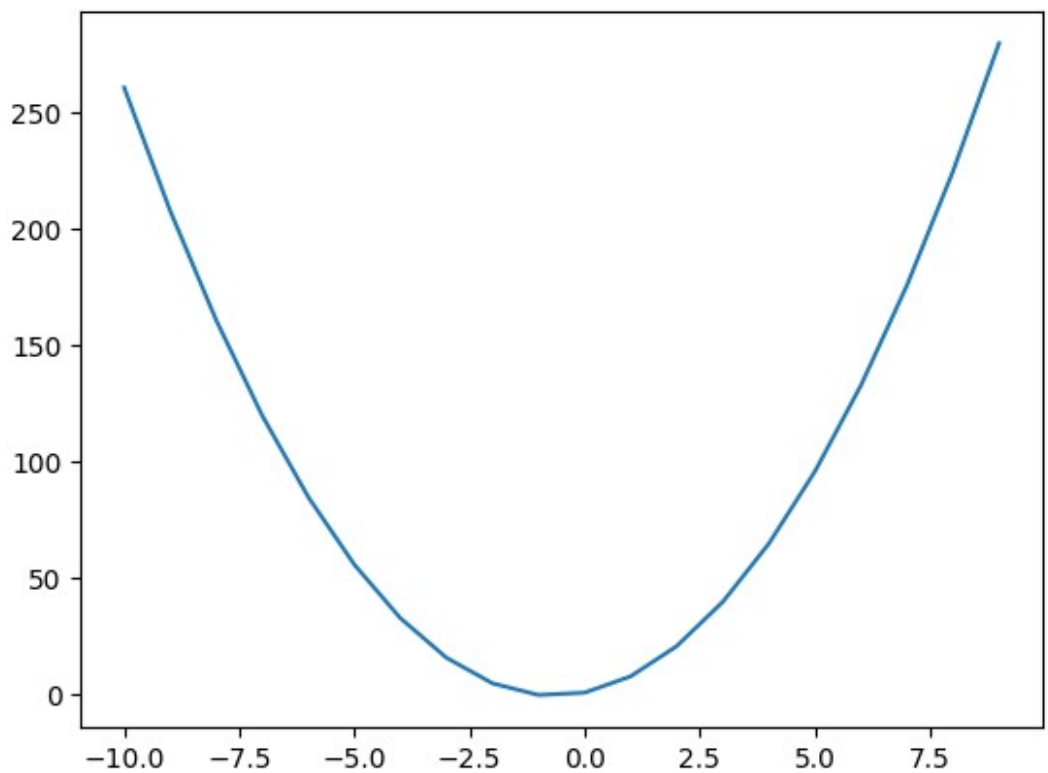
# Question 1

**Section 1**

```
import numpy as np
import matplotlib.pyplot as plt
```

**bold text**

```
f = lambda x: 1 + 4 * x + 3 * x ** 2


x_list = range(-10, 10)
y_list = [f(x) for x in x_list]
plt.plot(x_list, y_list)
plt.show()
```




**Section 2**

```
grad_f = lambda x: 4 + 6 * x
```

**Section 3**

$4+6x=0 \Leftrightarrow x=-\frac{2}{3}$

So the stationary point is $x^{¿}=-\frac{2}{3}$

**Section 4**

```
def grad_update(grad, x, eta):
  # Returns gradient descent iteration update
  return x - eta * grad(x)
```

**Section 5**

```
# The following values we can choose
eps = 1e-8
eta = 1e-3
x_prev = 8

# Now we'll perform the gradient descent
x_next = grad_update(grad_f, x_prev, eta)

while np.abs(f(x_prev) - f(x_next)) >= eps:
  x_prev = x_next
  x_next = grad_update(grad_f, x_prev, eta)

# When we reach here, the method converged
# We'll print the final convergence x value:
print(x_next)
```

```
-0.6661424914172498
```

The value is not exactly $\frac{2}{3}$, but we can choose $\varepsilon$ to be as small as we like, which will bring us as close as we want to the stationary point $\frac{2}{3}$.

**Section 6**

```
eps_list = [1e-5, 1e-15]
eta_list = [0.33, 0.2, 1e-1, 1e-5]
x0_list = [-3, 5]


def grad_descent(x0, eta, eps):
  x_prev = x0
  x_next = grad_update(grad_f, x_prev, eta)
  x_list = [x_prev]
  T = 1
  while np.abs(f(x_prev) - f(x_next)) >= eps:
```

```
    T += 1
    x_prev = x_next
    x_next = grad_update(grad_f, x_prev, eta)
    x_list.append(x_next)

  print(f'Current x0: {x0}, Current eta: {eta} and Current epsilon: 
{eps}')
  print(f'Number of iterations (T value): {T}')
  return x_list


for x0 in x0_list:
  for eta in eta_list:
    for eps in eps_list:
      grad_descent(x0, eta, eps)
  print('\n')
```

```
Current x0: -3, Current eta: 0.33 and Current epsilon: 1e-05
Number of iterations (T value): 276
Current x0: -3, Current eta: 0.33 and Current epsilon: 1e-15
Number of iterations (T value): 843
Current x0: -3, Current eta: 0.2 and Current epsilon: 1e-05
Number of iterations (T value): 6
Current x0: -3, Current eta: 0.2 and Current epsilon: 1e-15
Number of iterations (T value): 13
Current x0: -3, Current eta: 0.1 and Current epsilon: 1e-05
Number of iterations (T value): 9
Current x0: -3, Current eta: 0.1 and Current epsilon: 1e-15
Number of iterations (T value): 22
Current x0: -3, Current eta: 1e-05 and Current epsilon: 1e-05
Number of iterations (T value): 43984
Current x0: -3, Current eta: 1e-05 and Current epsilon: 1e-15
Number of iterations (T value): 233915


Current x0: 5, Current eta: 0.33 and Current epsilon: 1e-05
Number of iterations (T value): 320
Current x0: 5, Current eta: 0.33 and Current epsilon: 1e-15
Number of iterations (T value): 886
Current x0: 5, Current eta: 0.2 and Current epsilon: 1e-05
Number of iterations (T value): 6
Current x0: 5, Current eta: 0.2 and Current epsilon: 1e-15
Number of iterations (T value): 14
Current x0: 5, Current eta: 0.1 and Current epsilon: 1e-05
Number of iterations (T value): 10
Current x0: 5, Current eta: 0.1 and Current epsilon: 1e-15
Number of iterations (T value): 23
Current x0: 5, Current eta: 1e-05 and Current epsilon: 1e-05
Number of iterations (T value): 58772
Current x0: 5, Current eta: 1e-05 and Current epsilon: 1e-15
```

```
Number of iterations (T value): 249011
```

As we can see, the better $T$ value is consistently obtained for the larger value of epsilon, which is expected as getting a better result usually would mean more iterations.

The results for the learning rate are more complex. The best $T$ values are obtained for a learning rate of $0.2$, but the number of iterations can increase massively if we increase or decrease this value too much.

Finally, and perhaps a bit more obviously, the closer the initial guess of $x_0$ to the stationary point the faster the method converges.

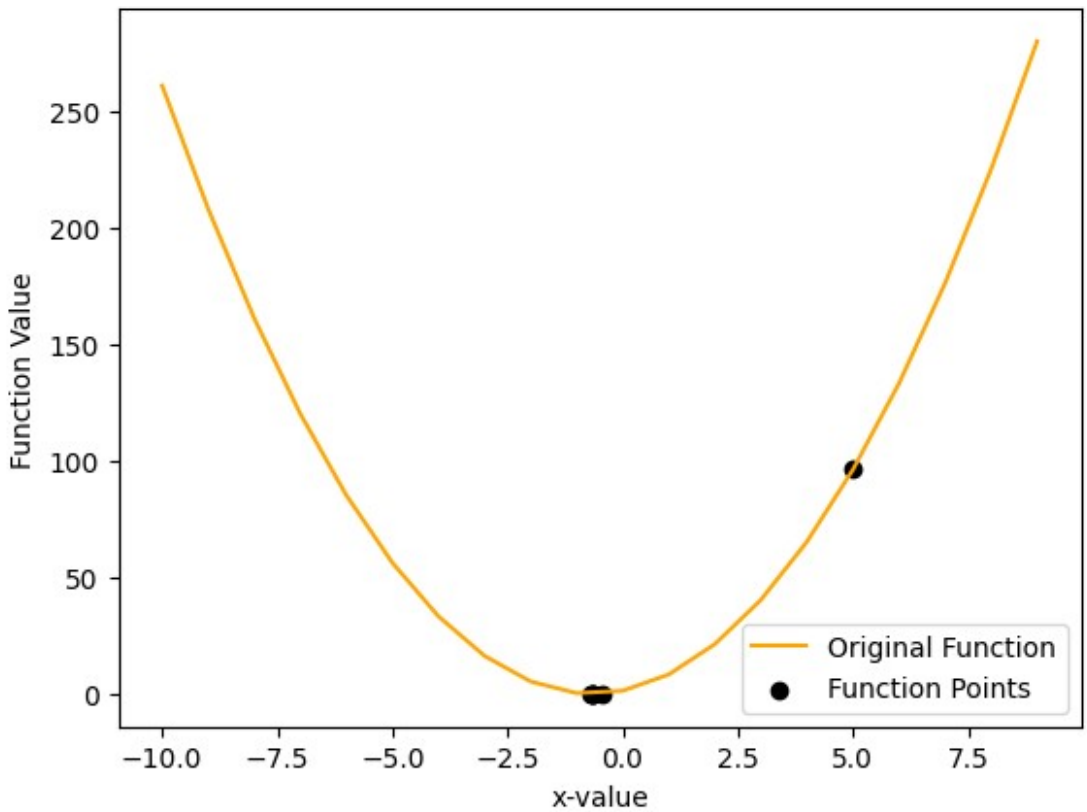
**Section 7**

The best values we obtained are:

$x_0 = 5, \eta = 0.2, \varepsilon = 10^{-5}, T = 6$

```
# We'll take the best results from the previous section
x0 = 5
eta = 0.2
eps = 1e-5

x_list = grad_descent(x0, eta, eps)
y_list = [f(x) for x in x_list]


actual_x_list = range(-10, 10)
actual_y_list = [f(x) for x in actual_x_list]
plt.plot(actual_x_list, actual_y_list, color='orange', label='Original
Function')
plt.scatter(x_list, y_list, color='black', label='Function Points')
plt.legend()
plt.xlabel('x-value')
plt.ylabel('Function Value')
plt.show()
```

```
Current x0: 5, Current eta: 0.2 and Current epsilon: 1e-05
Number of iterations (T value): 6
```




# Question 2

**Section 1**

We want to show that

$$f(w,b)=\frac{1}{m}\sum_{i=1}^{m} max\{0,1-y_i(\langle w,x_i\rangle+b)\}+\lambda\|w\|^2$$

Is a convex function.

First,

$$1-y_i(\langle w,x_i\rangle+b)=1-y_i b-y_i w^T x_i$$

Is a linear function of $w$ and $b$, and therefore is both convex and concave, specifically convex.

The constant function $0$ is also convex, and a known theorem states that the max of convex functions is also convex.

All these facts combine into the fact that the first expression in $f$ is a convex function.

We'll now prove that norm squared is convex, using the given hint:

$\| w \| \geq 0$ we know to be a convex function, and is non-negative by definition. It follows that the norm squared is also a convex function.

Finally, we have shown that both expressions we are summing in the objective functions are convex. By a known theorem, if $f_1, f_2$ are convex then $w_1 f_1 + w_2 f_2$ is also a convex function.

Thus, the objective function is a convex function.

**Section 2**

We'll separate into 3 cases:

- If $l(w, x_i, y_i) = l(w^{¿}, x_i, y_i) = 0$:

$\| 0 - 0 \| = 0 \leq R \| w^{¿} - w \|$ Holds because $R \geq 0$ and any norm is non-negative.

- If (WLOG) $l(w, x_i, y_i) \neq 0, l(w^{¿}, x_i, y_i) = 0$:

$$\| l(w, x_i, y_i) - l(w^{¿}, x_i, y_i) \| = \| l(w, x_i, y_i) \| = \| 1 - y_i \langle w, x_i \rangle \|$$

Since $1 - y_i \langle w, x_i \rangle \geq 0$, and $y_i \langle w^{¿}, x_i \rangle \geq 1$, it follows that

$$\| 1 - y_i \langle w, x_i \rangle \| \leq \| y_i \langle w^{¿}, x_i \rangle - y_i \langle w, x_i \rangle \| = |y_i) \| \langle w^{¿} - w, x_i \rangle \| = \| \langle w^{¿} - w, x_i \rangle \|$$

Using C.S inequality:

$$\| l(w, x_i, y_i) - l(w^{¿}, x_i, y_i) \| \leq \| w^{¿} - w \| \cdot \| x_i \| \leq max_k \leq \| w^{¿} - w \| \cdot \| x_k \| = R \| w^{¿} - w \|$$

- If $l(w, x_i, y_i) \neq 0, l(w^{¿}, x_i, y_i) \neq 0$:

$$\| l(w, x_i, y_i) - l(w^{¿}, x_i, y_i) \| = \| y_i \langle w^{¿}, x_i \rangle - y_i \langle w, x_i \rangle \|$$

And we continue the proof exactly as before.

In conclusion, we have shown that for all 3 cases the Hinge Loss function is R-Lipchitz with the required $R$ value.

**Section 3**

The objective function in a specific observation is:

$$f(w, b) = max\{0, 1 - y_i(\langle w, x_i \rangle + b)\} + \lambda \| w \|^2$$

For the subgradient by $b$, we only need to show the sub-gradient of the hinge-loss function, as the norm is not dependent on it. In this case, we can use the theorem shown in the tutorial to obtain

$$v = \begin{pmatrix} 0, & \text{if } 1 - y_i(\langle w, x_i \rangle + b) \leq 0 \\ -y_i, & \text{if } 1 - y_i(\langle w, x_i \rangle + b) > 0 \end{pmatrix}$$

This is the sub-gradient by $b$. The sub-gradient by $w$ is as follows:

The sub-gradient of the hinge-loss by $w$ we have shown in the tutorial is

$$z=\begin{pmatrix} 0, & \text{if } 1-y_i(\langle w, x_i\rangle+b)\leq 0 \\ -y_i x_i, & \text{if } 1-y_i(\langle w, x_i\rangle+b)>0 \end{pmatrix}$$

Now we need to include the sub-gradient of the norm squared. We notice that the function is differentiable at $w$ for all $w \in R^n$. It follows that

$$u=2\lambda w$$

Combined, the sub-gradient by $w$ is:

$$v=z+u=\begin{pmatrix} 2\lambda w, & \text{if } 1-y_i(\langle w, x_i\rangle+b)\leq 0 \\ 2\lambda w-y_i x_i, & \text{if } 1-y_i(\langle w, x_i\rangle+b)>0 \end{pmatrix}$$

**Section 4**

```
import numpy as np
import matplotlib.pyplot as plt

def svm_aux(X):
  # Receives the data matrix, finds m, d and generates uniform w and b
  m, d = np.shape(X)
  w = np.random.uniform(low=0.0, high=1.0, size=d)
  b = np.random.uniform(low=0.0, high=1.0)

  return m, d, w, b

def svm_with_sgd(X, y, lam=0, epochs=1000, l_rate=0.01,
sgd_type='practical'):
  np.random.seed(2)
  # Wrapper function for sgd type

  sgd = practical_svm_with_sgd if sgd_type == 'practical' else
theory_svm_with_sgd
  # sgd = aux if sgd_type == 'practical' else theory_svm_with_sgd

  return sgd(X, y, lam, epochs, l_rate)


def calc_sub_gradient(x, y, w, b, lam):
  '''
  Calculate sub-gradient of w and b, at data point x, y
  Return: sub-gradient of w, sub-gradient of b
  '''
  # Calculate sub-gradient in regards to w and b
  # Return tuple containing sub_w and sub_b
  sub_w = 2 * lam * w
  sub_b = 0
  if 1 - y * (w.dot(x) + b) > 0:
```

```
    sub_w -= y * x
    sub_b -= y

  return sub_w, sub_b


def practical_svm_with_sgd(X, y, lam, epochs, l_rate):
  # Practical SVM function
  m, d, w, b = svm_aux(X)

  for epoch in range(epochs):
    permutation = np.random.permutation(m)
    for idx in permutation:
      x = X[idx]
      y_cur = y[idx]
      sub_grad_w, sub_grad_b = calc_sub_gradient(x, y_cur, w, b, lam)

      w = w - l_rate * sub_grad_w
      b = b - l_rate * sub_grad_b

  return w, b


def theory_svm_with_sgd(X, y, lam, epochs, l_rate):
  # Theory SVM function
  m, d, w, b = svm_aux(X)

  b_list = [b]
  w_list = [w]

  for epoch in range(m * epochs):
    idx = np.random.randint(0, m)
    x = X[idx]
    y_cur = y[idx]

    sub_grad_w, sub_grad_b = calc_sub_gradient(x, y_cur, w, b, lam)

    w = w - l_rate * sub_grad_w
    b = b - l_rate * sub_grad_b

    w_list.append(w)
    b_list.append(b)

  w = 1 / len(w_list) * sum(w_list)
  b = 1 / len(b_list) * sum(b_list)

  return w, b
```

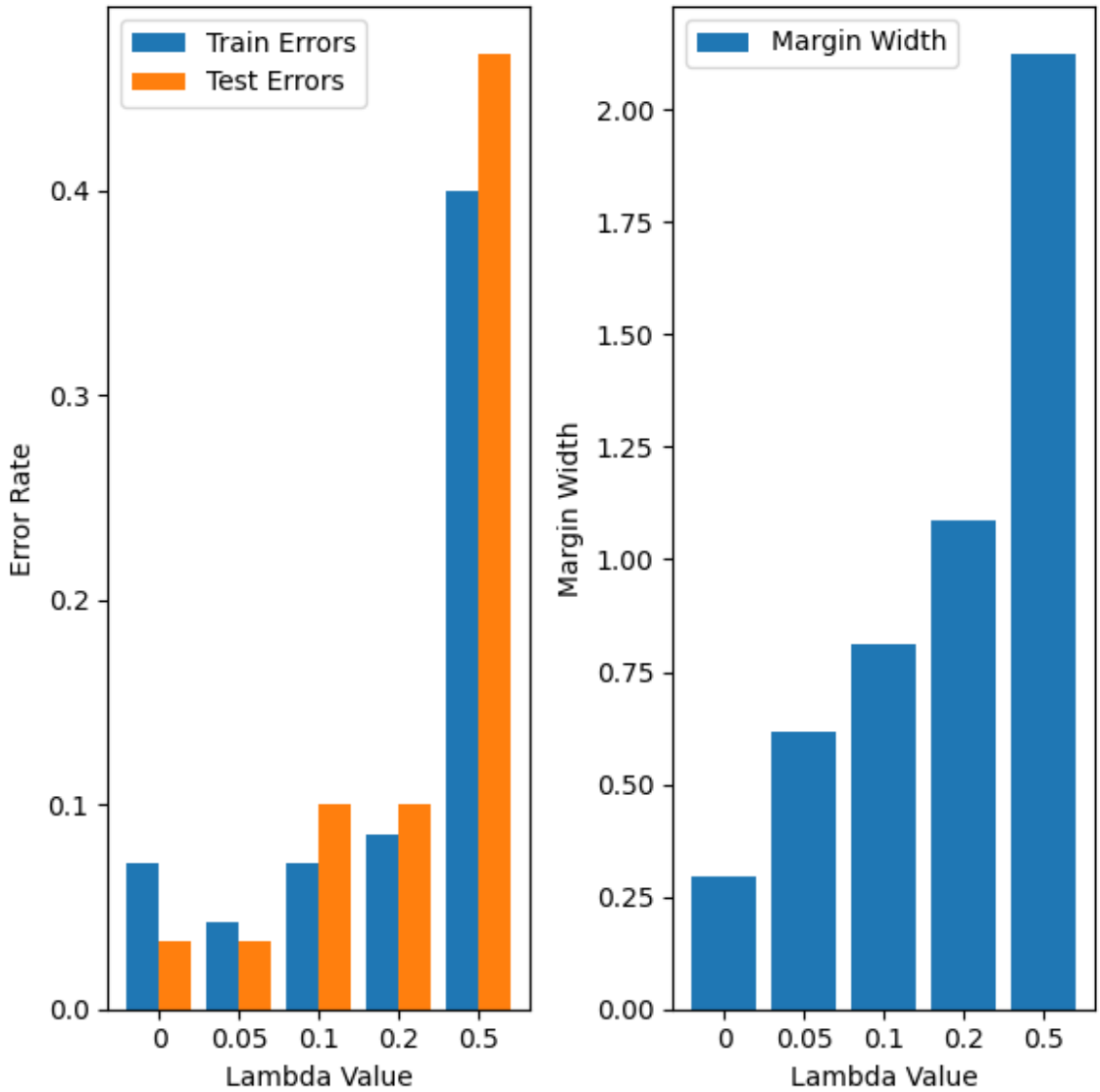
**Section 5**

```
def calculate_error(w, b, X, y):
  # Calculate error rate of classifier given w and bias

  y_pred = np.where(X.dot(w) + b > 0, 1, -1)

  return np.mean(y_pred != y)
```

**Section 6**

```
from sklearn.datasets import load_iris

X, y = load_iris(return_X_y=True)
X = X[y != 0]
y = y[y != 0]
y[y==2] = -1
X = X[:, 2:4]

from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3,
random_state=0)

lam_list = [0, 0.05, 0.1, 0.2, 0.5]

train_errors = []
test_errors = []
margins = []

for lam in lam_list:
  w, b = svm_with_sgd(X_train, y_train, lam)
  train_error = calculate_error(w, b, X_train, y_train)
  test_error = calculate_error(w, b, X_val, y_val)

  margin = 1 / np.linalg.norm(w) # As shown in previous HW

  train_errors.append(train_error)
  test_errors.append(test_error)
  margins.append(margin)

plt.subplots(1, 2, figsize=(6, 6))
plt.subplot(1, 2, 1)

width = 0.40
x = np.arange(len(lam_list))


plt.bar(x, train_errors, width=width, label='Train Errors')
plt.bar(x+width, test_errors, width=width, label='Test Errors')
plt.xticks(x+width/2, lam_list)
plt.xlabel('Lambda Value')
plt.ylabel('Error Rate')
```

```
plt.legend()



plt.subplot(1, 2, 2)
plt.bar(x, margins, label='Margin Width')

plt.xticks(x, lam_list)
plt.xlabel('Lambda Value')
plt.ylabel('Margin Width')
plt.legend()




plt.tight_layout()
plt.savefig('Q2_Fig1.png')
plt.show()
```

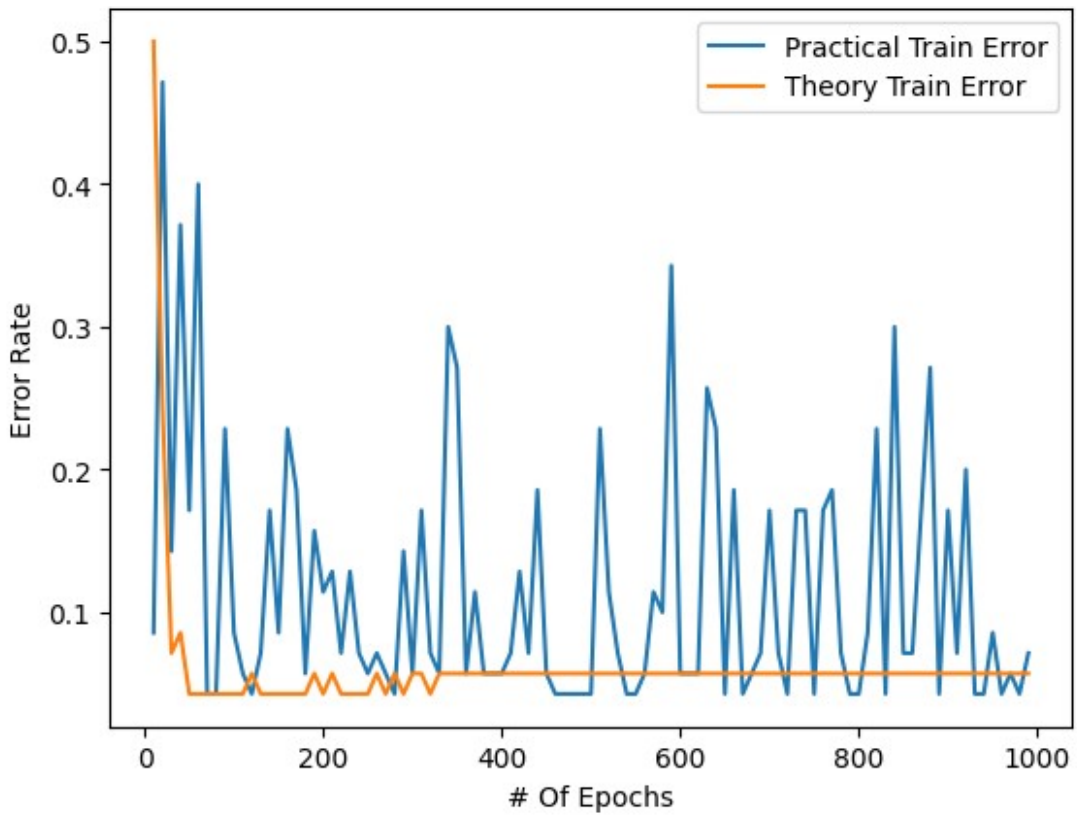



The best model seems to be the model with $\lambda = 0.2$, as we still get a good margin while maintaining good accuracy on the train and validation sets.

**Section 7**

```
lam = 0.2

epochs_list = range(10, 1000, 10)

prac_train_errors = []
prac_test_errors = []

theo_train_errors = []
theo_test_errors = []
```

```
for epochs in epochs_list:
  w, b = svm_with_sgd(X_train, y_train, lam, epochs,
sgd_type='practical')
  train_error = calculate_error(w, b, X_train, y_train)
  test_error = calculate_error(w, b, X_val, y_val)

  prac_train_errors.append(train_error)
  prac_test_errors.append(test_error)

  w, b = svm_with_sgd(X_train, y_train, lam, epochs,
sgd_type='theory')
  train_error = calculate_error(w, b, X_train, y_train)
  test_error = calculate_error(w, b, X_val, y_val)

  theo_train_errors.append(train_error)
  theo_test_errors.append(test_error)


plt.plot(epochs_list, prac_train_errors, label='Practical Train
Error')
plt.plot(epochs_list, theo_train_errors, label='Theory Train Error')
plt.xlabel('# Of Epochs')
plt.ylabel('Error Rate')
plt.legend()
plt.show()
```




# Question 3

**Section 1**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC as SVC

def cross_validation_error(X, y, model, folds):
  """
  Perform k-fold Cross Validation
  Returns (avg_train_error, avg_validation_error)
  Assumes model has fit and predict functions
  """

  train_error_rates = []
  val_error_rates = []

  fold_size = len(X) // folds

  indices = np.arange(len(X))

  for fold in range(folds):
```

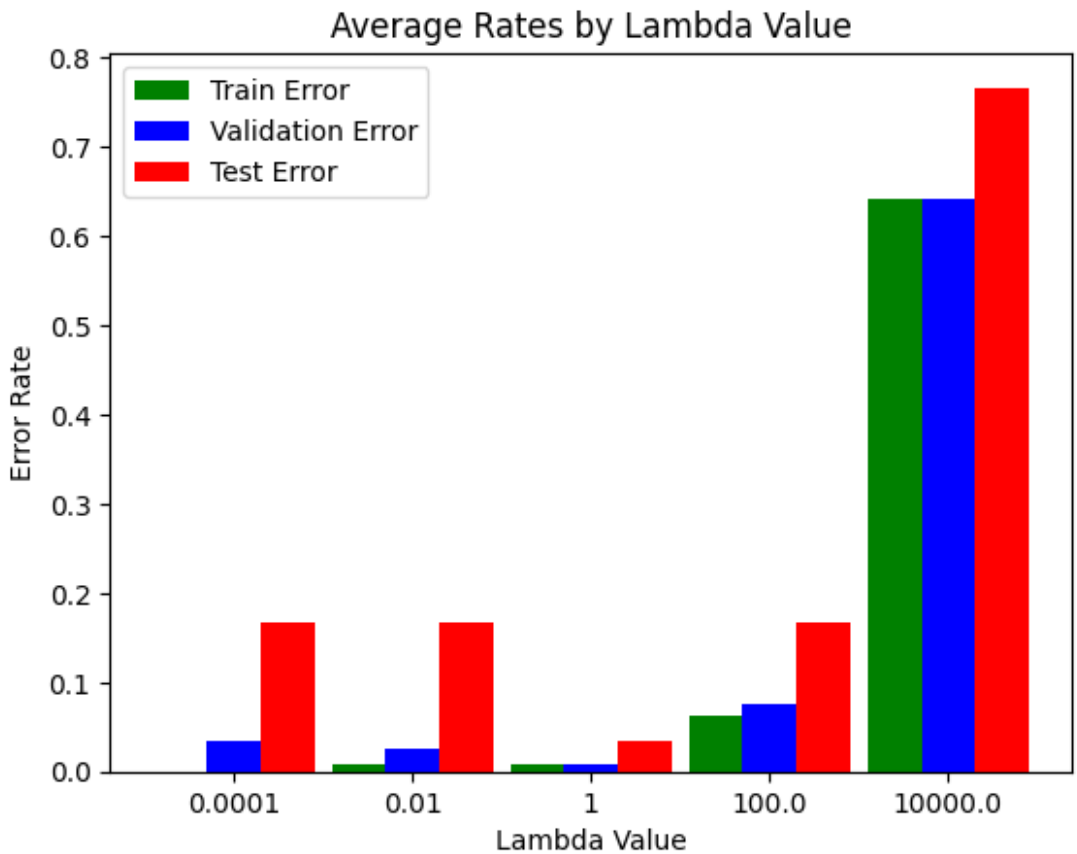
```
    val_start_idx = fold * fold_size
    val_end_idx = val_start_idx + fold_size if fold < folds - 1 else
len(X)

    val = indices[val_start_idx: val_end_idx]
    train = np.setdiff1d(np.arange(len(X)), val)


    X_val = X[val]
    y_val = y[val]
    X_train = X[train]
    y_train = y[train]

    model.fit(X_train, y_train)

    y_pred = model.predict(X_val)
    val_error_rate = np.mean(y_pred != y_val)
    val_error_rates.append(val_error_rate)

    y_pred = model.predict(X_train)
    train_error_rate = np.mean(y_pred != y_train)
    train_error_rates.append(train_error_rate)

  return np.mean(train_error_rates), np.mean(val_error_rates)
```

**Section 2**

```
def svm_results(X_train, y_train, X_test, y_test):
  """
  Calculate Train and Validation Errors over C values with 5-fold CV
  Returns Dictionary of results, keys are 'SVM_lambda_{value}'
  """

  lam_list = [1e-4, 1e-2, 1, 1e2, 1e4]
  resulting_dict = dict()

  for lam in lam_list:
    model_name = f'SVM_lambda_{lam}'
    C = 1 / lam
    model = SVC(kernel='linear', C=C)
    errors = cross_validation_error(X_train, y_train, model, folds=5)

    full_model = model.fit(X_train, y_train)
    test_error = np.mean(model.predict(X_test) != y_test)

    resulting_dict[model_name] = (errors[0], errors[1], test_error)

  return resulting_dict
```

**Section 3**

```
from sklearn.datasets import load_iris
iris_data = load_iris()
X, y = iris_data['data'], iris_data['target']

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=7)

results = svm_results(X_train, y_train, X_test, y_test)
lam_list = [1e-4, 1e-2, 1, 1e2, 1e4]


width = 0.30
x = np.arange(len(lam_list))

avg_train = [results[f'SVM_lambda_{lam}'][0] for lam in lam_list]
avg_val = [results[f'SVM_lambda_{lam}'][1] for lam in lam_list]
avg_test = [results[f'SVM_lambda_{lam}'][2] for lam in lam_list]

plt.bar(x-width, avg_train, color='green', label='Train Error',
width=width)
plt.bar(x, avg_val, color='blue', label='Validation Error',
width=width)
plt.bar(x+width, avg_test, color='red', label='Test Error',
width=width)

plt.xticks(x, lam_list)
plt.xlabel('Lambda Value')
plt.ylabel('Error Rate')
plt.title('Average Rates by Lambda Value')
plt.legend()
plt.show()
```




For Cross Validation, the best model is the model with $\lambda=1$, as it performs very well on both the train and validation sets. If we only look at the training set error though, then $\lambda=0.0001$ is the best.

The best model for the test set is $\lambda=1$, performing less than half as many mistakes as that of $\lambda=0.01$.

This is expected - the smaller the $\lambda$ value is, the better we would expect the model to perform on the train sets, as higher $\lambda$ represents giving less weight to performing well on the train set and more weight to generalizing.

For this reason, having too small of a value will make the model tend to overfit, while having too large of a value makes it underfit.

# Question 4

$g(w)+\langle u-w, \nabla g_j(w)\rangle = g_j(w)+\langle u-w, \nabla g_j(w)\rangle$

This holds directly from $j$'s definition.

Now, using the convexity of $g_j$, it holds that

$g_j(w)+\langle u-w, \nabla g_j(w)\rangle \leq g_j(u) \leq max_{1\leq i\leq r}\, g_i(u) = g(u)$

Combined, we have shown:

$$g(w)+\langle u-w, \nabla g_j(w)\rangle \leq g(u)$$

As required.