Midterm 1 Project, Problem 1

```
# Import Dependencies
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
```

Methodology

We consider a target function:

$$u_g(x) = \sin(2x+1) + 0.2e^{1.3x}$$

1. We generate training data by sampling 300 data points from it without noise, uniformly distributed within the domain $x \in [-1, 1)$.

2. To fit the training data, we use a fully-connected neural network made of 3 hidden layers each with 20 units and use hyperbolic tangent (tanh) as the activation function.

This code implements a two-stage neural network training process to approximate a nonlinear target function sin(2 * x + 1) + 0.2 * exp(1.3 * x).

- It starts by generating 300 training points from -1 to 1 and calculates the target function values.
- In the first stage, a neural network with 3 hidden layers (each containing 30 neurons) is trained to fit the target function, yielding initial predictions and computing the residual (error).
- The second stage trains another neural network with 3 hidden layers (each containing 20 neurons) to specifically learn this residual.
- The final prediction is obtained by combining the outputs from both stages—adding the initial predictions to the learned residuals—enhancing the accuracy in approximating the target function.

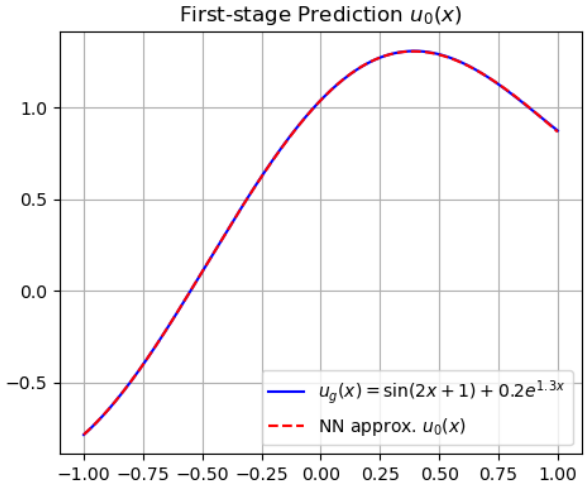
```
# Target function from Equation (2)
def target_function(x):
    return np.sin(2 * x + 1) + 0.2 * np.exp(1.3 * x)

# Generate training data (300 points between -1 and 1)
x_train = np.linspace(-1, 1, 300)
y_train = target_function(x_train)

# Define the neural network model (custom layers)
def create_model(layers):
    model = Sequential()
    model.add(Dense(layers[0], activation='tanh', input_shape=(1,)))
```

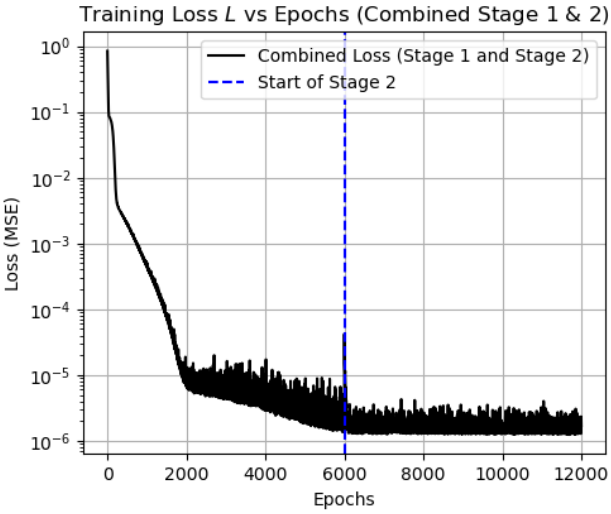
```
    for units in layers[1:]:
        model.add(Dense(units, activation='tanh'))
    model.add(Dense(1))  # Output layer
    return model

# Compile and train the model
def train_model(model, x, y, epochs=6000, learning_rate=0.0001):
    model.compile(optimizer=Adam(learning_rate=learning_rate),
loss='mse')
    history = model.fit(x, y, epochs=epochs, verbose=0)
    return model, history

# Train Stage 1 (Initial training on the target function with 30 units
in each layer)
model1 = create_model([30, 30, 30])
model1, history1 = train_model(model1, x_train, y_train)

# Stage 1 predictions and error (residual)
y_pred1 = model1.predict(x_train).squeeze()
error1 = y_train - y_pred1  # Residual error from Stage 1

# Train Stage 2 - Network with 3 hidden layers (each with 20 units)
model2 = create_model([20, 20, 20])
model2, history2 = train_model(model2, x_train, error1)

# Stage 2 predictions and combined results
y_pred2 = model2.predict(x_train).squeeze()
final_prediction = y_pred1 + y_pred2  # Combined prediction

# Residual error after Stage 2
residual_final = y_train - final_prediction

# Combine loss history for both stages
combined_loss = np.concatenate((history1.history['loss'],
history2.history['loss']))

# Adjust epochs for Stage 2 to appear after Stage 1
stage2_start_epoch = len(history1.history['loss'])
```

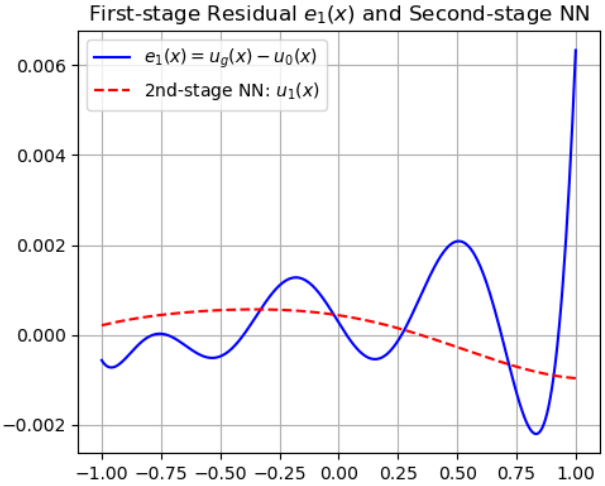
```
WARNING:tensorflow:From C:\Users\jhyan\AppData\Roaming\Python\
Python311\site-packages\keras\src\backend.py:873: The name
tf.get_default_graph is deprecated. Please use
tf.compat.v1.get_default_graph instead.

WARNING:tensorflow:From C:\Users\jhyan\AppData\Roaming\Python\
Python311\site-packages\keras\src\utils\tf_utils.py:492: The name
tf.ragged.RaggedTensorValue is deprecated. Please use
tf.compat.v1.ragged.RaggedTensorValue instead.
```

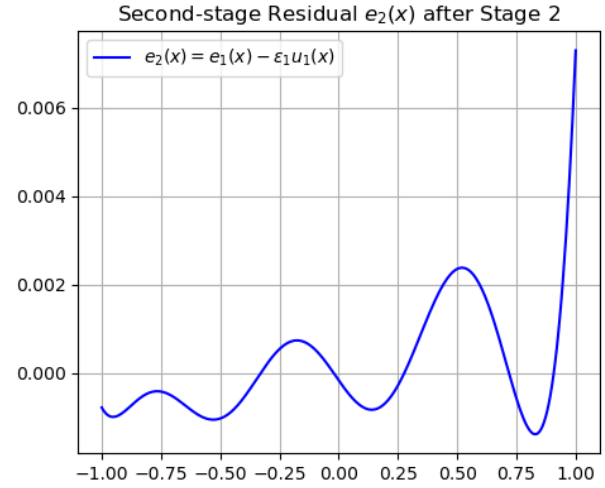
```
10/10 [==============================] - 0s 990us/step
10/10 [==============================] - 0s 773us/step

# Plotting the results
plt.figure(figsize=(10, 12))

# Plot (a) Target function vs Stage 1 Prediction
plt.subplot(3, 2, 1)
plt.plot(x_train, y_train, label=r"$u_g(x) = \sin(2x + 1) +
0.2e^{1.3x}$", color='b')
plt.plot(x_train, y_pred1, label="NN approx. $u_0(x)$", color='r',
linestyle='--')
plt.title("First-stage Prediction $u_0(x)$")
plt.legend()
plt.grid(True)

# Plot (b) First-stage residual (e1)
plt.subplot(3, 2, 3)
plt.plot(x_train, error1, label=r"$e_1(x) = u_g(x) - u_0(x)$",
color='b')
plt.plot(x_train, y_pred2, label="2nd-stage NN: $u_1(x)$", color='r',
linestyle='--')
plt.title("First-stage Residual $e_1(x)$ and Second-stage NN")
plt.legend()
plt.grid(True)

# Plot (c) Second-stage residual (e2)
plt.subplot(3, 2, 5)
plt.plot(x_train, residual_final, label=r"$e_2(x) = e_1(x) - \
epsilon_1u_1(x)$", color='b')
plt.title("Second-stage Residual $e_2(x)$ after Stage 2")
plt.legend()
plt.grid(True)

# Plot (d) Combined loss for Stage 1 and Stage 2
plt.subplot(3, 2, 2)
plt.plot(np.arange(len(combined_loss)), combined_loss, label="Combined
Loss (Stage 1 and Stage 2)", color='black')

# Mark transition between Stage 1 and Stage 2
plt.axvline(stage2_start_epoch, color='blue', linestyle='--',
label="Start of Stage 2")

plt.yscale("log")
plt.title("Training Loss $L$ vs Epochs (Combined Stage 1 & 2)")
plt.xlabel("Epochs")
plt.ylabel("Loss (MSE)")
plt.legend()
plt.grid(True)
```

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









Midterm 1 Project, Problem 1

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.initializers import RandomNormal
```

Methodology

We consider a target function:

$$u_g(x) = \sin(2x+1) + 0.2e^{1.3x}$$

1. We generate training data by sampling 300 data points from it without noise, uniformly distributed within the domain $x \in [-1, 1)$.

2. To fit the training data, we use a fully-connected neural network made of 3 hidden layers each with 20 units and use hyperbolic tangent (tanh) as the activation function.

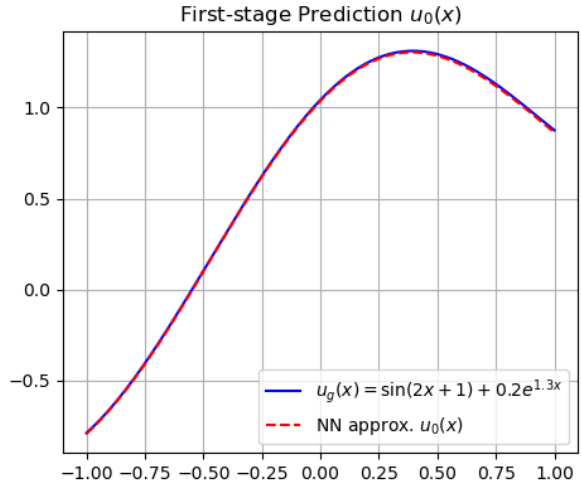
This code implements a two-stage neural network training process to approximate the nonlinear function sin(2 * x + 1) + 0.2 * exp(1.3 * x) using 300 training points between -1 and 1.

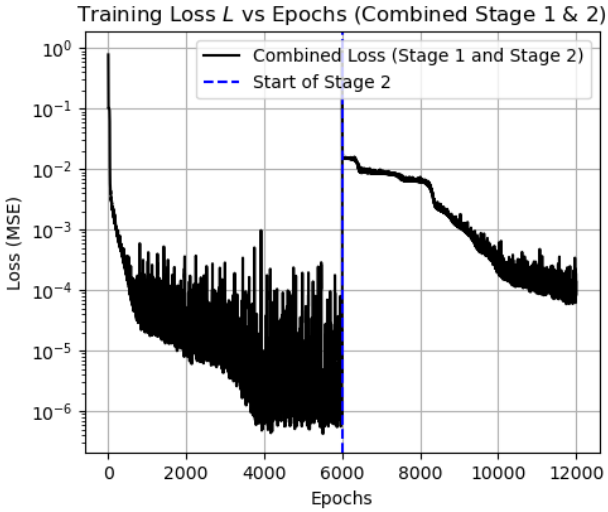
- In the first stage, a neural network with three hidden layers (30 neurons each) captures the primary behavior of the function. The resulting residual (difference between the true values and predictions) is then normalized.
- In the second stage, another neural network with three hidden layers (20 neurons each) learns this normalized residual.
- The final output combines the first stage's prediction with the scaled correction from the second stage, yielding a more accurate approximation of the target function through iterative refinement.

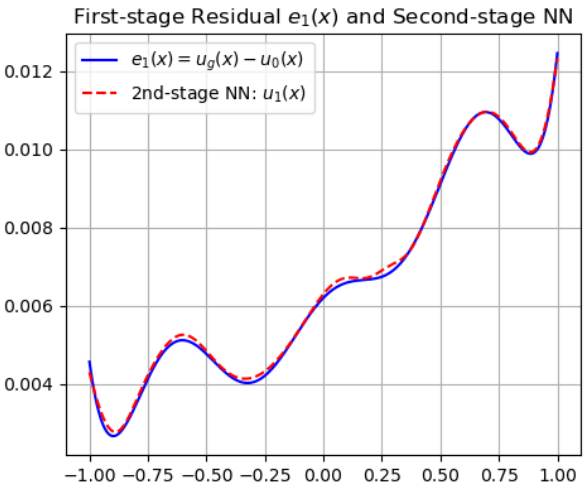
```
# Target function (Equation (2) from the Lecture)
def target_function(x):
    return np.sin(2 * x + 1) + 0.2 * np.exp(1.3 * x)

# Generate training data (300 points in the range [-1, 1])
x_train = np.linspace(-1, 1, 300)
y_train = target_function(x_train)

# Define the neural network model (with custom layers)
def create_model(layers):
    model = Sequential()
    # Use RandomNormal initialization for weights
    model.add(Dense(layers[0], activation='tanh', input_shape=(1,),
kernel_initializer=RandomNormal(mean=0.0, stddev=0.1)))
    for units in layers[1:]:
```

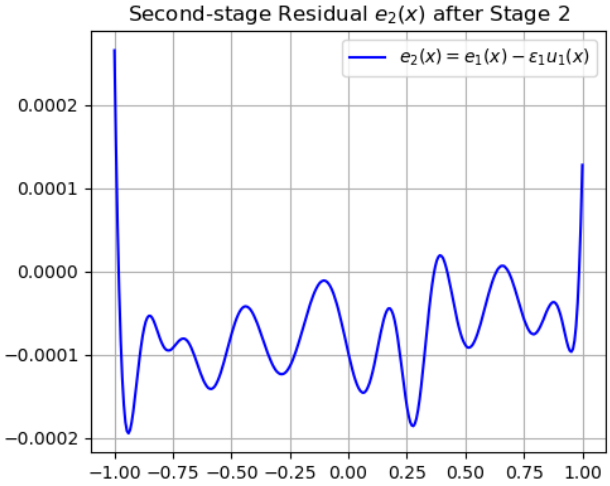
```
        model.add(Dense(units, activation='tanh',
kernel_initializer=RandomNormal(mean=0.0, stddev=0.1)))
    model.add(Dense(1))  # Output layer
    return model

# Compile and train the model
def train_model(model, x, y, epochs=6000, learning_rate=0.001):
    model.compile(optimizer=Adam(learning_rate=learning_rate),
loss='mse')
    history = model.fit(x, y, epochs=epochs, verbose=0)
    return model, history

# Stage 1 training (Initial model with 30 neurons in each hidden
layer)
model1 = create_model([30, 30, 30])
model1, history1 = train_model(model1, x_train, y_train)

# Stage 1 prediction and error (residual)
y_pred1 = model1.predict(x_train).squeeze()
error1 = y_train - y_pred1  # Residual from Stage 1

# Normalize the residual (using RMS normalization)
error1_norm = error1 / np.sqrt(np.mean(error1**2))

# Stage 2 training (3 hidden layers, each with 20 neurons)
model2 = create_model([20, 20, 20])
model2, history2 = train_model(model2, x_train, error1_norm)

# Stage 2 prediction and combined results
y_pred2 = model2.predict(x_train).squeeze()
final_prediction = y_pred1 + np.sqrt(np.mean(error1**2)) * y_pred2  #
Combine predictions

# Calculate the final residual
residual_final = y_train - final_prediction

# Combine loss history from both stages
combined_loss = np.concatenate((history1.history['loss'],
history2.history['loss']))

10/10 [==============================] - 0s 797us/step
10/10 [==============================] - 0s 889us/step

# Plotting the results
plt.figure(figsize=(10, 12))

# Plot (a) Target function vs Stage 1 Prediction
plt.subplot(3, 2, 1)
plt.plot(x_train, y_train, label=r"$u_g(x) = \sin(2x + 1) +
0.2e^{1.3x}$", color='b')
plt.plot(x_train, y_pred1, label="NN approx. $u_0(x)$", color='r',
```

```
linestyle='--')
plt.title("First-stage Prediction $u_0(x)$")
plt.legend()
plt.grid(True)

# Plot (b) First-stage residual (e1)
plt.subplot(3, 2, 3)
plt.plot(x_train, error1, label=r"$e_1(x) = u_g(x) - u_0(x)$",
color='b')
plt.plot(x_train, np.sqrt(np.mean(error1**2)) * y_pred2, label="2nd-
stage NN: $u_1(x)$", color='r', linestyle='--')
plt.title("First-stage Residual $e_1(x)$ and Second-stage NN")
plt.legend()
plt.grid(True)

# Plot (c) Second-stage residual (e2)
plt.subplot(3, 2, 5)
plt.plot(x_train, residual_final, label=r"$e_2(x) = e_1(x) - \
epsilon_1u_1(x)$", color='b')
plt.title("Second-stage Residual $e_2(x)$ after Stage 2")
plt.legend()
plt.grid(True)

# Plot (d) Combined loss for Stage 1 and Stage 2
# Define the starting epoch of Stage 2 (end of Stage 1 training)
stage2_start_epoch = len(history1.history['loss'])

# Plot (d) Combined loss for Stage 1 and Stage 2
plt.subplot(3, 2, 2)
plt.plot(np.arange(len(combined_loss)), combined_loss, label="Combined
Loss (Stage 1 and Stage 2)", color='black')

# Mark transition between Stage 1 and Stage 2
plt.axvline(stage2_start_epoch, color='blue', linestyle='--',
label="Start of Stage 2")

plt.yscale("log")
plt.title("Training Loss $L$ vs Epochs (Combined Stage 1 & 2)")
plt.xlabel("Epochs")
plt.ylabel("Loss (MSE)")
plt.legend()
plt.grid(True)

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


First-stage Prediction $u_0(x)$
$u_g(x) = \sin(2x+1) + 0.2e^{1.3x}$
NN approx. $u_0(x)$


Training Loss L vs Epochs (Combined Stage 1 & 2)
Combined Loss (Stage 1 and Stage 2)
Start of Stage 2
Loss (MSE)
Epochs

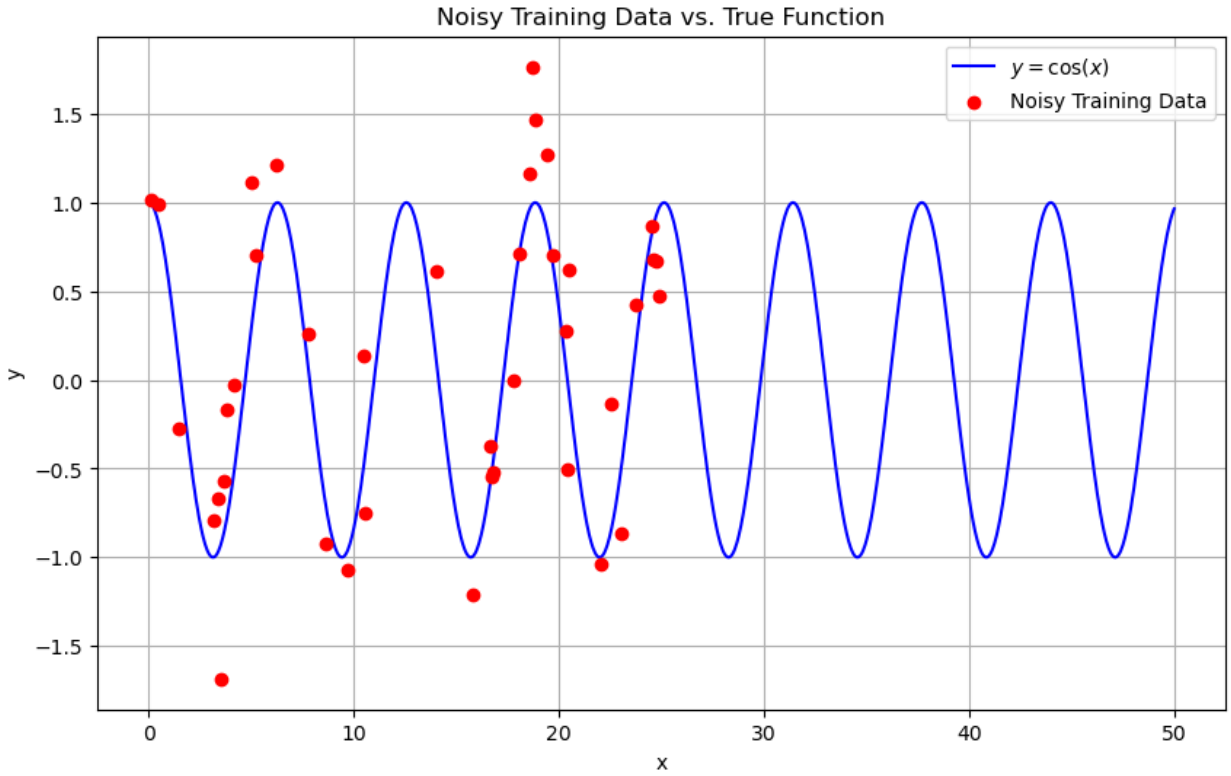

First-stage Residual $e_1(x)$ and Second-stage NN
$e_1(x) = u_g(x) - u_0(x)$
2nd-stage NN: $u_1(x)$


Second-stage Residual $e_2(x)$ after Stage 2
$e_2(x) = e_1(x) - \varepsilon_1 u_1(x)$

```
# Import Dependencies
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel,
ExpSineSquared, ConstantKernel
from sklearn.kernel_ridge import KernelRidge
```

Midterm 1 Project, Problem 2-1

```
# Generate 1000 evenly spaced x data points in the range [0, 50]
x = np.linspace(0, 50, 1000)

# True function y = cos(x)
y_true = np.cos(x)

# Randomly select 40 points from the first 500 data points (i.e., x ∈
[0, 25])
np.random.seed(42)  # Ensure reproducibility
indices = np.random.choice(np.arange(500), size=40, replace=False)

# Add i.i.d. random noise (mean 0, variance 0.16) to the 40 selected
points
noise = np.random.normal(0, np.sqrt(0.16), size=40)
x_train = x[indices]  # Select 40 x points
y_train_noisy = y_true[indices] + noise  # Add noise to the
corresponding y_true points

# Plot the results
plt.figure(figsize=(10, 6))

# Plot the true cos(x) function
plt.plot(x, y_true, label=r"$y = \cos(x)$", color='blue')

# Plot the noisy training data points
plt.scatter(x_train, y_train_noisy, label="Noisy Training Data",
color='red', zorder=5)

# Set labels, title, and legend
plt.title("Noisy Training Data vs. True Function")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```


Noisy Training Data vs. True Function
$y = \cos(x)$
Noisy Training Data
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
0
10
20
30
40
50
x

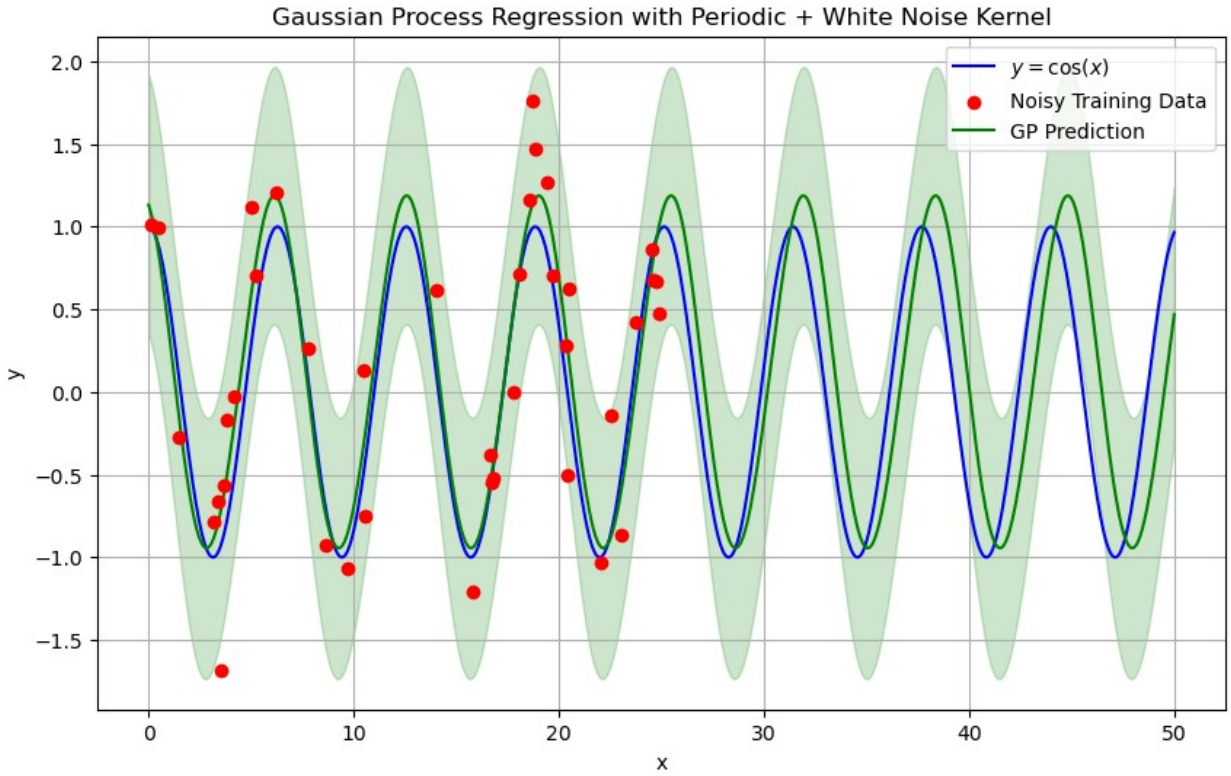
```
# Import Dependencies
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel,
ExpSineSquared, ConstantKernel
from sklearn.kernel_ridge import KernelRidge
```

Midterm 1 Project, Problem 2-2

```
# Part 1: Generate noisy data as done previously
x = np.linspace(0, 50, 1000)  # Generate 1000 evenly spaced x data
points
y_true = np.cos(x)  # True function y = cos(x)

# Randomly select 40 points from the first 500 data points
np.random.seed(42)  # Ensure reproducibility
indices = np.random.choice(np.arange(500), size=40, replace=False)

# Add i.i.d. random noise (mean 0, variance 0.16) to the 40 selected
points
noise = np.random.normal(0, np.sqrt(0.16), size=40)
x_train = x[indices].reshape(-1, 1)  # Select 40 x points from the
first 500
y_train_noisy = y_true[indices] + noise  # Add noise to the
corresponding y points

# Part 2: Fit a Gaussian Process (GP) model using a periodic kernel
and a white noise kernel
# Define the kernel: periodic kernel (ExpSineSquared) + white noise
(WhiteKernel)
kernel = ExpSineSquared(length_scale=1.0, periodicity=1.0) +
WhiteKernel(noise_level=1.0)

# Create the Gaussian Process Regressor with the specified kernel
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10)

# Fit the GP to the noisy data
gp.fit(x_train, y_train_noisy)

# Predict using the GP on the full x range
x_pred = x.reshape(-1, 1)
y_pred, y_std = gp.predict(x_pred, return_std=True)

# Extract the learned kernel hyperparameters
kernel_optimized = gp.kernel_
print("Optimized Kernel:", kernel_optimized)

# Extract the specific hyperparameters from the optimized kernel
p = kernel_optimized.k1.periodicity  # Periodicity (p)
```

```
ell = kernel_optimized.k1.length_scale  # Length scale (ℓ)
sigma = np.sqrt(kernel_optimized.k2.noise_level)  # Noise level (σ)

# Report the hyperparameters
print(f"Optimized hyperparameters:")
print(f"Periodicity (p): {p}")
print(f"Length Scale (ℓ): {ell}")
print(f"Noise Level (σ): {sigma}")

# Plot the results
plt.figure(figsize=(10, 6))

# Plot the true function y = cos(x)
plt.plot(x, y_true, label=r"$y = \cos(x)$", color='blue')

# Plot the noisy training data points
plt.scatter(x_train, y_train_noisy, label="Noisy Training Data",
color='red', zorder=5)

# Plot the GP predictions with uncertainty bounds
plt.plot(x_pred, y_pred, label="GP Prediction", color='green')
plt.fill_between(x_pred.flatten(), y_pred - 1.96 * y_std, y_pred +
1.96 * y_std, alpha=0.2, color='green')

# Set labels, title, and legend
plt.title("Gaussian Process Regression with Periodic + White Noise
Kernel")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```

```
Optimized Kernel: ExpSineSquared(length_scale=1.84, periodicity=6.45)
+ WhiteKernel(noise_level=0.14)
Optimized hyperparameters:
Periodicity (p): 6.447335368768318
Length Scale (ℓ): 1.8431891941054277
Noise Level (σ): 0.37480938853048
```


Gaussian Process Regression with Periodic + White Noise Kernel
$y = \cos(x)$
Noisy Training Data
GP Prediction
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
0
10
20
30
40
50
x

```
# Import Dependencies
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel,
ExpSineSquared, ConstantKernel
from sklearn.kernel_ridge import KernelRidge
```

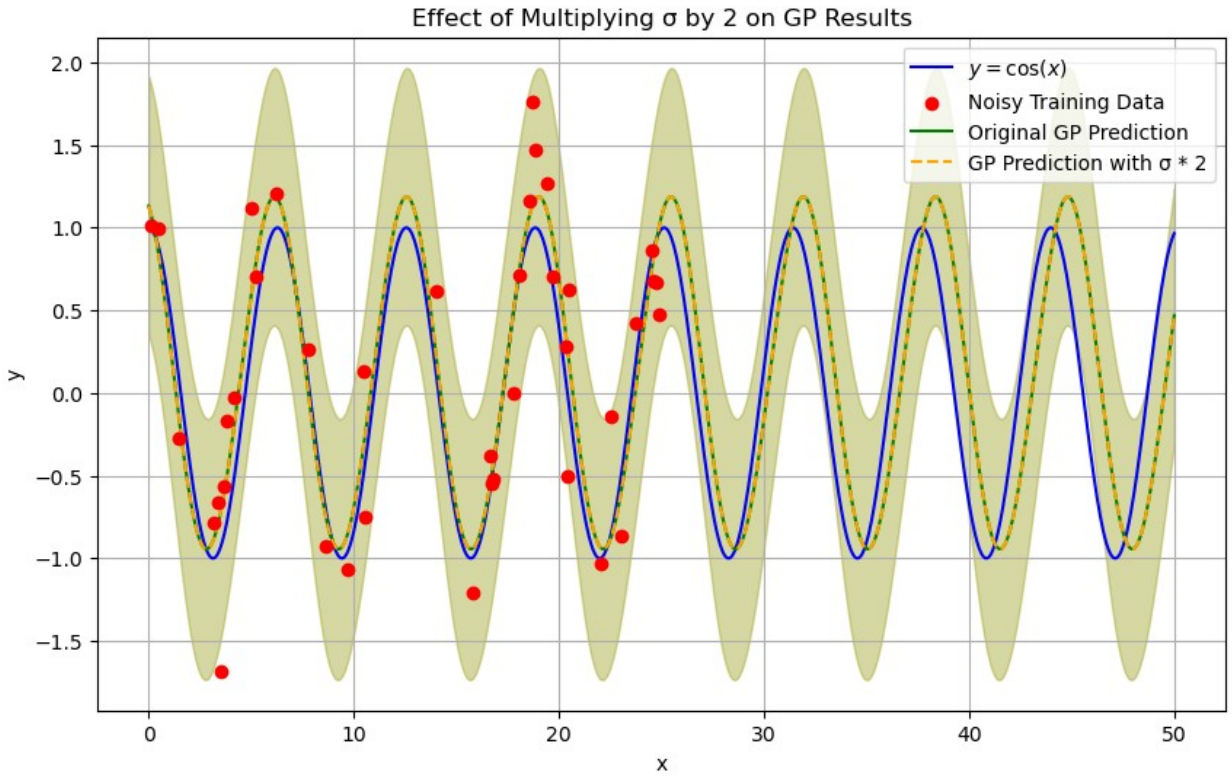
Midterm 1 Project, Problem 2-3

```
# Part 1: Generate noisy data as done previously
x = np.linspace(0, 50, 1000)  # Generate 1000 evenly spaced x data
points
y_true = np.cos(x)  # True function y = cos(x)

# Randomly select 40 points from the first 500 data points
np.random.seed(42)  # Ensure reproducibility
indices = np.random.choice(np.arange(500), size=40, replace=False)

# Add i.i.d. random noise (mean 0, variance 0.16) to the 40 selected
points
noise = np.random.normal(0, np.sqrt(0.16), size=40)
x_train = x[indices].reshape(-1, 1)  # Select 40 x points from the
first 500
y_train_noisy = y_true[indices] + noise  # Add noise to the
corresponding y points

# Part 2: Fit a Gaussian Process (GP) model using a periodic kernel
and a white noise kernel
# Define the kernel: periodic kernel (ExpSineSquared) + white noise
(WhiteKernel)
kernel = ExpSineSquared(length_scale=1.0, periodicity=1.0) +
WhiteKernel(noise_level=1.0)

# Create the Gaussian Process Regressor with the specified kernel
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10)

# Fit the GP to the noisy data
gp.fit(x_train, y_train_noisy)

# Predict using the GP on the full x range
x_pred = x.reshape(-1, 1)
y_pred, y_std = gp.predict(x_pred, return_std=True)

# Extract the learned kernel hyperparameters
kernel_optimized = gp.kernel_
print("Optimized Kernel:", kernel_optimized)

# Extract the specific hyperparameters from the optimized kernel
p = kernel_optimized.k1.periodicity  # Periodicity (p)
```

```
ell = kernel_optimized.k1.length_scale  # Length scale (ℓ)
sigma = np.sqrt(kernel_optimized.k2.noise_level)  # Noise level (σ)

# Report the hyperparameters
print(f"Optimized hyperparameters:")
print(f"Periodicity (p): {p}")
print(f"Length Scale (ℓ): {ell}")
print(f"Noise Level (σ): {sigma}")

# Part 3: Multiply the optimized σ by 2
new_sigma = 2 * sigma
print(f"New Noise Level (σ * 2): {new_sigma}")

# Update the kernel with the new noise level (σ * 2)
new_kernel = ExpSineSquared(length_scale=ell, periodicity=p) +
WhiteKernel(noise_level=new_sigma**2)

# Re-fit the GP with the modified kernel
gp_new = GaussianProcessRegressor(kernel=new_kernel,
n_restarts_optimizer=10)
gp_new.fit(x_train, y_train_noisy)

# Predict using the GP with modified noise level
y_pred_new, y_std_new = gp_new.predict(x_pred, return_std=True)

# Plot the results with the original GP and new GP after multiplying σ
by 2
plt.figure(figsize=(10, 6))

# Plot the true function y = cos(x)
plt.plot(x, y_true, label=r"$y = \cos(x)$", color='blue')

# Plot the noisy training data points
plt.scatter(x_train, y_train_noisy, label="Noisy Training Data",
color='red', zorder=5)

# Plot the original GP predictions with uncertainty bounds
plt.plot(x_pred, y_pred, label="Original GP Prediction",
color='green')
plt.fill_between(x_pred.flatten(), y_pred - 1.96 * y_std, y_pred +
1.96 * y_std, alpha=0.2, color='green')

# Plot the new GP predictions with increased noise level
plt.plot(x_pred, y_pred_new, label="GP Prediction with σ * 2",
color='orange', linestyle='--')
plt.fill_between(x_pred.flatten(), y_pred_new - 1.96 * y_std_new,
y_pred_new + 1.96 * y_std_new, alpha=0.2, color='orange')

# Set labels, title, and legend
plt.title("Effect of Multiplying σ by 2 on GP Results")
```

```
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```

```
Optimized Kernel: ExpSineSquared(length_scale=1.84, periodicity=6.45)
+ WhiteKernel(noise_level=0.14)
Optimized hyperparameters:
Periodicity (p): 6.447335368768318
Length Scale (ℓ): 1.8431891941054277
Noise Level (σ): 0.37480938853048
New Noise Level (σ * 2): 0.74961877706096
```



When σ is increased, the model assumes that the noise level in the data is higher, resulting in a broader prediction uncertainty range (confidence interval). In the plot, this effect is visible as a difference between the green region (original model) and the orange region (modified model). However, this change does not significantly affect the mean prediction. This is because the Gaussian Process (GP) mean function remains dominated by the observed data, with the increased noise level mainly affecting how confident the model is about its predictions rather than changing the prediction itself.

```
# Import Dependencies
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel,
ExpSineSquared, ConstantKernel
from sklearn.kernel_ridge import KernelRidge
```

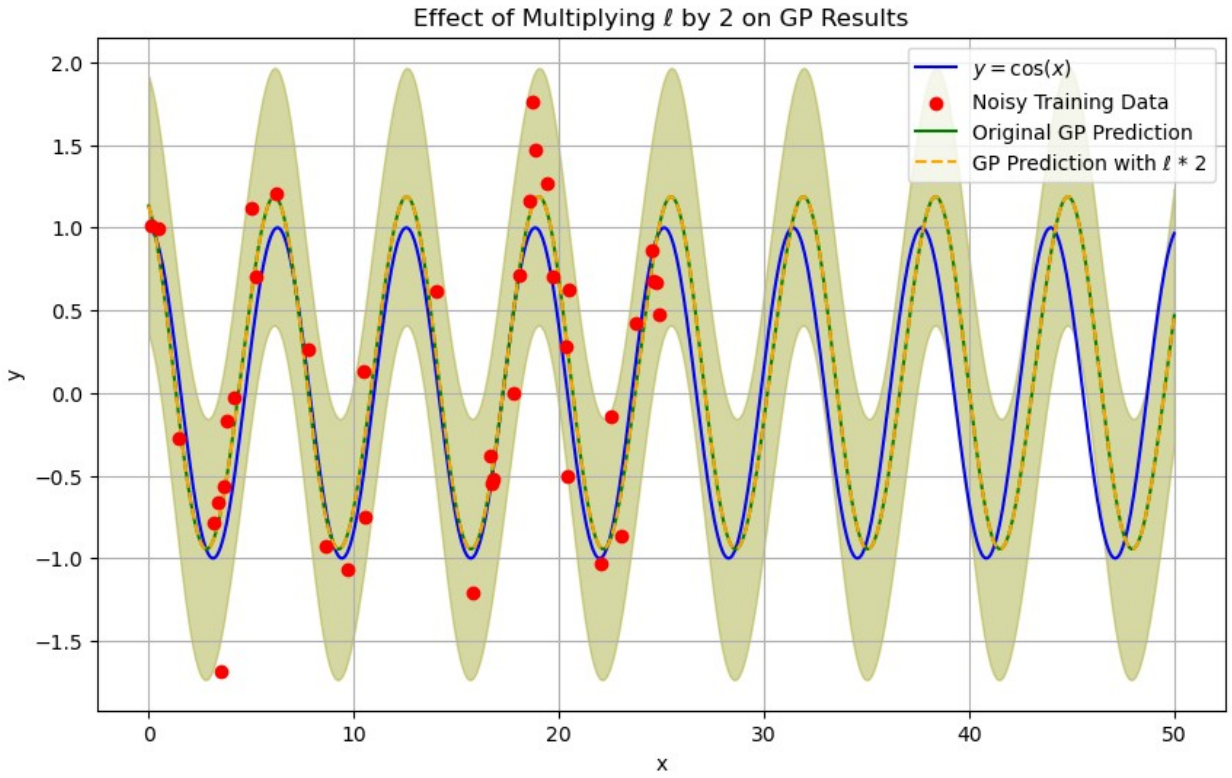
Midterm 1 Project, Problem 2-4

```
# Part 1: Generate noisy data as done previously
x = np.linspace(0, 50, 1000)  # Generate 1000 evenly spaced x data
points
y_true = np.cos(x)  # True function y = cos(x)

# Randomly select 40 points from the first 500 data points
np.random.seed(42)  # Ensure reproducibility
indices = np.random.choice(np.arange(500), size=40, replace=False)

# Add i.i.d. random noise (mean 0, variance 0.16) to the 40 selected
points
noise = np.random.normal(0, np.sqrt(0.16), size=40)
x_train = x[indices].reshape(-1, 1)  # Select 40 x points from the
first 500
y_train_noisy = y_true[indices] + noise  # Add noise to the
corresponding y points

# Part 2: Fit a Gaussian Process (GP) model using a periodic kernel
and a white noise kernel
# Define the kernel: periodic kernel (ExpSineSquared) + white noise
(WhiteKernel)
kernel = ExpSineSquared(length_scale=1.0, periodicity=1.0) +
WhiteKernel(noise_level=1.0)

# Create the Gaussian Process Regressor with the specified kernel
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10)

# Fit the GP to the noisy data
gp.fit(x_train, y_train_noisy)

# Predict using the GP on the full x range
x_pred = x.reshape(-1, 1)
y_pred, y_std = gp.predict(x_pred, return_std=True)

# Extract the learned kernel hyperparameters
kernel_optimized = gp.kernel_
print("Optimized Kernel:", kernel_optimized)

# Extract the optimized length scale (ℓ)
ell = kernel_optimized.k1.length_scale
```

```
print(f"Original Length Scale (ℓ): {ell}")

# Multiply the length scale by 2
new_ell = 2 * ell
print(f"New Length Scale (ℓ * 2): {new_ell}")

# Update the kernel with the new length scale (ℓ * 2)
new_kernel = ExpSineSquared(length_scale=new_ell,

periodicity=kernel_optimized.k1.periodicity) +
WhiteKernel(noise_level=kernel_optimized.k2.noise_level)

# Re-fit the GP with the modified kernel
gp_new = GaussianProcessRegressor(kernel=new_kernel,
n_restarts_optimizer=10)
gp_new.fit(x_train, y_train_noisy)

# Predict using the GP with modified length scale
y_pred_new, y_std_new = gp_new.predict(x_pred, return_std=True)

# Plot the results with the original GP and new GP after multiplying ℓ
by 2
plt.figure(figsize=(10, 6))

# Plot the true function y = cos(x)
plt.plot(x, y_true, label=r"$y = \cos(x)$", color='blue')

# Plot the noisy training data points
plt.scatter(x_train, y_train_noisy, label="Noisy Training Data",
color='red', zorder=5)

# Plot the original GP predictions with uncertainty bounds
plt.plot(x_pred, y_pred, label="Original GP Prediction",
color='green')
plt.fill_between(x_pred.flatten(), y_pred - 1.96 * y_std, y_pred +
1.96 * y_std, alpha=0.2, color='green')

# Plot the new GP predictions with increased length scale
plt.plot(x_pred, y_pred_new, label="GP Prediction with ℓ * 2",
color='orange', linestyle='--')
plt.fill_between(x_pred.flatten(), y_pred_new - 1.96 * y_std_new,
y_pred_new + 1.96 * y_std_new, alpha=0.2, color='orange')

# Set labels, title, and legend
plt.title("Effect of Multiplying ℓ by 2 on GP Results")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)
```

```
# Show the plot
plt.show()
```

```
Optimized Kernel: ExpSineSquared(length_scale=1.84, periodicity=6.45)
+ WhiteKernel(noise_level=0.14)
Original Length Scale (ℓ): 1.8431891941054277
New Length Scale (ℓ * 2): 3.6863783882108554
```



Increasing ℓ makes the model less sensitive to variations in the input variable, leading to a smoother prediction curve. This also causes the uncertainty range to widen, as the model assumes that changes in the input have less influence on the output over larger distances. Since the data has a strong periodic signal (cos(x)), increasing ℓ doesn't significantly alter the mean prediction. This is why the two curves (green and orange) appear similar over most of the range.

```
# Import Dependencies
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel,
ExpSineSquared, ConstantKernel
from sklearn.kernel_ridge import KernelRidge
```

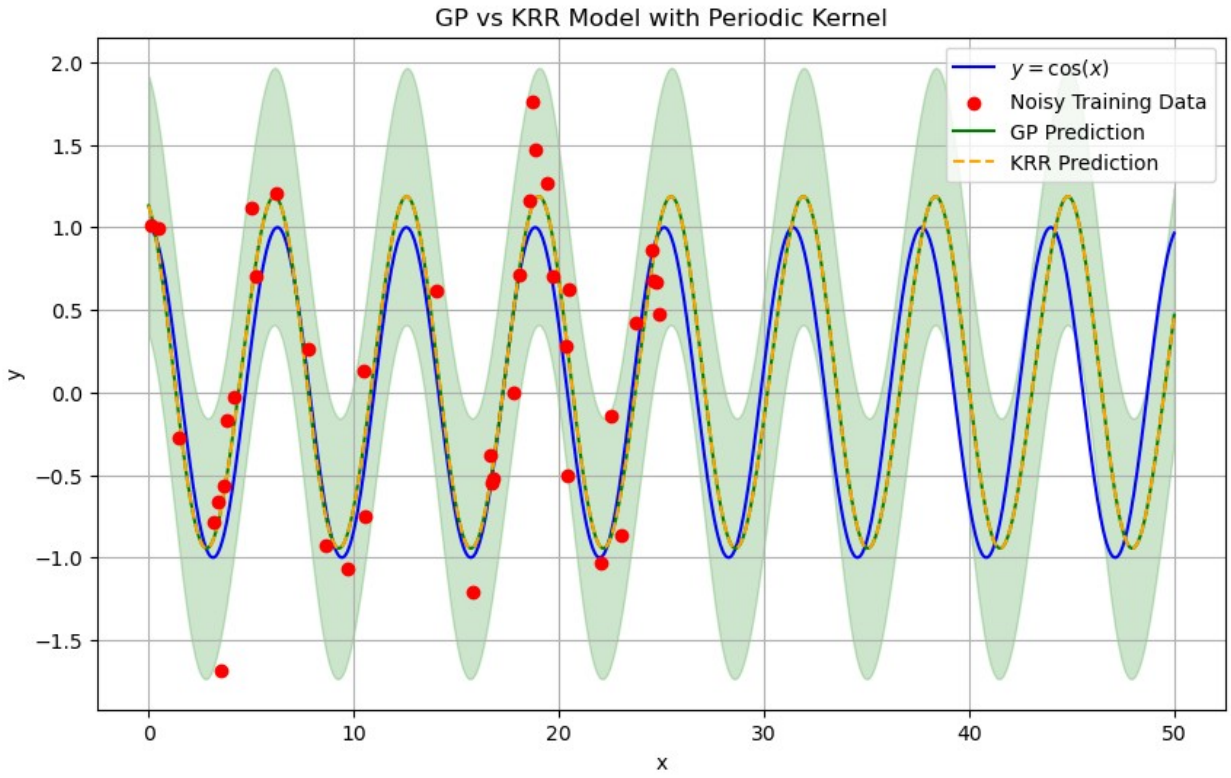
Midterm 1 Project, Problem 2-5

```
# Part 1: Generate noisy data as done previously
x = np.linspace(0, 50, 1000)  # Generate 1000 evenly spaced x data
points
y_true = np.cos(x)  # True function y = cos(x)

# Randomly select 40 points from the first 500 data points
np.random.seed(42)  # Ensure reproducibility
indices = np.random.choice(np.arange(500), size=40, replace=False)

# Add i.i.d. random noise (mean 0, variance 0.16) to the 40 selected
points
noise = np.random.normal(0, np.sqrt(0.16), size=40)
x_train = x[indices].reshape(-1, 1)  # Select 40 x points from the
first 500
y_train_noisy = y_true[indices] + noise  # Add noise to the
corresponding y points

# Part 2: Fit a Gaussian Process (GP) model using a periodic kernel
and a white noise kernel
# Define the kernel: periodic kernel (ExpSineSquared) + white noise
(WhiteKernel)
kernel = ExpSineSquared(length_scale=1.0, periodicity=1.0) +
WhiteKernel(noise_level=1.0)

# Create the Gaussian Process Regressor with the specified kernel
gp = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=10)

# Fit the GP to the noisy data
gp.fit(x_train, y_train_noisy)

# Predict using the GP on the full x range
x_pred = x.reshape(-1, 1)
y_pred_gp, y_std_gp = gp.predict(x_pred, return_std=True)

# Extract the learned kernel hyperparameters from the GP model
kernel_optimized = gp.kernel_
print("Optimized Kernel from GP:", kernel_optimized)

# Extract the optimized periodic kernel hyperparameters (from GP
model)
```

```
periodicity = kernel_optimized.k1.periodicity
length_scale = kernel_optimized.k1.length_scale

# Part 3: Kernel Ridge Regression (KRR)
# Use the same periodic kernel hyperparameters as in the GP model
krr_kernel = ExpSineSquared(length_scale=length_scale,
periodicity=periodicity)

# Choose the penalty coefficient (regularization parameter)
# Use a value similar to the noise variance in GP model to match the
GP's posterior mean
penalty_coefficient = kernel_optimized.k2.noise_level
print(f"Penalty Coefficient for KRR (λ): {penalty_coefficient}")

# Fit the KRR model
krr = KernelRidge(kernel=krr_kernel, alpha=penalty_coefficient)
krr.fit(x_train, y_train_noisy)

# Predict using the KRR model
y_pred_krr = krr.predict(x_pred)

# Plot the results with the GP and KRR
plt.figure(figsize=(10, 6))

# Plot the true function y = cos(x)
plt.plot(x, y_true, label=r"$y = \cos(x)$", color='blue')

# Plot the noisy training data points
plt.scatter(x_train, y_train_noisy, label="Noisy Training Data",
color='red', zorder=5)

# Plot the GP predictions with uncertainty bounds
plt.plot(x_pred, y_pred_gp, label="GP Prediction", color='green')
plt.fill_between(x_pred.flatten(), y_pred_gp - 1.96 * y_std_gp,
y_pred_gp + 1.96 * y_std_gp, alpha=0.2, color='green')

# Plot the KRR predictions
plt.plot(x_pred, y_pred_krr, label="KRR Prediction", color='orange',
linestyle='--')

# Optionally, you can fill KRR area based on GP's uncertainty for
visualization purposes (but this is still GP uncertainty)
#plt.fill_between(x_pred.flatten(), y_pred_krr - 1.96 * y_std_gp,
y_pred_krr + 1.96 * y_std_gp, alpha=0.2, color='orange')

# Set labels, title, and legend
plt.title("GP vs KRR Model with Periodic Kernel")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
```

```
plt.grid(True)

# Show the plot
plt.show()
```

```
Optimized Kernel from GP: ExpSineSquared(length_scale=1.84,
periodicity=6.45) + WhiteKernel(noise_level=0.14)
Penalty Coefficient for KRR (λ): 0.14048207773059232
```



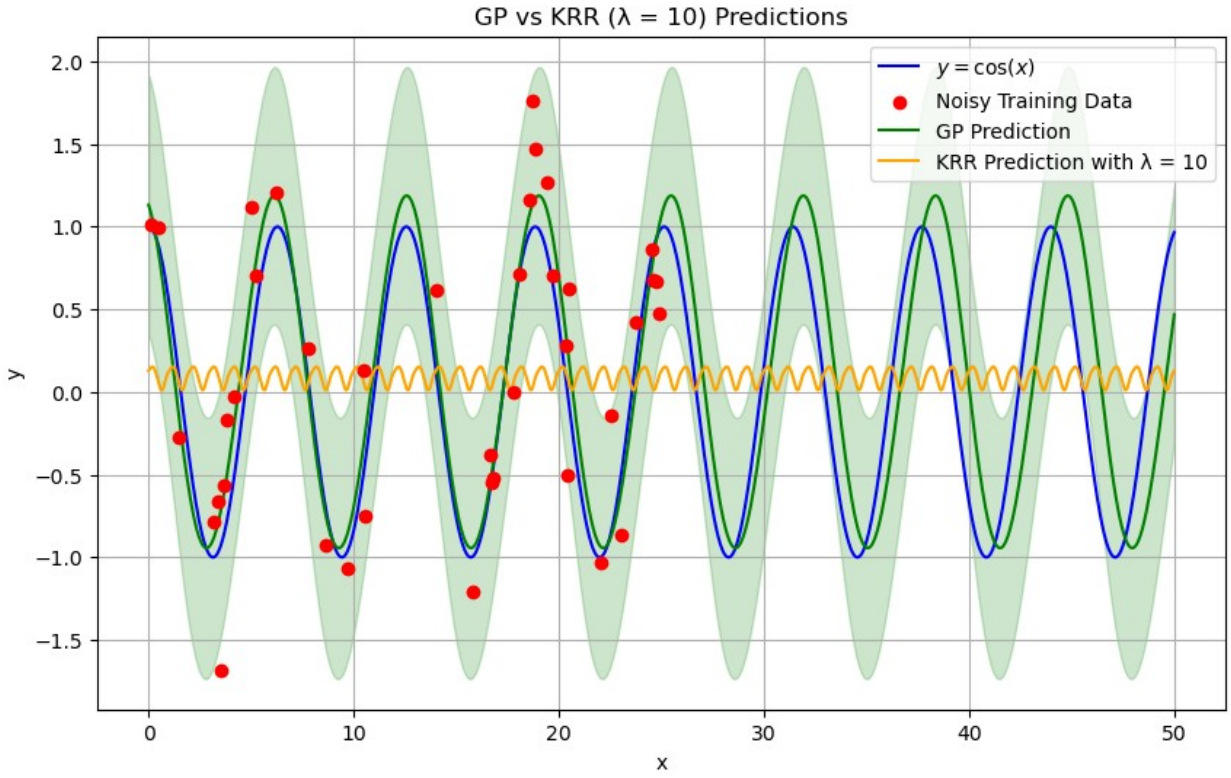
```
# Import Dependencies
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import RBF, WhiteKernel,
ExpSineSquared, ConstantKernel
from sklearn.kernel_ridge import KernelRidge
```

Midterm 1 Project, Problem 2-6

```
# Part 1: Generate noisy data as done previously
x = np.linspace(0, 50, 1000)  # Generate 1000 evenly spaced x data
points
y_true = np.cos(x)  # True function y = cos(x)

# Randomly select 40 points from the first 500 data points
np.random.seed(42)  # Ensure reproducibility
indices = np.random.choice(np.arange(500), size=40, replace=False)

# Add i.i.d. random noise (mean 0, variance 0.16) to the 40 selected
points
noise = np.random.normal(0, np.sqrt(0.16), size=40)
x_train = x[indices].reshape(-1, 1)  # Select 40 x points from the
first 500
y_train_noisy = y_true[indices] + noise  # Add noise to the
corresponding y points

# Part 2: Fit a Gaussian Process (GP) model using a periodic kernel
# Define the kernel: periodic kernel (ExpSineSquared) + white noise
(WhiteKernel)
kernel_gp = ExpSineSquared(length_scale=1.0, periodicity=1.0) +
WhiteKernel(noise_level=1.0)

# Create the Gaussian Process Regressor with the specified kernel
gp = GaussianProcessRegressor(kernel=kernel_gp,
n_restarts_optimizer=10)

# Fit the GP to the noisy data
gp.fit(x_train, y_train_noisy)

# Predict using the GP on the full x range
x_pred = x.reshape(-1, 1)
y_pred_gp, y_std_gp = gp.predict(x_pred, return_std=True)

# Part 3: Use the same periodic kernel for KRR and set λ to 10
krr_kernel = ExpSineSquared(length_scale=1.0, periodicity=1.0)

# Set penalty coefficient λ to 10
penalty_coefficient = 10
print(f"Penalty Coefficient for KRR (λ): {penalty_coefficient}")
```

```
# Fit the KRR model with λ = 10
krr = KernelRidge(kernel=krr_kernel, alpha=penalty_coefficient)
krr.fit(x_train, y_train_noisy)

# Predict using the KRR model
y_pred_krr = krr.predict(x_pred)

# Plot the results with GP, KRR λ = 10, and the true function
plt.figure(figsize=(10, 6))

# Plot the true function y = cos(x)
plt.plot(x, y_true, label=r"$y = \cos(x)$", color='blue')

# Plot the noisy training data points
plt.scatter(x_train, y_train_noisy, label="Noisy Training Data",
color='red', zorder=5)

# Plot the GP predictions with uncertainty bounds
plt.plot(x_pred, y_pred_gp, label="GP Prediction", color='green')
plt.fill_between(x_pred.flatten(), y_pred_gp - 1.96 * y_std_gp,
y_pred_gp + 1.96 * y_std_gp, alpha=0.2, color='green')

# Plot the KRR predictions with λ = 10
plt.plot(x_pred, y_pred_krr, label="KRR Prediction with λ = 10",
color='orange')

# Set labels, title, and legend
plt.title("GP vs KRR (λ = 10) Predictions")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```

```
Penalty Coefficient for KRR (λ): 10
```


GP vs KRR (λ = 10) Predictions
$y = \cos(x)$
Noisy Training Data
GP Prediction
KRR Prediction with λ = 10
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
0
10
20
30
40
50
x