ex01-solution

May 13, 2023

Ullrich Köthe: **Machine Learning Essentials**, Summer Semester 2023 Notebook adapted by Daniel Galperin, Fritz Haltenberger

1 Solutions for Exercise 1

Import libraries

[1]:

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

import sklearn
from sklearn.datasets import load_digits
from sklearn import model_selection
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

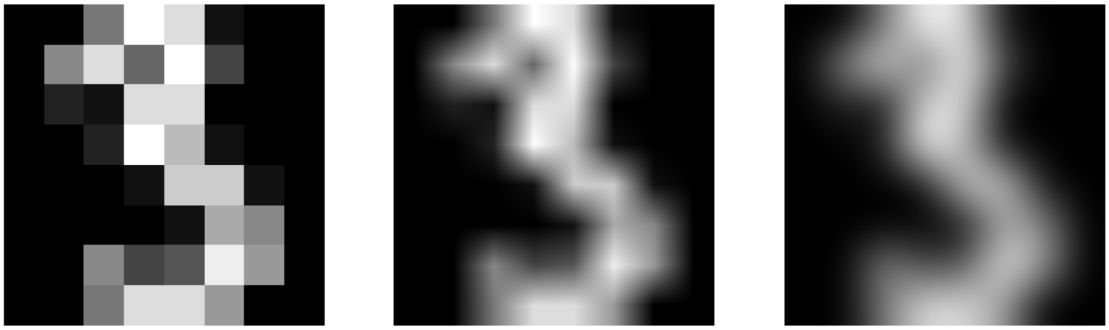
2 1 Exploring the Data

[2]:

```
# load digits data set
digits = load_digits()

data         = digits["data"]
images       = digits["images"]
target       = digits["target"]
target_names = digits["target_names"]

print(f"data.shape          = {data.shape}")
print(f"data.dtype          = {data.dtype}")
print(f"images.shape        = {images.shape}")
print(f"images.shape        = {images.dtype}")
print(f"target.shape        = {target.shape}")
print(f"target.shape        = {target.dtype}")
print(f"target_names.shape  = {target_names.shape}")
print(f"target_names.shape  = {target_names.dtype}")
print(f"target[:20]         = {target[:20]}")
```

```python
img = images[3, :, :]
assert 2 == len(img.shape)

fig, axes = plt.subplots(1, 3, figsize=(10, 3), tight_layout=True)

axes[0].imshow(img, cmap="gray", interpolation="nearest")
axes[0].set_axis_off()

axes[1].imshow(img, cmap="gray", interpolation="bilinear")
axes[1].set_axis_off()

axes[2].imshow(img, cmap="gray", interpolation="bicubic")
axes[2].set_axis_off()

plt.show()
```

```
data.shape          = (1797, 64)
data.dtype          = float64
images.shape        = (1797, 8, 8)
images.shape        = float64
target.shape        = (1797,)
target.shape        = int32
target_names.shape  = (10,)
target_names.shape  = int32
target[:20]         = [0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9]
```



The right-most image illustrates why we generally use `interpolation = 'nearest'`. For scientific analysis, it's important that we can see the exact value of each pixel in an image array, even if `interpolation = 'bicubic'` often yields visually more pleasing results.

If your image has the shape `(width, height)` matplotlib will display it using a colormap, viridis by default. You can change the `cmap` parameter to get any colormap you want, this parameter is ignored for RGB data (images of shape `(width, height, 3)`).

[3]:

```
"""
This function filters the digits (3, 9) from the dataset and randomly splits it␣
 ↪in train and test set.
"""
# Load data
digits = load_digits()

data = digits["data"]
target = digits["target"]

# Data filering
num_1, num_2 = 3, 9
mask = np.logical_or(target == num_1, target == num_2)
data = data[mask]/data.max()
target = target[mask]

# Relabel targets
target[target == num_1] = -1
target[target == num_2] = 1

# split into train and test data
X_all = data
y_all = target
X_train, X_test, y_train, y_test = model_selection.train_test_split(
    X_all, y_all, test_size=0.4 , random_state=0)
print(f"X_train.shape = {X_train.shape}")
print(f"X_test.shape  = {X_test.shape}")
print(f"y_train.shape = {y_train.shape}")
print(f"y_test.shape  = {y_test.shape}")
print(f"y_test[:10]   = {y_test[:10]}")
```

```
X_train.shape = (217, 64)
X_test.shape  = (146, 64)
y_train.shape = (217,)
y_test.shape  = (146,)
y_test[:10]   = [ 1  1  1 -1 -1  1 -1  1 -1  1]
```

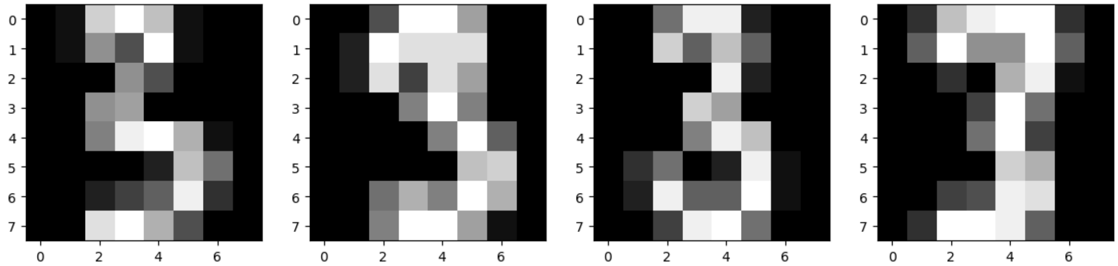
3 2 Hand-crafted classifier

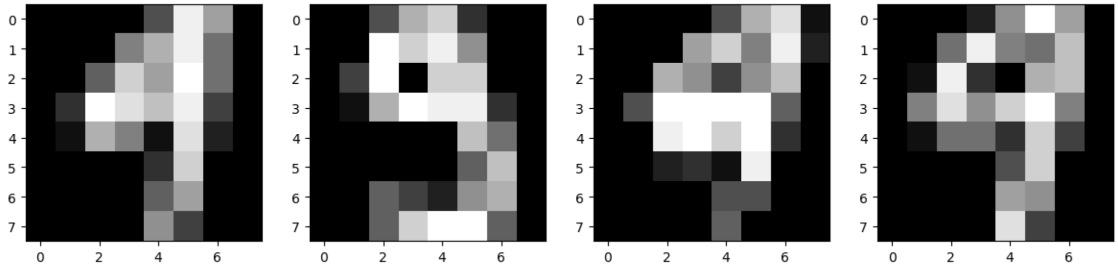
3.1 2.1 Feature construction

Visualize some images to get a sense of which features might be important to distinguish the digits.

[4]:

```
fig, axes = plt.subplots(1, 4, figsize=(12, 3), tight_layout=True)

index = np.random.randint(0, 100)
for i in range(4):
    axes[i].imshow(X_train[y_train==-1][index+i].reshape(8,8), cmap="gray",␣
 ↪interpolation="nearest")
fig, axes = plt.subplots(1, 4, figsize=(12, 3), tight_layout=True)

index = np.random.randint(0, 100)
for i in range(4):
    axes[i].imshow(X_train[y_train==1][index+i].reshape(8,8), cmap="gray",␣
 ↪interpolation="nearest")
```





[5]:
```
def features2d(x):
    """
    Perform a user defined dimension reduction
    input: #instances x 64 numpy array
    output: #instances x 2 numpy array
    """
    images = x.reshape(-1, 8, 8) #instances x row index x column index
    feat1 = np.mean(images[:, 2:4, 2:4], axis=(1, 2))
    feat2 = np.mean(images[:, 4:6, 4:7], axis=(1, 2))

    return np.array([feat1, feat2]).T

def worse_features2d(x):
    """
    Perform a user defined dimension reduction
    input x: #instances x 64 numpy array
    output: #instances x 2 numpy array
    """
    # mean(Image)
    feat1 = np.mean(x, axis=-1)

    # var(Image)
    feat2 = np.var(x, axis=-1)

    return np.array([feat1, feat2]).T
```

The `features2d()` function looks at the average pixel values of two patches in the image. They are informative enough to more or less differentiate between the digit '3' and '9' which we can see in the scatter plot below.

The `worse_features2d()` is much simpler and doesn't extract good features. We won't be using it for the rest of the exercise.

4 1.2 Scatter Plot

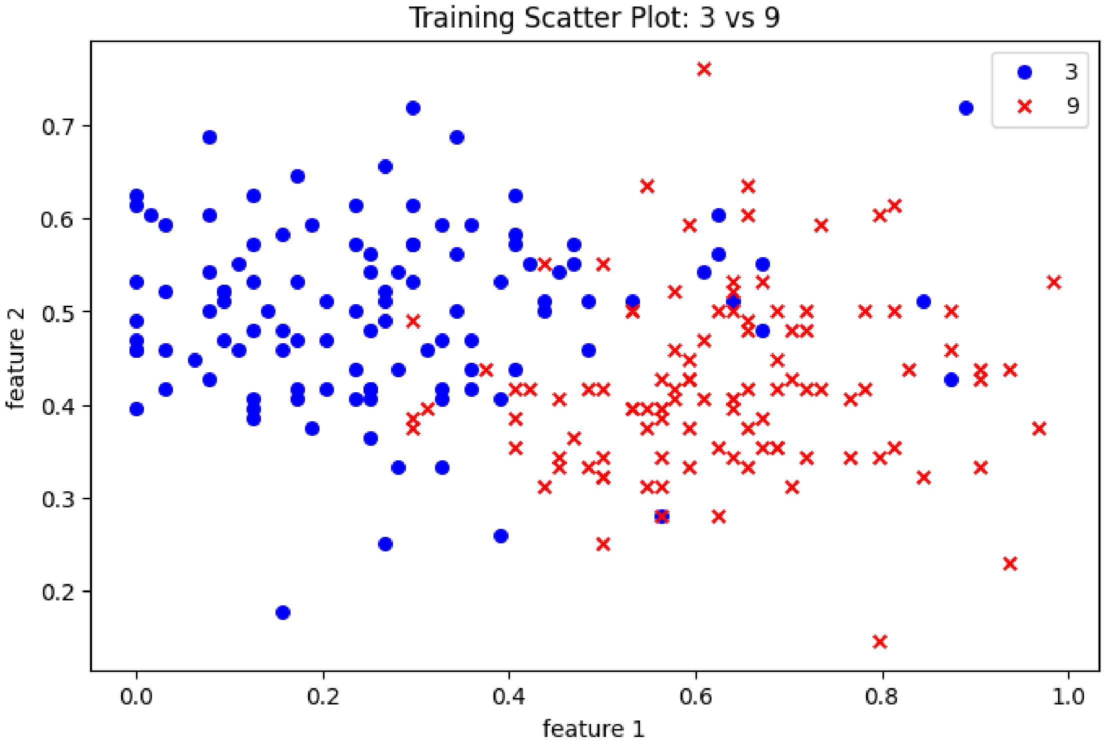
[6]:

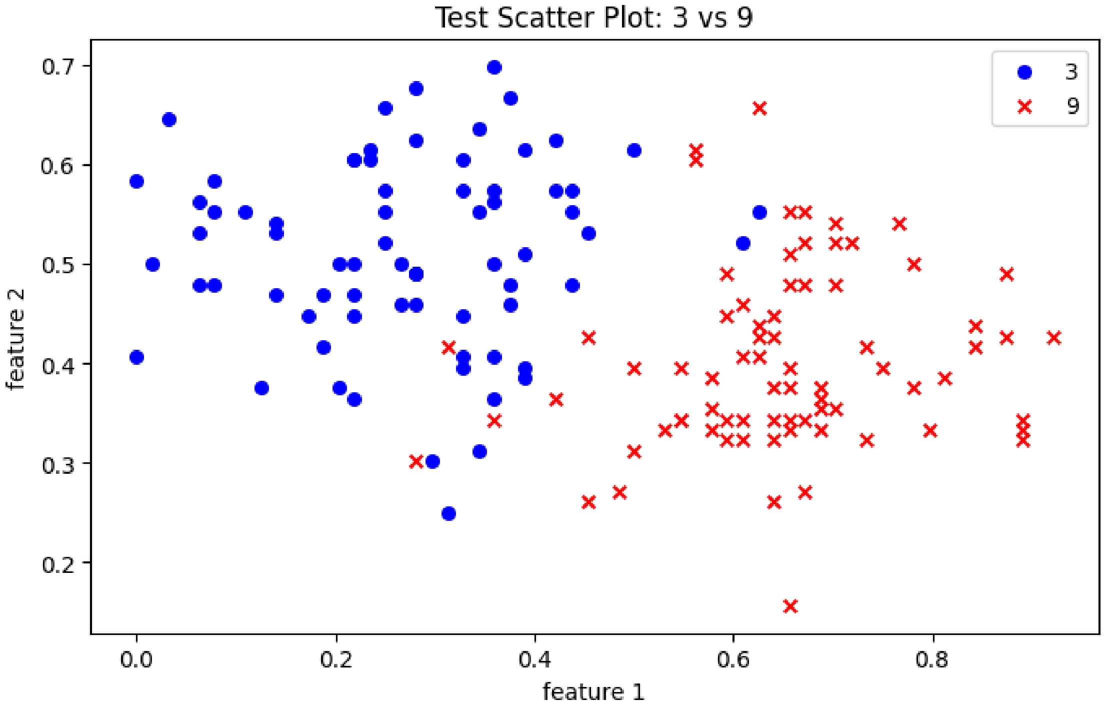
```
def scatter_plot_simple(x, y, title="Training"):
    """
    This function returns the dataset scatter plot
    input x: N x 2 numpy array
    input y: N x 1 numpy array
    output: None
    """
    plt.figure(figsize=(8,8))
    plt.gca().set_aspect('equal')
    plt.title(title + " Scatter Plot: 3 vs 9")

    # Scatter plot
    plt.scatter(x[y == -1, 0], x[y == -1, 1], marker="o", s=30, c="b", label=␣
 ↪"3")
    plt.scatter(x[y == 1, 0], x[y == 1, 1], marker="x", s=30, c="r", label= "9")

    plt.xlabel("feature 1")
    plt.ylabel("feature 2")
    plt.legend()
    plt.show()
```

[7]:

```
# Load data
digits = load_digits()

# Dimension Reduction
Xr_train, Xr_test = features2d(X_train), features2d(X_test)

# Scatter Plot
```

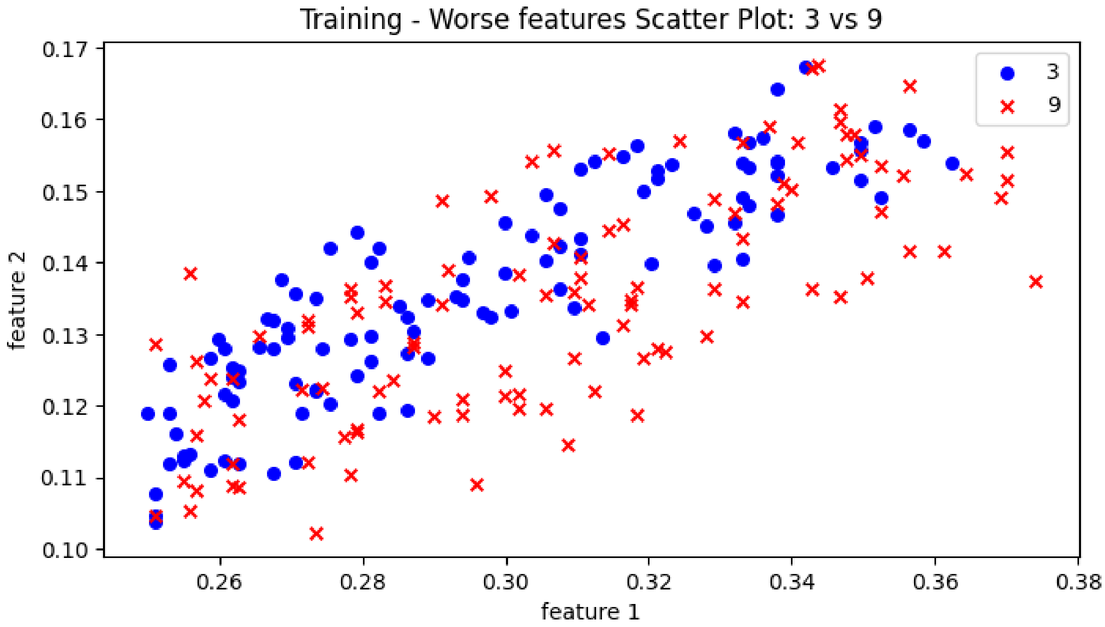
```
scatter_plot_simple(Xr_train, y_train, "Training")
scatter_plot_simple(Xr_test, y_test, "Test")
```





[8]:

```
# Dimension Reduction
_Xr_train, _Xr_test = worse_features2d(X_train), worse_features2d(X_test)

# Scatter Plot
scatter_plot_simple(_Xr_train, y_train, "Training - Worse features")
```



4.1 2.3 Decision rule

[9]:

```
# Compute the distance of a vector x from a mean point
def distance_from_mean(x, mean):
    """
    Computes the L2-distance
    """
    return np.sqrt(np.sum((x - mean[None, :])**2, axis=-1))

def nearest_mean(training_features, training_labels, test_features):
    """
    This function returns the nearest mean predictions given
    input training_features: N_training x 2 numpy array
    input training_labels: N_training x 1 numpy array
    input test_features: N_test x 2 numpy array
    output: test_predictions: N_test numpy array
    """
    classes_list = np.unique(training_labels)
    mean_points = []
    # Find all mean points
    for label in classes_list:
        mean_points.append(np.mean(training_features[training_labels == label],␣
 ↪axis=0))

    distance2mean = np.zeros((test_features.shape[0], classes_list.shape[0]))
    # Compute the distances between test_features and all mean
    for label in classes_list:
        distance2mean[:, 1*(label>0)] = distance_from_mean(test_features,␣
 ↪mean_points[1*(label>0)])

    return np.argmin(distance2mean, axis=-1)*2 - 1, mean_points
```

[10]:

```
# Dimension Reduction
Xr_test =  features2d(X_test)

# Find Nearest Mean Predictions
predicted_labels_test, mean_points = nearest_mean(Xr_train, y_train, Xr_test)
predicted_labels_train, _ = nearest_mean(Xr_train, y_train, Xr_train)

# Print error for test and training set
print("Test error Nearest Mean: ", np.mean(predicted_labels_test != y_test))
print("Training error Nearest Mean: ", np.mean(predicted_labels_train !=␣
 ↪y_train))
```

```
Test error Nearest Mean:  0.04794520547945205
Training error Nearest Mean:  0.11059907834101383
```

Notice here that the test error is lower than the training error. This usually isn´t the case, and is simply a consequence of the random samples that were drawn from the test_train_split above. If we use a different random_state, we usually will achieve a test error which is worse than the training error.

[11]:

```
def simple_linear_classifier(beta, b, test_features):
    """
    This function returns the label as sign(x_i * beta + b)
    input training_features: N_training x 2 numpy array
    input training_labels: N_training x 1 numpy array
    input test_features: N_test x 2 numpy array
    output: test_predictions: N_test numpy array
    """
    predicted_labels = np.sign(test_features @ beta + b) #@ = matrix␣
 ↪multiplication in numpy
    return predicted_labels
```

[12]:

```
# Find good beta and b values by experimentation and comparing the error on the␣
 ↪training data
beta = np.array([0.15, -0.1])
b = -0.02

# Find Nearest Mean Predictions
predicted_labels_test = simple_linear_classifier(beta, b, Xr_test)
predicted_labels_train = simple_linear_classifier(beta, b, Xr_train)

# Print error for test and training set
print("Test error Simple Linear Classifier: ", np.mean(predicted_labels_test !=␣
 ↪y_test))
print("Training error Simple Linear Classifier: ", np.
 ↪mean(predicted_labels_train != y_train))
```

```
Test error Simple Linear Classifier:  0.0547945205479452
Training error Simple Linear Classifier:  0.11059907834101383
```

4.2 2.4 Visualize the decision regions

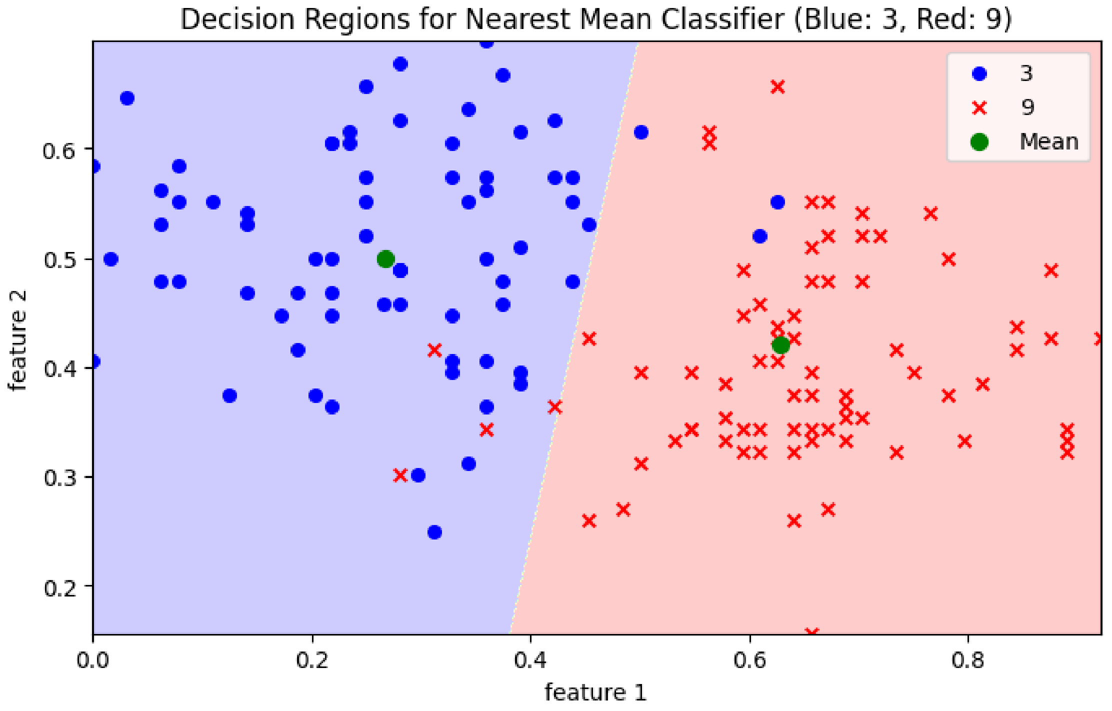
[13]:

```
def visualization_NearestMean(Xr_train, y_train, Xr_test, y_test):
    """
    This Function visualizes the decision regions for the Nearest Mean␣
 ↪Classifier
    """
    # Build Grid
    feat_min, feat_max = np.min(Xr_test, axis=0), np.max(Xr_test, axis=0)
    x, y = np.linspace(feat_min[0], feat_max[0], 2000), np.
 ↪linspace(feat_max[1], feat_min[1], 2000)
    xx = np.array(np.meshgrid(x, y)).reshape(2, -1).T  # meshgrid produces a␣
 ↪2x2 array with tuples of x and y value pairs

    # Predict grid labels
    predicted_labels, mean_points = nearest_mean(Xr_train, y_train, xx)

    # Decison boundary
    plt.figure(figsize=(8,8))
    plt.gca().set_aspect('equal')
    plt.title("Decision Regions for Nearest Mean Classifier (Blue: 3, Red: 9)")
    plt.imshow(((predicted_labels>0)*1).reshape(-1, 2000),
               cmap="prism_r", alpha=0.2, vmin=-5,
               extent=(feat_min[0], feat_max[0], feat_min[1], feat_max[1]))

    # Scatter data
    plt.scatter(Xr_test[y_test == -1, 0], Xr_test[y_test == -1, 1], marker="o",␣
 ↪s=30, c="b", label= "3")
```

```
    plt.scatter(Xr_test[y_test == 1, 0], Xr_test[y_test == 1, 1], marker="x",␣
↪s=30, c="r", label= "9")

    # Scatter mean points
    plt.scatter(mean_points[0][0], mean_points[0][1], marker="o", s=50, c="g",␣
↪label= "Mean")
    plt.scatter(mean_points[1][0], mean_points[1][1], marker="o", s=50, c="g")

    plt.xlim(feat_min[0], feat_max[0])
    plt.ylim(feat_min[1], feat_max[1])

    plt.xlabel("feature 1")
    plt.ylabel("feature 2")
    plt.legend()
    plt.show()
```

[14]:

```
visualization_NearestMean(Xr_train, y_train, Xr_test, y_test)
```



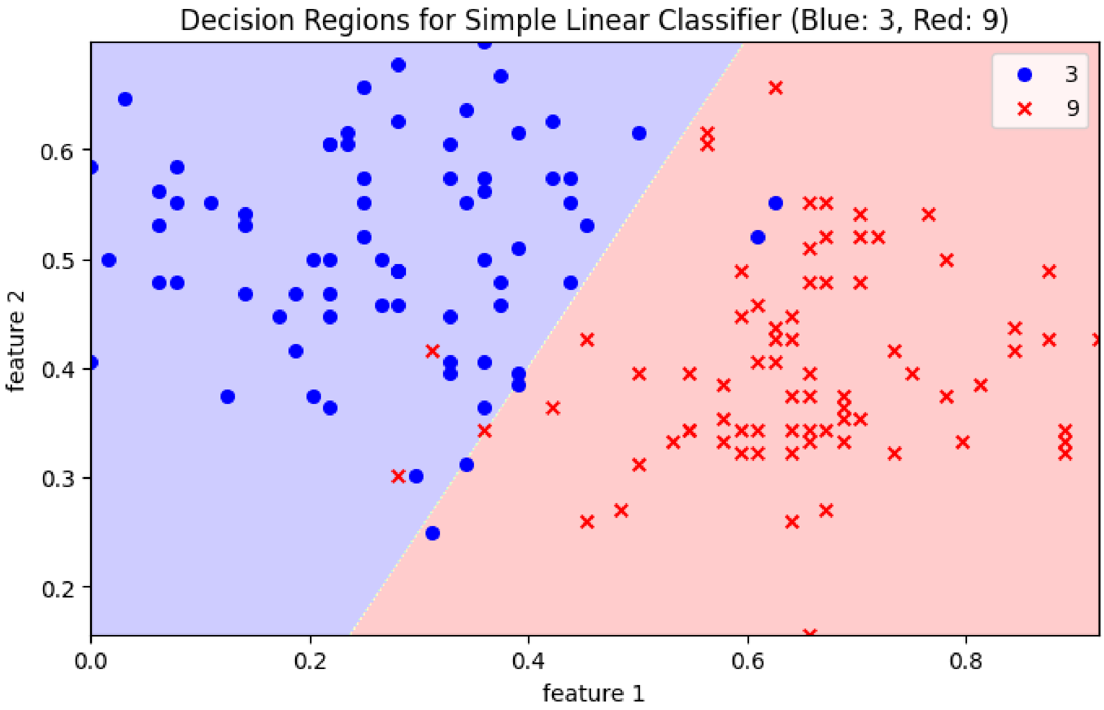
[15]:

```
def visualization_SimpleLinearClassifier(beta, b, Xr_test, y_test):
    """
    This Function visualizes the decision regions for the Simple Linear␣
↪Classifier
```

```
    """
    # Build Grid
    feat_min, feat_max = np.min(Xr_test, axis=0), np.max(Xr_test, axis=0)
    x, y = np.linspace(feat_min[0], feat_max[0], 2000), np.
↪linspace(feat_max[1], feat_min[1], 2000)
    xx = np.array(np.meshgrid(x, y)).reshape(2, -1).T

    # Predict grid labels
    predicted_labels = simple_linear_classifier(beta, b, xx)

    # Decison boundary
    plt.figure(figsize=(8,8))
    plt.gca().set_aspect('equal')
    plt.title("Decision Regions for Simple Linear Classifier (Blue: 3, Red: 9)")
    plt.imshow(((predicted_labels>0)*1).reshape(-1, 2000),
               cmap="prism_r", alpha=0.2, vmin=-5,
               extent=(feat_min[0], feat_max[0], feat_min[1], feat_max[1]))

    # Scatter data
    plt.scatter(Xr_test[y_test == -1, 0], Xr_test[y_test == -1, 1], marker="o",␣
↪s=30, c="b", label= "3")
    plt.scatter(Xr_test[y_test == 1, 0], Xr_test[y_test == 1, 1], marker="x",␣
↪s=30, c="r", label= "9")

    # Set axis limits
    plt.xlim(feat_min[0], feat_max[0])
    plt.ylim(feat_min[1], feat_max[1])

    plt.xlabel("feature 1")
    plt.ylabel("feature 2")
    plt.legend()
    plt.show()
```

[16]:

```
visualization_SimpleLinearClassifier(beta, b, Xr_test, y_test)
```



5 3 LDA

5.1 3.1 Implement LDA training

[17]:
```
def fit_lda(training_features, training_labels):
    """
    This function computes for each class: mean and priors, and the global␣
↪covariance matrix

    input training_features: N_training x 2 numpy array
    input training_labels: N_training x 1 numpy array
    output lists of means, covariances and priors
    """
    mu, cov, p = [], [], []
    for label in np.unique(training_labels):
        # filtering the correct class
        data = training_features[training_labels == label]

        # mean
        mean = np.mean(data, axis=0)
        mu.append(mean)

        # Priors
        p.append(data.shape[0]/training_features.shape[0])

    # Global Variance, subtract class means first
    global_feat = training_features - np.array(mu)[1*(training_labels>0)]
    cov = np.dot(global_feat.T, global_feat)/training_features.shape[0]

    return mu, cov, p
```

5.2 3.2 Implement LDA Prediction

[18]:

```
def predict_lda(mu, cov, p, test_features):
    """
    This function returns the LDA predictions given
    as input lists of means, priors and the global covariance matrix
    input test_features: N_test x 2 numpy array
    output: test_predictions: N_test numpy array
    """

    beta = np.linalg.inv(cov) @ (mu[1] - mu[0]).T
    b = -1/2*(mu[0] + mu[1]) @ beta + np.log(p[1]/p[0])

    predicted_labels = np.sign(test_features @ beta + b)
    return predicted_labels
```

[19]:

```
# Fit LDA
mu, cov, p = fit_lda(Xr_train, y_train)

# Find LDA Predictions
predicted_labels_test = predict_lda(mu, cov, p, Xr_test)
predicted_labels_train = predict_lda(mu, cov, p, Xr_train)

# Print error rates
print("Test error LDA: ", np.mean(predicted_labels_test != y_test))
print("Training error LDA: ", np.mean(predicted_labels_train != y_train))
```

```
Test error LDA:  0.06164383561643835
Training error LDA:  0.10599078341013825
```

Now we apply LDA to the full data i.e. we have 64 features instead of 2. As there are pixels which are always off (e.g. in the corners) the covariance matrix will have a zero determinant and it won't be possible to calculate the inverse for the prediction. Thus we first filter out the pixels which have a very low variance.

[20]:

```
# Fit LDA on full pixel vectors
vars = np.var(X_train, axis=0)
mask = (vars>0.001)
print('Number of remaining dimensions:', np.sum(mask))
```

```
x_test_clean = X_test[:, mask]
x_train_clean = X_train[:, mask]

mu_full, cov_full, p_full = fit_lda(x_train_clean, y_train)
print('Determinant of covariance matrix:', np.linalg.det(cov_full)) # The␣
 ↪determinant of the full covariance matrix has to be non-zero

# Find LDA Predictions
predicted_labels_test = predict_lda(mu_full, cov_full, p_full, x_test_clean)
predicted_labels_train = predict_lda(mu_full, cov_full, p_full, x_train_clean)

# Print error rates
print("Test error LDA: ", np.mean(predicted_labels_test != y_test))
print("Training error LDA: ", np.mean(predicted_labels_train != y_train))
```

```
Number of remaining dimensions: 47
Determinant of covariance matrix: 4.43956157663461e-81
Test error LDA:  0.0
Training error LDA:  0.004608294930875576
```

mu_full is a 2-tuple of 64-dim vectors and cov_full is a 64x64 matrix. In total we have $2 * 64 + 64 * 64 = 4224$ parameters to “train”. As our dataset only consists of 363 images (only ‘3’ and ‘9’ digits) this model can, from an information theory point of view, memorize the entire dataset and we achive a training error of nearly zero. For comparatively low-dimensional data like the 64 pixel images, it is feasible to train on the all dimensions of the input, but for higher-dimensional data and larger datasets it is usually necessary to use some kind of dimensionality reduction.

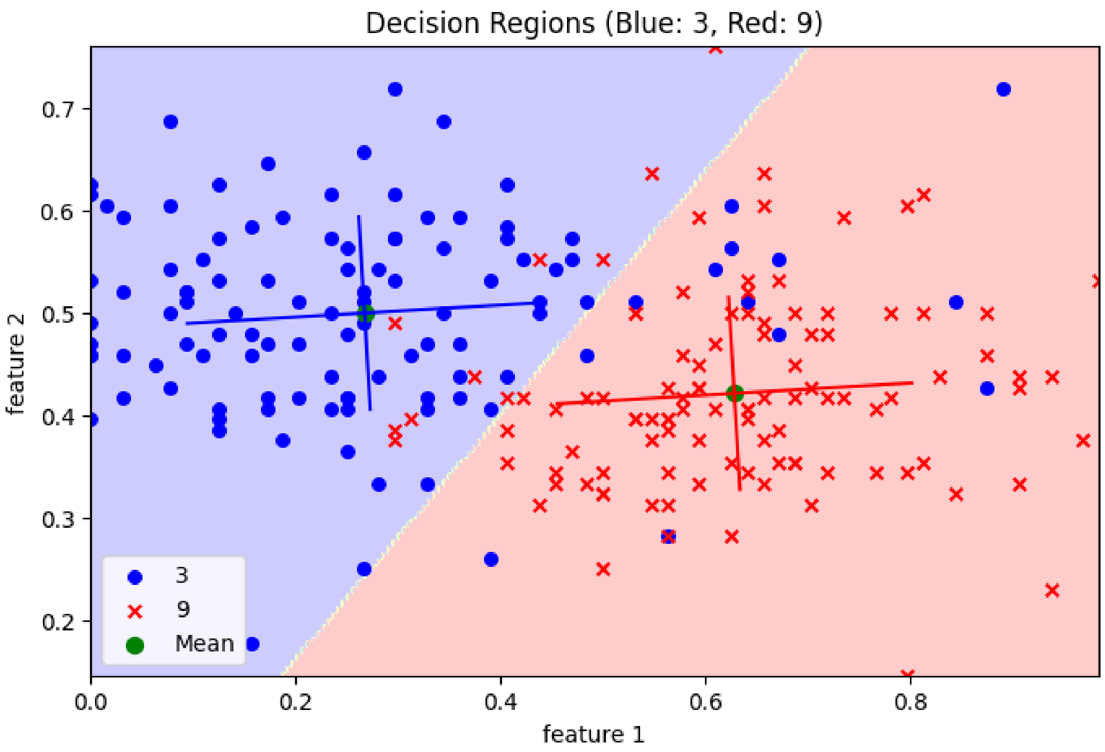
5.3 3.3 Visualization

[21]:
```
def plot_ellipse_axis(mu, cov, color="blue"):
    """
    This function plots the main axis of the distribution given mean and␣
 ↪covariance matrix
    """

    # Eigenvalues/Eigenvector decomposition
    [lamb1, lamb2], [vec_1, vec_2] = np.linalg.eig(cov)
    lamb1, lamb2 = np.sqrt(lamb1), np.sqrt(lamb2)

    # Plot axis 1
    x1, y1 = ([mu[0] - lamb1*vec_1[0], mu[0] + lamb1*vec_1[0]],
              [mu[1] - lamb1*vec_2[0], mu[1] + lamb1*vec_2[0]])
    plt.plot(x1, y1, color)

    # Plot axis 2
    x2, y2 = ([mu[0] - lamb2*vec_1[1], mu[0] + lamb2*vec_1[1]],
              [mu[1] - lamb2*vec_2[1], mu[1] + lamb2*vec_2[1]])
    plt.plot(x2, y2, color)
```

```
def visualization_LDA(Xr_test, y_test, mu, cov, p):
    """
    This Function visualizes the decision regions for the LDA Classifier
    """
    # Build Grid
    feat_min, feat_max = np.min(Xr_test, axis=0), np.max(Xr_test, axis=0)
    x, y = np.linspace(feat_min[0], feat_max[0], 200), np.linspace(feat_max[1],␣
 ↪feat_min[1], 200)
    xx = np.array(np.meshgrid(x, y)).reshape(2, -1).T

    # Predict grid labels
    predicted_labels = predict_lda(mu, cov, p, xx)

    plt.figure(figsize=(8,8))
    plt.gca().set_aspect('equal')
    plt.title("Decision Regions (Blue: 3, Red: 9)")
    plt.imshow(((predicted_labels>0)*1).reshape(-1, 200),
               cmap="prism_r", alpha=0.2, vmin=-5,
               extent=(feat_min[0], feat_max[0], feat_min[1], feat_max[1]))

    # Scatter data
    plt.scatter(Xr_test[y_test == -1, 0], Xr_test[y_test == -1, 1], marker="o",␣
 ↪s=30, c="b", label= "3")
    plt.scatter(Xr_test[y_test == 1, 0], Xr_test[y_test == 1, 1], marker="x",␣
 ↪s=30, c="r", label= "9")

    # Scatter Mean
    plt.scatter(mean_points[0][0], mean_points[0][1], marker="o", s=50, c="g",␣
 ↪label= "Mean")
    plt.scatter(mean_points[1][0], mean_points[1][1], marker="o", s=50, c="g")

    plot_ellipse_axis(mu[0], cov, color="blue")
    plot_ellipse_axis(mu[1], cov, color="red")

    plt.xlim(feat_min[0], feat_max[0])
    plt.ylim(feat_min[1], feat_max[1])

    plt.xlabel("feature 1")
    plt.ylabel("feature 2")
    plt.legend()
    plt.show()
```

[22]:

```
# Visualize training data
visualization_LDA(Xr_train, y_train, mu, cov, p)
```



Notice here that the axes of both clusters are the same, as LDA uses the same covariance matrix for each class.

5.4 3.4 Quantitative performance evaluation

[31]:
```
from sklearn.model_selection import KFold

def cross_validation(digits, fit_func, pred_func, num_sample=10):
    """
    Measure the correct accuracy with cross validation
    """
    # Get data
    data = digits["data"]
    target = digits["target"]

    # Need to prepare data again since 'data_preparation' returns test - train␣
↪split
    # Data filering
    num_1, num_2 = 3, 9
    mask = np.logical_or(target == num_1, target == num_2)
    data = data[mask]/data.max()
    target = target[mask]
    # Delete dead pixels
    vars = np.var(data, axis=0)
    mask = (vars>0.001)
    data = data[:, mask]

    # Relabel targets
    target[target == num_1] = -1
    target[target == num_2] = 1

    # Splits
    k_folds = KFold(n_splits=num_sample) # This creates a "KFold object" from␣
↪the sklearn library, with which k-fold
                                         # cross validation can easily be␣
↪performed. Check the sklearn documentation
                                         # for more info.
    mean_rate = np.zeros(num_sample)
    for i, (train, test) in enumerate(k_folds.split(data)):
        x_train, x_test = data[train], data[test]
        mu, cov, p = fit_func(x_train, target[train])
        predicted_labels = pred_func(mu, cov, p, x_test)
        mean_rate[i] = np.mean(predicted_labels != target[test])

    print("Mean Error Rate with Cross Validation: %f +/- %f"%(np.
↪mean(mean_rate), np.std(mean_rate)))
```

[32]:

```
# Digits
digits = load_digits()
# Print Cross validation accuracy
cross_validation(digits, fit_lda, predict_lda, 10)
```

```
Mean Error Rate with Cross Validation: 0.013739 +/- 0.025530
```

As k-fold cross validation performs multiple error estimations, it is less prone to statistical occurences like above, where due to the random selection of instances by chance resulted in the test error being lower than the training error, which usually is not the case statistically speaking.

6 4 SVM

[33]:

```
def predict_SVM(beta, b, test_features):
    """
    This function returns the SVM predictions given
    as input beta and b
    input test_features: N_test x 2 numpy array
    output: test_predictions: N_test numpy array
    """

    predicted_labels = np.sign(test_features @ beta + b)
    return predicted_labels
```

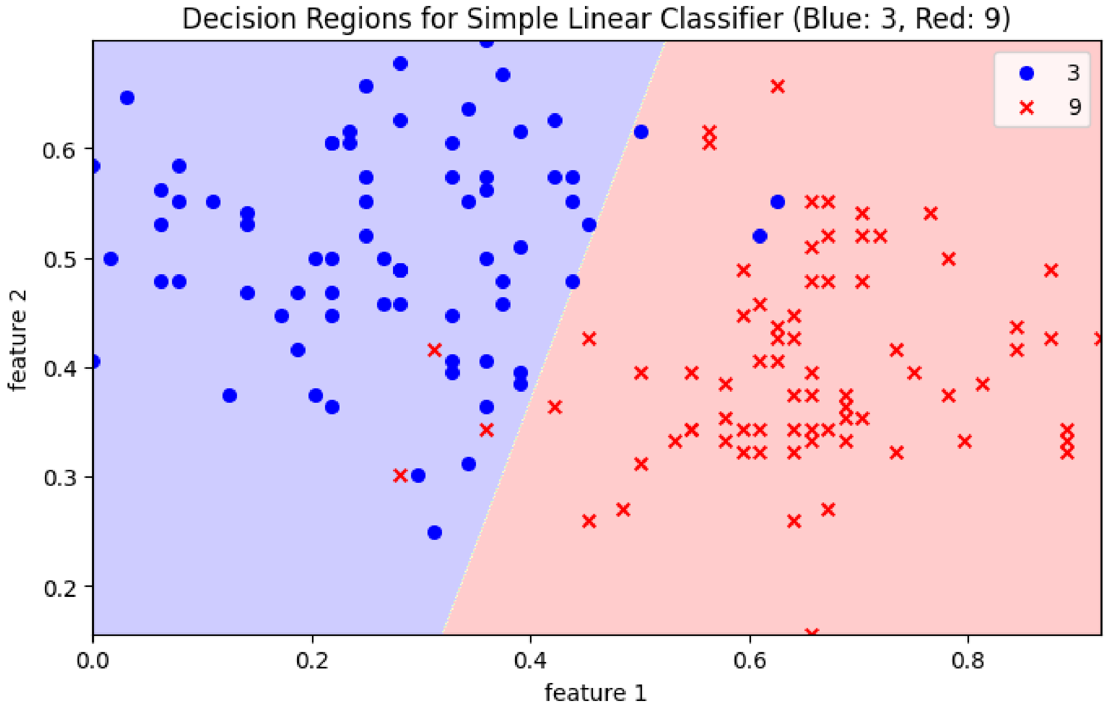
[34]:
```
def fit_SVM(training_features, training_labels, l=1, T=1000, tau=0.01):
    """
    input training_features: N_training x N_features numpy array
    input training_labels: N_training x 1 numpy array
    input l: float, T: int, tau: float
    output beta, b, List of Loss and train_error
    """

    # Initial guess for beta and b
    beta = np.random.normal(loc=0, scale=1, size=(training_features.shape[1]))
    b = 0

    def ReLU(x):
        return np.where(x>0, x, 0)

    Loss = []
    train_error = []

    N = len(training_labels)

    # Implement training loop using the gradient descent algorithm
    for i in range(T):
        # Helper variable
        temp = training_labels*(training_features @ beta + b)
        # Compute Loss
        Loss.append(1/2*np.dot(beta, beta) + l*np.mean(ReLU(1 - temp)))
        # Compute training error at each iteration with current guess of beta␣
 ↪and b
        predicted_labels_train = predict_SVM(beta, b, training_features)
        train_error.append(np.mean(predicted_labels_train != training_labels))
        # Compute gradients of beta and b
        grad_beta = beta + l*np.mean(np.where(temp[:,None] < 1,␣
 ↪-training_labels[:, None]*training_features, 0), axis=0)
        grad_b = l*np.mean(np.where(temp < 1, -training_labels, 0))
        # Update current guess of beta and b
        beta = beta - tau*grad_beta
        b = b - tau*grad_b

    return beta, b, Loss, train_error
```

[43]:
```
# Fit SVM with hand-picked hyperparameters
beta, b, Loss, train_error = fit_SVM(Xr_train, y_train, l=100, T=1000, tau=0.
 ↪001)
# Print beta and b
print('beta =', beta)
print('b =', b)
```

```
beta = [ 4.20689925 -1.58718237]
b = -1.0986175115207266
```

[44]:

```
# Visualizations for SVM
visualization_SimpleLinearClassifier(beta, b, Xr_test, y_test)
```



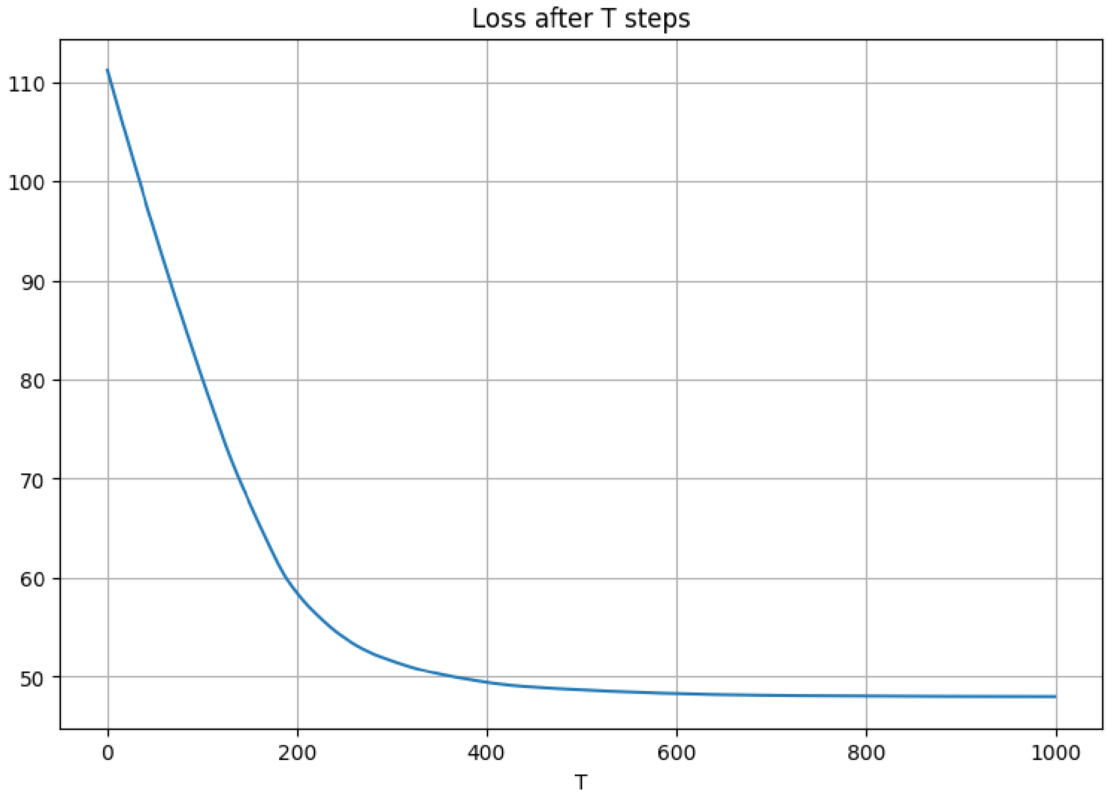
[45]:

```
# Find SVM Predictions
predicted_labels_test = predict_SVM(beta, b, Xr_test)
predicted_labels_train = predict_SVM(beta, b, Xr_train)

# Print error rates
print("Test error SVM: ", np.mean(predicted_labels_test != y_test))
print("Training error SVM: ", np.mean(predicted_labels_train != y_train))
```

```
Test error SVM:  0.0410958904109589
Training error SVM:  0.10138248847926268
```

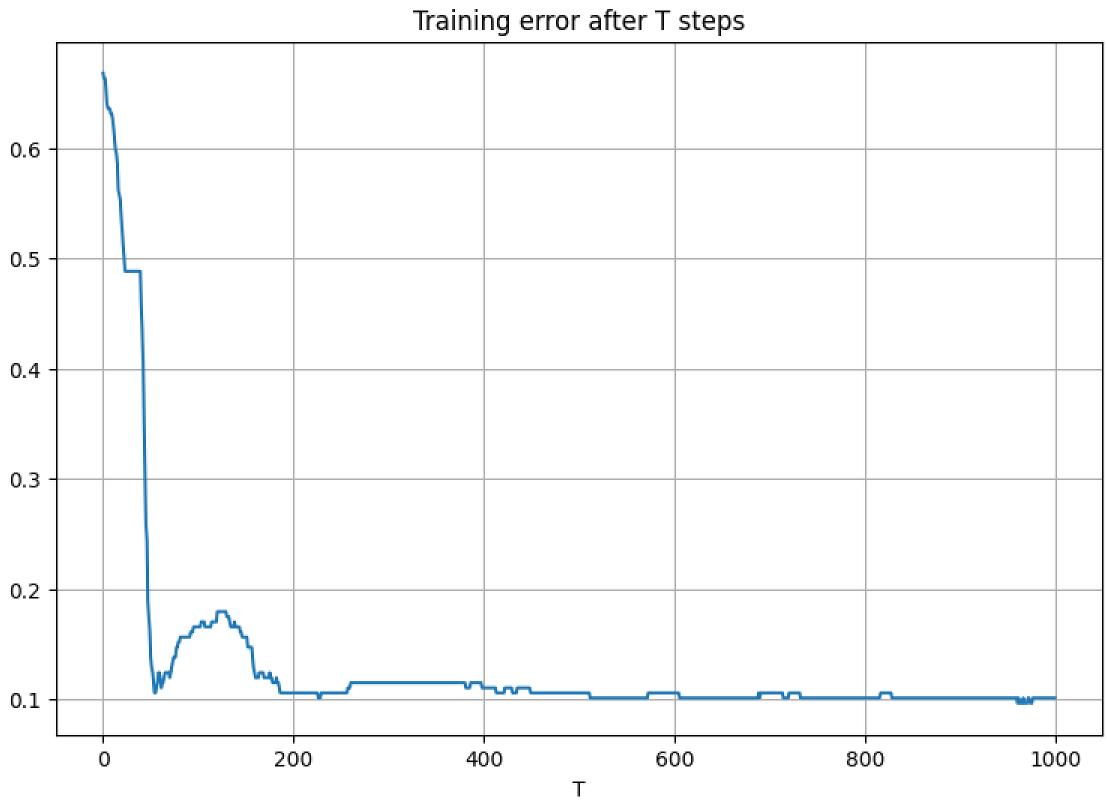
[53]:

```
plt.figure(figsize=(9,6))
plt.title('Loss after T steps')
plt.plot(Loss)
plt.xlabel('T')
plt.grid()
plt.show()
```



[54]:

```
plt.figure(figsize=(9, 6))
plt.title('Training error after T steps')
plt.plot(train_error)
plt.xlabel('T')
plt.grid()
plt.show()
```



[55]:
```
from sklearn.model_selection import KFold

def cross_validation_SVM(X_train, y_train, lambdas=[], T=1000, tau=0.001,␣
 ↪num_sample=10):
    """
    Measure the correct accuracy with cross validation for a given list of␣
 ↪hyperparameters (lambdas)
    """
    # Iterate through all parameters
    for l in lambdas:
        # Splits
        k_folds = KFold(n_splits=num_sample)

        mean_rate = np.zeros(num_sample)
        for i, (train, test) in enumerate(k_folds.split(X_train)):

            beta, b, _, _ = fit_SVM(X_train[train], y_train[train], l=l, T=T,␣
 ↪tau=tau)
            predicted_labels = predict_SVM(beta, b, X_train[test])

            mean_rate[i] = np.mean(predicted_labels != y_train[test])
        print("Mean Error Rate with Cross Validation for lambda=%f: %f +/-␣
 ↪%f"%(l,np.mean(mean_rate), np.std(mean_rate)))
```

[56]:

```
cross_validation_SVM(Xr_train, y_train, lambdas=[1,10,100,1e3,1e4,1e5], T=1000,␣
 ↪tau=0.001, num_sample=10)
```

```
Mean Error Rate with Cross Validation for lambda=1.000000: 0.567316 +/- 0.128417
Mean Error Rate with Cross Validation for lambda=10.000000: 0.362771 +/-
0.131922
Mean Error Rate with Cross Validation for lambda=100.000000: 0.115584 +/-
0.056693
Mean Error Rate with Cross Validation for lambda=1000.000000: 0.115152 +/-
0.056150
Mean Error Rate with Cross Validation for lambda=10000.000000: 0.151515 +/-
0.079157
Mean Error Rate with Cross Validation for lambda=100000.000000: 0.197619 +/-
0.137159
```

Best results are achived with $\lambda = 100$ or 1000.

And now once again with the full pixel vector.

[57]:

```
# Fit SVM on full pixel data
beta, b, Loss, train_error = fit_SVM(X_train, y_train, l=100, T=1000, tau=0.001)

# Find SVM Predictions
predicted_labels_test = predict_SVM(beta, b, X_test)
predicted_labels_train = predict_SVM(beta, b, X_train)

# Print error rates
print("Test error SVM: ", np.mean(predicted_labels_test != y_test))
print("Training error SVM: ", np.mean(predicted_labels_train != y_train))
```

```
Test error SVM:  0.0
Training error SVM:  0.009216589861751152
```

[58]:

```
cross_validation_SVM(X_train, y_train, lambdas=[1,10,100,1e3,1e4,1e5], T=1000,␣
 ↪tau=0.001, num_sample=10)
```

```
Mean Error Rate with Cross Validation for lambda=1.000000: 0.408009 +/- 0.236574
Mean Error Rate with Cross Validation for lambda=10.000000: 0.041558 +/-
0.038377
Mean Error Rate with Cross Validation for lambda=100.000000: 0.013853 +/-
0.021168
Mean Error Rate with Cross Validation for lambda=1000.000000: 0.022944 +/-
0.022952
Mean Error Rate with Cross Validation for lambda=10000.000000: 0.018398 +/-
0.022541
```

```
Mean Error Rate with Cross Validation for lambda=100000.000000: 0.022944 +/-
0.022952
```