exercise4

January 28, 2021

1 EXERCISE 4 - ML - Grundverfahren

Package notes: We will use different packages in this exersice: 1. Scipy: We will use scipy for optimizing the hinge loss. Scipy provides a numerical optimization package with various solvers such as L-BFGS-B or SLSQP. 2. Sklearn: Sklearn is a package providing different machine learning algorithms and tools. We will not use it for machine learning algorithms here but for loading the handwritten image data set, which we will use for applying probabilistic PCA. 3. CVXPY: CVXPY can be used to solve convex optimization problems such as a quadratic program. We will use the solver for optimizing for the dual problem of the SVMs.

You can install all those packages using pip (or conda or whatever).

[91]:
```python
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
from typing import Union, Optional
```

1.1 1.) Probabilistic PCA with Expectation Maximization (7 Points)

In this exercise we will implement probabilistic PCA as discussed in the lecture. We will apply it on a toy task and the handwritten digit data set. We will also generate our own images.

We start by defining some utilities for plotting. You don't need to do anything here.

[92]:
```python
def plot_data(X):
    plt.scatter(X[:, 0], X[:, 1], color='b')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.xlim(0, 7)
    plt.ylim(0, 7)

def draw_2d_gaussian(mu: np.ndarray, sigma: np.ndarray, plt_std: float = 2, *args, **kwargs) -> None:
    (largest_eigval, smallest_eigval), eigvec = np.linalg.eig(sigma)
    phi = -np.arctan2(eigvec[0, 1], eigvec[0, 0])

    plt.scatter(mu[0:1], mu[1:2], marker="x", *args, **kwargs)
    a = plt_std * np.sqrt(largest_eigval)
    b = plt_std * np.sqrt(smallest_eigval)

    ellipse_x_r = a * np.cos(np.linspace(0, 2 * np.pi, num=200))
    ellipse_y_r = b * np.sin(np.linspace(0, 2 * np.pi, num=200))

    R = np.array([[np.cos(phi), np.sin(phi)], [-np.sin(phi), np.cos(phi)]])
    r_ellipse = np.array([ellipse_x_r, ellipse_y_r]).T @ R
    plt.plot(mu[0] + r_ellipse[:, 0], mu[1] + r_ellipse[:, 1], *args, **kwargs)


def plot_ev(mu, eig_vec_1, eig_vec_2):
    arrow_1_end = mu + eig_vec_1
    arrow_1_x = [mu[0], arrow_1_end[0]]
    arrow_1_y = [mu[1], arrow_1_end[1]]

    arrow_2_end = mu + eig_vec_2
    arrow_2_x = [mu[0], arrow_2_end[0]]
    arrow_2_y = [mu[1], arrow_2_end[1]]

    plt.plot(mu[0], mu[1], 'xr')
    plt.plot((mu + eig_vec_1)[0], (mu + eig_vec_1)[1], 'xr')
    plt.plot(arrow_1_x, arrow_1_y, 'red')
    plt.plot(arrow_2_x, arrow_2_y, 'red')
```

1.1.1 Exercise 1.1) E-Step in Probabilistic PCA (3 Points)

We will implement the E-step in this exercise. Remember the equations in the E-Step as:

$$\boldsymbol{\mu}_{\boldsymbol{z}|\boldsymbol{x}_i} = (\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{W} + \sigma^2\boldsymbol{I})^{-1}\boldsymbol{W}^T(\boldsymbol{x}_i - \boldsymbol{\mu})$$

$$\boldsymbol{\Sigma}_{\boldsymbol{z}|\boldsymbol{x_i}} = \sigma^2(\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{W} + \sigma^2\boldsymbol{I})^{-1},$$

where $\boldsymbol{x}_i$ is one sample of the data, $\boldsymbol{W}$ is the transformation matrix, σ^2 is the variance and $\boldsymbol{\mu}$ is the mean of the likelihood model. Please note that we need to subtract the likelihood mean from the data. This subtraction previously was missing in the slides and we uploaded a corrected version. In the video recording you will also face that it is missing. However, the formula stated here is the one you should use. Implement the E-step of the EM-Algorithm for dimensionality reduction, according to the equations stated. The dimensions of the vectors/matrices are stated in the code snippet. Make sure that you have the same dimensionality as stated in the comments. The hints in the comments might be useful.

[93]:
```
def e_step(W, mu, X, sigma_quad):
    """
    Computes/Samples the Latent vectors in matrix Z given transformation matrix␣
 ↪W and data X.
```

```
    :param W: Transformation matrix W (shape: [DxM], where D is data dimension, M is latent Dimension)
    :param X: Data matrix containing the data (shape: [NxD])
    :param sigma_quad: sigma^2, the variance of the likelihood (already in quadratic form) (shape: float)
    :return: returns mu_z, the mean of the posterior for each sample x (shape: [NxM])
             returns z_samples, the latent variables (shape: [MxN])
             returns var_z, the covariance of the posterior (shape: [MxM])
    """

    # Hint: np.linalg.solve is useful. You could also use np.linalg.inv. But np.linalg.solve is in general prefered

    # compute mean of z -> NxM
    mu_z = np.linalg.solve(W.T @ W + sigma_quad * np.identity(W.shape[1]) , W.T @ (X.T - mu.reshape(-1,1)) ).T
    # compute covariance of z -> MxM
    var_z = np.linalg.solve(W.T @ W + sigma_quad * np.identity(W.shape[1]) , sigma_quad * np.identity(W.shape[1]))

    # sample z for each mean (mu_z is a Matrix (NxM), containg a mean for each data x_i)
    #z_samples = np.array([np.random.multivariate_normal(mu , var_z) for mu in mu_z]).reshape(mu_z.shape[0],var_z.shape[0])
    z_samples = mu_z + (np.dot(np.linalg.cholesky(var_z+(0.00001*np.identity(var_z.shape[0]))),\
                               np.random.normal(loc=0,scale=1,size=var_z.shape[0]*X.shape[0]).reshape(var_z.shape[0],X.shape[0]))).T
    return mu_z, z_samples, var_z
```

1.1.2 Exercise 1.2) M-Step in Probabilistic PCA (4Points)

We will implement the M-step in this exercise. The following equations can also be looked up in the slides

$$\begin{pmatrix} \boldsymbol{\mu} \\ \boldsymbol{W} \end{pmatrix} = (\boldsymbol{Z}^{\boldsymbol{T}}\boldsymbol{Z})^{-1}\boldsymbol{Z}^{T}\boldsymbol{X},$$

where

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_n^T \end{pmatrix}, \boldsymbol{Z} = \begin{pmatrix} 1, \boldsymbol{z}_1^T \\ \vdots \\ 1, \boldsymbol{z}_n^T \end{pmatrix}.$$

$\boldsymbol{Z}$ is the matrix containing the bias and all the latent variable samples $\boldsymbol{z}_i$ and $\boldsymbol{X}$ is the matrix containing all data points $\boldsymbol{x}$. We further need to implement the variance:

$$\sigma^2 = \frac{1}{ND} \sum_{i=1}^{N} \sum_{k=1}^{D} (y_{ik} - x_{ik})^2,$$

where $\boldsymbol{y}_i = \boldsymbol{W}\boldsymbol{z}_i + \boldsymbol{\mu}$ and N is the number of data points and D is the dimension of the data $\boldsymbol{x}$. Implement the M-step of the EM-Algorithm for dimensionality reduction, according to the equations stated. The dimensions of the vectors/matrices are stated in the code snippet. Make sure that you have the same dimensionality as stated in the comments. The hints in the comments might be useful.

[94]:

```
def m_step(z_samples, X):
    """
    Computes the variance and the transformation matrix W given the latent␣
↪vectors in z_samples and the data
    in matrix X.
    :param z_samples: The latent variable vectors stored in z_samples (shape:␣
↪[NxM])
    :param X: Data matrix containg the data (shape: [NxD])
    :return: returns the variance sigma_quad and the transformation matrix W␣
↪(shape: [DxM])
    """

    # Hint: np.linalg.solve is useful. You could also use np.linalg.inv. But np.
↪linalg.solve is in general prefered

    # create feature matrix Z
    Z = np.concatenate((np.ones([X.shape[0],1]),z_samples),axis=1)
    # Calculate W_tilde (Dx(M+1)) containg the mean of the likelihood and the␣
↪projection matrix W
    W_tilde = np.linalg.solve(Z.T @ Z , Z.T @ X).T
    mu = W_tilde[:,0]
    W = W_tilde[:,1:]
    # Perform the predictions y in matrix Y
    Y = Z @ W_tilde.T

    # calculate variance sigma_quad scalar
    sigma_quad = np.sum(np.square(Y-X))/(X.shape[0]*X.shape[1])
    return sigma_quad, mu, W
```

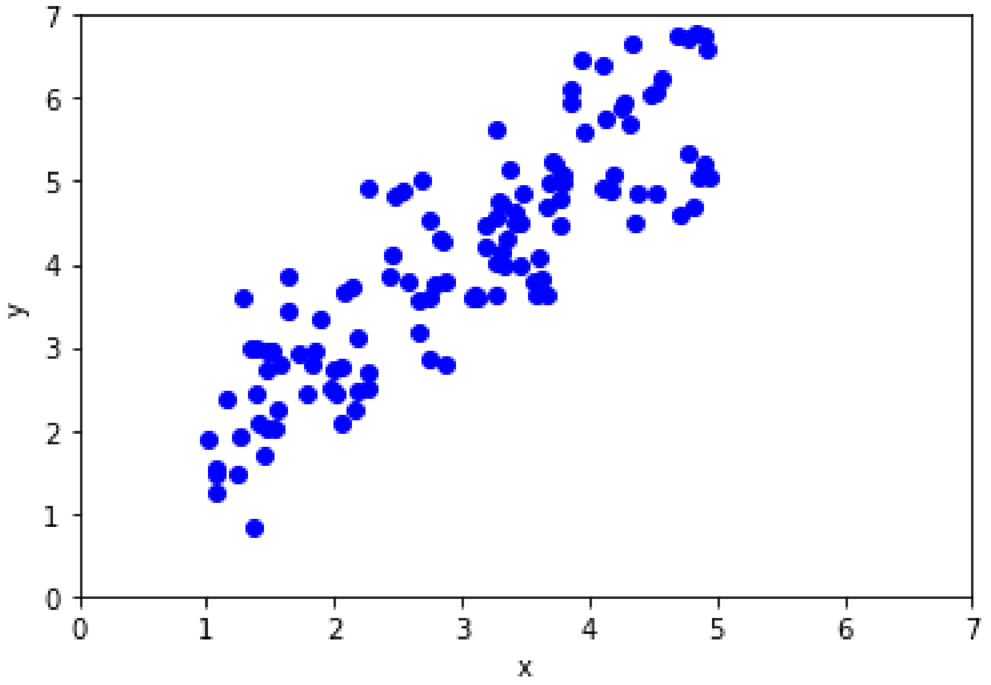
This is the EM-loop, where the E-step and the M-step alternates. You do not need to implement or change the function here.

[95]:

```
def do_ppca(X: np.ndarray, n_principle_comps: int, num_iters: int = 50):
    np.random.seed(0)
    W = np.random.normal(size=(X.shape[1], n_principle_comps))
    mu_X = np.mean(X, axis=0)
    mu = mu_X.copy()
    sigma_quad = 1
    for i in range(num_iters):
        mu_z, z_samples, var_z = e_step(W, mu, X, sigma_quad)
        sigma_quad, mu, W = m_step(z_samples, X)
    return W, z_samples, var_z, sigma_quad, mu
```

2D Toy Task from Lecture Notebook We will first apply pPCA on the toy task, which we also had in the lecture notebook. Here is the data:

[96]:
```
np.random.seed(0)

x = np.random.uniform(1,5, size=(120, 1))
y = x + 1 + np.random.normal(0, 0.7, size=x.shape)

X = np.concatenate((x, y), axis = 1)
plot_data(X)
```

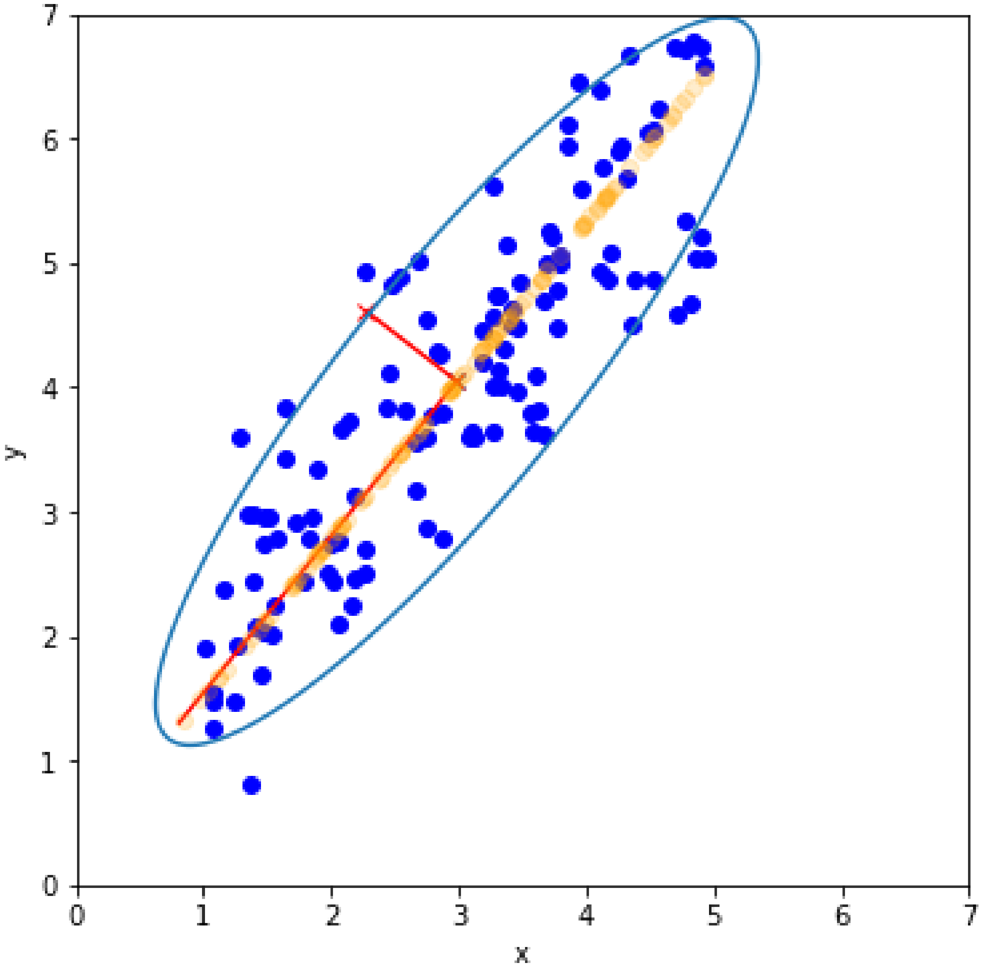


Let's now perform the algorithm on the data. You do not need to change anything.

[97]:
```
plt.figure(figsize=(6,6))

plot_data(X)

W, z_samples, var_z, sigma_quad, mu = do_ppca(X, n_principle_comps=1)
```

```
x_tilde = np.dot(W, z_samples.T).T + mu                         # reproject to␣
↪high-dim space

C = np.dot(W, W.T) + sigma_quad*np.eye(W.shape[0])       # covariance of p(x)␣
↪(reconstructed)

v, U = np.linalg.eig(np.cov(X.T))
mu_X = np.mean(X, axis=0)
plot_ev(mu_X, 2*np.sqrt(v[0])*U[:, 0], 2*np.sqrt(v[1])*U[:, 1])

draw_2d_gaussian(mu_X, C)

plt.plot(x_tilde[:, 0], x_tilde[:, 1], 'o', color='orange', alpha=0.2)    #␣
↪reprojected data points
```

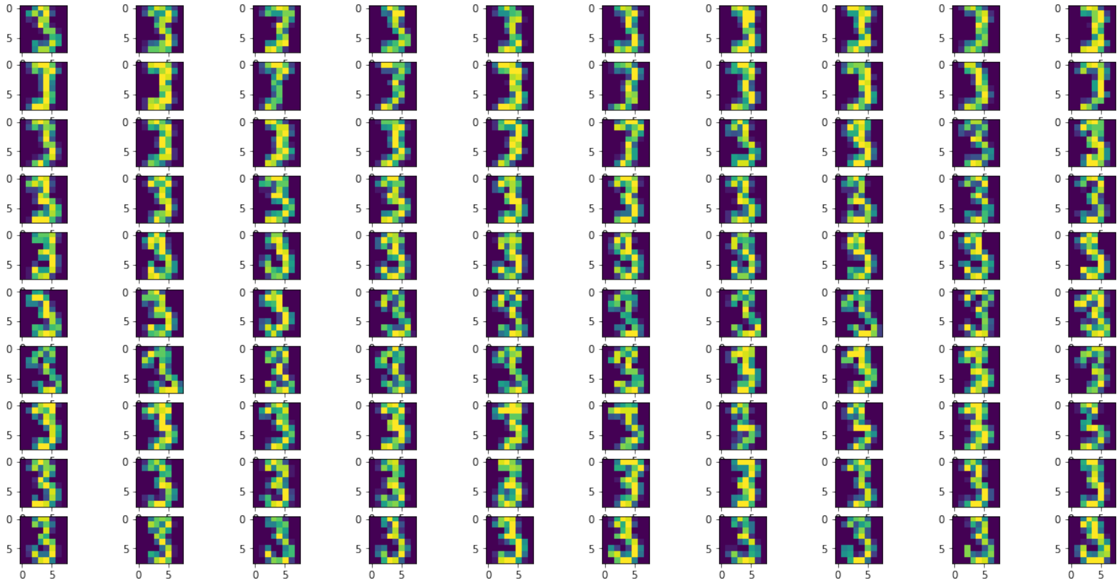
```
[97]: [<matplotlib.lines.Line2D at 0x21bb2290ca0>]
```



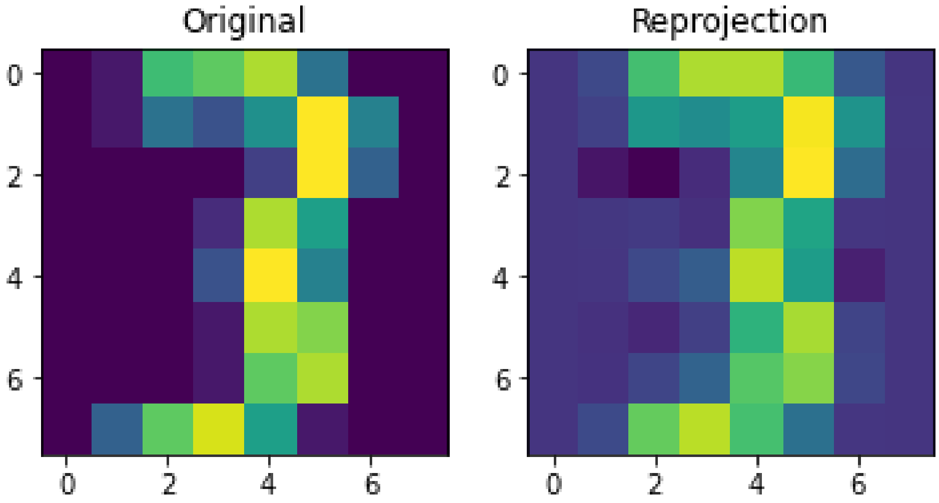
Hand-Written digits from Lecture Notebook Next, we apply pPCA on the handwritten digits data set. We will consider the digit 3 only. Here is how the data looks like

[98]:
```python
from sklearn.datasets import load_digits

digits = load_digits()
targets = digits.target

# get the images for digit 3 only
digits_3_indx = np.where(targets == 3)[0]
digit_3_data = digits.data[digits_3_indx]        # shape: (183, 64)  -> (8 x 8)
digit_3_targets = digits.target[digits_3_indx]        # only needed to verify that we load digit 3

mu_X_im = np.mean(digit_3_data, axis=0)

#Plot the original digit 3 images
plt.figure()
fig, axes = plt.subplots(10, 10, figsize=(20, 10))
for i, ax in enumerate(axes.flat):
    ax.imshow(digit_3_data[i].reshape(8, 8))
```

```
<Figure size 432x288 with 0 Axes>
```



[99]:
```
# let's perform ppca on the data
n_principle_comps = 10
W_im, z_samples_im, var_z_im, sigma_quad_im, mu_im = do_ppca(digit_3_data,␣
 ↪n_principle_comps=n_principle_comps)
x_tilde_im = np.dot(W_im, z_samples_im.T).T + mu_im

considered_im = digit_3_data[15]
considered_im_x_tilde = x_tilde_im[15, :]

plt.figure()
plt.subplot(121)
plt.title('Original')
plt.imshow(considered_im.reshape(8, 8))

plt.subplot(122)
plt.title('Reprojection')
plt.imshow(considered_im_x_tilde.reshape(8,8))
plt.show()
```



Although the reprojected data does not look same, you should definitely see the similarity to the original image.

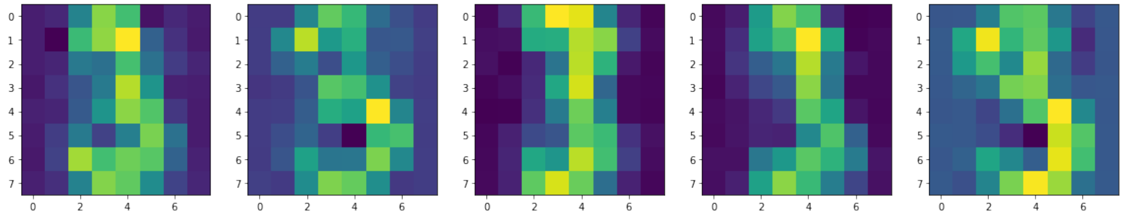
1.1.3 Random Image generation

One advantage of pPCA is that we can generate random images. The generative process, as described in the lecture is implemented here.

[100]:
```
# Sample some vectors z
z = np.random.normal(size=(5, n_principle_comps))
```

```
# Project back to D-dim space
y = np.dot(W_im, z.T).T + mu_im

# Sample noise
eps = np.random.normal(scale=sigma_quad, size=y.shape)
# Get image
x = y + eps

plt.figure('Sampled Image')
fig, axes = plt.subplots(1, 5, figsize=(20, 10))
for i, ax in enumerate(axes.flat):
     ax.imshow(x[i].reshape(8, 8))
```

```
<Figure size 432x288 with 0 Axes>
```



1.2 2.) Feature-Based Support Vector Machine (Hinge Loss) (5 Points)

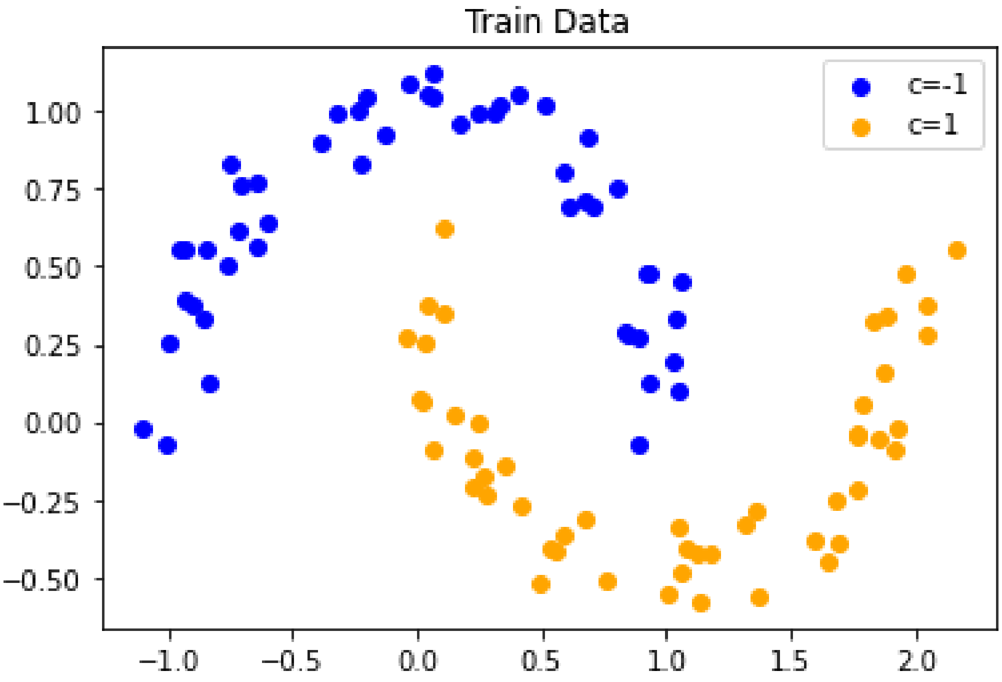
In this exercise we will train a feature-based SVM on the two moons dataset using the hinge loss. We start by loading and visualizing the data. We will use the l-bfgs-b algorithm, provided by scipy.optimize for the optimization. All you need to know about this optimizer is that it is gradient based. Otherwise you can treat it as a black-box. Yet, it's also worth a closer look if you are interested.

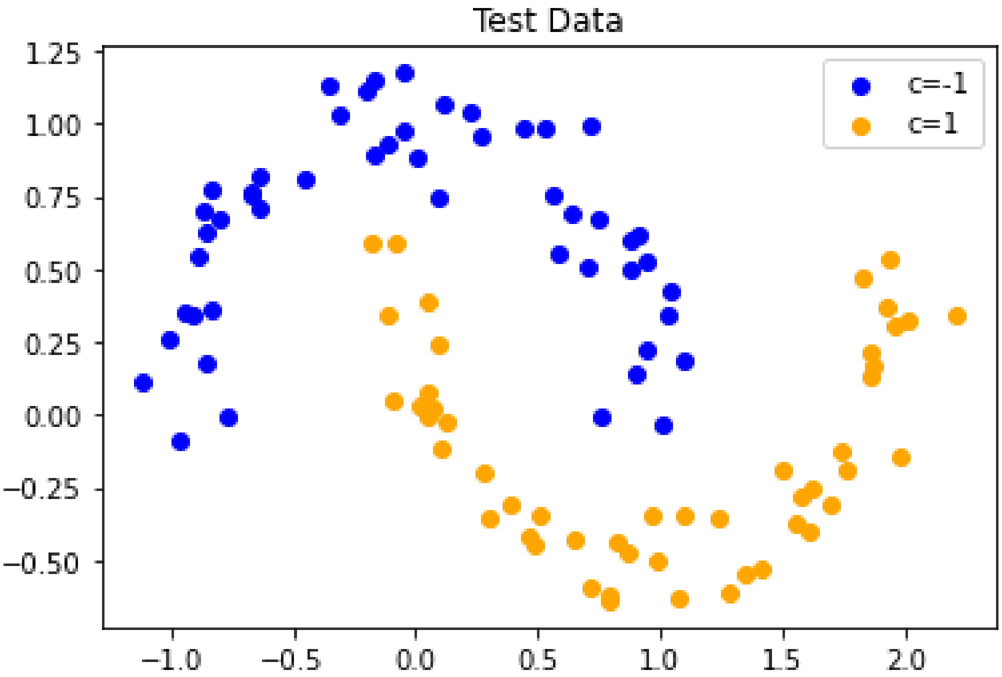
[101]:

```
import scipy.optimize as opt

train_data = dict(np.load("two_moons.npz", allow_pickle=True))
train_samples = train_data["samples"]
train_labels = train_data["labels"]
# we need to change the labels for class 0 to -1 to account for the different␣
 ↪labels used by an SVM
train_labels[train_labels == 0] = -1

test_data = dict(np.load("two_moons_test.npz", allow_pickle=True))
test_samples = test_data["samples"]
test_labels = test_data["labels"]
# we need to change the labels for class 0 to -1 to account for the different␣
 ↪labels used by an SVM
test_labels[test_labels == 0] = -1
```

```
plt.figure()
plt.title("Train Data")
plt.scatter(x=train_samples[train_labels == -1, 0],␣
 ↪y=train_samples[train_labels == -1, 1], label="c=-1", c="blue")
plt.scatter(x=train_samples[train_labels == 1, 0], y=train_samples[train_labels␣
 ↪== 1, 1], label="c=1", c="orange")
plt.legend()

plt.figure()
plt.title("Test Data")
plt.scatter(x=test_samples[test_labels == -1, 0], y=test_samples[test_labels ==␣
 ↪-1, 1], label="c=-1", c="blue")
plt.scatter(x=test_samples[test_labels == 1, 0], y=test_samples[test_labels ==␣
 ↪1, 1], label="c=1", c="orange")
plt.legend()
```

```
[101]: <matplotlib.legend.Legend at 0x21bb4bdaaf0>
```





Feature Function From the logistic classification exercise earlier we already know that cubic features are a good choice for the two moons, so we will reuse them here.

```
[102]: def cubic_feature_fn(samples: np.ndarray) -> np.ndarray:
           """
           :param x: Batch of 2D data vectors [x, y] [N x dim]
           :return cubic features: [x**3, y**3, x**2 * y, x * y**2, x**2, y**2, x*y,␣
        ↪x, y, 1]
           """
           x = samples[..., 0]
           y = samples[..., 1]
           return np.stack([x**3, y**3, x**2 * y, x * y**2, x**2, y**2, x*y, x, y, np.
        ↪ones(x.shape[0])], axis=-1)
```

1.2.1 Exercise 2.1) Hinge Loss Objective (2 Points)

We will implement the hinge loss objective in this exercise. Its given by

$$\mathcal{L}_{\boldsymbol{X},\boldsymbol{y}}(w) = \| \boldsymbol{w} \|^2 + C \sum_{i}^{N} \max\left(0, 1 - y_i \boldsymbol{w}^T \phi(\boldsymbol{x}_i)\right),$$

where $\boldsymbol{w}$ are our model parameters, $\phi(\boldsymbol{x})$ are our features (here cubic) and the $y_i \in \{-1, 1\}$ are the class labels.

Fill in the code snippets below. This function implements the hinge loss.

```
[103]: def objective_svm(weights: np.ndarray, features: np.ndarray, labels: np.
        ↪ndarray, slack_regularizer: float) -> float:
           """
           objective for svm training with hinge loss
           :param weights: current weights to evaluate (shape: [feature_dim])
           :param features: features of training samples (shape:[N x feature_dim])
           :param labels: class labels corresponding to train samples (shape: [N])
           :param slack_regularizer: Factor to weight the violation of margin with (C␣
        ↪in slides)
           :returns svm (hinge) objective (scalar)
           """
           n = features.shape[0]
           ind_loss = np.zeros(n)

           for i in range(n):
               ind_loss[i]  = np.maximum(0.0, 1.0 - labels[i] *  (weights @␣
        ↪features[i,:]))

           return np.linalg.norm(weights)**2 + slack_regularizer * np.sum(ind_loss)
```

1.2.2 Exercise 2.2) Hinge Loss Gradient (3 Points)

Derive and implement the gradient for the hinge loss objective stated above. For all non-differentiable points in the loss courves you can use any valid subgradient.

NOTE: The derivation is explicitly part of the grading, so state it in the solution file, not just implement it.

Proof

$$\frac{\partial \mathcal{L}_{\boldsymbol{X},\boldsymbol{y}}(\boldsymbol{w})}{\partial \boldsymbol{w}} = 2\boldsymbol{w} + C\sum_{i}^{N} \begin{cases} 0, & \text{if } y_i\boldsymbol{w}^T\phi(\boldsymbol{x}_i) \geq 1 \\ -y_i\phi(\boldsymbol{x}_i), & \text{otherwise} \end{cases}$$

```
[104]: def d_objective_svm(weights: np.ndarray, features: np.ndarray, labels: np.
        ↪ndarray, slack_regularizer: float) -> np.ndarray:
           """
           gradient of objective for svm training with hinge loss
           :param weights: current weights to evaluate (shape: [feature_dim])
           :param features: features of training samples (shape: [N x feature_dim])
           :param labels: class labels corresponding to train samples (shape: [N])
           :param slack_regularizer: Factor to weight the violation of margin with (C␣
        ↪in slides)
           :returns gradient of svm objective (shape: [feature_dim])
           """
           n = features.shape[0]

           # calculate mask for if statement
```

```
    mask = np.zeros((n,), dtype=bool)
    for i in range(n):
        mask[i] = features[i,:] @ weights  * labels[i] < 1

    # apply formula from above
    cond_sum = np.sum((-labels[:,None] * features)[mask], axis=0)
    return 2 * weights + slack_regularizer * cond_sum
```

1.2.3 Train and Evaluate

Finally, we can tie everything together and get our maximum margin classifier. Here we are using the L-BFGS-B optimizer provided by Scipy. With $C = 1000$ you should end up at a train accuracy of 1 and a test accuracy of 0.99, but feel free to play arround with C.

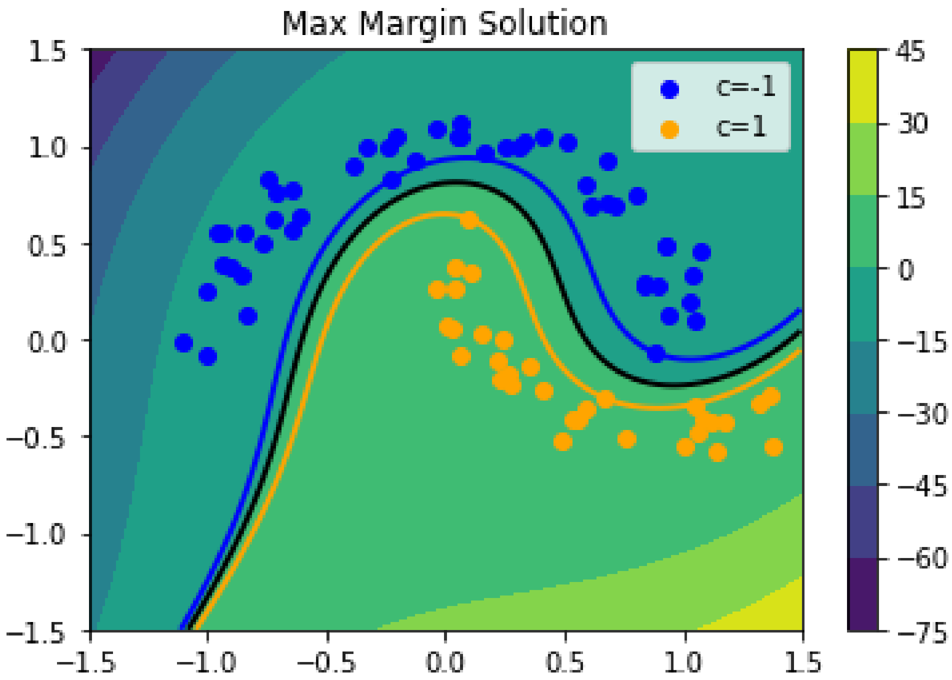
[105]:

```
feature_fn = cubic_feature_fn
C = 1000.0

# optimization

train_features = feature_fn(train_samples)

# For detail see: https://docs.scipy.org/doc/scipy/reference/optimize.
 ↪minimize-lbfgsb.html
res = opt.minimize(
    # pass objective
    fun=lambda w: objective_svm(w, train_features, train_labels, C),
    # pass initial point
    x0=np.ones(train_features.shape[-1]),
    # pass function to evaluate gradient (in scipy.opt "jac" for jacobian)
    jac=lambda w: d_objective_svm(w, train_features, train_labels, C),
    # specify method to use
    method="l-bfgs-b")

print(res)
w_svm = res.x

# evaluation
test_predictions = feature_fn(test_samples) @ w_svm
train_predictions = feature_fn(train_samples) @ w_svm

predicted_train_labels = np.ones(train_predictions.shape)
predicted_train_labels[train_predictions < 0] = -1
print("Train Accuracy: ", np.count_nonzero(predicted_train_labels ==␣
 ↪train_labels) / len(train_labels))

predicted_test_labels = np.ones(test_predictions.shape)
predicted_test_labels[test_predictions < 0] = -1
```

```
print("Test Accuracy: ", np.count_nonzero(predicted_test_labels == test_labels)␣
 ↪/ len(test_labels))

# plot train, contour, decision boundary and margins
plt.figure()
plt.title("Max Margin Solution")
plt_range = np.arange(-1.5, 1.5, 0.01)
plt_grid = np.stack(np.meshgrid(plt_range, plt_range), axis=-1)
flat_plt_grid = np.reshape(plt_grid, [-1, 2])
plt_grid_shape = plt_grid.shape[:2]

pred_grid = np.reshape(feature_fn(flat_plt_grid) @ w_svm, plt_grid_shape)

#plt.contour(plt_grid[..., 0], plt_grid[..., 1], pred_grid, levels=[-1.0, 0.0,␣
 ↪1.0], colors=["blue", "black", "orange"])
plt.contour(plt_grid[..., 0], plt_grid[..., 1], pred_grid, levels=[-1, 0, 1],␣
 ↪colors=('blue', 'black', 'orange'),
            linestyles=('-',), linewidths=(2,))
plt.contourf(plt_grid[..., 0], plt_grid[..., 1], pred_grid, levels=10)

plt.colorbar()

s0 =plt.scatter(x=train_samples[train_labels == -1, 0],␣
 ↪y=train_samples[train_labels == -1, 1], label="c=-1", c="blue")
s1 =plt.scatter(x=train_samples[train_labels == 1, 0],␣
 ↪y=train_samples[train_labels == 1, 1], label="c=1", c="orange")
plt.legend()

plt.xlim(-1.5, 1.5)
plt.ylim(-1.5, 1.5)
plt.show()
```

```
      fun: 151.92888107090405
 hess_inv: <10x10 LbfgsInvHessProduct with dtype=float64>
      jac: array([ 10.94642808,  -6.4959734 ,  -7.80646224,   8.78525238,
       -15.69203081,   0.98747269,   0.23433267,  -4.10185774,
        -3.47535264,   5.60347179])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 348
      nit: 51
     njev: 348
   status: 0
  success: True
        x: array([ 5.47321404, -3.2479867 , -3.90323112,  4.39262619,
-7.84601541,
        0.49373635,  0.11716633, -2.05092887, -1.73767632,  2.80173589])
Train Accuracy:  1.0
```

```
Test Accuracy:  0.99
```



1.3 3.) Kernelized Support Vector Machine (8 Points)

In this exercise we will implement another SVM on the two moons dataset, this time using the kernel trick.

The kernelized dual optimization problem for training an SVM is stated in the slides and can be written as a "Quadratic Program", i.e., an optimization of a quadratic objective under linear equality and inequality constraints - a problem of the form

$$min_{\boldsymbol{x}} 0.5\boldsymbol{x}^T\boldsymbol{Q}\boldsymbol{x} + \boldsymbol{q}^T\boldsymbol{x} \quad \text{s.t.} \quad \boldsymbol{G}\boldsymbol{x} \leq \boldsymbol{h}, \quad \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}.$$

Efficient solvers for those kind of problems are well known and implemented in most programming languages. Here we use the CVXPY library.

You can treat this function as a black-box here but also feel free to have a closer look. CVXPY can be used not only for quadratic programs but in general, for any convex optimization problem. The documentation can be found here https://cvxopt.org/index.html

[106]:

```
import cvxpy as cp

def solve_qp(Q: np.ndarray, q: np.ndarray,
             G:np.ndarray, h: Union[np.ndarray, float],
             A:np.ndarray, b: Union[np.ndarray, float]) -> np.ndarray:
    """
    solves quadratic problem: min_x  0.5x^T Q x + q.^T x s.t. Gx <= h and Ax = b
```

```
      in the following 'dim' refers to the dimensionality of the optimization variable x
    :param Q: matrix of the quadratic term of the objective, (shape [dim, dim])
    :param q: vector for the linear term of the objective, (shape [dim])
    :param G: factor for lhs of the inequality constraint (shape [dim], or [dim, dim])
    :param h: rhs of the inequality constraint (shape [dim], or scalar)
    :param A: factor for lhs of the equality constraint (shape [dim], or [dim, dim])
    :param b: rhs of the equality constraint (shape [dim], or scalar)
    :return: optimal x (shape [dim])
    """
    x = cp.Variable(q.shape[0])
    prob = cp.Problem(cp.Minimize(0.5 * cp.quad_form(x, Q) + q.T @ x), constraints=[G @ x <= h, A @ x == b])
    prob.solve()
    return x.value
```

As you may have noticed the problem solved by the solver above differs from the one stated in the slides. Yet the equations from the slides can be reformulated, such that the solver can be used.

1.3.1 Exercise 3.1 (2 Points)

Formulate the kernelized svm dual problem such that the solver can be used to solve it. I.e., state the quantities you need to pass for $\boldsymbol{Q}, \boldsymbol{q}, \boldsymbol{G}, \boldsymbol{h}, \boldsymbol{A}, \boldsymbol{b}$

Proof As per slide 51/53 the dual problem is given by:

$$\max_{\lambda} \sum_i \lambda_i - \frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j \boldsymbol{y}_i \boldsymbol{y}_j \phi\left(\boldsymbol{x}_i\right)^T \phi\left(\boldsymbol{x}_i\right)$$
$$\text{s.t. } \lambda_i \geq 0 \quad , \forall i \in [1 \ldots N] \quad \sum_i \lambda_i \boldsymbol{y}_i = 0$$

Substitue with $\boldsymbol{H}$, which is a $N \times N$ Matrix:

$$\boldsymbol{H}_{ij} = \boldsymbol{y}_i \boldsymbol{y}_j \phi\left(\boldsymbol{x}_i\right)^T \phi\left(\boldsymbol{x}_i\right)$$

The dual problem can be written as:

$$\max_{\lambda} \sum_i \lambda_i - \frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j \boldsymbol{y}_i \boldsymbol{y}_j \boldsymbol{H}_{ij}$$
$$\text{s.t. } \lambda_i \geq 0 \quad , \forall i \in [1 \ldots N] \quad \sum_i \lambda_i \boldsymbol{y}_i = 0$$

Convert to vector notation:

$$\max_{\lambda} \mathbf{1}^T\lambda - \frac{1}{2}\lambda^T \boldsymbol{H}\lambda$$

$$\text{s.t.} \quad \boldsymbol{I}\lambda \geq \mathbf{0} \qquad\qquad \boldsymbol{y}^T\lambda = 0$$

Multiplication by -1 yields:

$$\min_{\lambda} \frac{1}{2}\lambda^T \boldsymbol{H}\lambda - \mathbf{1}^T\lambda$$

$$\text{s.t.} \quad -\boldsymbol{I}\lambda \leq \mathbf{0} \qquad\qquad \boldsymbol{y}^T\lambda = 0$$

Therefore, one can use the following inputs for: $\boldsymbol{Q}, \boldsymbol{q}, \boldsymbol{G}, \boldsymbol{h}, \boldsymbol{A}, \boldsymbol{b}$:

$$\begin{aligned}
\boldsymbol{Q} &:= \boldsymbol{H} \\
\boldsymbol{q} &:= -\mathbf{1} \\
\boldsymbol{G} &:= -\boldsymbol{I} \\
\boldsymbol{h} &:= \mathbf{0} \\
\boldsymbol{A} &:= \boldsymbol{y}^T \\
\boldsymbol{b} &:= 0
\end{aligned}$$

1.3.2 Exercise 3.2 (3 Points)

Implement the functions below so that the SVM can be trained and use for prediction

[107]:

```python
def get_gaussian_kernel_matrix(x: np.ndarray, sigma: float, y: Optional[np.ndarray] = None) -> np.ndarray:
    """ Computes Kernel matrix K(x,y) between two sets of data points x, y  for a Gaussian Kernel with bandwidth sigma.
    If y is not given it is assumed to be equal to x, i.e. K(x,x) is computed
    :param x: matrix containing first set of points (shape: [N, data_dim])
    :param sigma: bandwidth of gaussian kernel
    :param y: matrix containing second set of points (shape: [M, data_dim])
    :return: kernel matrix K(x,y) (shape [M, N])
    """

    if y is None:
        y = x
    # see slide 54
    k_norm = np.sum(np.square(x[None,:,:]-y[:,None,:]),axis=-1)

    return np.exp((-0.5 / sigma) * k_norm)



def fit_svm(samples: np.ndarray, labels: np.ndarray, sigma: float) -> np.
 ↪ndarray:
    """
    fits an svm (with Gaussian Kernel)
    :param samples: samples to fit the SVM to (shape: [N, data_dim])
    :param labels: class labels corresponding to samples (shape: [N])
    :param sigma: bandwidth of gaussian kernel
    :return: "alpha" values, weight for each datapoint in the dual formulation␣
 ↪of SVM (shape [N])
    """

    n = samples.shape[0]
    k = get_gaussian_kernel_matrix(samples, sigma)
    Q = labels[None, :] * labels[:, None] * k
    q = -1 * np.ones(n)
    G = -1 * np.eye(n)
    h = np.zeros(n)
    A = labels
    b = 0

    return solve_qp(Q, q, G, h, A, b)


def predict_svm(samples_query: np.ndarray, samples_train: np.ndarray,␣
 ↪labels_train: np.ndarray,
                alphas: np.ndarray, sigma: float) -> np.ndarray:
    """
    predict labels for query samples given training data and weights
    :param samples_query: samples to query (i.e., predict labels) (shape:␣
 ↪[N_query, data_dim])
    :param samples_train: samples that where used to train svm (shape:␣
 ↪[N_train, data_dim])
    :param labels_train: labels corresponding to samples that where used to␣
 ↪train svm (shape: [N_train])
    :param alphas: alphas computed by training procedure (shape: [N_train])
    :param sigma: bandwidth of gaussian kernel
    :return: predicted labels for query points (shape: [N_query])
    """
    k_train = get_gaussian_kernel_matrix(samples_train,sigma)
    b = np.mean(labels_train - np.sum((labels_train * alphas)[None,:] *␣
↪k_train, axis=-1))
    k_query = get_gaussian_kernel_matrix(samples_train, sigma, samples_query)
    labels_predict = np.sum((labels_train * alphas)[None,:] * k_query ,axis=-1)␣
↪+ b

    return labels_predict
```

We can now execute the code, train and visualize an SVM. For $\sigma = 0.3$ you should get a train accuracy of 1.0 and a test accuracy of > 0.97. You will also get two plots. The first shows all datapoints together with the decision boundary, margins and a countour plot of the svm's predictions. The second one shows again the decision boundary and margins and support vectors (the lower the value α_i is, the more transparent the corresponding point in the plot is, so you will not see most points and only the "importent ones", i.e., the support vectors).

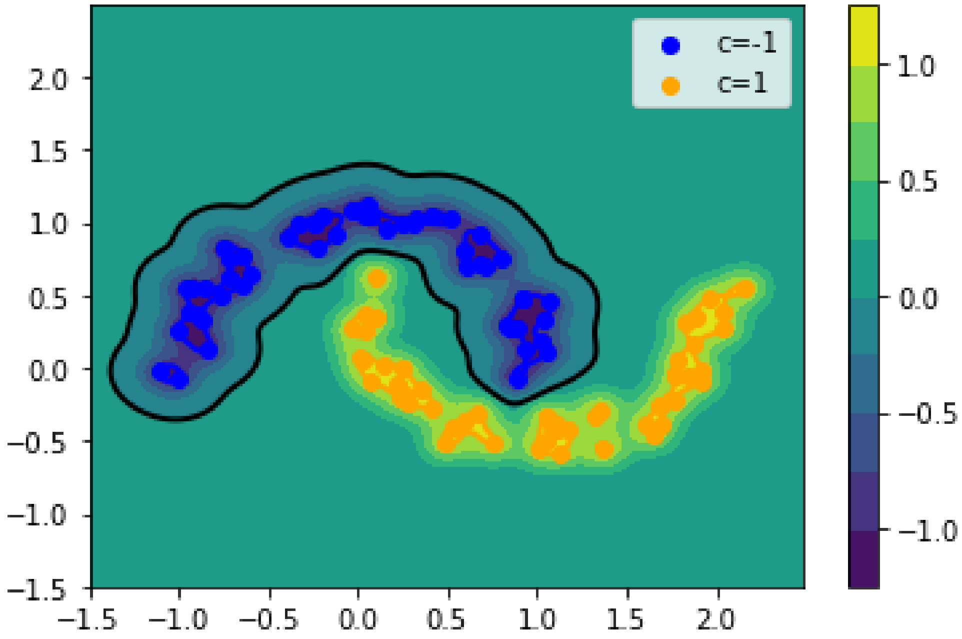
1.3.3 Exercise 3.3 (3 Points)

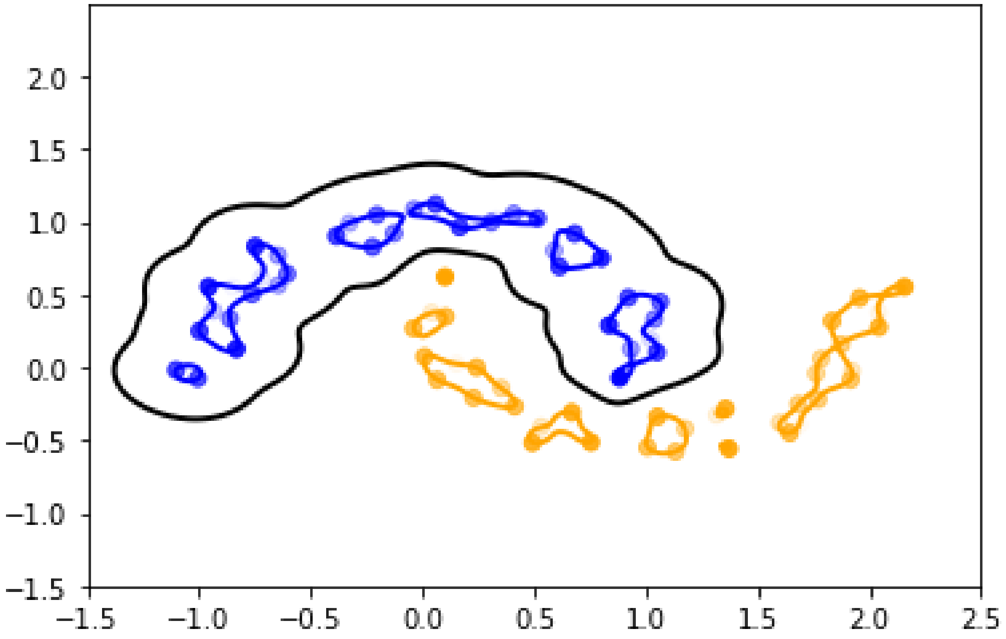
Evaluate different values of sigma in the range of 0.01 to 1.5. What do you observe: - How does the train accuracy change for different values? Why does it behave in this way? - How does the test accuracy change for different values? - How does the number of support vectors change for different values? What is the intuition behind this? - For large values of σ (roughly ≥ 1) you will get an "ArpackNoConvergence" error. This essentially means that the qp-solver was not able to find a solution. Why does this happen? How can we prevent it?

[108]:
```
sigma = 0.01

# train
alphas = fit_svm(train_samples, train_labels, sigma)

# evaluate
train_predictions = predict_svm(train_samples, train_samples, train_labels,␣
 ↪alphas, sigma)
test_predictions = predict_svm(test_samples, train_samples, train_labels,␣
 ↪alphas, sigma)

predicted_train_labels = np.ones(train_predictions.shape)
predicted_train_labels[train_predictions < 0] = -1
print("Train Accuracy: ", np.count_nonzero(predicted_train_labels ==␣
 ↪train_labels) / len(train_labels))

predicted_test_labels = np.ones(test_predictions.shape)
predicted_test_labels[test_predictions < 0] = -1
print("Test Accuracy: ", np.count_nonzero(predicted_test_labels == test_labels)␣
 ↪/ len(test_labels))
```

```
# plot train, contour, decision boundary and margins
plt.figure()
plt_range = np.arange(-1.5, 2.5, 0.01)
plt_grid = np.stack(np.meshgrid(plt_range, plt_range), axis=-1)
flat_plt_grid = np.reshape(plt_grid, [-1, 2])
plt_grid_shape = plt_grid.shape[:2]

pred_grid = np.reshape(predict_svm(flat_plt_grid, train_samples, train_labels,␣
 ↪alphas, sigma), plt_grid_shape)
plt.contour(plt_grid[..., 0], plt_grid[..., 1], pred_grid, levels=[-1, 0, 1],␣
 ↪colors=('blue', 'black', 'orange'),
             linestyles=('-',), linewidths=(2,))
plt.contourf(plt_grid[..., 0], plt_grid[..., 1], pred_grid, levels=10)

plt.colorbar()

plt.scatter(x=train_samples[train_labels == -1, 0],␣
 ↪y=train_samples[train_labels == -1, 1], label="c=-1", c="blue")
plt.scatter(x=train_samples[train_labels == 1, 0], y=train_samples[train_labels␣
 ↪== 1, 1], label="c=1", c="orange")
plt.legend()

# plot margin, decision boundary and support vectors
plt.figure()
plt.contour(plt_grid[..., 0], plt_grid[..., 1], pred_grid, levels=[-1, 0, 1],␣
 ↪colors=('blue', 'black', 'orange'),
             linestyles=('-',), linewidths=(2,))

# squeeze alpha values into interval [0, 1] for plotting
alphas_plt = np.clip(alphas / np.max(alphas), a_min=0.0, a_max=1.0)
for label, color in zip([-1, 1], ["blue", "orange"]):
    color_rgb = colors.to_rgb(color)
    samples = train_samples[train_labels == label]
    color_rgba = np.zeros((len(samples), 4))
    color_rgba[:, :3] = color_rgb
    color_rgba[:, 3] = alphas_plt[train_labels == label]
    plt.scatter(x=samples[:, 0], y=samples[:, 1], c=color_rgba)


plt.xlim(-1.5, 2.5)
plt.show()
```

```
Train Accuracy:  1.0
Test Accuracy:  0.99
```





Evaluate different values of sigma in the range of 0.01 to 1.5. What do you observe:

1. How does the train accuracy change for different values? Why does it behave in this way?

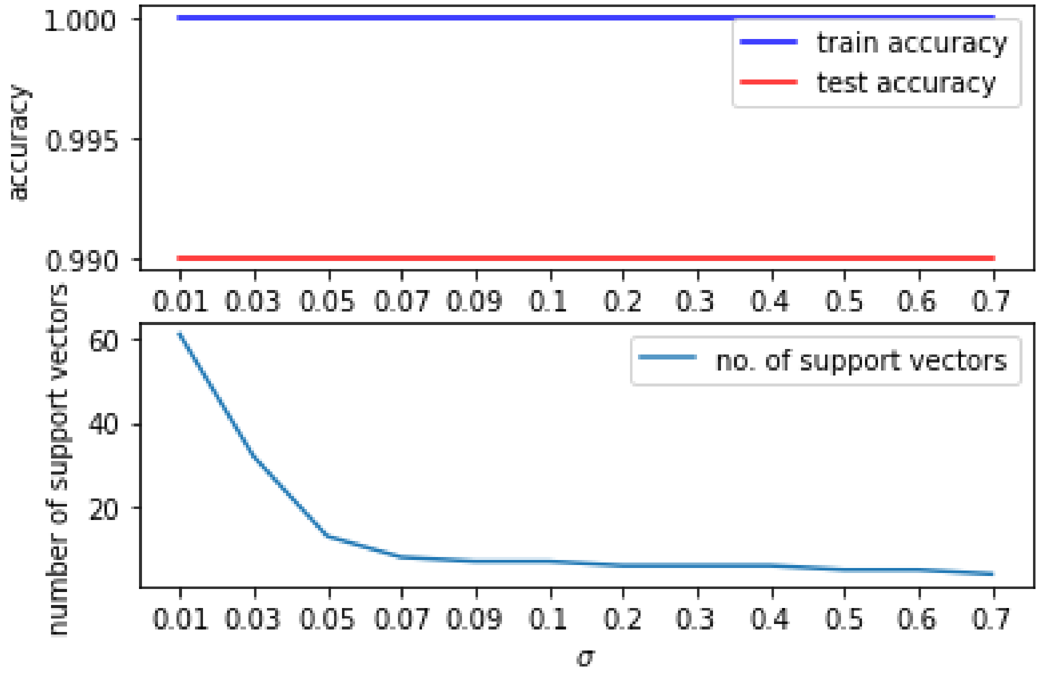
2. How does the test accuracy change for different values?

3. How does the number of support vectors change for different values? What is the intuition behind this?
4. For large values of σ (roughly ≥ 1) you will get an “ArpackNoConvergence” error. This essentially means that the qp-solver was not able to find a solution. Why does this happen? How can we prevent it?

[109]:

```
sigmas = np.array([0.01, 0.03, 0.05, 0.07, 0.09, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6,␣
 ↪0.7])

train_accuracy = []
test_accuracy = []
sv_num = []

for sigma in sigmas:
    alphas = fit_svm(train_samples, train_labels, sigma)
    train_predictions = predict_svm(train_samples, train_samples, train_labels,␣
 ↪alphas, sigma)
    test_predictions = predict_svm(test_samples, train_samples, train_labels,␣
 ↪alphas, sigma)

    alphas_plt = np.clip(alphas / np.max(alphas), a_min=0.0, a_max=1.0)
    alphas_plt[alphas_plt<0.3] = 0
    sv_num.append(np.count_nonzero(alphas_plt))

    predicted_train_labels = np.ones(train_predictions.shape)
    predicted_train_labels[train_predictions < 0] = -1
    train_accuracy.append(np.count_nonzero(predicted_train_labels ==␣
 ↪train_labels) / len(train_labels))

    predicted_test_labels = np.ones(test_predictions.shape)
    predicted_test_labels[test_predictions < 0] = -1
    test_accuracy.append(np.count_nonzero(predicted_test_labels == test_labels)␣
 ↪/ len(test_labels))

x_axis = [str(x) for x in sigmas]

plt.figure
plt.subplot(211)
plt.plot(x_axis,train_accuracy,'b')
plt.plot(x_axis,test_accuracy,'r')
plt.xlabel( "$\sigma$")
plt.ylabel( "accuracy" )
plt.legend(['train accuracy', 'test accuracy'])

plt.subplot(212)
plt.plot(x_axis,sv_num)
plt.xlabel( "$\sigma$")
plt.ylabel( "number of support vectors" )
plt.legend(["no. of support vectors"])
```

```
[109]: <matplotlib.legend.Legend at 0x21bb3489ee0>
```



Answers

Evaluate different values of sigma in the range of 0.01 to 1.5. What do you observe:

1. How does the train accuracy change for different values? Why does it behave in this way? → The train accuracy remains constant for diffent values of σ (as seen above). The reason for this is, that the "two moons data set" can be easily split into its destinct classes even for wide bandwiths of the RBF kernel. Though, generally speaking a larger bandwith would lead to an decrease in train accuaracy, as the model is more generally fitted. However, there is no evidence in the plot above.

2. How does the test accuracy change for different values? → The test accuracy remains constant at 0.99 (as seen above). Pherhaps when the σ becomes very small, the train accuracy could increase, but the test accuracy could decrease as a result of overfitting, but we can not observe this phenomenon in the figure above. When σ becomes very large, the test accuracy could also decline, as a result of underfitting.

3. How does the number of support vectors change for different values? What is the intuition behind this? → Generally speaking, the number of support vectors is decreasing with an increasing σ. The reason for this observation is, that for a larger σ the decision boundary becomes more smooth and therefore less support vectors are required. For smaller σ's the boundary becomes more precise (hence "sharp") and more support vectors are necessary. The

phenomenon for small σ is similar to overfitting and the phenomenon for large σ is similar to underfitting.

4. For large values of σ (roughly ≥ 1) you will get an “ArpackNoConvergence” error. This essentially means that the qp-solver was not able to find a solution. Why does this happen? How can we prevent it? $\rightarrow$ The solver tries to calculate eigenvalues /eigenvectors using an iterative approach and not analytically. For larger σ’e. g. 1.5 the solver for the quadratic problem doesn’t find an optimal solution due two slower convergence of the eigenvectors. As we use an iterative approach, one could increase the number of iterations or use a larger stop criterion.