# Answers to questions:

## Grade E&F:

**Q1)**

**Q2)**

**Q3)**

**Q4)**

**Q5)**

**Q6)**

## Grade C:

**Q7)**

**Data input:**

A = [[0.54, 0.26, 0.20], [0.19, 0.53, 0.2], [0.22, 0.18, 0.6]]

B = [[0.5, 0.2, 0.11, 0.19], [0.22, 0.28, 0.23, 0.27], [0.19, 0.21, 0.15, 0.45]]

init = [[0.3, 0.2, 0.5]]

Yes, using the data input shown the algorithm converges. For this experiment we said that the algorithm converges when P(O| A, B, pi) stops increasing.

**1,000 observations:**

Iterations: 2065

Output:

3 3 0.6964766442970416 0.013355500327566279 0.29016785537539225 0.10146488076010847 0.8120130398101552 0.08652207942973535 0.19211560411183184 0.3012796579335449 0.5066047379546228

3 4 0.688797084807084 0.22515630665157194 0.07537023801106281 0.010676370530282557 0.06786811354215612 0.41206671714071835 0.281391909974649 0.2386732593424755 1.0272023550311935e-47 6.998660721589178e-13 0.3533016260184206 0.6466983739808791

1 1.0000000000000093 0.0 0.0

**10,000 observations:**

Iterations: 15037

Output:

3 3 0.6942811145388932 0.044898639823539206 0.2608202456375609 0.11766838877994286 0.7460770992650253 0.1362545119550297 0.15418588819336737 0.25669380062547803 0.5891203111811582

3 4 0.7099995142759916 0.186408624423441 0.10359164828185681 2.1301870576781607e-07 0.09881201121335675 0.42112526065889333 0.3121739309602372 0.16788879716751484 0.032113199454733544 0.17132508129377239 0.18662834089300043 0.609933378358495

1 0.0 0.9999999999999932 5e-323

Overall, you can see that the algorithm converges for both input files. There is the possibility that this algorithm can get stuck in a local maximum. However, since we used non-uniform input values for A & B this should represent the global maximum. Using 10,000 gives a better estimate of A, B and pi although it takes a lot more time to run with 10 times the number of observations at every iteration. With a larger amount of data we reduce the effect of outliers and get better statistics.

**Q8)**

**Data input:**

A = [[0.40, 0.3, 0.3], [0.4, 0.3, 0.3], [0.4, 0.3, 0.3]]

B = [[0.2, 0.3, 0.35, 0.15], [0.2, 0.3, 0.35, 0.15], [0.2, 0.3, 0.35, 0.15]]

init = [[0.3, 0.3, 0.4]]

To initialise the parameters similar values were used for each data point in a row. However, we ensured that the matrices were row-stochastic without making the values exactly uniform.

The problem in estimating the distance between these matrices is how to define distance. Figure 1 shows 4 examples of distance metrics each with different properties. Depending on which of these you use you may get a different answer. For example, if you use the third metric the matrices will be deemed to be far apart even if there is only one value in the matrices that are far apart. To solve this issue you could use a number of distance metrics in order to reach consensus. Otherwise, the distance metric whose properties best fits the use case can be used.

Text

Description automatically generated

*Figure 1 -* [*Source - stack exchange*](https://math.stackexchange.com/questions/507742/distance-similarity-between-two-matrices)

For this question we implemented the second distance metric from Figure 1. You can see from the figures below that this performed significantly worse than using the initialisation values from question 7.

Using provided parameters from Q7 – (1000 input)

A 0.05834575500864262

B 0.1907634231000179

Own initialisation values – (1000 input)

A: 0.4945289168985252

B: 1.0751490820505725

**Q9)**

**2 hidden states**

Input

A = [[0.7, 0.3], [0.7, 0.3], [0.7, 0.3]]

B = [[0.2, 0.3, 0.35, 0.15], [0.2, 0.3, 0.35, 0.15]]

init = [[0.3, 0.3]]

Output

2 2 0.8098117584012856 0.1901882415987124 0.3567636032167863 0.6432363967832141

2 4 4.5732394211256464e-10 0.29230783967966256 0.3202511627871271 0.3874409970758843 0.6959795609063234 0.16540516131451025 0.07894627655561874 0.0596690012235495

1 0.0 1.000000000000019

**4 hidden states**

Input

A = [[0.40, 0.3, 0.2, 0.1], [0.4, 0.2, 0.3, 0.1], [0.4, 0.1, 0.3, 0.2], [0.4, 0.1, 0.3, 0.2]]

B = [[0.2, 0.3, 0.35, 0.15], [0.2, 0.3, 0.35, 0.15], [0.2, 0.3, 0.35, 0.15], [0.2, 0.3, 0.35, 0.15]]

init = [[0.3, 0.3, 0.3, 0.1]]

Output

4 4 0.7106102340360291 0.17115623299873758 9.15433544013585e-153 0.118233532965233 4.751304958608988e-09 0.5700845527032721 0.429915442545423 8.881617561166809e-130 0.1420621555569139 2.308286553874782e-114 0.4732529667224929 0.3846848777205923 0.04017718281146178 0.11326864097077143 0.3152064992343726 0.5313476769833925

4 4 0.6748683886219542 0.26928854339631997 0.055843067981724955 7.3757218971183e-210 0.5306563578936456 1.4017947277020926e-12 0.21544496013906897 0.2538986819658836 2.7368166341899825e-203 6.699037466601632e-25 0.27989579387397195 0.7201042061260275 0.06432683174495489 0.6184812417165031 0.31719192653854056 3.9456432917684145e-34

1 0.0 0.9999999999999933 0.0 0.0

When too few states are used with a HMM then there is the possibility that the problem is being over generalised (underfitting). Additional states could give us more information on the most likely observation sequence. However, if we use too many states then the model may be too specific. This model may then be overfitted to the data it is trained on and generalise poorly on new data. To use a high number of states a significant amount of data is needed. As we use more states we need more data so that the probabilities that underlie the model are statistically significant.

In this case we believe that three hidden states and four types of observations is the best choice for this problem as the observation sequence was generated from an HMM with these dimensions. We know that this is the ground truth when it comes to this observation sequence and that it is therefore the optimal solution.

**Q10)**

**Initialize your Baum-Welch algorithm with a uniform distribution.**

Input:

A = [[0.333, 0.333, 0.333], [0.333, 0.333, 0.333], [0.333, 0.333, 0.333]]

B = [[0.25, 0.25, 0.25, 0.25], [0.25, 0.25, 0.25, 0.25], [0.25, 0.25, 0.25, 0.25]]

init = [[0.333, 0.333, 0.333]]

Iterations: 2

How does this affect the learning?

When initialised with a uniform distribution the algorithm gets stuck in a local maximum and does not converge on the global maximum. When the B matrix is uniform it provides zero extra information. The algorithm is unable to move away from the uniform distribution and therefore after only two iterations it fails to improve.

**Initialize your Baum-Welch algorithm with a diagonal A matrix and π = [0, 0, 1].**

Input:

A = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

B = [[0.5, 0.2, 0.11, 0.19], [0.22, 0.28, 0.23, 0.27], [0.19, 0.21, 0.15, 0.45]]

init = [[0.3, 0.2, 0.5]]

Iterations: N/A

How does this affect the learning?

We get some values when using the file with 1000 observations. However, the diagonal matrix causes the program to throw an error during learning when using the file with 10,000 observations. Since many of the values in the A matrix are zero, there is a division by zero problem when re-calculating A and B. Some gamma values are set to zero as alpha is used in their calculation. This gamma is then used in updating the A and B matrices resulting in this error.

**Initialize your Baum-Welch algorithm with a matrices that are close to the solution. How does this affect the learning?**

Input:

A = [[0.699, 0.041, 0.26], [0.099, 0.791, 0.11], [0.199, 0.291, 0.51]]

B = [[0.699, 0.201, 0.099, 0.001], [0.099, 0.391, 0.305, 0.205], [0.001, 0.099, 0.199, 0.701]]

init = [[0.3, 0.2, 0.5]]

Iterations:

How does this affect the learning?

When using values close to the solution the algorithm converges quicker. As you are close to the solution from the start the log probability stops improving after less iterations. However, even with the values close to the solution the algorithm doesn’t converge on the true values. This is probably due to some error and rounding that takes place in the algorithm.