Homework 4

SHREYA CHETAN PAWASKAR
pawaskas@uci.edu
12041645

Question 1:

Code

```
!pip install bayesian-optimization
```

```
import numpy as np
import matplotlib.pyplot as plt
import random
from functools import partial
from bayes_opt import BayesianOptimization
from bayes_opt import UtilityFunction
```

```
from sklearn.gaussian_process.kernels import Matern, RBF
from sklearn.gaussian_process import GaussianProcessRegressor
from bayesian_optimization_utils import *
```

```
num_bo_steps = 50
num_diff_initializations = 5
plotter = Plotter()
```

```
# Part A: Bayesian Optimization with UCB acquisition function
ucb_hyperparameters = [1.0, 5.0, 10.0]
print(f"Part A: Activation Function UCB: ")
for kappa in ucb_hyperparameters:
   print(f"\nKappa: {kappa:0.3f}")
   func_max_values = np.zeros((num_runs_per_experiment, num_bo_steps))
   for init_iter in range(num_runs_per_experiment):
       print(f"Run #{init_iter + 1} of {num_runs_per_experiment}")
       # Create the BO optimizer
       bayesian_optimizer = BayesianOptimization(negative_modified_branin_func,
get_negative_modified_branin_func_bounds(), verbose=0)
       # Create the acquisition function for this problem
       acquisition_function = UtilityFunction(kind="ucb", kappa=kappa)
       # Initialize with 2 random points
       bayesian_optimizer.maximize(init_points=2, n_iter=0)
```

```
        for bo_iter in range(num_bo_steps):
            # Do 1 step Bayesian optimization
            bayesian_optimizer.maximize(init_points=0, n_iter=1,
acquisition_function=acquisition_function)
            # Grab the current max after the iteration
            best_x1 = bayesian_optimizer.max["params"]["x1"]
            best_x2 = bayesian_optimizer.max["params"]["x2"]
            func_max_values[init_iter, bo_iter] =
negative_modified_branin_func(best_x1, best_x2)
    label = f"UCB with Kappa {kappa:0.3f}"
    plotter.add_experiment_plot(func_max_values, label)

fig1 = plotter.create_figure(1)
plt.show()
fig1 = None
```

```
# Part B: Bayesian Optimization with POI acquisition function
poi_hyperparameters = [0.01, 0.1, 1.0]
for xi in poi_hyperparameters:
    print(f"\nXi: {xi:0.3f}")
    func_max_values = np.zeros((num_runs_per_experiment, num_bo_steps))
    for init_iter in range(num_runs_per_experiment):
        print(f"Run #{init_iter + 1} of {num_runs_per_experiment}")
        bayesian_optimizer = BayesianOptimization(negative_modified_branin_func,
get_negative_modified_branin_func_bounds(), verbose=0)
        # Create the acquisition function for this problem
        acquisition_function = UtilityFunction(kind="poi", xi=xi)
        # Initialize with 2 random points
        bayesian_optimizer.maximize(init_points=2, n_iter=0)
        for bo_iter in range(num_bo_steps):
            # Do 1 step Bayesian optimization
            bayesian_optimizer.maximize(init_points=0, n_iter=1,
acquisition_function=acquisition_function)
            # Grab the current max after the iteration
            best_x1 = bayesian_optimizer.max["params"]["x1"]
            best_x2 = bayesian_optimizer.max["params"]["x2"]
            func_max_values[init_iter, bo_iter] =
negative_modified_branin_func(best_x1, best_x2)
    label = f"POI with Xi {xi:0.3f}"
    plotter.add_experiment_plot(func_max_values, label)
```

```
fig1 = plotter.create_figure(1)
plt.show()
```

```
# Part C: Bayesian Optimization with EI acquisition function
ei_hyperparameters = [0.01, 0.1, 1.0]
print(f"Part C: Activation Function EI: ")
for xi in ei_hyperparameters:
   print(f"\nXi: {xi:0.3f}")
   func_max_values = np.zeros((num_runs_per_experiment, num_bo_steps))
   for init_iter in range(num_runs_per_experiment):
       print(f"Run #{init_iter + 1} of {num_runs_per_experiment}")
       # Create the BO optimizer
       bayesian_optimizer = BayesianOptimization(negative_modified_branin_func,
get_negative_modified_branin_func_bounds(), verbose=0)
       # Create the acquisition function for this problem
       acquisition_function = UtilityFunction(kind="ei", xi=xi)
       # Initialize with 2 random points
       bayesian_optimizer.maximize(init_points=2, n_iter=0)
       # Do many iterations 1 at a time so we can record the current best guess for
the max
       for bo_iter in range(num_bo_steps):
           # Do 1 step Bayesian optimization
           bayesian_optimizer.maximize(init_points=0, n_iter=1,
acquisition_function=acquisition_function)
           # Grab the current max after the iteration
           best_x1 = bayesian_optimizer.max["params"]["x1"]
           best_x2 = bayesian_optimizer.max["params"]["x2"]
           func_max_values[init_iter, bo_iter] =
negative_modified_branin_func(best_x1, best_x2)
   label = f"EI with Xi {xi:0.3f}"
   plotter.add_experiment_plot(func_max_values, label)

fig1 = plotter.create_figure(1)
```

```
# Part D: Bayesian Optimization with Matern kernel function
matern_hyperparameters = [0.5, 2.5, 10.0]
print(f"Part D: Activation Function EI with Matern Kernel: ")
for nu in matern_hyperparameters:
   print(f"\nNu: {nu:0.3f}")
   func_max_values = np.zeros((num_runs_per_experiment, num_bo_steps))
   for init_iter in range(num_runs_per_experiment):
       print(f"Run #{init_iter + 1} of {num_runs_per_experiment}")
```

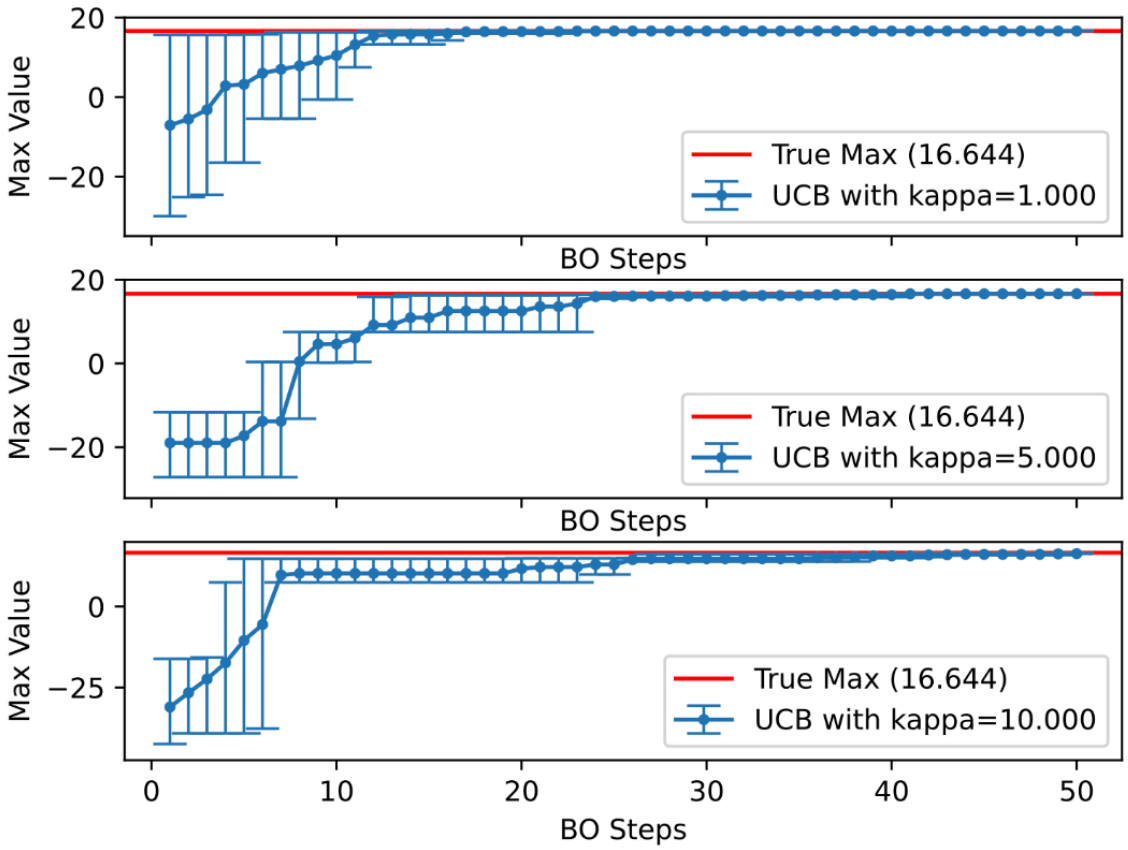
```
        # Create the BO optimizer
        bayesian_optimizer = BayesianOptimization(negative_modified_branin_func,
get_negative_modified_branin_func_bounds(), verbose=0)
        # Change the kernel to Matern
        kernel = Matern(nu=nu)
        bayesian_optimizer._gp = GaussianProcessRegressor(kernel=kernel, alpha=1e-6,
normalize_y=True, n_restarts_optimizer=5, random_state=None)
        # Create the acquisition function for this problem
        acquisition_function = UtilityFunction(kind="ei", xi=0.01)
        # Initialize with 2 random points
        bayesian_optimizer.maximize(init_points=2, n_iter=0)
        # Do many iterations 1 at a time so we can record the current best guess for
the max
        for bo_iter in range(num_bo_steps):
            # Do 1 step Bayesian optimization
            bayesian_optimizer.maximize(init_points=0, n_iter=1,
acquisition_function=acquisition_function)
            # Grab the current max after the iteration
            best_x1 = bayesian_optimizer.max["params"]["x1"]
            best_x2 = bayesian_optimizer.max["params"]["x2"]
            func_max_values[init_iter, bo_iter] =
negative_modified_branin_func(best_x1, best_x2)
    label = f"Matern Kernel with nu: {nu:0.3f}"
    plotter.add_experiment_plot(func_max_values, label)

fig1 = plotter.create_figure(1)
plt.show()
```

```
# Part E: Bayesian Optimization with RBF kernel function
rbf_hyperparameters = [0.001, 1.0, 3.0]
print(f"Part E: Activation Function EI with RBF Kernel: ")
for sigma in rbf_hyperparameters:
    print(f"\nSigma: {sigma:0.3f}")
    func_max_values = np.zeros((num_runs_per_experiment, num_bo_steps))
    for init_iter in range(num_runs_per_experiment):
        print(f"Run #{init_iter + 1} of {num_runs_per_experiment}")
        # Create the BO optimizer
        bayesian_optimizer = BayesianOptimization(negative_modified_branin_func,
get_negative_modified_branin_func_bounds(), verbose=0)
        # Change the kernel to RBF
        kernel = RBF(length_scale=sigma)
```

```
        bayesian_optimizer._gp = GaussianProcessRegressor(kernel=kernel, alpha=1e-6,
normalize_y=True, n_restarts_optimizer=5, random_state=None)
        # Create the acquisition function for this problem
        acquisition_function = UtilityFunction(kind="ei", xi=0.01)
        # Initialize with 2 random points
        bayesian_optimizer.maximize(init_points=2, n_iter=0)
        for bo_iter in range(num_bo_steps):
            # Do 1 step Bayesian optimization
            bayesian_optimizer.maximize(init_points=0, n_iter=1,
acquisition_function=acquisition_function)
            # Grab the current max after the iteration
            best_x1 = bayesian_optimizer.max["params"]["x1"]
            best_x2 = bayesian_optimizer.max["params"]["x2"]
            func_max_values[init_iter, bo_iter] =
negative_modified_branin_func(best_x1, best_x2)
    label = f"RBF Kernel with sigma: {sigma:0.3f}"
    plotter.add_experiment_plot(func_max_values, label)

fig1 = plotter.create_figure(1)
plt.show()
```

a)

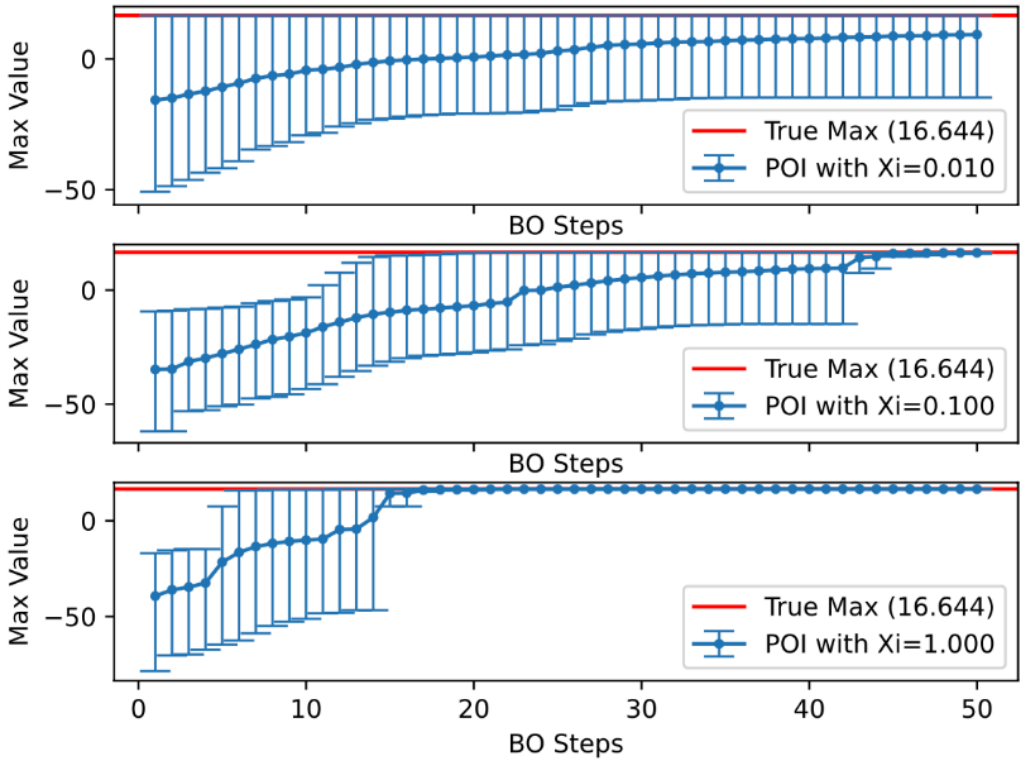
- Higher values of κ in UCB encourage exploration of uncertain regions of the function, which is particularly beneficial in the early stages of optimization when little information about the true function is available.
- This emphasis on exploration allows for a better understanding of the function's behavior rather than solely focusing on refining the estimate of the maximum.
- In the figure, UCB with the largest κ value demonstrates rapid and consistent improvements early in the optimization process, showcasing its effectiveness in exploring the function and gaining insights.
- In later stages of optimization, exploitation becomes more favorable as the focus shifts towards refining the estimate of the maximum value.
- In the top plot, UCB with the smallest κ value is slower in exploring the function but quicker in refining the maximum value once the relevant region is identified.
- A higher κ value helps you explore more in the beginning when you're still learning.
- A smaller κ value helps you refine your search once you have a better idea of where to look.



---

b)

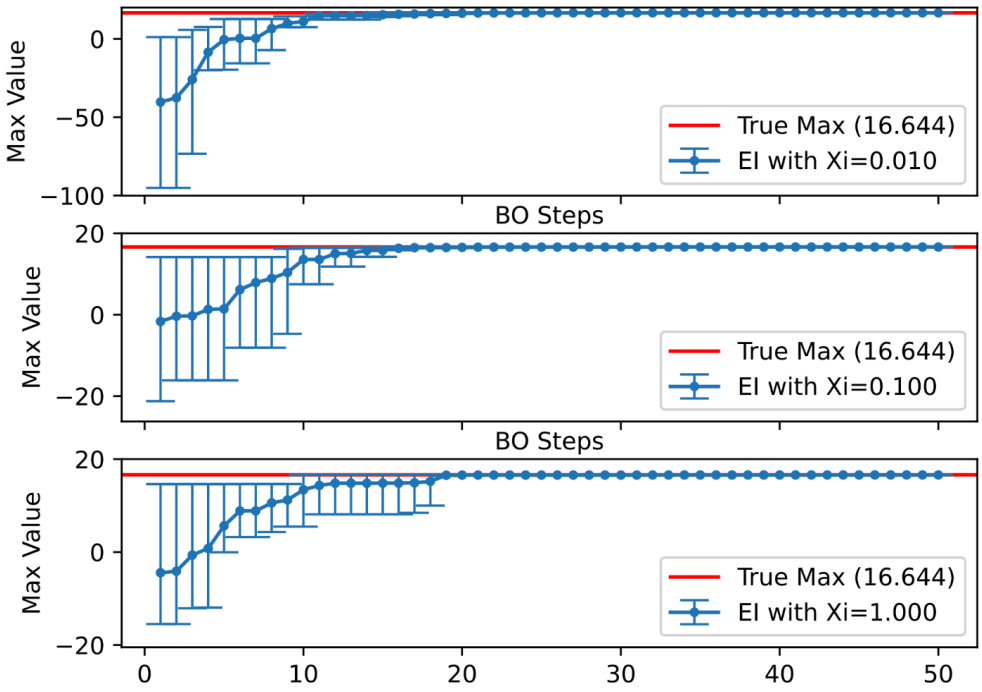
- Like answer 1, the argument for exploration being crucial early and late in optimization applies.
- Effect of ξ on Exploration:
  - Larger values of ξ promote more exploration.
  - Smaller values result in minimal exploration.
- Impact on Convergence:
  - For small ξ values, indicating limited exploration, the optimization process is slow to reach the true maximum.
  - It spends excessive time refining an inaccurate estimate of the maximum rather than exploring potentially better regions.
  - This leads to being 'stuck' in refining a maximum that is far from the true global maximum.



---

c)

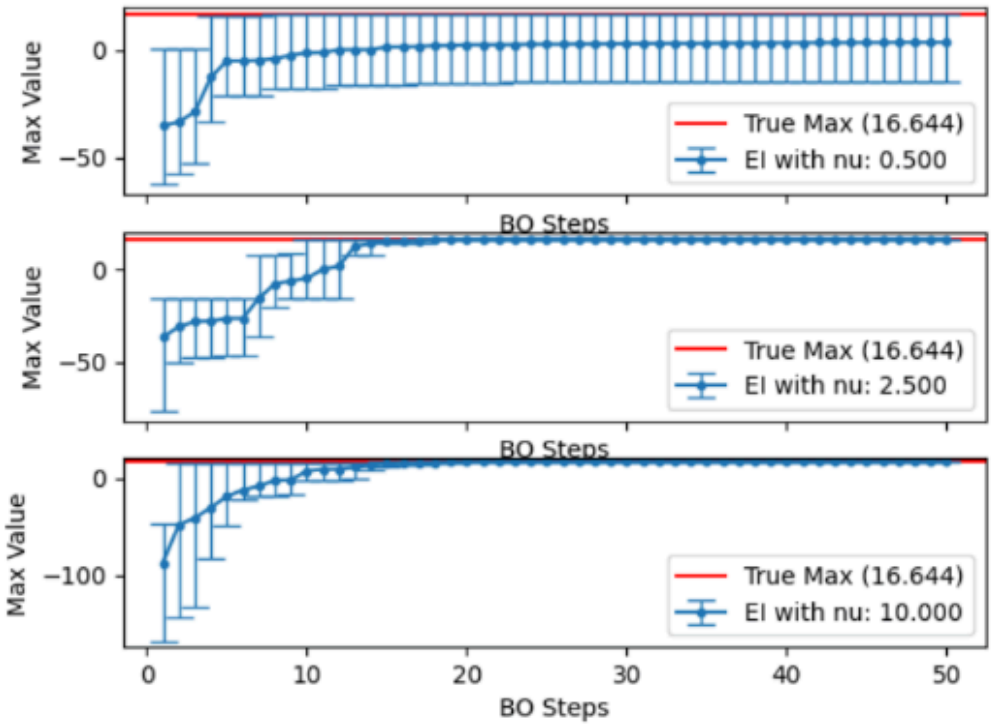
- Effect of ξ on Exploration in Expected Improvement (EI):
  - Larger ξ values lead to more exploration, similar to Probability of Improvement (POI).
  - However, excessive exploration can slow down optimization, as seen in part (a).
- Advantages of Expected Improvement (EI):
  - EI is generally more effective and less sensitive to hyperparameters than POI.
  - This is because EI considers the magnitude of expected improvement for all ξ values, resulting in more robust optimization.



---

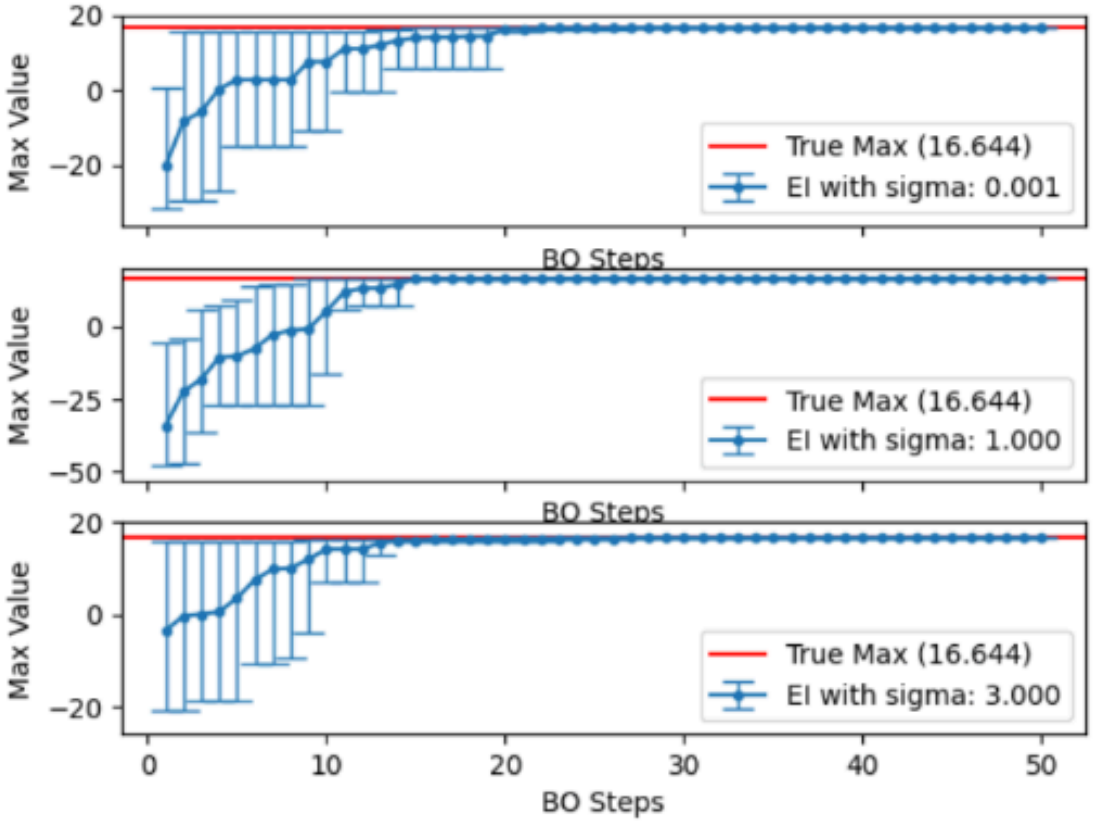
d)

- Small v values are less effective in Bayesian optimization as shown.
- This is because small v values assume the function to be very rough, leading to high correlation only for very similar input pairs.
- In Bayesian optimization, it's preferable to have a smoother kernel function to encourage exploration.
- However, excessively smooth kernel functions can also hinder performance, resulting in slower convergence, as observed when v = 10.



e)

- The plot displays Bayesian optimization with various σ values.
- Small σ values are less effective, although the difference isn't as pronounced as with the Matern kernel function.
- It is preferable to have a moderately smooth kernel function (moderate σ value) for Bayesian optimization to encourage exploration.
- However, being overly smooth can also be detrimental, as observed when σ = 10.



Question 2:

Code:

```
import numpy as np
import scipy.optimize
from functools import partial
import matplotlib.pyplot as plt
import time
from mnistSGD_utils import *
from stochastic_gradient_descent import *
```

```
data = np.load("MNIST_Digits49_19x19.npy", allow_pickle=True).item()
x_train_data = data["Xtrain"]
y_train_data = data["ytrain"]
x_test_data = data["Xtest"]
y_test_data = data["ytest"]
num_features = x_train_data.shape[-1]
```

```
loss_func = partial(logistic_loss_scaled, X=x_train_data, y=y_train_data)
grad_func = partial(logistic_loss_scaled_grad, X=x_train_data, y=y_train_data)
```

```
options = dict()
options["maxiter"] = 20000
options["ftol"] = 1e-7
initial_weights = np.zeros(num_features)
start_time = get_current_time_milliseconds()
```

```
results = scipy.optimize.minimize(fun=loss_func, x0=initial_weights, jac=grad_func,
                                  method="L-BFGS-B", options=options)
elapsed_time = get_current_time_milliseconds() - start_time
assert(results.success)  # Make sure it converged
optimized_weights = results.x

train_accuracy_full = calc_log_reg_accuracy(optimized_weights, x_train_data,
y_train_data)
test_accuracy_full = calc_log_reg_accuracy(optimized_weights, x_test_data,
y_test_data)
print(f" Elapsed time: {elapsed_time:.3f} msec")
print(f" Train accuracy: {train_accuracy_full:.3f}")
```

```
print(f" Test accuracy: {test_accuracy_full:.3f}")
max_epochs = 30
kappa_values = [0.51, 0.7]
batch_sizes = [1, 100, 11791]

def step_size_func(step_number, kappa_in):
   return np.power(step_number + 10, -kappa_in)
```

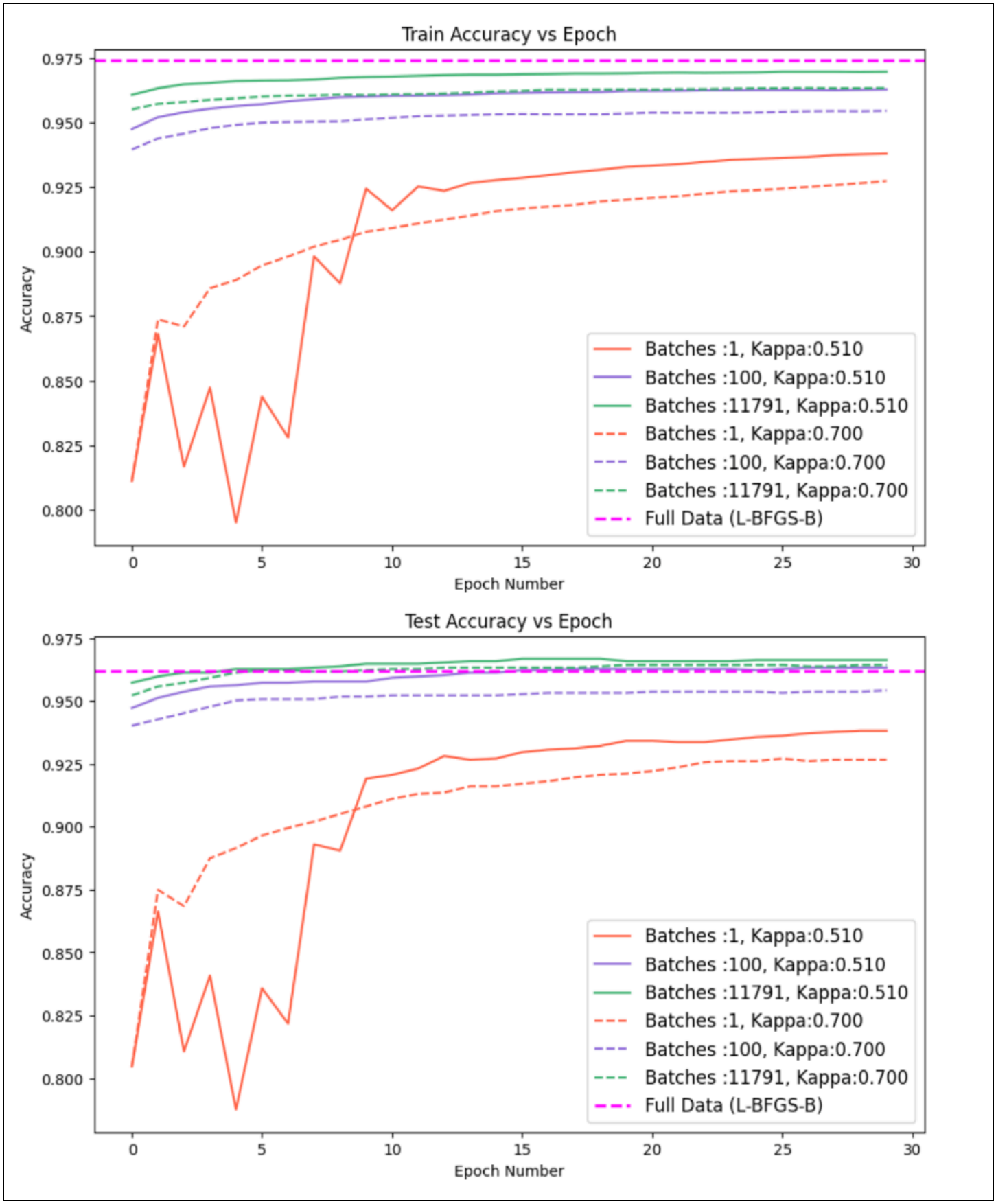
```
# Accuracy callback functions
accuracy_funcs = []
accuracy_funcs.append(partial(calc_log_reg_accuracy, X=x_train_data, y=y_train_data))
accuracy_funcs.append(partial(calc_log_reg_accuracy, X=x_test_data, y=y_test_data))

colors = ["#FF6347", "#9370DB", "#3CB371", "#FFD700"]
styles = ["-", "--", "-."]
for idx_kappa, kappa in enumerate(kappa_values):
   for idx_batch, batch_size in enumerate(batch_sizes):
       # Create batches
       training_batches = create_batches_from_full_dataset(x_train_data,
y_train_data, batch_size)
       # SGD options
       sgd_options = dict()
       sgd_options["max_epoch"] = max_epochs
       sgd_options["max_updates"] = int(batch_size * max_epochs)
       sgd_options["step_size_fn"] = partial(step_size_func, kappa_in=kappa)
       sgd_options["verbose"] = False
       initial_weights = np.zeros(num_features)
       start_time = get_current_time_milliseconds()
       optimized_weights, accuracy_results = min_func_sgd(logistic_loss_scaled,
       logistic_loss_scaled_grad, initial_weights, training_batches, sgd_options,
accuracy_funcs)
       elapsed_time = get_current_time_milliseconds() - start_time
       train_accuracy = accuracy_results[:, 0]
       test_accuracy = accuracy_results[:, 1]
       # Print results
       print(f"Stoch Grad Descent | Batch {batch_size}, Kappa {kappa:.2f}")
       print(f" Elapsed time: {elapsed_time:.3f} msec")
       print(f" Train acc: {train_accuracy[-1]:.3f}")
```

```
        print(f" Test acc: {test_accuracy[-1]:.3f}")
        # Plot accuracies
        fig1 = plt.figure(1, figsize=(10, 6))
        plt.plot(train_accuracy, color=colors[idx_batch], linestyle=styles[idx_kappa],
                 label="Batches :{:d}, Kappa:{:0.3f}".format(batch_size, kappa))
        fig2 = plt.figure(2, figsize=(10, 6))
        plt.plot(test_accuracy, color=colors[idx_batch], linestyle=styles[idx_kappa],
                 label="Batches :{:d}, Kappa:{:0.3f}".format(batch_size, kappa))
```

```
l1 = ["Train", "Test"]
for i1 in [1, 2]:
    fig = plt.figure(i1, figsize=(10, 6))
    plt.title("{} Accuracy vs Epoch".format(l1[i1 - 1]))
    plt.xlabel("Epoch Number")
    plt.ylabel("Accuracy")
    # Plot full data accuracy
    if(i1 == 1):
        plt.axhline(y=train_accuracy_full, color="magenta", linestyle="--",
linewidth=2,label="Full Data (L-BFGS-B)")
    else:
        plt.axhline(y=test_accuracy_full, color="magenta", linestyle="--",
linewidth=2,label="Full Data (L-BFGS-B)")
    plt.legend(fontsize=12)
```

A)

- L-BFGS attains a training accuracy of 0.974 and a test accuracy of 0.962.
- Training this model takes approximately 2.1 seconds.



B)

- Batch 1, Kappa 0.51
  - Elapsed time: 836.000 msec

    - Train accuracy: 0.938
    - Test accuracy: 0.938
- Batch 100, Kappa 0.51
    - Elapsed time: 823.000 msec
    - Train accuracy: 0.963
    - Test accuracy: 0.963
- Batch 11791, Kappa 0.51
    - Elapsed time: 47144.000 msec
    - Train accuracy: 0.970
    - Test accuracy: 0.966

- The solid lines in the plot represent relevant performance measures for a specific step size schedule.
- Usually, when we use more batches, the model gets better and better at predicting both training and test data accurately.
- This happens because with more batches, the model gets to adjust its weights more times in each round of learning, using the data more effectively.
- The run with only one batch appears particularly noisy, especially when fewer than **10** epochs have been completed.
- This is due to a very aggressive step size schedule, with relatively large step sizes for early iterations.
- Since there is only one batch, each epoch corresponds to exactly one update of the weights (w), explaining why performance stabilizes after more than 10 epochs.

C)

- Batch 1, Kappa 0.70
    - Elapsed time: 856.000 msec
    - Train accuracy: 0.927
    - Test accuracy: 0.927

- Batch 100, Kappa 0.70
    - Elapsed time: 1901.000 msec
    - Train accuracy: 0.954
    - Test accuracy: 0.954

- Batch 11791, Kappa 0.70
    - Elapsed time: 47791.000 msec
    - Train accuracy: 0.963
    - Test accuracy: 0.964

- The dashed lines in the plot represent the performance of a specific step size schedule.
- Learning appears to be sensitive to the step-size schedule.It seems that the step size really matters for learning.
- After many passes through the data, the earlier schedule (with $\kappa = 0.51$) is noticeably better than $\kappa = 0.7$ for all numbers of batch settings.
- But when we're only using one batch, the latter schedule is better at the beginning.
- It's important to adjust the step size schedule to fit the specific data we're working with to get the best results with SGD.

---

d)

- The optimal configuration for SGD with **11791** batches and κ=0.51 achieves a test accuracy of **0.964** after **30** epochs.
- This accuracy is roughly equal to, and marginally superior to, the test accuracy of L-BFGS run to convergence, which is **0.962.**
- L-BFGS shows signs of slight overfitting, as indicated by the fact the training accuracy remains higher than SGD's for any number of epochs.
- The most successful SGD run achieves near-ideal performance (test accuracy 0.957) after only 1 pass through the data.
- This shows that SGD is quite competitive and can get even better with more data to learn from.

Question 3:

A)

The MAP classification rule aims to minimize the probability of error by comparing the likelihoods of data belonging to different categories.

$$\log \frac{p(t_n = 1 \mid x_n)}{p(t_n = 0 \mid x_n)} = (x_n \mid t_n = 1) - \log p(x_n \mid t_n = 0)$$
$$= -\frac{1}{2}(x_n - \mu_1)^T \Sigma^{-1}(x_n - \mu_1) + \frac{1}{2}(x_n - \mu_0)^T \Sigma^{-1}(x_n - \mu_0)$$
$$= \mu_1^T \Sigma^{-1} x_n - \frac{1}{2}\mu_1^T \Sigma^{-1} \mu_1$$

Cancel the quadratic terms involving only x_n.

We know $\mu_0 = [0, 0]^T$ zeros out terms involving μ_0.

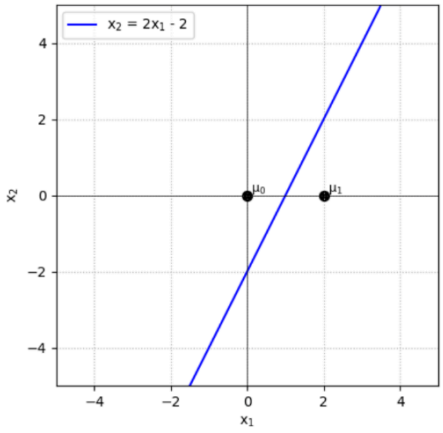
The optimal decision rule chooses class 1 if

$$\mu_1^T \Sigma^{-1} x_n \geq \frac{1}{2}\mu_1^T \Sigma^{-1} \mu_1$$

Put the in the values of μ_1, Σ

$$2x_{n1} - x_{n2} \geq 2$$
$$x_{n2} \leq 2x_{n1} - 2$$

The optimal decision boundary is shown by the blue solid line

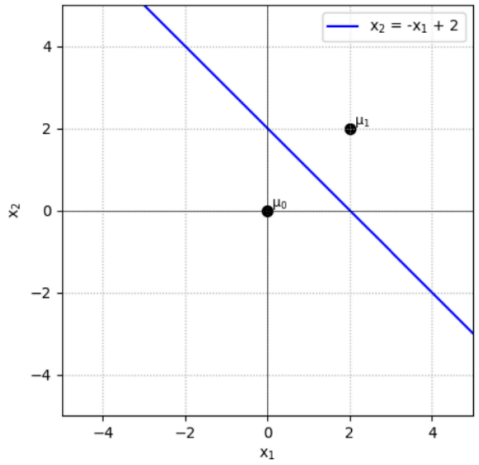


---

b) From answer A we take values

$$\mu_1^T \Sigma^{-1} x_n \geq \frac{1}{2}\mu_1^T \Sigma^{-1} \mu_1$$
$$x_{n1} + x_{n2} \geq 2$$
$$x_{n2} \geq -x_{n1} + 2$$

The optimal decision boundary is the solid blue line.



---

c)

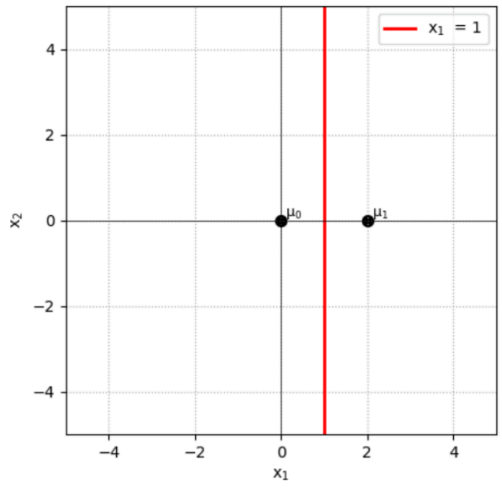
As $N \to \infty$, the ML parameter estimates will match the means and variances of the true distribution $p(x,t)$.
The naive Bayes model will exactly recover the equal class frequencies, and the class means μ_0, μ_1.
But the independence assumption will cause the covariance matrix to be incorrectly estimated as follows:

$$\hat{\Sigma} = \begin{bmatrix} 4/3 & 0 \\ 0 & 4/3 \end{bmatrix}, \quad \hat{\Sigma}^{-1} = \begin{bmatrix} 3/4 & 0 \\ 0 & 3/4 \end{bmatrix}$$

The naive Bayes decision boundary will then be as in part (a), but using $\hat{\Sigma}$ in place of Σ :

$$\begin{aligned} \mu_1^T \hat{\Sigma}^{-1} x_n &\geq \frac{1}{2} \mu_1^T \hat{\Sigma}^{-1} \mu_1 \\ 3/2 x_{n1} + 0 x_{n2} &\geq 3/2 \\ x_{n1} &\geq 1 \end{aligned}$$

The boundary is different from the optimal decision rule, and less accurate in the figure.



D)Applying the analysis answer C, μ_1,

$$\mu_1^T \hat{\Sigma}^{-1} x_n \geq \frac{1}{2} \mu_1^T \hat{\Sigma}^{-1} \mu_1$$

$$1.5x_{n1} + 1.5x_{n2} \geq 3$$

$$x_{n2} \geq -x_{n1} + 2$$

- Even though the covariance is not accurately estimated, the naive Bayes decision rule coincidentally works just as well as the true decision rule.
- Both rules are equally accurate despite this inaccurate estimation.