Task_1_2

December 20, 2023

0.1 Task 1

[1]:
```
import numpy as np
import os

def read_data(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()
    return np.array([float(line.strip()) for line in lines])

def maximum_spacing_estimator(data):
    # Step 1: Sort the data
    sorted_data = np.sort(data)

    # Step 2: Calculate differences between consecutive order statistics
    differences = np.diff(sorted_data)
    #print(differences)

    # Step 3: Find the maximum spacing
    max_spacing = np.max(differences)
    print(max_spacing)

    # Step 4: Compute Maximum Spacing Estimator
    n = len(data)
    theta_mse = (n + 1) / n * max_spacing

    return theta_mse

# Generate three sets of sample data
np.random.seed(42)  # for reproducibility

# Getting the home directory
home_dir = os.path.expanduser("~")

# Constructing file paths
file_path_1 = os.path.join('sampleset_1_problemsheet4_ex1.txt')
file_path_2 = os.path.join('sampleset_2_problemsheet4_ex1.txt')
file_path_3 = os.path.join('sampleset_3_problemsheet4_ex1.txt')
```

```python
# Read data from the three files
sample_data_1 = read_data(file_path_1)
sample_data_2 = read_data(file_path_2)
sample_data_3 = read_data(file_path_3)

# Calculate Maximum Spacing Estimators for each set of samples
theta_mse_1 = maximum_spacing_estimator(sample_data_1)
theta_mse_2 = maximum_spacing_estimator(sample_data_2)
theta_mse_3 = maximum_spacing_estimator(sample_data_3)

# Display the results
print(f"Set 1: Maximum Spacing Estimator for  = {theta_mse_1:.4f}")
print(f"Set 2: Maximum Spacing Estimator for  = {theta_mse_2:.4f}")
print(f"Set 3: Maximum Spacing Estimator for  = {theta_mse_3:.4f}")
```

```
0.45489999999999986
0.2287999999999999
0.7923
Set 1: Maximum Spacing Estimator for  = 0.4701
Set 2: Maximum Spacing Estimator for  = 0.2334
Set 3: Maximum Spacing Estimator for  = 0.8913
```

0.2 Task 2

[2]:
```python
def get_eps(m):
    return np.random.normal(0, 0.5, size=(m, 1))


def evolution(z_n):
    return 0.99 * z_n + get_eps(m=len(z_n))


def mse(y, _y):
    return (y - _y) ** 2


def mse_kalman(kalman_means, true_signal):
    sse = 0
    for i, point in enumerate(true_signal):
        sse += mse(point, kalman_means[i])

    return sse / len(true_signal)


def mse_enkf(enkf_means, true_signal):
    sse = 0
```

```
    for i, point in enumerate(true_signal):
        sse += mse(point, enkf_means[i].mean())

    return sse / len(true_signal)
```

0.3 Kalman Filter

[3]:
```
def kalman_filter(initial_mean, initial_cov, evolution_noise, evolution_coef,␣
↪observations, num_steps):
    # Initialize the means and covariances over time for plotting
    state_means = np.zeros(num_steps)
    state_covariances = np.zeros(num_steps)

    # Initialization of the mean and the variances
    state_mean = initial_mean  # m_0
    state_covariance = initial_cov  # C_0

    for step in range(num_steps):
        # 1. produce prediction
        predicted_state_mean = evolution_coef * state_mean  # M^-1 0.99
        predicted_state_covariance = evolution_coef ** 2 * state_covariance +␣
↪evolution_noise ** 2

        # 2 update the model by using the new observation to compute the kalman␣
↪gain
        k_gain = predicted_state_covariance / (predicted_state_covariance +␣
↪evolution_noise ** 2)
        state_mean = predicted_state_mean - (k_gain * (predicted_state_mean -␣
↪observations[step]))
        state_covariance = predicted_state_covariance - (k_gain *␣
↪predicted_state_covariance)

        state_means[step] = state_mean
        state_covariances[step] = state_covariance

    return state_means, state_covariances
```

0.4 Ensemble Kalman Filter

[4]:
```
def ensemble_kalman_filter(obs, initial_ensemble, num_steps):
    ensemble_size = initial_ensemble.shape[0]
    state_means = np.zeros((num_steps, ensemble_size))

    # Initialization
    ensemble = initial_ensemble
```

```
    for step in range(num_steps):
        # Prediction step
        ensemble = evolution(ensemble)

        # Update step
        kalman_gain = np.cov(ensemble, rowvar=False) / (np.cov(ensemble,␣
 ↪rowvar=False) + 0.5 ** 2)

        ensemble = ensemble - kalman_gain * (ensemble - obs[step])

        # compute the mean for the ensemble
        state_means[step, :] = np.mean(ensemble, axis=1)

    return state_means
```

0.5 Comparison

[5]:
```
data_path: str = "./reference_signal.txt"
obs_path: str = "./data.txt"

# load the true signal
signal: list = []
observations: list = []
with open(data_path) as reference_txt:
    for line in reference_txt:
        signal.append(float(line))

with open(data_path) as reference_txt:
    for line in reference_txt:
        observations.append(float(line))

# Define the number of steps
num_steps: int = len(signal)

# Generate true signal and observations
true_signal = np.array(signal)
observations = np.array(observations)

# Kalman Filter
kf_means, _ = kalman_filter(observations=observations, initial_mean=0,␣
 ↪initial_cov=0.5, evolution_coef=0.99, evolution_noise=0.5,␣
 ↪num_steps=num_steps)

# Ensemble Kalman Filter with different ensemble sizes
ensemble_sizes = [5, 10, 25, 50]
mse_kal = mse_kalman(kalman_means=kf_means, true_signal=true_signal)
```

```
mse_enkfs = []
for m in ensemble_sizes:
    initial_ensemble = np.random.normal(0, 0.5, size=(m, 1))
    enkf_means = ensemble_kalman_filter(observations, initial_ensemble,␣
 ↪num_steps=num_steps)
    mse_enkfs.append(mse_enkf(enkf_means=enkf_means, true_signal=true_signal))

print(f"Mean Squared Error of Kalman Filter: {mse_kal}")
for ensemble_size, mse_en in zip(ensemble_sizes, mse_enkfs):
    print(f"Mean Squared Error of Ensemble Kalman Filter with Ensemble Size␣
 ↪{ensemble_size}: {mse_en}")
```

```
Mean Squared Error of Kalman Filter: 0.01522766848180877
Mean Squared Error of Ensemble Kalman Filter with Ensemble Size 5:
0.05372808647281595
Mean Squared Error of Ensemble Kalman Filter with Ensemble Size 10:
0.036104322433904774
Mean Squared Error of Ensemble Kalman Filter with Ensemble Size 25:
0.025994421293569134
Mean Squared Error of Ensemble Kalman Filter with Ensemble Size 50:
0.024108600491226966
```

0.6 Comments:

It Appears, that the Kalman Filter performs a bit better than the Ensemble Kalman Filter. This can have different reasons. One would be the simplicity of the data. Ensemble Kalman Filters are mostly used when the data has high dimensionality (e.g. 10e8). However in this case the dimensionality of the data is 1. Another reason could be that the number of Ensemble members is rather small. It is observable that the MSE for an increasing number of members is becoming smaller. So increasing the number of ensemble members does improve the performance. At the end it is hard to say why exactly the performance is worse but it will be a combination of the above mentioned facts

Task 3.

For $A \in R^{m \times n}$ & $u, v \in R^{m}$, proove Sherman Morrison Formula.

We can proove by considering LHS $= X^{-1}$ & RHS $= Y$

so $X^{-1} = Y$
if & only if $XY = YX = I$.

Starting with $XY = I$.

$$XY = (A + uv^T)\left(A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}\right)$$

$$= AA^{-1} + uv^TA^{-1} - \frac{AA^{-1}uv^TA^{-1} + uv^TA^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}$$

$$= I + uv^TA^{-1} - \frac{uv^TA^{-1} + uv^TA^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}$$

$$= I + uv^TA^{-1} - \frac{u(1 + v^TA^{-1}u)v^TA^{-1}}{1 + v^TA^{-1}u}$$

$$= I + uv^TA^{-1} - uv^TA^{-1}$$

$$= I$$

Page No

Date

for $YX = I$

$$= \left(A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}\right)(A + uv^T)$$

$$= A^{-1}A + A^{-1}uv^T - \frac{A^{-1}uv^TA^{-1}A}{1+v^TA^{-1}u} - \frac{A^{-1}uv^TA^{-1}uv^T}{1+v^TA^{-1}u}$$

$$= A^{-1}A + A^{-1}uv^T - \frac{(A^{-1}uv^T + A^{-1}uv^TA^{-1}uv^T)}{1+v^TA^{-1}u}$$

$$= I + A^{-1}uv^T - \frac{A^{-1}u(v^T + v^TA^{-1}uv^T)}{1+v^TA^{-1}u}$$

$$= I + A^{-1}uv^T - \frac{A^{-1}u(1 + v^TA^{-1}u)v^T}{1+v^TA^{-1}u}$$

$$= I + A^{-1}uv^T - A^{-1}uv^T$$

$$= I$$

Thus we have prooved $XY = YX = I$

Hence y is inverse of X

$$\therefore X^{-1} = (A + uv^T)^{-1} = Y = \left(A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}\right)$$