bioqtm385-inclass-05-model-selection-2024

October 28, 2024

##BIO/QTM 385: In class exercise for Wednesday, October 16th & Monday, October 21st

(answers will be the part of Assignment #3, Due 10/28)

Carol Zhou. Collaborated with Aanya Vusirikala and Sweta Balaji. Generative AI was used for question 3 and 13 Question 3 AI prompt: I copied and pasted the quesiton into AI for coding the training and test sets. Question 13 AI prompt: I asked it to plot the mean and mean squared error plot at the end.

Aspects of this notebook have been adapted from the Neuromatch course materials here.

As always, all questions to be answered will be in blue and places to write your answers will be in green.

[1]:
```
#import useful libraries
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as random
from scipy.optimize import minimize
from scipy.optimize import fsolve
import pandas as pd

#importing dimensionality reduction packages
from sklearn.decomposition import PCA
from sklearn.decomposition import NMF
from sklearn.manifold import MDS
from sklearn.manifold import Isomap
from sklearn.manifold import LocallyLinearEmbedding
from sklearn.manifold import TSNE
!pip install -q umap-learn[plot]
import umap

#importing clustering packages
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
```

```
                        56.9/56.9 kB
733.6 kB/s eta 0:00:00
                        18.3/18.3 MB
```

```
29.3 MB/s eta 0:00:00
                                    88.8/88.8 kB
4.5 MB/s eta 0:00:00
```

[2]:
```
#@title Figure Settings
import ipywidgets as widgets       # interactive display
%config InlineBackend.figure_format = 'retina'
plt.style.use("https://raw.githubusercontent.com/NeuromatchAcademy/
  ↪course-content/NMA2020/nma.mplstyle")
```

0.1 Model selection

Today, we will be looking at model selection – how to select an individual model out of a set of many potential options. The goal is to select a model that not only describes the data that we have, but also the the data that we would expect to get if we were to either collect more or to repeat the experiment. This process forces us to confront many of the key trade-offs that we have seen so far in class. These include representation vs. fidelity, incorporating new information vs. trusting prior knowledge, and a new trade-off: bias vs. variance. We will start here.

##Bias-variance tradeoff

Training Set is the portion of a data set that you use to fit a given model. These data are used to find best-fit parameters, θ, or, more generally, the posterior distribution $p(\theta|\vec{x})$.

Test Set is the portion of a data that is held-aside during the fitting procedure and is used to assess how the model generalizes to new data.

Bias is the difference between the prediction of the model and the "true" output variables that you are trying to predict. Models with high bias will not fit the training set data well, since the predictions are quite different from the true process generating the data. These high bias models are overly simplified - they do not have enough parameters and complexity to accurately capture the patterns in the data (*underfitting*).

Variance refers to the variability of model predictions across different data samples. Essentially, do the model predictions change a lot with changes in the exact training data used? Models with high variance are highly dependent on the exact training data used - they will not generalize well to test data (*overfitting*).

- High bias, low variance models have high train and test error.
- Low bias, high variance models have low train error, high test error
- Low bias, low variance models have low train and test error
- High bias, high variance models should just be avoided

Having a low bias and a high variance are typically in conflict though - models with enough complexity to have low bias also tend to overfit and depend on the training data more. We need to decide on the correct tradeoff.

In this section, we will see the bias-variance tradeoff in action with polynomial regression models of different orders.

Graphical illustration of bias and variance. (Source: http://scott.fortmann-roe.com/docs/BiasVariance.html)

##An Example: Fitting Polynomials

A typical exersize in data analysis is to fit a polynomial to a data set. Computing this fit is typically straight-forward (in python, you can use `np.polyfit()`), regardless of the order of the polynomial. However, selecting how many orders is the "best" is often rather challenging without using a formal methodology like Bayesian model selection.

We will start with a case where we know the answer already, data that is generated from the function $y = f(x) = 0.1x^2 + 0.5x + 2 + N(0, \sigma^2)$, where $N(0, \sigma^2)$ is a Gaussian random variable with mean of 0 and standard deviation σ.

Question #1: Write a function, `output_quadratic(x,sigma)`, where `x` is a numpy array of input values, `sigma` is the noise standard deviation, and the function returns `y`, the output of applying the equation above to `x` (make sure that the noise is independent for each value of `x`).

[ ]:
```
def output_quadratic(x,sigma):
  y = 0.1*x**2 + 0.5*x + 2 + np.random.normal(0,sigma**2,len(x))
  return y
```

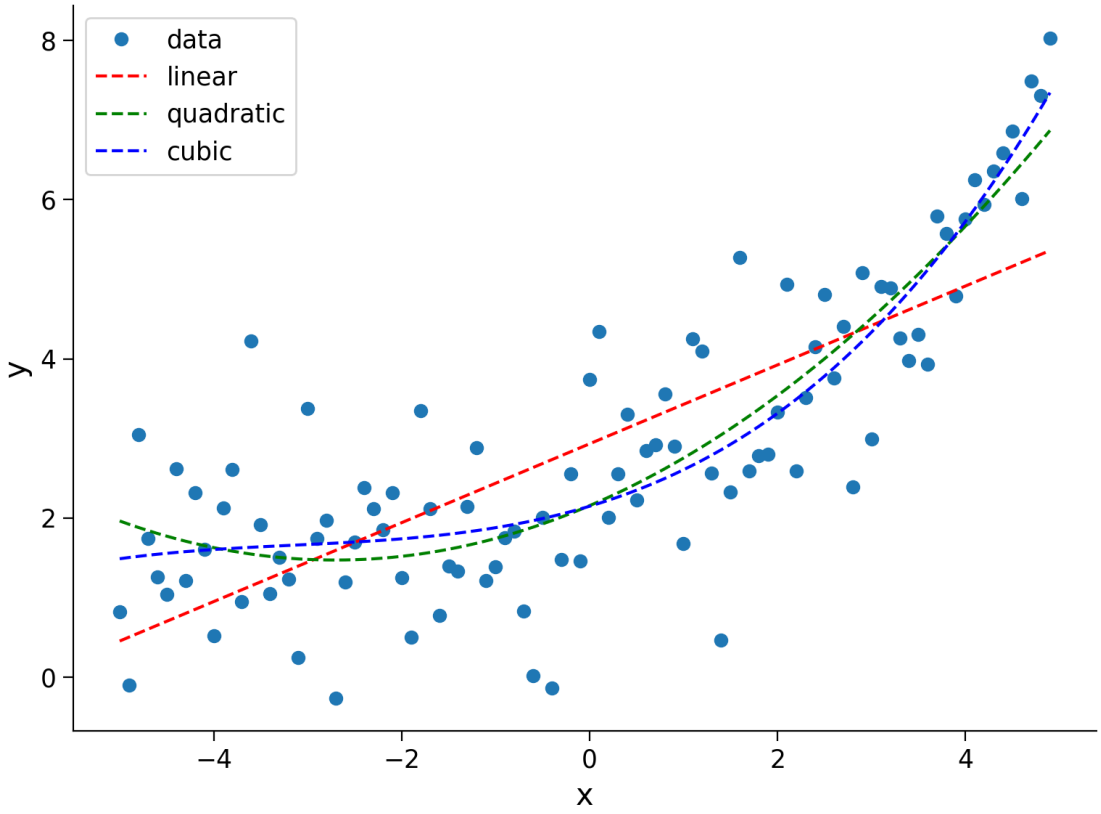
Question #2: For $\sigma = 1$, fit outputs of your function above to a linear, a quadratic, and a cubic polynomial using `p = np.polyfit(x,y,order)`, where `x=np.arange(-5,5,.1)` and plot the outputs. Compare the mean squared errors for all three fits (you can use `np.polyval()`). Which fit has the lowest mean squared error? Does this answer surprise you? Why or why not?

[ ]:
```
x = np.arange(-5,5,.1)
y = output_quadratic(x,1)

# fit
p_linear = np.polyfit(x,y,1)
p_quadratic = np.polyfit(x,y,2)
p_cubic = np.polyfit(x,y,3)

# evaluate the fit
y_linear = np.polyval(p_linear, x)
y_quadratic = np.polyval(p_quadratic, x)
y_cubic = np.polyval(p_cubic, x)

# calculate the mean squared errors
print(np.mean((y - y_linear) ** 2))
print(np.mean((y - y_quadratic) ** 2))
print(np.mean((y - y_cubic) ** 2))

plt.plot(x, y, 'o')
plt.plot(x, y_linear, '--',c='red')
plt.plot(x, y_quadratic, '--',c='green')
plt.plot(x, y_cubic, '--',c='blue')
plt.legend(['data','linear','quadratic','cubic'])
```

```
plt.xlabel('x')
plt.ylabel('y')
```

```
1.4090316171599542
0.9261897411729092
0.8899594295163766
```

[ ]: `Text(0, 0.5, 'y')`

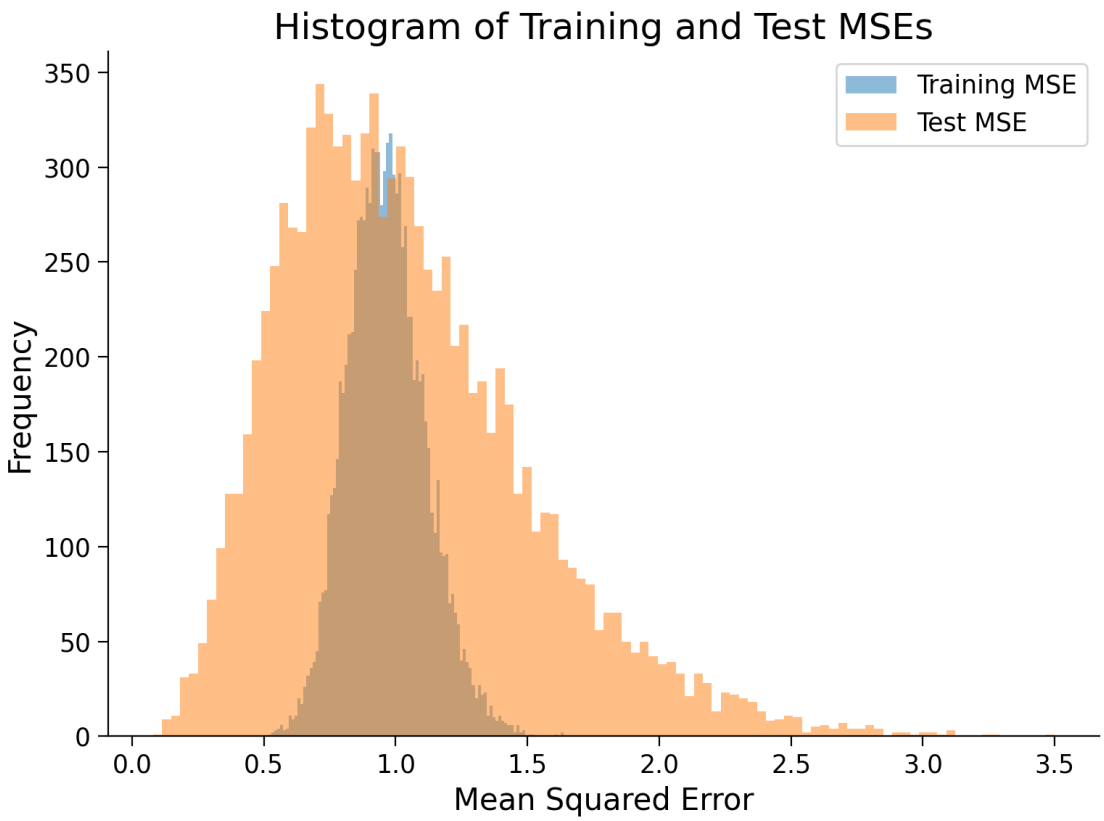


Which fit has the lowest mean squared error? Does this answer surprise you? Why or why not? The cubic fit has the lowest mean squared error which surprised me because we know the data is generated from a quadratic function and the quadratic fit should has the lowest mean squared error. This could due to overfitting.

Question #3: Now, regenerate y 10,000 times using your function from Question #1 (still, $\sigma = 1$), and fit a quadratic polynomial to a training set that consists of a random 90% of the values (90 points out of 100). Save the fitted parameter values, the training set mean-squared-error (MSE), and the test set MSE each time. Plot histograms for the two MSEs. Is your training or test MSE higher? Why is this? Do the parameter estimates appear biased to you? (Remember: the model generating the data is $y = f(x) = 0.1x^2 + 0.5x + 2 + N(0, \sigma^2)$)

[ ]:
```
#Hint: you can use:
#   idx = np.arange(np.shape(x)[0])
#   random.shuffle(idx);
#   x_train = x[idx[:L]];y_train = y[idx[:L]];x_test = x[idx[L:]];_test =␣
 ↪y[idx[L:]]
#to generate a random test and training set (where L = 90, here)

train_mse_list = []
test_mse_list = []
params_list = []

for i in range(10000):
  y = output_quadratic(x, 1)

  idx = np.arange(np.shape(x)[0])
  random.shuffle(idx)

  x_train = x[idx[:90]]
  y_train = y[idx[:90]]
  x_test = x[idx[90:]]
  y_test = y[idx[90:]]

  p_quadratic_train = np.polyfit(x_train,y_train, 2)
  p_quadratic_test = np.polyfit(x_train,y_train, 2)
  params_list.append(p_quadratic_train)

  y_train_predict = np.polyval(p_quadratic_train, x_train)
  y_test_predict = np.polyval(p_quadratic_test, x_test)

  train_mse = np.mean((y_train - y_train_predict) ** 2)
  test_mse = np.mean((y_test - y_test_predict) ** 2)

  train_mse_list.append(train_mse)
  test_mse_list.append(test_mse)

train_mse_list = np.array(train_mse_list)
test_mse_list = np.array(test_mse_list)

plt.hist(train_mse_list, bins=100, alpha=0.5, label='Training MSE')
plt.hist(test_mse_list, bins=100, alpha=0.5, label='Test MSE')
plt.title('Histogram of Training and Test MSEs')
plt.xlabel('Mean Squared Error')
plt.ylabel('Frequency')
plt.legend()
plt.show()

print(f'Mean Training MSE: {np.mean(train_mse_list)}')
```

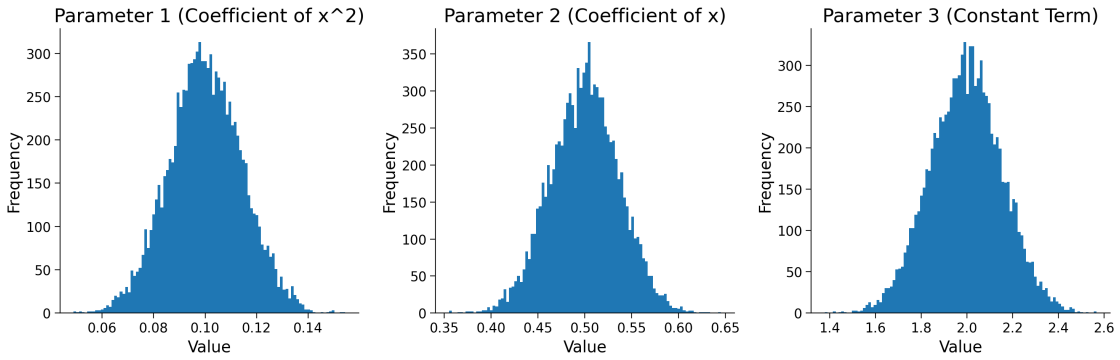
```
print(f'Mean Test MSE: {np.mean(test_mse_list)}')
```




```
Mean Training MSE: 0.9662308982497918
Mean Test MSE: 1.0332707938697774
```

[ ]:
```
params_array = np.array(params_list)

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.hist(params_array[:, 0], bins=100)
plt.title("Parameter 1 (Coefficient of x^2)")
plt.xlabel('Value')
plt.ylabel('Frequency')

plt.subplot(1, 3, 2)
plt.hist(params_array[:, 1], bins=100)
plt.title("Parameter 2 (Coefficient of x)")
plt.xlabel('Value')
plt.ylabel('Frequency')
```

```
plt.subplot(1, 3, 3)
plt.hist(params_array[:, 2], bins=100)
plt.title("Parameter 3 (Constant Term)")
plt.xlabel('Value')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```



Is your training or test MSE higher? Why is this? Do the parameter estimates appear biased to you? My test MSE is higher. This is expected because the model fits the training data closely. But the test data is not used for fitting and will likely show more noise and variance as the model is not generalized to fit it perfectly. The parameter estimates do not appear biased to me because they all follow a normal distribution and the mean of each distribution is near the true parameters of 0.1, 0.5, and 2 correspondingly.

Question #4: Repeat Question #3, but with cubic fits instead of quadratic fits. Compare the means and variances of the training and test set MSEs with those that you got from the quadratic fits. Based on this information, which fit would you pick (pretend that you don't know the answer)? Justify your answer.

[ ]:
```
#Type your code for Question #4 here
train_mse_list_cubic = []
test_mse_list_cubic = []

for i in range(10000):
  y = output_quadratic(x, 1)

  idx = np.arange(np.shape(x)[0])
  random.shuffle(idx)

  x_train = x[idx[:90]]
  y_train = y[idx[:90]]
  x_test = x[idx[90:]]
  y_test = y[idx[90:]]
```

```
  p_cubic_train = np.polyfit(x_train,y_train, 3)
  p_cubic_test = np.polyfit(x_train,y_train, 3)

  y_train_predict = np.polyval(p_cubic_train, x_train)
  y_test_predict = np.polyval(p_cubic_train, x_test)

  train_mse = np.mean((y_train - y_train_predict) ** 2)
  test_mse = np.mean((y_test - y_test_predict) ** 2)

  train_mse_list_cubic.append(train_mse)
  test_mse_list_cubic.append(test_mse)

train_mse_list_cubic = np.array(train_mse_list_cubic)
test_mse_list_cubic = np.array(test_mse_list_cubic)

print(f'Mean Training MSE (Quadratic): {np.mean(train_mse_list)}')
print(f'Variance of Training MSE (Quadratic): {np.var(train_mse_list)}')
print(f'Mean Test MSE (Quadratic): {np.mean(test_mse_list)}')
print(f'Variance of Test MSE (Quadratic): {np.var(test_mse_list)}')
print(f'Mean Training MSE (Cubic): {np.mean(train_mse_list_cubic)}')
print(f'Variance of Training MSE (Cubic): {np.var(train_mse_list_cubic)}')
print(f'Mean Test MSE (Cubic): {np.mean(test_mse_list_cubic)}')
print(f'Variance of Test MSE (Cubic): {np.var(test_mse_list_cubic)}')
```

```
Mean Training MSE (Quadratic): 0.9662308982497918
Variance of Training MSE (Quadratic): 0.021805165265324358
Mean Test MSE (Quadratic): 1.0332707938697774
Variance of Test MSE (Quadratic): 0.21722508016899864
Mean Training MSE (Cubic): 0.9562997914415511
Variance of Training MSE (Cubic): 0.021163454015228303
Mean Test MSE (Cubic): 1.0462719696016947
Variance of Test MSE (Cubic): 0.2165747944273804
```

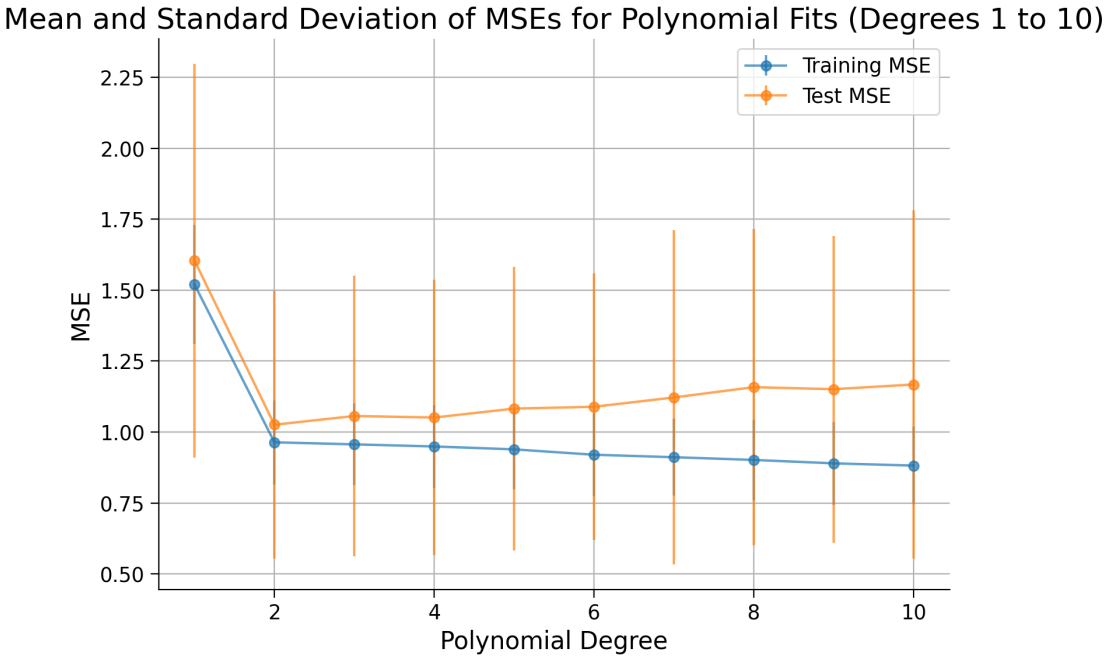
Compare the means and variances of the training and test set MSEs with those that you got from the quadratic fits. Based on this information, which fit would you pick (pretend that you don't know the answer)? Justify your answer. I would pick quadratic fit because its mean test MSE (1.03) is lower than that of cubic (1.04) with similar variance of test MSE. This suggests that the quadratic fit is more likely to generalize better to new data.

Question #5: Plot the mean and standard deviation for the test and training MSEs for polynomial fits from order 1 to 10 ($\sigma = 1$, still). For the sake of time, you can reduce the number of instantiations per polynomial to 1,000 instead of 10,000. Based on these results, which order would you pick? Why? Does this agree with the correct result? Discuss your results in light of the bias-variance trade-off. (You can use `plt.errorbar()` to make your plots).

[ ]:
```
train_mse_mean = []
train_mse_std = []
```

```
test_mse_mean = []
test_mse_std = []

for i in range(1,11):
  train_mse_list = []
  test_mse_list = []

  for j in range(1000):
    y = output_quadratic(x, 1)

    idx = np.arange(np.shape(x)[0])
    random.shuffle(idx)

    x_train = x[idx[:90]]
    y_train = y[idx[:90]]
    x_test = x[idx[90:]]
    y_test = y[idx[90:]]

    p_train = np.polyfit(x_train, y_train, i)
    p_test = np.polyfit(x_train, y_train, i)

    y_train_predict = np.polyval(p_train, x_train)
    y_test_predict = np.polyval(p_test, x_test)

    train_mse = np.mean((y_train - y_train_predict) ** 2)
    test_mse = np.mean((y_test - y_test_predict) ** 2)

    train_mse_list.append(train_mse)
    test_mse_list.append(test_mse)

  train_mse_mean.append(np.mean(train_mse_list))
  train_mse_std.append(np.std(train_mse_list))
  test_mse_mean.append(np.mean(test_mse_list))
  test_mse_std.append(np.std(test_mse_list))

train_mse_mean = np.array(train_mse_mean)
train_mse_std = np.array(train_mse_std)
test_mse_mean = np.array(test_mse_mean)
test_mse_std = np.array(test_mse_std)

plt.errorbar(range(1,11), train_mse_mean, yerr=train_mse_std, alpha=0.7,␣
 ↪fmt='-o', label='Training MSE')
plt.errorbar(range(1,11), test_mse_mean, yerr=test_mse_std, alpha=0.7,␣
 ↪fmt='-o', label='Test MSE')
plt.title("Mean and Standard Deviation of MSEs for Polynomial Fits (Degrees 1␣
 ↪to 10)")
plt.xlabel("Polynomial Degree")
```

```
plt.ylabel("MSE")
plt.legend()
plt.grid(True)
plt.show()
```



[ ]:
```
print("Training MSE Means:", train_mse_mean)
print("Training MSE Variances:", train_mse_std)
print("Test MSE Means:", test_mse_mean)
print("Test MSE Variances:", test_mse_std)
```

```
Training MSE Means: [1.5198343  0.96362712 0.95637846 0.94893228 0.93878003
0.91977019
 0.91109152 0.90148849 0.88948856 0.8815458 ]
Training MSE Variances: [0.20933972 0.14799901 0.1444833  0.14595276 0.13953595
0.14539549
 0.13555231 0.14189347 0.14613673 0.13735639]
Test MSE Means: [1.6039013  1.0256038  1.05636911 1.05096153 1.08241015
1.08887699
 1.12165888 1.15798429 1.15079993 1.16742886]
Test MSE Variances: [0.69426694 0.4727873  0.49525349 0.48509988 0.49953038
0.46966713
 0.58928727 0.55778054 0.54087095 0.61282749]
```

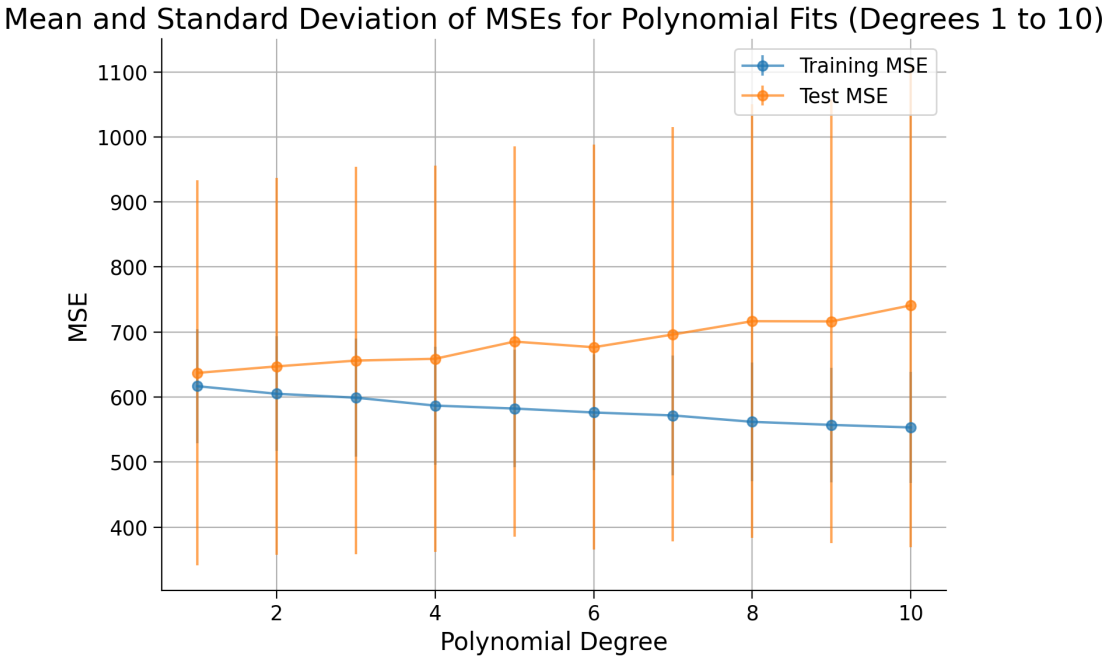
Based on these results, which order would you pick? Why? Does this agree with the correct result? Discuss your results in light of the bias-variance trade-off. I would pick order 2 because it has the lowest test MSE means and relatively low test MSE varainces. This means the model generalize

well to new data. Though low-order polynomials like quadratic and cubic might not capture the underlying complexity of data as well as high-order polynomial (high bias), they generalize well to test data (low test MSE so low variance). While higher polynomial order might have lower training MSE (low bias), their test MSE are high and they don't fit the test data well (high varaince). But polynomial order of 2 tries to keep bias relatively low while still fitting the data consistently well between different runs.

Question #6: Repeat the previous question but for $\sigma = 5$. How do you results change? Why do you think these effects are occurring?

[ ]:
```
train_mse_mean = []
train_mse_std = []
test_mse_mean = []
test_mse_std = []

for i in range(1,11):
  train_mse_list = []
  test_mse_list = []

  for j in range(1000):
    y = output_quadratic(x, 5)

    idx = np.arange(np.shape(x)[0])
    random.shuffle(idx)

    x_train = x[idx[:90]]
    y_train = y[idx[:90]]
    x_test = x[idx[90:]]
    y_test = y[idx[90:]]

    p_train = np.polyfit(x_train, y_train, i)
    p_test = np.polyfit(x_train, y_train, i)

    y_train_predict = np.polyval(p_train, x_train)
    y_test_predict = np.polyval(p_test, x_test)

    train_mse = np.mean((y_train - y_train_predict) ** 2)
    test_mse = np.mean((y_test - y_test_predict) ** 2)

    train_mse_list.append(train_mse)
    test_mse_list.append(test_mse)

  train_mse_mean.append(np.mean(train_mse_list))
  train_mse_std.append(np.std(train_mse_list))
  test_mse_mean.append(np.mean(test_mse_list))
  test_mse_std.append(np.std(test_mse_list))

train_mse_mean = np.array(train_mse_mean)
```

```
train_mse_std = np.array(train_mse_std)
test_mse_mean = np.array(test_mse_mean)
test_mse_std = np.array(test_mse_std)

plt.errorbar(range(1,11), train_mse_mean, yerr=train_mse_std, alpha=0.7,␣
 ↪fmt='-o', label='Training MSE')
plt.errorbar(range(1,11), test_mse_mean, yerr=test_mse_std, alpha=0.7,␣
 ↪fmt='-o', label='Test MSE')
plt.title("Mean and Standard Deviation of MSEs for Polynomial Fits (Degrees 1␣
 ↪to 10)")
plt.xlabel("Polynomial Degree")
plt.ylabel("MSE")
plt.legend()
plt.grid(True)
plt.show()
```



[ ]:
```
print("Training MSE Means:", train_mse_mean)
print("Training MSE Variances:", train_mse_std)
print("Test MSE Means:", test_mse_mean)
print("Test MSE Variances:", test_mse_std)
```

```
Training MSE Means: [616.70296666 605.04883788 599.03990986 586.86810324
582.35273478
 576.27430583 571.71024826 561.91158369 557.18427394 553.26554714]
Training MSE Variances: [87.67011923 88.12274481 90.9631341  90.82860527
```

```
90.37352646 88.93663256
 92.21678834 91.23643996 88.16797393 85.42687939]
Test MSE Means: [637.13040805 647.12189313 656.0924908  658.85543875
685.31051569
 676.62508616 696.22481669 716.68600564 716.36290172 741.02010007]
Test MSE Variances: [296.20596973 290.1308003  297.93530839 297.17979629
300.39244182
 311.40817177 318.56170446 333.62542627 341.24783343 371.60389191]
```

How do your results change? Why do you think these effects are occurring? The training and test MSE are larger. This is because sigma = 1 is changed to sigma = 5, which changes sigma**2 from 1 to 25. This increased noise thus results in higher training and test MSEs. Moreover, it seems like the linear fit generalize the test data the best. And this could because of its simplicity, generalizability, and tendency of not overfitting noise in the data. The test MSEs also increase more rapidly with higher-order polynomials probably due to overfitting, especially with significant noise.

##Estimating errors and maximum likelihood

Mean-squared-error, while a useful measure, doesn't fit directly into a Bayesian framework, which is based purely on probabilities. For this, we need to translate MSE into a *likelihood* - p(Data|Model), which we will call $p(X|M,\theta)$. This is done by assuming that the data we see is generated from a true underlying model and is then corrupted by a noise distribution that we know. We can also make our lives easier by assuming that each data point is arrived at independently. Thus, our likelihood is aproximated by:

$$p(X|M,\theta) \approx p(x_1|M,\theta)p(x_2|M,\theta)\,...\,p(x_N|M,\theta) = \prod_{i=1}^{N} p(x_i|M,\theta), \tag{1}$$

where each $x_i \in X$ is a seperate data point. Or, if we write this in terms of the log-likelihood, $L(X|M,\theta)$,

$$L(X|M,\theta) \approx \log p(x_1|M,\theta) + \log p(x_2|M,\theta) + ... + \log p(x_N|M,\theta) = \sum_{i=1}^{N} \log p(x_i|M,\theta), \tag{2}$$

Although the distribution $p(x_i|M,\theta)$ could have many potential forms, due ot the central limit theorem, the most common form for this is a gaussian distribution centered around our model's estimate, $\hat{f}(x_i|M,\theta)$:

$$p(x_i|M,\theta) = \frac{1}{\sqrt{2\pi\hat{\sigma^2}}} \exp\left[\frac{-(x_i - \hat{f}(x_i|M,\theta)^2}{2\hat{\sigma^2}}\right], \tag{3}$$

where $\hat{\sigma^2}$ is the measured mean-squared error of the fit.

It should be noted that the mean-squared error often isn't a good assumption in the case of high-bias/low-variance models that underfit the data (e.g., the linear fit back in Question #1), but it is a good assumption when comparing models that don't obviously under-fit the data.

Nonetheless, if we plug this definition into the previous equation (and play around with logrithms and exponentials), we find that:

$$L(X|M,\theta) = \sum_{i=1}^{N} \log p(x_i|M,\theta) = -\frac{N}{2}\log(2\pi\hat{\sigma}^2) - \frac{N}{2}. \tag{4}$$

Thus, as our MSE $\hat{\sigma^2}$ decreases, $L(X|M,\theta)$ increases, as the likelihood becomes less negative. Thus, minimizing MSE and maximizing likelihood are equivalent!

Question #7: Write a function, `calculateLikelihood(x,y,order)`, that fits a polynomial of order `order` to numpy arrays `x` and `y` and returns the likelihood of generating the observed data given the fitted model (using the equation above).

[ ]:
```
def calculateLikelihood(x, y, order):
    hat_sigma = np.mean((y - np.polyval(np.polyfit(x, y, order), x))**2)
    N = len(x)
    likelihood = - (N / 2) * np.log(2 * np.pi * hat_sigma**2) - (N / 2)
    return likelihood
```

Question #8: Use your functions from Questions #1 and #7 to calculate the likelihoods for polynomial fits from order 1 to 10 for a single instantiation of the data. Use `x=np.arange(-5,5,.1)` and $\sigma = 1$. Based soley on the likelihood, which model would you pick? Why?

[ ]:
```
x = np.arange(-5,5,.1)
y = output_quadratic(x,1)

likelihood_list = []
for i in range(1,11):
  likelihood_list.append(calculateLikelihood(x,y,i))

for order, likelihood in zip(range(1, 11), likelihood_list):
    print(f'Order {order}: Likelihood = {likelihood:.4f}')
```

```
Order 1: Likelihood = -187.7734
Order 2: Likelihood = -127.3245
Order 3: Likelihood = -127.3207
Order 4: Likelihood = -124.5193
Order 5: Likelihood = -124.2315
Order 6: Likelihood = -122.7521
Order 7: Likelihood = -122.5159
Order 8: Likelihood = -122.4768
Order 9: Likelihood = -121.1647
Order 10: Likelihood = -120.9537
```

Based soley on the likelihood, which model would you pick? Why? I would pick order 10 because it has the maximum likelihood. Higher likelihood values indicate that the model is more consistent with the observed data given the assumptions of normality in the residuals. But this doesn't make sense because we know that the 10th order would overfit the data.

##The Akaike Information Criteron (AIC)

As discussed in class and in the previous sections of this notebook, however, just minimizing the likelihood often leads to low-bias/high-variance models (e.g., overfitting). Really, we want to minimize the error on (or, more generally, maximize the likelihood of generating) new data that the model has not seen during fitting. The Akaike Information Critereon (AIC) is a way of estimating this likehood in the case of $N \to \infty$. Specifically, if the model is trained on data X and then sees new data $X^{(0)}$ generated by the same process as X,

$$L(X^{(0)}|M,\theta) \approx \frac{1}{N} L(X|M,\hat{\theta}) - \frac{p}{N}, \tag{5}$$

where M is the model in question, $\hat{\theta}$ is the parameter set that optimizes the likelihood of X, N is the number of data points, and p is the number of parameters in the model.

Given this observation, then we usually define the AIC to be -2N times this quantity:

$$\text{AIC} = -2L(X|M,\hat{\theta}) + 2p. \tag{6}$$

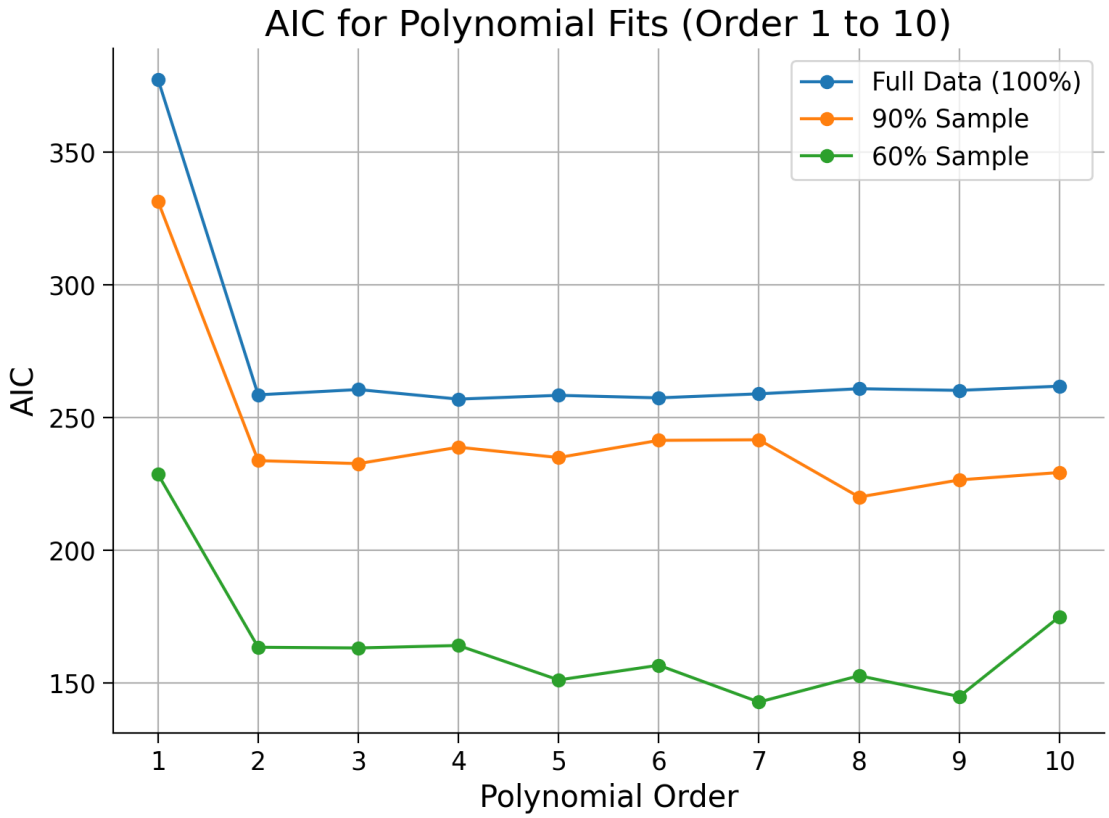
Thus, a smaller value of the AIC implies a larger likelihood on the new data, and, thus, a model that generalizes better. It should be noted, however, that corrections need to be made in cases where N is not effectively infinite, and there are many methods that have been developed to adjust for these effects in different ways.

Question #9: Use your functions from Questions #1 and #7 to calculate and plot the AIC for polynomial fits of order 1 through 10 for a single instantiation of the data (same σ and `x` as the previous question). Repeat the analysis for random subsamples of 60% and 90% the data. How do your results change? What might be causing this?

[ ]:
```
AIC = []
for i in range(1,11):
  AIC.append(-2*calculateLikelihood(x,y,i) + 2*i)
print(AIC)
print(f'Lowest AIC: Order {np.argmin(AIC)+1} with AIC = {AIC[np.argmin(AIC)]}')

AIC_60 = []
AIC_90 = []

for i in range(1,11):
  idx = np.arange(np.shape(x)[0])
  random.shuffle(idx)

  x_60 = x[idx[:int(len(x) * 0.6)]]
  y_60 = y[idx[:int(len(x) * 0.6)]]
  AIC_60.append(-2*calculateLikelihood(x_60,y_60,i) + 2*i)

  x_90 = x[idx[:int(len(x) * 0.9)]]
  y_90 = y[idx[:int(len(x) * 0.9)]]
  AIC_90.append(-2*calculateLikelihood(x_90,y_90,i) + 2*i)

print(AIC_60)
print(AIC_90)
```

```
plt.plot(range(1,11), AIC, marker='o', label='Full Data (100%)')
plt.plot(range(1,11), AIC_90, marker='o', label='90% Sample')
plt.plot(range(1,11), AIC_60, marker='o', label='60% Sample')
plt.title('AIC for Polynomial Fits (Order 1 to 10)')
plt.xlabel('Polynomial Order')
plt.ylabel('AIC')
plt.xticks(range(1,11))
plt.legend()
plt.grid()
plt.show()
```

```
[377.54671742364354, 258.64897813993036, 260.6413671833434, 257.0385428572837,
258.46295189950274, 257.50424350546325, 259.0317500783508, 260.9536523325299,
260.3294722639205, 261.90732158830787]
Lowest AIC: Order 4 with AIC = 257.0385428572837
[228.73376153519095, 163.54242291528445, 163.26401324203374, 164.2058668023326,
151.21463587432967, 156.7562530688441, 142.88578844414016, 152.81755741365637,
144.9604429467379, 175.04539113735814]
[331.5693750429094, 233.85814863986678, 232.73459893331477, 238.9172545003686,
235.01661605546636, 241.49092575930402, 241.7152944405147, 220.1469342269349,
226.57928341355324, 229.3911901635533]
```



How do your results change? What might be causing this? Since a smaller AIC value indicates a better generalizable model, for the full date, order 4 has the lowest AIC and thus is a better model instead of order 10 from question 8. This could probably because AIC takes the balance between goodness of fit and model complexity into consideration and uses a penalty of 2p for the number of parameters. Order 4 is complex enough to capture the underlying structure of the data without overfitting significantly to noise. However, with 60% and 90% samples, order 8 and 7 have the lowest AIC values. This could be because of the variability in the estimates and the fact that with smaller samples. These two orders just happen to capture complexity without fitting too closely to the noise the best.

##The Bayesian Information Criteron (BIC)

While the AIC provides an improved estimate of a model's generalizability, it remains difficult to translate that estimate into a *posterior distribution* - $p(M|X)$ - of a model given the data. Written more explicitly,

$$p(M|X) \propto p(M)p(X|M) = p(M) \int p(X|M,\theta)p(\theta|M)d\theta. \tag{7}$$

If we assume that $p(X|M,\theta)$ is sufficiently "peaky" near its maximum, $\hat{\theta}$, then we can use Laplace's Method to approximate the integral, so that the log-likehood, $L(X|M)$ is approximated by

$$L(X|M) \approx L(X|M,\hat{\theta}) - \frac{p}{2}\log N, \tag{8}$$

where p, again, is the number of parameters, and N is the number of data points.

Mutiplying by -2, we thus define the Bayesian Information Critereon (BIC) as:

$$\text{BIC} = -2L(X|M,\hat{\theta}) + p\log N. \tag{9}$$

Thus, the smaller the BIC, the higher the posterior distrtibution of the model, and the more likely that the model is the "true" model of the data.

Moreover, because the the BIC is proportional to -2 times the log-posterior, if given a set of models, $M_1, M_2, \ldots, M_k$, then if we assume a uniform prior over models, we can calculate the posterior distribution over our models via:

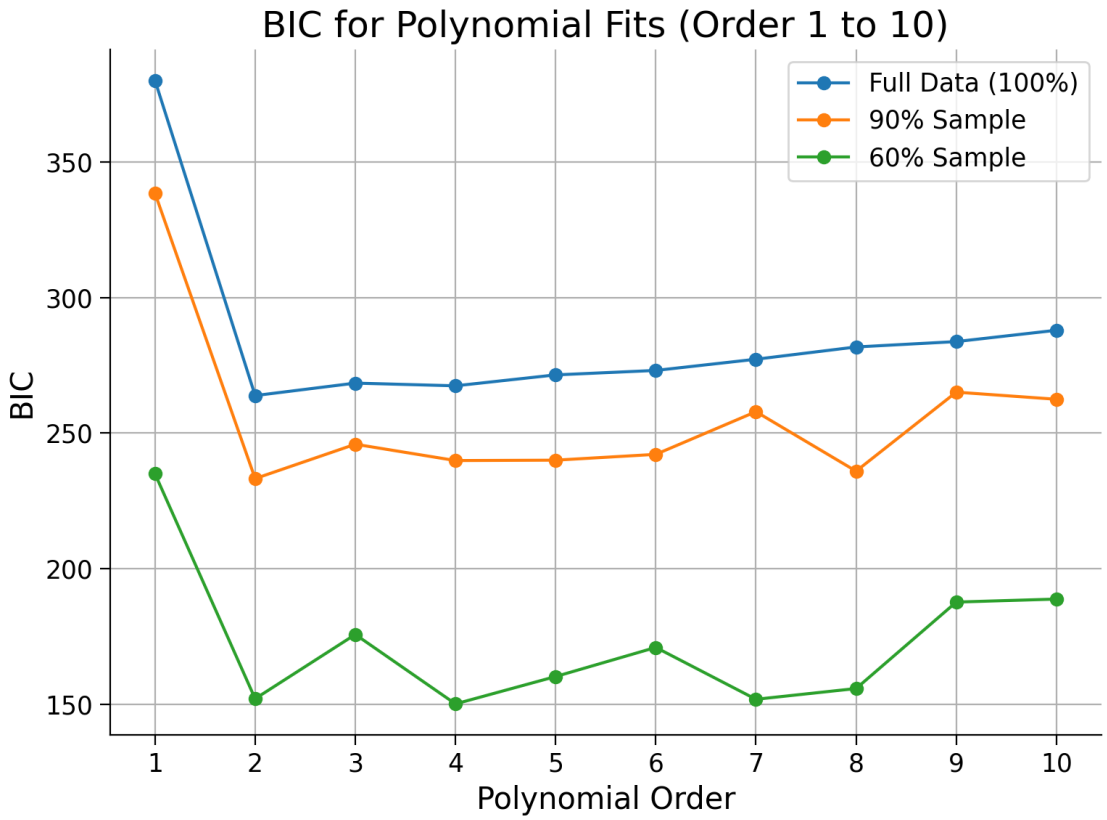
$$p(i|X) = \frac{exp\left[-\frac{1}{2}\text{BIC}(M_i)\right]}{\sum_{\ell=1}^{k} exp\left[-\frac{1}{2}\text{BIC}(M_\ell)\right]}. \tag{10}$$

Note that often we are actually uncertain about the parameters that maximize $p(X|M,\theta)$ (this was the whole point of Bayesian inference!), so in many cases, the assumptions that underlie the BIC do not hold, and we need to estimate the full integral in the first equation of this section using Monte Carlo or other methods.

Question #10: Use your functions from Questions #1 and #7 to calculate and plot the BIC for polynomial fits of order 1 through 10 for a single instantiation of the data (same σ and `x` as the previous question). Repeat the analysis for random subsamples of 60% and 90% the data. How do your results compare to the AIC results? What might be causing these differences?

[ ]:
```
BIC = []
for i in range(1,11):
  BIC.append(-2*calculateLikelihood(x,y,i) + i*np.log(len(x)))
print(BIC)
print(f'Lowest BIC: Order {np.argmin(BIC)+1} with BIC = {BIC[np.argmin(BIC)]}')

BIC_60 = []
BIC_90 = []

for i in range(1,11):
  idx = np.arange(np.shape(x)[0])
  random.shuffle(idx)

  x_60 = x[idx[:int(len(x) * 0.6)]]
  y_60 = y[idx[:int(len(x) * 0.6)]]
  BIC_60.append(-2*calculateLikelihood(x_60,y_60,i) + i*np.log(len(x_60)))

  x_90 = x[idx[:int(len(x) * 0.9)]]
  y_90 = y[idx[:int(len(x) * 0.9)]]
  BIC_90.append(-2*calculateLikelihood(x_90,y_90,i) + i*np.log(len(x_90)))

print(BIC_60)
print(BIC_90)

plt.plot(range(1,11), BIC, marker='o', label='Full Data (100%)')
plt.plot(range(1,11), BIC_90, marker='o', label='90% Sample')
plt.plot(range(1,11), BIC_60, marker='o', label='60% Sample')
plt.title('BIC for Polynomial Fits (Order 1 to 10)')
plt.xlabel('Polynomial Order')
plt.ylabel('BIC')
plt.xticks(range(1,11))
plt.legend()
plt.grid()
plt.show()
```

```
[380.1518876096316, 263.85931851190657, 268.45687774130766, 267.45922360123603,
271.4888028294432, 273.1352646213918, 277.2679413802674, 281.79501382043463,
283.7760039378133, 287.95902344818876]
Lowest BIC: Order 2 with BIC = 263.85931851190657
[234.99931093503852, 151.99774703105578, 175.7114039919618, 150.0907715727634,
160.18856023609942, 170.93219540640513, 151.7990711927487, 155.76224500122655,
187.66429088243652, 188.7722064296217]
[338.52437177483944, 233.22736932823992, 245.9041967926908, 239.88474773651168,
240.02682101812314, 242.1669482857166, 257.9120043265406, 235.96976894077804,
265.11999287411464, 262.49634805138476]
```

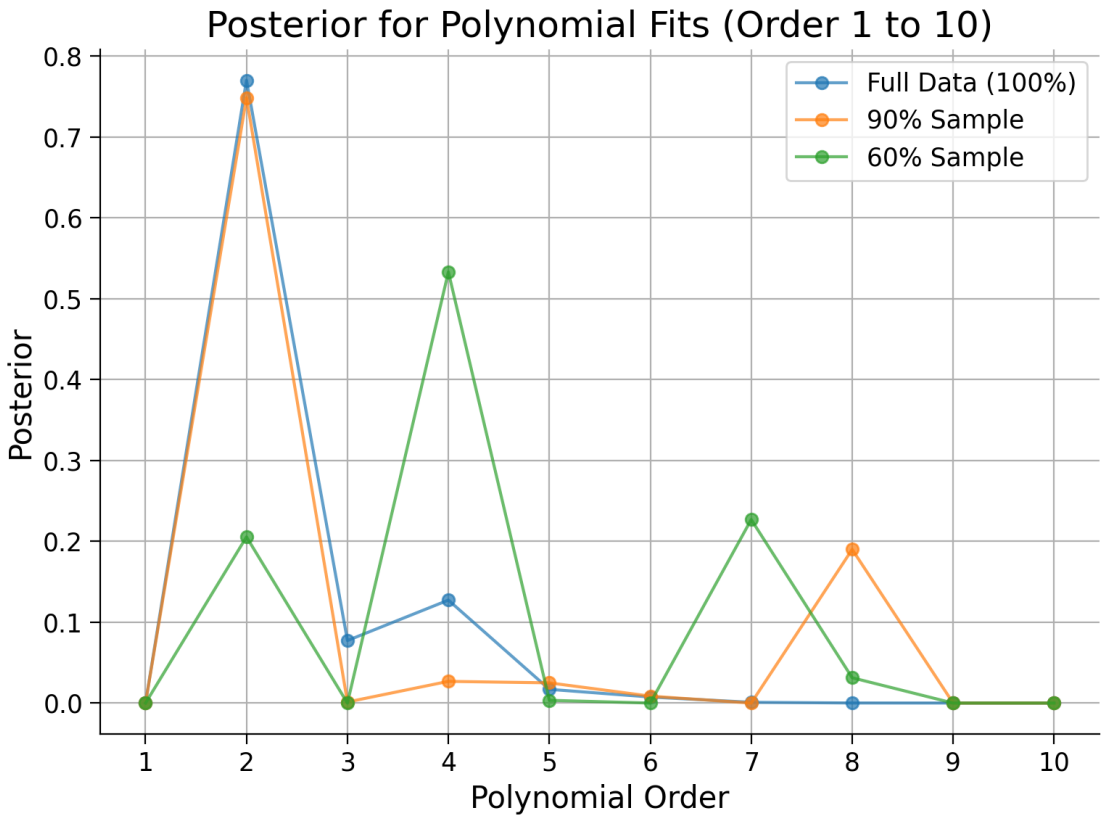


How do your results compare to the AIC results? What might be causing these differences? For full data and 90% data, order 2 has the lowest BIC, indicating that this order/model is more likely the "true" model of the data. This is different than the AIC results because while AIC uses a penalty of 2p for the number of parameters, BIC uses a penalty of p*log(N) which grows with the sample size N. This means BIC will penalize a model more if the sample size and number of parameters are big. That is why for full data and larger samples, BIC tends to favor simpler models more than AIC does. For 60% data, order 4 has the lowest BIC. This could be because smaller number of samples leads to greater variability. However, in this case, BIC still prefer a simpler model than AIC.

Question #11: Use your results question #10 to compute and plot the posterior distributions over the 10 models for all three cases. Hint: to keep your code from getting infinities, you can subtract the maximum value of the BIC from each of the values before computing the posterior. Are your results more or less certain as you increase the number of data points?

[ ]:
```
BIC_normalized = BIC - np.max(BIC)
posterior = np.exp(-0.5 * BIC_normalized) / np.sum(np.exp(-0.5 *␣
 ↪BIC_normalized))

BIC_60_normalized = BIC_60 - np.max(BIC_60)
posterior_60 = np.exp(-0.5 * BIC_60_normalized) / np.sum(np.exp(-0.5 *␣
 ↪BIC_60_normalized))
```

```
BIC_90_normalized = BIC_90 - np.max(BIC_90)
posterior_90 = np.exp(-0.5 * BIC_90_normalized) / np.sum(np.exp(-0.5 *␣
 ↪BIC_90_normalized))

plt.plot(range(1,11), posterior, marker='o', alpha=0.7, label='Full Data␣
 ↪(100%)')
plt.plot(range(1,11), posterior_90, marker='o', alpha=0.7, label='90% Sample')
plt.plot(range(1,11), posterior_60, marker='o', alpha=0.7, label='60% Sample')
plt.title('Posterior for Polynomial Fits (Order 1 to 10)')
plt.xlabel('Polynomial Order')
plt.ylabel('Posterior')
plt.xticks(range(1,11))
plt.legend()
plt.grid()
plt.show()
```



Are your results more or less certain as you increase the number of data points? Overall, as the number of data points increases, my results are more certain because the model estimates become more stable and lead to less uncertainty. For 60% data, there are several peaks, indicating several polynomial orders that perform better than others and a lot of uncertainty. For 90% data, there

are only two peaks, while one of the major peak is at order 2 (less uncertainty compared to 60% data). For full data, there is a major peak at order 2 and a small one at order 4. This suggests the full data has the least uncertainty.

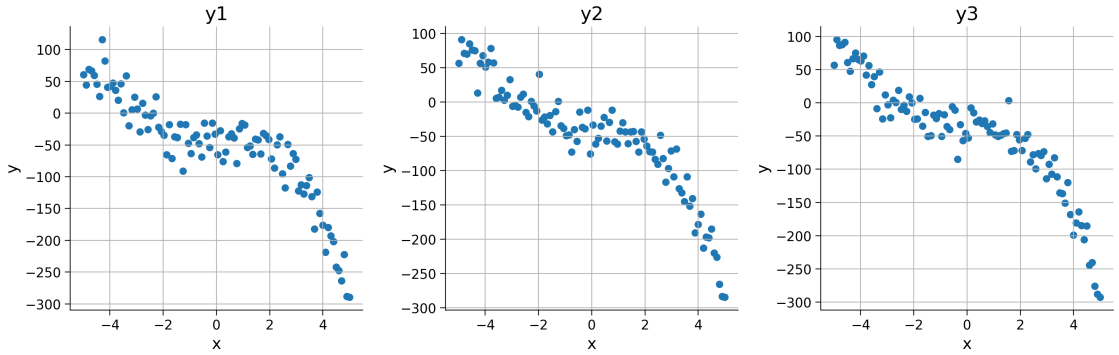
##Fitting unknown data

For this portion of the exercise, you will now be given data from an unknown polynomial that has been corrupted with an unknown amount of noise. Run the code below to import the data. `x` are the x-axis values, and `y1`, `y2`, and `y3` are three different instantiations of the data set (same polynomial, same amount of noise).

[ ]:
```
url = 'https://raw.githubusercontent.com/gordonberman/bioqtm385_fall2020/master/
 ↪data/test_data.csv'
test_data_df = pd.read_csv(url,header=None)
test_data = test_data_df.to_numpy()
x = test_data[:,0]
y1 = test_data[:,1]
y2 = test_data[:,2]
y3 = test_data[:,3]
```

Question #12: Using the data above, what can you say about the underlying polynomial that generated the data? Use model-selection-based arguments to back-up your assertions.

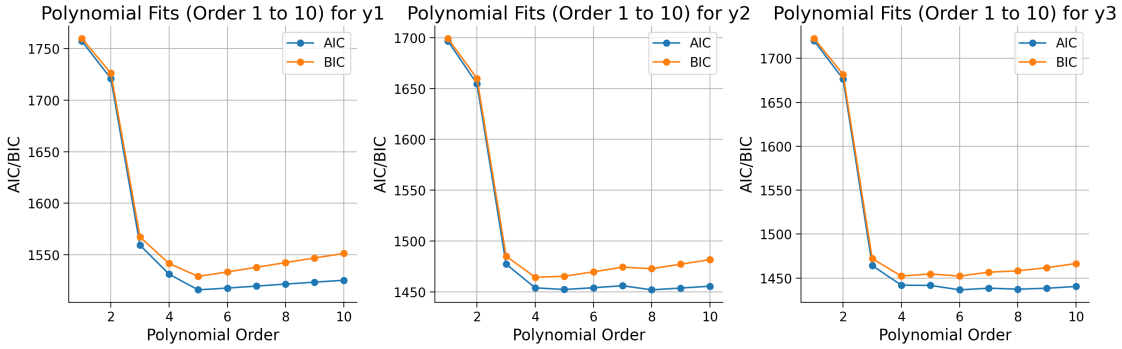
[ ]:
```
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

axs[0].scatter(x, y1)
axs[0].set_title('y1')
axs[0].set_xlabel('x')
axs[0].set_ylabel('y')
axs[0].grid()

axs[1].scatter(x, y2)
axs[1].set_title('y2')
axs[1].set_xlabel('x')
axs[1].set_ylabel('y')
axs[1].grid()

axs[2].scatter(x, y3)
axs[2].set_title('y3')
axs[2].set_xlabel('x')
axs[2].set_ylabel('y')
axs[2].grid()

plt.tight_layout()
plt.show()
```



[ ]:
```
AIC_y1 = []
for i in range(1,11):
  AIC_y1.append(-2*calculateLikelihood(x,y1,i) + 2*i)
print(f'Lowest AIC for y1: Order {np.argmin(AIC_y1)+1} with AIC = {AIC_y1[np.
 ↪argmin(AIC_y1)]}')

BIC_y1 = []
for i in range(1,11):
  BIC_y1.append(-2*calculateLikelihood(x,y1,i) + i*np.log(len(x)))
print(f'Lowest BIC for y1: Order {np.argmin(BIC_y1)+1} with BIC = {BIC_y1[np.
 ↪argmin(BIC_y1)]}')

AIC_y2 = []
for i in range(1,11):
  AIC_y2.append(-2*calculateLikelihood(x,y2,i) + 2*i)
print(f'Lowest AIC for y2: Order {np.argmin(AIC_y2)+1} with AIC = {AIC_y2[np.
 ↪argmin(AIC_y2)]}')

BIC_y2 = []
for i in range(1,11):
  BIC_y2.append(-2*calculateLikelihood(x,y2,i) + i*np.log(len(x)))
print(f'Lowest BIC for y2: Order {np.argmin(BIC_y2)+1} with BIC = {BIC_y2[np.
 ↪argmin(BIC_y2)]}')

AIC_y3 = []
for i in range(1,11):
  AIC_y3.append(-2*calculateLikelihood(x,y3,i) + 2*i)
print(f'Lowest AIC for y3: Order {np.argmin(AIC_y3)+1} with AIC = {AIC_y3[np.
 ↪argmin(AIC_y3)]}')

BIC_y3 = []
for i in range(1,11):
  BIC_y3.append(-2*calculateLikelihood(x,y3,i) + i*np.log(len(x)))
```

```
print(f'Lowest BIC for y3: Order {np.argmin(BIC_y3)+1} with BIC = {BIC_y3[np.
 ↪argmin(BIC_y3)]}')

fig, axs = plt.subplots(1, 3, figsize=(15, 5))

axs[0].plot(range(1,11), AIC_y1, marker='o', label='AIC')
axs[0].plot(range(1,11), BIC_y1, marker='o', label='BIC')
axs[0].set_title('Polynomial Fits (Order 1 to 10) for y1')
axs[0].set_xlabel('Polynomial Order')
axs[0].set_ylabel('AIC/BIC')
axs[0].legend()
axs[0].grid()

axs[1].plot(range(1,11), AIC_y2, marker='o', label='AIC')
axs[1].plot(range(1,11), BIC_y2, marker='o', label='BIC')
axs[1].set_title('Polynomial Fits (Order 1 to 10) for y2')
axs[1].set_xlabel('Polynomial Order')
axs[1].set_ylabel('AIC/BIC')
axs[1].legend()
axs[1].grid()

axs[2].plot(range(1,11), AIC_y3, marker='o', label='AIC')
axs[2].plot(range(1,11), BIC_y3, marker='o', label='BIC')
axs[2].set_title('Polynomial Fits (Order 1 to 10) for y3')
axs[2].set_xlabel('Polynomial Order')
axs[2].set_ylabel('AIC/BIC')
axs[2].legend()
axs[2].grid()

plt.tight_layout()
plt.show()
```

```
Lowest AIC for y1: Order 5 with AIC = 1515.9825911921844
Lowest BIC for y1: Order 5 with BIC = 1529.0084421221247
Lowest AIC for y2: Order 8 with AIC = 1452.0606547566015
Lowest BIC for y2: Order 4 with BIC = 1464.452461744376
Lowest AIC for y3: Order 6 with AIC = 1436.5117073981662
Lowest BIC for y3: Order 6 with BIC = 1452.1427285140949
```



What can you say about the underlying polynomial that generated the data? Use model-selection-based arguments to back-up your assertions. I think the underlying polynomial is likely of higher ordrer (between 4 and 6), with varying complexity depending on the specific instantiation. Since AIC aims to minimize prediction error and often favors more complex models and BIC applies stronger penalty for complexity as the sample size increases, it is surprising that AIC and BIC choose the same polynomial order for y1 and y3. But this indicates that 5th and 6th-degree polynomial could probably model the data well. However for y2, AIC chooses order 8 while BIC chooses order 4. The discrepancy with BIC may indicate that while 8th order fits well, it may overfit noise. The variations in prefered model order across different instantiations could due to variability or other factors other than noise since we are given that they have the same amount of noise.

##Spike sorting revisited

Use this code to load the spiking data from in-class exercise 3b.

[3]:
```
url = 'https://raw.githubusercontent.com/gordonberman/bioqtm385_fall2020/master/
  ↪data/spike_data.csv'
spike_data_df = pd.read_csv(url,header=None)
spike_data = spike_data_df.to_numpy()
np.shape(spike_data)
```

[3]:
```
(3636, 34)
```

Question #13: Revisit Question #2 from In-class Exercise 3b. Knowing what you know now, use ideas from model selection to estimate the number of neurons in the data set. Justify your answer. Note that `aic` and `bic` can be calculated directly from GaussianMixture fits.

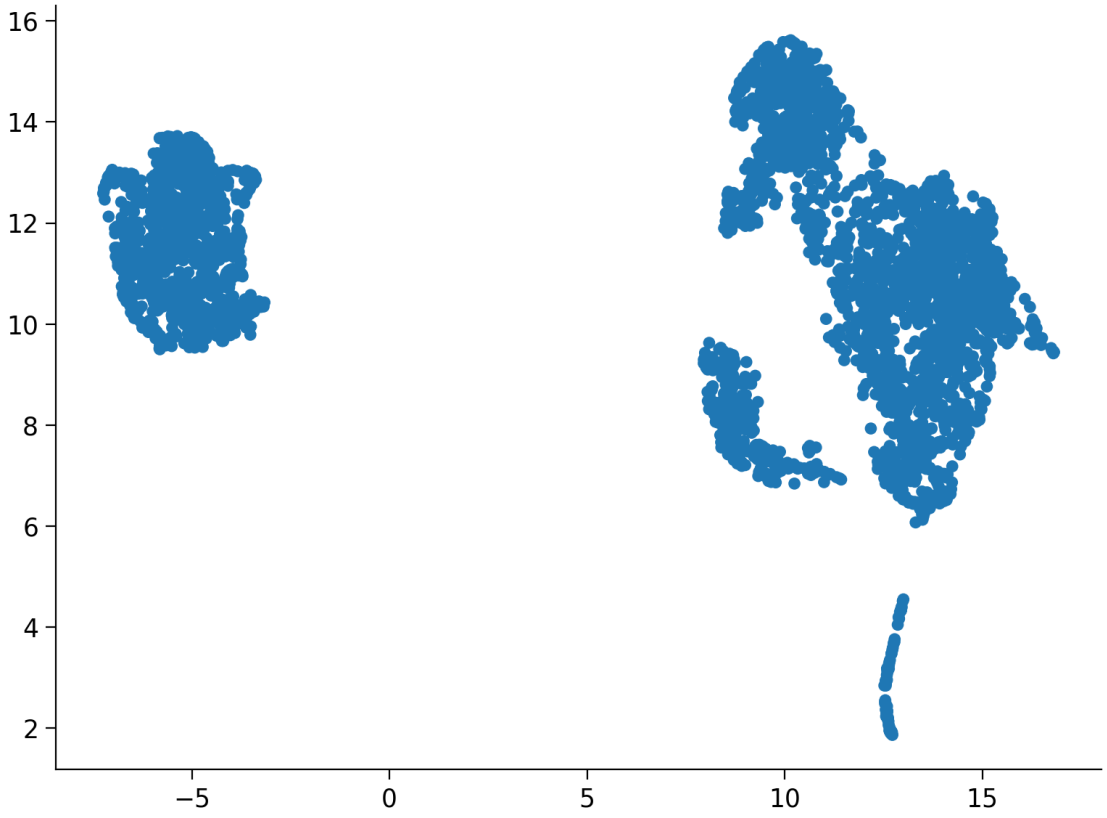
[4]:
```
# @title More Helper Functions


from matplotlib.patches import Ellipse

def draw_ellipse(position, covariance, ax=None, **kwargs):
    """Draw an ellipse with a given position and covariance"""
    ax = ax or plt.gca()
```

```
    # Convert covariance to principal axes
    if covariance.shape == (2, 2):
        U, s, Vt = np.linalg.svd(covariance)
        angle = np.degrees(np.arctan2(U[1, 0], U[0, 0]))
        width, height = 2 * np.sqrt(s)
    else:
        angle = 0
        width, height = 2 * np.sqrt(covariance)

    # Draw the Ellipse
    for nsig in range(1, 4):
        ax.add_patch(Ellipse(position, nsig * width, nsig * height,
                             angle, **kwargs))

def plot_gmm(gmm, X, label=True, ax=None):
    ax = ax or plt.gca()
    labels = gmm.fit(X).predict(X)
    if label:
        ax.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis', zorder=2)
    else:
        ax.scatter(X[:, 0], X[:, 1], s=40, zorder=2)
    ax.axis('equal')

    w_factor = 0.2 / gmm.weights_.max()
    for pos, covar, w in zip(gmm.means_, gmm.covariances_, gmm.weights_):
        draw_ellipse(pos, covar, alpha=w * w_factor)
```

[5]:
```
spikeUMAP = umap.UMAP(n_components=2, n_neighbors=10, min_dist=.1)
projections_umap = spikeUMAP.fit_transform(spike_data)
plt.scatter(projections_umap[:, 0], projections_umap[:, 1], edgecolor="none",␣
 ↪cmap=plt.colormaps['nipy_spectral'])
```

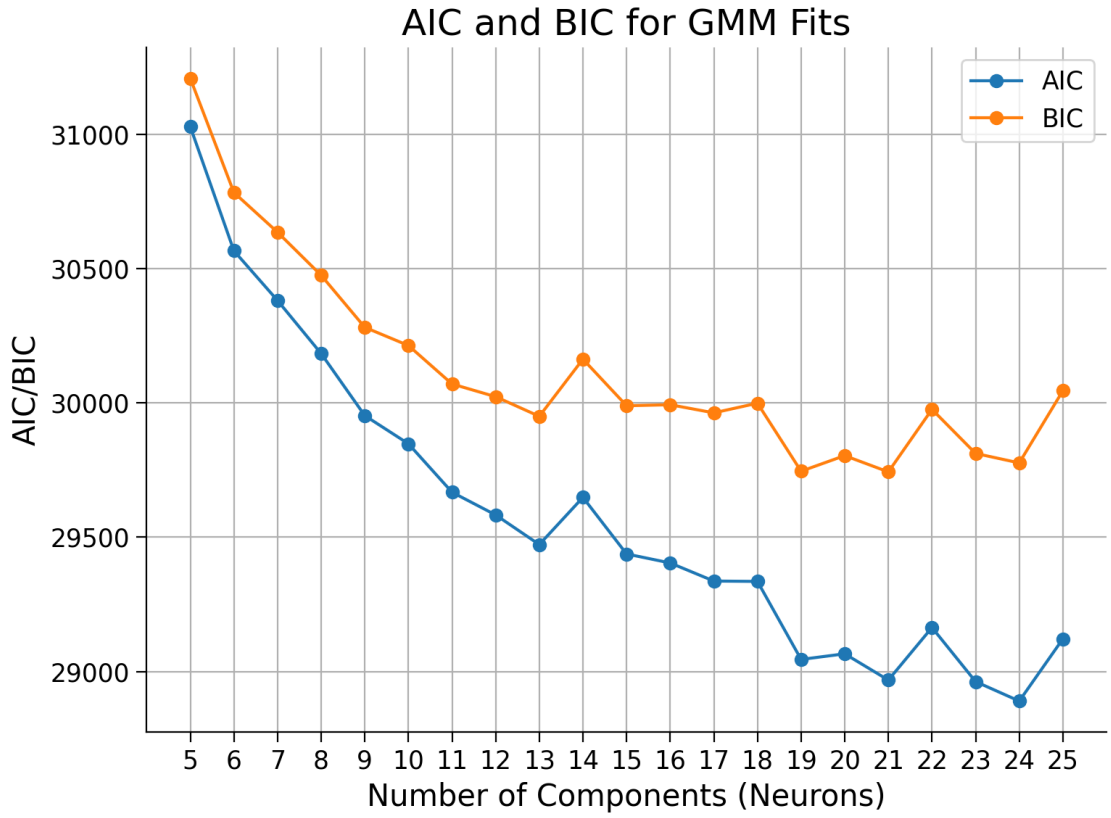
```
<ipython-input-5-79ce6a138bfd>:3: UserWarning: No data for colormapping provided
via 'c'. Parameters 'cmap' will be ignored
  plt.scatter(projections_umap[:, 0], projections_umap[:, 1], edgecolor="none",
cmap=plt.colormaps['nipy_spectral'])
```

[5]: `<matplotlib.collections.PathCollection at 0x7dda19e92dd0>`



[12]:
```
aic_umap = []
bic_umap = []
n_components = range(5, 26)

for n in n_components:
    gmm = GaussianMixture(n_components=n, n_init=5).fit(projections_umap)
    aic_umap.append(gmm.aic(projections_umap))
    bic_umap.append(gmm.bic(projections_umap))

plt.plot(n_components, aic_umap, marker='o', label='AIC')
plt.plot(n_components, bic_umap, marker='o', label='BIC')
plt.title('AIC and BIC for GMM Fits')
plt.xlabel('Number of Components (Neurons)')
plt.ylabel('AIC/BIC')
plt.xticks(n_components)
plt.legend()
plt.grid()
plt.show()

print(f'Lowest AIC: Order {np.argmin(aic_umap)+5} with AIC = {aic_umap[np.
  ↪argmin(aic_umap)]}')
```

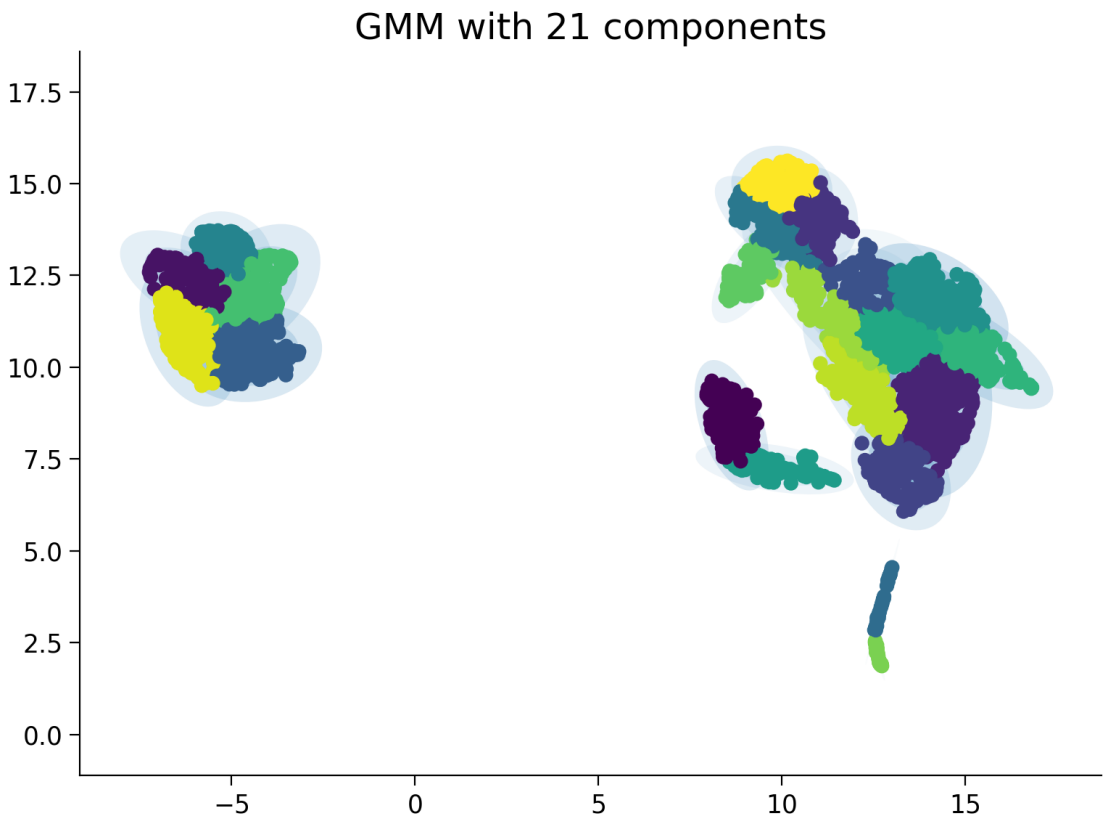
```
print(f'Lowest BIC: Order {np.argmin(bic_umap)+5} with BIC = {bic_umap[np.
  ↪argmin(bic_umap)]}')
```




```
Lowest AIC: Order 24 with AIC = 28890.27358875095
Lowest BIC: Order 21 with BIC = 29743.177357937428
```

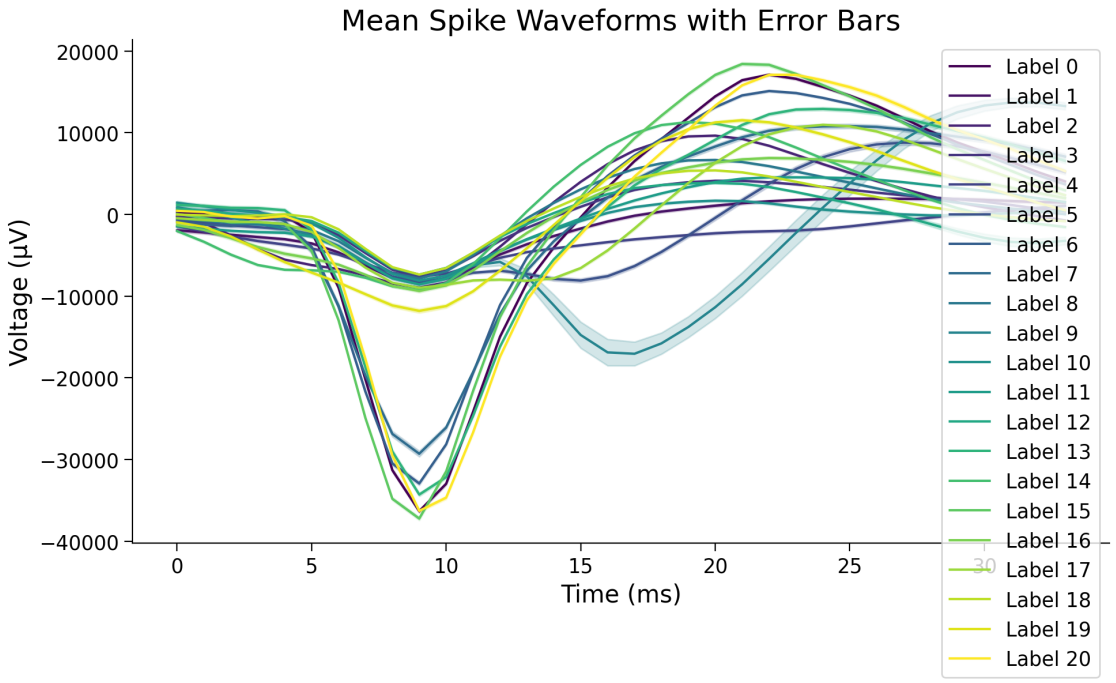
[13]:
```
gmm = GaussianMixture(n_components=21).fit(projections_umap)
gmm_labels = gmm.predict(projections_umap)
gmm_probs = gmm.predict_proba(projections_umap)
plot_gmm(gmm, projections_umap)
plt.title(f'GMM with 21 components')
plt.show()
```

```
<ipython-input-4-66766f42fbde>:21: MatplotlibDeprecationWarning: Passing the
angle parameter of __init__() positionally is deprecated since Matplotlib 3.6;
the parameter will become keyword-only two minor releases later.
  ax.add_patch(Ellipse(position, nsig * width, nsig * height,
```



[14]:
```
new_spikes = np.hstack((spike_data, gmm_labels.reshape(-1, 1)))

# Define colors for GMM labels for consistency
unique_labels = np.unique(gmm_labels)
colors = plt.cm.viridis(np.linspace(0, 1, len(unique_labels)))
label_color_map = {label: colors[i] for i, label in enumerate(unique_labels)}

mean_waveforms = {}
sem_waveforms = {}
for label in unique_labels:
    label_indices = np.where(gmm_labels == label)[0]
    mean_waveforms[label] = np.mean(new_spikes[label_indices, :-1], axis=0)
    sem_waveforms[label] = np.std(new_spikes[label_indices, :-1], axis=0) / np.
 ↪sqrt(len(label_indices))

# Plot mean waveforms with error bars
plt.figure(figsize=(10, 6))
for label in unique_labels:
    plt.plot(mean_waveforms[label], color=label_color_map[label], label=f'Label␣
 ↪{label}')
    plt.fill_between(np.arange(mean_waveforms[label].shape[0]),
```

```
                    mean_waveforms[label] - sem_waveforms[label],
                    mean_waveforms[label] + sem_waveforms[label],
                    color=label_color_map[label], alpha=0.2)

plt.xlabel('Time (ms)')
plt.ylabel('Voltage ( V)')
plt.title('Mean Spike Waveforms with Error Bars')
plt.legend()
plt.show()
```



Knowing what you know now, use ideas from model selection to estimate the number of neurons in the data set. Justify your answer. According to the BIC values (using BIC instead of AIC because it's a better estimate), they identify 21 neurons in the dataset since the BIC value for this is the lowest. However, by plotting the GMM plot for n_components = 21 and the spike waveforms' means and error bars based on the GMM labels, it seems like this number is way higher than we expected. There are no distinct clusters on the GMM plot, and the mean and error bar plot also doesn't clearly separate the different neurons (a lot of overlapping). This is possible because one neuron may just has a slightly different firing waveform based on how exactly it is firing, which would cause AIC/BIC to think there are more neurons than there actualy are. This is also possible because we are calculating AIC/BIC using GMM. If we assume the underlying data is Gaussian, we would need this many parameters to fit. But in reality, the underlying data structure may not be fully Gaussian. So the number of parameters BIC suggested may not be right.

Instructions on how to save PDF files: (repeated from last time)

To make nice looking files to hand-in (which makes grading much easier!), please follow the instruc-