practical_3_LevinWiebelt_MahdiEnyati_ShamiraDey

June 7, 2021

1 Practical 3

[27]:
```
import numpy as np
from matplotlib import pyplot as plt
import cvxopt  # needed later on
```

1.1 recycle functions from last sheets

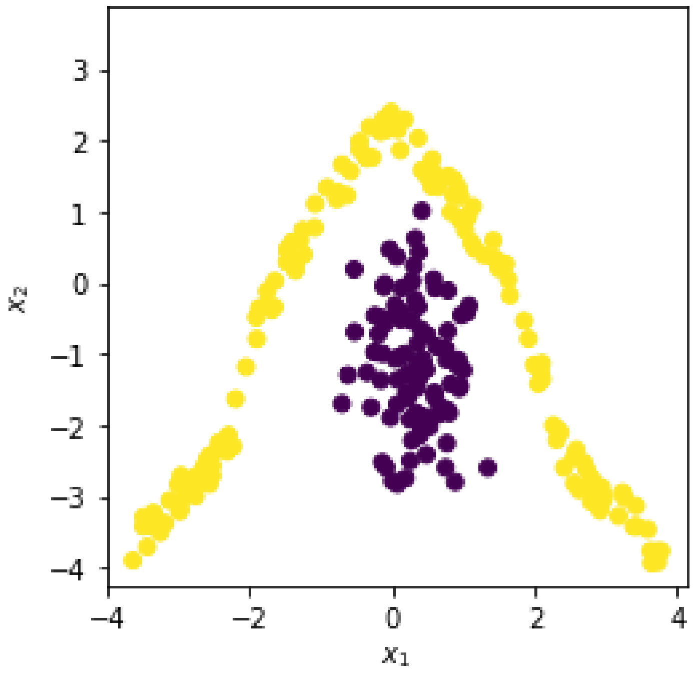
[28]:
```
def split_data(X, y, frac=0.3, seed=None):
    if seed is not None:
        np.random.seed(seed)

    # shuffle vectors
    idx = list(range(len(y)))
    idx_shuffled = np.random.permutation(idx)
    y_shuffled = y[idx_shuffled]
    X_shuffled = X[idx_shuffled]

    # slice vectors
    idx_cutoff = round((1-frac)*len(y)) #seperates training from test (in this␣
 ↪order)
    X_train = X_shuffled[:idx_cutoff, :]
    X_test = X_shuffled[idx_cutoff:, :]
    y_train = y_shuffled[:idx_cutoff]
    y_test = y_shuffled[idx_cutoff:]
    return X_train, X_test, y_train, y_test

def preprocess(X):
    X_norm = (X - X.mean(axis = 0)) / X.std(axis = 0)
    return X_norm

def calculate_accuracy(t_pred, t_true):
    accuracy_list = [1 if pred == null else 0 for pred, null in zip(t_pred,␣
 ↪t_true)]
    accuracy = sum(accuracy_list)/len(accuracy_list)
    return accuracy
```

```
def cross_table(y_pred, y_true):
    import pandas as pd
    print(pd.crosstab(y_pred,y_true, rownames = ['y_pred'], colnames =␣
 ↪['y_true']))
```

2 Task 0: Preliminaries

This time we will work with a synthetic 2d dataset to facilitate visualization and be able to focus on the algorithms rather than spending energy to understand the dataset. The dataset consists of two classes and two predictors. Below we load the dataset and plot it.

[29]:
```
X_2d, t_2d = np.load('data/nonlin_2d_data.npy')[:, :2], np.load('data/
 ↪nonlin_2d_data.npy')[:, 2]

plt.scatter(X_2d[:, 0], X_2d[:, 1], c=t_2d)
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.axis('square'); #yellow=0, purple=1
```



[30]:
```
X_2d.shape
```

[30]:
```
(250, 2)
```

[31]:
```
t_2d.shape
```

[31]: (250,)

2.0.1 Task 0.1

Apply logistic regression and LDA to the provided data, compute classification accuracy and plot the predictions. How do they perform? You're welcome to use sklearn for this task.

[32]:
```
# Split data in training and test set
np.random.seed(seed=1337)
X_train, X_test, t_train, t_test = split_data(X_2d, t_2d)
X_train = preprocess(X_train)
X_test = preprocess(X_test)
```

[33]:
```
# Logistic Regression
from sklearn.linear_model import LogisticRegression

# fit model - training set
logreg = LogisticRegression(penalty = 'none')
logreg.fit(X_train, t_train.flatten())

# predict - test set
print('Logistic Regression')
print('\n')
print('Training accuracy')
t_pred_train = logreg.predict(X_train)
print(calculate_accuracy(t_pred_train, t_train))
print('Test accuracy')
t_pred_test = logreg.predict(X_test)
print(calculate_accuracy(t_pred_test, t_test))
print('\n')

cross_table(t_pred_test, t_test)
```

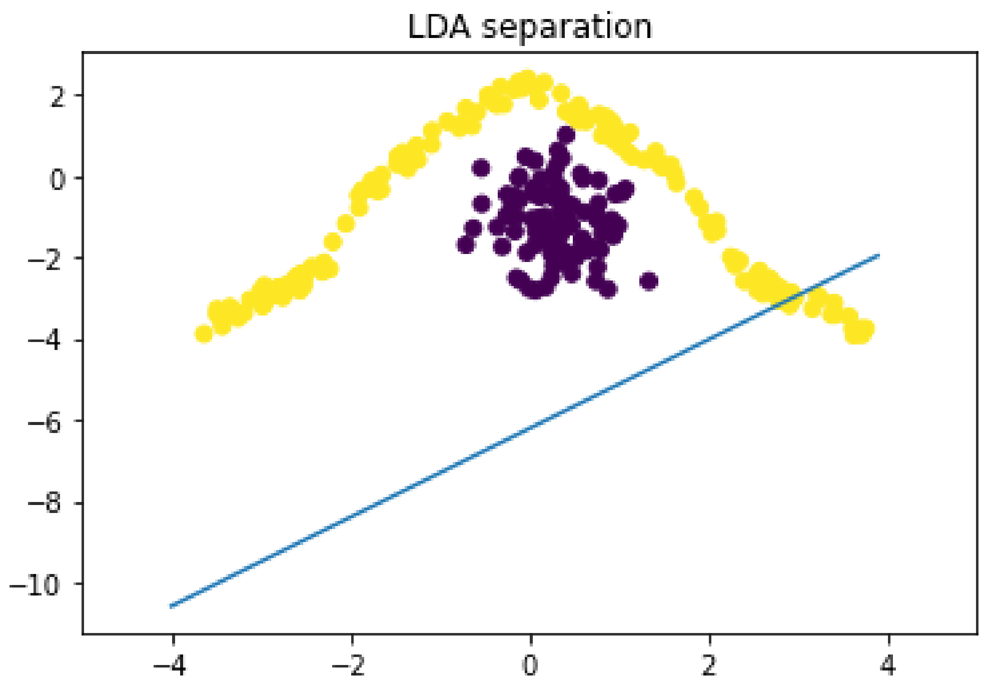
```
Logistic Regression


Training accuracy
0.5942857142857143
Test accuracy
0.6133333333333333


y_true  0.0  1.0
y_pred
1.0      29   46
```

[34]:
```
# LDA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# fit model - training set
lda = LinearDiscriminantAnalysis()
lda.fit(X_train, t_train.flatten())

# predict - test set
print('Linear Discriminant Analysis')
print('\n')
print('Training accuracy')
t_pred_train = lda.predict(X_train)
print(calculate_accuracy(t_pred_train, t_train))
print('Test accuracy')
t_pred_test = lda.predict(X_test)
print(calculate_accuracy(t_pred_test, t_test))

print('\n')
cross_table(t_pred_test, t_test)
```

```
Linear Discriminant Analysis


Training accuracy
0.5942857142857143
Test accuracy
0.6133333333333333


y_true  0.0  1.0
y_pred
1.0      29   46
```

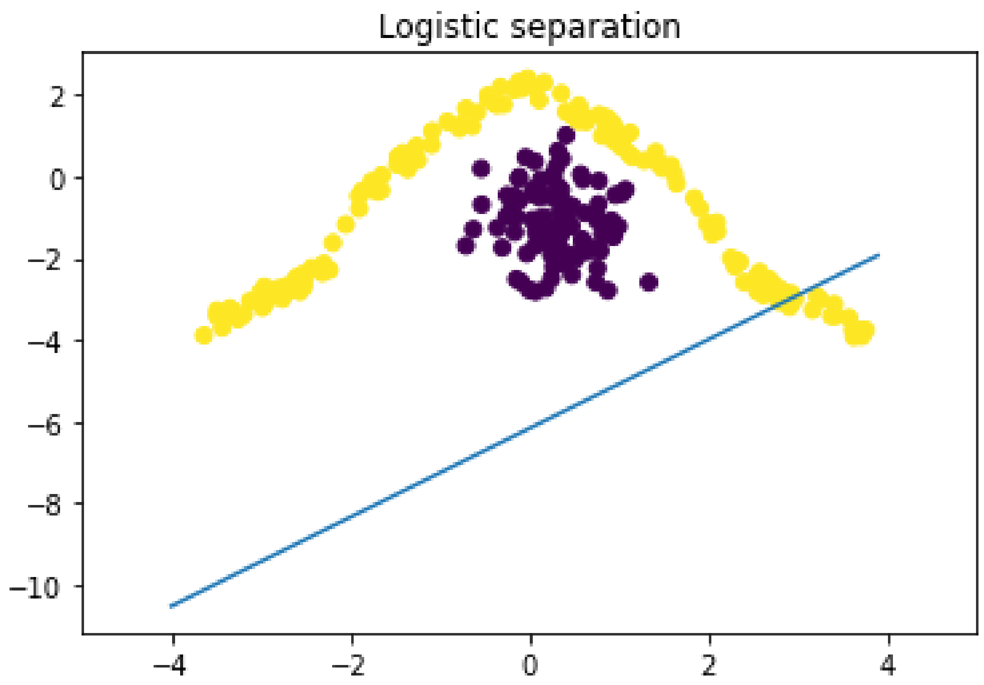
[35]:
```
b1, b2 = lda.coef_.T
c = lda.intercept_
print('b1: {}, b2: {}, c: {}'.format(b1, b2, c))
x1 = np.arange(-4,4,.1)
plt.scatter(X_2d[:, 0], X_2d[:, 1], c=t_2d)
plt.plot(x1, -1/b2 * (c + b1*x1))
plt.xlim(-5, 5)
plt.title('LDA separation')
plt.show()
```

```
b1: [-0.06708334], b2: [0.06148455], c: [0.38256344]
```



[36]:
```
b1, b2 = logreg.coef_.T
c = logreg.intercept_
print('b1: {}, b2: {}, c: {}'.format(b1, b2, c))
x1 = np.arange(-4,4,.1)
plt.scatter(X_2d[:, 0], X_2d[:, 1], c=t_2d)
plt.plot(x1, -1/b2 * (c + b1*x1))
plt.xlim(-5, 5)
plt.title('Logistic separation')
plt.show()
```

```
b1: [-0.06728335], b2: [0.06199473], c: [0.38252068]
```

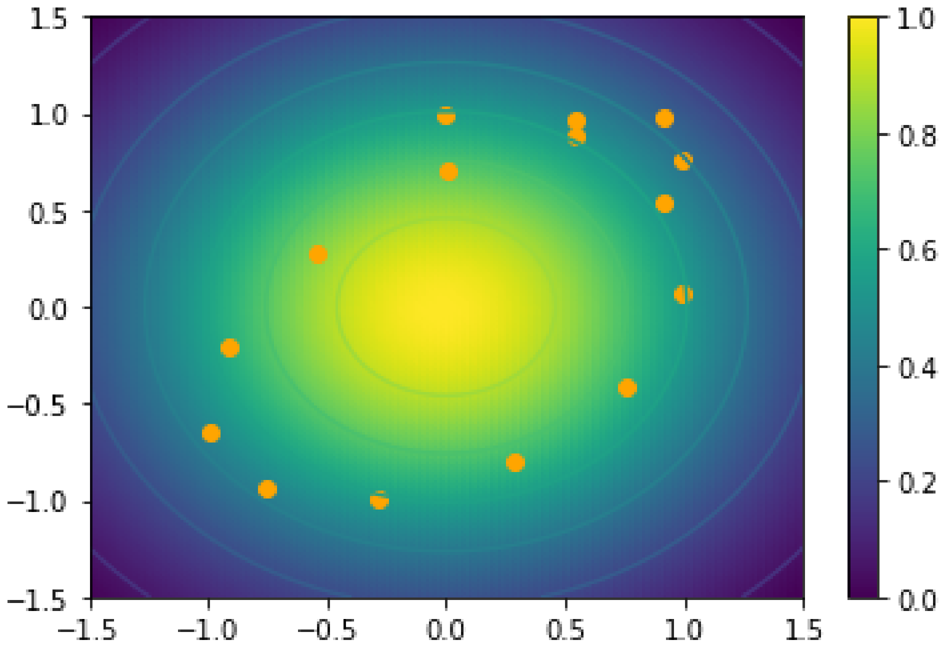


2.0.2 Task 0.2

Implement the Gaussian RBF and visualize the pairwise similarities of x.

[37]:
```python
def gauss_kernel(diff_x1_x2, sigma_sq):
    return np.exp((-1 * np.linalg.norm(diff_x1_x2) ** 2) / (2 * sigma_sq))

gauss_kernel_vectorized = np.vectorize(gauss_kernel)

# unfortunately I was not able to handle the meshgrid/contour stuff, so here␣
 ↪somewhat unnice code
sigma = 1
coordinates = [(x,y) for x in np.linspace(-1.5, 1.5, 100) for y in np.
 ↪linspace(-1.5, 1.5, 100)]
kernel_values = [gauss_kernel(pair, sigma) for pair in coordinates] #reference␣
 ↪value is the zero-vector

#for plotting create vectors
x=[-1000] * len(coordinates)
y=[-1000] * len(coordinates)
for i in range(len(coordinates)):
    x[i], y[i] = coordinates[i]

#create the datapoints
```

```
data = np.stack([np.sin(np.linspace(0, 8, 15)), np.cos(np.linspace(0, 7, 15))]).
↪T

plt.figure()
plt.tricontour(x,y,kernel_values)
plt.scatter(x,y, c=kernel_values)
plt.scatter(data.T[0], data.T[1], color='orange')
plt.colorbar()
plt.show()
```



3 Task 1: Kernel Discriminant Analysis

We will be implementing Kernel Discriminant Analysis, a nonlinear extension to LDA based on the kernel trick, following the original paper by Mika, Rätsch, Weston, Schölkopf and Müller (PDF on StudIP). Note, this is the original notation from the paper, mind the difference betweeen the matrix M and the vectors M_1, M_2.

The goal of Kernel Discriminant Analysis is find a vector of α's that maximizes

$$J(\alpha) = \frac{\alpha^T M \alpha}{\alpha^T N \alpha}$$

where

$$M := (M_1 - M_2)(M_1 - M_2)^T \text{ with } (M_i)_j = \frac{1}{l_i} \sum_{k=1}^{l_i} k(x_j, x_k^i)$$

and

$$N = \sum_{j=1,2} K_j (I - 1_{l_j}) K_j^T$$

Here, the number of samples for class j is denoted by l_j. In the definition of N, $(K_j)_{n,m} = k(x_n, x_m^j)$, I is the identity matrix and 1_{l_j} is a matrix with each entry equal to $1/l_j$. M_1, M_2 have shape $(l_0 + l_1)$ each, N and M have shape $(l_0 + l_1) \times (l_0 + l_1)$.

We obtain the optimal solution by $\alpha = N^{-1}(M_1 - M_0)$. Note that you may have to regularize N^{-1} for numerical stability (see paper or Wikipedia).

Prediction for an arbitrary sample x is then carried out by building a weighted sum over it's kernel with the samples from the training set.

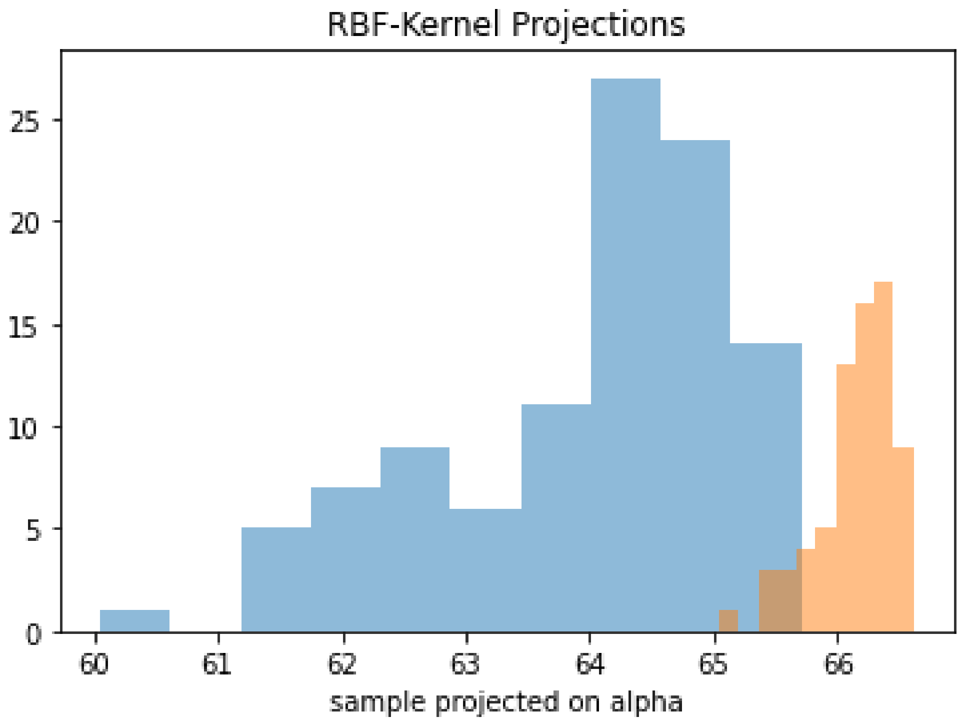
1. Implement a function `kda(X, t, kernel)` that carries out Kernel Discriminant Analysis on the data X, t and returns a vector `alpha`.
2. Implement a function `def predict(x, X, alpha, kernel)`.
3. Visualize results given a linear kernel and a Gaussian Radial Basis Function kernel.

Use the following snippet as a starting point.

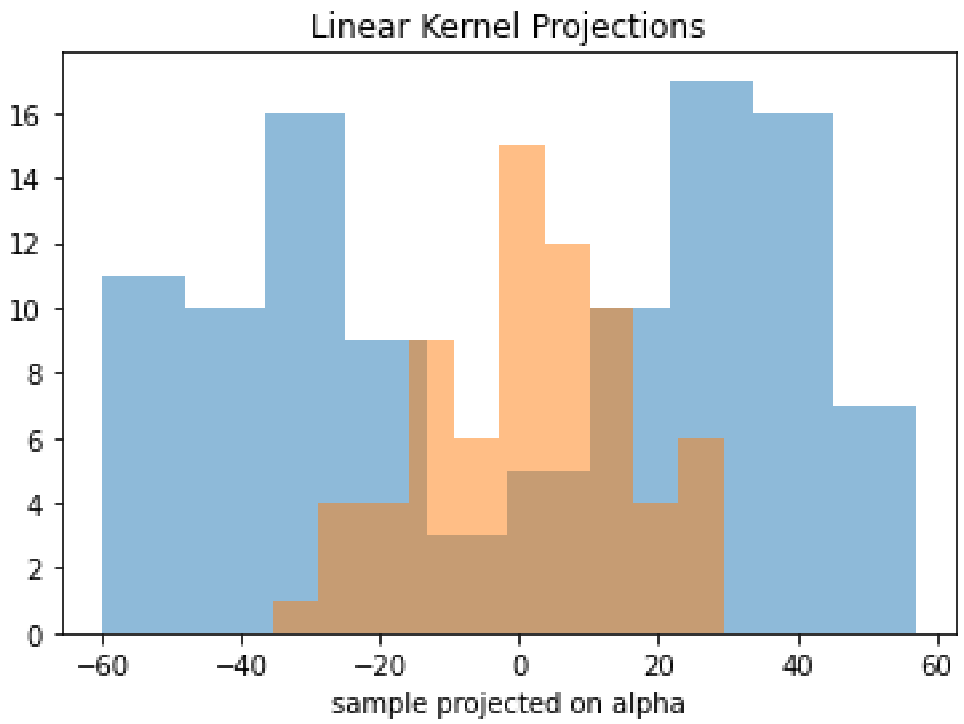
[38]:
```
# Todo: pass kernel parameter with *args
```

[39]:
```
def kda(X, t, kernel, mu=0):
    l0, l1 = [(t == 0).sum(), (t == 1).sum()]
    X_split = X[(t == 0)], X[(t == 1)]
    x0, x1 = X_split
    nr_samples, nr_features = X.shape

    M_1 = np.array( [sum ( [kernel (X[i], x1[k]) for k in range(l1)] ) for i in␣
 ↪range(nr_samples) ] )
    M_0 = np.array( [sum ( [kernel (X[i], x0[k]) for k in range(l0)] ) for i in␣
 ↪range(nr_samples) ] )
    K1 = np.array( [[kernel (X[n], x1[m]) for n in range(nr_samples)] for m in␣
 ↪range(l1)] ).T
    K0 = np.array( [[kernel (X[n], x0[m]) for n in range(nr_samples)] for m in␣
 ↪range(l0)] ).T
    centering_mat_0 = np.identity(l0) - np.ones((l0, l0)) / l0
    centering_mat_1 = np.identity(l1) - np.ones((l1, l1)) / l1
    N_1 = K1 @ centering_mat_1 @ K1.T
    N_0 = K0 @ centering_mat_0 @ K0.T

    N = N_1 + N_0 + np.identity(nr_samples) * mu
    alpha = np.linalg.inv(N) @ (M_1 - M_0)
    return alpha


def predict(x, X, alpha, kernel):
    # idea: w@phi(x) = sum of alpha_i * kernel_i
    nr_samples = len(alpha)
    projected_x = sum( np.array( [alpha[i] * kernel(x, X[i]) for i in␣
↪range(nr_samples)] ) )
    return projected_x


def linear_kernel(x_n, x_m):
    return(x_n.T @ x_m)


def rbf_kernel(x_n, x_m, gamma=5):
    return np.exp((-1 * np.linalg.norm(x_n - x_m) ** 2) / (2 * gamma))

# Perform KDA with linear and RBF kernel
alphas_rbf = kda(X_train, t_train, rbf_kernel, mu = 2)
alphas_linear = kda(X_train, t_train, linear_kernel)

t_train = t_train.astype('bool')
```

[40]:
```
train_rbf_projections_class_1 = [predict(x, X_train, alphas_rbf, rbf_kernel)␣
↪for x in X_train[t_train]]
train_rbf_projections_class_0 = [predict(x, X_train, alphas_rbf, rbf_kernel)␣
↪for x in X_train[~t_train]]
plt.hist(train_rbf_projections_class_1, alpha = .5)
plt.hist(train_rbf_projections_class_0, alpha = .5)
plt.title('RBF-Kernel Projections')
plt.xlabel('sample projected on alpha')
plt.show()
```

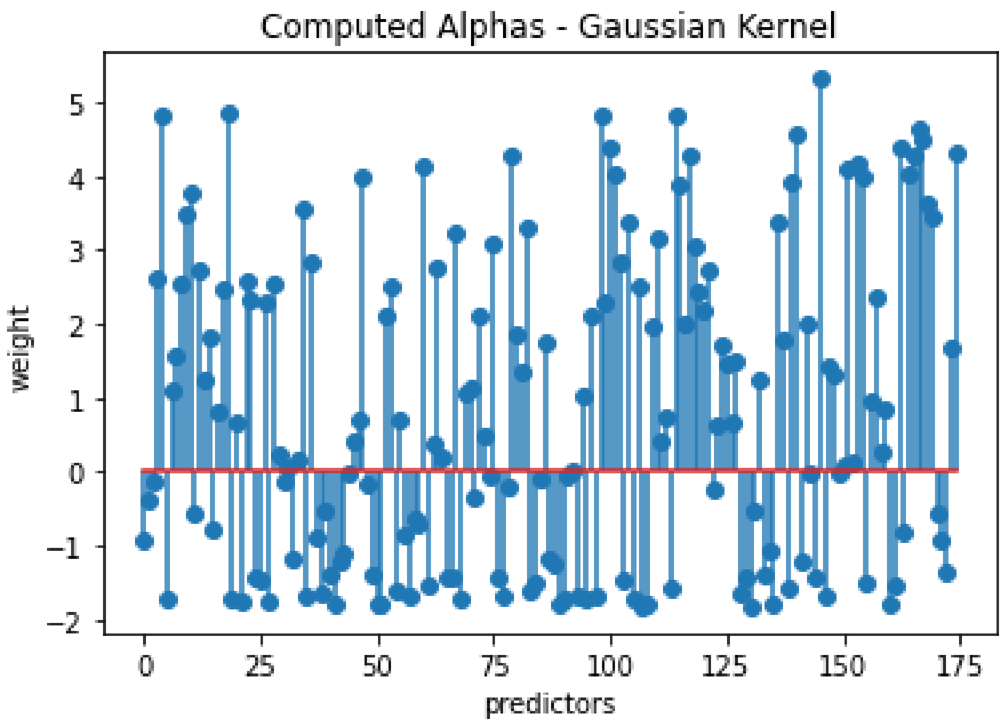


```
[41]: train_linear_projections_class_1 = [predict(x, X_train, alphas_linear,␣
       ↪linear_kernel) for x in X_train[t_train]]
      train_linear_projections_class_0 = [predict(x, X_train, alphas_linear,␣
       ↪linear_kernel) for x in X_train[~t_train]]

      plt.hist(train_linear_projections_class_1, alpha = .5)
      plt.hist(train_linear_projections_class_0, alpha = .5)
      plt.title('Linear Kernel Projections')
      plt.xlabel('sample projected on alpha')
      plt.show()
```

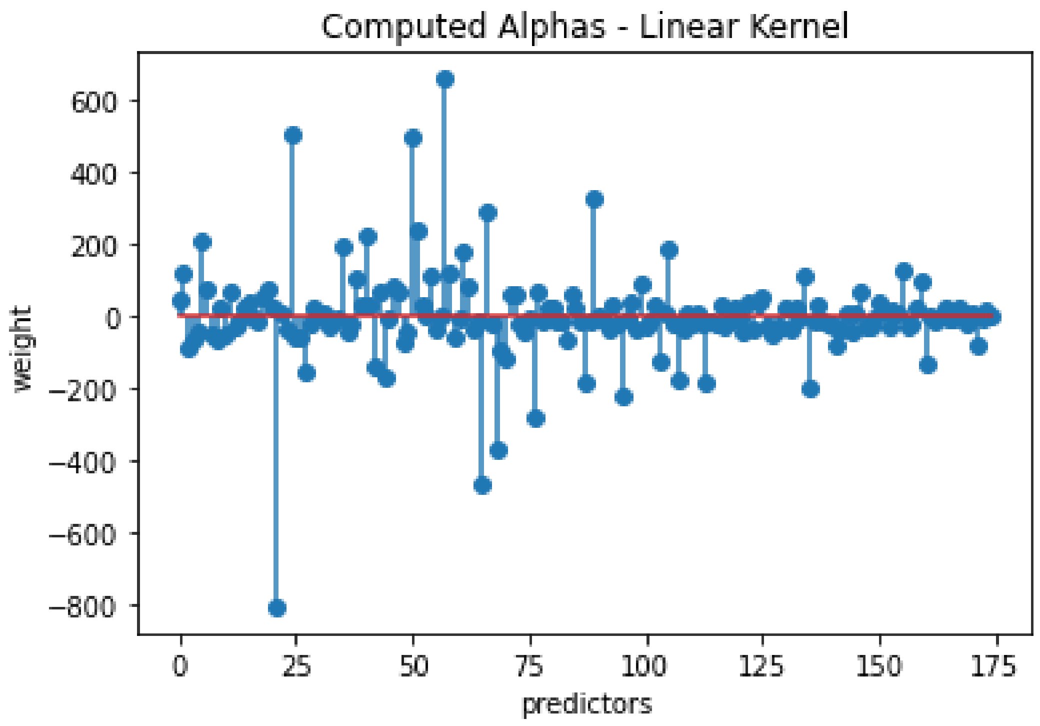


```
[42]: # Visualize results
      plt.stem(alphas_rbf, use_line_collection=True)
      plt.title('Computed Alphas - Gaussian Kernel')
      plt.ylabel('weight')
      plt.xlabel('predictors')
```

```
[42]: Text(0.5, 0, 'predictors')
```



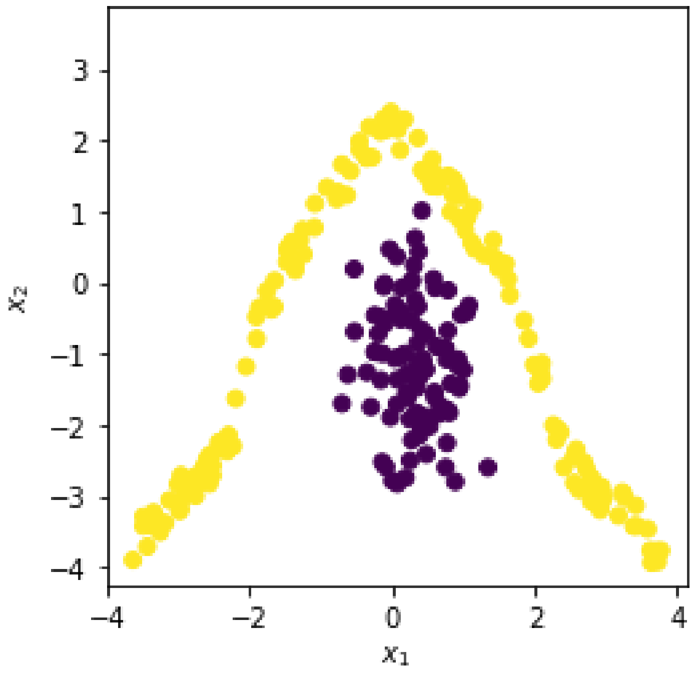
[43]:
```
# Visualize results
plt.stem(alphas_linear, use_line_collection=True)
plt.title('Computed Alphas - Linear Kernel')
plt.ylabel('weight')
plt.xlabel('predictors')
plt.show()
#plt.yscale('exp')
```



3.0.1 Task 1.2

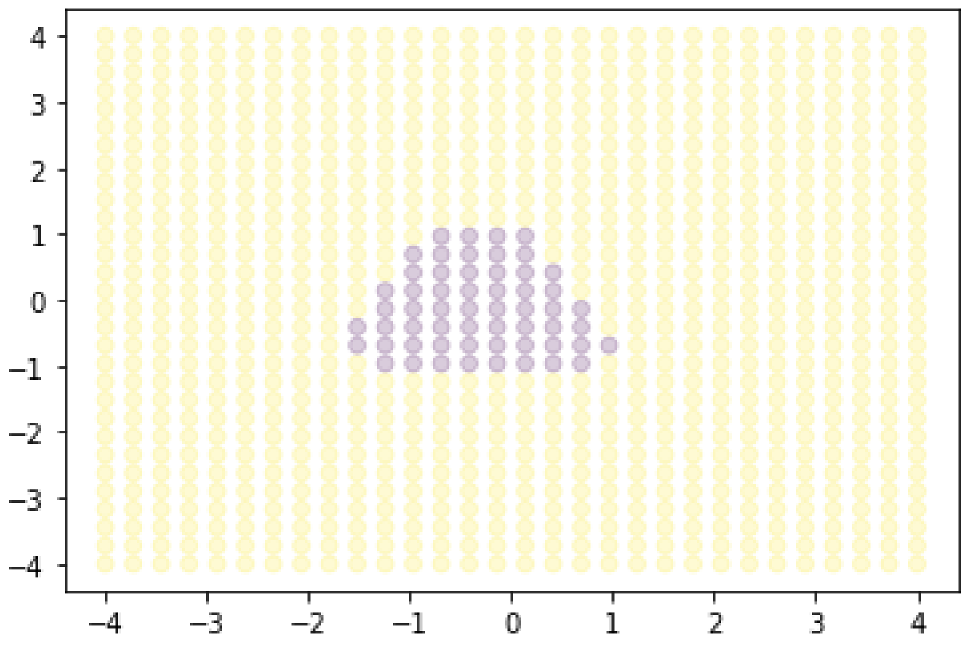
Which value would you use as a threshold to separate the classes? Use this threshold value to visualize the decision boundaries in a 2d grid (see example code below) and make predictions on the test set.

[44]:
```
plt.scatter(X_2d[:, 0], X_2d[:, 1], c=t_2d)
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.axis('square'); #yellow=0, purple=1
```



[74]:
```
def plot_decision_boundary(predict_fn):
    ''' Plot decision boundary.

    predict_fn: function handle to a predict function that takes
                as input a data point x and outputs the predicted
                label t
    '''
    grid = np.meshgrid(np.linspace(-4, 4, 30), np.linspace(-4, 4, 30))
    grid_x, grid_y = grid[0].flatten(), grid[1].flatten()
    grid_t = np.array([
        predict_fn( np.array([x, y]) ) for x, y, in zip(grid_x, grid_y)])
    plt.scatter(grid_x, grid_y, c=grid_t, alpha=0.2)
```

[75]:
```
predict_fn = lambda x: predict(x, X_train, alphas_rbf, rbf_kernel) < 65.5
plot_decision_boundary(predict_fn)
```



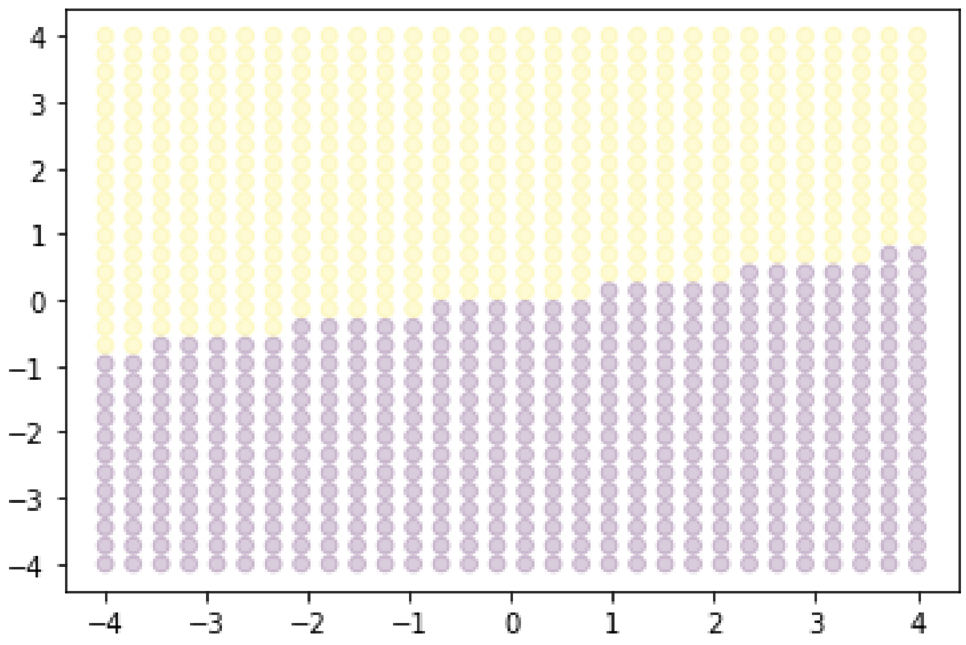
```
[58]: t_test = t_test.astype('bool')
      t_pred = np.array( [predict(x, X_train, alphas_rbf, rbf_kernel) for x in␣
       ↪X_test] ) < 65.5

      print('Gaussian RBF Kernel')
      cross_table(t_pred, t_test)
```

```
Gaussian RBF Kernel
y_true  False  True
y_pred
False      27     4
True        2    42
```

```
[ ]:
```

```
[ ]:
```

```
[73]: predict_fn = lambda x: predict(x, X_train, alphas_linear, linear_kernel) < 0
      plot_decision_boundary(predict_fn)
```



[ ]:
```
t_pred = np.array( [predict(x, X_train, alphas_linear, linear_kernel) for x in␣
 ↪X_test] ) > 0
print('Linear Kernel')
cross_table(t_pred, t_test)
```