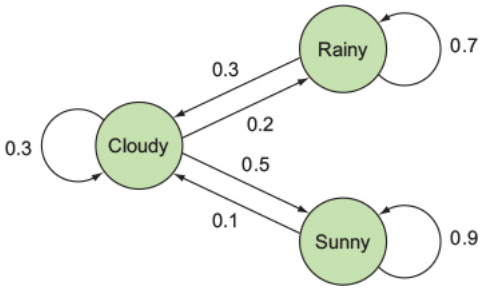
A robot that decides which action to perform based on only its current state is said to follow the Markov property. What are the advantages and disadvantages of such a decision-making process?

ANSWER

The Markov property is computationally easy to work with. But these models aren't able to generalize to situations that require accumulating a history of knowledge. Examples of these are models in which a trend over time is important, or in which knowledge of more than one past state gives a better idea of what to expect next.

Figure 6.2 demonstrates the transitions as directed edges drawn between nodes, with the arrow pointing toward the next future state. Each edge has a weight representing the probability (for example, there's a 30% chance that if today is rainy, tomorrow will be cloudy). The lack of an edge between two nodes is an elegant way of showing that the probability of that transformation is near zero. The transition probabilities can be learned from historical data, but for now, let's assume they're given to us.

Figure 6.2. Transition probabilities between weather conditions are represented as directed edges.

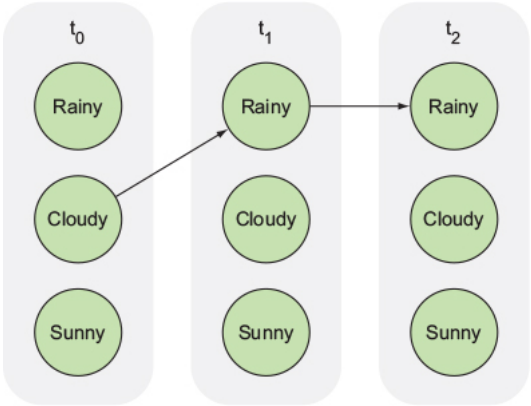


If you have three states, you can represent the transitions as a 3 × 3 matrix. Each element of the matrix (at row *i* and column *j*) corresponds to the probability associated with the edge from node *i* to node *j*. In general, if you have *N* states, the *transition matrix* will be *N* × *N* in size (see figure 6.4 for an example).

We call this system a *Markov model*. Over time, a state changes using the transition probabilities defined in figure 6.2. In our example, Sunny has a 90% chance of Sunny again tomorrow, so we show an edge of probability 0.9, looping back to itself. There's a 10% chance of a sunny day being followed by a cloudy day, shown in the diagram as the edge 0.1, pointing from Sunny to Cloudy.

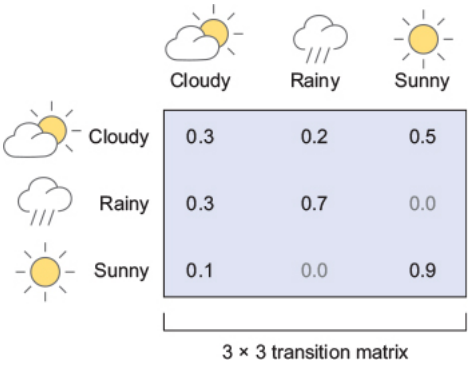
Figure 6.3 is another way to visualize how the states change, given the transition probabilities. It's often called a *trellis diagram*, which turns out to be an essential tool, as you'll see later when we implement the TensorFlow algorithms.

Figure 6.3. A trellis representation of the Markov system changing states over time



You've seen in previous chapters how TensorFlow code builds a graph to represent computation. It might be tempting to treat each node in a Markov model as a node in TensorFlow. But even though figures 6.2 and 6.3 nicely illustrate state transitions, there's a more efficient way to implement them in code, as shown in figure 6.4.

Figure 6.4. A transition matrix conveys the probabilities of a state from the left (rows) transitioning to a state at the top (columns).



Remember, nodes in a TensorFlow graph are tensors, so you can represent a transition matrix (let's call it *T*) as a node in TensorFlow. Then, you can apply mathematical operations on the TensorFlow node to achieve interesting results.

For example, suppose you prefer sunny days over rainy ones, so you have a score associated with each day. You represent your scores for each state in a 3×1 matrix called *s*. Then, multiplying the two matrices in TensorFlow using `tf.matmul(T*s)` gives the expected preference of transitioning from each state.

Representing a scenario in a Markov model allows you to greatly simplify how you view the world. But it's frequently difficult to measure the state of the world directly. Often, you have to use evidence from multiple observations to figure out the hidden meaning. And that's what the next section aims to solve!

6.3. HIDDEN MARKOV MODEL

The Markov model defined in the previous section is convenient when all the states are observable, but that's not always the case. Consider having access to only temperature readings of a town. Temperature isn't weather, but it's related to it. How then can you infer the weather from this indirect set of measurements?

Rainy weather most likely causes a lower temperature reading, whereas a sunny day most likely causes a higher temperature reading. With temperature knowledge and

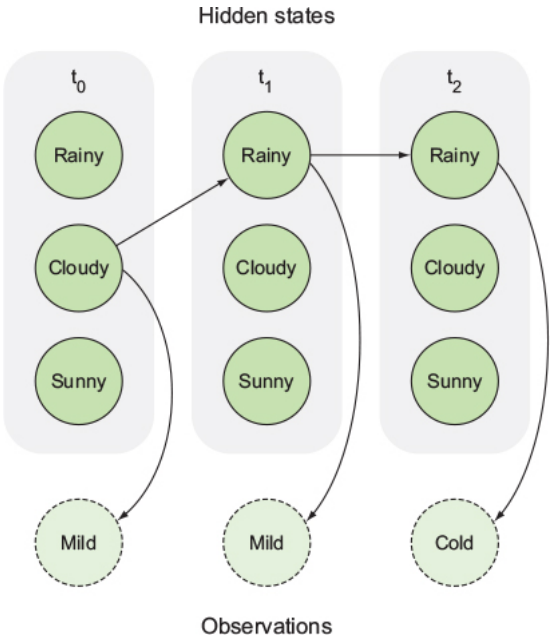
transition probabilities alone, you can still make intelligent inferences about the most likely weather. Problems like this are common in the real world. A state might leave traces of hints behind, and those hints are all you have available to you.

Models like these are HMMs because the true states of the world (such as whether it's raining or sunny) aren't directly observable. These hidden states follow a Markov model, and each state emits a measurable observation with a certain likelihood. For example, the hidden state of Sunny might emit high temperature readings, but occasionally also low readings for one reason or another.

In an HMM, you have to define the emission probability, which is usually represented as a matrix called the *emission matrix*. The number of rows in the matrix is the number of states (Sunny, Cloudy, Rainy), and the number of columns is the number of observation types (Hot, Mild, Cold). Each element of the matrix is the probability associated with the emission.

The canonical way of visualizing an HMM is by appending the trellis with observations, as shown in figure 6.5.

Figure 6.5. A hidden Markov model trellis showing how weather conditions might produce temperature readings



So that's *almost* it. The HMM is a description of transition probabilities, emission probabilities, and one more thing: *initial probabilities*. The initial probability is the probability of each state happening with no prior knowledge. If you're modeling the weather in Los Angeles, perhaps the initial probability of Sunny would be much greater. Or let's say you're modeling the weather in Seattle; well, you know you can set the initial probability of Rainy to something higher.

An HMM lets you understand a sequence of observations. In this weather-modeling scenario, one question you may ask is, what's the probability of observing a certain sequence of temperature readings? We'll answer this question by using the *forward algorithm*.

6.4. FORWARD ALGORITHM

The *forward algorithm* computes the probability of an observation. Many permutations may cause a particular observation, so enumerating all possibilities the naïve way will take an exponentially long time to compute.

Instead, you can solve the problem by using *dynamic programming*, which is a strategy of breaking a complex problem into simple little ones and using a lookup table to cache the results. In your code, you'll save the lookup table as a NumPy array and feed it to a TensorFlow op to keep updating it.

As shown in the following listing, create an `HMM` class to capture the hidden Markov model parameters, which include the initial probability vector, transition probability matrix, and emission probability matrix.

Listing 6.1. Defining the `HMM` class

```
import numpy as np                                              1
import tensorflow as tf                                         1


class HMM(object):
    def __init__(self, initial_prob, trans_prob, obs_prob):
        self.N = np.size(initial_prob)                          2
        self.initial_prob = initial_prob                        2
        self.trans_prob = trans_prob                            2
        self.emission = tf.constant(obs_prob)                   2

        assert self.initial_prob.shape == (self.N, 1)           3
        assert self.trans_prob.shape == (self.N, self.N)        3
        assert obs_prob.shape[0] == self.N                      3

        self.obs_idx = tf.placeholder(tf.int32)                 4
        self.fwd = tf.placeholder(tf.float64)                   4
```

- ***1* Imports the required libraries**
- ***2* Stores the parameters as method variables**
- ***3* Double-checks that the shapes of all the matrices make sense**
- ***4* Defines the placeholders used for the forward algorithm**

Next, you'll define a quick helper function to access a row from the emission matrix. The code in the following listing is a helper function to efficiently obtain data from an arbitrary matrix. The `slice` function extracts a fraction of the original tensor. This function requires as input the relevant tensor, the starting location specified by a tensor, and the size of the slice specified by a tensor.

Listing 6.2. Creating a helper function to access emission probability of an observation

```
    def get_emission(self, obs_idx):
        slice_location = [0, obs_idx]                           1
        num_rows = tf.shape(self.emission)[0]
        slice_shape = [num_rows, 1]                             2
        return tf.slice(self.emission, slice_location, slice_shape)  3
```

- ***1* The location of where to slice the emission matrix**

- ***2* The shape of the slice**
- ***3* Performs the slicing operation**

You need to define two TensorFlow ops. The first one, in the following listing, will be run only once to initialize the forward algorithm's cache.

Listing 6.3. Initializing the cacheee

```
    def forward_init_op(self):
        obs_prob = self.get_emission(self.obs_idx)
        fwd = tf.multiply(self.initial_prob, obs_prob)
        return fwd
```

And the next op will update the cache at each observation, as shown in listing 6.4. Running this code is often called *executing a forward step*. Although it looks like this `forward_op` function takes no input, it depends on placeholder variables that need to be fed to the session. Specifically, `self.fwd` and `self.obs_idx` are the inputs to this function.

Listing 6.4. Updating the cache

```
    def forward_op(self):
        transitions = tf.matmul(self.fwd,
     tf.transpose(self.get_emission(self.obs_idx)))
        weighted_transitions = transitions * self.trans_prob
        fwd = tf.reduce_sum(weighted_transitions, 0)
        return tf.reshape(fwd, tf.shape(self.fwd))
```

Outside the `HMM` class, let's define a function to run the forward algorithm, as shown in the following listing. The forward algorithm runs the forward step for each observation. In the end, it finally outputs a probability of observations.

Listing 6.5. Defining the forward algorithm given an HMM

```
def forward_algorithm(sess, hmm, observations):
    fwd = sess.run(hmm.forward_init_op(), feed_dict={hmm.obs_idx:
     observations[0]})
    for t in range(1, len(observations)):
        fwd = sess.run(hmm.forward_op(), feed_dict={hmm.obs_idx:
     observations[t], hmm.fwd: fwd})
    prob = sess.run(tf.reduce_sum(fwd))
    return prob
```

In the main function, let's set up the `HMM` class by feeding it the initial probability vector, transition probability matrix, and emission probability matrix. For consistency, the example in listing 6.6 is lifted directly from the Wikipedia article on HMMs: http://mng.bz/8ztL (http://mng.bz/8ztL), as shown in figure 6.6.

In general, the three concepts are defined as follows:

- *Initial probability vector*—Starting probability of the states
- *Transition probability matrix*—Probabilities associated with landing on the next states, given the current state

- *Emission probability matrix*—Likelihood of an observed state implying the state you're interested in has occurred

Figure 6.6. Screenshot of HMM example scenario from Wikipedia

```
states = ('Rainy', 'Sunny')

observations = ('walk', 'shop', 'clean')

start_probability = {'Rainy': 0.6, 'Sunny': 0.4}

transition_probability = {
   'Rainy' : {'Rainy': 0.7, 'Sunny': 0.3},
   'Sunny' : {'Rainy': 0.4, 'Sunny': 0.6},
}

emission_probability = {
   'Rainy' : {'walk': 0.1, 'shop': 0.4, 'clean': 0.5},
   'Sunny' : {'walk': 0.6, 'shop': 0.3, 'clean': 0.1},
}
```

Given these matrices, you'll call the forward algorithm that you just defined.

Listing 6.6. Defining the HMM and calling the forward algorithm

```
if __name__ == '__main__':
    initial_prob = np.array([[0.6],
                             [0.4]])

    trans_prob = np.array([[0.7, 0.3],
                           [0.4, 0.6]])

    obs_prob = np.array([[0.1, 0.4, 0.5],
                         [0.6, 0.3, 0.1]])

    hmm = HMM(initial_prob=initial_prob, trans_prob=trans_prob,
     obs_prob=obs_prob)

    observations = [0, 1, 1, 2, 1]
    with tf.Session() as sess:
        prob = forward_algorithm(sess, hmm, observations)
        print('Probability of observing {} is {}'.format(observations, prob))
```

When you run listing 6.6, the algorithm outputs the following:

```
Probability of observing [0, 1, 1, 2, 1] is 0.0045403
```

6.5. VITERBI DECODING

The *Viterbi decoding algorithm* finds the most likely sequence of hidden states, given a sequence of observations. It requires a caching scheme similar to the forward algorithm. You'll name the cache `viterbi`. In the HMM constructor, append the line shown in the following listing.

Listing 6.7. Adding the Viterbi cache as a member variable

```
def __init__(self, initial_prob, trans_prob, obs_prob):
  ...
  ...
  ...
  self.viterbi = tf.placeholder(tf.float64)
```

In the next listing, you'll define a TensorFlow op to update the `viterbi` cache. This will be a method in the `HMM` class.

Listing 6.8. Defining an op to update the forward cache

```
def decode_op(self):
        transitions = tf.matmul(self.viterbi,
     tf.transpose(self.get_emission(self.obs_idx)))
        weighted_transitions = transitions * self.trans_prob
        viterbi = tf.reduce_max(weighted_transitions, 0)
        return tf.reshape(viterbi, tf.shape(self.viterbi))
```

You'll also need an op to update the back pointers.

Listing 6.9. Defining an op to update the back pointers

```
def backpt_op(self):
    back_transitions = tf.matmul(self.viterbi, np.ones((1, self.N)))
    weighted_back_transitions = back_transitions * self.trans_prob
    return tf.argmax(weighted_back_transitions, 0)
```

Lastly, in the following listing, define the Viterbi decoding function outside the HMM.

Listing 6.10. Defining the Viterbi decoding algorithm

```
def viterbi_decode(sess, hmm, observations):
    viterbi = sess.run(hmm.forward_init_op(), feed_dict={hmm.obs:
     observations[0]})
    backpts = np.ones((hmm.N, len(observations)), 'int32') * -1
    for t in range(1, len(observations)):
        viterbi, backpt = sess.run([hmm.decode_op(), hmm.backpt_op()],
                                   feed_dict={hmm.obs: observations[t],
                                              hmm.viterbi: viterbi})
        backpts[:, t] = backpt
    tokens = [viterbi[:, -1].argmax()]
    for i in range(len(observations) - 1, 0, -1):
        tokens.append(backpts[tokens[-1], i])
    return tokens[::-1]
```

You can run the code in the next listing in the main function to evaluate the Viterbi decoding of an observation.

Listing 6.11. Running the Viterbi decode

```
seq = viterbi_decode(sess, hmm, observations)
print('Most likely hidden states are {}'.format(seq))
```

6.6. USES OF HIDDEN MARKOV MODELS

Now that you've implemented the forward algorithm and Viterbi algorithm, let's take a look at interesting uses for your newfound power.

6.6.1. Modeling a video

Imagine being able to recognize a person based solely (no pun intended) on how they walk. Identifying people based on their gait is a pretty cool idea, but first you need a model to recognize the gait. Consider an HMM in which the sequence of hidden states

for a gait are (1) rest position, (2) right foot forward, (3) rest position, (4) left foot forward, and finally (5) rest position. The observed states are silhouettes of a person walking/jogging/running taken from a video clip (a dataset of such examples is available at http://mng.bz/Tqfx (http://mng.bz/Tqfx)).

6.6.2. Modeling DNA

DNA is a sequence of nucleotides, and we're gradually learning more about its structure. One clever way to understand a long DNA string is by modeling the regions, if we know some probability about the order in which they appear. Just as cloudy days are common after a rainy day, maybe a certain region on the DNA sequence (*start codon*) is more common before another region (*stop codon*).

6.6.3. Modeling an image

In handwriting recognition, we aim to retrieve the plaintext from an image of handwritten words. One approach is to resolve characters one at a time and then concatenate the results. You can use the insight that characters are written in sequences—words—to build an HMM. Knowing the previous character could probably help you rule out possibilities of the next character. The hidden states are the plaintext, and the observations are cropped images containing individual characters.

6.7. APPLICATION OF HIDDEN MARKOV MODELS

Hidden Markov models work best when you have an idea about what the hidden states are and how they change over time. Luckily, in the field of natural language processing, tagging a sentence's parts of speech can be solved using HMMs:

- A sequence of words in a sentence corresponds to the observations of the HMM. For example, the sentence "Open the pod bay doors, HAL" has six observed words.
- The hidden states are the parts of speech, such as verb, noun, adjective, and so on. The observed word *open* in the previous example should correspond to the hidden state *verb*.
- The transition probabilities can be designed by the programmer or obtained through data. These probabilities represent the rules of the parts of speech. For example, the probability of two verbs occurring one after another should be low. By setting up a transition probability, you avoid having the algorithm brute-forcing all possibilities.
- The emitting probabilities of each word can be obtained from data. A traditional part-of-speech tagging dataset is called Moby; you can find it at www.gutenberg.org/ebooks/3203 (http://www.gutenberg.org/ebooks/3203).

Note

You now have what it takes to design your own experiments using hidden Markov models! It's a powerful tool, and we urge you to try it on your own data. Predefine some transitions and emissions, and see if you can recover hidden states. Hopefully, this chapter can help get you started.

6.8. SUMMARY

- A complicated, entangled system can be simplified using a Markov model.
- The hidden Markov model is particularly useful in real-world applications because most observations are measurements of hidden states.
- The forward and Viterbi algorithms are among the most common algorithms used on HMMs.

PREV
Chapter 5. Automatically clustering data

NEXT
Part 3. The neural network paradigm