Sanjay Dorairaj · Follow
Mar 20, 2018 · 18 min read

Hidden Markov Models Simplified

Sanjay Dorairaj

Adapted from HMM/NLP lectures by James Kunz et al @UC Berkeley

Overview

Hidden Markov Models (HMMs) are a class of probabilistic graphical model that allow us to predict a sequence of unknown (hidden) variables from a set of observed variables. A simple example of an HMM is predicting the weather (hidden variable) based on the type of clothes that someone wears (observed). An HMM can be viewed as a Bayes Net unrolled through time with observations made at a sequence of time steps being used to predict the best sequence of hidden states.

The below diagram from Wikipedia shows an HMM and its transitions. The scenario is a room that contains urns **X1, X2 and X3**, each of which contains a known mix of balls, each ball labeled **y1, y2, y3 and y4**. A sequence of four balls is randomly drawn. In this particular case, **the user observes a sequence of balls y1,y2,y3 and y4 and is attempting to discern the hidden state which is the right sequence of three urns that these four balls were pulled from.**

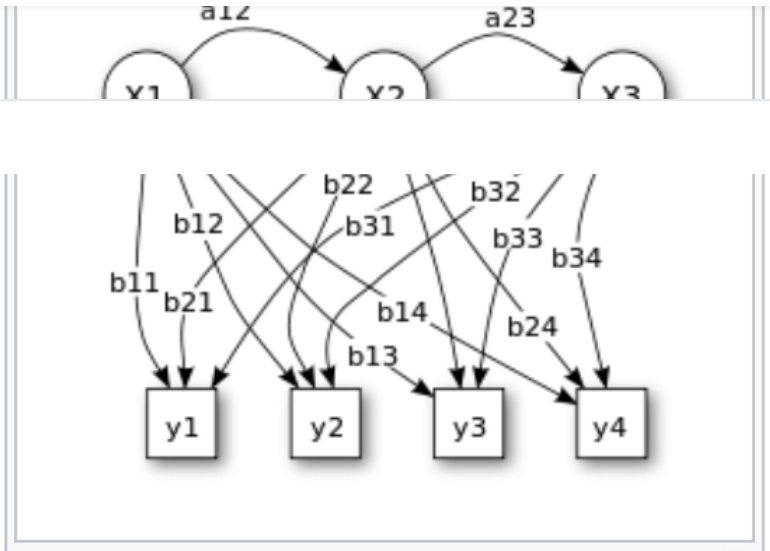


Figure 1. Probabilistic parameters of a hidden Markov model (example)
X — states
y — possible observations
a — state transition probabilities
b — output probabilities

Figure 1: HMM hidden and observed states

Source:
https://en.wikipedia.org/wiki/Hidden_Markov_model#/media/File:HiddenMarkovModel.svg

In this post, we focus on the use of Hidden Markov Models for Parts of Speech (POS) diagram and walk through the intuition and the code for POS tagging using HMMs.

Complete source code for this post is available in Github at
https://github.com/dorairajsanjay/hmm_tutorial

Why Hidden, Markov Model?

The reason it is called a Hidden Markov Model is because we are constructing an inference model based on the assumptions of a Markov process. The Markov process assumption is simply that the "future is independent of the past given the present". In other words, assuming we know our present state, we do not need any other historical information to predict the future state.

To make this point clear, let us consider the scenario below where the weather, the hidden variable, can be hot, mild or cold and the observed variables are the type of clothing worn. The arrows represent transitions from a hidden state to

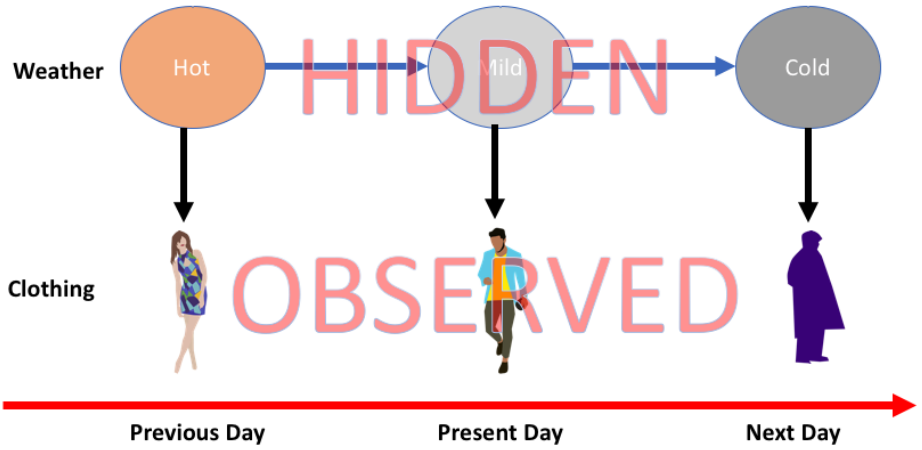


Figure 2: HMM State Transitions

Intuition behind HMMs

HMMs are probabilistic models. They allow us to compute the joint probability of a set of hidden states given a set of observed states. The hidden states are also referred to as latent states. Once we know the joint probability of a sequence of hidden states, we determine the best possible sequence i.e. the sequence with the highest probability and choose that sequence as the best sequence of hidden states.

The ratio of hidden states to observed states is not necessarily 1 is to 1 as is evidenced by **Figure 1** above. The key idea is that one or more observations allow us to make an inference about a sequence of hidden states.

In order to compute the joint probability of a sequence of hidden states, we need to assemble three types of information.

Generally, the term “states” are used to refer to the hidden states and “observations” are used to refer to the observed states.

1. **Transition data** — the probability of transitioning to a new state conditioned on a present state.
2. **Emission data** — the probability of transitioning to an observed state conditioned on a hidden state.
3. **Initial state information** — the initial probability of transitioning to a hidden state. This can also be looked at as the prior probability.

The above information can be computed directly from our training data. For

The example tables show a set of possible values that could be derived for the weather/clothing scenario.

Priors

Hot	0.6
Mild	0.3
Cold	0.1

Transitions

	Hot	**Mild**	**Cold**
Hot	0.6	0.3	0.1
Mild	0.4	0.3	0.2
Cold	0.1	0.4	0.5

Emissions

	Hot	**Mild**	**Cold**
Casual Wear	0.8	0.19	0.01
Semi Casual Wear	0.5	0.4	0.1
Winter apparel	0.01	0.2	0.79

Figure 3: HMM State Transitions — Weather Example

Once this information is known, then the joint probability of the sequence, by the conditional probability chain rule and by Markov assumption, can be shown to be proportional to **P(Y)** below

The probability of observing a sequence

$$Y = y(0), y(1), \ldots, y(L-1)$$

of length L is given by

$$P(Y) = \sum_{X} P(Y \mid X) P(X),$$

where the sum runs over all possible hidden-node sequences

$$X = x(0), x(1), \ldots, x(L-1).$$

Figure 4: HMM — Basic Math (HMM lectures)

Note that as the number of observed states and hidden states gets large the computation gets more computationally intractable. If there are k possible values for each hidden sequence and we have a sequence length of n, there there are **n^k** total possible sequences that must be all scored and ranked in order to determine a winning candidate.

Variations of the Hidden Markov Model — HMM EM

The probability distributions of hidden states is not always known. In this case, we use Expectation Maximization (EM) models in order to determine hidden state distributions. A popular algorithm is the Baum-Welch algorithm (https://en.wikipedia.org/wiki/Baum%E2%80%93Welch_algorithm)

As seen in the above sections on HMM, the computations become intractable as the sequence length and possible values of hidden states become large. It has been found that the problem of scoring an HMM sequence can be solved efficiently using dynamic programming, which is nothing but **cached recursion.**

Shown below is an image of the recursive computation of a fibonnaci series

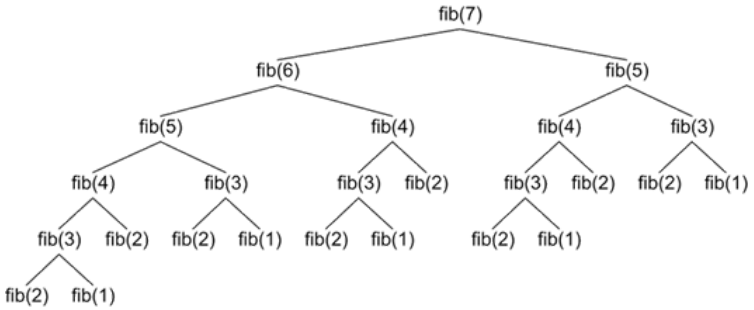


Figure 5: Fibonnacci Series — Tree

One of the things that becomes obvious when looking at this picture is that several results (fib(x) values) are reused in the computation. By caching these results, we can greatly speed up our operations

Fibonacci Computation without Dynamic Programming

```
import time

# non-cached recursion
def fib(x):

    if x == 0:
        return 0
    elif x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)

startTime = time.time()


print("%-14s:%d" % ("Result",fib(32)))
print("%-14s:%.4f seconds" % ("Elapsed time",time.time() - startTime))


Result        :2178309
Elapsed time  :2.0062 seconds
```

Fibonacci Computation with Dynamic Programming

```
fib_cache = {}
```

```
    result = None
    if x == 0:
        result = 0
    elif x == 1:
        result = 1
    else:
        result = fib(x-1) + fib(x-2)

    if fib_cache.get(x) == None:
        fib_cache[x] = result

    return result

startTime = time.time()

print("%-14s:%d" % ("Result",fib(32)))
print("%-14s:%.4f seconds" % ("Elapsed time",time.time() - startTime))


Result        :2178309
Elapsed time  :0.0007 seconds
```

Notice the significant improvement in performance when we move to dynamic programming or cached recursion. We use this same idea when trying to score HMM sequences as well using an algorithm called the Forward-Backward algorithm which we will talk about later

Manipulating Summations

Here we look at an idea that will be leveraged in the forward backward algorithm. This is idea that double summations of terms can be rearrangeed as a product of each of the individual summation.

The example below explains this idea further.

$$1 \times 10 + 2 \times 10 + 3 \times 10 + 1 \times 20 + 2 \times 20 + 3 \times 20 + 1 \times 30 + 2 \times 30 + 3 \times 30$$

$$= \Sigma_{x=1,2,3} \Sigma_{y=10,20,30} x*y$$

$$= (\Sigma_{x=1,2,3} x) (\Sigma_{y=10,20,30} y)$$

Figure 6: HMM — Manipulation Summations

The code below demonstrates this equivalency relationship

```
# double summations

x = [1,2,3]
y = [10,20,30]

total = 0
for i in x:
```

```
360

# product of individual summations

total = 0
total1 = 0
for i in x:
    total1 = total1 + i

total2 = 0
for j in y:
    total2 = total2 + j

total = total1*total2

print total

360
```

Manipulating Maxes

Similar to manipulating double summations, the max of a double maxation can be viewed as the product of each of the individual maxations.

$$\max(1 \times 10, 2 \times 10, 3 \times 10, 1 \times 20, 2 \times 20, 3 \times 20, 1 \times 30, 2 \times 30, 3 \times 30)$$

$$= \max_{x=1,2,3} \max_{y=10,20,30} x\, y$$

$$= (\max_{x=1,2,3} x\, (\max_{y=10,20,30} y))$$

Figure — 7: HMM — Manipulation Maxes

```
# double maxation

x = [1,2,3]
y = [10,20,30]

# form 1
form1 = []
for i in x:
    for j in y:

        form1.append(i*j)

print "form1:",form1
print "max(form1):",max(form1)

form1: [10, 20, 30, 20, 40, 60, 30, 60, 90]
max(form1): 90

# product of individual maxations

form2_1 = []
```

```
print "\nform2_1",form2_1
print "form2_2",form2_2
print "max(form2_1) * max(form2_2)", max(form2_1) * max(form2_2)


form2_1 [1, 2, 3]
form2_2 [10, 20, 30]
max(form2_1) * max(form2_2) 90
```

Training an HMM

In this section, we will consider the toy example below and use the information from that example to train our simple HMM model

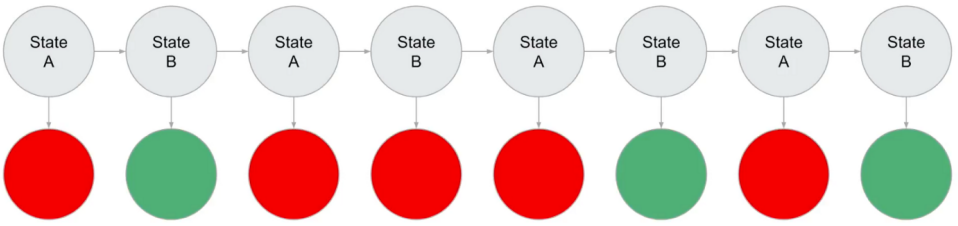


Figure — 8: HMM — Toy Example — Graph

State transition and emission probabilities

Priors

State A	P(A) = 1
State B	P(B) = 0

Transitions

	State A	State B
State A	P(A/A) = 0	P(B/A) = 1
State B	P(A/B) = 1	P(B/B) = 0

Emissions

	State A	State B
Red	P(A/Red) = 0	P(B/Red) = 1
Green	P(A/Green) = 1	P(B/Green) = 0

Figure — 9: HMM — Toy Example — Transition Tables

Scoring a known sequence given observed text

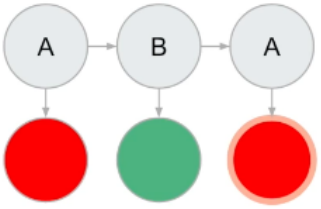


Figure — 10: HMM — Toy Example — Graph

The score for this sequence can be computed as

$$P(y_0, y_1, y_2, x_0, x_1, x_2) = P(y_0) \times P(x_0 \mid y_0) \times P(y_1 \mid y_0) \times P(x_1 \mid y_1) \times P(y_2 \mid y_1) \times P(x_2 \mid y_2)$$

Figure — 11: HMM — Toy Example — Scoring Known Sequence

The joint probability for our unknown sequence is therefore

P(A,B,A,Red,Green,Red) = [P(y_0=A) *P(x_0=Red/y_0=A)]*
[P(y_1=B|y_0=A|)

P(x_1=Green/y_1=B)] *[P(y_2=A|y_1=B)* P(x_2=Red/y_2=A)]

=(1∗1)∗(1∗0.75)∗(1∗1)(1)(1)=(1∗1)∗(1∗0.75)∗(1∗1)

=0.75(2)(2)=0.75

Scoring a set of sequences given some observations

Assuming that we need to determine the parts of speech tags (hidden state) given some sentence (the observed values), we will need to first score every possible sequence of hidden states and then pick the best sequence to determine the parts of speech for this sentence.

We will score this using the below steps

1. Generate the initial, transition and emission probability distribution from the sample data.
2. Generate a list of all unknown sequence
3. Score all unknown sequences and select the best sequence

Generate list of unknown sequences

```
states)^(sequence length)

def generate_sequence(states,sequence_length):

    all_sequences = []
    nodes = []

    depth = sequence_length

    def gen_seq_recur(states,nodes,depth):

        if depth == 0:
            #print nodes
            all_sequences.append(nodes)
        else:
            for state in states:
                temp_nodes = list(nodes)
                temp_nodes.append(state)
                gen_seq_recur(states,temp_nodes,depth-1)

    gen_seq_recur(states,[],depth)

    return all_sequences
```

Score all possible sequences

```
def
score_sequences(sequences,initial_probs,transition_probs,emission_prob
s,obs):

    best_score = -1
    best_sequence = None

    sequence_scores = []


    for seq in sequences:


        total_score = 1
        total_score_breakdown = []
        first = True
        for i in range(len(seq)):
            state_score = 1

            # compute transitition probability score
            if first == True:
                state_score *= initial_probs[seq[i]]


                # reset first flag
                first = False
            else:
                state_score *= transition_probs[seq[i] + "|" + seq[i-
1]]


            # add to emission probability score
            state_score *= emission_probs[obs[i] + "|" + seq[i]]


            # update the total score
            #print state_score
            total_score_breakdown.append(state_score)
            total_score *= state_score

        sequence_scores.append(total_score)
```

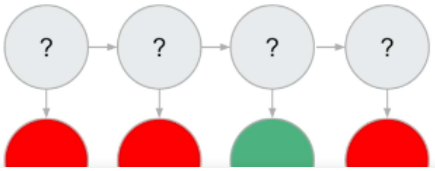
```
from tabulate import tabulate

def pretty_print_probs(distribs):

    rows = Set()
    cols = Set()
    for val in distribs.keys():
        temp = val.split("|")
        rows.add(temp[0])
        cols.add(temp[1])

    rows = list(rows)
    cols = list(cols)


    df = []
    for i in range(len(rows)):
        temp = []
        for j in range(len(cols)):


            temp.append(distribs[rows[i]+"|"+cols[j]])

        df.append(temp)

    I = pd.Index(rows, name="rows")
    C = pd.Index(cols, name="cols")
    df = pd.DataFrame(data=df,index=I, columns=C)

    print tabulate(df, headers='keys', tablefmt='psql')

def initializeSequences(_obs):
    # Generate list of sequences

    seqLen = len(_obs)


    seqs = generate_sequence(states,seqLen)


    # Score sequences
    seq_scores =
score_sequences(seqs,initial_probs,transition_probs,emission_probs,obs
)

    return (seqLen,seqs,seq_scores)
```

Compute the best sequence (Viterbi)

Note that selecting the best scoring sequence is also known as the **Viterbi** score. The alternative approach is the **Minimum Bayes Risk** approach which selects the highest scoring position across all sequence scores.

Example 1

Let us consider the below graph



```
# We can use a dictionary to capture our state transitions

# set of hidden states
states = ['A','B']

# set of observations
obs = ['Red','Green','Red']

# initial state probability distribution (our priors)
initial_probs = {'A':1.0,'B':0.0}

# transition probabilities
transition_probs = {'A|A':0,'A|B':1,'B|A':1,'B|B':0}

# emission probabilities
emission_probs = {'Red|A':1,'Green|A':0,'Red|B':0.25,'Green|B':0.75}

# Generate list of sequences
sequence_length,sequences,sequence_scores = initializeSequences(obs)

# print results

print("Initial Distributions")
print initial_probs

print("\nTransition Probabilities")
pretty_print_probs(transition_probs)

print("\nEmission Probabilities")
pretty_print_probs(emission_probs)

print("\nScores")

# Display sequence scores
for i in range(len(sequences)):
    print("Sequence:%10s,Score:%0.4f" %
(sequences[i],sequence_scores[i]))
```

```
Initial Distributions
{'A': 1.0, 'B': 0.0}

Transition Probabilities
+--------+-----+-----+
| rows   |   A |   B |
|--------+-----+-----|
| A      |   0 |   1 |
| B      |   1 |   0 |
+--------+-----+-----+

Emission Probabilities
+--------+-----+------+
| rows   |   A |    B |
|--------+-----+------|
| Green  |   0 | 0.75 |
| Red    |   1 | 0.25 |
+--------+-----+------+

Scores
Sequence:['A', 'A', 'A'],Score:0.0000
Sequence:['A', 'A', 'B'],Score:0.0000
Sequence:['A', 'B', 'A'],Score:0.7500
Sequence:['A', 'B', 'B'],Score:0.0000
Sequence:['B', 'A', 'A'],Score:0.0000
```

Example 2 — Parts of Speech (POS)Tagging Example

In the example below, we look at Parts of Speech tagging for a simple sentence. The sequence of words in the sentence are the observations and the Parts of Speech are the hidden states. Given a sentence, we are looking to predict the corresponding POS tags.

Let us consider the below graph, where the states are known and represent the POS tags and the red/green circles represent the observations or the sequence of words.

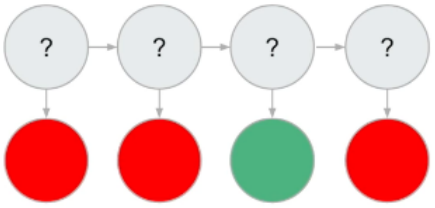


Figure — 13: HMM — Toy Example — Scoring an Unknown Sequence

The code below initializes probability distributions for our priors, hidden states and observations. It then generates a set of all possible sequences for the hidden states. We will use this later to compute the score for each possible sequence.

```
# generate new sequences

states = ['Noun','Verb','Determiner']
initial_probs = {'Noun':0.9,'Verb':0.05,'Determiner':0.05}
transition_probs =
{'Noun|Noun':0.1,'Noun|Verb':0.1,'Noun|Determiner':0.8,

'Verb|Noun':0.8,'Verb|Verb':0.1,'Verb|Determiner':0.1,

'Determiner|Noun':0.1,'Determiner|Verb':0.8,'Determiner|Determiner':0.
1}
emission_probs =
{'Bob|Noun':0.9,'ate|Noun':0.05,'the|Noun':0.05,'fruit|Noun':0.9,\

'Bob|Verb':0.05,'ate|Verb':0.9,'the|Verb':0.05,'fruit|Verb':0.05,\

'Bob|Determiner':0.05,'ate|Determiner':0.05,'the|Determiner':0.9,'frui
t|Determiner':0.05}


print("Initial Distributions")
print initial_probs


print("\nTransition Probabilities")
pretty_print_probs(transition_probs)


print("\nEmission Probabilities")
pretty_print_probs(emission_probs)
```

```
Transition Probabilities
+------------+--------+--------+--------------+
| rows       |   Verb |   Noun |   Determiner |
|------------+--------+--------+--------------|
| Verb       |    0.1 |    0.8 |          0.1 |
| Noun       |    0.1 |    0.1 |          0.8 |
| Determiner |    0.8 |    0.1 |          0.1 |
+------------+--------+--------+--------------+


Emission Probabilities
+--------+--------+--------+--------------+
| rows   |   Verb |   Noun |   Determiner |
|--------+--------+--------+--------------|
| Bob    |   0.05 |   0.9  |         0.05 |
| fruit  |   0.05 |   0.9  |         0.05 |
| the    |   0.05 |   0.05 |         0.9  |
| ate    |   0.9  |   0.05 |         0.05 |
+--------+--------+--------+--------------+


obs = ['Bob','ate','the','fruit']


# print results
print("\nScores")


# Generate list of sequences
sequence_length,sequences,sequence_scores = initializeSequences(obs)


# Display sequence scores
for i in range(len(sequences)):
    print("Sequence:%-60s Score:%0.6f" %
(sequences[i],sequence_scores[i]))

# Display the winning score
print("\n Best Sequence")
print(sequences[sequence_scores.index(max(sequence_scores))],max(seque
nce_scores))


Scores
Sequence:['Noun', 'Noun', 'Noun', 'Noun']
Score:0.000002
Sequence:['Noun', 'Noun', 'Noun', 'Verb']
Score:0.000001
Sequence:['Noun', 'Noun', 'Noun', 'Determiner']
Score:0.000000
Sequence:['Noun', 'Noun', 'Verb', 'Noun']
Score:0.000015
Sequence:['Noun', 'Noun', 'Verb', 'Verb']
Score:0.000001
Sequence:['Noun', 'Noun', 'Verb', 'Determiner']
Score:0.000006
Sequence:['Noun', 'Noun', 'Determiner', 'Noun']
Score:0.000262
Sequence:['Noun', 'Noun', 'Determiner', 'Verb']
Score:0.000002
Sequence:['Noun', 'Noun', 'Determiner', 'Determiner']
Score:0.000002
Sequence:['Noun', 'Verb', 'Noun', 'Noun']
Score:0.000262
Sequence:['Noun', 'Verb', 'Noun', 'Verb']
Score:0.000117
Sequence:['Noun', 'Verb', 'Noun', 'Determiner']
Score:0.000015
Sequence:['Noun', 'Verb', 'Verb', 'Noun']
Score:0.000262
Sequence:['Noun', 'Verb', 'Verb', 'Verb']
Score:0.000015
Sequence:['Noun', 'Verb', 'Verb', 'Determiner']
Score:0.000117
Sequence:['Noun', 'Verb', 'Determiner', 'Noun']
Score:0.000231
```

```
Score:0.000015
Sequence:['Noun', 'Determiner', 'Noun', 'Verb']
Score:0.000006
Sequence:['Noun', 'Determiner', 'Noun', 'Determiner']
Score:0.000001
Sequence:['Noun', 'Determiner', 'Verb', 'Noun']
Score:0.000002
Sequence:['Noun', 'Determiner', 'Verb', 'Verb']
Score:0.000000
Sequence:['Noun', 'Determiner', 'Verb', 'Determiner']
Score:0.000001
Sequence:['Noun', 'Determiner', 'Determiner', 'Noun']
Score:0.000262
Sequence:['Noun', 'Determiner', 'Determiner', 'Verb']
Score:0.000002
Sequence:['Noun', 'Determiner', 'Determiner', 'Determiner']
Score:0.000002
Sequence:['Verb', 'Noun', 'Noun', 'Noun']
Score:0.000000
Sequence:['Verb', 'Noun', 'Noun', 'Verb']
Score:0.000000
Sequence:['Verb', 'Noun', 'Noun', 'Determiner']
Score:0.000000
Sequence:['Verb', 'Noun', 'Verb', 'Noun']
Score:0.000000
Sequence:['Verb', 'Noun', 'Verb', 'Verb']
Score:0.000000
Sequence:['Verb', 'Noun', 'Verb', 'Determiner']
Score:0.000000
Sequence:['Verb', 'Noun', 'Determiner', 'Noun']
Score:0.000001
Sequence:['Verb', 'Noun', 'Determiner', 'Verb']
Score:0.000000
Sequence:['Verb', 'Noun', 'Determiner', 'Determiner']
Score:0.000000
Sequence:['Verb', 'Verb', 'Noun', 'Noun']
Score:0.000000
Sequence:['Verb', 'Verb', 'Noun', 'Verb']
Score:0.000000
Sequence:['Verb', 'Verb', 'Noun', 'Determiner']
Score:0.000000
Sequence:['Verb', 'Verb', 'Verb', 'Noun']
Score:0.000000
Sequence:['Verb', 'Verb', 'Verb', 'Verb']
Score:0.000000
Sequence:['Verb', 'Verb', 'Verb', 'Determiner']
Score:0.000000
Sequence:['Verb', 'Verb', 'Determiner', 'Noun']
Score:0.000117
Sequence:['Verb', 'Verb', 'Determiner', 'Verb']
Score:0.000001
Sequence:['Verb', 'Verb', 'Determiner', 'Determiner']
Score:0.000001
Sequence:['Verb', 'Determiner', 'Noun', 'Noun']
Score:0.000000
Sequence:['Verb', 'Determiner', 'Noun', 'Verb']
Score:0.000000
Sequence:['Verb', 'Determiner', 'Noun', 'Determiner']
Score:0.000000
Sequence:['Verb', 'Determiner', 'Verb', 'Noun']
Score:0.000000
Sequence:['Verb', 'Determiner', 'Verb', 'Verb']
Score:0.000000
Sequence:['Verb', 'Determiner', 'Verb', 'Determiner']
Score:0.000000
Sequence:['Verb', 'Determiner', 'Determiner', 'Noun']
Score:0.000006
Sequence:['Verb', 'Determiner', 'Determiner', 'Verb']
Score:0.000000
Sequence:['Verb', 'Determiner', 'Determiner', 'Determiner']
Score:0.000000
Sequence:['Determiner', 'Noun', 'Noun', 'Noun']
Score:0.000000
```

```
Sequence:['Determiner', 'Noun', 'Verb', 'Verb']
Score:0.000000
Sequence:['Determiner', 'Noun', 'Verb', 'Determiner']
Score:0.000000
Sequence:['Determiner', 'Noun', 'Determiner', 'Noun']
Score:0.000006
Sequence:['Determiner', 'Noun', 'Determiner', 'Verb']
Score:0.000000
Sequence:['Determiner', 'Noun', 'Determiner', 'Determiner']
Score:0.000000
Sequence:['Determiner', 'Verb', 'Noun', 'Noun']
Score:0.000000
Sequence:['Determiner', 'Verb', 'Noun', 'Verb']
Score:0.000000
Sequence:['Determiner', 'Verb', 'Noun', 'Determiner']
Score:0.000000
Sequence:['Determiner', 'Verb', 'Verb', 'Noun']
Score:0.000000
Sequence:['Determiner', 'Verb', 'Verb', 'Verb']
Score:0.000000
Sequence:['Determiner', 'Verb', 'Verb', 'Determiner']
Score:0.000000
Sequence:['Determiner', 'Verb', 'Determiner', 'Noun']
Score:0.000117
Sequence:['Determiner', 'Verb', 'Determiner', 'Verb']
Score:0.000001
Sequence:['Determiner', 'Verb', 'Determiner', 'Determiner']
Score:0.000001
Sequence:['Determiner', 'Determiner', 'Noun', 'Noun']
Score:0.000000
Sequence:['Determiner', 'Determiner', 'Noun', 'Verb']
Score:0.000000
Sequence:['Determiner', 'Determiner', 'Noun', 'Determiner']
Score:0.000000
Sequence:['Determiner', 'Determiner', 'Verb', 'Noun']
Score:0.000000
Sequence:['Determiner', 'Determiner', 'Verb', 'Verb']
Score:0.000000
Sequence:['Determiner', 'Determiner', 'Verb', 'Determiner']
Score:0.000000
Sequence:['Determiner', 'Determiner', 'Determiner', 'Noun']
Score:0.000001
Sequence:['Determiner', 'Determiner', 'Determiner', 'Verb']
Score:0.000000
Sequence:['Determiner', 'Determiner', 'Determiner', 'Determiner']
Score:0.000000


 Best Sequence
(['Noun', 'Verb', 'Determiner', 'Noun'], 0.30233088000000014)
```

Computing the Minimum Bayes Risk (MBR) Score

Here, we try to find out the best possible value for a particular y location location where y represents our hidden states starting from y=0,1…n-1, where n is the sequence length

For example, if we need to first pick the position we are interested in, let's say we are in the second position of the hidden sequence i.e. y1. We examine the set of sequences and their scores, only this time, we group sequences by possible values of y1 and compute the total scores within each group. The group with the highest score is the forward/backward score

```
obs =
['Bob','ate','the','fruit','the','fruit','Bob','ate','ate','the','frui
t']


## recompute sequences and sequence scores
sequence_length,sequences,sequence_scores = initializeSequences(obs)


## compute MBR for a particular position
mbr_index = 4 # MBR 0-based


fb_scores = {}


# display sequence scores
for i in range(len(sequences)):

    if fb_scores.get(sequences[i][mbr_index]) == None:
        fb_scores[sequences[i][mbr_index]] = sequence_scores[i]
    else:
        fb_scores[sequences[i][mbr_index]] += sequence_scores[i]

print "Minimum Bayes Risk Scores for :"
for key in fb_scores:
    print("%-20s %0.8f" % (key,fb_scores[key]))

best_fb_score = max(fb_scores.values())
best_y = [key for (key,value) in fb_scores.items() if value ==
best_fb_score]
print "\nBest Score:%0.8f, \nState Index:%d \nBest y state:%s" %
(best_fb_score,mbr_index,best_y )


Minimum Bayes Risk Scores for :
Verb                 0.00001816
Noun                 0.00000108
Determiner           0.00008688


Best Score:0.00008688,
State Index:4
Best y state:['Determiner']
```

Optimizing the efficiency of HMM computations using Dynamic Programming

Since predicting the optimal sequence using HMM can become computational tedious as the number of the sequence length increases, we resort to dynamic programming (cached recursion) in order to improve its performance.

Examining computation complexity as the sequence length increases

In this section, we will increase our sequence length to a much longer sentence and examine the impact on computation time. The same performance issues will also be encountered if the number of states is large, although in this case, we will only tweak the sequence length.

```
# initialize a new sequence of observations
obs =
```

```
import time
startTime = time.time()


## recompute sequences and sequence scores
sequence_length,sequences,sequence_scores = initializeSequences(obs)


# Display partial list of sequence scores
rangeLen = 0
if len(sequences) > 10:
    rangeLen = 10
else:
    rangeLen = len(sequences)

for i in range(rangeLen):
    print("Sequence:%-60s Score:%0.6f" %
(sequences[i],sequence_scores[i]))

# Display the winning score
print("\n Best Sequence")
print(sequences[sequence_scores.index(max(sequence_scores))],max(seque
nce_scores))


# print time
print("\n Sequence length:%5d, State count: %5d, Time taken:%0.4f
seconds" %\
      (sequence_length,len(initial_probs),time.time()-startTime))


Sequence:['Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun',
'Noun', 'Noun', 'Noun', 'Noun'] Score:0.000000
Sequence:['Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun',
'Noun', 'Noun', 'Noun', 'Verb'] Score:0.000000
Sequence:['Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun',
'Noun', 'Noun', 'Noun', 'Determiner'] Score:0.000000
Sequence:['Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun',
'Noun', 'Noun', 'Verb', 'Noun'] Score:0.000000
Sequence:['Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun',
'Noun', 'Noun', 'Verb', 'Verb'] Score:0.000000
Sequence:['Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun',
'Noun', 'Noun', 'Verb', 'Determiner'] Score:0.000000
Sequence:['Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun',
'Noun', 'Noun', 'Determiner', 'Noun'] Score:0.000000
Sequence:['Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun',
'Noun', 'Noun', 'Determiner', 'Verb'] Score:0.000000
Sequence:['Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun',
'Noun', 'Noun', 'Determiner', 'Determiner'] Score:0.000000
Sequence:['Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun', 'Noun',
'Noun', 'Verb', 'Noun', 'Noun'] Score:0.000000


 Best Sequence
(['Noun', 'Verb', 'Determiner', 'Noun', 'Determiner', 'Noun', 'Noun',
'Verb', 'Verb', 'Determiner', 'Noun'], 5.92297667290203e-05)


 Sequence length:   11, State count:     3, Time taken:2.7134 seconds
```

Notice that the time taken get very large even for small increases in sequence length and for a very a small state count.

Dynamic Programming Intuition for HMMs

Intuition behind the Dynamic Programming algorithm for computing MBR scores

improvements in performance compared to other methods of computation.

The MBR solution can be computed using dynamic programming. MBR allows us to compute the sum over all sequences conditioned on keeping one of the hidden states at a particular position fixed. This in turn allows us to determine the best score for a given state at a given position. We do this by computing the best score for every state at that position and pick the state that has the highest score.

Consider the example of a sequence of four words — "Bob ate the fruit". Let us assume that we would like to compute the MBR score conditioned on the hidden state at position 1 (y1) being a Noun (N).

This computation can be mathematically shown to be equivalent to

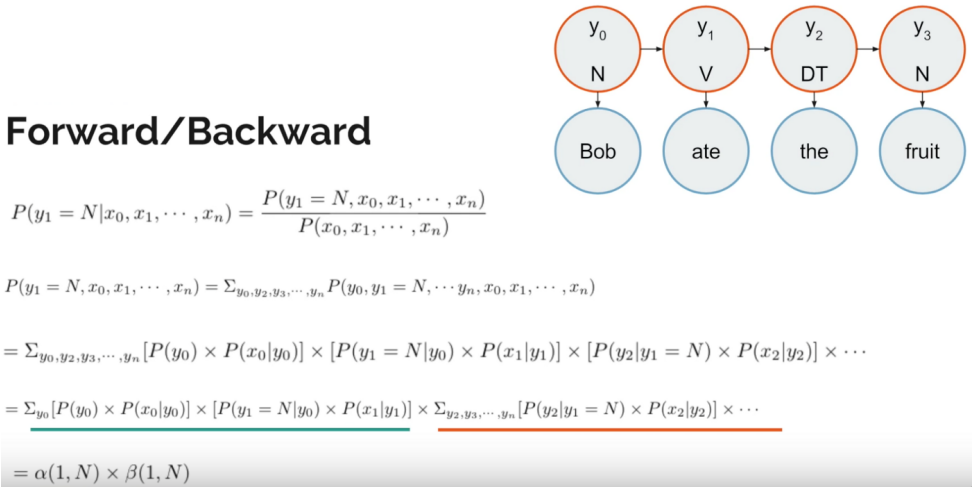


Figure — 14: HMM — Dynamic Programming — Finding the MBR Score
Source: UC Berkeley lectures

We break up the computation into two parts. In the first part, we compute **alpha**, the sum of all possible ways that the sequence can end up as a Noun in position 1 and in the second part, we compute beta, the sum of all possible ways that the sequence can start as a Noun.

Forward/Backward Algorithm for computing the MBR scores

The below code computes our alpha and beta values. We make dynamic caching an argument in order to demonstrate performance differences with and without caching.

```
alpha_cache = {}
```

```
        if alpha_cache.get((pos,state)) != None:
            return alpha_cache[(pos,state)]

    if pos == 0:
        return initial_probs[state] * emission_probs[textList[pos] +
"|" + state]
    else:
        total = 0
        for state_val in states:
            total += alpha(pos-1,state_val,textList,cachedRecursion) \
            * transition_probs[state+"|"+state_val] *
emission_probs[textList[pos]+"|"+state]

        # if cache is enabled, then cache
        if cachedRecursion == True:
            if alpha_cache.get((pos,state)) == None:
                alpha_cache[(pos,state)] = total

        return total


beta_cache = {}


def beta(pos,state,textList,currIndex=0,currState=None,cachedRecursion
= True):

    if cachedRecursion == True:

        # if cache is enabled, try to read from cache
        if beta_cache.get((currIndex,currState)) != None:
            return beta_cache[(currIndex,currState)]

    if currIndex == 0:
        currIndex = len(textList)

        if pos == currIndex - 1:
            return 1

        total = 0
        for state_val in states:
            tempSum = beta(pos,state,textList,currIndex-
1,state_val,cachedRecursion)

            total += tempSum

        # if cache is enabled, then cache
        if cachedRecursion == True:
            if beta_cache.get((currIndex,currState)) == None:
                beta_cache[(currIndex,currState)] = total

        return total

    elif currIndex == pos+1:


        return transition_probs[currState+"|"+state] *
emission_probs[textList[currIndex]+"|"+currState]

    else:
        total = 0
        for state_val in states:

            inStateProb = transition_probs[currState+"|"+state_val] *
emission_probs[textList[currIndex]+"|"+currState]
            tempSum = inStateProb *beta(pos,state,textList,currIndex-
1,state_val,cachedRecursion)


            total += tempSum

        # if cache is enabled, then cache
        if cachedRecursion == True:
            if beta_cache.get((currIndex,currState)) == None:
                beta_cache[(currIndex,currState)] = total
```

We will now test out the dynamic programming algorithm with and without caching enabled to look at performance improvements.

For our dataset, we will use a much longer sequence since we have a much more efficient algorithm.

```
# set observations
obs = ['Bob','ate','the','fruit','the','fruit','Bob','ate',\

'ate','the','fruit','Bob','ate','the','fruit','the','fruit','Bob']


position = 4
part_of_speech = "Determiner"
```

Forward/Backward algorithm for computing MBR score without caching

```
# measure time taken without caching (dynamic programming)
import time


startTime = time.time()


# variables for saving alpha and beta values
alpha_cache = {}
beta_cache = {}


# get alpha
alpha_res = alpha(position,part_of_speech,obs,cachedRecursion=False)


# get beta
beta_res = beta(position,part_of_speech,obs,cachedRecursion=False)


mbr_score = alpha_res * beta_res


print("MBR Score for Position:%d and POS:%s is %0.8f" %
(position,part_of_speech,mbr_score))
# print time
print("\n Time taken:%0.4f seconds" % (time.time()-startTime))


MBR Score for Position:4 and POS:Determiner is 0.00000004


 Time taken:3.7872 seconds
```

Forward/Backward algorithm for computing MBR score with caching

```
# measure time taken with caching (dynamic programming)
import time


startTime = time.time()
```

```
# get alpha
alpha_res = alpha(position,part_of_speech,obs,cachedRecursion=True)

# get beta
beta_res = beta(position,part_of_speech,obs,cachedRecursion=True)

mbr_score = alpha_res * beta_res

print("MBR Score for Position:%d and POS:%s is %0.8f" %
(position,part_of_speech,mbr_score))
# print time
print("\n Time taken:%0.4f seconds" % (time.time()-startTime))

MBR Score for Position:4 and POS:Determiner is 0.00000004

 Time taken:0.0010 seconds
```

Notice the significant improvements in time when we use the version with cached recursion.

Conclusion

HMMs are used in a variety of scenarios including Natural Language Processing, Robotics and Bio-genetics. In this post, we saw some of the basics of HMMs, especially in the context of NLP and Parts of Speech tagging. In later posts, I hope to elaborate on other HMM concepts based on Expectation Maximization and related algorithms.

I want to acknowledge my gratitude to James Kunz and Ian Tenney, lecturers at the UC Berkeley Information and Data Science program, for their help and support.