Learning Algorithms for Mixtures of Linear Dynamical Systems: A Practical Approach

by

Nitin A. Kumar

S.B. Computer Science and Engineering and Mathematics
Massachusetts Institute of Technology, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Nitin A. Kumar Department of Electrical Engineering and Computer Science May 17, 2024
Certified by:	Ankur Moitra Norbert Wiener Professor of Mathematics, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Learning Algorithms for Mixtures of Linear Dynamical Systems: A Practical Approach

by

Nitin A. Kumar

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

In this work, we give the first implementation of an algorithm to learn a mixture of linear dynamical systems (LDS's), and an analysis of algorithms to learn a single linear dynamical system. Following the work of Bakshi et al. ([1]), we implement a recent polynomial-time algorithm based on a tensor decomposition with learning guarantees in a general setting, with some simplifications and minor optimizations. Our largest contribution is giving the first expectation-maximization (E-M) algorithm for learning a mixture of LDS's, and an experimental evaluation against the Tensor Decomposition algorithm. We find that the E-M algorithm performs extremely well, and much better than the Tensor Decomposition algorithm. We analyze performance of these and other algorithms to learn both a single LDS and a mixture of LDS's under various conditions (such as how much noise is present) and algorithm settings.

Thesis supervisor: Ankur Moitra
Title: Norbert Wiener Professor of Mathematics

Acknowledgments

I would like to express my deepest appreciation to everyone who has supported me through this journey and on the path leading up to it.

This thesis would not have been possible without the patient mentorship of Professor Ankur Moitra, whose encouragement and insightful guidance have been invaluable.

I am also deeply indebted to my family, especially my parents, and my friends, for their love, support, and advice, and for helping to shape me into the person I am today.

Contents

Chapter 1

LDS Background

1.1 What does it mean to learn an LDS?

Definition 1 (Linear Dynamical System)**.** *Formally, a **linear dynamical system** (LDS) consists of a sequence of **inputs** $\mathbf{u}_t \in \mathbb{R}^p$, **hidden states** $\mathbf{x}_t \in \mathbb{R}^n$, and **outputs** $\mathbf{y}_t \in \mathbb{R}^m$. The hidden state evolves as a linear function of its previous value and the previous input, plus some noise term:*

$$\mathbf{x}_t = \mathbf{A}\mathbf{x}_{t-1} + \mathbf{B}\mathbf{u}_{t-1} + \mathbf{w}_t$$

and the output is a linear function of the hidden state and the input, again with some noise:

$$\mathbf{y}_t = \mathbf{C}\mathbf{x}_t + \mathbf{D}\mathbf{u}_t + \mathbf{z}_t$$

The matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$, and $\mathbf{D}$ are called the **system parameters**. This model describes a plethora of systems. For instance, because differentiation and integration are linear operators, a discretization of system evolving according to Newtonian mechanics would be a linear dynamical system.

Our goal is to learn (mixtures of) LDS's, by which we mean that we would like to estimate the system parameters $\mathbf{A}, \mathbf{B}, \mathbf{C}$, and $\mathbf{D}$. This would be much easier if we could observe the input $\mathbf{u}_t$, the hidden state $\mathbf{x}_t$, and the output $\mathbf{y}_t$, as we could just regress $\mathbf{x}_t$ on $\mathbf{x}_{t-1}$ and $\mathbf{u}_t$ to find $\mathbf{A}$ and $\mathbf{B}$, and similarly regress $\mathbf{y}_t$ on $\mathbf{x}_t$ and $\mathbf{u}_t$ to find $\mathbf{C}$ and $\mathbf{D}$. We will restrict ourselves to the partially observed setting, where we do not observe the hidden state; we only observe the input and the output. We call a set of inputs and outputs together a **trajectory**.

Because we never observe the hidden state, we can only hope to learn it up to some invertible linear transformation $\mathbf{T}$. To illustrate this point, let $\mathbf{x'}_t = \mathbf{T}\mathbf{x}_t$. Then, our LDS could be equivalently described by the equations:

$$\begin{aligned}\mathbf{x'}_t &= \mathbf{T}\mathbf{A}\mathbf{T}^{-1}\mathbf{x'}_{t-1} + \mathbf{T}\mathbf{B}\mathbf{u}_{t-1} + \mathbf{w}_t \\ \mathbf{y}_t &= \mathbf{C}\mathbf{T}^{-1}\mathbf{x'}_t + \mathbf{D}\mathbf{u}_t + \mathbf{z_t}\end{aligned}$$

Thus, if we can learn some $\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}$, and $\hat{\mathbf{D}}$ from the inputs and outputs such that for some invertible lienar transformation $\mathbf{T}$, we have

$$\hat{\mathbf{A}} \approx \mathbf{TAT}^{-1}, \hat{\mathbf{B}} \approx \mathbf{TB}, \hat{\mathbf{C}} \approx \mathbf{CT}^{-1}, \hat{\mathbf{D}} \approx \mathbf{D}$$

then we have succeeded.

There are some more technical conditions for the learnability of an LDS. (To give some idea of why this is: if we had $\mathbf{C} = \mathbf{0}$, then the output would not depend on the hidden state, so learning $\mathbf{A}$ and $\mathbf{B}$ would be impossible). See [1] for more details. All the examples we will use in this paper satisfy learnability assumptions that make learning possible.

We make a few more assumptions on the problem setup. For simplicity, we will also assume that the noises $\mathbf{w}_t$ and $\mathbf{z}_t$ are each i.i.d. and Gaussian. Further, we will assume that the $\mathbf{u}_t$ are exogenous (as opposed to being chosen by us) and that they are also i.i.d. Gaussian random variables with distribution $\mathcal{N}(\mathbf{0}, \mathbf{I}_p)$.

1.2 Markov Parameters of an LDS

As a conceptual aid, we can imagine that the effect of the input is propagated to the hidden state and the output via the following diagram:

$$\begin{tikzcd}
\mathbf{u}_t \arrow[d, "\mathbf{B}"] \arrow[dd, bend right=40, "\mathbf{D}"'] \\
\mathbf{x}_t \arrow[d, "\mathbf{C}"] \arrow[loop right, "\mathbf{A}"] \\
\mathbf{y}_t
\end{tikzcd}$$

Informally, $\mathbf{u}_t$ contributes $\mathbf{D}\mathbf{u}_t$ to $\mathbf{y}_t$ and $\mathbf{B}\mathbf{u}_t$ to $\mathbf{x}_{t+1}$. Then, $\mathbf{x}_{t+1}$ contributes $\mathbf{A}\mathbf{x}_{t+1}$ to $\mathbf{x}_{t+2}$, and similarly $\mathbf{x}_{t+2}$ contributes $\mathbf{A}\mathbf{x}_{t+2}$ to $\mathbf{x}_{t+3}$, and so on, so the net effect is that for $i \geq 1$, $\mathbf{u}_t$ contributes $\mathbf{A}^{i-1}\mathbf{B}\mathbf{u}_t$ to $\mathbf{x}_{t+i}$, and thus contributes $\mathbf{C}\mathbf{A}^{i-1}\mathbf{B}\mathbf{u}_t$ to $\mathbf{y}_{t+i}$. The contribution of $\mathbf{u}_t$ to $\mathbf{y}_{t+i}$ for $i \geq 0$ motivates our definition of the ith **Markov parameter** of an LDS.

Definition 2 (Markov Parameters of an LDS)**.** *We define the ith **Markov parameter** $\mathbf{M}_i$ of an LDS to be $\mathbf{D}$ if $i = 0$ and $\mathbf{C}\mathbf{A}^{i-1}\mathbf{B}$ for $i \geq 1$. Note that if we ignore the effects of noise* *(i.e. set $\mathbf{w}_t = \mathbf{0}$ and $\mathbf{z}_t = \mathbf{0}$) and the initial state (i.e. set $\mathbf{x}_0 = \mathbf{0}$), then we get that*

$$\mathbf{y}_t = \sum_{i=0}^{t} \mathbf{M}_{t-i}\mathbf{u}_i \tag{1.1}$$

It is apparent that how the size of the Markov parameters changes is important. Since $\mathbf{M}_i = \mathbf{C}\mathbf{A}^{i-1}\mathbf{B}$ for $i \geq 1$, the Markov parameters will grow if the spectral radius $\rho(A) > 1$, shrink if $\rho(A) < 1$, and remain relatively constant if $\rho(A) = 1$. If $\rho(A) > 1$, the system will blow up exponentially, with earlier inputs having a dominant effect on the output. We won't work much with this setting because as the Markov parameters grow, the noise becomes relatively smaller and smaller, making the system easy to learn. If $\rho(A) < 1$, the system is easier to learn, since the impact of earlier inputs becomes negligible eventually, and consequently the covariance between outputs tends to zero as the time between the outputs increases. This is called the **strictly stable** case. The $\rho(A) = 1$ case is the hardest for learning, and is called the **marginally stable** case.

The system is also determined by these Markov parameters. As discussed in the previous section, we can only learn the system parameters of an LDS up to an invertible transformation $\mathbf{T}$ of the hidden state, but such a transformation preserves the Markov parameters. That is, if we replace $\mathbf{A}$ with $\mathbf{T}\mathbf{A}\mathbf{T}^{-1}$, $\mathbf{B}$ with $\mathbf{T}\mathbf{B}$, and $\mathbf{C}$ with $\mathbf{C}\mathbf{T}^{-1}$, then the Markov parameters $\mathbf{D}$ and $\mathbf{C}\mathbf{T}^{-1}(\mathbf{T}\mathbf{A}\mathbf{T}^{-1})^{i-1}\mathbf{T}\mathbf{B} = \mathbf{C}\mathbf{A}^{i-1}\mathbf{B}$ are unchanged. Further, if the system is learnable, then the Markov parameters will determine the system parameters up to a linear transformation of the hidden state. Consequently, we will measure our learning error by measuring the error in the Markov parameters that we recover.

1.3 Existing Literature and Our Contributions

The problem of learning a mixture of linear dynamical systems from unlabeled trajectories is relatively recent. The first work ([2]) was done by Chen and Poor in 2022. However, this work was done in the fully observed case, without inputs or a hidden state (i.e., with $\mathbf{B} = \mathbf{0}, \mathbf{C} = \mathbf{I}, \mathbf{D} = \mathbf{0}$, and $\mathbf{z}_t = 0$). The partially observed setting is significantly more challenging.

Bakshi, Liu, Moitra, and Yau give the first algorithm for learning a mixture of linear dynamical systems in the partially observed setting in polynomial time ([1]), and do so even in great generality (including when the system is marginally stable). We write down for the first time an expectation-maximization (EM) algorithm for learning a mixture of linear dynamical systems. Given that linear dynamical systems are so widely used, it is important to implement them and evaluate the practicality of algorithms to learn them, which we aim to do in this work. The code we wrote can be found in Appendix A.

It is worth mentioning that much research has been done on how to learn a single LDS from a single long trajectory (see [3]) for example. This problem is related to ours, but not the same.

Chapter 2

Learning a Single LDS

2.1 Overview

We now turn to the problem of learning a single LDS, given many short trajectories from the LDS. These ideas will be helpful when trying to learn mixtures of LDS's, and sometimes algorithms for learning a single LDS may even be used as a subroutine.

The two major ideas in this section are both based on the approximation from earlier (1.1):

$$\mathbf{y}_t \approx \sum_{i=0}^{t} \mathbf{M}_{t-i}\mathbf{u}_i$$

In both cases, we will first estimate the first few Markov parameters, and then use the Robust Ho-Kalman algorithm ([4]) to recover system parameters. First, we explain how to estimate the Markov parameters.

For some parameter s, we will want to estimate $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_{2s}$. There are a few considerations to account for when choosing s:

- There are some technical conditions about the **observability** and **controllability** of the system that may require us to choose fairly large values of s, or make matrices better-conditioned for larger values of s. See [1] for more details.
- Larger values of s will give us more Markov parameters from which to estimate the system parameters, which can improve accuracy. On the other hand, if we increase s too much, our estimates of Markov parameters can become very poor, which can hurt accuracy.
- Larger values of s will increase time complexity.
- In order for $\mathbf{M}_{2s}$ to appear in the above sum, we need trajectories of length at least $2s + 1$.

2.2 Direct Covariance Algorithm

Our first algorithm is based on the following idea: for $j \leq t$,

$$\begin{aligned}
\mathbb{E}\left[\mathbf{y}_t \mathbf{u}_j^\top\right] &= \sum_{i=0}^{t} \mathbf{M}_{t-i} \mathbb{E}\left[\mathbf{u}_i \mathbf{u}_j^\top\right] && \text{(linearity of expectation)} \\
&= \mathbf{M}_{t-j} \mathbb{E}\left[\mathbf{u}_j \mathbf{u}_j^\top\right] && (\mathbf{u}_i \text{ are zero-mean and independent}) \\
&= \mathbf{M}_{t-j} \mathbf{I} && (\mathbf{u}_j \sim \mathcal{N}(\mathbf{0}, \mathbf{I})) \\
&= \mathbf{M}_{t-j} && (2.1)
\end{aligned}$$

Of course, there can be significant variance, but as long as the length of the trajectories is bounded, the variance is finite, so we should expect that the empirical mean of $\mathbf{y}_t \mathbf{u}_j^\top$ does converge to $\mathbf{M}_{t-j}$ with enough samples. This gives us our estimate of the Markov parameters for what we call the Direct Covariance algorithm.

Here is the Python code that implements this algorithm:

```python
def get_markov_params_cov(trajectories, s):
    return [np.mean([
                        np.outer(y[i+k1], u[i]) for u, y in trajectories
                        ↪  for i in range(len(u) - k1)
            ], axis=0) for k1 in range(2*s+1)]
```

2.3 Regression Algorithm

We can do a bit better with the same number of trajectories by trying to account for more terms in the sum. For convenience, we define $\mathbf{u}_i = \mathbf{0}$ for $i < 0$. Then, we can approximate:

$$\mathbf{y}_t \approx \sum_{i=0}^{t} \mathbf{M}_{t-i} \mathbf{u}_i \approx \sum_{i=t-s}^{t} \mathbf{M}_{t-i} \mathbf{u}_i = \sum_{j=0}^{s} \mathbf{M}_j \mathbf{u}_{t-j}$$

Since we know the $\mathbf{y}_t$ and $\mathbf{u}_i$, we can now estimate $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_{2s}$ via linear regression. This gives us our estimate of the Markov parameters for what we call the Regression algorithm.

Here is the Python code that implements this algorithm:

```python
def get_markov_params_regression(trajectories, s):
    traj_len, p = trajectories[0][0].shape
    coeffs = np.empty((len(trajectories)*traj_len, (2*s+1)*p))
    targets = np.concatenate([y for _, y in trajectories], axis=0)
    for i, (u, _) in enumerate(trajectories):
        flat_inp = np.concatenate([np.zeros(2*s*p), np.concatenate(u)])
        for j in range(traj_len):
            coeffs[i*traj_len + j] = flat_inp[j*p:(j+2*s+1)*p]

    markov_params = la.lstsq(coeffs, targets)[0].reshape((2*s+1, p,
     ↪  -1)).transpose((0, 2, 1))
    return np.flip(markov_params, axis=0)
```

2.4 Robust Ho-Kalman Algorithm

Now, we need to estimate the system parameters. We already have our estimate of $\mathbf{D} = \mathbf{M}_0$. The Robust Ho-Kalman algorithm to estimate the remaining system parameters is motivated by the following block matrix factorizations:

$$
\begin{aligned}
\mathbf{H}^{-} :=& \begin{bmatrix} \mathbf{M}_1 & \mathbf{M}_2 & \dots & \mathbf{M}_s \\ \mathbf{M}_2 & \mathbf{M}_3 & \dots & \mathbf{M}_{s+1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{M}_s & \mathbf{M}_{s+1} & \dots & \mathbf{M}_{2s-1} \end{bmatrix} \\
=& \begin{bmatrix} \mathbf{CB} & \mathbf{CAB} & \dots & \mathbf{CA}^{s-1}\mathbf{B} \\ \mathbf{CAB} & \mathbf{CA}^2\mathbf{B} & \dots & \mathbf{CA}^{s}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{CA}^{s-1}\mathbf{B} & \mathbf{CA}^{s}\mathbf{B} & \dots & \mathbf{CA}^{2s-2}\mathbf{B} \end{bmatrix} \\
=& \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \vdots \\ \mathbf{CA}^{s-1} \end{bmatrix} \begin{bmatrix} \mathbf{B} & \mathbf{AB} & \dots & \mathbf{A}^{s-1}\mathbf{B} \end{bmatrix}
\end{aligned}
$$

and

$$\begin{aligned}
\mathbf{H}^+ &:= \begin{bmatrix} \mathbf{M}_2 & \mathbf{M}_3 & \dots & \mathbf{M}_{s+1} \\ \mathbf{M}_3 & \mathbf{M}_4 & \dots & \mathbf{M}_{s+2} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{M}_{s+1} & \mathbf{M}_{s+2} & \dots & \mathbf{M}_{2s} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{CAB} & \mathbf{CA}^2\mathbf{B} & \dots & \mathbf{CA}^s\mathbf{B} \\ \mathbf{CA}^2\mathbf{B} & \mathbf{CA}^3\mathbf{B} & \dots & \mathbf{CA}^{s+1}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{CA}^s\mathbf{B} & \mathbf{CA}^{s+1}\mathbf{B} & \dots & \mathbf{CA}^{2s-1}\mathbf{B} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \vdots \\ \mathbf{CA}^{s-1} \end{bmatrix} \mathbf{A} \begin{bmatrix} \mathbf{B} & \mathbf{AB} & \dots & \mathbf{A}^{s-1}\mathbf{B} \end{bmatrix}
\end{aligned}$$

Using these matrix factorizations, it turns out that if we take any $\mathbf{O} \in \mathbb{R}^{ms\times n}$ and $\mathbf{Q} \in \mathbb{R}^{n\times ps}$ such that $\mathbf{H}^- = \mathbf{OQ}$, this is equivalent to taking $\mathbf{O} = \begin{bmatrix} \mathbf{C} \\ \mathbf{CA} \\ \vdots \\ \mathbf{CA}^{s-1} \end{bmatrix}$ and $\mathbf{Q} = \begin{bmatrix} \mathbf{B} & \mathbf{AB} & \dots & \mathbf{A}^{s-1}\mathbf{B} \end{bmatrix}$.
That is, we can take $\mathbf{C}$ to be the first m rows of $\mathbf{O}$, $\mathbf{B}$ to be the first p columns of $\mathbf{Q}$, and $\mathbf{A}$ to be $\mathbf{O}^{-1}\mathbf{H}^+\mathbf{Q}^{-1}$ (here $\mathbf{X}^{-1}$ denotes the pseudo-inverse of $\mathbf{X}$) to recover the parameters of the LDS up to a rotation of the hidden state. This algorithm is fairly stable when we use estimations of the Markov parameters instead of the real values. See [4] for more details.
Here is the Python code that implements this algorithm:

```
def get_lds_parameters(markov_params, m, n, p):
    markov_params = np.array(markov_params).reshape((-1, m, p))
    D = markov_params[0]
    s = len(markov_params)//2

    # Ho-Kalman
    H = np.block([[markov_params[i+j+1] for j in range(s+1)] for i in
     ↪  range(s)])
    H_minus, H_plus = H[:, :p*s], H[:, -p*s:]
    U, S, Vh = la.svd(H_minus)
    U, S, Vh = U[:,:n], S[:n], Vh[:n,:]
    O, Q = U * np.sqrt(S), np.sqrt(S.reshape((-1, 1))) * Vh
    C, B = O[:m], Q[:,:p]
    A = la.pinv(O) @ H_plus @ la.pinv(Q)
    return A, B, C, D
```

We can piece together the two algorithms to approximate the Markov parameters with the Robust Ho-Kalman Algorithm:

```
def learn_single_cov(trajectories, s, m, n, p):
    return LDS(*get_lds_parameters(get_markov_params_cov(trajectories,
     ↪  s), m, n, p))

def learn_single_regression(trajectories, s, m, n, p):
    return
     ↪  LDS(*get_lds_parameters(get_markov_params_regression(trajectories,
     ↪  s), m, n, p))
```

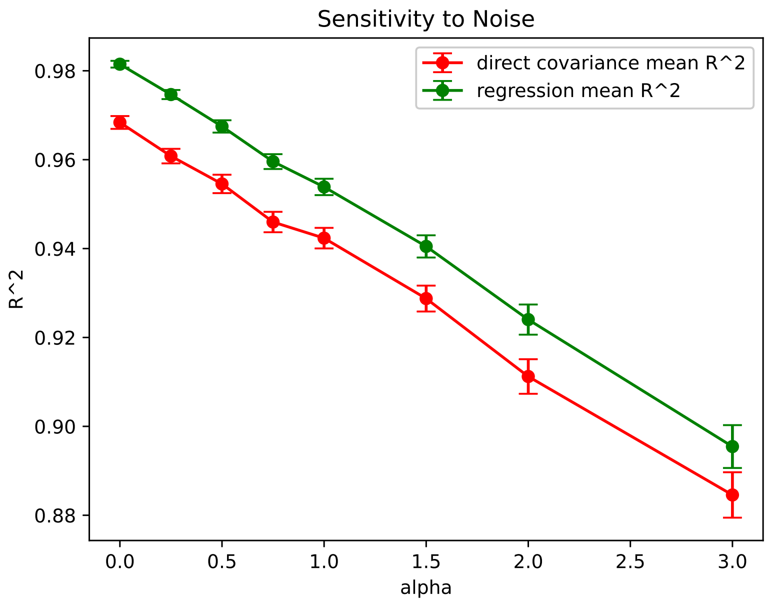
2.5 Results

Both the Direct Covariance algorithm and the Regression algorithm worked fairly well. To measure the performance of these algorithms, we first ran the algorithms to recover an approximation to the LDS. Then, we computed the first 10 Markov parameters of the recovered LDS and measured the R^2 value of the recovered Markov parameters as a prediction of the true Markov parameters. That is, if the true Markov parameters are $\mathbf{M}^{(10)} := \begin{bmatrix}\mathbf{M}_0 & \mathbf{M}_1 & \dots \mathbf{M}_9\end{bmatrix}$ and the recovered Markov parameters are $\hat{\mathbf{M}}^{(10)} := \begin{bmatrix}\hat{\mathbf{M}}_0 & \hat{\mathbf{M}}_1 & \dots \hat{\mathbf{M}}_9\end{bmatrix}$, we report

$$R^2 = 1 - \frac{||\mathbf{M}^{(10)} - \hat{\mathbf{M}}^{(10)}||_2^2}{||\mathbf{M}^{(10)}||_2^2}$$

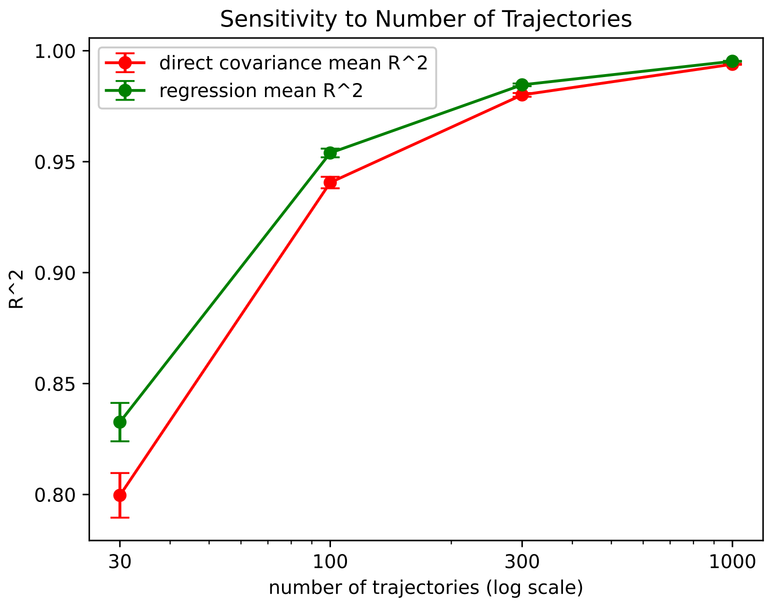
Unless otherwise specified, we took $\mathbf{A}, \mathbf{B}, \mathbf{C}$, and $\mathbf{D}$ to be identity matrices, $s = 2$, $\mathbf{w}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, $\mathbf{z}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, $m = n = p = 2$, and 100 trajectories of length 20. We varied several parameters to test the algorithm under different conditions. Note that varying $m = n = p$ should not affect the performance of the algorithm significantly, since the components are essentially kept separate. (Even if we pick the system parameters differently, the components are still essentially unmixed in the eigenbasis of $\mathbf{A}$).

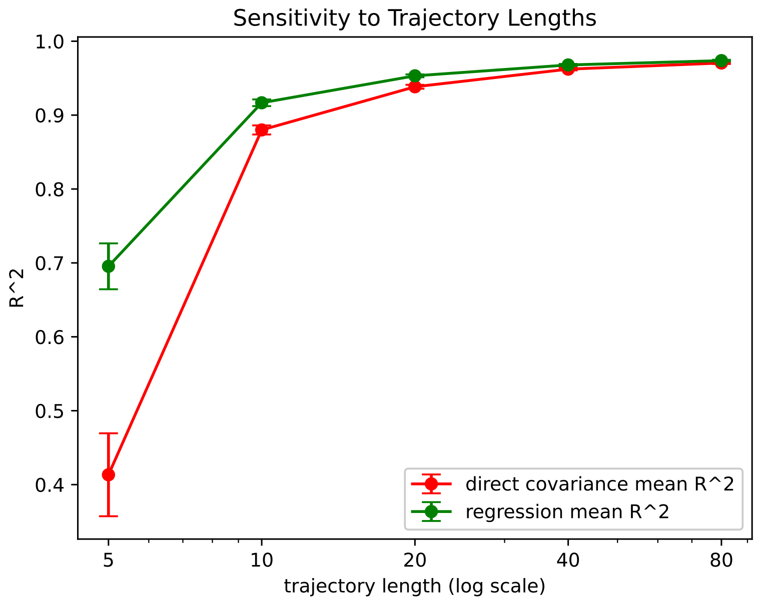
First, to test the effect of noise, we changed the noise distributions to $\mathbf{w_t} \sim \mathcal{N}(\mathbf{0}, \alpha\mathbf{I})$ and $\mathbf{z}_t \sim \mathcal{N}(\mathbf{0}, \alpha\mathbf{I})$. When reporting results, the error bars are always two standard errors of the mean. Here are the results for each setting:



(Note that the only reason that the regression algorithm does not perform perfectly in the zero-noise ($\alpha = 0$) case is because we limit the number of inputs that it takes into account, so the old inputs are effectively a source of noise. Since $\mathbf{A} = \mathbf{I}$, the old inputs are as important as the new inputs, and since $s = 2$, we account for the last $2s + 1 = 5$ inputs. This means that for a trajectory of length 20, we have accounted for all the inputs at the start of the trajectory, but only 25% of the inputs at the end of the trajectory.)

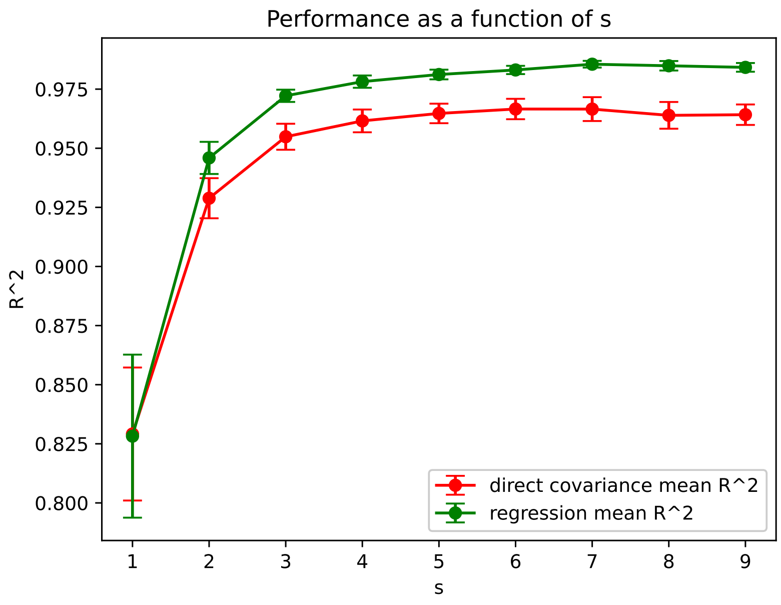
We also tested the effect of the number of trajectories used to learn the system and the length of trajectories used to learn the system.



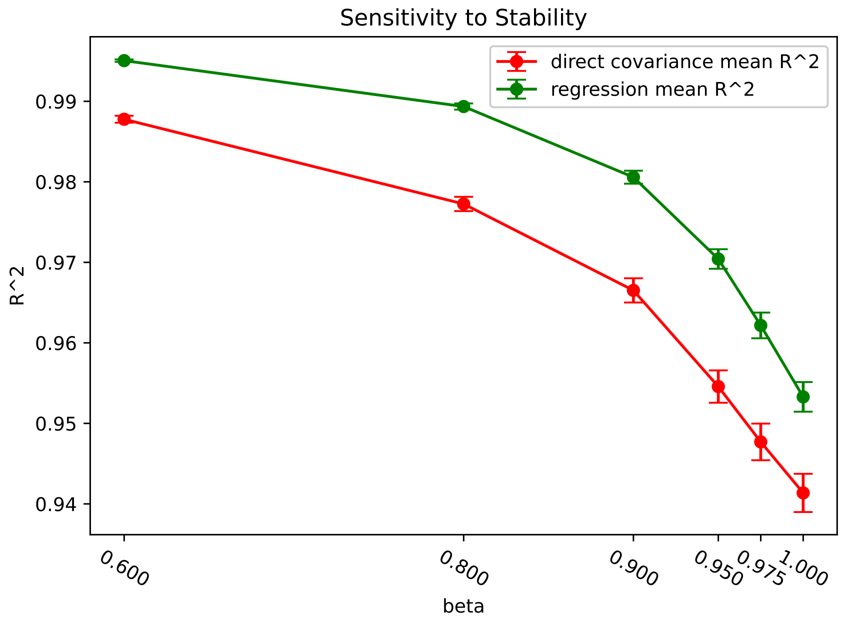


We found that after a trajectory length of about 15, increasing the length of trajectories was essentially like increasing the number of trajectories. Both increase the number of data points used by the Direct Covariance and Regression algorithms in much the same way.

Varying s can improve the algorithm, but eventually performance will drop off, for the reasons discussed above. When $s = 1$, the Direct Covariance and Regression algorithms become very similar, so their performance turns out to be very similar as well.



As the spectral norm of $\mathbf{A}$ decreases, the importance of old inputs decreases, and so the system is easier to learn. We can measure the resulting performance increase by changing $\mathbf{A}$ to $\beta \cdot \mathbf{I}$. As β goes farther from 1, the system becomes more and more strictly stable.



The system also becomes harder to learn if we cannot fully observe the hidden state, which happens when $\mathbf{C}$ does not have full rank. To measure this effect, we set $\mathbf{C} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$. However, we need to mix the components so that we no longer only observe the first component, so we set $\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. We will call the original system $\mathcal{S}$ and call this new system with partial observability $\mathcal{S}'$.

Algorithm	System	Mean	2x SEM
Direct Covariance	$\mathcal{S}$	0.938	$2.7 \cdot 10^{-3}$
Regression	$\mathcal{S}$	0.950	$2.6 \cdot 10^{-3}$
Direct Covariance	$\mathcal{S}'$	0.952	$2.0 \cdot 10^{-3}$
Regression	$\mathcal{S}'$	0.964	$1.7 \cdot 10^{-3}$

2.5.1 Conclusions

As expected, the results bore out that the Regression algorithm performs slightly better than the Direct Covariance algorithm. Both can learn the trajectory quite well, even with a few short trajectories, especially for slightly higher values for s. We have made the problem especially difficult by choosing a marginally stable A, a low value of s, and a fair amount of noise, so this success is fairly promising. The algorithms were also fairly robust in harder settings, such as when they had fewer or shorter trajectories or more noise.

Chapter 3

Learning a Mixture of LDS's

3.1 Overview

It may happen that we want to learn a **mixture** of LDS's. Formally, a mixture of LDS's is a finite set of LDS's $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_k$ along with corresponding nonnegative mixing weights $w_1, w_2, \ldots, w_k$ that sum to 1. We observe trajectories from this mixture of LDS's, which means that we first sample a system $\mathcal{S}_i$ according to the mixing weights, and then obtain a trajectory from $\mathcal{S}_i$. From unlabeled trajectories from the mixture of LDS's, we would like to learn the mixing weights and the parameters of each LDS.

To motivate this problem, imagine a biological setting in which we are modeling the evolution of a population of some species. The populations may differ in phenotype, or have qualitative differences in their environment that affect the population dynamics, even if we may not be able to observe these differences.

Our learning problem is now slightly more complicated. As before, we can only hope to learn the system parameters up to a rotation of the hidden state. Additionally, we want to learn the approximate mixing weight for each LDS. We have no canonical order for the component LDS's, so we will also only try to learn the mixture up to a permutation of the component LDS's (and corresponding mixing weights). Additionally, we need the LDS's to be reasonably different from each other, so that we can distinguish them. We measure similarity by ensuring the Markov parameters are not too close to linearly dependent. There are some more technical conditions for learnability; see [1] for more details.

Bakshi et. al. devised a polynomial algorithm for solving this problem in a recent paper ([1]). We implement this algorithm (which we call the Tensor Decomposition algorithm) and compare it to an E-M algorithm, which is a classic approach to learning mixture models ([5]). We also try using the output from the first algorithm as a warm start to the E-M algorithm to see if we can improve performance further.

3.2 Tensor Decomposition Algorithm

3.2.1 Algorithm Overview

To motivate this algorithm, let's first imagine what would happen if we tried to use the Direct Covariance algorithm on trajectories from a mixture of LDS's. For ease of notation, we denote the ith Markov parameter of system $\mathcal{S}_j$ by $\mathbf{M}_i^{(j)}$. As before, if we sampled from $\mathcal{S}_j$, we would find that $\mathbb{E}\left[\mathbf{y}_t \mathbf{u}_a^\top\right] = \mathbf{M}_{t-a}^{(j)}$. Therefore, if our trajectory is from the mixture, we end up with a mixture of the Markov parameters:

$$\mathbb{E}\left[\mathbf{y}_t \mathbf{u}_a^\top\right] = \sum_{j=1}^{k} w_j \mathbf{M}_{t-a}^{(j)} \tag{3.1}$$

If we knew how to decompose $\mathbb{E}\left[\mathbf{y}_t \mathbf{u}_a^\top\right]$ into the terms in the sum, we could recover the Markov parameters. To do such a general decomposition, we turn to three-dimensional tensors. In general, we can decompose a three-dimensional tensor into rank-one components, and further there is an efficient algorithm called Jennrich's algorithm to do so (see [6] for a full explanation of Jennrich's algorithm, or the original paper on the algorithm [1] for an overview of how it is used). Formally, for a tensor

$$\mathbf{T} \approx \sum_{i=1}^{r} (\mathbf{a}_i \otimes \mathbf{b}_i \otimes \mathbf{c}_i)$$

Jennrich's algorithm can approximately recover the terms in the sum in general. (It is impossible to recover the terms if, for instance, $(\mathbf{a}_i \otimes \mathbf{b}_i) \approx (\mathbf{a}_j \otimes \mathbf{b}_j)$ for $i \neq j$.)

Let f be the "flattening operation" that converts two-dimensional matrices into one-dimensional vectors by concatenating their rows. Further, since we will again use the Robust Ho-Kalman algorithm to recover the Markov parameters, we define:

$$\mathbf{M}^{(j)} = \begin{bmatrix} \mathbf{M}_0^{(j)} & \mathbf{M}_1^{(j)} & \dots & \mathbf{M}_{2s}^{(j)} \end{bmatrix}$$

Then, we will try to estimate the 3-d tensors that are a linear combination of $f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right)$. In particular, it follows from (2.1) that:

$$\begin{aligned} &\mathbb{E}\left[f\left(\mathbf{y}_{k_1+1}\mathbf{u}_1^\top\right) \otimes f\left(\mathbf{y}_{k_1+k_2+2}\mathbf{u}_{k_1+2}^\top\right) \otimes f\left(\mathbf{y}_{k_1+k_2+k_3+3}\mathbf{u}_{k_1+k_2+3}^\top\right)\right] \\ =&\sum_{j=1}^{k} w_j f\left(\mathbf{M}_{k_1}^{(j)}\right) \otimes f\left(\mathbf{M}_{k_2}^{(j)}\right) \otimes f\left(\mathbf{M}_{k_3}^{(j)}\right) \end{aligned}$$

We have succeeded at our goal! We can form a "block tensor", where the block indexed by (k_1, k_2, k_3) is the quantity above, and so we can estimate each block from the trajectories.

The full tensor will be:

$$\mathbf{\Pi} := \sum_{j=1}^{k} \left[w_j f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right) \right]$$
$$= \sum_{j=1}^{k} \left[f\left(w_j^{1/3}\mathbf{M}^{(j)}\right) \otimes f\left(w_j^{1/3}\mathbf{M}^{(j)}\right) \otimes f\left(w_j^{1/3}\mathbf{M}^{(j)}\right) \right]$$

and so from Jennrich's algorithm we can recover $w_j^{1/3}\mathbf{M}^{(j)}$, which we will call **Markov components**. We also know from (3.1) that we can estimate $\sum_{j=1}^{k} w_j \mathbf{M}^{(j)}$, which is a linear combination of the Markov components. From linear regression, we can find the weights of the linear combination, which will be the $w_j^{2/3}$, from which we can find the weights and the Markov parameters. As before, we finish with the Robust Ho-Kalman algorithm for each LDS.

Intuitively, if the system is well-conditioned, every quantity we need to find the average of has finite variance, and so we can expect the quantities converge to the mean in time polynomial in the error tolerance. Thus, the algorithm should be able to approximate the mixture of LDS's in polynomial time. For more details, see the original paper ([1]) on the algorithm.

3.2.2 Optimizations and Implementation Tricks

The algorithm as described above (and in the original paper) is fairly impractical. There are a number of issues, but the main ones are:

- Simply writing down and averaging the terms in the three-dimensional tensor (each of which is really a product of six vectors) takes a long time.
- While the variance may be bounded because the trajectories have finite length, the variance can still be very large – after all, to calculate each term, we are multiplying six vector entries.
- Jennrich's algorithm can be fairly sensitive to noise. The decomposition of a tensor into rank-one components is inherently somewhat delicate. This means we need a fairly good approximation to reasonably approximate the mixture.

Two Dimensions Instead of Three

To ameliorate these issues, we would really like to get away with a two-dimensional matrix instead of a three-dimensional tensor. Unfortunately, decomposing the matrix

$$\sum_{j=1}^{k} \left[w_j f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right) \right]$$

into the terms in the sum is an impossible problem in general. However, if for random weights $w'_1, w'_2, \dots w'_k$, we can also find

$$\sum_{j=1}^{k} \left[w'_j f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right) \right]$$

then the decomposition becomes possible. In fact, this is the main idea of Jennrich's algorithm ([6]).

Firstly, if we could find two sources of trajectories that are mixtures of the same LDS's, but with different weights, then we would be able to do the decomposition immediately.

Otherwise, Jennrich's algorithm attempts to do the re-weighting via taking a random linear combination of each layer of the three-dimensional tensor. Formally, for a random vector $\mathbf{v}$, we can approximate

$$\begin{aligned} \mathbf{\Pi} \cdot \mathbf{v} &= \sum_{j=1}^{k} \left[w_j f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right) \cdot \left(f\left(\mathbf{M}^{(j)}\right) \cdot \mathbf{v} \right) \right] \\ &= \mathbb{E}\left[f\left(\mathbf{y}_{k_1+1}\mathbf{u}_1^\top\right) \otimes f\left(\mathbf{y}_{k_1+k_2+2}\mathbf{u}_{k_1+2}^\top\right) \cdot \left(f\left(\mathbf{y}_{k_1+k_2+k_3+3}\mathbf{u}_{k_1+k_2+3}^\top\right) \cdot \mathbf{v} \right) \right] \end{aligned}$$

which allows us to use

$$w'_j := w_j \cdot \left(f\left(\mathbf{M}^{(j)}\right) \cdot \mathbf{v} \right)$$

This is already a big improvement for the practicality of implementation: instead of writing out the full three-dimensional tensor, we can do this dot product as we go so that we only have to write down a two-dimensional tensor instead.

Reducing Noise from Collapsing a Dimension

We can make a couple more improvements: first, since $\mathbf{M}^{(j)}$ must be significantly different for each j, there will often be some l for which the $\mathbf{M}_l^{(j)}$ are significantly different. (Further, when the mixture consists of only a few LDS's, this must happen.) In this case, we can get a replace the re-weighting factor $\left(f\left(\mathbf{M}^{(j)}\right) \cdot \mathbf{v} \right)$ with $\left(f\left(\mathbf{M}_l^{(j)}\right) \cdot \mathbf{v} \right)$ (where, in each case, $\mathbf{v}$ is just a random vector). This greatly reduces the amount of computation we need to do, since we only have to estimate one of the Markov parameters, rather than several of them. In practice, we may not be able to determine l a priori, but we can try multiple values of l.

Additionally, especially when l is small, our estimate of $\mathbf{M}_l^{(j)}$ will have lower variance. The estimation we are making is based on (2.1); if we know that y and u are from LDS j:

$$\mathbb{E}\left[\mathbf{y}_{t_2}\mathbf{u}_{t_1}^\top\right] = \mathbf{M}_{t_2-t_1}^{(j)}$$

$\mathbf{y}_{t_2}\mathbf{u}_{t_1}^\top$ will have lower variance for lower values of t_2 and $(t_2 - t_1)$, because inputs and noise before t_2 other than $\mathbf{u}_{t_1}$ are effectively noise. l corresponds to $(t_2 - t_1)$, so if l can be small, then our estimate can have lower variance, which means we will need fewer samples. Moreover, we can set t_1 to be 1 by moving the dot product with $\mathbf{v}$ to the first dimension of the tensor. This means that in our estimation of

$$\sum_{j=1}^{k} \left[w'_j f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right) \right]$$

our estimate of w'_j will become better, but our estimate of $f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right)$ will become worse, because the inputs and outputs we use to estimate them now come later in the trajectory. We might believe this could benefit us overall, because wildly varying weightings can slow convergence of our estimate, and also because for a small value of k_1, the inputs and outputs we use to estimate $f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right)$ will not come much later in the trajectory. This does help our estimates empirically.

Other Tricks to Reduce Noise

The original Jennrich's algorithm takes two random weightings to collapse the three-dimensional tensor into two matrices, and uses both to extract the Markov components. We obtain a large savings by only getting one matrix this way, and constructing the other matrix without doing any re-weighting, as discussed. When we have two sets of trajectories with exogeneously different mixing weights, we don't need to do any random weighting at all, and so we achieve even better rates.

Another simple observation which can help us reduce noise is that the matrices we are trying to estimate,

$$\sum_{j=1}^{k} \left[w_j f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right) \right] \text{ and } \sum_{j=1}^{k} \left[w'_j f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right) \right]$$

are the sum of symmetric matrices, and thus themselves symmetric. Therefore, symmetrizing our estimates of these matrices (by averaging them with their transpose) is an easy way to reduce noise. As a bonus, it also ensures that when we do the decomposition into rank-one components, the rank-one components are also symmetric matrices.

3.2.3 Code

Here is the code which implements this algorithm. We have the function that estimates $\sum_{j=1}^{k} \left[w_j f\left(\mathbf{M}^{(j)}\right) \otimes f\left(\mathbf{M}^{(j)}\right) \right]$:

```
def get_equal_weighted_Pi_M(trajectories, s):
    ans = np.block([[
                        np.mean([
                            np.outer(np.kron(y[k1 + k2 + 1], u[k1+1]),
                             ↪  np.kron(y[k1], u[0]))
                        for u, y in trajectories], axis=0)
                    for k1 in range(2*s + 1)] for k2 in tqdm(range(2*s +
                     ↪  1))])
    return (ans + ans.T)/2 # denoise
```

and the function for estimating $\sum_{j=1}^{k}\left[w'_j f\left(\mathbf{M}^{(j)}\right)\otimes f\left(\mathbf{M}^{(j)}\right)\right]$:

```
def get_random_weighting(m, p):
    return normal(size=m*p)

def get_random_weighted_Pi_M(trajectories, s, differs_at,
 ↪  random_weighting = None):
    if random_weighting is None:
        u0, y0 = trajectories[0]
        random_weighting = get_random_weighting(y0.shape[1],
         ↪  u0.shape[1])

    ans = np.block([[
                        np.mean([
                            np.outer(np.kron(y[differs_at + k1 + k2 +
                             ↪  2], u[differs_at + k1 + 2]),
                             ↪  np.kron(y[differs_at + k1 + 1],
                             ↪  u[differs_at + 1])) *
                             ↪  np.dot(random_weighting,
                             ↪  np.kron(y[differs_at], u[0]))
                        for u, y in trajectories], axis=0)
                    for k1 in range(2*s + 1)] for k2 in tqdm(range(2*s +
                     ↪  1))])
    return (ans + ans.T)/2
```

From these two estimations, we can extract an estimate of $\pm w_j^{1/2}\cdot f(\mathbf{M}^{(j)})$ (these are the Markov components when using the two-dimensional tensor). We take the top k rank one components from the decomposition, since we have a mixture of k LDS's. The code is:

```
def extract_components(Pi_M1, Pi_M2, k):
    U1, S1, Vh1 = la.svd(Pi_M1, hermitian=True)
    U2, S2, Vh2 = la.svd(Pi_M2, hermitian=True)
    S1[k:] = 0
    S2[k:] = 0

    # because of symmetry, we actually don't need the full Jennrich's
     ↪  algorithm, we can get by with just the eigenvectors of U
    U = (U1 * S1) @ Vh1 @ (Vh2.T * np.reciprocal(S2, where=(S2 != 0)))
     ↪  @ U2.T
    #V = (U2 * S2) @ Vh2 @ (Vh1.T * np.reciprocal(S1, where=(S2 != 0)))
     ↪  @ U1.T
    U_lambdas, U_eigvecs = la.eig(U)
    U_lambdas, U_eigvecs = np.real(U_lambdas), np.real(U_eigvecs).T
    # V_lambdas, V_eigvecs = la.eig(V)
    return [vec for lmbda, vec in zip(U_lambdas, U_eigvecs) if
     ↪  np.abs(lmbda) > 1e-4]

def rescale_components(components, Pi_M):
    square_reweighting, _, _ , _ = la.lstsq(np.array([np.kron(vec, vec)
     ↪  for vec in components]).T, Pi_M.flatten())
    return components * np.sqrt(square_reweighting).reshape((-1, 1))
```

Finally, we need to find the weights, in a process analogous to in the three-dimensional tensor case. First, we estimate $\sum_{j=1}^{k} w_j \mathbf{M}^{(j)}$:

```
def get_R(trajectories, s):
    return np.array([np.mean([np.kron(y[k1], u[0]) for u, y in
     ↪  trajectories], axis=0) for k1 in range(2*s+1)]).flatten()
```

and then we use linear regression to recover the weights and the Markov parameters:

```
def get_weights_and_components(components, R):
    sqrt_weights, _, _, _ = la.lstsq(components.T, R)
    return sqrt_weights**2, components/sqrt_weights.reshape((-1, 1))
```

We then finish by converting the Markov parameters into system parameters via the same function we used when learning a single LDS.

3.3 E-M Algorithm

3.3.1 Algorithm Overview

An E-M Algorithm is a classic approach to learning mixture models ([5]). It is harder to give provable guarantees for an E-M algorithm, but it has historically been a practically efficient and accurate algorithm. For instance, an E-M algorithm is a popular approach to learning a mixture of Gaussians given a set of points ([7]). It has become popular due to its general robustness, although it is possible to run into spurious local minima. Such issues can largely be avoided by re-initializing the algorithm randomly and running it again, and after this fix the algorithm becomes fairly robust.

Let's describe what an E-M algorithm would look like for learning a mixture of LDS's.

After somehow initializing the mixture model, the algorithm consists of two steps that we repeat until convergence:

1. E-step: Classify trajectories according to which LDS they most likely came from.
2. M-step: Re-learn the mixture model. For each label, learn an LDS from the trajectories with that label as if all the trajectories are the same.

We know how to do the M-step already: this is just learning a single LDS, as in the previous chapter. We will use the Regression algorithm to do this. For the E-step, we will be able to utilize the celebrated Kalman filter. The Kalman filter ([8]) is traditionally used for estimating the hidden state when the LDS is known, but we will be able to adapt it to our purposes.

We will also examine various methods of initialization, and test accuracy in learning the mixture (the outcome of the M-step) and classifying trajectories according to which LDS they came from (the outcome of the E-step).

3.3.2 Finding Likelihood via the Kalman Filter

The Kalman filter ([8]) is widely used in controls for estimating the hidden state of a known LDS based on the previous inputs. It treats the inputs as exogenous, and updates its state upon seeing the outputs. The main idea is that for an LDS with Gaussian noise, the hidden state is a Gaussian process. There are three main updates we need make:

- Accounting for a new input: to account for $\mathbf{u}_t$, we just need to add $\mathbf{B}\mathbf{u}_t$ to the mean of the distribution of $\mathbf{x}_{t+1}$
- Updating upon seeing an output: this turns out to be a linear update to the Gaussian

- Accounting for the introduction of noise: we just need to add to the covariance of the distribution of the hidden state

It turns out we can also utilize the Kalman filter to estimate the likelihood of a trajectory from a given LDS. The Kalman filter can be broken down into two stages: first, a "prediction" phase in which we find the Gaussian distribution of the output, and secondly, an "update" phase in which we account for the actual value of the output. The first stage will give us what we need. Formally, for a trajectory of length l from a given LDS, we want to calculate the likelihood of the outputs given the inputs:

$$\begin{aligned}
&p\big((\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_l)\big|(\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_l)\big) \\
&= \prod_{i=1}^{l} p\big(\mathbf{y}_i\big|(\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_l), (\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{i-1})\big) && \text{(Bayes's Rule)} \\
&= \prod_{i=1}^{l} p\big(\mathbf{y}_i\big|(\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_i), (\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{i-1})\big)
\end{aligned}$$

We can calculate the terms in the product as we run the Kalman filter, since we calculate the Gaussian that is the PDF of the $\mathbf{y}_i$ given the previous inputs and outputs. Here is the code that finds the negative log-likelihood of a trajectory given an LDS:

```
def get_nll(self, trajectory):
    ans = np.double(0)

    u, y = trajectory
    x_mean, x_cov = np.zeros(self.n), np.eye(self.n)
    for t in range(len(u)):
        prefit_resid = y[t] - self.C @ x_mean - self.D @ u[t]
        prefit_cov = self.C @ x_cov @ self.C.T + self.output_cov
        prefit_cov_inv = la.inv(prefit_cov)
        nl_pdf = self.m/2 * np.log(2*np.pi) +
         ↪  np.log(la.det(prefit_cov))/2 + (prefit_resid @
         ↪  prefit_cov_inv @ prefit_resid)/2
        ans += nl_pdf

        kalman_gain = x_cov @ self.C.T @ prefit_cov_inv
        postfit_x_mean = x_mean + kalman_gain @ prefit_resid
        postfit_x_cov = x_cov - kalman_gain @ self.C @ x_cov
        x_mean = self.A @ postfit_x_mean + self.B @ u[t]
        x_cov = self.A @ postfit_x_cov @ self.A.T + self.process_cov

    return ans
```

3.3.3 Initializing the E-M Algorithm

We consider two main possibilities for initializing the E-M algorithm. We could initialize the algorithm with random LDS's, but we might worry that all the trajectories would be more likely to come from LDS's. To address this issue, for a random initialization, we can do the first E-step randomly; randomly assigning labels to each trajectory. Then, we can do the first M-step as normal, specifying a mixture for the first time by learning an LDS from each set of trajectories with the same label. For our E-M algorithm, we fell into spurious local minima when all the trajectories would get the same label. Like in the case of mixtures of Gaussians ([7]), when reaching these spurious local minima, we reinitialize and restart. This makes our algorithm very reliable in achieving high classification accuracy.

Another possibility is to use the Tensor Decomposition algorithm as a "warm start" for the E-M algorithm. This could help us get much faster convergence, since we will hopefully start with a mixture that is not too far from correct. From another lens, this is also an augmentation to the Tensor Decomposition algorithm: after running the Tensor Decomposition algorithm, we can run a few steps of the E-M algorithm to make the results even better. This also may allow us to use much less data than we would need to get good convergence from the Tensor Decomposition algorithm, since instead of relying on statistics that are fourth- or sixth-order to learn the Markov parameters, we can start using lower-noise second-order statistics in the E-M algorithm. The E-M algorithm cannot decrease the likelihood arising from the mixture outputted together with the label for each trajectory, so this is a strict improvement over the Tensor Decomposition algorithm alone.

3.4 Results

Despite a lack of proven guarantees about performance, we found that an E-M algorithm performed much better than the Tensor Decomposition algorithm in every setting we tested. (We tested over many variables, including sensitivity to noise, number of trajectories, trajectory lengths, values of s, sensitivity to stability, whether or not C was full-rank, different numbers of LDS's in the mixture, and different weights.) The E-M algorithm did not even benefit much from a warm start from the tensor decomposition algorithm: it converged in less than 10 iterations (often 5 or less) to the correct labels in all of the settings where the Tensor Decomposition algorithm was able to give any useful recovery of the mixture. As such, we would recommend using the E-M algorithm for learning mixtures of LDS's over the Tensor Decomposition algorithm.

To report the performance of each mixture-learning algorithm, we will use two fairly natural metrics:

- As before, when we were testing performance of algorithms to learn a single LDS, we will report the R^2 value from the first 10 Markov parameters arising from the outputted systems to the true Markov parameters. In all of our tests, each system in the mixture has an equal weight, we will take the average R^2 value for each system.

- We will report classification accuracy of the trajectories for the E-M algorithm, and the reported mixing weights for the Tensor Decomposition algorithm. The proportions of trajectories classified into each system are the mixing weights for the E-M algorithm.

Our mixture consisted of two systems. Both systems had mixing weight 1/2, $m = n = p = 2$, $\mathbf{B} = \mathbf{D} = \mathbf{I}$, and the same noise distributions $\mathbf{w}_t \sim \mathcal{N}(\mathbf{0}, \alpha\mathbf{I})$ and $\mathbf{z}_t \sim \mathcal{N}(\mathbf{0}, \alpha\mathbf{I})$. The systems were otherwise given by

1. $\mathbf{A} = \beta\mathbf{I}$, $\mathbf{C} = \mathbf{I}$
2. $\mathbf{A} = \beta \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, $\mathbf{C} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$

Notice that while system 1 is fairly simple, and the components operate independently, system 1 is different in that the components of the hidden state switch places at every step, and because of the construction of C, we only observe one of the components at each step. α represents how much noise the system has, and β represents how stable the system is, with $\beta = 1$ giving marginally stable systems and $\beta < 1$ giving strictly stable systems.

It is notable that the algorithm does eventually converge: with 100,000 trajectories, we achieve an average R^2 of 0.947, and an average mixture weight error of about .0203.

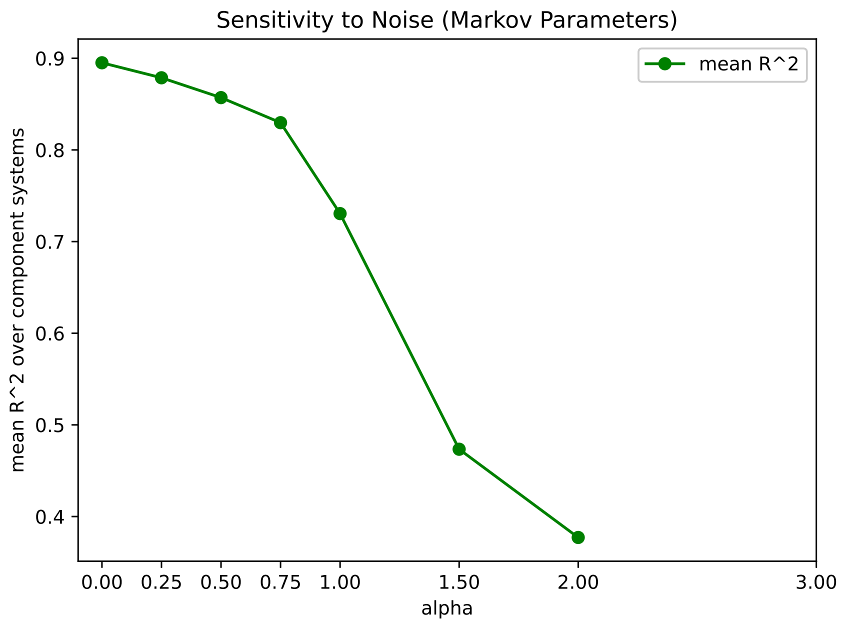
If we make the weights exogenously different, so that we have 5,000 trajectories with mixing weights of 50% for both systems 1 and 2, and 5,000 trajectories with a mixing weight of 25% for system 1 and 75% for system 2, we achieve an average R^2 of 0.965 and an average mixture weight error of 0.082. In this way, we can use many fewer trajectories to get comparable results.

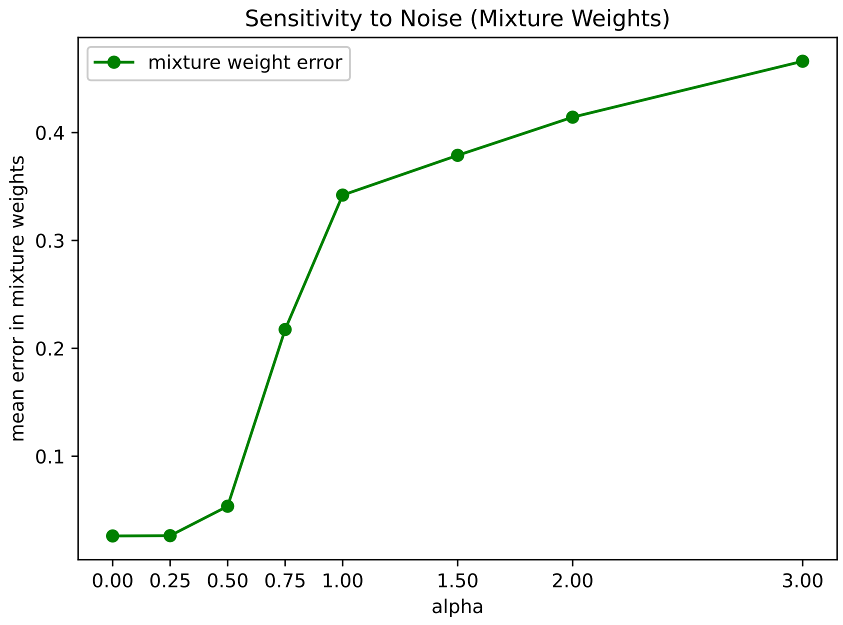
3.4.1 Tensor Decomposition Algorithm Results

The E-M outperformed the Tensor Decomposition algorithm so heavily that comparing the two would not be useful. However, we will examine the sensitivity to noise of the Tensor Decomposition algorithm to understand its effects. Additionally, because the Tensor Decomposition algorithm can operate in the marginally stable case, we also examine the sensitivity to stability.

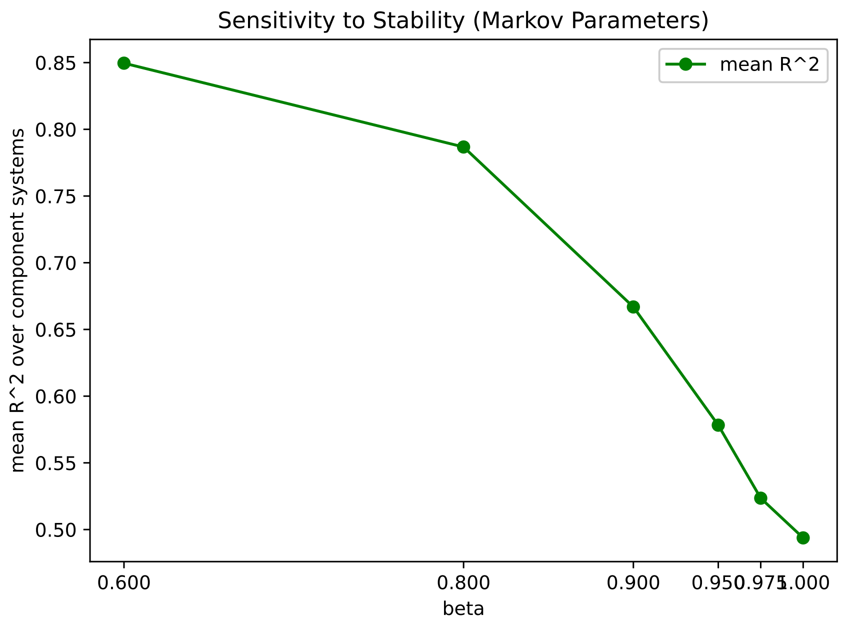
For learning, we used $s = 3$ and 10,000 trajectories of length 30 from each LDS (note that these are much more generous conditions than the 100 trajectories of length 20 with $s = 2$ that we used for learning a single LDS).

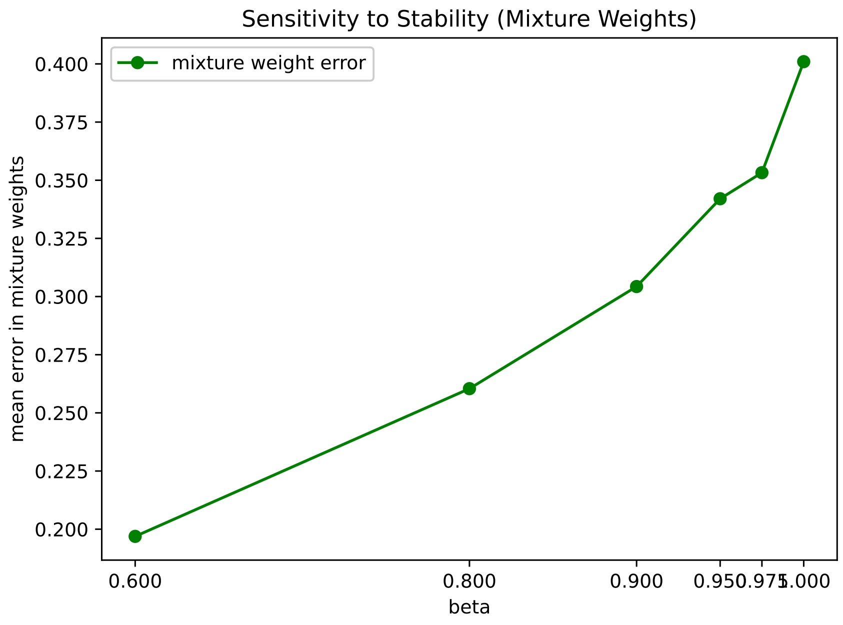
Here are the results for sensitivity to noise, where we vary α and keep $\beta = 0.95$ (we don't show the R^2 value for $\alpha = 3$, since it is significantly less than 0 and makes the graph hard to read):





Here are the results for sensitivity to stability, where we vary β and keep $\alpha = 1$:



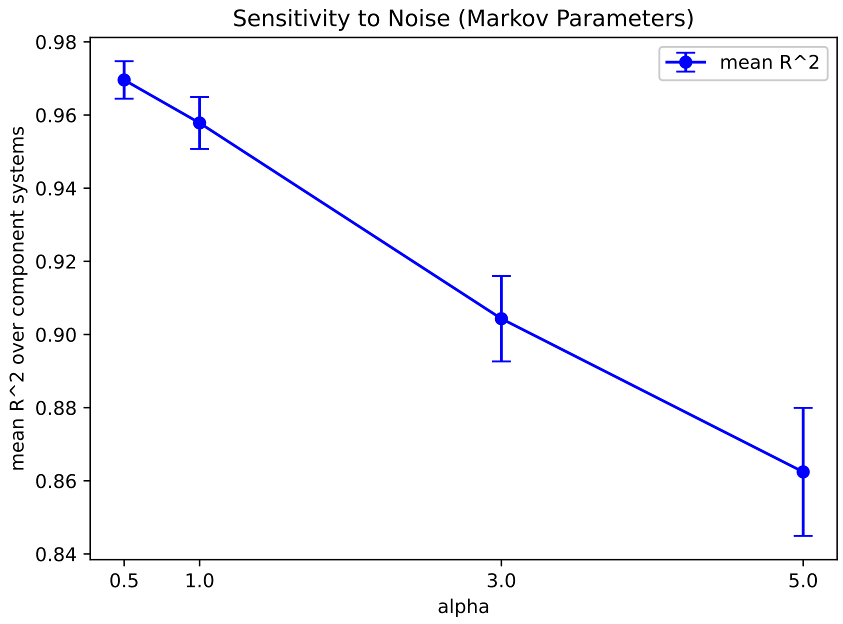


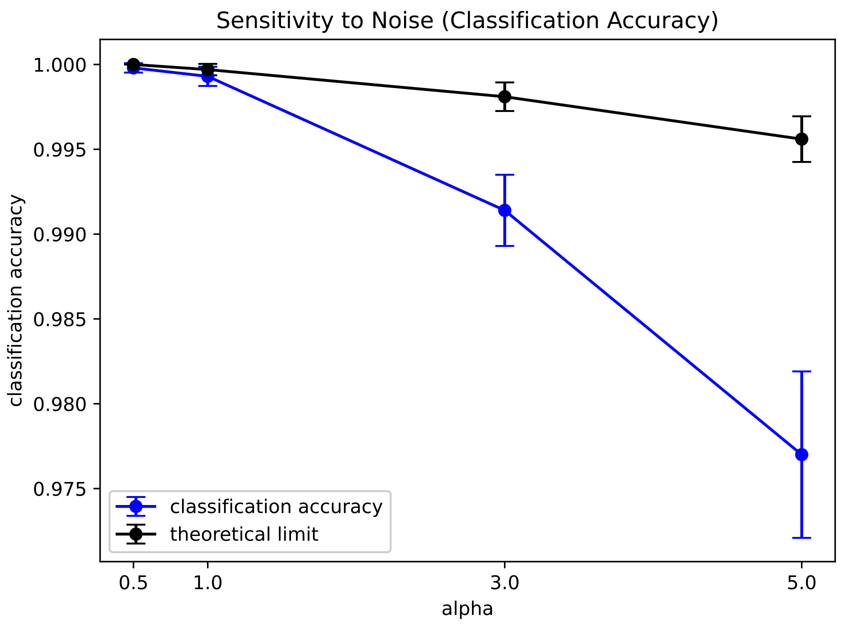
3.4.2 E-M Algorithm Results

Since the E-step step of the E-M algorithm is based on the Regression algorithm, which we have already studied, we are mostly interested in cases where not every trajectory will be perfectly classified eventually. (Otherwise, we are just running the Regression algorithm for multiple LDS's at the end). To this end, we will vary the amount of noise, number of trajectories, and lengths of trajectories (in particular, we will keep $\beta = 1$ throughout).

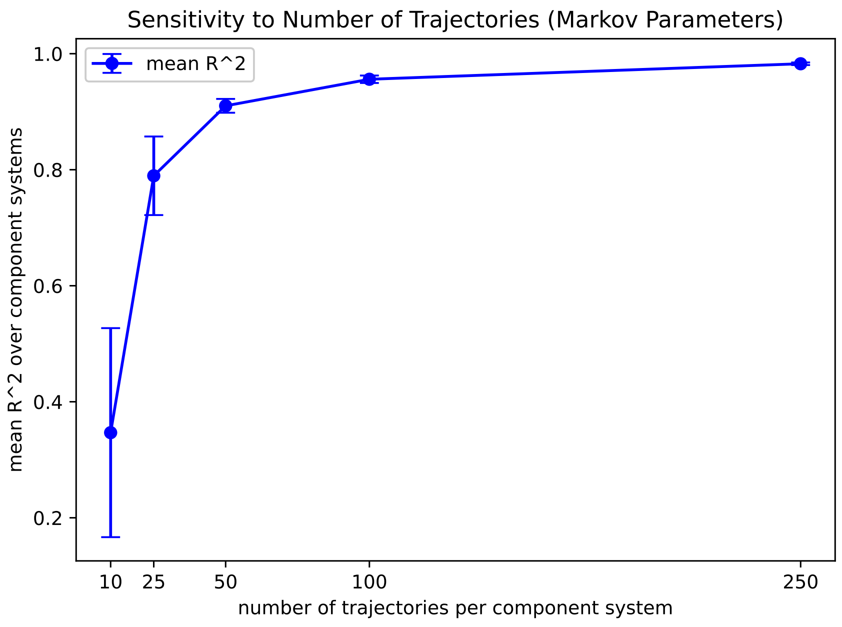
Unless otherwise specified, we used $s = 2$ and 100 trajectories of length 20 from each LDS. Note that these are the same conditions we used when learning a single LDS, and vastly more generous parameters then we used for the Tensor Decomposition algorithm. As before, the error bars represent two standard errors of the mean. For classification error, when the trajectories are very short or the noise is very high, trajectories may be misclassified even if the true mixture is known. This establishes a theoretical limit on the classification accuracy, which we also report.

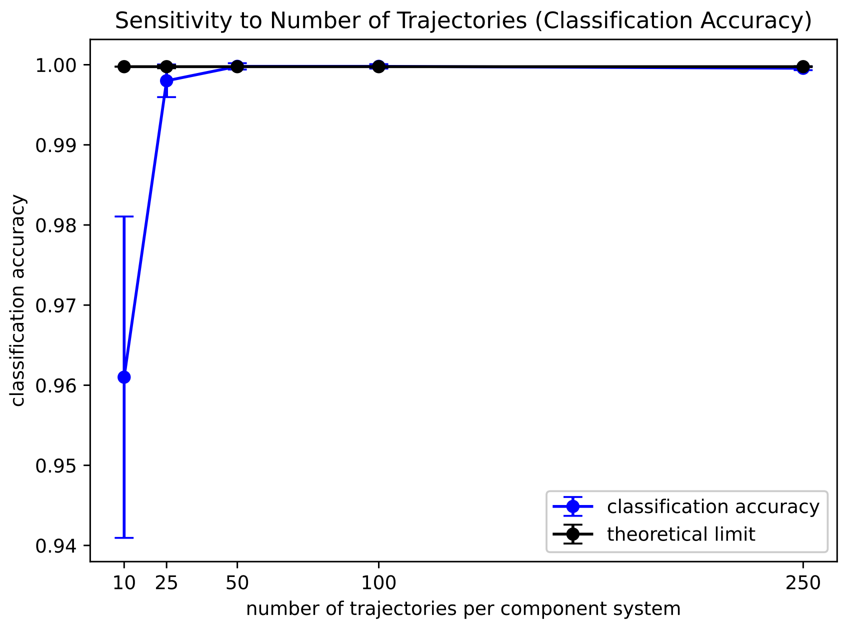
Here are the results for sensitivity to noise, where we vary α:



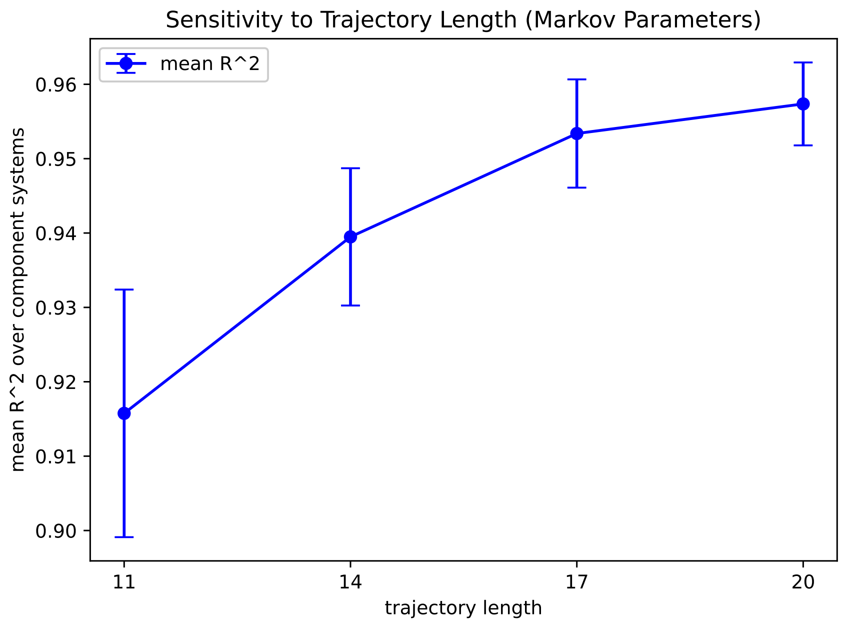


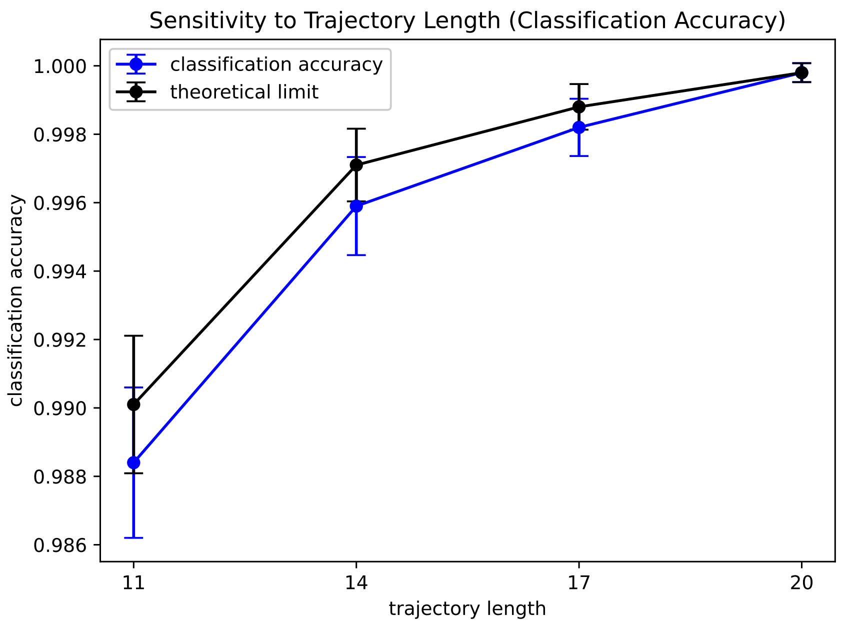
Here are the results for varying the number of trajectories:





Here are the results for varying the length of each trajectory:





3.4.3 Conclusions

The better methods, practically speaking, for learning mixtures of an LDS, were the ones that used lower-order statistics. The E-M algorithm uses the Regression algorithm, which uses only second-order statistics of the trajectories, and performed extremely well. On the other hand, the Tensor Decomposition algorithm, which uses sixth-order statistics of the trajectories, performed much worse, despite theoretical guarantees on the performance. The performance of the Tensor Decomposition algorithm with access to two sets of trajectories with different mixing weights was somewhere in between, and it used fourth-order statistics.

In addition to improved accuracy in recovering the mixture, the E-M algorithm also had a much lower runtime on the same set of inputs.
The E-M algorithm proved to be extremely robust, and we had to intentionally introduce very challenging conditions for learning in order to see cases of failure. For example, when $\alpha = 5$, the noise has more than twice the effect as the input on the hidden state, and the output also has a lot of noise. Despite this, the E-M algorithm is still able to achieve a classification accuracy above 97%, handicapped by using $s = 2$, without a large number of trajectories, and without long trajectories.

Appendix A

Project Code

The full code can be found at https://github.com/nitinjan06/learning_lds_mixtures.

There are a few sections of the code which may be relevant for understanding other code excerpts in the thesis, listed below.

The first is `lds.py`, which defines the LDS class:

```python
from macros import *
from typing import Optional

@dataclass
class LDS:
    A: np.ndarray
    B: np.ndarray
    C: np.ndarray
    D: np.ndarray
    process_cov: Optional[np.ndarray] = None
    output_cov: Optional[np.ndarray] = None

    markov_params: list = field(default_factory=list)

    def get_observability(self, s):
        return np.concatenate([self.C @ mpow(self.A, i) for i in range(s)])

    def get_controllability(self, s):
        return np.concatenate([mpow(self.A, i) @ self.B for i in range(s)],
         ↪  axis = 1)

    def get_s(self):
        # can make this faster
        self.s = 1
        while rank(self.get_observability(self.s)) != self.n: self.s += 1
        while rank(self.get_controllability(self.s)) != self.n: self.s += 1

    def get_markov_param(self, i):
        # can make this faster
        while len(self.markov_params) <= i:
            self.markov_params.append(self.C @ mpow(self.A,
             ↪  len(self.markov_params)-1) @ self.B)

        return self.markov_params[i]

    def __post_init__(self):
        self.n, self.p = self.B.shape
        self.m = self.C.shape[0]
        assert self.A.shape == (self.n, self.n)
        assert self.C.shape == (self.m, self.n)
        assert self.D.shape == (self.m, self.p)

        if self.process_cov is None: self.process_cov = np.eye(self.n)
        if self.output_cov is None: self.output_cov = np.eye(self.m)
        self.markov_params.append(self.D)

        # check individual assumptions
        opA, opB, opC, opD = [op_norm(x) for x in [self.A, self.B, self.C,
         ↪  self.D]]
        # assert 1 <= opB and 1 <= opC
        self.kappa = max(opA, opB, opC, opD)

        self.get_s()

    def generate_trajectory(self, length):
        u, w, z = normal(size=(length, self.p)),
         ↪  multivariate_normal(np.zeros(self.n), self.process_cov,
         ↪  size=length), multivariate_normal(np.zeros(self.m),
         ↪  self.output_cov, size=length)
        x, y = [normal(size=self.n)], []
        for t in range(length):
            y.append(self.C @ x[t] + self.D @ u[t] + z[t])
            x.append(self.A @ x[t] + self.B @ u[t] + w[t])

        return u, np.array(y)
```

```python

    def get_nll(self, trajectory):
        ans = np.double(0)

        u, y = trajectory
        x_mean, x_cov = np.zeros(self.n), np.eye(self.n)
        for t in range(len(u)):
            prefit_resid = y[t] - self.C @ x_mean - self.D @ u[t]
            prefit_cov = self.C @ x_cov @ self.C.T + self.output_cov
            prefit_cov_inv = la.inv(prefit_cov)
            nl_pdf = self.m/2 * np.log(2*np.pi) +
             ↪  np.log(la.det(prefit_cov))/2 + (prefit_resid @
             ↪  prefit_cov_inv @ prefit_resid)/2
            ans += nl_pdf

            kalman_gain = x_cov @ self.C.T @ prefit_cov_inv
            postfit_x_mean = x_mean + kalman_gain @ prefit_resid
            postfit_x_cov = x_cov - kalman_gain @ self.C @ x_cov
            x_mean = self.A @ postfit_x_mean + self.B @ u[t]
            x_cov = self.A @ postfit_x_cov @ self.A.T + self.process_cov

        return ans
```

The second is macros.py, which sets us some imports and shortcuts used throughout the code:

```python
from functools import partial
from dataclasses import dataclass, field
import numpy as np
import numpy.linalg as la
normal = np.random.normal
multivariate_normal = np.random.multivariate_normal
op_norm = partial(la.norm, ord=2)
min_sv = partial(la.norm, ord=-2)
rank = la.matrix_rank
```

```
mpow = la.matrix_power

def get_r2(pred, goal):
    return 1-la.norm(goal - pred)**2/la.norm(goal)**2
```

References

[1] A. Bakshi, A. Liu, A. Moitra, and M. Yau, "Tensor decompositions meet control theory: Learning general mixtures of linear dynamical systems," in *Proceedings of the 40th International Conference on Machine Learning*, 2023, pp. 1549–1563.

[2] Y. Chen and H. V. Poor, "Learning mixtures of linear dynamical systems," in *International Conference on Machine Learning*, 2022, pp. 3507–3557.

[3] A. Bakshi, A. Liu, A. Moitra, and M. Yau, "A new approach to learning linear dynamical systems," in *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, 2023, pp. 335–348.

[4] S. Oymak and N. Ozay, "Non-asymptotic identification of lti systems from a single trajectory," in *2019 American control conference (ACC)*, 2019, pp. 5655–5661.

[5] M. Haugh. "The em algorithm." (2015), URL: https://www.columbia.edu/~mh2078/MachineLearningORFE/EM_Algorithm.pdf.

[6] A. Moitra, *Algorithmic Aspects of Machine Learning*. Cambridge University Press, 2018.

[7] R. Sridharan. "Gaussian mixture models and the em algorithm." (2014), URL: https://people.csail.mit.edu/rameshvs/content/gmm-em.pdf.

[8] I. Reid. "Estimation ii." (2001), URL: https://www.robots.ox.ac.uk/~ian/Teaching/Estimation/LectureNotes2.pdf.