MTH 602 Scientific Machine Learning

Homework 4

10/27/2025

S. M. Mahfuzul Hasan

02181922

UMass | Dartmouth

I. PAPER & PENCIL WORK

1. (i) A positive definite matrix of the size $D \times D$ is symmetric, and can be written in the form:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & \dots & a_{1D} \\ a_{12} & a_{22} & a_{23} & \dots & \dots & a_{2D} \\ a_{13} & a_{23} & a_{33} & \dots & \dots & a_{3D} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ a_{1D} & a_{2D} & a_{3D} & \dots & \dots & a_{DD} \end{bmatrix}$$

The no. of diagonal elements in A are D and the no. of off-diagonal elements are $(D^2 - D)$, but since this is a symmetric matrix, only half of them will be independent. As a result, the no. of the independent off-diagonal elements will be $\left(D^2 - D\right)/2$. So, the total no. of independent parameters is

$$D + \frac{D^2 - D}{2} = \frac{D\,(D+1)}{2} \quad \text{(Ans.)}$$

(ii) The no. of independent parameters in a D-dimensional Gaussian should be the sum of total parameters in symmetric $D \times D$ co-variance matrix, $\mathbf{\Sigma}$ and D-dimensional mean vector, $\boldsymbol{\mu}$.

No. of independent parameters in $\mathbf{\Sigma} = D\,(D+1)\,/2$ [as obtained from (i)].
No. of independent parameters in $\boldsymbol{\mu} = D$ [since, its dimension is D].

So, total no. of independent parameters in a D-dimensional Gaussian is

$$\frac{D\,(D+1)}{2} + D = \frac{D\,(D+3)}{2} \quad \text{(Ans.)}$$

(iii) Gaussian mixture model (GMM) can be written as,

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}\left(\mathbf{x} | \boldsymbol{\mu}_k, \mathbf{\Sigma}_k\right)$$

where, π_k are mixing coefficients and $\sum\limits_{k=1}^{K} \pi_k = 1$.

Since, each D-dimensional Gaussian has $\frac{D(D+3)}{2}$ independent parameters, GMM with K components will have total independent parameters as a summation of the independent parameters due to GMM and the mixing coefficients. Since, the total K mixing coefficients sums up to 1, the no. of independent coefficients will be the degree of freedom, which is $(K-1)$. So, the total no. of independent parameters are

$$K\frac{D\,(D+3)}{2} + (K-1) \quad \text{(Ans.)}$$

II. CHAPTER 3: MULTIVARIATE GAUSSIAN MIXTURE MODELING

A. Stage 1 — One Gaussian: MLE vs. GMM with 1 component

Given ground-truth Gaussian:

$$\mu = (2,\ -1), \qquad \Sigma = \begin{bmatrix} 2 & 0.8 \\ 0.8 & 1 \end{bmatrix}$$

1. Computed eigen-values:

$$\lambda_1 = 2.44339811320566, \qquad \lambda_2 = 0.5566018867943396$$

Computed eigen-vectors:

$$\mathbf{u}_1 = \begin{pmatrix} -0.8746424812468178 \\ -0.4847685323929453 \end{pmatrix}, \qquad \mathbf{u}_2 = \begin{pmatrix} 0.4847685323929453 \\ -0.8746424812468178 \end{pmatrix}$$

Principal axes and orientation of ellipse:

$$2{\lambda_1}^{\frac{1}{2}} = 3.1262745325423102, \qquad 2{\lambda_1}^{\frac{1}{2}} = 1.492115125309491, \qquad \theta = 28.997308395958242^\circ$$

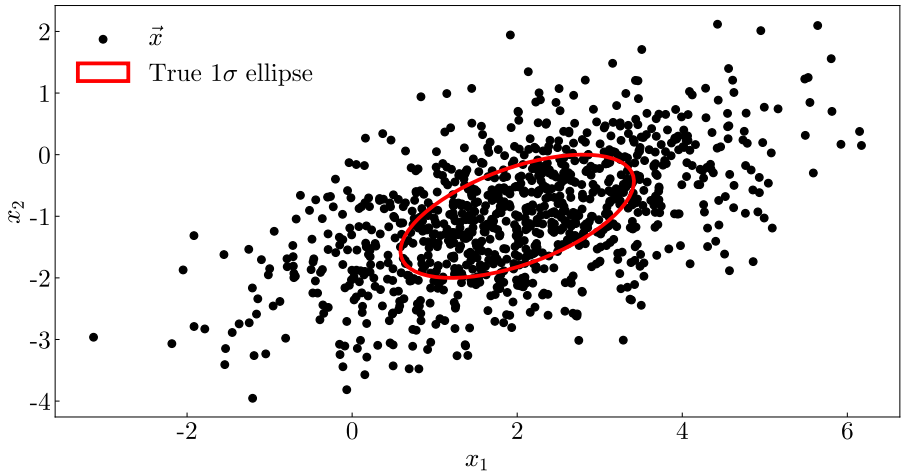


Figure 1: 1000 samples $\vec{x} \sim \mathcal{N}(\mu, \Sigma)$ drawn using `sample_2d_gaussian` along with true 1σ ellipse of Σ.

2. Analytic maximum likelihood solution:

$$\hat{\mu} = (2.014655285377038,\ -1.0132083392419486),$$

$$\hat{\Sigma} = \begin{bmatrix} 2.069533458212539 & 0.7855473289101127 \\ 0.7855473289101127 & 0.9810586095342764 \end{bmatrix}$$

Percentage relative error compared to ground-truth solution:

$$\delta\mu = \frac{|\hat{\mu}_i - \mu_i|}{|\mu_i|} \times 100\% = (0.7327642688518932\%,\ 1.3208339241948641\%)\,,$$

$$\hat{\Sigma} = \frac{|\hat{\Sigma}_{ij} - \Sigma_{ij}|}{|\Sigma_{ij}|} \times 100\% = \begin{bmatrix} 3.4766729106269434\% & 1.8065838862359185\% \\ 1.8065838862359185\% & 1.8941390465723629\% \end{bmatrix}$$

The analytic maximum likelihood solution is close to the ground truth solution within a comparatively small upper error bound of $\sim 1.5\%$ for mean and $\sim 3.5\%$ for co-variance.

3. Fitted one-component GMM:

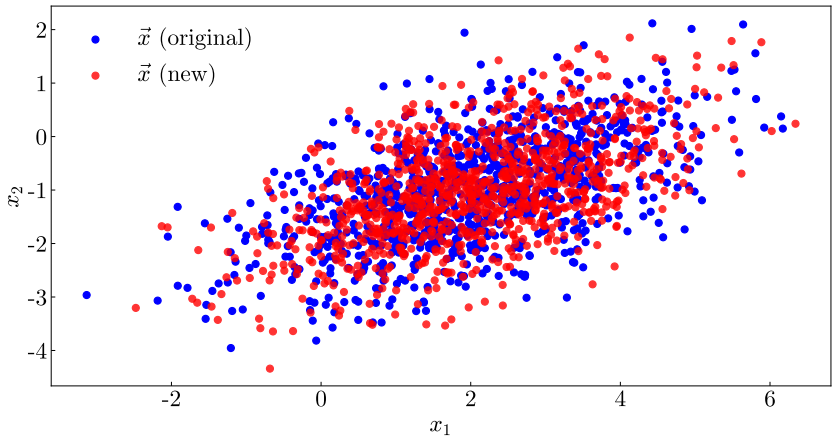
$$\mu_{gmm} = (2.014655285377037,\ -1.0132083392419482),$$
$$\Sigma_{gmm} = \begin{bmatrix} 2.0695344582125386 & 0.7855473289101127 \\ 0.7855473289101127 & 0.9810596095342764 \end{bmatrix}$$

Comparison to the maximum likeligood solution:

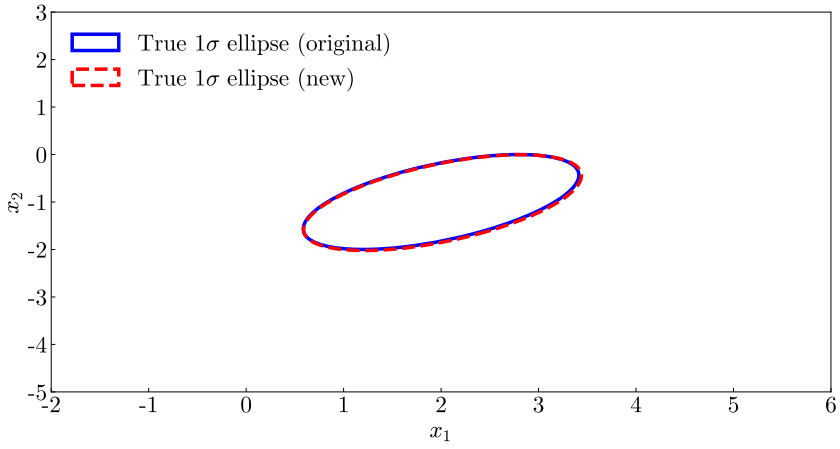
$$\|\hat{\mu} - \mu_{gmm}\|_2 = 9.930136612989092 \times 10^{-16}, \quad \|\hat{\Sigma} - \Sigma_{gmm}\|_2 = 1.0000000000287557 \times 10^{-06}$$

Agreement is very close.

4. New points ($M = 1000$) are drawn from the fitted one-component GMM model.



(a)



(b)

Figure 2: (a) Overlay of new samples on the original samples. (b) True 1σ ellipse of Σ for new and original samples.

B. Stage 2 — Two-component mixture in 2D: Fit, Compare, Sample, Classify

Given ground truth mixture:

$$\omega = (0.2,\ 0.8); \quad \mu_1 = (0,\ 0), \ \ \Sigma_1 = \begin{bmatrix} 1 & 0.6 \\ 0.6 & 1.5 \end{bmatrix}; \quad \mu_2 = (4,\ 3), \ \ \Sigma_2 = \begin{bmatrix} 1.2 & -0.5 \\ -0.5 & 0.8 \end{bmatrix}$$

Observed pulsar population:

$$P(\vec{x}) = 0.2\mathcal{N}\left(\vec{x}|\mu_1, \Sigma_1\right) + 0.8\mathcal{N}\left(\vec{x}|\mu_2, \Sigma_2\right)$$

1. Sampling of the two components and their corresponding Gaussian $\vec{x} \sim \mathcal{N}\left(\mu_1, \Sigma_1\right)$ and $\vec{x} \sim \mathcal{N}\left(\mu_2, \Sigma_2\right)$ respectively:

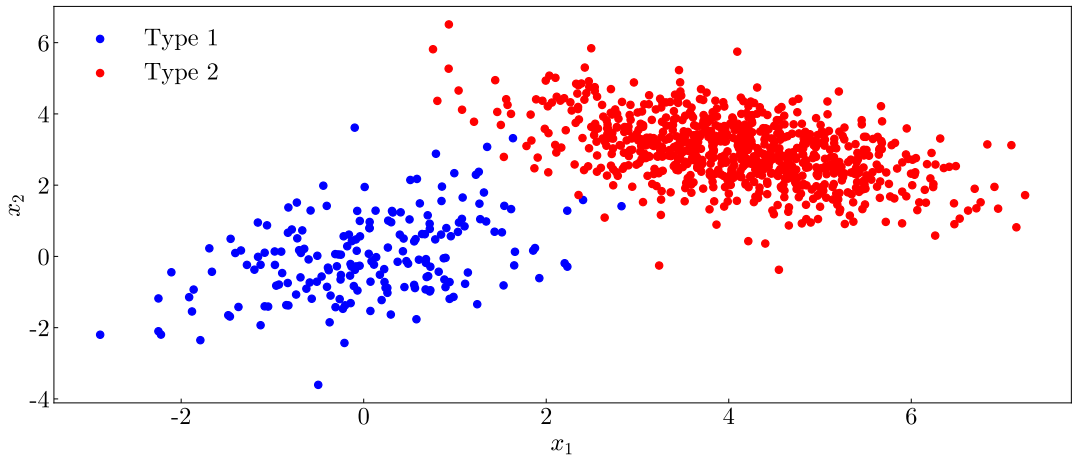


Figure 3: Total 1000 samples drawn from $\vec{x} \sim \mathcal{N}\left(\mu_1, \Sigma_1\right)$ and $\vec{x} \sim \mathcal{N}\left(\mu_2, \Sigma_2\right)$ for type-1 and type-2 pulsar using `sample_2d_gaussian`.

2. Fitted GMM for $K = 1$:

$$\mu_{gmm} = (3.3043244939443674,\ 2.3696992901480693),$$

$$\Sigma_{gmm} = \begin{bmatrix} 3.6926089015268726 & 1.561316587647234 \\ 1.561316587647234 & 2.25828069278701 \end{bmatrix}$$

Fitted GMM for $K = 2$:

$$\mu_{1,gmm} = (0.0887631709875105,\ -0.0210976144553744),$$

$$\Sigma_{1,gmm} = \begin{bmatrix} 1.0761406229476318 & 0.476672845843364 \\ 0.476672845843364 & 1.229744902269525 \end{bmatrix};$$

$$\mu_{2,gmm} = (4.0816341486534675,\ 2.9476355599773143),$$

$$\Sigma_{2,gmm} = \begin{bmatrix} 1.2214003949426913 & -0.4841133862236404 \\ -0.4841133862236405 & 0.7911739159757286 \end{bmatrix}$$

It is easily evident that $K = 2$ produces the means and co-variances that are closer to the ground truth means and co-variances.

Diagnostics for these two models are compared using log-likelihood and Bayesian information criterion diagonstics:

$$\text{log-likelihood}_{K=1} : -3725.457633, \quad \text{log-likelihood}_{K=2} : -3200.461304$$

$$\text{Bayesian information criterion}_{K=1} : 7485.454043,$$

$$\text{Bayesian information criterion}_{K=2} : 6476.907916$$

Higher value for the log-likelihood and lower value for Bayesian information criterion are desired for the preferred model. Since, log-likehood is greater for $K = 2$ than $K = 1$ and Bayesian information criterion is smaller for $K = 2$ than $K = 1$, GMM for $K = 2$ is the better and preferred model.

3. For $K = 2$,

Weights:

$$\omega_1 = 0.1946743731658157, \quad \omega_2 = 0.8053256268341843$$

Means:

$$\hat{\mu}_1 = (0.0887631709875105, \ -0.0210976144553744),$$

$$\hat{\mu}_2 = (4.0816341486534675, \ 2.9476355599773143)$$

Predicted $\hat{\mu}_k$ is close to true μ_k. The maximum deviation is $\sim \pm 0.09$.

Co-variances:

$$\hat{\Sigma}_1 = \begin{bmatrix} 1.0761406229476318 & 0.476672845843364 \\ 0.476672845843364 & 1.229744902269525 \end{bmatrix};$$

$$\hat{\Sigma}_2 = \begin{bmatrix} 1.2214003949426913 & -0.4841133862236404 \\ -0.4841133862236405 & 0.7911739159757286 \end{bmatrix}$$

Predicted $\hat{\Sigma}_2$ gives very good estimation to true Σ_2 while Predicted $\hat{\Sigma}_1$ is a bit off compared to true Σ_1, which can be attributed to the small sample size for that component.

4. New sampled data cover type-2 mode really well compared to the true sampled data as can

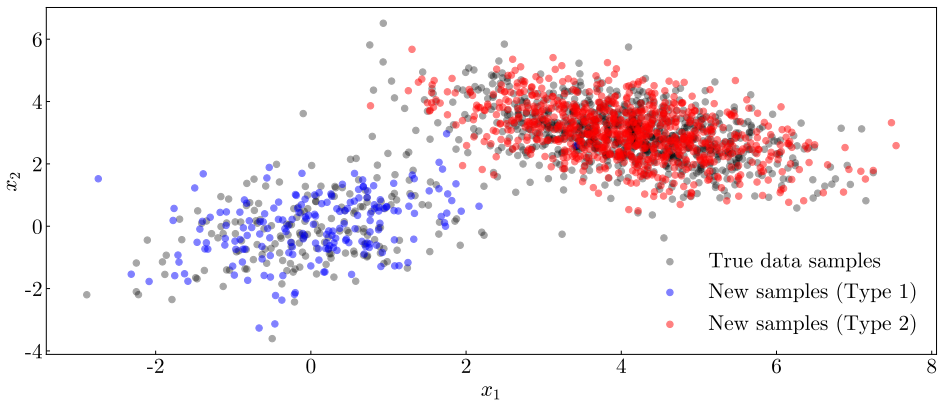


Figure 4: Samples drawn from fitted GMM.

be seen from figure 4. Type-1 mode is also covered well but it is not covered as good as type-2. However, the overlap between the two modes is minimal which is a good sign for classification.

5. Classifier's accuracy = 0.998 = 99.8%

	Predicted: Type-1	Predicted: Type-2
Actual: Type-1	184	1
Actual: Type-2	1	814

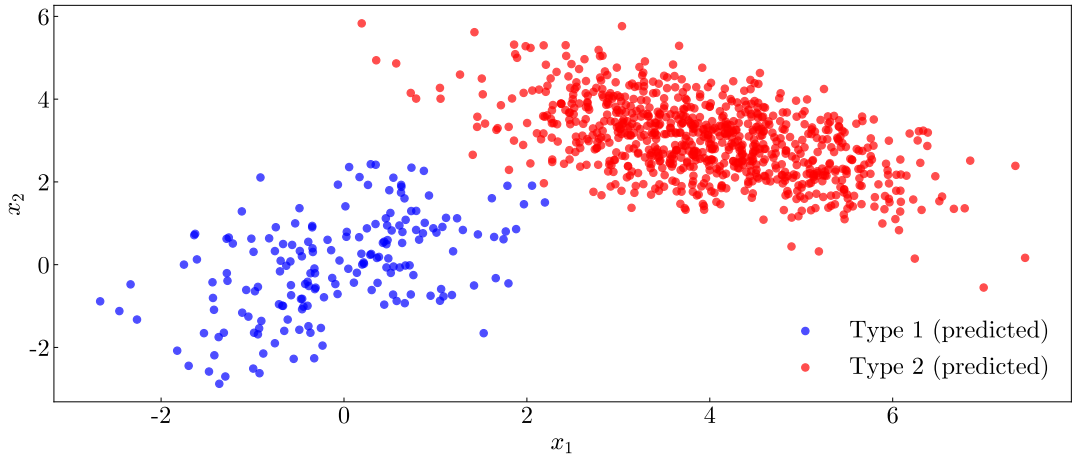


Figure 5: Visual check for classification of pulsar type.

The two-component GMM classifier is working really good with only two misclassifications out of 1000 data samples. It falsely classifies only one type-1 out of 185 samples, and one type-2 as type-1 out of 815 samples, which is a great prediction accuracy.

```
from sklearn.mixture import GaussianMixture
from sklearn.metrics import confusion_matrix, accuracy_score
from scipy.special import logsumexp
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.patches import Ellipse

# ===== II. CHAPTER 3: MULTIVARIATE GAUSSIAN MIXTURE MODELING =====
def sample_2d_gaussian(mu, Sigma, n, rng=None):
    if rng is None:
        rng = np.random.default_rng()
    return rng.multivariate_normal(mean=mu, cov=Sigma, size=n)

# ===== A. STAGE 1: ONE GAUSSIAN: MLE VS. GMM WITH 1 COMPONENT =====
mu = np.array([2, -1])
Sigma = np.array([[2, 0.8], [0.8, 1]])
rng = np.random.default_rng(29)
n = 1000

# ===== 1. Visualization =====
# sampling
x = sample_2d_gaussian(mu, Sigma, n, rng)

np.set_printoptions(precision=16, suppress=False)
```

```
print("1.")
print("x: ", x)

# compute eigenvalues and eigenvectors, and sort them
eigen_val, eigen_vec = np.linalg.eigh(Sigma)
order = np.argsort(eigen_val)[::-1] # descending order
eigen_val = eigen_val[order]
eigen_vec = eigen_vec[:, order]

print("\nEigen values: ", eigen_val)
print("\nEigen vectors: ", eigen_vec)

# $1\sigma$ ellipse parameters
wid  = 2*np.sqrt(eigen_val[0]) # major axis
hei = 2*np.sqrt(eigen_val[1]) # minor axis
angle  = np.degrees(np.arctan2(eigen_vec[1,0], eigen_vec[0,0])) % 180 #
    orientation

print("\nWidth: ", wid)
print("\nHeight: ", hei)
print(f"\nAngle: {angle} degree")

# parameters for plotting
plt.rcParams["font.family"] = "serif"
plt.rcParams["font.serif"] = ["CMU Serif"]
plt.rcParams["mathtext.fontset"] = "cm"
plt.rcParams["font.size"] = 20
mpl.rcParams["axes.unicode_minus"] = False

# scatter plot (with $1\sigma$ of $\Sigma$)
fig, ax = plt.subplots(figsize=(12, 6))
ax.scatter(x[:,0], x[:,1], color="black", label=r"$\vec{x}$")
ax.add_patch(Ellipse(mu, wid, hei, angle=angle, edgecolor='r', fill=False, lw
    =3, label=r"True 1$\sigma$ ellipse"))
plt.xlabel(r"$x_1$")
plt.ylabel(r"$x_2$")
plt.legend(loc="upper left", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig(f"x_sample.pdf", dpi=1080)
plt.show()

# ===== 2. Analytic maximum-likelihood solution =====
mu_hat = x.mean(axis=0)
Sigma_hat = ((x - mu_hat).T @ (x - mu_hat)) / len(x)
print("\n2.")
print(f"Maximum likelihood: mu_hat = ", mu_hat)
print(f"\nMaximum likelihood: Sigma_hat = ", Sigma_hat)

# ===== 3. GMM with one component =====
g1 = GaussianMixture(n_components=1, covariance_type='full', n_init=10,
    random_state=0).fit(x) # one-component fit
mu_gmm = g1.means_[0] # mean
Sigma_gmm = g1.covariances_[0] #covariance
print("\n3.")
print(f"GMM w/ one component: mu_gmm = ", mu_gmm)
print(f"\nGMM w/ one component: Sigma_gmm = ", Sigma_gmm)

mu_err = np.linalg.norm(mu_hat - mu_gmm) # L2 norm
Sigma_err = np.linalg.norm(Sigma_hat - Sigma_gmm, 2) # matrix 2-norm
```

```
print(f"\nmu_error:", mu_err)
print(f"\nSigma_error:", Sigma_err)

# ===== 4. Sampling from the fitted model =====
m = 1000
x_new, _ = g1.sample(m)

print("\n4.")
print("x (new): ", x_new)

# compute eigenvalues and eigenvectors, and sort them
eigen_val_new, eigen_vec_new = np.linalg.eigh(Sigma_gmm)
order_new = np.argsort(eigen_val_new)[::-1]
eigen_val_new = eigen_val_new[order_new]
eigen_vec_new = eigen_vec_new[:, order_new]

print("\nEigen values (new): ", eigen_val_new)
print("\nEigen vectors (new): ", eigen_vec_new)

# $1\sigma$ ellipse parameters
wid_new  = 2 * np.sqrt(eigen_val_new[0])
hei_new = 2 * np.sqrt(eigen_val_new[1])
angle_new  = np.degrees(np.arctan2(eigen_vec_new[1,0], eigen_vec_new[0,0])) %
    180

print("\nWidth (new): ", wid_new)
print("\nHeight (new): ", hei_new)
print(f"\nAngle (new): {angle_new} degree")

# scatter plot (with $1\sigma$ of $\Sigma$)
fig, ax = plt.subplots(figsize=(12, 6))
ax.scatter(x[:,0], x[:,1], color="blue", label=r"$\vec{x}$ (original)")
ax.scatter(x_new[:,0], x_new[:,1], color="red", alpha=0.8, label=r"$\vec{x}$ (
    new)")
plt.xlabel(r"$x_1$")
plt.ylabel(r"$x_2$")
plt.legend(loc="upper left", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig(f"x_valid1.pdf", dpi=1080)
plt.show()

# ellipses
fig, ax = plt.subplots(figsize=(12, 6))
ax.add_patch(Ellipse(mu, wid, hei, angle=angle, edgecolor="blue", fill=False,
    lw=3, label=r"True 1$\sigma$ ellipse (original)"))
ax.add_patch(Ellipse(mu_gmm, wid_new, hei_new, linestyle="--", angle=angle,
    edgecolor="red", fill=False, lw=3, label=r"True 1$\sigma$ ellipse (new)"))
ax.set_xlim(mu[0] - 4, mu[0] + 4)
ax.set_ylim(mu[1] - 4, mu[1] + 4)
plt.xlabel(r"$x_1$")
plt.ylabel(r"$x_2$")
plt.legend(loc="upper left", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig(f"x_valid2.pdf", dpi=1080)
plt.show()

# ===== STAGE 2: TWO-COMPONENT MIXTURE IN 2D: FIT, COMPARE, SAMPLE, CLASSIFY
    =====
w_true = np.array([0.2, 0.8])
```

```
mu1 = np.array([0.0, 0.0])
Sigma1 = np.array([[1.0, 0.6], [0.6, 1.5]])
mu2 = np.array([4.0, 3.0])
Sigma2 = np.array([[1.2, -0.5], [-0.5, 0.8]])

z = rng.choice([0, 1], size=n, p=[0.2, 0.8])
x_ = np.zeros((n, 2))

# ===== 1. Visualization =====
# sampling
for k in [0, 1]:
    n_ = np.sum(z==k)
    x_[z==k] = sample_2d_gaussian([mu1, mu2][k], [Sigma1, Sigma2][k], n_, rng)

# scatter plot
fig, ax = plt.subplots(figsize=(15, 6))

ax.scatter(x_[z==0, 0], x_[z==0, 1], color='blue', label='Type 1')
ax.scatter(x_[z==1, 0], x_[z == 1, 1], c='red', label='Type 2')

plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
plt.legend(loc="upper left", frameon=False)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig(f'x_sample2.pdf', dpi=1080)
plt.show()

# ===== 2. Modeling =====
# fit GMMs
g1 = GaussianMixture(n_components=1, covariance_type='full', random_state=42)
g1.fit(x_)
print("\n2. ")
print("For K=1,\nMean:", g1.means_)
print("\nCovariance:", g1.covariances_)

g2 = GaussianMixture(n_components=2, covariance_type='full', random_state=42)
g2.fit(x_)
print("\nFor K=2,\nMean:", g2.means_)
print("\nCovariance:", g2.covariances_)

# compute diagnostics
logL1, bic1 = g1.score(x_) * len(x_), g1.bic(x_)
logL2, bic2 = g2.score(x_) * len(x_), g2.bic(x_)

print(f"\nK=1: log-likelihood = {logL1:.16f}, Bayesian information criterion =
    {bic1:.16f}")
print(f"\nK=2: log-likelihood = {logL2:.16f}, Bayesian information criterion =
    {bic2:.16f}")

# ===== 3. Comparisons =====
means2, covs2, weights2 = g2.means_, g2.covariances_, g2.weights_
print("\n3. ")
print("For K = 2, ")
print(f"\nWeights: ", weights2)
print(f"\nMeans: ", means2)
print(f"\nCovariances: ", covs2)

# ===== Sampling from the fitted model =====
x1_new, z_new = g2.sample(m)
```

```
193
194 # scatter plot
195 fig, ax = plt.subplots(figsize=(15, 6))
196 ax.scatter(x_[:,0], x_[:,1], color="black", alpha=0.35, label="True data
    samples")
197 ax.scatter(x1_new[z_new==0, 0], x1_new[z_new==0, 1], color="blue", alpha=0.5,
    label="New samples (Type 1)")
198 ax.scatter(x1_new[z_new==1, 0], x1_new[z_new==1, 1], c="red", alpha=0.5, label
    ="New samples (Type 2)")
199
200 plt.xlabel(r"$x_1$")
201 plt.ylabel(r"$x_2$")
202 plt.legend(loc="lower right", frameon=False)
203 plt.tick_params(axis="both", which="both", direction="in")
204 plt.savefig(f"x_sample_from_fit.pdf", dpi=1080)
205 plt.show()
206
207 # ===== 5. Classification =====
208 # setup
209 n_test = 1000
210 rng = np.random.default_rng(129)
211 z_true = rng.choice([0, 1], size=n_test, p=w_true)
212 x_test = np.zeros((n_test, 2))
213
214 # constructing ground truth by sampling and testing the GMM model
215 for k in [0, 1]:
216     n_ = np.sum(z_true==k)
217     x_test[z_true==k] = sample_2d_gaussian([mu1, mu2][k], [Sigma1, Sigma2][k],
          n_, rng)
218
219 predicted_labels = g2.predict(x_test)
220
221 print("\n5. ")
222 #print(f"Predicted labels: ", predicted_labels)
223
224 # plots
225 fig, ax = plt.subplots(figsize=(15, 6))
226 ax.scatter(x_test[predicted_labels==0,0], x_test[predicted_labels==0,1], color
    ="blue",
227            alpha=0.7, label="Type 1 (predicted)")
228 ax.scatter(x_test[predicted_labels==1,0], x_test[predicted_labels==1,1], color
    ="red",
229            alpha=0.7, label="Type 2 (predicted)")
230
231 plt.xlabel(r"$x_1$")
232 plt.ylabel(r"$x_2$")
233 plt.legend(loc="lower right", frameon=False)
234 plt.tick_params(axis="both", which="both", direction="in")
235 plt.savefig(f"class.pdf", dpi=1080)
236 plt.show()
237
238 # compute accuracy and build confusion matrix
239 acc = accuracy_score(z_true, predicted_labels)
240 cm  = confusion_matrix(z_true, predicted_labels)
241
242 print(f"\nClassification accuracy: {acc:.3f}")
243 print("\nConfusion matrix:\n", cm)
```

Listing 1: *gmm.py*

```
1.
x:  [[ 2.517704588970049  -0.6704820620140608]
 [ 2.286594412408074  -0.9844312171306702]
 [ 1.9960242828701187 -1.1947318652011147]
 ...
 [ 1.0688979908502234 -1.3148043634736226]
 [ 1.940267350973554  -0.1494367115551481]
 [ 1.7265282647654883 -0.3770063494557934]]

Eigen values:  [2.44339811320566   0.5566018867943396]

Eigen vectors:  [[-0.8746424812468178  0.4847685323929453]
 [-0.4847685323929453 -0.8746424812468178]]

Width:  3.1262745325423102

Height:  1.492115125309491

Angle: 28.997308395958242 degree

2.
Maximum likelihood: mu_hat =  [ 2.014655285377038  -1.0132083392419486]

Maximum likelihood: Sigma_hat =  [[2.069533458212539  0.7855473289101127]
 [0.7855473289101127 0.9810586095342764]]

3.
GMM w/ one component: mu_gmm =  [ 2.014655285377037  -1.0132083392419482]

GMM w/ one component: Sigma_gmm =  [[2.0695344582125386 0.7855473289101127]
 [0.7855473289101127 0.9810596095342764]]

mu_error: 9.930136612989092e-16

Sigma_error: 1.0000000000287557e-06

4.
x (new):  [[-5.8688379488870535e-01 -2.0347950258630947e+00]
 [-1.3568660671935939e-01 -2.3019255565739227e-01]
 [-2.4897321278945261e-01 -3.0313846599294836e+00]
 ...
 [ 1.7043925473325234e+00 -1.0924208747152311e+00]
 [ 3.6037557851935631e-03 -1.9314825003720597e+00]
 [ 4.0670328613369993e+00 -1.0552435599485150e+00]]

Eigen values (new):  [2.4809533423908077 0.5696407253560073]

Eigen vectors (new):  [[-0.8858585514333069  0.4639554147248238]
 [-0.4639554147248238 -0.8858585514333069]]

Width (new):  3.150208464461238

Height (new):  1.509490941153351

Angle (new): 27.64263921232103 degree

2.
For K=1,
```

```
Mean: [[3.3043244939443674 2.3696992901480693]]

Covariance: [[[3.6926089015268726 1.561316587647234 ]
  [1.561316587647234  2.25828069278701  ]]]

For K=2,
Mean: [[ 0.0887631709875105 -0.0210976144553744]
 [ 4.0816341486534675  2.9476355599773143]]

Covariance: [[[ 1.0761406229476318  0.476672845843364 ]
  [ 0.476672845843364   1.229744902269525 ]]

 [[ 1.2214003949426913 -0.4841133862236404]
  [-0.4841133862236405  0.7911739159757286]]]

K=1: log-likelihood = -3725.4576335585434208, Bayesian information criterion =
     7485.4540435119979520

K=2: log-likelihood = -3200.4613040281610665, Bayesian information criterion =
     6476.9079161251256664

3.
For K = 2,

Weights:  [0.1946743731658157 0.8053256268341843]

Means:  [[ 0.0887631709875105 -0.0210976144553744]
 [ 4.0816341486534675  2.9476355599773143]]

Covariances:  [[[ 1.0761406229476318  0.476672845843364 ]
  [ 0.476672845843364   1.229744902269525 ]]

 [[ 1.2214003949426913 -0.4841133862236404]
  [-0.4841133862236405  0.7911739159757286]]]

5.
Classification accuracy: 0.998

Confusion matrix:
 [[184   1]
 [  1 814]]
```

Listing 2: Output terminal for *gmm.py*

III. CHAPTER 4: POLYNOMIAL REGRESSION WITH REGULARIZATION

1. Ridge implementation, diagnostics, and interpretation

1.

- For training and testing data, please refer to listing 3 and 4.
- For $\hat{\omega}_\lambda$, please refer to listing 3 and 4.
- Train and test MSE, $\kappa\left(A^TA\right)$, and $\kappa\left(A^TA+\lambda\Gamma^T\Gamma\right)$ are reported in listing 4.

  Below is a representative example of how ridge regression affect the condition number for $M=6$.

  Without regularization:

  $$\kappa\left(A^TA\right)=507514157.93791586$$

  With regularization for $\lambda=\left\{10^{-6},10^{-4},10^{-2},10^{0},10^{2}\right\}$:

  $$\kappa\left(A^TA+\lambda\Gamma^T\Gamma\right)=\{117405576.39922069,1522812.5394117208,15273.969715717929,$$
  $$153.2201656595726,3.9522336795648543\}$$

  We can see that with higher value of λ, condition number becomes smaller. So, the linear system becomes well-condition and numerically more stable with greater value of regularization term.

- Test MSE generally increases with higher value of λ as evident from figure 6. The increment is steeper for higher M. For $M=3,8$ and 9, MSE becomes smaller when

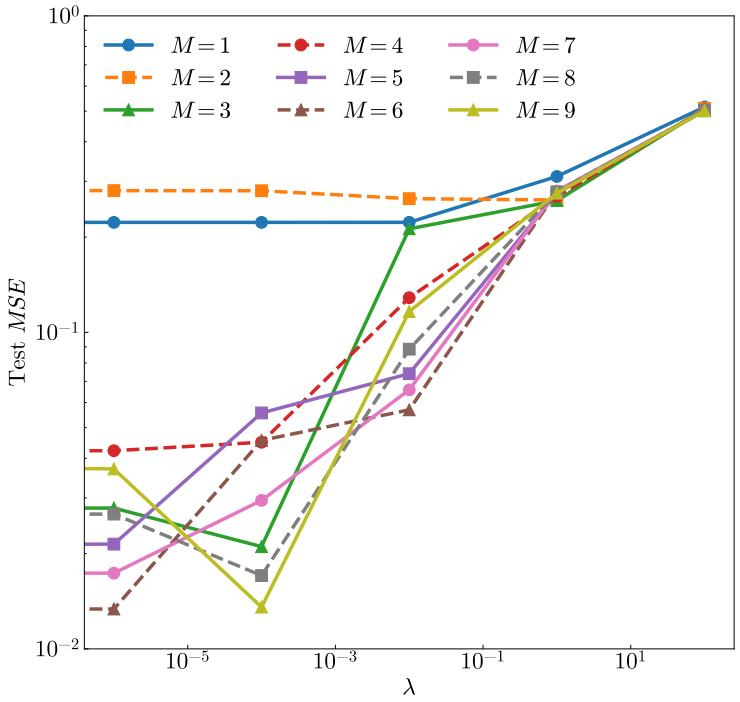


Figure 6: Test MSE vs. λ.

$\lambda=10^{-4}$ from $\lambda=10^{-6}$, but then it rises most steeply. High λ penalizes the co-efficients

to great extent inducing high bias and underfitting. This is particularly true for $M \geq 3$. As a result, MSE increases for these. For low order polynomial $M \leq 2$, regularization term has very little effect, till it becomes really large, e.g., $\lambda = 10^2$.

- Without regularization, condition number increases somewhat proportionately with M as can be seen from figure 7.

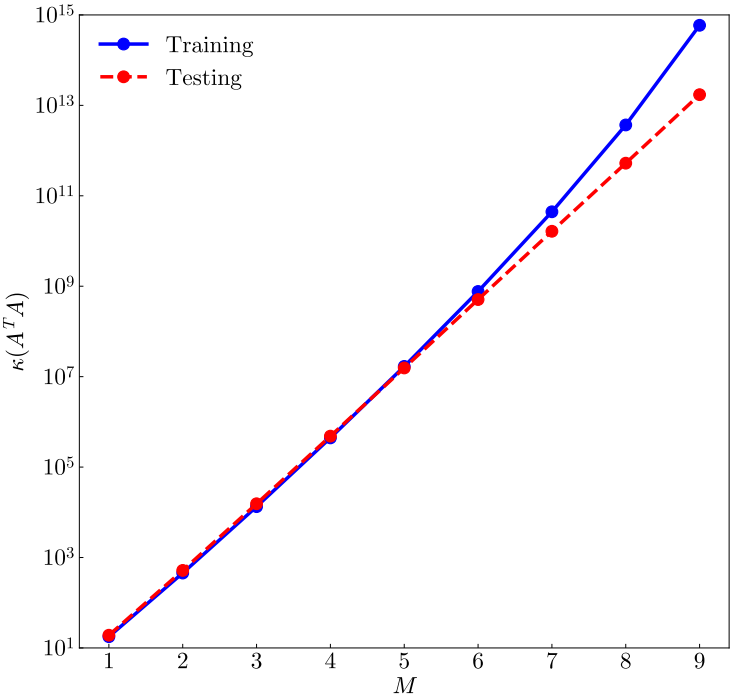
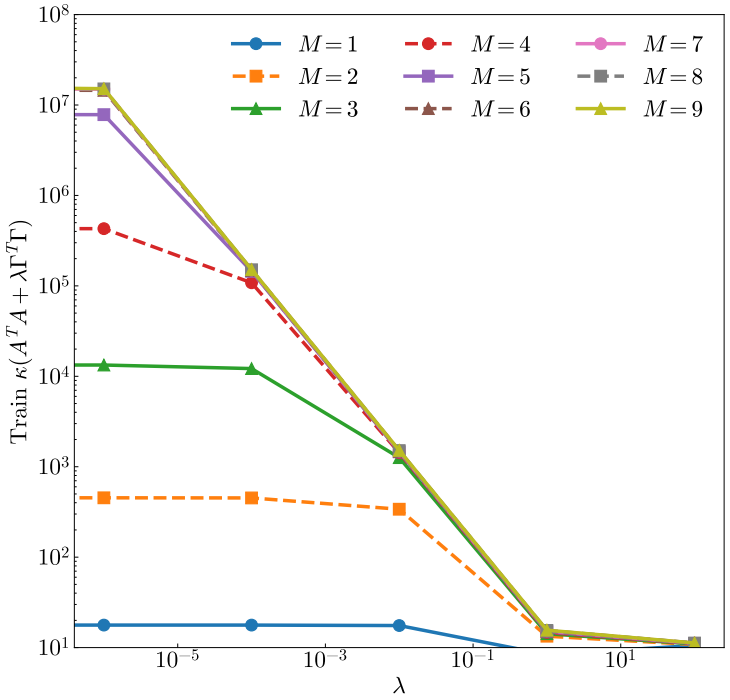
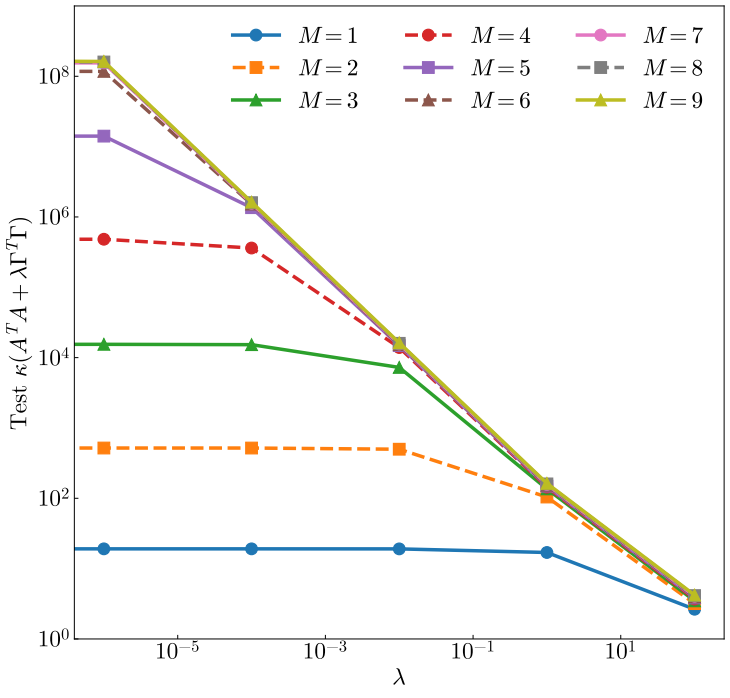


Figure 7: Unregularized condition number with respect to different M.

Now if we look into figure 8, both train and test regularized condition number decreases with higher value of λ. This is again echoing the same message as we got from the comparison of unregularized and regularized example from point-3 with $M = 6$. The linear system becomes well-conditioned with higher value of regularized term parameter.



(a)



(b)

Figure 8: Regularized condition number for (a) training and (b) testing.

2. The verification of closed form ridge regression with `scikit-learn` has been done in listing 3, and shown in 4 in the form of $L2$ norm of error. It can be observed that most of the error

are $< 10^{-10}$. Relatively larger error shows up when regularization is not done, and that is related to the different methods `scikit-learn` (Cholesky, SVD, or other) can use for solving linear system based on the size of the system and conditioning, and the operations might not be the same as `np.linalg.solve()`. If the operations are diffent, then the accumulated round-off error can come out to be different causing a large difference between the closed form and `scikit-learn` solutions.

3. With higher λ, the penalization term penalizes large co-efficients forcing them to be as small as possible. As a result, curve becomes smoother and less sensitive towards noise, i.e., the variance reduces. However, the model underfits, deviating the predicted value from true value, which leads to increased bias. If we see figure 6, we can understand the effect of underfitting and increased bias with higher λ as MSE increases drastically.

If λ is too small, it introduces high variance and overfitting for higher M. That is why as M increases, it is desirable to increase λ to ensure the ill-conditioning and overfitting is remedied. That is why optimal λ^* is high for higher M, but it must be ensured that it is not as high that it underfits the model. For low M, the system has less chance to be ill-conditioned and is already smooth, so they require very little regularization, lower λ. This can also be checked from figure 6, where we can see that the regularization has very little impact on MSE. This is why the optimal λ^* differs across M and one has to find a sweet spot that does not induce high bias or high variance based on M.

2. Model selection over polynomial degree and regularization

1. (M, λ) that minimizes test MSE along with the minimized test MSE:

$$\left(\hat{M}, \hat{\lambda}\right) = \left(6, 10^{-6}\right), \qquad MSE_{minimized} = 0.013362127882421209$$

2. The co-efficient vectors are given below.

Best fit $(\hat{M}, \hat{\lambda}) = (6, 10^{-6})$:

$$\hat{\omega} = \begin{pmatrix} -0.08958942 \\ 8.47143248 \\ -7.40357783 \\ -40.30544707 \\ 29.78715023 \\ 46.21868656 \\ -36.71097017 \end{pmatrix}$$

Poorly conditioned $(M, \lambda) = (7, 0)$:

$$\hat{\omega} = \begin{pmatrix} -0.0854431228 \\ 8.45284633 \\ -6.55162206 \\ -63.4142787 \\ 170.700841 \\ -294.806939 \\ 323.931451 \\ -139.156624 \end{pmatrix}$$

Overly penalized $(M, \lambda) = (7, 10^0)$:

$$\hat{\omega} = \begin{pmatrix} 0.45998642 \\ -0.42911245 \\ -0.45704387 \\ -0.33176584 \\ -0.20989608 \\ -0.11739408 \\ -0.05236374 \\ -0.00827713 \end{pmatrix}$$

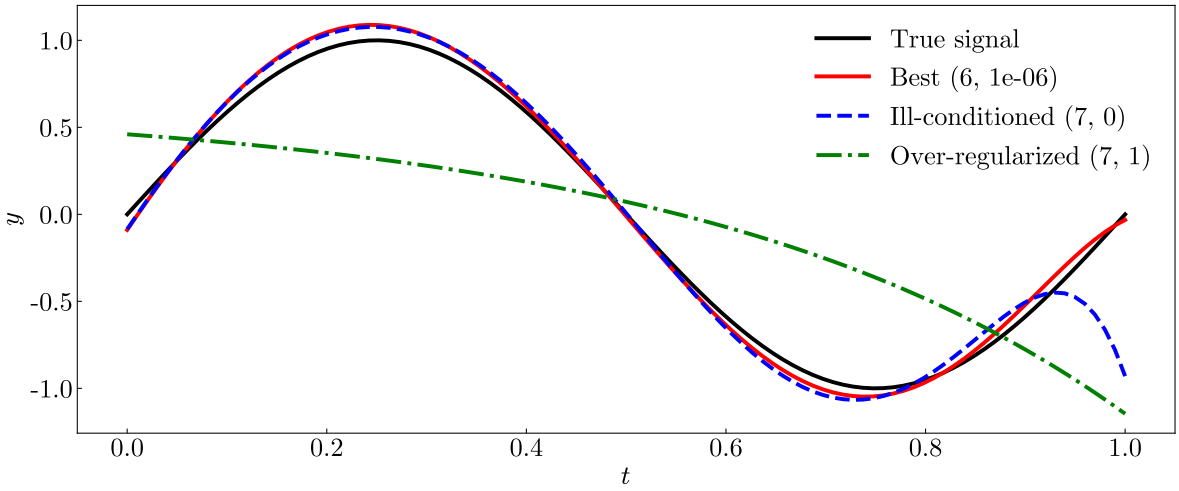


Figure 9: Comparison of the three different fit with true noise-free signal.

We can clearly see from the co-efficient vectors and figure 9 that overly penalized case enforces the co-efficient for higher degree term to be very small, which translates to over-smoothing, increased bias and classic underfitting. This is the worst fit of the three, which suggests that penalizing with high regularization parameter ($\lambda >>$) can be a disaster.

For poorly conditioned system, the co-efficients for higher degree terms are very big because of no penalization ($\lambda = 0$), and that translates to classic overfitting of the true data. The oscillation near the end of data completely throws the model off the balance despite it doing well for previous data. This is an example of high variance in the model which has the propensity to induce such oscillations towards the either ends of the curve.

The best fit shows a balance of bias and variance, and retains its pattern throughout the data faithfully. The predicted value has some error towards the peak of the true signal, but that is a trade-off that we have to accept in order to faithfully represent the true signal's characteristics. This is achieved with a balanced choice of the regularized parameter ($\lambda = 10^{-6}$). This suggests that the parameter needs to be chosen in such a way that it ensures a good balance between bias and variance.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from sklearn.linear_model import Ridge

def generate_data(scale_noise, N, seed=56):
    """ Generate a noisy time series dataset {t_i, y_i}_{i=1}^N.
    Parameters
    ----------
    A : float
    Amplitude of the signal (A >= 0)
    N : int
    Number of data points. Time points uniformly sampled from [0, 2pi]
    seed : int or None, optional
    Random seed for reproducibility """

    rng = np.random.default_rng(seed)
    t = np.linspace(0.0, 1.0, N)
    dt = (2*np.pi) / (N - 1)
    A=np.sqrt(np.pi/dt)
    s_hat = np.sqrt(dt / np.pi) * np.sin(2*np.pi*t)
    # ||s_hat||_2 == 1 exactly on this grid
    s = A * s_hat
    n = rng.normal(loc=0.0, scale=scale_noise, size=N)
    y = s + n
    return t, y, s, n, s_hat, dt

# ===== 1. Ridge implementation, diagnostics, and interpretation =====

# 1. Implementation and comparison
# generate testing and training data
x_data,y_data,s,n,s_hat, dt = generate_data(.1,100)
t_train = x_data[::10]
y_train = y_data[::10]
s_train = s[::10]
plt.plot(t_train,y_train,'bo')
plt.plot(t_train,s_train,'r--')
plt.plot(x_data,s,'k--')
plt.show()
print("1. Implementation and comparison")
print("\nTraining data (t, y):\n", np.column_stack((t_train, y_train)))

# generate testing data
# TODO for YOU: Write code to remove the training data below
# t_test and y_test will have 90 data points

t_test  = np.delete(x_data, np.arange(0, len(x_data), 10))
y_test  = np.delete(y_data, np.arange(0, len(y_data), 10))
print("\nTesting data (t, y):\n", np.column_stack((t_test, y_test)))

# function to construct Vandermonde matrix
def vandermonde(t, M):
    return np.vstack([t**i for i in range(M+1)]).T

# function to construct $\gamma$ matrix
def gamma(M):
    g = np.ones(M+1)
```

```
    g[0] = 0.0
    return np.diag(g)

M = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9]) # polynomial orders
lambda_ = np.array([0, 1e-6, 1e-4, 1e-2, 1e0, 1e2]) # regularization factors

omega_train_list = {} # omega for training data
mse_train_list = {} # training MSE
mse_test_list = {} # testing MSE
cond_no_train_list = {} # training condition no. w/o regularization factor
cond_no_ridge_train_list = {} # training condition no. w/ regularization
    factor
cond_no_test_list = {} # testing condition no. w/o regularization factor
cond_no_ridge_test_list = {} # testing condition no. w/ regularization factor

for i in M:
    A_train = vandermonde(t_train, i)
    A_test = vandermonde(t_test, i)
    for j in lambda_:
        omega_train = np.linalg.solve(((A_train.T @ A_train) +
                                       j * (gamma(i).T @ gamma(i))), (A_train.T @
                                           y_train))
        omega_train_list[(i, j)] = omega_train
        y_train_predict = A_train @ omega_train
        y_test_predict = A_test @ omega_train
        mse_train = np.mean((abs(y_train_predict - y_train))**2)
        mse_train_list[(i, j)] = mse_train # update traing MSE list
        mse_test = np.mean((abs(y_test_predict - y_test))**2)
        mse_test_list[(i, j)] = mse_test # update testing MSE list
        # compute and update condition no. lists
        cond_no_train = np.linalg.cond(A_train.T @ A_train, 2)
        cond_no_train_list[(i, j)] = cond_no_train
        cond_no_ridge_train = np.linalg.cond((A_train.T @ A_train + j * gamma(
            i).T @ gamma(i)), 2)
        cond_no_ridge_train_list[(i, j)] = cond_no_ridge_train
        cond_no_test = np.linalg.cond(A_test.T @ A_test, 2)
        cond_no_test_list[(i, j)] = cond_no_test
        cond_no_ridge_test = np.linalg.cond((A_test.T @ A_test + j * gamma(i).
            T @ gamma(i)), 2)
        cond_no_ridge_test_list[(i, j)] = cond_no_ridge_test

print("\n$omega$:", omega_train_list)
print("\nMSE (training):", mse_train_list)
print("\nMSE (testing):", mse_test_list)
print("\nCondition no. w/o regularization (training):", cond_no_train_list)
print("\nCondition no. w/ regularization (training):",
    cond_no_ridge_train_list)
print("\nCondition no. w/o regularization (testing):", cond_no_test_list)
print("\nCondition no. w/ regularization (testing):", cond_no_ridge_test_list)

df = pd.DataFrame([[M_val, lam, mse_train_list[(M_val, lam)],
                    mse_test_list[(M_val, lam)],
                    cond_no_train_list[(M_val, lam)],
                    cond_no_ridge_train_list[(M_val, lam)],
                    cond_no_test_list[(M_val, lam)], cond_no_ridge_test_list[(
                        M_val, lam)]]
    for M_val in M for lam in lambda_],
columns=["M", "$\lambda$", "MSE_train", "MSE_test", "$kappa(A^TA)_train$",
         "$kappa(A^TA+\lambda\gamma^T\gamma)_train$", "$kappa(A^TA)_test$",
```

```
         "$kappa(A^TA+\lambda\gamma^T\gamma)_test$"])
print("\nTable for all combinations:\n", df)

# parameters for plotting
plt.rcParams["font.family"] = "serif"
plt.rcParams["font.serif"] = ["CMU Serif"]
plt.rcParams["mathtext.fontset"] = "cm"
plt.rcParams["font.size"] = 20
mpl.rcParams["axes.unicode_minus"] = False

# mse plots
fig, ax = plt.subplots(figsize=(10, 10))
markers = ["o", "s", "^"]
linestyles = ["-", "--"]

for i, M_val in enumerate(M):
    subset = df[df["M"] == M_val]
    ax.loglog(subset["$\\lambda$"], subset["MSE_test"],
              marker=markers[i % len(markers)], markersize=10,
              linestyle=linestyles[i % len(linestyles)], linewidth=3,
              label=fr"$M={M_val}$")

plt.xlabel(r"$\lambda$")
plt.ylabel(r"Test $MSE$")
plt.ylim(1e-2, 1e0)
plt.legend(frameon=False, loc="best", ncol=3)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("ridge_mse.pdf", dpi=1080)
plt.show()

# condition no. plot w/o regularization
fig, ax = plt.subplots(figsize=(10, 10))

subset = df[df["$\\lambda$"] == 0].sort_values("M")

ax.semilogy(subset["M"], subset["$kappa(A^TA)_train$"], "-o", markersize=10,
    linewidth=3, color="blue", label="Training")
ax.semilogy(subset["M"], subset["$kappa(A^TA)_test$"], "--o", markersize=10,
    linewidth=3, color="red", label="Testing")

plt.xlabel(r"$M$")
plt.ylabel(r"$\kappa(A^TA)$")
plt.ylim(1e1, 1e15)
plt.legend(frameon=False, loc="upper left")
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("cond.pdf", dpi=1080)
plt.show()

# condition no. plot w/ regularizations
fig, ax = plt.subplots(figsize=(10, 10))
markers = ["o", "s", "^"]
linestyles = ["-", "--"]

for i, M_val in enumerate(M):
    subset = df[df["M"] == M_val]
    ax.loglog(subset["$\\lambda$"], subset["$kappa(A^TA+\\lambda\\gamma^T\\
        gamma)_train$"],
              marker=markers[i % len(markers)], markersize=10,
              linestyle=linestyles[i % len(linestyles)], linewidth=3,
```

```
                label=fr"$M={M_val}$")

plt.xlabel(r"$\lambda$")
plt.ylabel(r"Train $\kappa(A^TA+\lambda\Gamma^T\Gamma)$")
plt.ylim(1e1, 1e8)
plt.legend(frameon=False, loc="upper right", ncol=3)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("train_ridge_cond.pdf", dpi=1080)
plt.show()

fig, ax = plt.subplots(figsize=(10, 10))
markers = ["o", "s", "^"]
linestyles = ["-", "--"]

for i, M_val in enumerate(M):
    subset = df[df["M"] == M_val]
    ax.loglog(subset["$\\lambda$"], subset["$kappa(A^TA+\\lambda\\gamma^T\\
        gamma)_test$"],
                marker=markers[i % len(markers)], markersize=10,
                linestyle=linestyles[i % len(linestyles)], linewidth=3,
                label=fr"$M={M_val}$")
plt.xlabel(r"$\lambda$")
plt.ylabel(r"Test $\kappa(A^TA+\lambda\Gamma^T\Gamma)$")
plt.ylim(1e0, 1e9)
plt.legend(frameon=False, loc='upper right', ncol=3)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("test_ridge_cond.pdf", dpi=1080)
plt.show()

# 2. Equivalence check with scikit-learn
diff = []

print("\n2. Equivalence check with scikit-learn")
for M_val in M:
    A_train = vandermonde(t_train, M_val)
    I = np.eye(M_val + 1)
    for lam in lambda_:
        # closed form solution
        omega_closed = np.linalg.solve((A_train.T @ A_train) +
                                       lam * (I.T @ I), A_train.T @ y_train)

        # scikit-learn solution
        ridge = Ridge(alpha=lam, fit_intercept=False, solver='auto')
        ridge.fit(A_train, y_train)
        omega_sklearn = ridge.coef_

        # compute l2 norm
        diff_ = np.linalg.norm(omega_closed - omega_sklearn)
        diff.append([M_val, lam, diff_])

        print(f"\nM={M_val:2d}, lambda={lam:8.1e},  error={diff_:.2e}")

# dataframe
df_diff = pd.DataFrame(diff, columns=["M", "$lambda$", "l2"])
print("\nCoefficient L2-norm differences across all (M, $\lambda$):\n")
print(df_diff)

# ===== 2. Model selection over polynomial degree and regularization =====

```

```python
# 1. Grid search over (M, $\lambda$)
M_hat, lambda_hat = min(mse_test_list, key=mse_test_list.get)
min_mse_test = mse_test_list[M_hat, lambda_hat]

print(f"\n1. Grid search over (M, $\lambda$)")
print(f"\nBest (M_hat, Lambda) = ({M_hat}, {lambda_hat})")
print(f"\nMinimum Test MSE = {min_mse_test}")

# 2. Final reflections
cases = [
    ("Best", M_hat, lambda_hat),
    ("Ill-conditioned", 7, 0),
    ("Over-regularized", 7, 1e0)
]

coeffs_list = {}

print(f"\n2. Final reflections")

# pick the ($M, \lambda$) for comparison
for label, M_val, lam in cases:
    coeffs_list[label] = omega_train_list[(M_val, lam)]
    print(f"\n{label} coefficients (M={M_val}, $\lambda$={lam:g}):")
    print(coeffs_list[label])

# noise-free true signal
_, _, s_true, _, _, _ = generate_data(0.0, 100, seed=56)

# plot for comparison
fig, ax = plt.subplots(figsize=(15, 6))

ax.plot(x_data, s_true, "k-", linewidth=3, label="True signal")

colors = ["red", "blue", "green"]
linestyles = ["-", "--", "-."]

for (label, M_val, lam), color, ls in zip(cases, colors, linestyles):
    A_plot = vandermonde(x_data, M_val)
    y_fit = A_plot @ coeffs_list[label]
    ax.plot(x_data, y_fit, color=color, linestyle=ls, linewidth=3,
            label=rf"{label} ({M_val}, {lam:g})")

plt.xlabel(r"$t$")
plt.ylabel("$y$")
plt.legend(frameon=False, loc="upper right")
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("signal_fit.pdf", dpi=1080)
plt.show()
```

Listing 3: *ridge.py*

```
1. Implementation and comparison

Training data (t, y):
 [[ 0.         -0.08499407]
 [ 0.1010101   0.64733005]
 [ 0.2020202   1.05181989]
 [ 0.3030303   0.98179583]
 [ 0.4040404   0.66274571]
 [ 0.50505051 -0.07132869]
 [ 0.60606061 -0.66085845]
 [ 0.70707071 -1.06338572]
 [ 0.80808081 -0.9015869 ]
 [ 0.90909091 -0.47937759]]

Testing data (t, y):
 [[ 0.01010101  0.095618  ]
 [ 0.02020202  0.30549674]
 [ 0.03030303  0.27718516]
 [ 0.04040404  0.28830727]
 [ 0.05050505  0.46080282]
 [ 0.06060606  0.44452901]
 [ 0.07070707  0.63791978]
 [ 0.08080808  0.35246993]
 [ 0.09090909  0.55317597]
 [ 0.11111111  0.57540486]
 [ 0.12121212  0.69204348]
 [ 0.13131313  0.6287832 ]
 [ 0.14141414  0.64133984]
 [ 0.15151515  0.93655855]
 [ 0.16161616  0.60970634]
 [ 0.17171717  0.72579539]
 [ 0.18181818  0.86814952]
 [ 0.19191919  0.83364087]
 [ 0.21212121  1.03648661]
 [ 0.22222222  1.01153419]
 [ 0.23232323  1.14854167]
 [ 0.24242424  1.11652099]
 [ 0.25252525  1.11487025]
 [ 0.26262626  1.03486301]
 [ 0.27272727  0.99333582]
 [ 0.28282828  1.06649211]
 [ 0.29292929  1.06171274]
 [ 0.31313131  0.93978624]
 [ 0.32323232  0.94949208]
 [ 0.33333333  0.95672969]
 [ 0.34343434  0.93146967]
 [ 0.35353535  0.71523196]
 [ 0.36363636  0.72778199]
 [ 0.37373737  0.78754805]
 [ 0.38383838  0.67999822]
 [ 0.39393939  0.61619001]
 [ 0.41414141  0.50515954]
 [ 0.42424242  0.52153018]
 [ 0.43434343  0.28565485]
 [ 0.44444444  0.51476062]
 [ 0.45454545  0.22247404]
 [ 0.46464646  0.18517499]
 [ 0.47474747  0.40135512]
```

```
 [ 0.48484848  0.23889984]
 [ 0.49494949  0.08067348]
 [ 0.51515152 -0.14084141]
 [ 0.52525253 -0.31686598]
 [ 0.53535354 -0.29180951]
 [ 0.54545455 -0.55303597]
 [ 0.55555556 -0.30226842]
 [ 0.56565657 -0.38534422]
 [ 0.57575758 -0.56476957]
 [ 0.58585859 -0.50590258]
 [ 0.5959596  -0.73001717]
 [ 0.61616162 -0.62649814]
 [ 0.62626263 -0.6127254 ]
 [ 0.63636364 -0.7203109 ]
 [ 0.64646465 -0.87832832]
 [ 0.65656566 -0.88175041]
 [ 0.66666667 -0.69083886]
 [ 0.67676768 -1.09075381]
 [ 0.68686869 -0.80149587]
 [ 0.6969697  -1.15451343]
 [ 0.71717172 -0.90304549]
 [ 0.72727273 -1.02024582]
 [ 0.73737374 -1.08341421]
 [ 0.74747475 -0.91948624]
 [ 0.75757576 -0.90510595]
 [ 0.76767677 -1.06077573]
 [ 0.77777778 -0.9593354 ]
 [ 0.78787879 -0.77922223]
 [ 0.7979798  -0.95741281]
 [ 0.81818182 -0.88424218]
 [ 0.82828283 -0.85004402]
 [ 0.83838384 -0.82680995]
 [ 0.84848485 -0.85390869]
 [ 0.85858586 -0.73882638]
 [ 0.86868687 -0.83223967]
 [ 0.87878788 -0.71675667]
 [ 0.88888889 -0.70125887]
 [ 0.8989899  -0.60138096]
 [ 0.91919192 -0.54189279]
 [ 0.92929293 -0.45149955]
 [ 0.93939394 -0.30593343]
 [ 0.94949495 -0.38558525]
 [ 0.95959596 -0.20319345]
 [ 0.96969697 -0.279404  ]
 [ 0.97979798 -0.01414699]
 [ 0.98989899 -0.03039937]
 [ 1.         -0.00182266]]

$omega$: {(1, 0.0): array([ 0.84357789, -1.83779614]), (1, 1e-06): array([
    0.84357689, -1.83779395]), (1, 0.0001): array([ 0.84347866, -1.83757783]),
     (1, 0.01): array([ 0.8337703 , -1.81621945]), (1, 1.0): array([
    0.39000846, -0.83994339]), (1, 100.0): array([ 0.01518898, -0.01534054]),
    (2, 0.0): array([ 0.51577525,  0.59613845, -2.67732804]), (2, 1e-06):
    array([ 0.51578274,  0.5960845 , -2.67726948]), (2, 0.0001): array([
    0.51652185,  0.59076159, -2.6714907 ]), (2, 0.01): array([ 0.57061433,
    0.19444453, -2.23800574]), (2, 1.0): array([ 0.45480323, -0.57581975,
    -0.63569277]), (2, 100.0): array([ 0.01958626, -0.01522441, -0.01530357]),
     (3, 0.0): array([ -0.164077  ,  12.90874823, -38.37321301,  26.1769823 ])
    , (3, 1e-06): array([ -0.16328965,  12.89579109, -38.33718259,
```

```
    26.15122789]), (3, 0.0001): array([ -0.09215868,  11.72520575,
    -35.08211406,  23.82453613]), (3, 0.01): array([ 0.58025357,  0.64435101,
    -4.29019856,  1.83330497]), (3, 1.0): array([ 0.46317807, -0.48035173,
    -0.52482635, -0.40253213]), (3, 100.0): array([ 0.02219063, -0.01514302,
    -0.01522006, -0.01277272]), (4, 0.0): array([ -0.1185112 ,  11.0291592 ,
    -27.93209599,   7.75513031,
        10.1320186 ]), (4, 1e-06): array([ -0.11717306,  10.98625765,
            -27.71580306,   7.39630302,
        10.32093145]), (4, 0.0001): array([ -0.06509959,   9.68935503,
            -21.99274875,  -1.19588134,
        14.49595856]), (4, 0.01): array([ 0.54144274,  1.43945469,
            -5.29936503, -1.16069536,  3.73196998]), (4, 1.0): array([
            0.46230324, -0.44542829, -0.48024667, -0.35691717, -0.23482327]),
            (4, 100.0): array([ 0.02377879, -0.01508824, -0.01516214,
            -0.0127189 , -0.01028149]), (5, 0.0): array([-7.79703349e-02,
            6.96655882e+00,  9.31862724e+00, -1.08069785e+02,
        1.56169754e+02, -6.42566036e+01]), (5, 1e-06): array([ -0.10300144,
            9.23795398, -10.92052739, -46.33522981,
        79.43328801, -30.86400209]), (5, 0.0001): array([ -0.05449944,
            9.36004116, -20.93710874,  -0.32493449,
         9.47892068,   3.55110614]), (5, 0.01): array([ 0.49476977,  1.9688623
              , -5.32955816, -2.56648054,  1.18645405,
        4.0496286 ]), (5, 1.0): array([ 0.46084429, -0.43315373, -0.46317842,
            -0.33863434, -0.2168465 ,
       -0.12411209]), (5, 100.0): array([ 0.02478529, -0.01505119,
           -0.01512213, -0.01268113, -0.01024714,
       -0.00825127]), (6, 0.0): array([-8.62324250e-02,  9.39177142e+00,
           -2.35939207e+01,  4.77547764e+01,
       -1.75070174e+02,  2.59851470e+02, -1.18839627e+02]), (6, 1e-06): array
           ([ -0.08958942,   8.47143248,  -7.40357783, -40.30544707,
        29.78715023,  46.21868656, -36.71097017]), (6, 0.0001): array([
            -0.06089955,   9.4621997 , -20.82055338,  -1.82092653,
         9.89463292,   7.69965897,  -3.41750461]), (6, 0.01): array([
             0.46066378,  2.22151529, -5.04457925, -3.06247596, -0.08613074,
        2.19370018,  3.64540497]), (6, 1.0): array([ 0.4601021 , -0.42951707,
            -0.45770141, -0.33252608, -0.21068122,
       -0.11816405, -0.05309823]), (6, 100.0): array([ 0.02544632,
           -0.01502572, -0.01509416, -0.01265441, -0.01022262,
       -0.00822909, -0.00666902]), (7, 0.0): array([-8.54431228e-02,
           8.45284633e+00, -6.55162206e+00, -6.34142787e+01,
        1.70700841e+02, -2.94806939e+02,  3.23931451e+02, -1.39156624e+02]),
            (7, 1e-06): array([ -0.08606502,   8.57170586, -10.37918302,
            -24.81850355,
         7.88300721,  30.69524957,  17.51710662, -29.79242729]), (7, 0.0001):
             array([ -0.06663618,   9.43831155, -19.89409092,  -3.94292132,
         8.53854839,   9.92002457,   3.43048862,  -6.77386304]), (7, 0.01):
             array([ 0.43998056,  2.31382545, -4.74637107, -3.16565676,
             -0.66748565,
        1.21707478,  2.38599059,  3.01407311]), (7, 1.0): array([ 0.45998642,
            -0.42911245, -0.45704387, -0.33176584, -0.20989608,
       -0.11739408, -0.05236374, -0.00827713]), (7, 100.0): array([
           0.02589412, -0.01500786, -0.01507431, -0.01263527, -0.01020492,
       -0.00821297, -0.00665446, -0.00544518]), (8, 0.0): array([-8.53460225e
           -02,  7.68844533e+00,  1.04051404e+01, -2.04101761e+02,
        7.54510095e+02, -1.63178503e+03,  2.03971983e+03, -1.29468748e+03,
        3.17770964e+02]), (8, 1e-06): array([ -0.08615891,   8.7863877 ,
            -12.99165873, -16.38121648,
         3.55068973,  18.04569895,  17.23284674,   2.41985936,
       -21.29447286]), (8, 0.0001): array([ -0.06832276,   9.30339119,
```

```
           -18.74152461,  -5.37864909,
133         6.65863436,   9.938852  ,   6.60400447,  -0.13089784,
134        -7.90884346]), (8, 0.01): array([ 0.42859457,  2.33224272,
           -4.51855987, -3.13044096, -0.91207784,
135        0.72057211,  1.69669712,  2.19173531,  2.37688458]), (8, 1.0): array([
            0.46024244, -0.42980936, -0.45826299, -0.33322199, -0.21143069,
136       -0.11892075, -0.05383594, -0.00967079,  0.0195075 ]), (8, 100.0): array
           ([ 0.02620557, -0.01499511, -0.01506   , -0.01262136, -0.01019196,
137       -0.00820112, -0.0066437 , -0.00543545, -0.00449369]), (9, 0.0): array
           ([-8.49935269e-02, -3.27648701e+01,  1.03824219e+03, -1.03879742e
           +04,
138        5.33810068e+04, -1.59342687e+05,  2.86104933e+05, -3.04371861e+05,
139        1.76702113e+05, -4.31160604e+04]), (9, 1e-06): array([ -0.08689569,
            8.92113169, -14.27388064, -13.41613075,
140         4.29170421,  13.35759818,  12.35062437,   5.34641257,
141        -3.98552164, -13.45642539]), (9, 0.0001): array([ -0.06727238,
           9.14152576, -17.77551295,  -6.05787573,
142         5.10066393,   9.081699  ,   7.54903078,   3.08772435,
143        -2.40401567,  -7.79971759]), (9, 0.01): array([ 0.42272997,  2.322687
            , -4.36441073, -3.06238671, -1.0029335 ,
144        0.47461423,  1.32506153,  1.72841883,  1.85269188,  1.816474  ]), (9,
           1.0): array([ 0.46067677, -0.43077367, -0.46007601, -0.33545251,
           -0.21382411,
145       -0.12133206, -0.05618346, -0.01190976,  0.01739998,  0.03623266]), (9,
           100.0): array([ 0.02642711, -0.01498585, -0.01504952, -0.01261111,
           -0.01018237,
146       -0.00819231, -0.00663567, -0.00542816, -0.00448709, -0.00374667])}
147
148 MSE (training): {(1, 0.0): 0.27073288909194354, (1, 1e-06):
    0.27073288909234483, (1, 0.0001): 0.27073289310345405, (1, 0.01):
    0.2707720770906607, (1, 1.0): 0.3545468710021419, (1, 100.0):
    0.5503073352430806, (2, 0.0): 0.23133288031654056, (2, 1e-06):
    0.23133288033543628, (2, 0.0001): 0.23133306802503034, (2, 0.01):
    0.23239419334679398, (2, 1.0): 0.2936344906347173, (2, 100.0):
    0.5456413991349919, (3, 0.0): 0.006521765184228492, (3, 1e-06):
    0.00652198737491233, (3, 0.0001): 0.00837558444932197, (3, 0.01):
    0.20527662605749836, (3, 1.0): 0.27984623261605035, (3, 100.0):
    0.5424190489982516, (4, 0.0): 0.004689033380580246, (4, 1e-06):
    0.004689750410450222, (4, 0.0001): 0.0053854390750167885, (4, 0.01):
    0.15196831328445443, (4, 1.0): 0.2789739042905329, (4, 100.0):
    0.5403477737449263, (5, 0.0): 0.001127982418387338, (5, 1e-06):
    0.0021024362826733345, (5, 0.0001): 0.006416159311522294, (5, 0.01):
    0.11093727362946038, (5, 1.0): 0.2804542007363015, (5, 100.0):
    0.5390234911746371, (6, 0.0): 0.0006273934530143964, (6, 1e-06):
    0.0009639024324000196, (6, 0.0001): 0.005353219860754832, (6, 0.01):
    0.08829109466639389, (6, 1.0): 0.2814963019244499, (6, 100.0):
    0.5381641994522264, (7, 0.0): 0.0006049563056163806, (7, 1e-06):
    0.0006884547510555836, (7, 0.0001): 0.0037819831459206853, (7, 0.01):
    0.07812673336176454, (7, 1.0): 0.28168632608569505, (7, 100.0):
    0.5375948849228811, (8, 0.0): 0.0006022598739992099, (8, 1e-06):
    0.0006172325489378455, (8, 0.0001): 0.0025103931650422677, (8, 0.01):
    0.0745949645894507, (8, 1.0): 0.28122052407527365, (8, 100.0):
    0.5372093532130965, (9, 0.0): 4.818198085563335e-10, (9, 1e-06):
    0.0006085594474656156, (9, 0.0001): 0.0016956592073576534, (9, 0.01):
    0.07409067937027043, (9, 1.0): 0.28036961174950115, (9, 100.0):
    0.5369427468444984}
149
150 MSE (testing): {(1, 0.0): 0.22261110362410202, (1, 1e-06):
    0.22261110635622774, (1, 0.0001): 0.22261138092837973, (1, 0.01):
```

```
    0.22267879221522333, (1, 1.0): 0.3108906853064988, (1, 100.0):
    0.5151969175036057, (2, 0.0): 0.28042234737895, (2, 1e-06):
    0.2804200797763094, (2, 0.0001): 0.2801965329381177, (2, 0.01):
    0.26453268557958626, (2, 1.0): 0.2619305402127051, (2, 100.0):
    0.510378961209461, (3, 0.0): 0.027955457730456247, (3, 1e-06):
    0.027850996146955728, (3, 0.0001): 0.021040029311505455, (3, 0.01):
    0.212400846114817, (3, 1.0): 0.2607610057042226, (3, 100.0):
    0.5070333388086519, (4, 0.0): 0.04200443273608897, (4, 1e-06):
    0.04227354957577295, (4, 0.0001): 0.04496655673075375, (4, 0.01):
    0.12869505641147927, (4, 1.0): 0.2687708907125886, (4, 100.0):
    0.5048638765266599, (5, 0.0): 0.013598955728502369, (5, 1e-06):
    0.021424921356019807, (5, 0.0001): 0.055649802455593436, (5, 0.01):
    0.07404361848552804, (5, 1.0): 0.27543325642130384, (5, 100.0):
    0.5034631774899775, (6, 0.0): 0.02226625122424576, (6, 1e-06):
    0.013362127882421209, (6, 0.0001): 0.045585420826319646, (6, 0.01):
    0.05691540624294651, (6, 1.0): 0.2788259463366386, (6, 100.0):
    0.5025451230060959, (7, 0.0): 0.036230362978373815, (7, 1e-06):
    0.017346162184612397, (7, 0.0001): 0.029457080712510582, (7, 0.01):
    0.06581897323774094, (7, 1.0): 0.27939573697046327, (7, 100.0):
    0.5019307939090133, (8, 0.0): 0.022477650238683654, (8, 1e-06):
    0.02666117445689405, (8, 0.0001): 0.017043494538511818, (8, 0.01):
    0.0884974052350431, (8, 1.0): 0.2780155166666279, (8, 100.0):
    0.5015107445771732, (9, 0.0): 14.817636096390192, (9, 1e-06):
    0.03703316605967967, (9, 0.0001): 0.013548928088499857, (9, 0.01):
    0.11638720751257137, (9, 1.0): 0.27544618856728054, (9, 100.0):
    0.5012175707086594}
151
152 Condition no. w/o regularization (training): {(1, 0.0): 17.737248092888933,
    (1, 1e-06): 17.737248092888933, (1, 0.0001): 17.737248092888933, (1, 0.01)
    : 17.737248092888933, (1, 1.0): 17.737248092888933, (1, 100.0):
    17.737248092888933, (2, 0.0): 453.31428795133763, (2, 1e-06):
    453.31428795133763, (2, 0.0001): 453.31428795133763, (2, 0.01):
    453.31428795133763, (2, 1.0): 453.31428795133763, (2, 100.0):
    453.31428795133763, (3, 0.0): 13339.252757834696, (3, 1e-06):
    13339.252757834696, (3, 0.0001): 13339.252757834696, (3, 0.01):
    13339.252757834696, (3, 1.0): 13339.252757834696, (3, 100.0):
    13339.252757834696, (4, 0.0): 442460.9052181754, (4, 1e-06):
    442460.9052181754, (4, 0.0001): 442460.9052181754, (4, 0.01):
    442460.9052181754, (4, 1.0): 442460.9052181754, (4, 100.0):
    442460.9052181754, (5, 0.0): 16844073.40840483, (5, 1e-06):
    16844073.40840483, (5, 0.0001): 16844073.40840483, (5, 0.01):
    16844073.40840483, (5, 1.0): 16844073.40840483, (5, 100.0):
    16844073.40840483, (6, 0.0): 765246943.950359, (6, 1e-06):
    765246943.950359, (6, 0.0001): 765246943.950359, (6, 0.01):
    765246943.950359, (6, 1.0): 765246943.950359, (6, 100.0):
    765246943.950359, (7, 0.0): 44306043078.680084, (7, 1e-06):
    44306043078.680084, (7, 0.0001): 44306043078.680084, (7, 0.01):
    44306043078.680084, (7, 1.0): 44306043078.680084, (7, 100.0):
    44306043078.680084, (8, 0.0): 3686302260714.4976, (8, 1e-06):
    3686302260714.4976, (8, 0.0001): 3686302260714.4976, (8, 0.01):
    3686302260714.4976, (8, 1.0): 3686302260714.4976, (8, 100.0):
    3686302260714.4976, (9, 0.0): 590145630456648.8, (9, 1e-06):
    590145630456648.8, (9, 0.0001): 590145630456648.8, (9, 0.01):
    590145630456648.8, (9, 1.0): 590145630456648.8, (9, 100.0):
    590145630456648.8}
153
154 Condition no. w/ regularization (training): {(1, 0.0): 17.737248092888933, (1,
     1e-06): 17.737227579792254, (1, 0.0001): 17.735197024882076, (1, 0.01):
    17.534529534421594, (1, 1.0): 8.383151748716001, (1, 100.0):
```

```
    10.546960037426274, (2, 0.0): 453.31428795133763, (2, 1e-06):
    453.298987459858, (2, 0.0001): 451.7893379320636, (2, 0.01):
    338.93725077096707, (2, 1.0): 13.368857651021033, (2, 100.0):
    10.806747474461753, (3, 0.0): 13339.252757834696, (3, 1e-06):
    13326.481598240725, (3, 0.0001): 12172.70886340197, (3, 0.01):
    1260.66627613596, (3, 1.0): 14.241365669312243, (3, 100.0):
    10.957353723416785, (4, 0.0): 442460.9052181754, (4, 1e-06):
    429203.0997441497, (4, 0.0001): 108209.51621759933, (4, 0.01):
    1428.1239060044268, (4, 1.0): 14.684340726195504, (4, 100.0):
    11.05434066861605, (5, 0.0): 16844073.40840483, (5, 1e-06):
    7822923.186889193, (5, 0.0001): 144812.3062395995, (5, 0.01):
    1460.9120827517897, (5, 1.0): 14.986651913795011, (5, 100.0):
    11.120899252384472, (6, 0.0): 765246943.950359, (6, 1e-06):
    14528872.529721871, (6, 0.0001): 148072.25610161768, (6, 0.01):
    1481.377810557409, (6, 1.0): 15.203834525460577, (6, 100.0):
    11.16853439456841, (7, 0.0): 44306043078.680084, (7, 1e-06):
    14955163.972538766, (7, 0.0001): 149602.0130738537, (7, 0.01):
    1496.4067593622703, (7, 1.0): 15.364271864925177, (7, 100.0):
    11.203642912481108, (8, 0.0): 3686302260714.4976, (8, 1e-06):
    15073306.004044555, (8, 0.0001): 150734.0592951826, (8, 0.01):
    1507.7299069289086, (8, 1.0): 15.485194686523528, (8, 100.0):
    11.230080149708563, (9, 0.0): 590145630456648.8, (9, 1e-06):
    15159936.566786056, (9, 0.0001): 151599.7644476303, (9, 0.01):
    1516.3927581144233, (9, 1.0): 15.577731739022951, (9, 100.0):
    11.250312536364762}

Condition no. w/o regularization (testing): {(1, 0.0): 19.106223493478968, (1,
     1e-06): 19.106223493478968, (1, 0.0001): 19.106223493478968, (1, 0.01):
    19.106223493478968, (1, 1.0): 19.106223493478968, (1, 100.0):
    19.106223493478968, (2, 0.0): 518.3247156838922, (2, 1e-06):
    518.3247156838922, (2, 0.0001): 518.3247156838922, (2, 0.01):
    518.3247156838922, (2, 1.0): 518.3247156838922, (2, 100.0):
    518.3247156838922, (3, 0.0): 15453.409539349419, (3, 1e-06):
    15453.409539349419, (3, 0.0001): 15453.409539349419, (3, 0.01):
    15453.409539349419, (3, 1.0): 15453.409539349419, (3, 100.0):
    15453.409539349419, (4, 0.0): 483506.95964456553, (4, 1e-06):
    483506.95964456553, (4, 0.0001): 483506.95964456553, (4, 0.01):
    483506.95964456553, (4, 1.0): 483506.95964456553, (4, 100.0):
    483506.95964456553, (5, 0.0): 15576875.806916751, (5, 1e-06):
    15576875.806916751, (5, 0.0001): 15576875.806916751, (5, 0.01):
    15576875.806916751, (5, 1.0): 15576875.806916751, (5, 100.0):
    15576875.806916751, (6, 0.0): 507514157.93791586, (6, 1e-06):
    507514157.93791586, (6, 0.0001): 507514157.93791586, (6, 0.01):
    507514157.93791586, (6, 1.0): 507514157.93791586, (6, 100.0):
    507514157.93791586, (7, 0.0): 16416963221.690588, (7, 1e-06):
    16416963221.690588, (7, 0.0001): 16416963221.690588, (7, 0.01):
    16416963221.690588, (7, 1.0): 16416963221.690588, (7, 100.0):
    16416963221.690588, (8, 0.0): 526138212360.06726, (8, 1e-06):
    526138212360.06726, (8, 0.0001): 526138212360.06726, (8, 0.01):
    526138212360.06726, (8, 1.0): 526138212360.06726, (8, 100.0):
    526138212360.06726, (9, 0.0): 17234710195847.965, (9, 1e-06):
    17234710195847.965, (9, 0.0001): 17234710195847.965, (9, 0.01):
    17234710195847.965, (9, 1.0): 17234710195847.965, (9, 100.0):
    17234710195847.965}

Condition no. w/ regularization (testing): {(1, 0.0): 19.106223493478968, (1,
    1e-06): 19.10622106719703, (1, 0.0001): 19.105980868437282, (1, 0.01):
    19.08199246082843, (1, 1.0): 16.961404261076325, (1, 100.0):
    2.6451792838145556, (2, 0.0): 518.3247156838922, (2, 1e-06):
```

```
    518.3226504337313, (2, 0.0001): 518.1182721658406, (2, 0.01):
    498.46409096921354, (2, 1.0): 104.24772742787077, (2, 100.0):
    3.1793725472959626, (3, 0.0): 15453.409539349419, (3, 1e-06):
    15451.66371493911, (3, 0.0001): 15280.758144376809, (3, 0.01):
    7255.702908857381, (3, 1.0): 135.96014308288903, (3, 100.0):
    3.4819683571427062, (4, 0.0): 483506.95964456553, (4, 1e-06):
    481880.15127956885, (4, 0.0001): 361474.59815095615, (4, 0.01):
    13910.423706190382, (4, 1.0): 143.60323838786084, (4, 100.0):
    3.6842914558245314, (5, 0.0): 15576875.806916751, (5, 1e-06):
    14097391.456519287, (5, 0.0001): 1355131.3306713083, (5, 0.01):
    14828.880943341705, (5, 1.0): 148.8798051509931, (5, 100.0):
    3.8339619443682857, (6, 0.0): 507514157.93791586, (6, 1e-06):
    117405576.39922069, (6, 0.0001): 1522812.5394117208, (6, 0.01):
    15273.969715717929, (6, 1.0): 153.2201656595726, (6, 100.0):
    3.9522336795648543, (7, 0.0): 16416963221.690588, (7, 1e-06):
    154944836.79662007, (7, 0.0001): 1564063.012888165, (7, 0.01):
    15642.600206473078, (7, 1.0): 156.92276068406156, (7, 100.0):
    4.049999161567223, (8, 0.0): 526138212360.06726, (8, 1e-06):
    159586290.01653725, (8, 0.0001): 1596342.7662737397, (8, 0.01):
    15963.988048226052, (8, 1.0): 160.15416845668432, (8, 100.0):
    4.133439868338479, (9, 0.0): 17234710195847.965, (9, 1e-06):
    162487441.82454658, (9, 0.0001): 1624890.1125685496, (9, 0.01):
    16249.430491895779, (9, 1.0): 163.02397875475458, (9, 100.0):
    4.206349805749418}

2. Equivalence check with scikit-learn

Coefficient L2-norm differences across all (M, $\lambda$):

     M    $lambda$            l2
0    1    0.000000  1.180183e-15
1    1    0.000001  2.234281e-15
2    1    0.000100  1.601186e-15
3    1    0.010000  6.280370e-16
4    1    1.000000  1.110223e-16
5    1  100.000000  1.734723e-18
6    2    0.000000  1.651249e-14
7    2    0.000001  1.546362e-15
8    2    0.000100  5.925929e-15
9    2    0.010000  6.179030e-15
10   2    1.000000  2.001483e-16
11   2  100.000000  4.911328e-18
12   3    0.000000  3.748011e-12
13   3    0.000001  3.077815e-12
14   3    0.000100  2.359466e-12
15   3    0.010000  3.748890e-14
16   3    1.000000  2.543841e-16
17   3  100.000000  5.222209e-18
18   4    0.000000  1.165418e-10
19   4    0.000001  6.354021e-11
20   4    0.000100  1.529848e-11
21   4    0.010000  3.272221e-14
22   4    1.000000  2.676651e-16
23   4  100.000000  5.502793e-18
24   5    0.000000  1.690115e-08
25   5    0.000001  1.350438e-09
26   5    0.000100  1.231346e-11
27   5    0.010000  9.366574e-14
28   5    1.000000  1.520235e-16
```

```
29  5  100.000000  4.925668e-18
30  6    0.000000  1.266891e-06
31  6    0.000001  3.742331e-09
32  6    0.000100  1.236106e-11
33  6    0.010000  1.322486e-13
34  6    1.000000  1.825338e-16
35  6  100.000000  6.329364e-18
36  7    0.000000  2.679460e-05
37  7    0.000001  8.541492e-10
38  7    0.000100  9.051553e-12
39  7    0.010000  1.518729e-13
40  7    1.000000  2.258674e-16
41  7  100.000000  7.961322e-18
42  8    0.000000  8.934914e-03
43  8    0.000001  2.702463e-09
44  8    0.000100  2.718318e-11
45  8    0.010000  1.369087e-13
46  8    1.000000  3.341178e-16
47  8  100.000000  5.818439e-18
48  9    0.000000  4.322710e+02
49  9    0.000001  2.989578e-09
50  9    0.000100  1.409848e-11
51  9    0.010000  1.333104e-13
52  9    1.000000  2.436777e-16
53  9  100.000000  7.256869e-18

1. Grid search over (M, $\lambda$)

Best (M_hat, Lambda) = (6, 1e-06)

Minimum Test MSE = 0.013362127882421209

2. Final reflections

Best coefficients (M=6, $\lambda$=1e-06):
[ -0.08958942   8.47143248  -7.40357783 -40.30544707  29.78715023
  46.21868656 -36.71097017]

Ill-conditioned coefficients (M=7, $\lambda$=0):
[-8.54431228e-02  8.45284633e+00 -6.55162206e+00 -6.34142787e+01
  1.70700841e+02 -2.94806939e+02  3.23931451e+02 -1.39156624e+02]

Over-regularized coefficients (M=7, $\lambda$=1):
[ 0.45998642 -0.42911245 -0.45704387 -0.33176584 -0.20989608 -0.11739408
 -0.05236374 -0.00827713]
```

Listing 4: Output terminal (selected) for *ridge.py*

IV. GMM & REGRESSION FOR MARINE CRAFT DYNAMICS

- The predictor model used in this work is a combination of smooth deterministic terms and harmonic terms. The smooth deterministic terms are expressed in the form of *Legendre* polynomials and the harmonic terms are expressed in the form of sine/cosine wave. The predictor for this problem is written as,

$$\hat{y}(t) = \sum_{i=0}^{n} \beta_i P_i(\tilde{t}) + \sum_{j=1}^{m} \left(a_j cos(\omega_j \pi \tilde{t}) + b_j sin(\omega_j \pi \tilde{t}) \right) \tag{1}$$

where, P_i is the *Legendre* polynomial of degree i with corresponding weights β_i, and a_j and b_j are the weights (amplitude) for the harmonic terms with frequency ω_j. This predictor is formulated to capture both the smooth varying trend and the oscillatory responses due to waves.

For improving the condition number of the design matrix, t is normalized and expressed as:

$$\tilde{t} = \frac{2\,(t - t_{min})}{t_{max} - t_{min}} - 1$$

which conforms to the bound of $[-1, 1]$.

After setting $n = m = 2$ and $\omega \in \{1, 2\}$, the final form of equation (1) stands as,

$$\hat{y}(t) = \sum_{i=0}^{2} \beta_i P_i(\tilde{t}) + \sum_{j=1}^{2} \left(a_j cos(\omega_j \pi \tilde{t}) + b_j sin(\omega_j \pi \tilde{t}) \right) \tag{2}$$

$\{(t_i, y_i)\}_{i=1}^{N}$ are split into 70% training and 30% testing data, and the design matrices were formed. The condition number of the design matrices were found to be-

$$\kappa \left(A^T A\right)_{train} = 1.1 \times 10^5, \qquad \kappa \left(A^T A\right)_{test} = 2.86 \times 10^{10}$$

which points to a potential ill-conditioned system. Performing regression to this system to

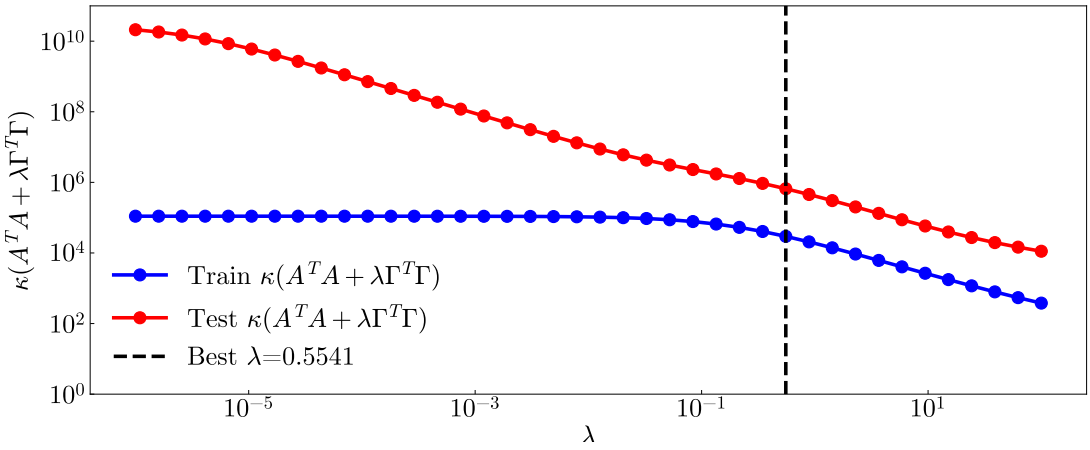


Figure 10: Condition number as a function of λ.

find the optimal weights can result in overfitting of the data. To remedy this, regularization

was performed on A^TA which improved the condition number as can be seen from figure 10 and normal equations were formulated to find out the weights.

$$\hat{\theta} = \left(A^TA + \lambda\Gamma^T\Gamma\right)^{-1} A^Ty$$

The best weights were picked by doing a grid search to find the value of λ for which the test MSE is minimum. Figure 11 shows the train and test MSE, and how the minimum test MSE was clearly distinguishable. The best weights corresponding to $\lambda = 0.5541$ came out

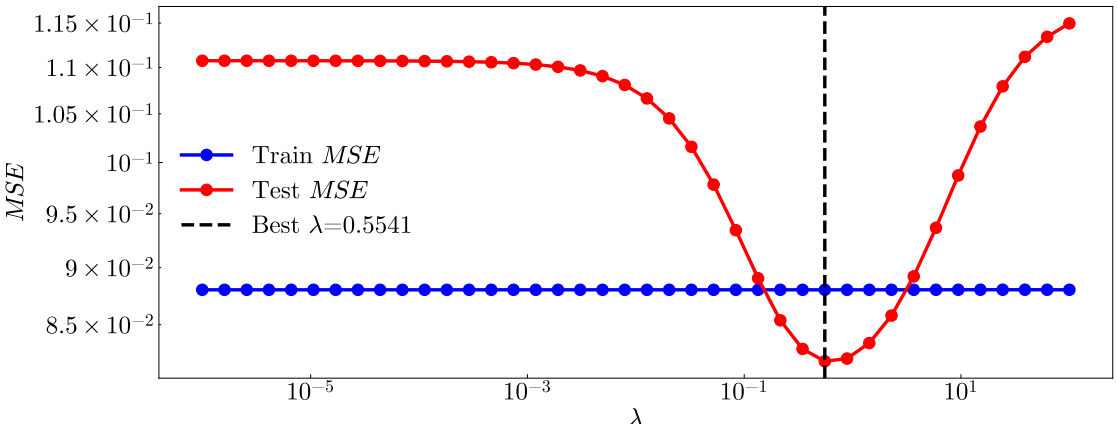


Figure 11: MSE as a function of λ.

to be-

$$\beta_i = \begin{pmatrix} -0.00744026 \\ 1.17051422 \\ -0.46714436 \end{pmatrix}, \qquad a_j = \begin{pmatrix} -0.05609375 \\ 0.0089983 \end{pmatrix}, \qquad b_j = \begin{pmatrix} -0.07363466 \\ 0.00805816 \end{pmatrix}$$

The model shown in figure 12 is obtained by finding a balance between bias and variance,

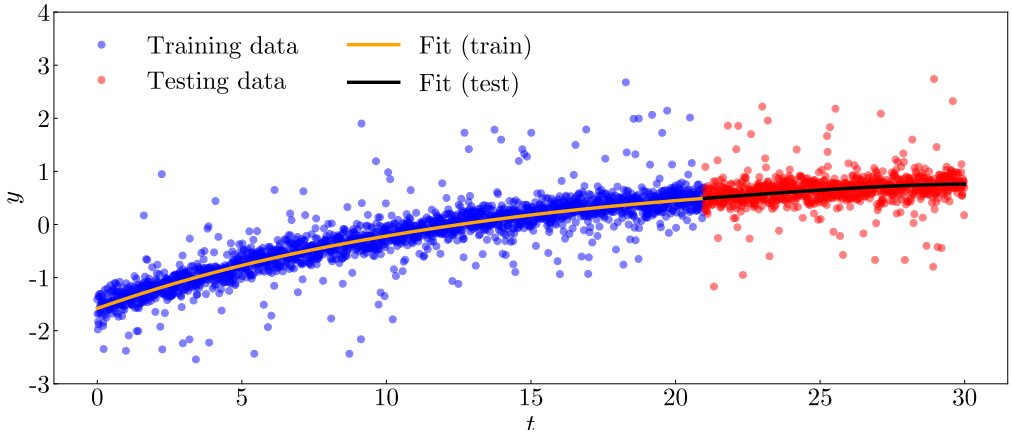


Figure 12: Predicted model via *Ridge* regression.

so that it does not suffer from underfitting due to high λ or overfitting due to ill-conditioned system with unregularized weights.

After building the forecasting model, the noise model was formulated in the form-

$$p(\epsilon) = \sum_{k=1}^{2} \pi_k \mathcal{N}\left(\epsilon|\mu_k, \sigma_k^2\right) \tag{3}$$

This is a 1D Gaussian mixture model and has two components- one relates to background sea state and another to slamming events. The distribution of training residual was plotted in figure 13. The parameters for the 1D GMM are found to be-

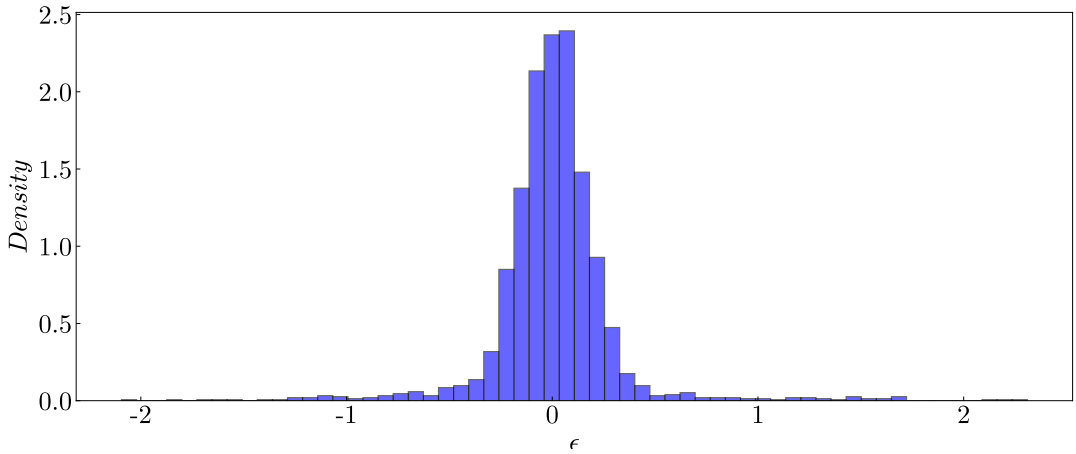


Figure 13: Training residual distribution.

For $k = 1$ (background sea state),

$$\pi_1 = 0.9017, \qquad \mu_1 = 0.0061, \qquad \sigma_1^2 = 0.0226, \qquad \sigma_1 = 0.1502$$

For $k = 2$ (Slamming events),

$$\pi_2 = 0.0983, \qquad \mu_2 = 0.0153, \qquad \sigma_2^2 = 0.6882, \qquad \sigma_2 = 0.8296$$

As expected, figure 14 shows the large variance in the slamming events with probability of

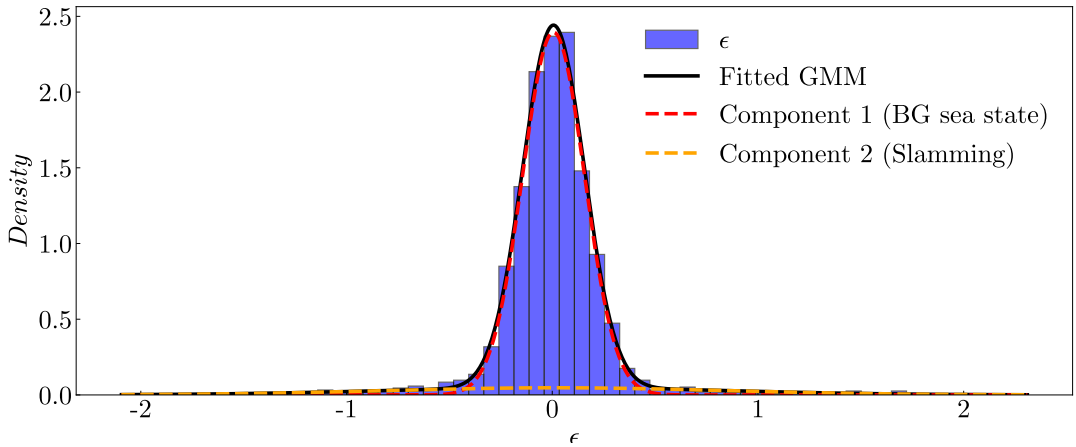


Figure 14: Distribution of two components of the 1D GMM along with the total fit.

~ 0.1, while background sea state has a smaller variance with the probability of ~ 0.9.

- As mentioned in the previous part, *Legendre* polynomial was chosen to account for the smooth and steadily varying trend in acceleration signal. *Legendre* polynomial was particularly chosen instead of monomial basis because of its orthogonal property within $[-1, 1]$, which helps to improve the condition number of the system.

$$\int_{-1}^{1} P_i(\tilde{t}) P_j(\tilde{t}) d\tilde{t} = \frac{2}{2i+1} \delta_{ij}$$

For $n = 2$, the basis is

$$P_0 = 1, \qquad P_1 = \tilde{t}, \qquad P_2 = \frac{1}{2}\left(3\tilde{t}^2 - 1\right)$$

The harmonic basis was considered for oscillatory response of the acceleration signal. The final predictor after the regression takes the form

$$\begin{aligned} \hat{y}(t) = &- 0.00744026 + 1.17051422t - 0.46714436 \cdot \frac{1}{2}\left(3\tilde{t}^2 - 1\right) - 0.05609375cos\left(\pi\tilde{t}\right) \\ &- 0.07363466sin\left(\pi\tilde{t}\right) + 0.0089983cos\left(2\pi\tilde{t}\right) + 0.00805816sin\left(2\pi\tilde{t}\right) \end{aligned} \tag{4}$$

To achieve equation (4), regularization was done using a penalization term $\lambda\Gamma^T\Gamma$ to give the system better stability by decreasing its condition number. Penalization matrix $\Gamma = diag(0, 10^{-3}, 10^{-3}, 1, 1, 4, 4)$ was chosen in such a way that the oscillatory high frequency coefficients in the equation (1) get penalization weights equal to ω_j^2, while the low frequency coefficients get very less regularized. Optimal hyper-parameter λ was found by grid-searching over $[10^{-6}, 10^2]$ so that test MSE ($\lambda = 0.5541$) was minimized. This process of finding predictor is called *Ridge* regression.

The residuals $\epsilon_i = y_i - \hat{y}_i$ were modeled by a two-component 1D Gaussian Mixture Model (GMM) shown in equation (3). `sklearn.mixture.GaussianMixture` from the `scikit-learn` was used to find the weight, mean and variance of the two components. The fitted weights $(\pi_1, \pi_2) = (0.9017, 0.0983)$ and variances $(\sigma_1^2, \sigma_2^2) = (0.0226, 0.6882)$ separated the signal into two components - dominant low variance background sea and low-probability high variance slamming events, which make the noise model a good interpreter of the sea state.

A. Challenge submission I: Using the predictive model

Predicted $Vibration\ Dose\ Value$ (VDV) = 1.1563152.

B. Challenge submission II: Using the GMM model

$\sigma_{bg,pred} = 0.15019996$.

```python
import numpy as np
from numpy.polynomial.legendre import legvander
import matplotlib.pyplot as plt
import matplotlib as mpl
from numpy.linalg import solve
from sklearn.mixture import GaussianMixture
from scipy.integrate import simpson

# Load and shuffle data
data = np.loadtxt("hw4_train.csv", delimiter=",")
t = data[:, 0]
y = data[:, 1]

# normalize time to [-1, 1]
t_norm = 2 * (t - np.min(t)) / (np.max(t) - np.min(t)) - 1

# sort by time
id = np.argsort(t_norm)
t_norm = t_norm[id]
y = y[id]

# Use early time as training, later as validation
split = int(0.7 * len(t_norm))
t_train, y_train = t_norm[:split], y[:split]
t_test,  y_test  = t_norm[split:], y[split:]

# ===== Ridge regression model =====

# construct design matrix
def design_matrix(t, n_poly=2, omega=None):
    t = np.asarray(t, dtype=float)
    if omega is None:
        omega = [1.0, 2.0, 3.0]

    cols = []

    # orthogonal polynomial trend (Legendre basis)
    trend = legvander(t, n_poly)
    for i in range(trend.shape[1]):
        cols.append(trend[:, i])

    # harmonic oscillations
    for w in omega:
        cols.append(np.cos(w * np.pi * t))
        cols.append(np.sin(w * np.pi * t))

    return np.column_stack(cols)

# train and test design matrix
A_train = design_matrix(t_train, n_poly=2, omega=[1.0, 2.0])
A_test  = design_matrix(t_test,  n_poly=2, omega=[1.0, 2.0])

print("\nCondition no. of $A^TA$ (train):", np.linalg.cond(A_train.T @ A_train))
print("\nCondition no. of $A^TA$ (test):", np.linalg.cond(A_test.T @ A_test))


# regularization function
def gamma_t_gamma(n_poly, omega):
```

```
59
60      weights = []
61      # polynomial terms lightly penalized
62      for i in range(n_poly + 1):
63          if i == 0:
64              weights.append(0.0)
65          else:
66              weights.append(1e-3)
67      # harmonics penalized proportionally to the square of frequency
68      for k, w in enumerate(omega, start=1):
69          strength = (k**2)
70          weights += [strength, strength]
71      return np.diag(weights)
72
73  # ridge regression function
74  def ridge_regression(A, y, Gamma_t_Gamma, lam=1e-3, trim=0.02):
75      # initial fit
76      omega = solve(A.T @ A + lam * Gamma_t_Gamma, A.T @ y)
77      y_pred = A @ omega
78      resid = y - y_pred
79
80      # trim extreme residuals (slamming)
81      if trim > 0:
82          threshold = np.quantile(np.abs(resid), 1 - trim)
83          mask = np.abs(resid) < threshold
84          A_trim, y_trim = A[mask], y[mask]
85          omega = solve(A_trim.T @ A_trim + lam * Gamma_t_Gamma, A_trim.T @ y_trim)
86
87      return omega
88
89  Gamma_t_Gamma = gamma_t_gamma(n_poly=2, omega=[1.0, 2.0])
90
91  # grid search for best fit
92  lambda_ = np.logspace(-6, 2, 40)
93  mse_train, mse_test = [], []
94  cond_train, cond_test = [], []
95
96  for lam in lambda_:
97      theta = ridge_regression(A_train, y_train, Gamma_t_Gamma, lam=lam, trim=0.02)
98      y_train_pred = A_train @ theta
99      y_test_pred  = A_test  @ theta
100     mse_train.append(np.mean((y_train_pred - y_train)**2))
101     mse_test.append(np.mean((y_test_pred - y_test)**2))
102
103     # regularized condition no.
104     cond_train.append(np.linalg.cond(A_train.T @ A_train + lam * Gamma_t_Gamma))
105     cond_test.append(np.linalg.cond(A_test.T  @ A_test  + lam * Gamma_t_Gamma))
106
107 best_id = np.argmin(mse_test)
108 lambda_best = lambda_[best_id]
109 print(f"\nBest $\lambda$ = {lambda_best:.4e}, Test MSE = {mse_test[best_id]:.8f}")
110
111 # parameters for plotting
112 plt.rcParams["font.family"] = "serif"
113 plt.rcParams["font.serif"] = ["CMU Serif"]
114 plt.rcParams["mathtext.fontset"] = "cm"
115 plt.rcParams["font.size"] = 22
116 mpl.rcParams["axes.unicode_minus"] = False
117
```

```
# condition number plot
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(lambda_, cond_train, "bo-", ms=10, lw=3, label=fr"Train $\kappa(A^TA+\
    lambda\Gamma^T\Gamma)$")
ax.loglog(lambda_, cond_test,  "ro-", ms=10, lw=3, label=fr"Test $\kappa(A^TA+\
    lambda\Gamma^T\Gamma)$")
ax.axvline(lambda_best, color="k", lw=3, ls="--", label=fr"Best $\lambda$={
    lambda_best:.4f}")
plt.xlabel(r"$\lambda$")
plt.ylabel(r"$\kappa(A^TA+\lambda\Gamma^T\Gamma)$")
plt.ylim(1e0, 1e11)
ax.legend(frameon=False, loc='best')
ax.tick_params(axis="both", which="both", direction="in")
plt.savefig("marine_cond.pdf", dpi=1080)
plt.show()

# mse plot
fig, ax = plt.subplots(figsize=(15, 6))
ax.loglog(lambda_, mse_train, "bo-", ms=10, lw= 3, label="Train $MSE$")
ax.loglog(lambda_, mse_test, "ro-", ms=10, lw=3, label=r"Test $MSE$")
ax.axvline(lambda_best, color="k", lw=3, ls="--", label=rf"Best $\lambda$={
    lambda_best:.4f}")
plt.xlabel("$\lambda$")
plt.ylabel("$MSE$")
plt.legend(frameon=False, loc="best")
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("marine_mse.pdf", dpi=1080)
plt.show()

# best fit model
theta_best = ridge_regression(A_train, y_train, Gamma_t_Gamma, lam=lambda_best,
    trim=0.02)
print("\nBest $\theta$ =", theta_best)

y_pred_train = A_train @ theta_best
y_pred_test  = A_test  @ theta_best
print("\nPredicted acceleration (train): \n", y_pred_train)
print("\nPredicted acceleration (test): \n", y_pred_test)

# de-normalize time
t_o = 0.5 * (t_norm + 1) * (np.max(t) - np.min(t)) + np.min(t)
t_train_o = t_o[:split]
t_test_o  = t_o[split:]

# prediction plot
fig, ax = plt.subplots(figsize=(15, 6))
ax.plot(t_train_o, y_train, "bo", alpha=0.5, label="Training data")
ax.plot(t_test_o,  y_test,  "ro", alpha=0.5, label="Testing data")
ax.plot(t_train_o, y_pred_train, color="orange", ls="-", lw=3, label="Fit (train)"
    )
ax.plot(t_test_o,  y_pred_test,  "k-", lw=3, label='Fit (test)')
plt.xlabel("$t$")
plt.ylabel("$y$")
plt.ylim(-3, 4)
plt.legend(frameon=False, loc="best", ncol=2)
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("marine_predict.pdf", dpi=1080)
plt.show()

```

```python
# ===== GMM noise model =====

# residuals
resid_train = y_train - y_pred_train
resid_test  = y_test  - y_pred_test

# residual plot
fig, ax = plt.subplots(figsize=(15, 6))
ax.hist(resid_train, bins=60, density=True, color="blue", edgecolor="black", alpha=0.6)
plt.xlabel(r"$\epsilon$")
plt.ylabel(r"$Density$")
plt.tick_params(axis="both", which="both", direction="in")
plt.savefig("marine_residual.pdf", dpi=1080)
plt.show()

# 1D GMM with 2 components
gmm = GaussianMixture(n_components=2, covariance_type="full", random_state=23)
gmm.fit(resid_train.reshape(-1, 1))

means   = gmm.means_.ravel()
vars_   = gmm.covariances_.ravel()
stds    = np.sqrt(gmm.covariances_).ravel()
weights = gmm.weights_.ravel()

print("\nGMM parameters using training residual")
for i, (w, m, v, s) in enumerate(zip(weights, means, vars_, stds)):
    print(f"\nComponent {i+1}: weight={w:.4f}, mean={m:.4f}, variance={v:.4f}, std={s:.4f}")

# GMM plot
x_grid = np.linspace(min(resid_train), max(resid_train), 600).reshape(-1, 1)
pdf_total = np.exp(gmm.score_samples(x_grid))

# PDF of individual component
pdf_components = np.array([
    w * (1 / (np.sqrt(2*np.pi) * s)) * np.exp(-0.5 * ((x_grid - m) / s)**2)
    for w, m, s in zip(weights, means, stds)
])

colors = ["red", "orange"]
labels = ["BG sea state", "Slamming"]
fig, ax = plt.subplots(figsize=(15, 6))
ax.hist(resid_train, bins=60, density=True, color="blue", edgecolor="black", alpha=0.6, label="$\epsilon$")
ax.plot(x_grid, pdf_total, "k-", lw=3, label="Fitted GMM")
for k, (pdf, c, lbl) in enumerate(zip(pdf_components, colors, labels)):
    ax.plot(x_grid, pdf, "--", lw=3, color=c, label=f"Component {k+1} ({lbl})")
plt.xlabel(r"$\epsilon$")
plt.ylabel(r"$Density$")
plt.tick_params(axis="both", which="both", direction="in")
plt.legend(frameon=False, loc="best")
plt.savefig("marine_gmm.pdf", dpi=1080)
plt.show()

# ===== Challenge Submission I =====

# acceleration^4
a4 = np.abs(y_pred_test)**4
```

```
227
228 # Trapezoidal rule
229 vdv_trap = (np.trapz(a4, t_test_o)) ** 0.25
230
231 # Simpson's rule
232 vdv_simp = (simpson(a4, t_test_o)) ** 0.25
233
234 print("\nVibration Dose Value (VDV):")
235 print(f"\nTrapezoidal rule: {vdv_trap:.8f}")
236 print(f"\nSimpsons 1/3rd rule : {vdv_simp:.8f}")
237
238 # ===== Challenge Submission II =====
239
240 id_bgc = np.argmin(stds)
241 sigma_bgc_pred = stds[id_bgc]
242 id_slm = np.argmax(stds)
243
244 sigma_ratio = stds[id_slm] / stds[id_bgc]
245 p_slam = weights[id_slm]
246
247 print("\nGMM Noise:")
248 print(f"\nBackground sea state standard deviation = {sigma_bgc_pred:.8f}")
249 print(f"\nSlamming standard deviation = {stds[id_slm]:.8f}")
250 print(f"\nRatio of standard deviation of slamming and background sea = {
    sigma_ratio:.8f}")
251 print(f"\nSlamming probability = {p_slam:.4f}")
```

Listing 5: *marine.py*

```
Condition no. of $A^TA$ (train): 110063.1089055772

Condition no. of $A^TA$ (test): 28642884212.86928

Best $\lambda$ = 5.5410e-01, Test MSE = 0.08193715

Best $\theta$ = [-0.00744026  1.17051422 -0.46714436 -0.05609375 -0.07363466
    0.0089983
  0.00805816]

Predicted acceleration (train):
 [-1.58000678 -1.57810406 -1.57620236 ...  0.49134531  0.49176704
  0.49218858]

GMM parameters using training residual

Component 1: weight=0.9017, mean=0.0061, variance=0.0226, std=0.1502

Component 2: weight=0.0983, mean=0.0153, variance=0.6882, std=0.8296

Vibration Dose Value (VDV):

Trapezoidal rule: 1.15631521

Simpsons 1/3rd rule : 1.15631523

GMM Noise:

Background sea state standard deviation = 0.15019996

Slamming standard deviation = 0.82955471

Ratio of standard deviation of slamming and background sea = 5.52300217

Slamming probability = 0.0983
```

Listing 6: Output terminal (selected) for *marine.py*