cs146-problem-set-2

December 13, 2023

1 CS146 Problem Set 2: Analytical approximations

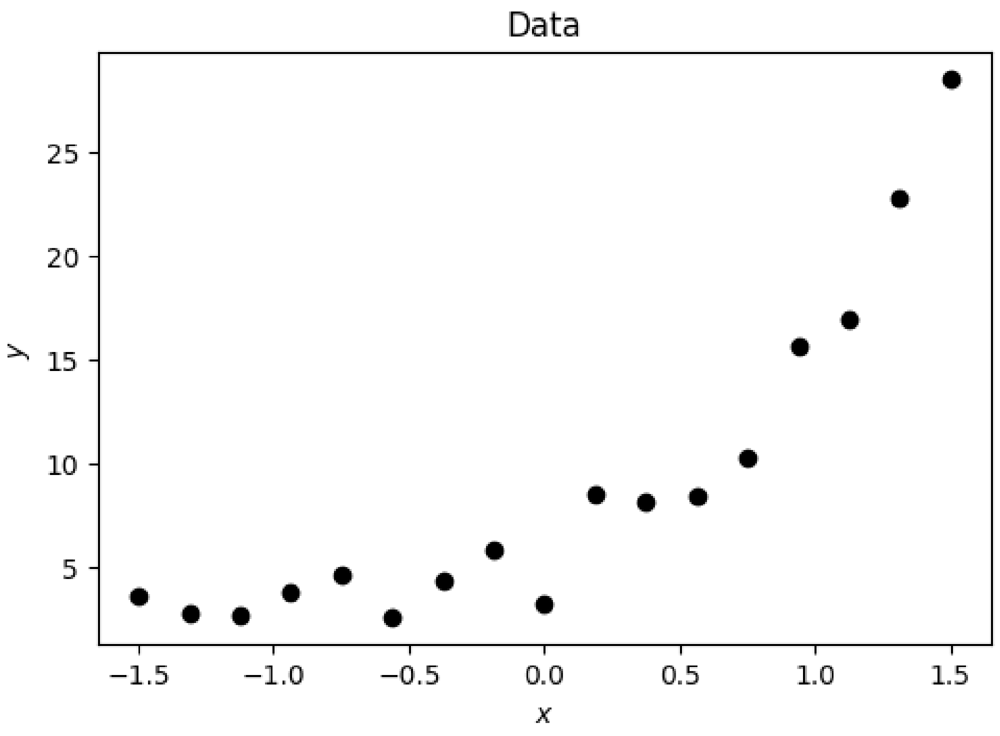
- Complete all the problems below.
- Show and explain your work in detail.
- **Edit and submit this Jupyter notebook.** Do not upload only a PDF. Your instructor needs the notebook to run your code and check that it correctly reproduces your results.
- **Do not use ChatGPT** or similar AI tools in this assignment.
- **Learning outcomes:** #cs146-AnalyticalApproximation, #cs114-ModelSelection, #modeling (optional), #composition, #professionalism

1.1 Overview

In this problem set, we revisit the polynomial linear regression models from Session 10. We fit polynomials of various degrees to the data set below. You might find it helpful to revisit the pre-class and breakout workbooks from that class to review the work we did there. The data set used here is different from the ones used in class.

[ ]:
```
import arviz as az
import matplotlib.pyplot as plt
import numpy as np
import pymc as pm


data_x = np.array([-1.500, -1.312, -1.125, -0.938, -0.750, -0.562, -0.375, -0.
↪188, 0.000, 0.188, 0.375, 0.562, 0.750, 0.938, 1.125, 1.312, 1.500])
data_y = np.array([3.571, 2.777, 2.697, 3.749, 4.669, 2.618, 4.312, 5.833, 3.
↪276, 8.461, 8.108, 8.412, 10.262, 15.631, 16.960, 22.713, 28.491])

plt.figure(figsize=(6, 4))
plt.title('Data')
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.plot(data_x, data_y, 'ko')
plt.show()
```

```
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
```



The polynomial models all have a likelihood function of the form

$$y_i = \text{Normal}(\mu_i, \sigma^2)$$

$$\mu_i = \alpha + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_k x_i^k$$

where k is the degree of the polynomial.

```
[ ]: def make_model(degree):
         # Make a polynomial model with the given degree in PyMC
         with pm.Model() as model:
             # Data
             x = pm.MutableData('x', data_x)
             # Noise scale
             sigma = pm.LogNormal('sigma', mu=0, sigma=1)
             # Constant term
             alpha = pm.Normal('alpha', mu=0, sigma=1)
             # Coefficients of the x^i terms
             beta = pm.Normal('beta', size=degree, mu=0, sigma=1)
             # Compute the mean of the polynomial given the alpha and beta␣
      ↪coefficients
             mu = alpha
```

```
        for i in range(1, degree + 1):
            mu = mu + beta[i-1] * x ** i
        mu = pm.Deterministic('mu', mu)
        # Likelihood
        y = pm.Normal('y', mu=mu, sigma=sigma, observed=data_y)
    return model

#cubic_model = make_model(3)
#pm.model_to_graphviz(cubic_model) (I got an error running the graphviz for␣
 ↪this model in my VS code
# and I was unable to solve it so I commented this out.)
```

1.2 Problem 1: Estimating model evidence

The goal of the activity in Session 10 was to understand how the *model evidence* (the denominator in Bayes' equation, also known as the *marginal likelihood*) helps us to select the best model or models from a collection of models. We used function fitting in PyMC to estimate the model evidence for various polynomial degrees. The code for doing this is provided in Problem 2 below.

Explain why it is possible to use a function-fitting algorithm to estimate the model evidence. You may explain this with reference to the function-fitting algorithm used in PyMC (ADVI) or with reference to the function-fitting algorithm we explored in Sessions 22 and 23 (the Laplace approximation). The answers are somewhat different in those two cases however, you should address the same question in either case — how can we use the algorithm to estimate model evidence?

- Explain why it is possible conceptually.
- Provide the basic mathematical detail for where the model evidence shows up in the function-fitting algorithm.
- (You do not actually have to compute the model evidence here. We will do this in Problem 2.)

YOUR ANSWER GOES HERE **:**

Conceptual Explanation: Model evidence (or marginal likelihood) is computed by integrating the product of the likelihood and the prior over all possible values of the parameters. This integral is difficult or impossible to compute directly, especially for high dimensionsional models. (Wikipedia)

Mathematical detail: Algorithms like ADVI or the Laplace approximation can therefore help us to approximate these difficult integrals in Bayesian inference by transforming the problem at hand into a simpler one. For example, when using the Laplace approximation, we follow the following steps to approximate a posterior distribution that is difficult to compute.

- Parameter Transformation: Transform the model parameters to be unconstrained because the Normal distribution, which we are going to use to approximate the posterior, is defined on the entire real line.

- Find the Mode: Locate the peak of the unnormalized log posterior distribution. This is because the peak of the log posterior corresponds to the peak of the posterior distribution itself.

- Compute Second-order Derivatives: Calculate all the second-order derivatives of the log posterior at the mode. This Hessian matrix of derivatives provides information about the curvature of the log posterior, which informs us about the spread of the posterior around the mode.

- Approximate the Posterior: Use the mode as the mean of the Normal distribution and the inverse of the Hessian matrix as the covariance matrix to form the Normal approximation of the posterior distribution.

Since the peak of the Normal distribution obtained represents the highest point of the posterior and its spread represents the uncertainty or dispersion of the posterior, integrating (summing up) under this distribution gives an approximation of the total probability mass, ie, the model evidence.

Therefore, by finding the Normal approximation of the posterior we take into account the model evidence, allowing for its estimation without necessarily computing it directly.

1.2.1 (Optional) Problem 1b: Compare ADVI and the Laplace approximation

Explain how model evidence is computed in *both* function-fitting algorithms. The two algorithms will give different answers. Highlight the pros and cons of estimating model evidence using each of the algorithms.

This is a challenging problem. Excellent (correct, detailed, well-explained) answers will get an extra on the #cs146-AnalyticalApproximation LO. There is no penalty for not completing this problem or for getting it wrong.

YOUR ANSWER GOES HERE **:**

ADVI (Automatic Differentiation Variational Inference)

Computation: ADVI estimates model evidence using the Evidence Lower Bound (ELBO). The ELBO is not the model evidence itself but a lower bound for the log of the model evidence. The goal is to minimize the Kullback-Leibler (KL) divergence between the approximating distribution q and the target distribution p. This is achieved indirectly by maximizing the Evidence Lower Bound (ELBO), which is a function involving the expected log likelihood of the data under the variational distribution and the entropy of the variational distribution itself. The choice of the variational distribution is important, and it is usually a standard distribution like the Normal in order to make calculation and sampling easy. The computation of the expected log likelihood of the data under the variational distribution, is approximated using sampling methods since it is a complex task. ADVI employs stochastic optimization techniques, like the Adam algorithm, to efficiently maximize the ELBO. By maximizing the ELBO, we're effectively maximizing this lower bound, which brings the approximation closer to the true model evidence. (from notes in session 23)

Pros:

It can handle large datasets and complex models efficiently. It tends to converge faster than traditional MCMC methods.

Cons:

The approximation can be less accurate, especially for posteriors that are not well-represented by the chosen simpler distribution. The model evidence is not computed directly, leading to potential inaccuracies in its estimation.

Laplace Approximation

Computation: The Laplace approximation involves approximating the posterior distribution around its mode (peak) with a Normal distribution. The curvature around the mode (defined by the Hessian matrix) is used in this approximation. The model evidence here is estimated indirectly by evaluating the fit of this approximation to the true posterior as explained in my answer above in detail.

Pros:

It is straightforward and easy to implement with few parameters. It is accurate for posteriors that are unimodal, approximately normal and not too skewed.

Cons:

In high deimensional models, it becomes computationally costly since we have to calculate the Hessian It may be less accurate hence give a bad approximation for multimodal posteriors or those that are highly non-Normally distributed.

1.3 Problem 2: Was using mean-field a good idea?

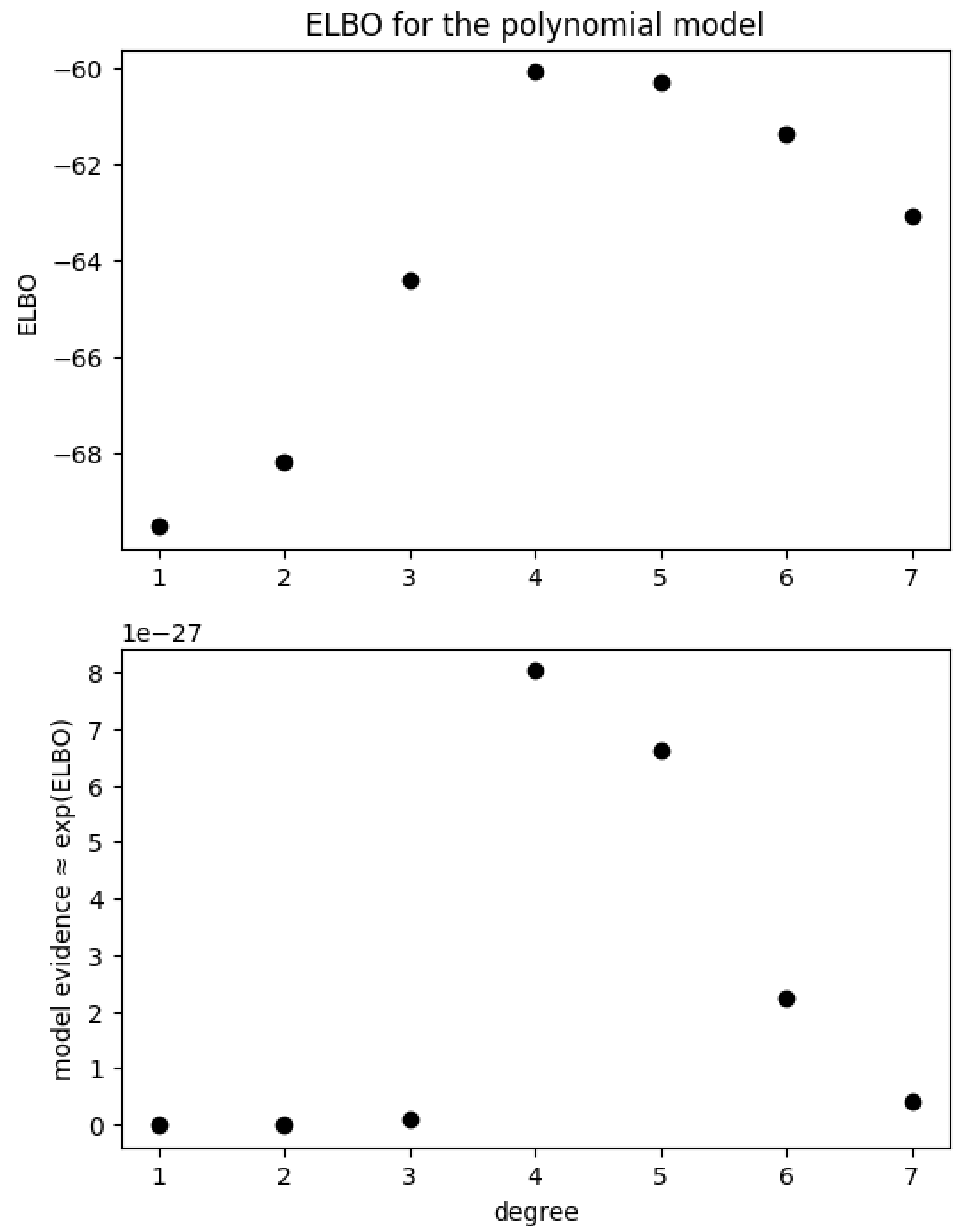
The code below was used to determine the best degree of a polynomial for the data set above. Should it be linear, quadratic, etc.?

In Session 23, we learned that using a mean-field approximation is not necessarily a good idea when fitting a Multivariate Normal to a posterior since it cannot represent correlations between variables. PyMC uses the mean-field approximation by default.

Your task is to analyze the impact of using the mean-field approximation *versus* the full-rank approximation.

[ ]:
```
# Store the PyMC models, the function fits (approximations), and estimates
# of the ELBO produced by the function fitting algorithm
models = []
approximations = []
elbos = []

# Create the models and compute the ELBOs for polynomial degrees 1-7
degrees = range(1, 8)
for degree in degrees:
    model = make_model(degree)
    with model:
        approx = pm.fit(100000, progressbar=False)
    models.append(model)
    approximations.append(approx)
    elbos.append(-np.mean(approx.hist[-10000:]))

# Plot the results
plt.figure(figsize=(6, 8))
plt.subplot(2, 1, 1)
plt.title('ELBO for the polynomial model')
plt.plot(degrees, elbos, 'ko')
plt.ylabel('ELBO')
plt.subplot(2, 1, 2)
plt.plot(degrees, np.exp(elbos), 'ko')
plt.xlabel('degree')
plt.ylabel('model evidence   exp(ELBO)')
plt.show()
```

```
Finished [100%]: Average Loss = 69.548
Finished [100%]: Average Loss = 68.156
Finished [100%]: Average Loss = 64.119
Finished [100%]: Average Loss = 60.107
Finished [100%]: Average Loss = 60.321
Finished [100%]: Average Loss = 61.453
Finished [100%]: Average Loss = 63.036
```



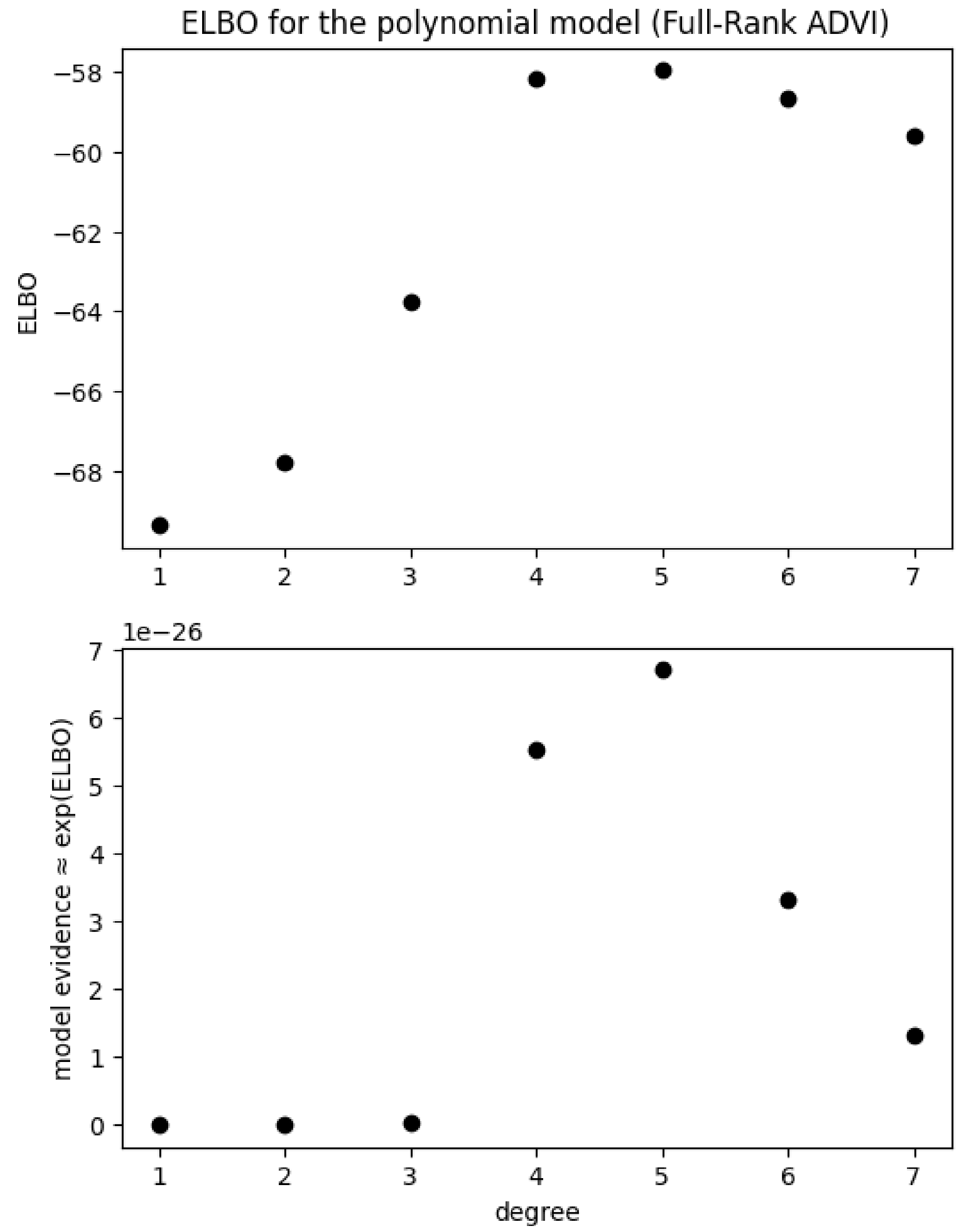
Do all of the following.

- Rerun this code using the full-rank ADVI algorithm and record the evidence lower bound (ELBO) values. The ELBO values are approximations of the log of the model evidence.
- How much worse is the mean-field approximation than the full-rank approximation? Quantify your answer to this question and explain your work.
- Do any of our conclusions change about which degrees are better or worse than others?

Re-running the code using full-rank ADVI The full-rank ADVI considers the correlations between variables, which can be important for more accurate modeling, especially in cases where the variables are not independent.

[ ]:
```
# Running full rank
# Store the models, approximations, and ELBOs
models_full_rank = []
approximations_full_rank = []
elbos_full_rank = []

# Create the models and compute the ELBOs for polynomial degrees 1-7 using␣
 ↪full-rank ADVI
degrees = range(1, 8)
for degree in degrees:
    model = make_model(degree)
    with model:
        # Use full-rank ADVI
        approx = pm.fit(100000, method='fullrank_advi', progressbar=False)
    models_full_rank.append(model)
    approximations_full_rank.append(approx)
    elbos_full_rank.append(-np.mean(approx.hist[-10000:]))

# Plot the results
plt.figure(figsize=(6, 8))
plt.subplot(2, 1, 1)
plt.title('ELBO for the polynomial model (Full-Rank ADVI)')
plt.plot(degrees, elbos_full_rank, 'ko')
plt.ylabel('ELBO')
plt.subplot(2, 1, 2)
plt.plot(degrees, np.exp(elbos_full_rank), 'ko')
plt.xlabel('degree')
plt.ylabel('model evidence   exp(ELBO)')
plt.show()
```

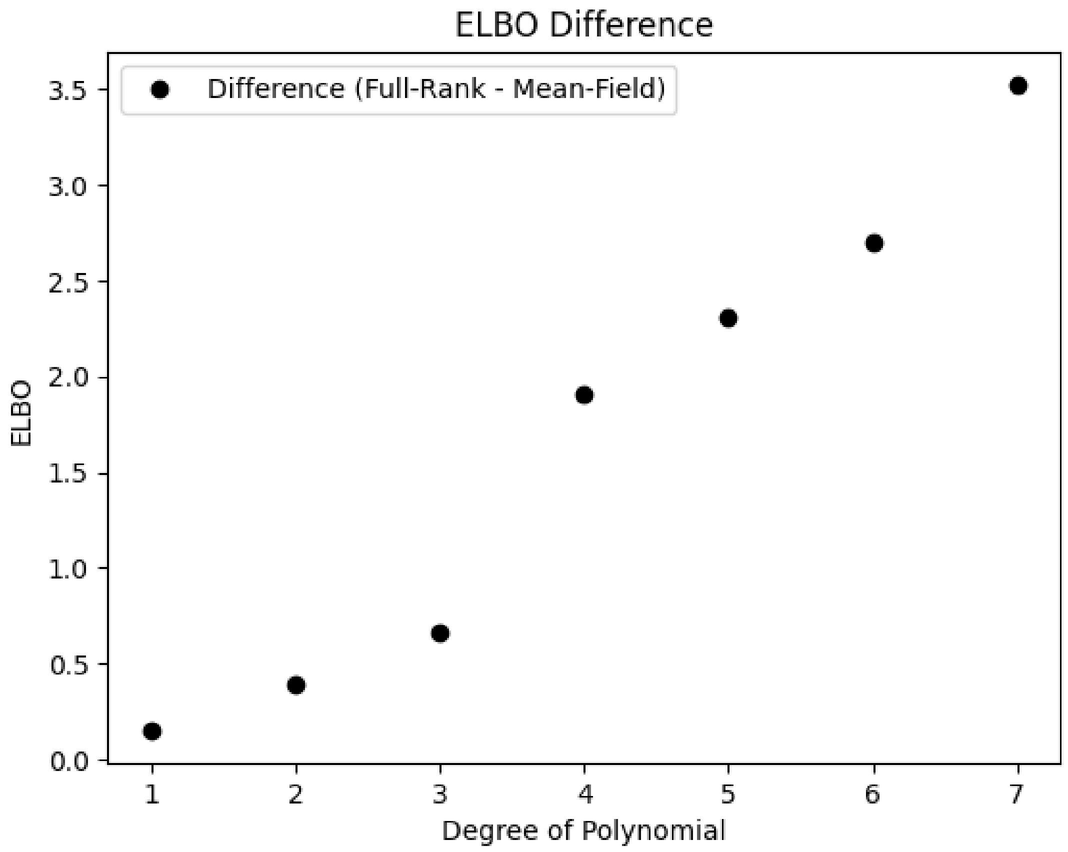
```
Finished [100%]: Average Loss = 69.344
Finished [100%]: Average Loss = 67.773
Finished [100%]: Average Loss = 63.737
Finished [100%]: Average Loss = 58.075
Finished [100%]: Average Loss = 57.953
Finished [100%]: Average Loss = 58.714
Finished [100%]: Average Loss = 59.594
```

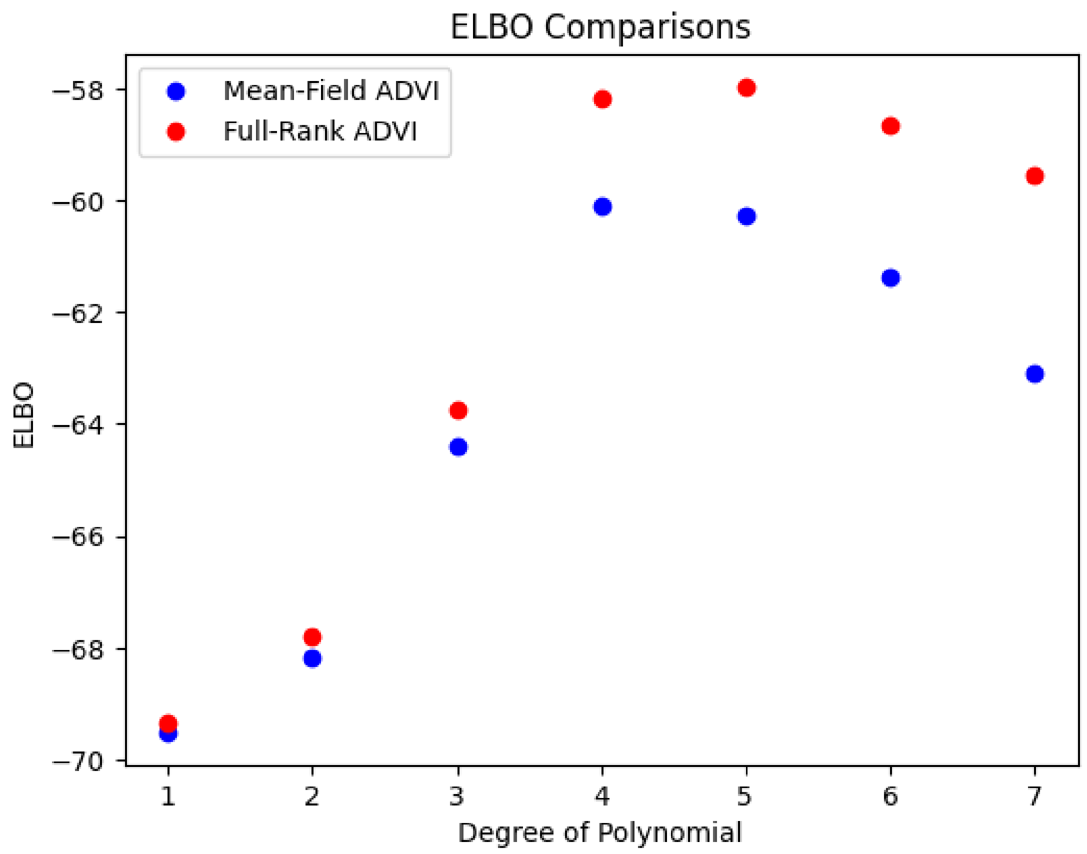


Quantifying how much worse the mean-field approximation is compared to the full-rank approximation. To do this we need to compare the evidence lower bound (ELBO) values obtained for each polynomial degree. The ELBO, an approximation of the logarithm of the model evidence, indicates the quality of the posterior approximation. By comparing ELBO values, we can determine which approximation fits the data better. We'll calculate the differences in ELBO values for each degree, with positive differences suggesting a better fit by the full-rank approximation. Additionally, computing the average difference across all degrees will provide an overall performance comparison. A plot of both ELBO values can tell us if one method consistently outperforms the other or if performance varies with model complexity.

```
[ ]: # Calculate differences
elbo_differences = np.array(elbos_full_rank) - np.array(elbos)

print('Average ELBO difference is:', np.mean(elbo_differences))

# Plotting
plt.figure()
plt.subplot()
degrees = np.arange(1, 8)
plt.plot(degrees, elbo_differences, 'ko', label='Difference (Full-Rank -␣
  ↪Mean-Field)')
plt.xlabel('Degree of Polynomial')
plt.ylabel('ELBO')
plt.title('ELBO Difference')
plt.legend()
plt.show()

plt.subplot()
degrees = np.arange(1, 8)
plt.plot(degrees, elbos, 'o', label='Mean-Field ADVI', color = 'blue')
plt.plot(degrees, elbos_full_rank, 'o', label='Full-Rank ADVI', color = 'red')
plt.xlabel('Degree of Polynomial')
plt.ylabel('ELBO')
plt.title('ELBO Comparisons')
plt.legend()
plt.show()
```

```
Average ELBO difference is: 1.664526863040507
```


ELBO Difference
Difference (Full-Rank - Mean-Field)
ELBO
Degree of Polynomial
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
1
2
3
4
5
6
7

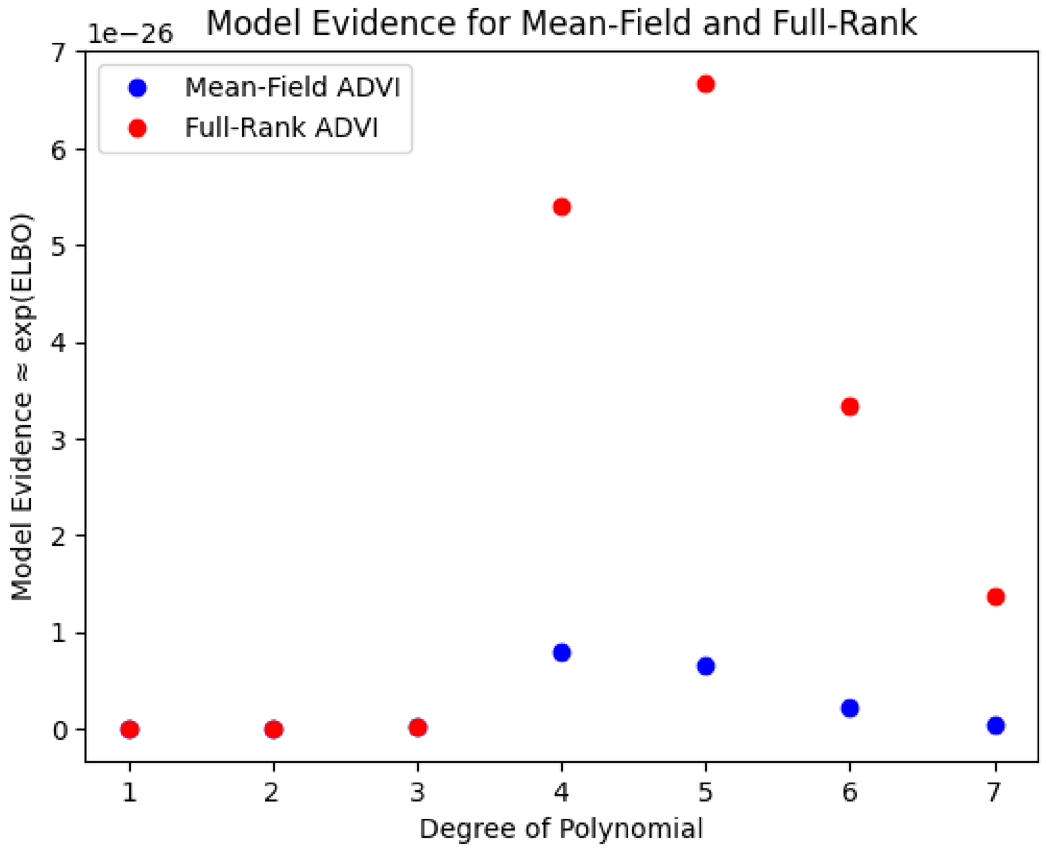


From the first plot, the positive and increasing trend in the difference between the full-rank and mean-field ELBO values suggests that the full-rank approximation provides a better fit than the mean-field approximation, especially as the degree of the polynomial increases.

The second plot, showing the ELBO values for each approximation across polynomial degrees, indicates that both methods improve their fit up to a certain polynomial degree, after which the ELBO value decreases. The highest ELBO values for both methods are around the 4th degree, which suggests this is the most appropriate model complexity given the trade-off between fit and overfitting. The average ELBO difference of 1.645 is however small and at lower degrees, the difference is even smaller. The mean-field approximation therefore doesn't perform that much worse compared to the full-rank. If computational cost is a major factor, we might prefer to use the mean-field. However, if accuracy is more important even by the slightest increase or decrease in ELBO, we might prefer full-rank.

Do any of our conclusions change about which degrees are better or worse than others? To determine if the conclusions about which polynomial degrees are better or worse change when using full-rank ADVI instead of mean-field ADVI, we should compare the model evidence (approximated by exp(ELBO) as shown in the code in problem 2) for each degree under both approximation methods.

```
[ ]: # Calculate model evidence (exp(ELBO))
evid_mf = np.exp(elbos)
evid_fr = np.exp(elbos_full_rank)

# Plotting model evidence
plt.figure()
degrees = np.arange(1, 8)
plt.plot(degrees, evid_mf, 'o', label='Mean-Field ADVI', color = 'blue')
plt.plot(degrees, evid_fr, 'o', label='Full-Rank ADVI', color = 'red')
plt.xlabel('Degree of Polynomial')
plt.ylabel('Model Evidence   exp(ELBO)')
plt.title('Model Evidence for Mean-Field and Full-Rank')
plt.legend()
plt.show()
```



From the plot above, we can see that our conclusions about which degrees are better or worse than others, changes depending on whether we use mean-field or full-rank. In mean-field, the best degree is 4 but in full-rank, the degree with the highest model evidence is 5.

1.4 (Optional) Problem 3. Discrete Fourier Transform basis functions

Instead of fitting polynomial basis functions to the given data set, we fit trigonometric basis functions.

A polynomial function of degree n can be written as a sum of monomial basis functions

$$f(x) = \sum_{j=0}^{n} c_j x^j$$

We can also write a function as a sum of Discrete Fourier Transform (DFT) basis functions

$$f(x) = \sum_{j=0}^{n-1} a_j \sin(jx) + b_j \cos(jx)$$

where j is the frequency of a basis function.

Background information: The DFT is used extensively in digital signal processing. For example, audio MP3s, JPEG images, and various video formats are compressed using the DFT as part of the compression algorithm. Usually, the DFT takes n equal to the number of samples in the digital signal. However, we can approximate a signal by reducing n — that is, by using only the first n frequencies in the signal.

Tasks:

- Use the same data set we used for polynomial linear regression above.
- Read up as much as you need to about the DFT so you can explain your work clearly in this problem. (Feel free to use ChatGPT or other AI tools to speed up your research. Do not use these tools to write your answer!)
- Reuse and modify the code for polynomial linear regression to use the DFT basis functions instead.
- Compare the mean-field and full-rank approximations to the model evidence to identify a good value (or values) for n.
- Plot some samples from the posterior distribution on top of the data to show that the samples fit the data.
- What happens if we use the posterior samples to predict y-values for x-values outside the data range? Explain the behavior of the predictions with reference to the properties of the DFT basis functions.

This is a challenging problem. Excellent (correct, detailed, well-explained) answers will get a on the #modeling LO. There is no penalty for not completing this problem or for getting it wrong.

Using DFT Basis functions To modify our code to use DFT basis functions instead of polynomial linear regression, we implement the following steps.

- Set priors for the DFT coefficients which are the amplitudes for sine and cosine terms as normal distributions with mean zero and a standard deviation of one, which means we have no preference for specific frequencies.

- Defined a noise scale using a log-normal prior for the standard deviation parameter (sigma), which assumes that noise levels are positive and can vary over orders of magnitude. We use log-normal distribution since this value cannot be negative.
- Constructed the mean function by summing sine and cosine functions, with frequencies ranging from zero up to n—1, which are weighted by their respective coefficients from the priors. This represents our observed variable as a combination of oscillations altering between sin and cosine.

In this way, we are basically saying that our data is periodic in nature, and we are assuming that the underlying process generating the data can be represented by a sum of sinusoidal functions, whereby each one has its own amplitude and phase which is defined by the sine and cosine coefficients. We can define it as a function so as to use different n/ evaluate different values of n which is the first n frequencies in the signal.

[ ]:
```python
# Function to make model with DFT basis functions
def my_dft_model(n):
    with pm.Model() as model:
        # Data
        x = pm.MutableData('x', data_x)

        # Noise scale
        sigma = pm.LogNormal('sigma', mu=0, sigma=1)

        # DFT coefficients
        a = pm.Normal('a', mu=0, sigma=1, shape=n)  # Sine coefficients
        b = pm.Normal('b', mu=0, sigma=1, shape=n)  # Cosine coefficients

        # The mean function using DFT basis functions
        mu = pm.math.zeros_like(data_x)
        for j in range(n):
            mu += a[j] * pm.math.sin(j * x) + b[j] * pm.math.cos(j * x)

        # The observed variable
        y = pm.Normal('y', mu=mu, sigma=sigma, observed=data_y)

    return model
```

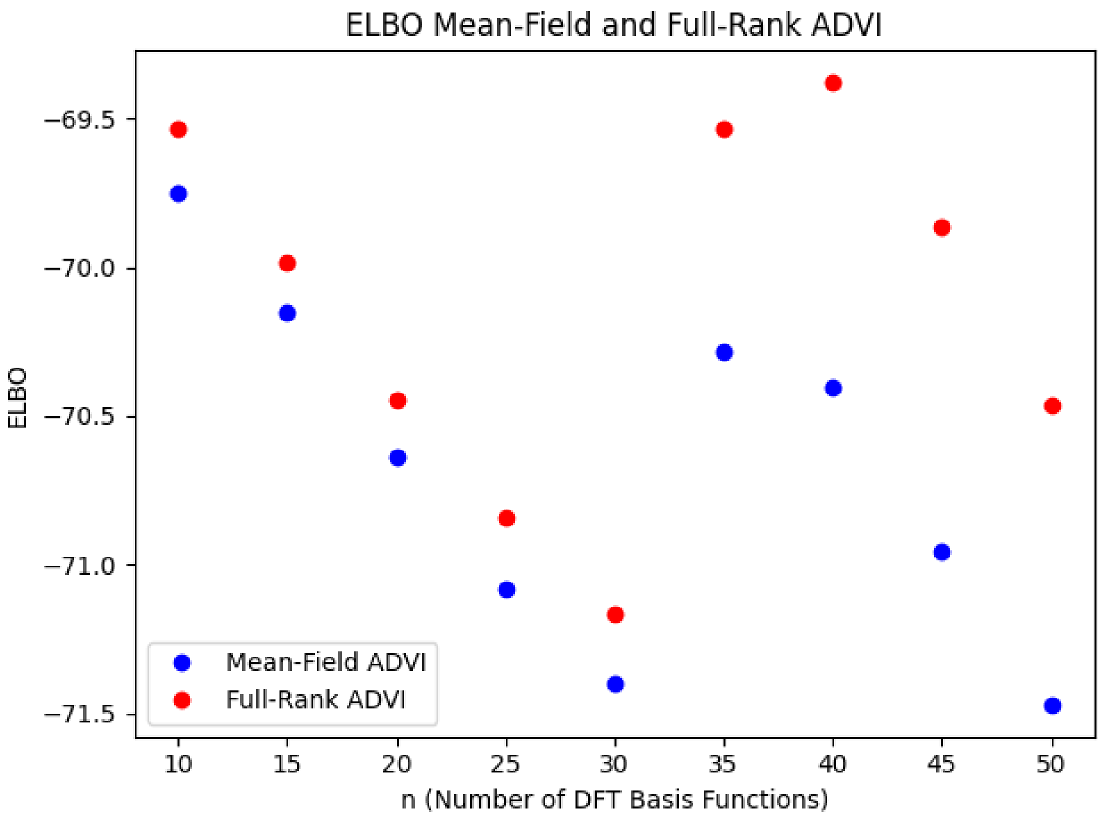
Comparing the mean-field and full-rank approximations to the model evidence to identify a good value (or values) for n ELBO Values

Just like in the problems above, we begin by computing and comparing ELBO values for both mean-field and full-rank.

[ ]:
```python
# WARNING: This cell takes really long to run
# Function to compute ELBOs
def compute_elbos(n_values, method):
    elbos = []
    for n in n_values:
        model = my_dft_model(n)
        with model:
            approx = pm.fit(100000, method=method, progressbar=False)
        elbos.append(-np.mean(approx.hist[-10000:]))
    return elbos

# Range of n values to test (minimum of 10 functions and max of 50)
# below 10 wasn't good, and code broke above 50
# and difference of 5 is used because difference of 1 takes too long to run
n_values = [10, 15, 20, 25, 30, 35, 40, 45, 50]

# Compute ELBOs for mean-field and full-rank
elbos_mf_dft = compute_elbos(n_values, 'advi')
elbos_fr_dft = compute_elbos(n_values, 'fullrank_advi')
```

```
Finished [100%]: Average Loss = 69.748
Finished [100%]: Average Loss = 70.143
Finished [100%]: Average Loss = 70.602
Finished [100%]: Average Loss = 71.133
Finished [100%]: Average Loss = 71.429
Finished [100%]: Average Loss = 70.277
Finished [100%]: Average Loss = 70.35
Finished [100%]: Average Loss = 70.991
Finished [100%]: Average Loss = 71.412
Finished [100%]: Average Loss = 69.561
Finished [100%]: Average Loss = 69.981
Finished [100%]: Average Loss = 70.425
Finished [100%]: Average Loss = 70.816
Finished [100%]: Average Loss = 71.106
Finished [100%]: Average Loss = 69.539
Finished [100%]: Average Loss = 69.388
Finished [100%]: Average Loss = 69.816
Finished [100%]: Average Loss = 70.485
```

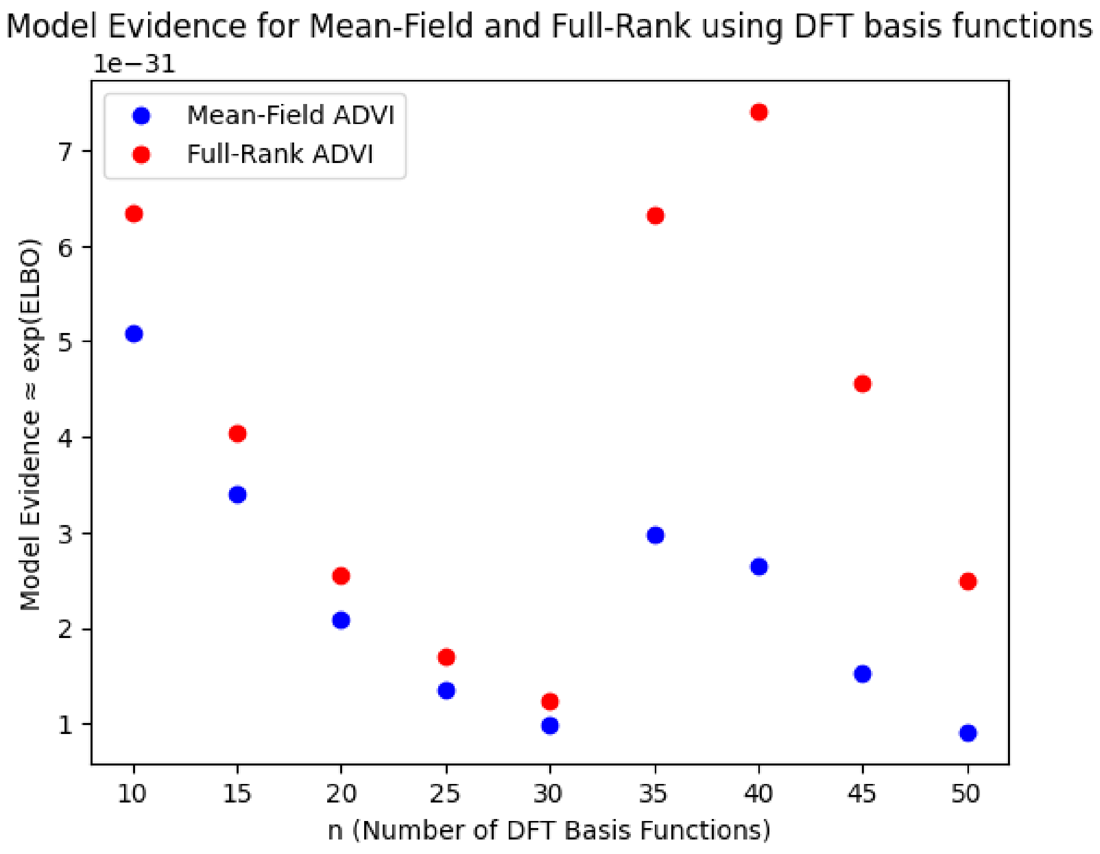
[ ]:
```
# Plot
plt.figure()
plt.plot(n_values, elbos_mf_dft, 'o', label='Mean-Field ADVI', color = 'blue')
plt.plot(n_values, elbos_fr_dft, 'o', label='Full-Rank ADVI', color = 'red')
plt.xlabel('n (Number of DFT Basis Functions)')
plt.ylabel('ELBO')
plt.title('ELBO Mean-Field and Full-Rank ADVI')
plt.legend()
plt.tight_layout()
plt.show()
```



Model Evidence

We then use exp(ELBO) to approximate the model evidence and compare for both mean-field and full-rank to determine a good value for n.

```
[ ]: # Calculate model evidence (exp(ELBO))
evid_mf_dft = np.exp(elbos_mf_dft)
evid_fr_dft = np.exp(elbos_fr_dft)

# Plotting model evidence
plt.figure()
plt.plot(n_values, evid_mf_dft, 'o', label='Mean-Field ADVI', color = 'blue')
plt.plot(n_values, evid_fr_dft, 'o', label='Full-Rank ADVI', color = 'red')
plt.xlabel('n (Number of DFT Basis Functions)')
plt.ylabel('Model Evidence   exp(ELBO)')
plt.title('Model Evidence for Mean-Field and Full-Rank using DFT basis␣
  ↪functions')
plt.legend()
plt.show()
```



From the plot above, we can see that a good value of n is 10 using mean field and 40 using full-rank. Although, values of n at 10, and 35 also seem to be good values when using full-rank. The differences in model evidence for n values above 30 are really large and we can infer that the full-rank adds a lot to the model which we miss out on when we use mean-field.

I however chose to use an n value of 30 in the next section to compare both mean-field and full-rank posterior predictives because any values larger than 30 caused sampler issues and led to a lot of divergence and bad r-hat values and low n values, for example 10 led to a bad fitting of the original data.

Plotting some samples from the posterior predictive distribution on top of the data
The posterior predictive distribution refers to the distribution of possible unobserved values predicted by the model, given the observed data. (Matibe, 2023) We can therefore plot some samples from the posterior predictive distribution on top of the data to show that the samples fit the data.

[ ]:
```
# mean-field
n = 30 #chosen n value for mean-field
dft_model = my_dft_model(n)

with dft_model:
    pm.fit(100000, method='advi', progressbar=False)
```

```
# Sample from the posterior
    trace_mf = pm.sample(1000, tune=2000, target_accept=0.99)

# Posterior predictive
with dft_model:
    ppc_mf = pm.sample_posterior_predictive(trace_mf)
```

```
Finished [100%]: Average Loss = 71.376
Auto-assigning NUTS sampler…
Initializing NUTS using jitter+adapt_diag…
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sigma, a, b]
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Sampling 4 chains for 2_000 tune and 1_000 draw iterations (8_000 + 4_000 draws
total) took 143 seconds.
/Users/macbethmatibe/opt/anaconda3/lib/python3.8/site-
packages/arviz/utils.py:184: NumbaDeprecationWarning: The 'nopython' keyword

argument was not supplied to the 'numba.jit' decorator. The implicit default

value for this argument is currently False, but it will be changed to True in

Numba 0.59.0. See

https://numba.readthedocs.io/en/stable/reference/deprecation.html#deprecation-

of-object-mode-fall-back-behaviour-when-using-jit for details.
  numba_fn = numba.jit(**self.kwargs)(self.function)
Sampling: [y]

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

[ ]:
```
#full-rank
n2 = 30 #chosen n value for mean-field
dft_model2 = my_dft_model(n2)

with dft_model2:
    pm.fit(100000, method='fullrank_advi', progressbar=False)

# Sample from the posterior
    trace_fr = pm.sample(1000, tune=2000, target_accept=0.99)

# Posterior predictive
with dft_model2:
    ppc_fr = pm.sample_posterior_predictive(trace_fr)
```

```
Finished [100%]: Average Loss = 71.106
Auto-assigning NUTS sampler…
Initializing NUTS using jitter+adapt_diag…
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sigma, a, b]
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

Sampling 4 chains for 2_000 tune and 1_000 draw iterations (8_000 + 4_000 draws
total) took 144 seconds.
Sampling: [y]

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

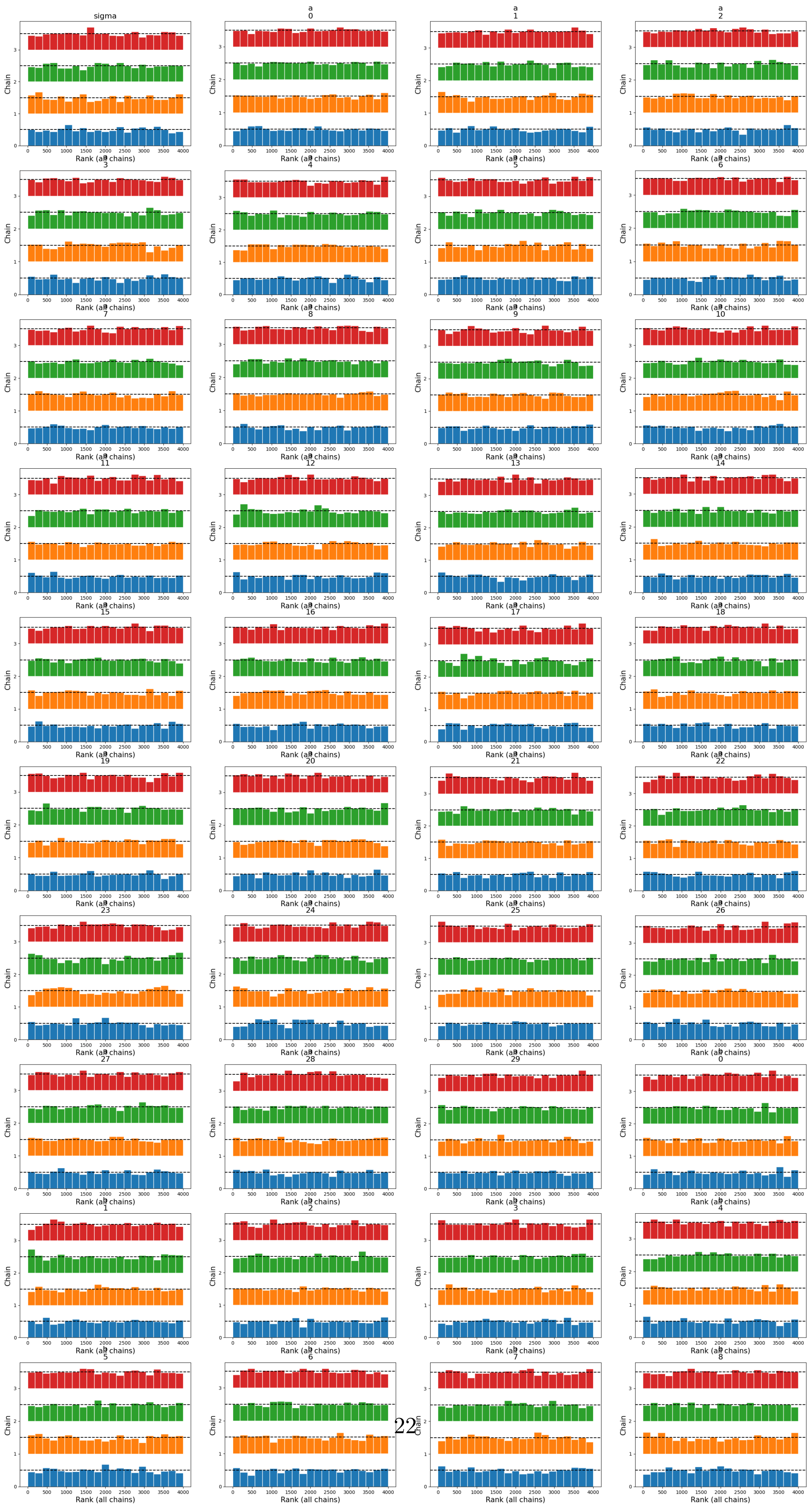
Checking that the Sampler works

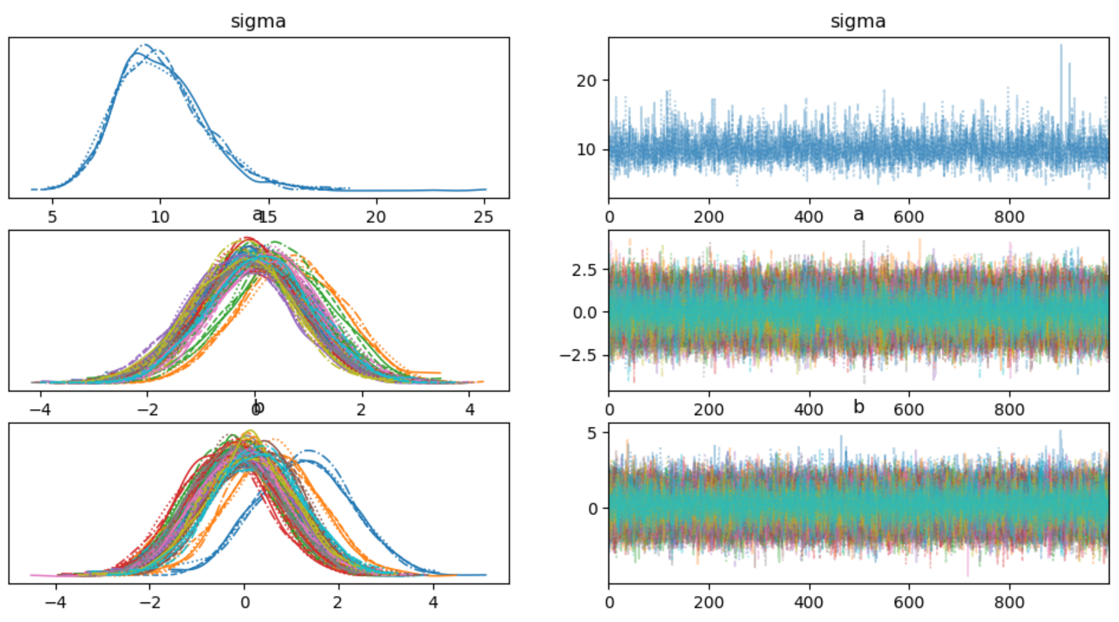
```
[ ]: # Ensuring the sampler works - mean-field
az.plot_rank(trace_mf, var_names=['sigma', 'a', 'b'])
az.plot_trace(trace_mf, var_names=['sigma', 'a', 'b'])
az.summary(trace_mf, var_names=['sigma', 'a', 'b'])
```

```
/Users/macbethmatibe/opt/anaconda3/lib/python3.8/site-
packages/arviz/plots/plot_utils.py:271: UserWarning:
rcParams['plot.max_subplots'] (40) is smaller than the number of variables to
plot (61) in plot_rank, generating only 40 plots
  warnings.warn(
/Users/macbethmatibe/opt/anaconda3/lib/python3.8/site-
```

```
packages/arviz/utils.py:184: NumbaDeprecationWarning: The 'nopython' keyword
argument was not supplied to the 'numba.jit' decorator. The implicit default
value for this argument is currently False, but it will be changed to True in
Numba 0.59.0. See
https://numba.readthedocs.io/en/stable/reference/deprecation.html#deprecation-
of-object-mode-fall-back-behaviour-when-using-jit for details.
  numba_fn = numba.jit(**self.kwargs)(self.function)
```

[ ]:
```
          mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  \
sigma   10.032  2.066   6.445   13.901      0.032    0.023    4253.0    3337.0
a[0]     0.005  1.018  -1.831    2.002      0.010    0.020    9713.0    2572.0
a[1]     0.673  0.981  -1.149    2.499      0.010    0.013    9245.0    3327.0
a[2]     0.465  0.960  -1.413    2.221      0.009    0.015   10315.0    2874.0
a[3]    -0.186  0.967  -2.116    1.550      0.011    0.017    8205.0    2635.0
…          …    …      …        …          …        …         …
b[25]    0.183  0.962  -1.509    2.048      0.011    0.017    8426.0    2894.0
b[26]   -0.070  0.973  -1.954    1.689      0.009    0.019   10495.0    2612.0
b[27]   -0.195  0.943  -1.967    1.533      0.009    0.017   10374.0    2651.0
b[28]    0.132  0.949  -1.675    1.865      0.009    0.017   10072.0    3208.0
b[29]    0.296  0.950  -1.452    2.129      0.010    0.017    9179.0    2875.0

       r_hat
sigma    1.0
a[0]     1.0
a[1]     1.0
a[2]     1.0
a[3]     1.0
…       …
b[25]    1.0
b[26]    1.0
b[27]    1.0
b[28]    1.0
b[29]    1.0

[61 rows x 9 columns]
```



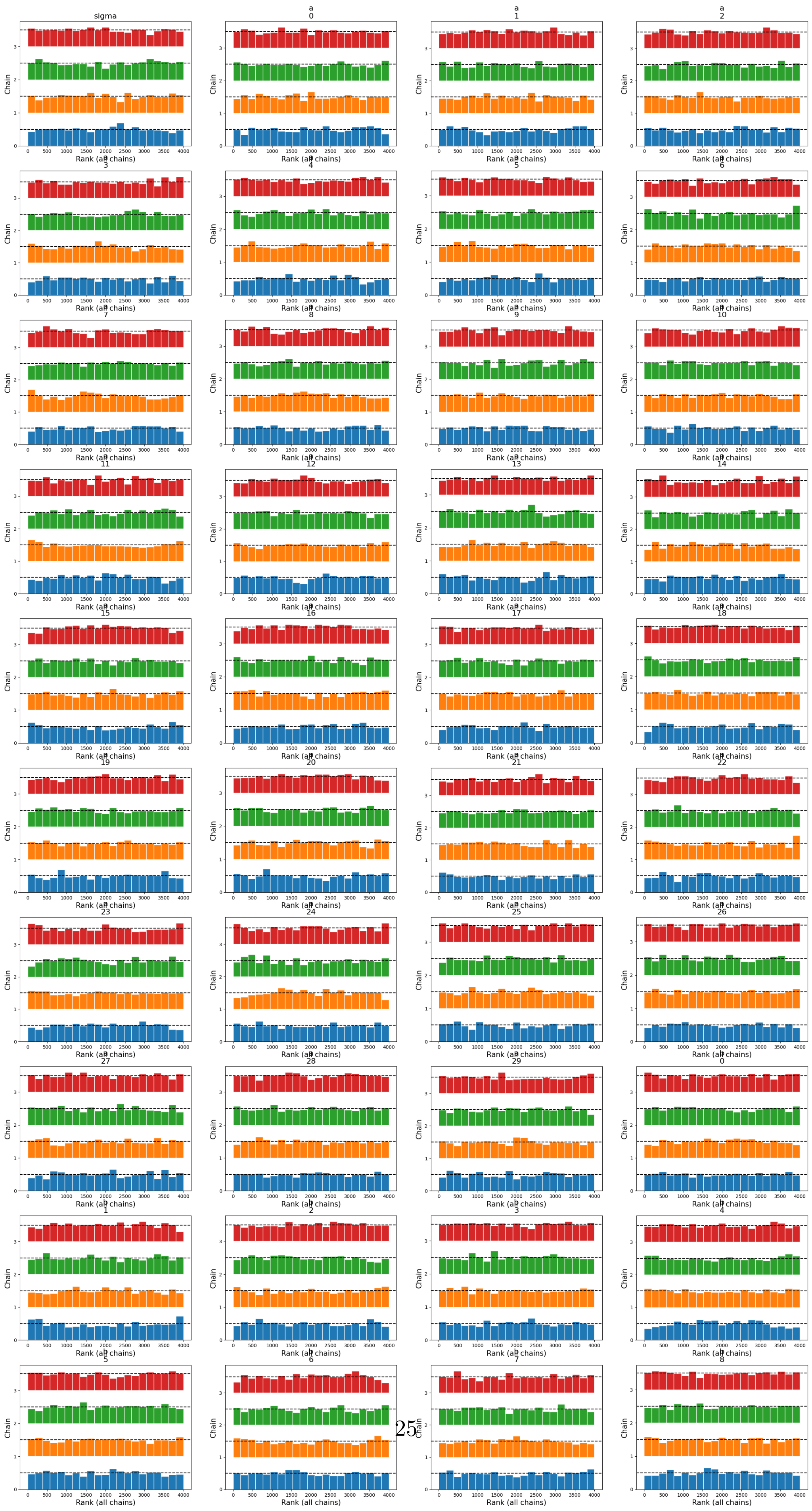


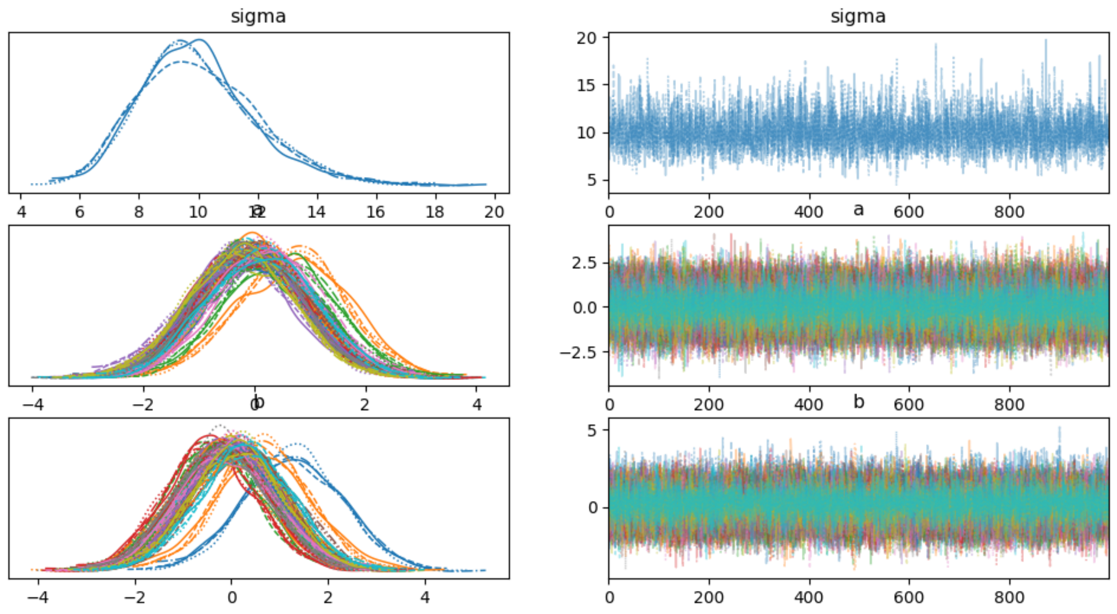
```
[ ]: # Ensuring the sampler works - full-rank
     az.plot_rank(trace_fr, var_names=['sigma', 'a', 'b'])
     az.plot_trace(trace_fr, var_names=['sigma', 'a', 'b'])
     az.summary(trace_fr, var_names=['sigma', 'a', 'b'])
```

```
/Users/macbethmatibe/opt/anaconda3/lib/python3.8/site-
packages/arviz/plots/plot_utils.py:271: UserWarning:
rcParams['plot.max_subplots'] (40) is smaller than the number of variables to
plot (61) in plot_rank, generating only 40 plots
  warnings.warn(
```

```
[ ]:          mean     sd  hdi_3%  hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tail  \
     sigma   9.988  1.988   6.622   13.896      0.029    0.021    4707.0    3401.0
     a[0]    0.004  1.040  -1.957    1.947      0.011    0.020    9144.0    2858.0
     a[1]    0.696  0.983  -1.114    2.584      0.010    0.013    8876.0    2865.0
     a[2]    0.462  0.983  -1.313    2.333      0.010    0.015    8796.0    2734.0
     a[3]   -0.192  0.960  -1.975    1.604      0.010    0.017    8756.0    2817.0
     …        …      …       …        …         …         …        …
     b[25]   0.169  0.944  -1.654    1.892      0.010    0.016    9104.0    3048.0
     b[26]  -0.101  0.985  -1.912    1.753      0.010    0.019    8885.0    2758.0
     b[27]  -0.211  0.982  -2.011    1.622      0.010    0.018    9404.0    2803.0
     b[28]   0.133  0.962  -1.626    2.018      0.010    0.017    9412.0    2575.0
     b[29]   0.296  0.968  -1.599    2.046      0.010    0.017   10374.0    2495.0

         r_hat
```

```
sigma    1.0
a[0]     1.0
a[1]     1.0
a[2]     1.0
a[3]     1.0
…       …
b[25]    1.0
b[26]    1.0
b[27]    1.0
b[28]    1.0
b[29]    1.0

[61 rows x 9 columns]
```


sigma
a
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
b
0
1
2
3
4
5
6
7
8
Chain
Rank (all chains)



Trace Plots: Show a 'hairy caterpillar' appearance (the distributions of different chains also overlap), indicating that the chains are mixing well and exploring the posterior space effectively.

Rank Plots: Uniformly distributed ranks across chains, suggest that the chains are targeting the same posterior distribution and that there are no major convergence issues.

R-hat: The R-hat values are close to 1.0 for all parameters, suggesting that within-chain and between-chain variances are similar, which is an indicator of convergence.

Effective Sample Size (ESS): The ESS values are high for all parameters (both bulk and tail ESS), indicating that we have a sufficient number of independent samples to estimate the posterior distribution reliably.

Based on these diagnostics, the sampler appears to be working correctly. (Matibe, 2023) –> from my last assignment.

[ ]:
```
# View the posterior predictive - mean-field
ppc_mf
```

[ ]:
```
Inference data with groups:
        > posterior_predictive
        > observed_data
        > constant_data
```

[ ]:
```
# View the posterior predictive - full-rank
ppc_fr
```

[ ]:
```
Inference data with groups:
        > posterior_predictive
        > observed_data
        > constant_data
```

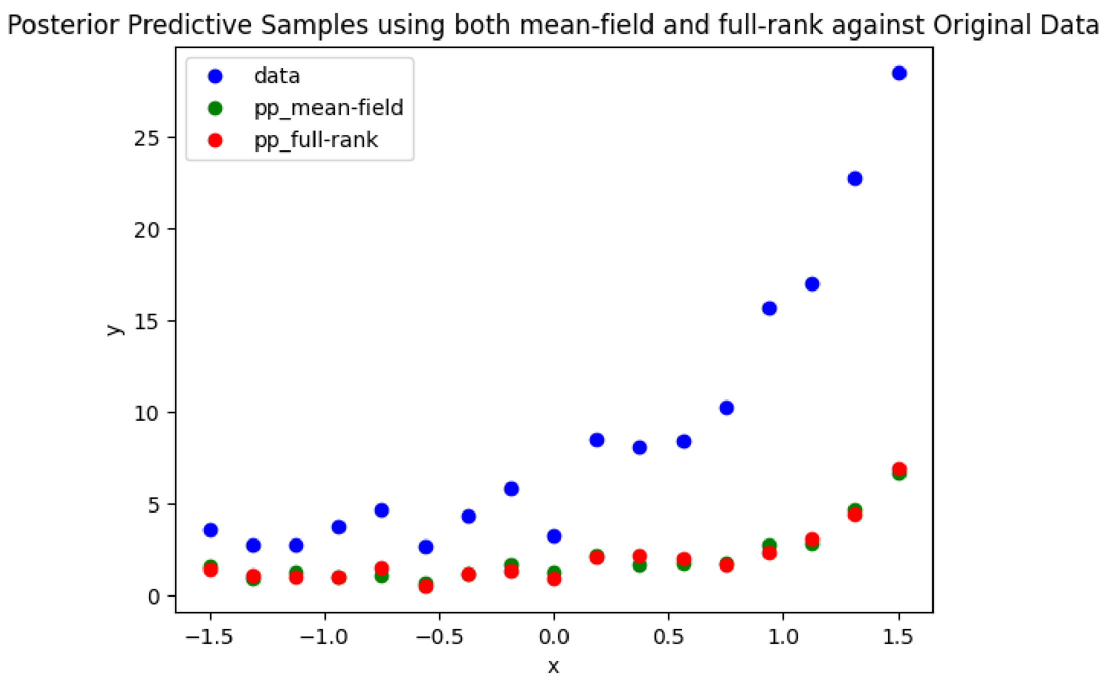
[ ]:
```
# posterior predictive mean - mean-field
ppc_mf_mean = ppc_mf.posterior_predictive['y'].mean(axis=0).mean(axis=0)
len(ppc_mf_mean) # Sanity check
```

[ ]:
```
17
```

[ ]:
```
# posterior predictive mean - full-rank
ppc_fr_mean = ppc_fr.posterior_predictive['y'].mean(axis=0).mean(axis=0)
len(ppc_fr_mean) # Sanity check
```

[ ]:
```
17
```

[ ]:
```
# Plot the original data and posterior predictives
plt.figure()
plt.plot(data_x, data_y, 'o', label='data', color = 'blue')
plt.plot(data_x, ppc_mf_mean, 'o', label='pp_mean-field', color = 'green')
plt.plot(data_x, ppc_fr_mean, 'o', label='pp_full-rank', color = 'red')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Posterior Predictive Samples using both mean-field and full-rank␣
 ↪against Original Data')
plt.legend()
plt.show()
```



From the plot above, we can see that using both mean-field and full-rank lead to somewhat good approximations and goot data fit using the DFT basis functions. To get better approximations, we can increase the number of functions to 40 and use full-rank because this leads to the best model evidence although this causes issues with the sampler. This is done below.

```
[ ]: # best n value
n_best = 40 #chosen n value for best model evidence
dft_model_best = my_dft_model(n_best)

with dft_model_best:
    pm.fit(100000, method='fullrank_advi', progressbar=False)

# Sample from the posterior
    trace_best = pm.sample(1000, tune=2000, target_accept=0.99)

# Posterior predictive
with dft_model_best:
    ppc_best = pm.sample_posterior_predictive(trace_best)
```

```
Finished [100%]: Average Loss = 69.402
Auto-assigning NUTS sampler…
Initializing NUTS using jitter+adapt_diag…
Multiprocess sampling (4 chains in 4 jobs)
NUTS: [sigma, a, b]
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
```

```
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
WARNING (pytensor.tensor.blas): Using NumPy C-API based implementation for BLAS
functions.
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
Sampling 4 chains for 2_000 tune and 1_000 draw iterations (8_000 + 4_000 draws
total) took 579 seconds.
The rhat statistic is larger than 1.01 for some parameters. This indicates
problems during sampling. See https://arxiv.org/abs/1903.08008 for details
The effective sample size per chain is smaller than 100 for some parameters.  A
higher number is needed for reliable rhat and ess computation. See
https://arxiv.org/abs/1903.08008 for details
There were 2 divergences after tuning. Increase `target_accept` or
reparameterize.
Sampling: [y]
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

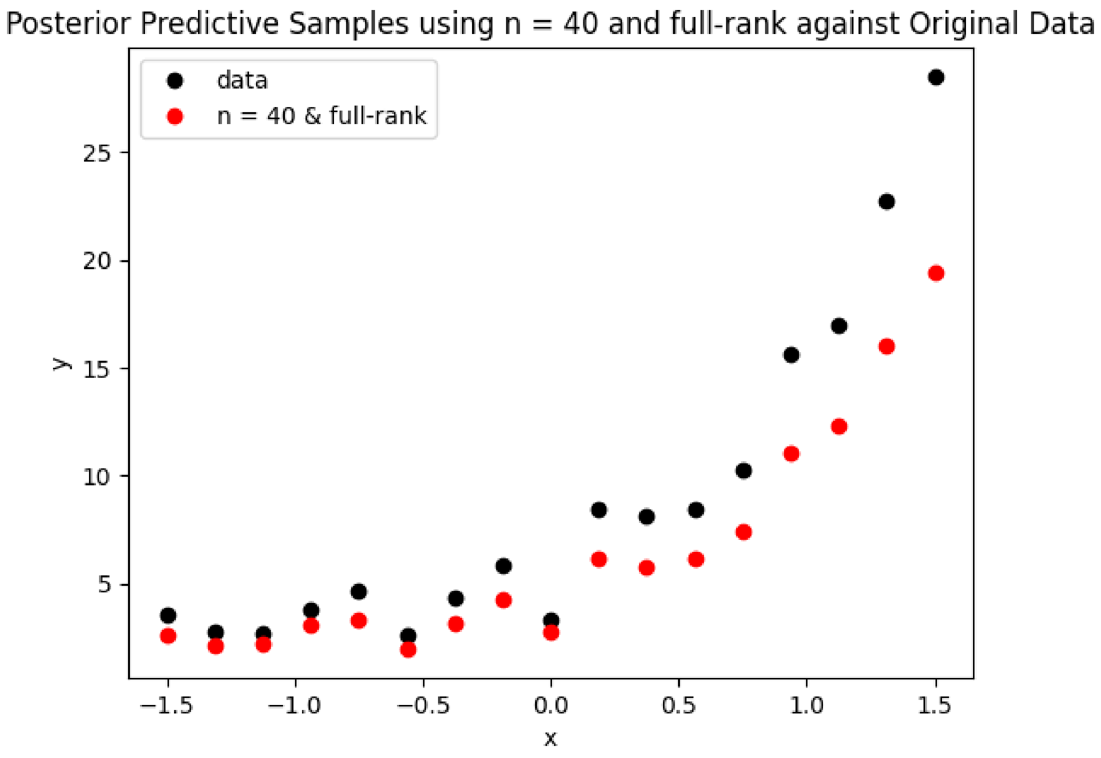
[ ]:
```
# View the posterior predictive - n = 40
ppc_best
```

[ ]:
```
Inference data with groups:
        > posterior_predictive
        > observed_data
        > constant_data
```

[ ]:
```
# posterior predictive mean - n = 40
ppc_best_mean = ppc_best.posterior_predictive['y'].mean(axis=0).mean(axis=0)
len(ppc_best_mean) # Sanity check
```

[ ]:
```
17
```

[ ]:
```
# Plot the original data and posterior predictives
plt.figure()
plt.plot(data_x, data_y, 'o', label='data', color = 'black')
plt.plot(data_x, ppc_best_mean, 'o', label='n = 40 & full-rank', color = 'red')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Posterior Predictive Samples using n = 40 and full-rank against␣
 ↪Original Data')
plt.legend()
plt.show()
```



Predicting y-values for x-values outside data range Using posterior samples to predict y-values for x-values outside the data range, is known as extrapolation, and can lead to unpredictable behavior when DFT basis functions are used. This is because sinusoidal functions are inherently periodic, and extending them beyond the range of the data will result in repeated patterns that may not correspond to the true underlying process that is producing the data since we are only function fitting to the data that we currently have. The predictions will reflect the periodic nature of the sine and cosine functions, potentially leading to significant deviations from any non-periodic trend that might be present in the actual data.

AI Statement: Grammarly for spelling check. I used ChatGPT and Bard to understand and research on DFT but did the work and write-up myself.