CSCI-UA.0473-001 Introduction to Machine Learning

Homework 2

Name: Lejia Hu

Due: Oct. 14, 2020

Goal: The goal of this homework is to practice implementing a dual form of a linear support vector machine without the optimizing algorithm.

Please DO NOT change the position of any cell in this assignment. Read every line till the end, and try your best implement everything.

You will need the following packages below to do the homework. Please DO NOT import any other packages.

In [1]:
```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from scipy.optimize import minimize
import matplotlib.pyplot as plt
```

1. Implementing Linear SVM (55 pts total)

In this problem you will implement linear SVM by solving the dual problem and apply it to the Iris dataset. The original dataset has 4 features with 3 classes, but in this homework we'll only use the first two features and ignore the first class (to make this a binary classification problem).

Loading and spliting the data (nothing to do)

The following cell loads the data and pre-processes it to be used for training later. **Do not modify anything in this cell**.

In [2]:
```
# Load in the data.
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Ignore the first class and use only the first 2 features.
X = X[y != 0, :2]
y = y[y != 0]

# Make sure that the class labels are either +1 or -1.
y[y==2] = -1

n_sample = len(X)

# Randomly order the data.
np.random.seed(0)
order = np.random.permutation(n_sample)
X = X[order]
y = y[order].astype(np.float)

# Split the data into 10% testing and 90% training.
X_train = X[:int(.9 * n_sample)]
y_train = y[:int(.9 * n_sample)]
X_test = X[int(.9 * n_sample):]
y_test = y[int(.9 * n_sample):]
```

Part (a) (5 pts) Computing the matrix of inner products.

For training data $X \in \mathbb{R}^{n\times p}$, $x_i \in \mathbb{R}^p$, the inner product matrix is

$$\mathbf{M} = (m_{ij}) \in \mathbb{R}^{n\times n}, \quad m_{ij} = \langle x_i, x_j \rangle \in \mathbb{R}.$$

Implement the following function below that takes in the matrix X of training data and returns the corresponding inner product matrix $\mathbf{M}$. For this you may use the numpy function `np.dot()` . Also answer the following two questions below.

1. What is the fewest number of inner products do you need to compute? Explain why.
2. What are the diagonal entries of the matrix $\mathbf{M}$?

Answer goes here

1. The minimal number of calculations is $1 + 2 + \ldots + n - 1 + n = \frac{(1+n)n}{2}$, since M is a symmetric matrix. The calculation result above diagonal are equal to the part below. So we only need to calculate upper triangular matrix plus the diagonal.
2. The diagonal entries are consist of $m_{ii}, i = 1, 2, \ldots, n$, which is the dot product of x and itself.

In [3]:
```
def inner_product_matrix(X):
    """
    Compute the inner product matrix of the training data X where

    Input:
        X: np.ndarray(n, p), n data points in dimension of p

    Return:
        M: np.ndarray(n, n), each entry is the inner product of t
    """
    ##TODO-start##
    M = np.dot(X, X.T)
    return M
    ##TODO-end##
```

Part (b) (10 pts) The dual problem for linear SVM

Recall that for linear SVM the dual problem is

$$\max_{\alpha} W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \langle x_i, x_j \rangle$$
$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, n$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

where $\alpha \in \mathbb{R}^n$ is a vector.

Implement the objective function $W(\alpha)$which also takes the training data features and labels as parameters. Do not worry about the constraints for the moment since you will deal with these next.

- You may also find the function `np.diag()` useful to help vectorize your code and make it run faster.

Let's simplify second part of above formula, then we get

$$\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \langle x_i, x_j \rangle = \left(y_1\alpha_1 \sum_{j=1}^{n} y_j\alpha_j \langle x_1, x_j \rangle + \ldots + y_n\alpha_n \sum_{j=1}^{n} y_j\alpha_j \langle x_n, x_j \right.$$

$$= \begin{bmatrix} y_1\alpha_1 & y_2\alpha_2 & \ldots & y_n\alpha_n \end{bmatrix} \underline{\begin{bmatrix} \sum_{j=1}^{n} y_j\alpha_j \langle x_1, x_j \rangle \\ \sum_{j=1}^{n} y_j\alpha_j \langle x_2, x_j \rangle \\ \vdots \\ \sum_{j=1}^{n} y_j\alpha_j \langle x_n, x_j \rangle \end{bmatrix}}$$

Now, let's convert this part to the form of matrix multiplication

$$\sum_{j=1}^{n} y_j\alpha_j \langle x_1, x_j \rangle = y_1\alpha_1 \langle x_1, x_1 \rangle + y_2\alpha_2 \langle x_1, x_2 \rangle + \ldots + y_n\alpha_n \langle x_1, x_n \rangle$$

$$= \begin{bmatrix} y_1\alpha_1 & y_2\alpha_2 & \ldots & y_n\alpha_n \end{bmatrix} \begin{bmatrix} \langle x_1, x_1 \rangle \\ \langle x_1, x_2 \rangle \\ \vdots \\ \langle x_1, x_n \rangle \end{bmatrix}$$

Therefore, we can rewrite the underline matrix as follow

$$\begin{bmatrix} \sum_{j=1}^{n} y_j\alpha_j \langle x_1, x_j \rangle \\ \sum_{j=1}^{n} y_j\alpha_j \langle x_2, x_j \rangle \\ \vdots \\ \sum_{j=1}^{n} y_j\alpha_j \langle x_n, x_j \rangle \end{bmatrix} = \begin{bmatrix} \langle x_1, x_1 \rangle & \langle x_2, x_1 \rangle & \ldots & \langle x_n, x_1 \rangle \\ \langle x_1, x_2 \rangle & \langle x_2, x_2 \rangle & \ldots & \langle x_n, x_2 \rangle \\ \vdots & & & \\ \langle x_1, x_n \rangle & \langle x_2, x_n \rangle & \ldots & \langle x_n, x_n \rangle \end{bmatrix} \begin{bmatrix} y_1\alpha_1 \\ y_2\alpha_2 \\ \vdots \\ y_n\alpha_n \end{bmatrix}$$

Finally, we get

$$\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \langle x_i, x_j \rangle$$

$$= \begin{bmatrix} y_1\alpha_1 & y_2\alpha_2 & \ldots & y_n\alpha_n \end{bmatrix} \begin{bmatrix} \langle x_1, x_1 \rangle & \langle x_2, x_1 \rangle & \ldots & \langle x_n, x_1 \rangle \\ \langle x_1, x_2 \rangle & \langle x_2, x_2 \rangle & \ldots & \langle x_n, x_2 \rangle \\ \vdots & & & \\ \langle x_1, x_n \rangle & \langle x_2, x_n \rangle & \ldots & \langle x_n, x_n \rangle \end{bmatrix} \begin{bmatrix} y_1\alpha_1 \\ y_2\alpha_2 \\ \vdots \\ y_n\alpha_n \end{bmatrix}$$

In [4]:

```python
def objective_function(a, X, y):
    """
    The objective function of the dual problem W.

    Input:
        a: np.ndarray(n,), the parameter alpha we want to optimiz
        X: np.ndarray(n, p), the matrix of training data features
        y: np.ndarray (n,), the vector of training data labels, m

    Return:
        W: float, value of the objective function.
    """

    ##TODO-start##
    #np multiply at each position
    YA = y * a
    inner = inner_product_matrix(X)

    # @ means np.dot
    W = np.sum(a) - (YA @ inner @ YA) / 2
    ##TODO-end##
    return W
```

Part (c) (35 pts) Implementing the `fit()` function.

Instead of writing your own optimization algorithm, use the scipy function `scipy.optimize.minimize()` to automatically optimize α. (Reference: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html (https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html))

- Note that the `minimize()` function minimizes the objective function, so you'll need to reformulate the dual problem as a minimization problem with the following trick.
$$\alpha^* = \text{argmax}_\alpha[W(\alpha)] = \text{argmin}_\alpha[-W(\alpha)].$$

There are several pieces you will need to implement. Read the documentation carefully to set everythign up correctly.

1. Get the box constraints $0 \leq \alpha_i \leq C$ for $i = 1, \ldots, n$. You will pass this into `minimize()` as the `bounds` argument.
2. Get the linear constraint $\sum_{i=1}^n \alpha_i y_i = 0$. You will pass this into `minimize()` as the `constraints` argument.
3. Call `minimize()` using the correct objective function $-W(\alpha)$ as well as the constraints and bounds from the previous 2 parts. Use $\alpha_0 = 0 \in \mathbb{R}^n$ as the initial point and use the SLSQP method.
4. Compute the primal variable $w \in \mathbb{R}^p$ using the formula
$$w = \sum_{i=1}^n \alpha_i y_i x_i$$
5. Compute the bias term using the formula
$$b = \frac{1}{n}\sum_{i=1}^n \left(y_i - \sum_{j=1}^n \alpha_j y_j \langle x_j, x_i \rangle \right) = \frac{1}{n}\sum_{i=1}^n \left(y_i - w^T x_i \right)$$
once you have computed the minimizer α.

Hints:

- Read the explanations of `fun`, `x0`, `bounds`, `constraints`, corresponding to objective function, initialization, bounds, and constraints.
- Equality constraints mean that the constraint function result is zero.
- If `res = minimize(...)`, then `res.x` is the minimizer.

In [5]:
```python
def fit(X, y, C):
    """
    Computes the parameters alpha and bias that determine the max

    Input:
        X: np.ndarray(n,p), matrix of training data features
        y: np.ndarray(n, ), vector of training data labels
        C: float, slack parameter that is non-negative

    Return:
        w: np.ndarray(p,), vector of primal variable values (vect
        bias: float, the bias term in SVM
        alpha: np.ndarray(n, ), vector of dual variable values
    """
    ## TO-DO STARTS HERE##
    # setting bound
    bnd = (0, C)
    size = X.shape[0]
    bnds = tuple([bnd] * size)

    # setting constraints
    def contraint(a, y):
        sum_ = 0
        sum_ -= np.dot(a, y)
        return sum_

    # setting the initializer
    alpha = minimize(
        fun=lambda a: -objective_function(a, X, y), # -W(a)
        x0=np.zeros(size),                          # a0 = (0, 0,
        bounds=bnds,
        constraints=({'type': 'eq', 'fun': contraint, 'args': (y,
        method='SLSQP'
    ).x

    # Compute the primal variables w.
    d = alpha * y
    d = np.repeat(d.reshape(-1, 1), 2, axis=1)

    w = np.sum(d * X, axis=0)

    # Compute the bias.
    bias = np.mean(y - X @ w)

    ## TO-DO ENDS HERE ##
    return (w, alpha, bias)
```

Plotting the results (nothing to do here)

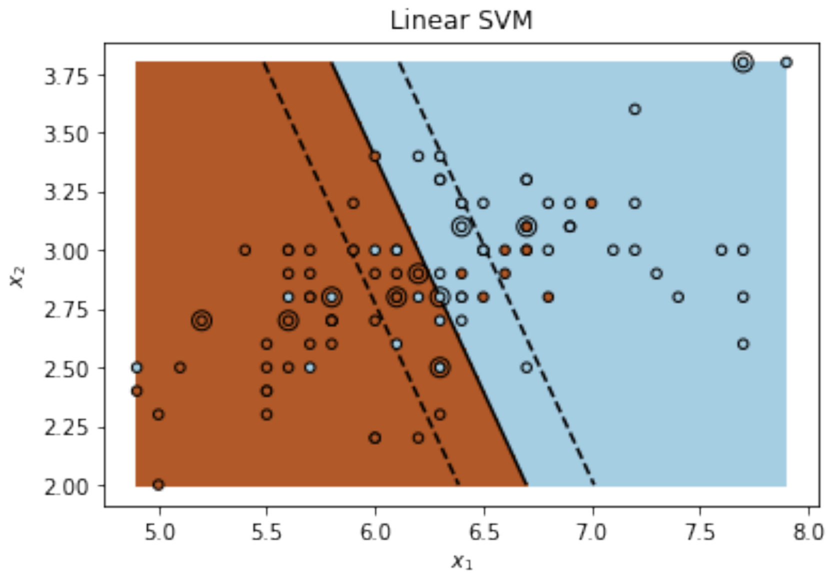
Please DO NOT change anything here. This may take a few minutes to finish the optimization.

- If your implementation is perfect there won't be any errors thrown and it will show a figure similar to the first one in https://scikit-learn.org/stable/auto_examples/exercises/plot_iris_exercise.html#sphx-glr-auto-examples-exercises-plot-iris-exercise-py (https://scikit-learn.org/stable/auto_examples/exercises/plot_iris_exercise.html#sphx-glr-auto-examples-exercises-plot-iris-exercise-py) (ignore if the colors are swapped).

In [6]:
```python
w, alpha, bias = fit(X_train, y_train, C = 10)
```

In [7]:
```python
plt.figure(1)
plt.clf()
plt.scatter(X[:, 0], X[:, 1], c=y, zorder=10, cmap=plt.cm.Paired,
            edgecolor='k', s=20)

# Circle out the test data
plt.scatter(X_test[:, 0], X_test[:, 1], s=80, facecolors='none',
            zorder=10, edgecolor='k')

plt.axis('tight')
x_min = X[:, 0].min()
x_max = X[:, 0].max()
y_min = X[:, 1].min()
y_max = X[:, 1].max()

XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
XXYY = np.c_[XX.ravel(), YY.ravel()]
ZZ = []
for i in range(XXYY.shape[0]):
    ZZ.append(XXYY[i]@w+bias)

Z = np.array(ZZ)

# Put the result into a color plot
Z = Z.reshape(XX.shape)
plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
plt.contour(XX, YY, Z, colors=['k', 'k', 'k'],
            linestyles=['--', '-', '--'], levels=[-.5, 0, .5])
plt.title('Linear SVM')
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
plt.show()
```

```
/Users/wangt/miniconda3/envs/ds/lib/python3.7/site-packages/ipykernel
_launcher.py:26: MatplotlibDeprecationWarning: shading='flat' when X
and Y have the same dimensions as C is deprecated since 3.3.  Either
specify the corners of the quadrilaterals with X and Y, or pass shadi
ng='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].
This will become an error two minor releases later.
```



Part (d) (5 pts) Implementing the `predict()` function

Implement the `predict()` function below which computes the discriminant function on the test data and returns a vector whose entries are either $+1$ or -1. For this use the primal variable w along with the bias b. You **do not** need to modify the `accuracy()` function or other code in this cell. If your method is correct you should achieve around 70% accuracy on both training and testing sets.

In [8]:
```
def predict(x_test, w, bias):
    """
    Compute the predictions y_pred on the test set using only the

    Input:
        x_test: np.ndarray(n,p), matrix of the test data
        alpha: np.ndarray(n,), vector of the dual variables
        bias: float, the bias term

    Output:
        y_pred: np.ndarray(n,), vector of the predicted labels, e
    """
    ##TODO-start##
    y_pred_pos = np.where(x_test @ w + bias >= 0, 1, 0)
    y_pred_neg = np.where(x_test @ w + bias < 0, -1, 0)
    y_pred = y_pred_pos + y_pred_neg

    return y_pred
    ##TODO-end##



def accuracy(y_pred, y_true):
    """
    Computes the accuracy on the test set given the class predict

    Input:
        y_pred: np.ndarray(n,), vector of predicted class labels
        y_true: np.ndarray(n,), vector of true class labels

    Output:
        float, accuracy of predictions
    """
    return np.mean(y_pred*y_true > 0)

y_pred = predict(X_test, w, bias)
y_pred_train = predict(X_train, w, bias)
print("Training accuracy = {:0.2f}%".format(100*accuracy(y_pred_t
print("Testing accuracy = {:0.2f}%".format(100*accuracy(y_pred, y
```

```
Training accuracy = 73.33%
Testing accuracy = 70.00%
```

2. Solving the primal SVM problem with SGD (30 pts total)

In the previous problem we looked at solving the dual problem for a linear SVM. However, we could have instead solved the primal problem directly

$$\min_{w \in \mathbb{R}^p} \; L(w) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \max\{0,\; 1 - y_i w^T x_i\}$$

This is an unconstrained problem and the loss $\max\{0,\; 1 - y_i w^T x_i\}$ is called the hinge loss. Note that even though this function is not differentiable when $1 - y_i w^T x_i = 0$ we will use stochastic gradient descent anyways. Technically we will be using the "sub-gradient" but this is not something you need to worry about. Below is the pseudo-code to train SVM with SGD.

1. Initialize $w_0 \in \mathbb{R}^p$ A suggested choice is $w_0 = 0$
2. For iterations $t = 1, \dots, T$ do
   - Select a training example (x_i, y_i) at random from the dataset.
   - Compute the gradient $\nabla L_i(w_{t-1})$ where
   $$L_i(w) = \frac{1}{2}\|w\|^2 + C\max\{0, 1 - y_i w^T x_i\}$$
   - Update the parameters:
   $$w_t \leftarrow w_{t-1} - \gamma_t \nabla L_i(w_{t-1})$$
   where $\gamma_t > 0$ is the learning rate.
3. Return the final parameters w_T.
4. Compute the bias b using the formula
$$b = \frac{1}{n}\sum_{i=1}^{n}(y_i - w_T^T x_i)$$

Part (a) (10 pts) Implementing the loss function.

Implement the loss function for the primal problem

$$L(w) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\max\{0,\ 1 - y_i w^T x_i\}$$

which takes in the training data matrix X and labels y as well as the parameter C as additional arguments. Use the functions `np.dot()` , `np.max()` , and `np.sum()` to vectorize your code i.e. **do not** use loops.

In [9]:
```
def loss(w, X, y, C):
    """
    Compute the loss function on all training examples L(w).

    Input:
        w: np.ndarray(p,), the vector of parameters for the SVM
        X: np.ndarray(n,p), the training data matrix of features
        y: np.ndarray(n,), the training data vector of labels
        C: float, the penalty parameter in SVM

    Return:
        L: float, the loss on the training set.
    """
    tmp = y * np.dot(X, w)
    L = np.dot(w, w) / 2 + C * np.sum(np.where(tmp > 0, tmp, 0))
    return L
```

Part (b) (10 pts) Implementing the gradient of the loss function.

Derive and implement the gradient of the loss function $L_i(w)$ which only considers one training example (x_i, y_i) and is different from the loss function $L(w)$ that you implemented above. As a reminder

$$L_i(w) = \frac{1}{2}\|w\|^2 + C\max\{0,\ 1 - y_i w^T x_i\}$$

To derive the gradient separate it into two cases: 1. when $y_i w^T x_i > 1$ and 2. when $y_i w^T x_i < 1$. Don't worry about deriving the gradient when $y_i w^T x_i = 1$ since the function is not differentiable here. Instead just treat the gradient as the same as when $y_i w^T x_i > 1$.

Answer goes here:

The gradient in the first case is just

$$\nabla L_i(w) = w$$

since $\max\{0,\ 1 - y_i w^T x_i\} = 0$. In the second case we have $\max\{0,\ 1 - y_i w^T x_i\} = 1 - y_i w$
so the gradient is

$$\nabla L_i(w) = w - C y_i x_i$$

In [10]:
```python
def grad_loss(w, x_i, y_i, C):
    """
    Compute the gradient of the loss function L_i(w) at training (

    Input:
        w: np.ndarray(p,), vector of SVM parameters
        x_i: np.ndarray(p,), vector of features for one training (
        y_i: float, the label of training example i, must be eith
        C: float, the penalty parameter

    Return:
        g: np.ndarray(p,), the gradient at w of L_i
    """
    s = 1 - y_i * np.dot(x_i, w)
    g = w if s < 0 else w - C * y_i * x_i
    return g
```

Part (c) (10 pts) Implementing `fit_sgd()`

Fill in the skeleton code below which behaves similar to `fit()` from the previous problem but instead optimizes the primal objective function using stochastic gradient descent. You need to do 3 things:

1. Get the random index for the training example. You will find the function `np.random.randint()` useful for this purpose.
2. Implement the SGD update using your training example with a variable learning rate which is
$$\gamma_t = \frac{\mathrm{lr}_0}{t+1}$$
where $\mathrm{lr}_0 > 0$s the initial learning rate.
3. Compute the bias term using your solution w.

In [11]:
```
def fit_sgd(X, y, C, w0, lr0, T):
    """
    Fit the SVM parameters using SGD.

    Input:
        X: np.ndarray(n,p), the training data matrix of features
        y: np.ndarray(n,), the training data vector of labels
        C: float, the penalty parameter
        w0: np.ndarray(p,), the initial parameters for the optimi
        lr0: float, the initial learning rate
        T: int, the number of iterations to perfrom

    Return:
        w: np.ndarray(p,), the SVM parameters after T iterations
        b: float, the SVM bias term
    """

    n = X.shape[0]  # Number of training samples.
    w = np.copy(w0) # Initial point for optimization.

    # Do T iterations.
    for t in range(T):

        # Get the random index for the training example.
        ##TODO-start##
        i = np.random.randint(0, n, 1)
        ##TODO-end##
        x_i = X[i]
        y_i = y[i]

        # Implement the SGD update using your grad_loss function.
        ## TODO-start##
        w = w - lr0 * grad_loss(w, x_i.reshape(2), y_i, C)
        lr0 = lr0 / (t+1)
        ## TODO-end##

    # Compute the bias.  Hint: You have already done this.
    ##TODO-start##
    b = np.mean(y - X @ w)
    ##TODO-end##

    return (w, b)
```

Comparing to the dual solution (nothing to do here)

Run the following 3 code cells to test your implemenation against the solution to the previous problem. You should obtain similar accuracies on both the training and testing sets. To see that the classifiers do the same thing you should also obtain a similar plot to the one from earlier. Note that the cell below may take a minute to run because it is 1 million iterations of SGD.

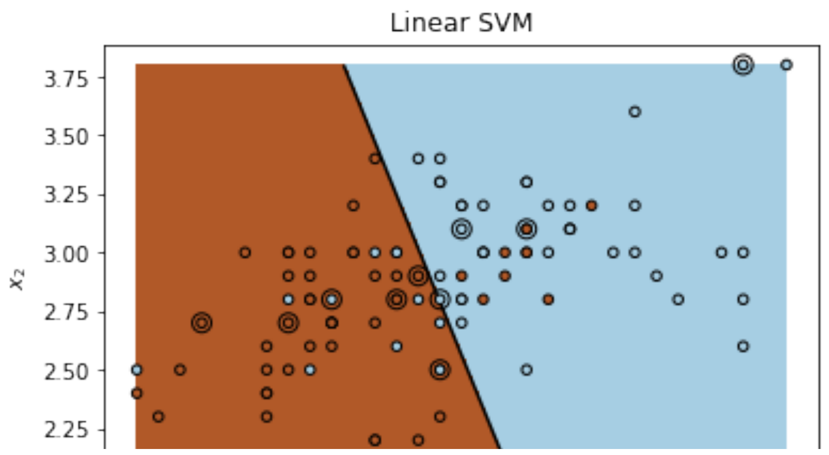
In [12]:
```
w_sgd, b_sgd = fit_sgd(X_train, y_train, 10, np.zeros(X.shape[1]),
```

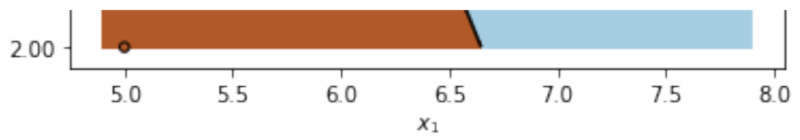
In [13]:
```
y_pred_sgd = predict(X_test, w_sgd, b_sgd)
y_pred_train_sgd = predict(X_train, w_sgd, b_sgd)

print("Training accuracy = {:0.2f}%".format(100*accuracy(y_pred_tr
print("Testing accuracy = {:0.2f}%".format(100*accuracy(y_pred_sgd
```

```
Training accuracy = 74.44%
Testing accuracy = 70.00%
```

In [14]:
```
plt.figure(2)
plt.clf()
plt.scatter(X[:, 0], X[:, 1], c=y, zorder=10, cmap=plt.cm.Paired,
            edgecolor='k', s=20)

# Circle out the test data
plt.scatter(X_test[:, 0], X_test[:, 1], s=80, facecolors='none',
            zorder=10, edgecolor='k')
            zorder=10, edgecolor='k')

plt.axis('tight')
x_min = X[:, 0].min()
x_max = X[:, 0].max()
y_min = X[:, 1].min()
y_max = X[:, 1].max()

XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
XXYY = np.c_[XX.ravel(), YY.ravel()]
ZZ = []
for i in range(XXYY.shape[0]):
    ZZ.append(XXYY[i]@w_sgd+b_sgd)

Z = np.array(ZZ)

# Put the result into a color plot
Z = Z.reshape(XX.shape)
plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
plt.contour(XX, YY, Z, colors=['k', 'k', 'k'],
            linestyles=['--', '-', '--'], levels=[-.5, 0, .5])
plt.title('Linear SVM')
plt.xlabel(r'$x_1$')
plt.ylabel(r'$x_2$')
plt.show()
```

```
/Users/wangt/miniconda3/envs/ds/lib/python3.7/site-packages/ipykernel
_launcher.py:26: MatplotlibDeprecationWarning: shading='flat' when X
and Y have the same dimensions as C is deprecated since 3.3.  Either
specify the corners of the quadrilaterals with X and Y, or pass shadi
ng='auto', 'nearest' or 'gouraud', or set rcParams['pcolor.shading'].
This will become an error two minor releases later.
```





3. Constrained Optimization (15 pts total)

In this problem you will practice writing down the Lagrangians of different constrained optimization problems.

```
### Part (a) (5 pts)

Consider the constrained optimization problem
$$
\min_{x}\ x^TAx, \quad \text{ such that }\quad \|x\|^2 = 1
$$
where $A$ is a real, symmetric, and positive definite matrix. Write down the Lagrangian and the system of equations that the stationary points must satisfy.
```

Answer goes here:

The Lagrangian function is:

$$F(x) = x^T Ax - \lambda(\|x\|^2 - 1)$$

And the system of equations are:

$$\begin{cases} \dfrac{\partial F(x)}{\partial x} = Ax - x\lambda = 0 \\ \dfrac{\partial F(x)}{\partial \lambda} = x^T x - 1 = 0 \end{cases}$$

Part (b) (5 pts)

What are the stationary points of the Lagrangian in terms of the matrix $A$? In particular, what is the minimizer x and minimal value?

Answer goes here:
Let's expand the equations.

$$\frac{\partial F(x)}{\partial x} = (A + A^T)x + 2\lambda x = (A + A^T + 2\lambda I)x = 0$$
$$\frac{\partial F(x)}{\partial \lambda} = x^T x - 1 = 0$$

We already know A is a symmetric matrix, so $A = A^T$. Then,

$$(A + A^T + 2\lambda I)x = (2A + 2\lambda I)x = (A + \lambda I)x = 0$$

Apparently, λ is a eigen value of matrix A. And, x is the eigen vector corresponding to λ of A. Besides, A is a symmetric, positive definite matrix. Its eigen vector x is orthonormal, which satisfies $x^T x = 1$. That means, the solution x^* of equation $\frac{\partial F(x)}{\partial x} = 0$ is the eigen vector of A.

And, we substitute x^* into the equation. Then, we get
$$x^T Ax = x^T(\lambda x) = \lambda x^T x = \lambda$$
The minimal value is λ, a eigen value of A.

Part (c) (5 pts)

Write down the Lagrangian and system of equations for the stationary points for the following constrained minimization problem.
$$\min_{x,y} 10x^2 + 5y(y-1), \quad \text{such that} \quad x + y = 4$$
What is the minimizer (x, y)?

Answer goes here:
First, we define the Lagrangian.

$$F(x, y, \lambda) = 10x^2 + 5y(y-1) + \lambda(x + y - 4)$$

Then, we write down the system of equations.

$$\begin{cases} \dfrac{\partial F(x, y, \lambda)}{\partial x} = 20x + \lambda = 0 \\ \dfrac{\partial F(x, y, \lambda)}{\partial y} = 10y - 5 + \lambda = 0 \\ \dfrac{\partial F(x, y, \lambda)}{\partial \lambda} = x + y - 4 = 0 \end{cases}$$

Next, we calculate the solution of this system of equations. We can get $minimizer(x, y) = (\frac{7}{6}, \frac{17}{6})$

In [15]:
```python
# We can use minimize method for verification.
def obj(g):
    x = g[0]
    y = g[1]
    return 10 * x * x + 5 * y * (y - 1)

def con(g):
    x = g[0]
    y = g[1]
    sum = 4
    return sum - (x + y)

print(minimize(fun=obj, x0=[3, 3], constraints=({'type': 'eq', 'f
print('-' * 50)
print("x = {}, y = {}".format(7/6, 17/6))
```

```
     fun: 39.583333333333333
     jac: array([23.33333397, 23.33333397])
 message: 'Optimization terminated successfully'
    nfev: 14
     nit: 4
    njev: 4
  status: 0
 success: True
       x: array([1.16666667, 2.83333333])
--------------------------------------------------
x = 1.1666666666666667, y = 2.8333333333333335
```

In [ ]:

```

```

In [ ]:

```

```