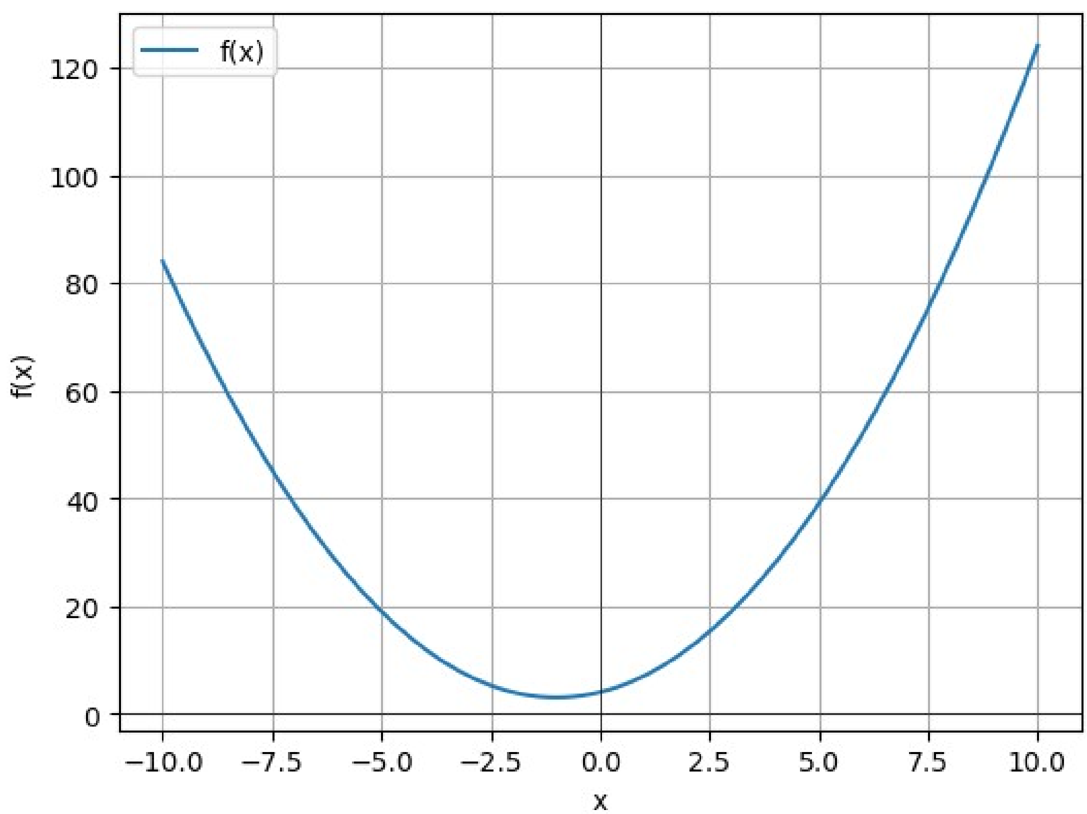
Question 1

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

1.

```
a, b, c = 4, 2, 1

def f(x):
    return a + b * x + c * x ** 2

x_range = np.linspace(-10, 10, 400)
plt.plot(x_range, f(x_range), label='f(x)')
plt.xlabel('x')
plt.ylabel('f(x)')

plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(0, color='black',linewidth=0.5)
plt.grid(True)
plt.legend()
plt.show()
```



2.

f' = b + 2cx = 2 + 2x

```
def grad_f(x):
    return b + 2 * c * x
```

3.

0 = b + 2cx = 2 + 2x

argmin(f(x)) = -1

min(f(x)) = 3

4.

```
def grad_update(grad, x, eta):
    return x - eta * grad
```

5.

```
# Prev/Next x
xp = 0
xq = 1251

# Step of algorithm
eta = 0.05

eps = 1e-7

while abs(f(xq) - f(xp)) > eps:
    xp = xq
    xq = grad_update(grad_f(xq), xq, eta)

print(f'The algorithm converged to x = {xq}, f(x) = {f(xq)}')
```

```
The algorithm converged to x = -0.9993931649857986, f(x) =
3.0000003682487346
```

The value is very close to the actual minimum of our function. This is what we expected, since gradient descent is used in order to find minimum of convex functions (each iteration going against the gradient to find the lowest point on graph).

6.

```
xk_dict = {}

xs = [-42, 1251, 999]
etas = [0.001, 0.01, 0.1]
epss = [1e-6, 1e-5, 1e-4]
```

```
for x in xs:
    for et in etas:
        for ep in epss:
            xk_dict[(x, et, ep)] = [x, grad_update(grad_f(x), x, et)]
            prev = xk_dict[(x, et, ep)][-2]
            cur = xk_dict[(x, et, ep)][-1]

            while abs(f(cur) - f(prev)) > ep:
                xk_dict[(x, et, ep)].append(grad_update(grad_f(cur),
cur, et))
                prev = cur
                cur = xk_dict[(x, et, ep)][-1]

xk_list = sorted(xk_dict.items(), key=lambda item: len(item[1]))

for (x, et, ep), x_iter in xk_list:
    print(f'for x = {x}, eta = {et}, ep = {ep} it takes {len(x_iter)}
iterations (x = {x_iter[-1]})')
```

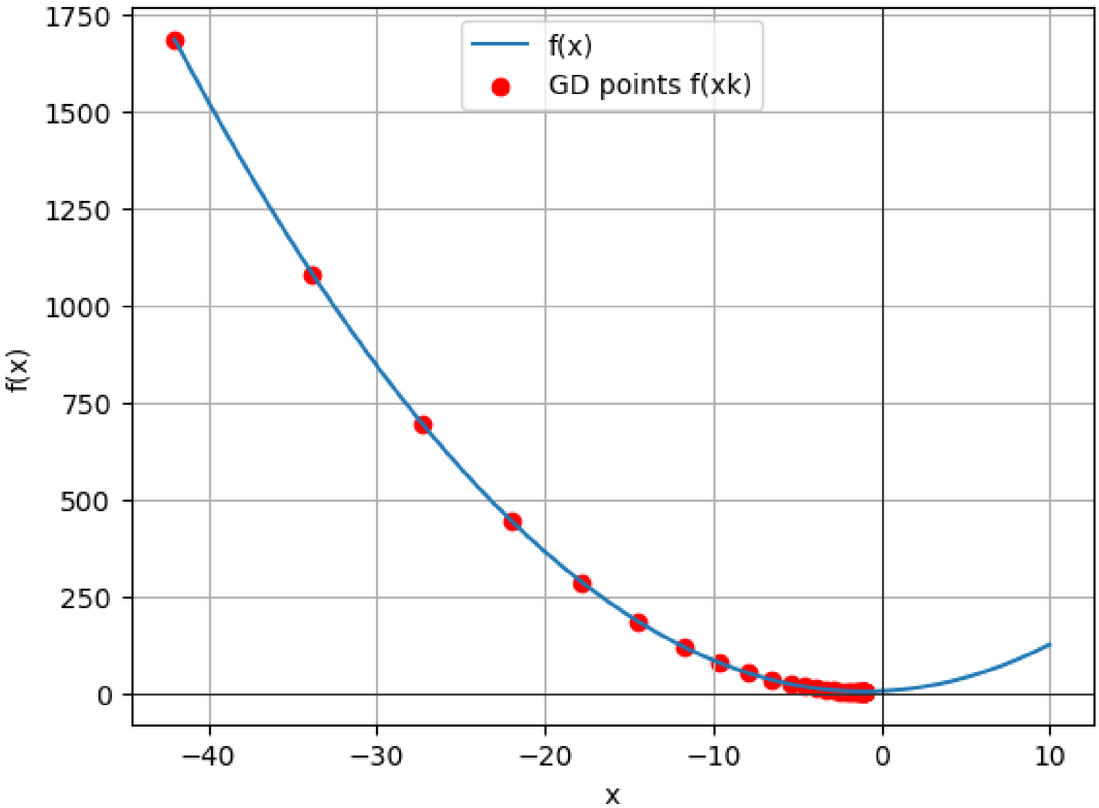
```
for x = -42, eta = 0.1, ep = 0.0001 it takes 37 iterations (x = -
1.0133052606999953)
for x = -42, eta = 0.1, ep = 1e-05 it takes 43 iterations (x = -
1.0034878942609395)
for x = -42, eta = 0.1, ep = 1e-06 it takes 48 iterations (x = -
1.0011429131914247)
for x = 999, eta = 0.1, ep = 0.0001 it takes 52 iterations (x = -
0.9885820184583524)
for x = 1251, eta = 0.1, ep = 0.0001 it takes 53 iterations (x = -
0.9885637496878856)
for x = 999, eta = 0.1, ep = 1e-05 it takes 57 iterations (x = -
0.9962585558084329)
for x = 1251, eta = 0.1, ep = 1e-05 it takes 58 iterations (x = -
0.9962525694977263)
for x = 999, eta = 0.1, ep = 1e-06 it takes 62 iterations (x = -
0.9987740035673073)
for x = 1251, eta = 0.1, ep = 1e-06 it takes 63 iterations (x = -
0.9987720419730151)
for x = -42, eta = 0.01, ep = 0.0001 it takes 334 iterations (x = -
1.0490983686379647)
for x = -42, eta = 0.01, ep = 1e-05 it takes 391 iterations (x = -
1.015522203641003)
for x = -42, eta = 0.01, ep = 1e-06 it takes 448 iterations (x = -
1.0049072670346617)
for x = 999, eta = 0.01, ep = 0.0001 it takes 492 iterations (x = -
0.9507956878460939)
for x = 1251, eta = 0.01, ep = 0.0001 it takes 504 iterations (x = -
0.9516584688229491)
for x = 999, eta = 0.01, ep = 1e-05 it takes 549 iterations (x = -
0.9844443028463915)
```

```
for x = 1251, eta = 0.01, ep = 1e-05 it takes 561 iterations (x = -
0.9847170667363508)
for x = 999, eta = 0.01, ep = 1e-06 it takes 606 iterations (x = -
0.9950821441588718)
for x = 1251, eta = 0.01, ep = 1e-06 it takes 618 iterations (x = -
0.9951683770982406)
for x = -42, eta = 0.001, ep = 0.0001 it takes 2778 iterations (x = -
1.1578690558902842)
for x = -42, eta = 0.001, ep = 1e-05 it takes 3354 iterations (x = -
1.0498297597515271)
for x = -42, eta = 0.001, ep = 1e-06 it takes 3929 iterations (x = -
1.0157597757502377)
for x = 999, eta = 0.001, ep = 0.0001 it takes 4374 iterations (x = -
0.8422907923943299)
for x = 1251, eta = 0.001, ep = 0.0001 it takes 4486 iterations (x = -
0.8422090820341461)
for x = 999, eta = 0.001, ep = 1e-05 it takes 4949 iterations (x = -
0.9501209366048236)
for x = 1251, eta = 0.001, ep = 1e-05 it takes 5061 iterations (x = -
0.9500950938763143)
for x = 999, eta = 0.001, ep = 1e-06 it takes 5524 iterations (x = -
0.9842246308699947)
for x = 1251, eta = 0.001, ep = 1e-06 it takes 5636 iterations (x = -
0.9842164575292421)
```

As we can see, out of the set of hyperparameters that we defined, for starting point -42, with step 0.1 and epsilon = 0.0001 it took us 37 iterations to meet the stopping condition. It took the closest point to minimum, with largest possible step out of the options we've set.

7.

```
x_range = np.linspace(-42, 10, 400)
plt.plot(x_range, f(x_range), label='f(x)')

# Take the value (list of iterations x) of the configuration with
lowest T
x_t = xk_list[0][1]
plt.scatter(x_t, [f(i) for i in x_t], label='GD points f(xk)',
color='red')

plt.xlabel('x')
plt.ylabel('f(x)')

plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(0, color='black',linewidth=0.5)

plt.grid(True)
plt.legend()
plt.show()
```


f(x)
GD points f(xk)
1750
1500
1250
1000
750
500
250
0
−40
−30
−20
−10
0
10
x
f(x)

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

- נסתכל על $h(x)=\sqrt{\|w\|}$ שמוגדרת אי שלילית וקמורה $\Leftarrow$ $h^2(x)$ קמורה שקיבלנו כי צירוף של שני פונקציות קמורות ובפרט קמורה.
- $1-y_i(\langle w,x_i\rangle+b)$ לינארית $\Leftarrow$ קמורה ובפרט מקסימום של שני פונקצי' קמורות גם קמורה.

בפרט $g(w,b):=\frac{1}{n}\sum f(w,b)_i$ קמורה כצירוף של פונקצי' קמורות עם מכפלה בקבוע.

לכן קיבלנו שסו"ג הפונקציות שורה קמורה.

ב. נחלק למקרים:

- עבור $0 > 1-y_i\cdot\langle w,x_i\rangle$ נקבל $l(w,x_i,y_i)=0$

באופן טריוויאלי: $\Rightarrow |0-0| \le \|w_2-w_1\|\cdot\max_k\|x_k\|$

- עבור $0<1-y_i\cdot\langle w,x_i\rangle$

$l(w,x_i,y_i)\ge 0 \Rightarrow \forall w_1,w_2\in\mathbb{R}^d: |l(w_1,x_i,y_i)-l(w_2,x_i,y_i)|$

$= |\cancel{1} - y_i\cdot\langle w_1,x_i\rangle \cancel{-1} + y_i\langle w_2,x_i\rangle| = |\langle w_2-w_1,x_i\rangle| \overset{\text{קו"ש}}{\le} \|w_2-w_1\|\cdot\|x_i\|$

נזכור כי שה-אי-שוויון אז קיים.

$|l(w_1,x_i,y_i)-l(w_2,x_i,y_i)| \le \|w_2-w_1\|\cdot\max_k\|x_k\|$

- אחד הביטויים $l(w_1,x_i,y_i)=0$ והשני $l(w_2,x_i,y_i)>0$ בה"כ $1-y_i\langle w_1,x_i\rangle\le 0 \Rightarrow 1\le y_i\langle w_1,x_i\rangle$

$\|1-y_i\langle w_2,x_i\rangle\| \le \|y_i\langle w_1,x_i\rangle - y_i\langle w_2,x_i\rangle\| = \|\langle w_1-w_2, y_i x_i\rangle\|$

$\underset{\text{קוש}}{\le}\|w_1-w_2\|\|y_ix_i\| \underset{y_i\in\{\pm1\}}{\le} \|w_1-w_2\|\cdot\max_i\|x_i\|$

3. Sgd עבור הדגימה i לפי b:

$$g_b^i = \begin{cases} 0 & 1-y_i(\langle w, x_i\rangle+b)\le 0 \\ -y_i & 1-y_i(\langle w, x_i\rangle+b) > 0 \end{cases}$$

Sgd עבור הדגימה i לפי w:

$$g_w^i = \begin{cases} \lambda w & 1-y_i(\langle w, x_i\rangle+b)\le 0 \\ -y_i x_i+\lambda w & 1-y_i(\langle w, x_i\rangle+b) > 0 \end{cases}$$

rgufdxuwz

July 26, 2024

Q2 Code - Stochastic Gradient Descent

[1]:
```python
import random
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

[2]:
```python
def sgd_b(yi, xi, w, b):
    return 0 if 1 - yi * (np.dot(w, xi) + b) <= 0 else -yi


def sgd_w(yi, xi, w, b, lamb):
    return lamb * w if 1 - yi * (np.dot(w, xi) + b) <= 0 else -yi * xi + lamb *␣
↪w


def sub_gradients_calc(yi, xi, w, b, lamb, l_rate):
    w_grad = sgd_w(yi, xi, w, b, lamb)
    b_grad = sgd_b(yi, xi, w, b)
    w = w - l_rate * w_grad
    b = b - l_rate * b_grad
    return w, b


def svm_with_sgd(X, Y, lam=0.0, epochs=1000, l_rate=0.01, sgd_type='practical'):
    m, d = X.shape
    w = np.random.uniform(0, 1, d)
    b = np.random.uniform(0, 1)
    w_ls, b_ls = [w], [b]
    epoch_train, epoch_test = [], []

    if sgd_type == 'theory':
        for j in range(epochs+1):
            randomize = np.arange(m)
            np.random.shuffle(randomize)
            X_shuffled = X[randomize]
            Y_shuffled = Y[randomize]
```

```
            for xi, yi in zip(X_shuffled, Y_shuffled):
                w, b = sub_gradients_calc(yi, xi, w, b, lam, l_rate)
                w_ls.append(w)
                b_ls.append(b)
            if 10 <= j <= 1000 and j % 10 == 0:
                epoch_train.append(calculate_error(sum(w_ls) / len(w_ls),␣
↪sum(b_ls) / len(b_ls), X, Y))
                epoch_test.append(calculate_error(sum(w_ls) / len(w_ls),␣
↪sum(b_ls) / len(b_ls), X_test, y_test))
        return sum(w_ls) / (m * epochs), sum(b_ls) / (m * epochs), epoch_train,␣
↪epoch_test

    elif sgd_type == 'practical':
        for i in range(epochs + 1):
            for j in range(m):
                idx = random.choice(range(m))
                w, b = sub_gradients_calc(Y[idx], X[idx], w, b, lam, l_rate)
            if 10 <= i <= 1000 and i % 10 == 0:
                epoch_train.append(calculate_error(w, b, X, Y))
                epoch_test.append(calculate_error(w, b, X_test, y_test))
        return w, b, epoch_train, epoch_test
    return 0, 0, 0



def calculate_error(w, bias, X, Y):
    predictions = np.sign(np.dot(X, w) + bias)
    accuracy = np.mean(predictions == Y)
    return 1 - accuracy
```

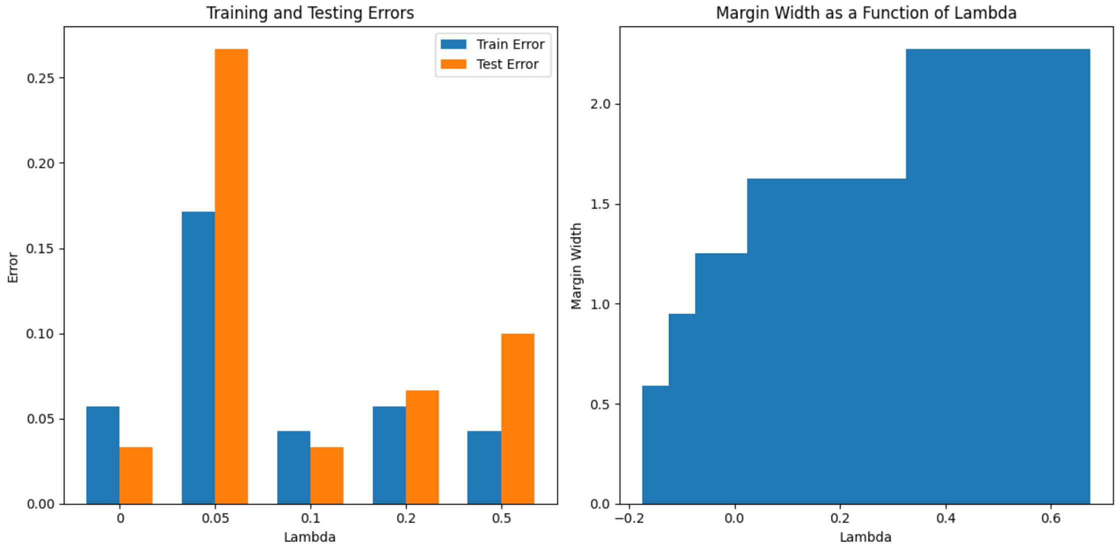
[4]:
```
if __name__ == "__main__":
    np.random.seed(2)
    X, y = load_iris(return_X_y=True)
    X = X[y != 0]
    y = y[y != 0]
    y[y == 2] = -1
    X = X[:, 2:4]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
↪random_state=0)

    lambdas = [0, 0.05, 0.1, 0.2, 0.5]
    train_errors = []
    test_errors = []
    margin_widths = []
```

Part 4 - Training Models and reporting train, test error and the margin width

[5]:
```
for lam in lambdas:
        w, b, _, _ = svm_with_sgd(X_train, y_train, lam=lam, epochs=1000,␣
 ↪l_rate=0.01, sgd_type='practical')
        train_errors.append(calculate_error(w, b, X_train, y_train))
        test_errors.append(calculate_error(w, b, X_test, y_test))
        margin_widths.append(2 / np.linalg.norm(w))

        print(f"Lambda: {lam}, Train Error: {train_errors[-1]}, Test Error:␣
 ↪{test_errors[-1]}")
        print(f"margin width of the model is: {margin_widths[-1]}\n")
```

```
Lambda: 0, Train Error: 0.05714285714285716, Test Error: 0.033333333333333326
margin width of the model is: 0.5906904043247769

Lambda: 0.05, Train Error: 0.17142857142857137, Test Error: 0.2666666666666667
margin width of the model is: 0.9475053997396179

Lambda: 0.1, Train Error: 0.042857142857142816, Test Error: 0.033333333333333326
margin width of the model is: 1.2525339071264805

Lambda: 0.2, Train Error: 0.05714285714285716, Test Error: 0.06666666666666665
margin width of the model is: 1.627126508596363

Lambda: 0.5, Train Error: 0.042857142857142816, Test Error: 0.09999999999999998
margin width of the model is: 2.2722510182671187
```

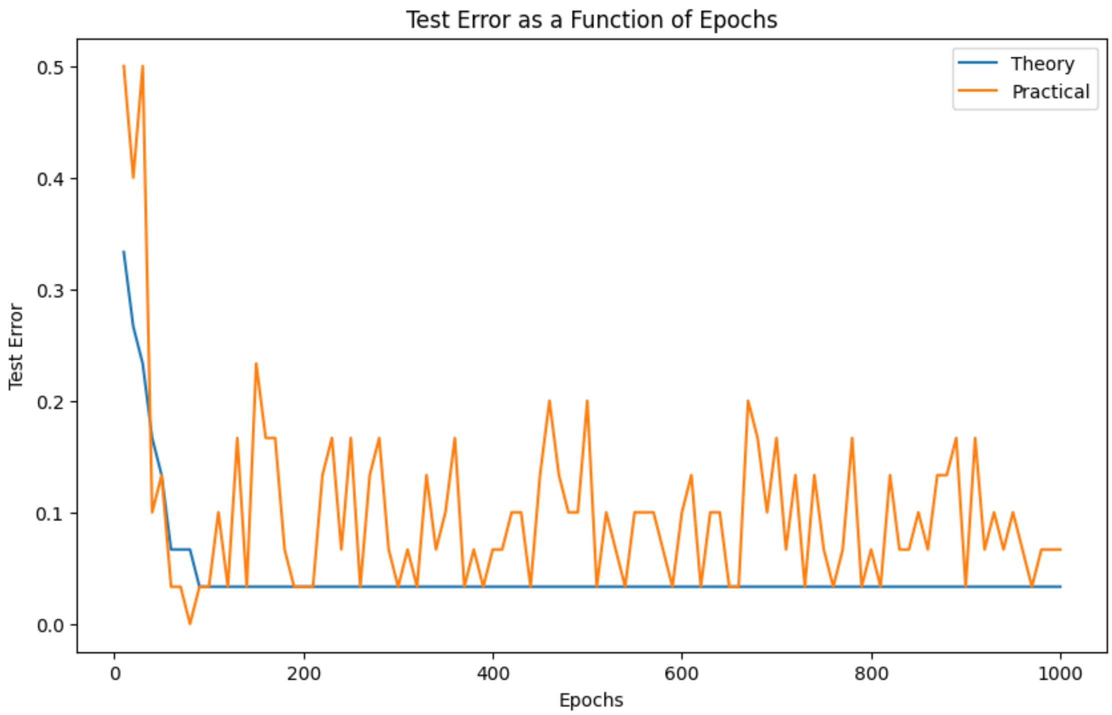
[8]:
```
plt.figure(figsize=(12, 6))
  plt.subplot(1, 2, 1)
  bar_width = 0.35
  index = np.arange(len(lambdas))
  plt.bar(index, train_errors, bar_width, label='Train Error')
  plt.bar(index + bar_width, test_errors, bar_width, label='Test Error')
  plt.xlabel('Lambda')
  plt.ylabel('Error')
  plt.title('Training and Testing Errors')
  plt.xticks(index + bar_width / 2, lambdas)
  plt.legend()

  plt.subplot(1, 2, 2)
  plt.bar(lambdas, margin_widths, bar_width)
  plt.xlabel('Lambda')
  plt.ylabel('Margin Width')
  plt.title('Margin Width as a Function of Lambda')

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



We can see that the model with lambda = 0.1 looks the best. It has the lowest training error out of all the models. With the right lambda the training set will not overfit, we can see for lambda = 0.05,0.2, 0.5 the model overfits since the training error is better than the test error. for lambda = 0 they get similar test error but lambda = 0.1 has a better training error.

[11]:
```
# Part 7a
lam = 0.05
w1, b1, epoch_errors_theory_train, epoch_errors_theory_test =␣
↪svm_with_sgd(X_train, y_train, lam=lam,  l_rate=0.01, sgd_type='theory')
w2, b2, epoch_errors_practical_train, epoch_errors_practical_test =␣
↪svm_with_sgd(X_train, y_train, lam=lam, l_rate=0.01, sgd_type='practical')

epochs = list(range(10, 1001, 10))

plt.figure(figsize=(10, 6))
plt.plot(epochs, epoch_errors_theory_train, label='Theory')
plt.plot(epochs, epoch_errors_practical_train, label='Practical')
plt.xlabel('Epochs')
plt.ylabel('Training Error')
plt.title('Training Error as a Function of Epochs')
plt.legend()
plt.show()
```

[12]:
```python
# Part 7b
plt.figure(figsize=(10, 6))
plt.plot(epochs, epoch_errors_theory_test, label='Theory')
plt.plot(epochs, epoch_errors_practical_test, label='Practical')
plt.xlabel('Epochs')
plt.ylabel('Test Error')
plt.title('Test Error as a Function of Epochs')
plt.legend()
plt.show()
```



The results are logical since the practical method is more hands on and computes until it converges to the result and thus it makes sense that it would take longer to converge in comparison to the theoretical method. The theoretical method as can be seen by it's name uses calculations and proofs that state it's structure and thus it makes sense that it will converge pretty early as we can see in the resulting graphs.

Question 3

1.

```
def cross_validation_error(X, y, model, folds):
    fold_size = X.shape[0] // folds
    sum_train_error = 0
    sum_val_error = 0

    # Shuffle data (no bias by the initial order)
    permutation = np.random.permutation(len(y))
    X = X[permutation]
    y = y[permutation]

    for i in range(folds):
        v_start = i * fold_size
        v_end = (i + 1) * fold_size

        X_val = X[v_start:v_end]
        y_val = y[v_start:v_end]

        X_train = np.concatenate((X[:v_start], X[v_end:]), axis=0)
        y_train = np.concatenate((y[:v_start], y[v_end:]), axis=0)

        model.fit(X_train, y_train)
        y_val_pred = model.predict(X_val)
        y_train_pred = model.predict(X_train)
```

```
        sum_train_error += 1 - np.mean(y_train == y_train_pred)
        sum_val_error += 1 - np.mean(y_val == y_val_pred)

    return sum_train_error / folds, sum_val_error / folds
```

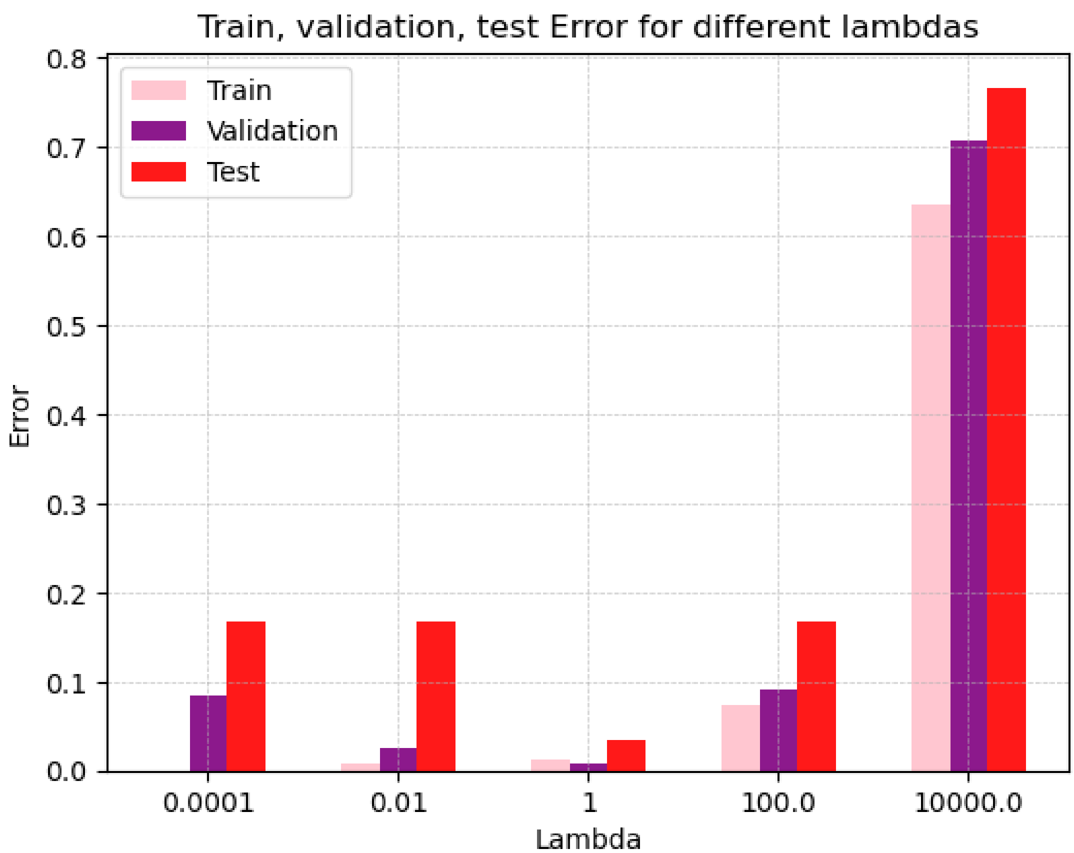
2.

```
from sklearn.svm import SVC

def svm_results(X_train, y_train, X_test, y_test):
    folds = 5
    lambdas = [1e-4, 1e-2, 1, 1e2, 1e4]
    svm_errors = {}

    for l in lambdas:
        model = SVC(kernel='linear', C=1/l)
        train_error, val_error = cross_validation_error(X_train,
y_train, model, folds)
        model.fit(X_train, y_train)
        y_test_pred = model.predict(X_test)
        test_error = 1 - np.mean(y_test == y_test_pred)
        svm_errors[f'{l}'] = (train_error, val_error, test_error)

    return svm_errors
```

3.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

np.random.seed(1)

iris_data = load_iris()
X, y = iris_data['data'], iris_data['target']

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=7)

svm_errors = svm_results(X_train, y_train, X_test, y_test)

fig, ax = plt.subplots()

train_errors = [v[0] for v in svm_errors.values()]
val_error = [v[1] for v in svm_errors.values()]
test_errors = [v[2] for v in svm_errors.values()]

bar_width = 0.2
index = np.arange(len(svm_errors.keys()))
ax.grid(True, which='both', linestyle='--', linewidth=0.5, alpha=0.7)
```

```
bar_train = ax.bar(index - bar_width, train_errors, bar_width,
label='Train', color='pink', alpha=0.9)
bar_val = ax.bar(index, val_error, bar_width, label='Validation',
color='purple', alpha=0.9)
bar_test = ax.bar(index + bar_width, test_errors, bar_width,
label='Test', color='red', alpha=0.9)

ax.set_xlabel('Lambda')
ax.set_ylabel('Error')
ax.set_title('Train, validation, test Error for different lambdas')
ax.set_xticks(index)
ax.set_xticklabels(svm_errors.keys())
ax.legend()

plt.show()
```




```
svm_errors
```

```
{'0.0001': (0.0, 0.08333333333333333, 0.16666666666666663),
 '0.01': (0.008333333333333326, 0.024999999999999977,
0.16666666666666663),
 '1': (0.0125, 0.008333333333333326, 0.033333333333333326),
 '100.0': (0.07291666666666666, 0.09166666666666667,
```

```
0.16666666666666663),
 '10000.0': (0.6354166666666667, 0.7083333333333334,
0.7666666666666666)}
```

According to the CV approach, the model with lambda=1 is the best, giving us an error of 0.0083 on validation set. It is also the best model for test set, giving us 0.0333 error.

This result might be because usually for smaller error on validation set we expect smaller error on test set. As we can see, for smaller values of lambda our model overfits, which leads to increase both in validation and test error. Similarly, we have even greater error for bigger lambda's (our model underfits). Lambda=1 is optimal in this case.

```
```

(4 יהי w, $g(w)=\max_i g_i(w)$ ניקח $j\in\arg\max_i g_i(w)$

יהא: $u\in R^d$

$$g(w)+\langle u-w,\nabla g_j(w)\rangle \underset{\substack{\text{def}\\ \exists j \text{ of } g}}{\le} g_j(w)+\langle u-w,\nabla g_j(w)\rangle$$

g_j קמור גזיר

$$\le g_j(u) \le g(u)$$

$g(w)=\max_i g_i = g_j$

לכן הוכחנו את הנדרש •