# ECE M146 - HW 4

April 30, 2020

John Rapp Farnes | 405461225

## 1

### 1.1 (a), (b)

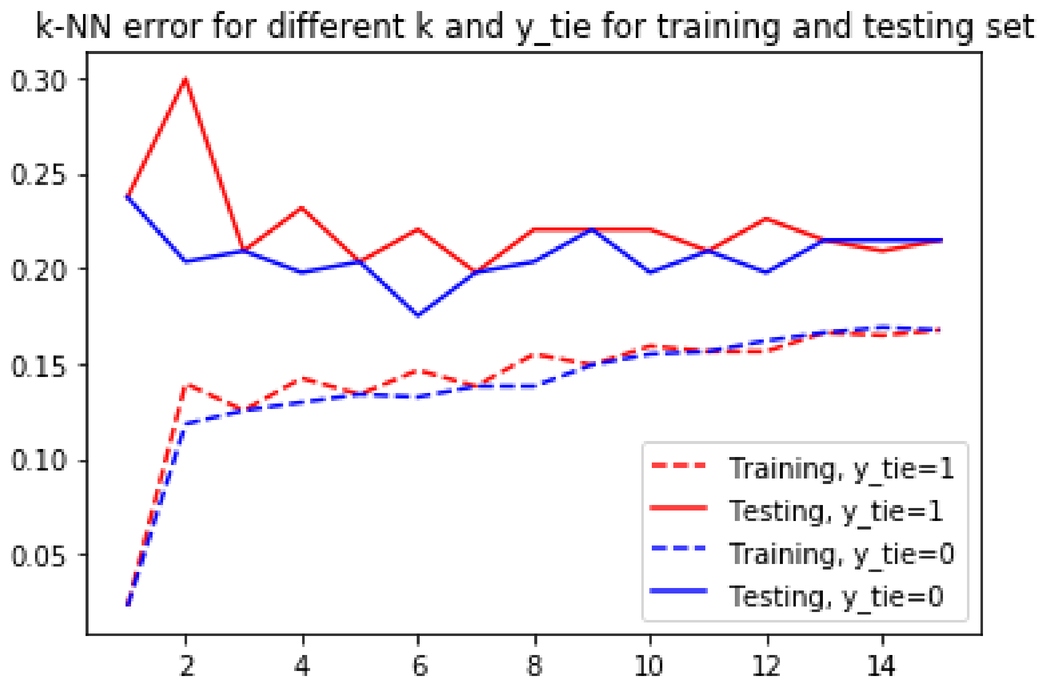
[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import mode
from sklearn.metrics import accuracy_score

filenames = ["dataTesting_X.csv", "dataTesting_Y.csv", "dataTraining_X.csv",␣
 ↪"dataTraining_Y.csv"]

testing_x, testing_y, training_x, training_y = list(map(lambda f : pd.
 ↪read_csv(f, header=None), filenames))
```

[2]:
```
training_y = training_y[0].to_numpy()
testing_y = testing_y[0].to_numpy()

training_x = training_x.to_numpy()
testing_x = testing_x.to_numpy()
```

[3]:
```
def squared_dist(x, y):
    d = x - y
    return np.dot(d, d)

def find_k_min_indices(X, k):
    return np.argpartition(X, k)[:k]

def find_most_common(min_ys):
    frac_of_1 = sum(min_ys == 1) / len(min_ys)

    if frac_of_1 == 0.5: return None, False
    else:
        return (1 if frac_of_1 > 0.5 else 0), True
```

```
def break_odd_tie(min_indices, dists, min_ys):
    min_dists = [dists[i] for i in min_indices]
    closest_dist = np.min(min_dists)

    indices_with_closest_dist = np.flatnonzero(min_dists[i] == closest_dist)
    return min_ys[np.min(indices_with_closest_dist)]

def knn_classifier(x, k, training_x, training_y, y_tie):
    dists = [squared_dist(x, x_t) for x_t in training_x]

    min_indices = find_k_min_indices(dists, k)
    min_ys = training_y[min_indices]

    most_common, has_dominant_value = find_most_common(min_ys)

    if has_dominant_value:
        return most_common
    else:
        even = k % 2 == 0
        return y_tie if even else break_odd_tie(min_indices, dists, min_ys)

def pred_y(X, k, y_tie):
    return [knn_classifier(x, k, training_x, training_y, y_tie) for x in X]
```

[4]:
```
for y_tie in [1, 0]:
    training_err = []
    testing_err = []
    ks = range(1, 16)
    for k in ks:
        training_err.append(1 - accuracy_score(training_y, pred_y(training_x,␣
 ↪k, y_tie)))
        testing_err.append(1 - accuracy_score(testing_y, pred_y(testing_x, k,␣
 ↪y_tie)))

    col = 'red' if y_tie == 1 else 'blue'
    plt.plot(ks, training_err, linestyle = 'dashed', label=f'Training,␣
 ↪y_tie={y_tie}', color = col)
    plt.plot(ks, testing_err, label = f'Testing, y_tie={y_tie}', color = col)

plt.title("k-NN error for different k and y_tie for training and testing set")
plt.legend()
plt.show()
```



## 1.2 (c)

In (a) and (b), I have used $1 -$ accuracy to serve as the error, i.e. the percentage of points that were missclassified.

We can see that even for $k = 1$ on the training set there is an error $\neq 0$, which might be counter intuitive as this classifier "knows"about the tested points already and seeming directly compares to them. This effect is due to there sometimes being duplicate $x$ values, with different labels that may be picked because they have a lower index. In general we can see that the classifier has a lower error on training data than testing data which is intuitive as it already knows of those examples. For the training data, the error increases as $k$ increases, which can be explained by there being a higher chance that a point other than the original point for the tested example is picked for the label as it "competes" with other points. When $y_{\text{tie}} = 0$, there is a slightly lower error for even $k$, this is because of the bias in the data where there are more points labeled 0 than 1, making 0 a better "guess" than 1. Hence, the opposite is true for $y_{\text{tie}} = 1$ on the training data.

Looking at the testing data, the lowest error is acheived for $k = 6$ and $y_{\text{tie}} = 0$. In general, even $k$ have a higher error than neighboring uneven $k$ when $y_{\text{tie}} = 1$ and lower when $y_{\text{tie}} = 0$. This can be explained by the bias in the data as described above. This implies that an even $k$ with the majority class as $y_{\text{tie}}$ may be beneficial if there is such bias in the data. In general, $k$ increasing tend to decrease the error to a certain point (in this example between 5 and 7), after which it starts increasing again. The reason for small $k$, e.g. $k = 1$ or $k = 3$ not being as accurate as slightly higher $k$ is likely because of noise in the data, where very similar kinds of passengers happend to survive or not by chance. This effect is "averaged out" or "smoothed" with a higher $k$.

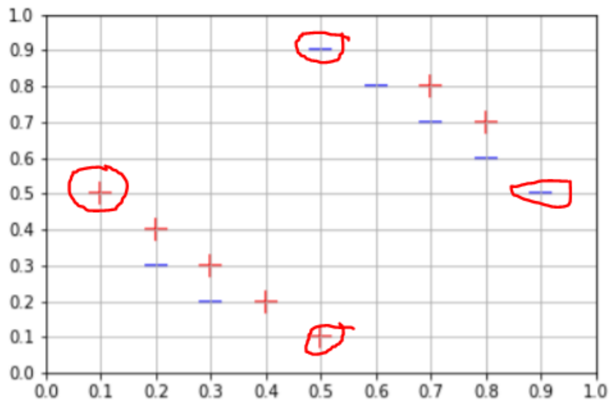
Overall, the best classifier manages to correctly classify >80% of points in the test data, far greater than the most simple "majority class" classifier.

# 2

## 2.1 (a)

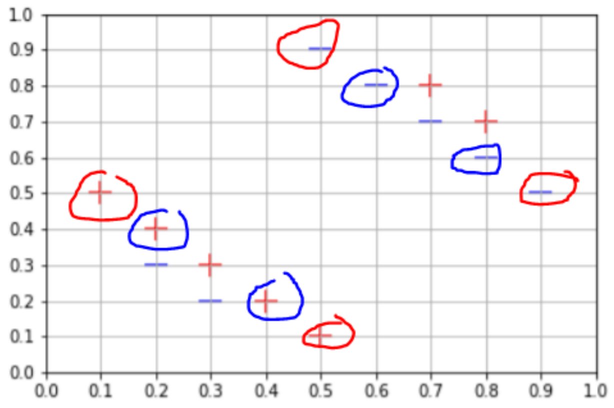
### 2.1.1 Example 1

$k = 1$ Only the following marked points are correctly classified, while the rest are missclassified, as there are points of the other class closer to the points left out.



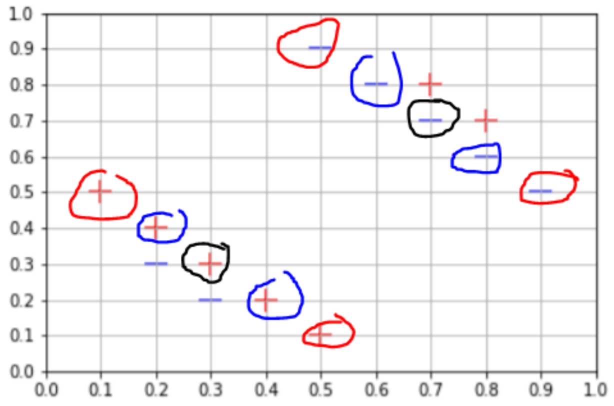
As such, the cross-validation error is $\frac{10}{14}$.

$k = 3$ The result is improved, as four more points are correctly classified (marked in blue):



This time, the cross-validation error is $\frac{6}{14}$.

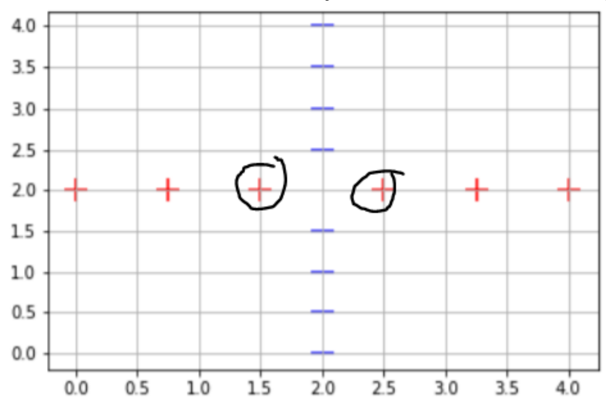
$k \geq 5$ Two additional points are correctly classified (marked in black):



The minority data points are still missclassified. The cross-validation error is $\frac{4}{14}$, which means $k = 5$ and $k = 7$ are both optimal for this example.

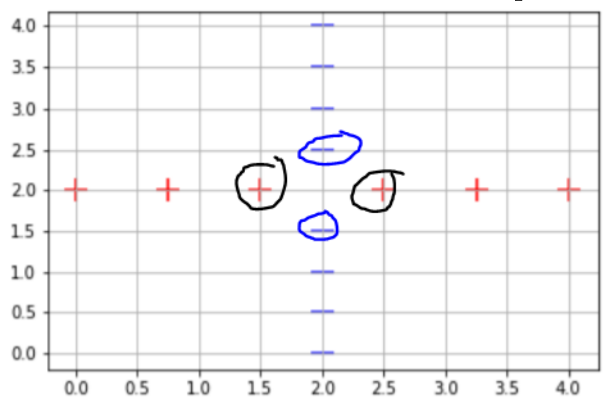
### 2.1.2 Example 2

$k = 1$ Only the following marked points are missclassified:



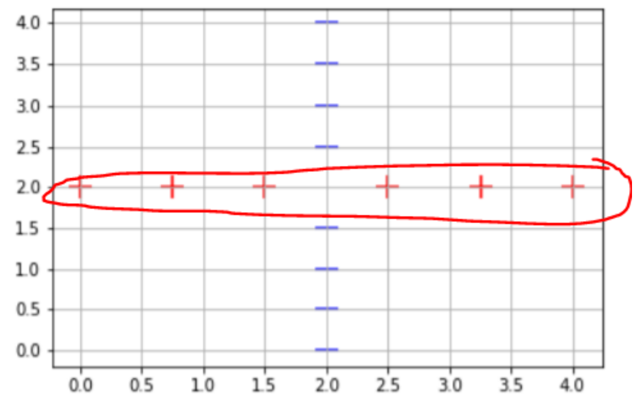
As such, the cross-validation error is $\frac{2}{14}$.

$k = 3$ Two additional points are missclassified (marked in blue):



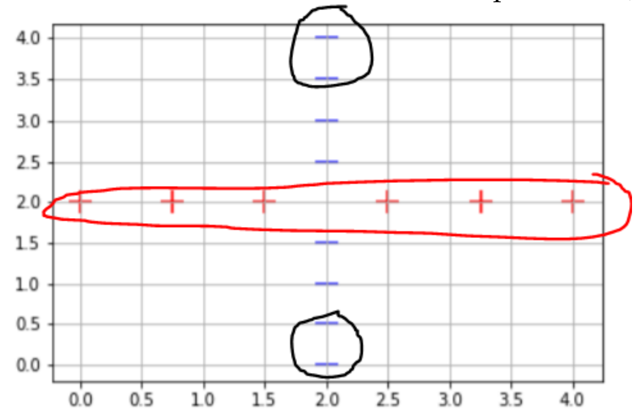
This time, the cross-validation error is $\frac{4}{14}$.

$k = 5$ This time, all the red crosses are missclassified:



As such, the cross-validation error is $\frac{6}{14}$.

$k = 7$ 4 additional points (marked in black) are missclassified:



As such, the cross-validation error is $\frac{10}{14}$.

Comparing the cross-validation error of all tested values of $k$, we can see that $k = 1$ is optimal in this case, with cross-validation error $\frac{2}{14}$.

### 2.1.3 (b)

In general, too big of a $k$ will assign all points in a cluster of points to the majority class in that cluster, leaving out the "fine grained" details. Too small of a $k$ on the other hand might introduce noise, e.g. if some points are misslabeled, leading to overfitting. As seen in (a), the optimal $k$ depend on the structure of the data set.

In Figure 1, the cross-validation error for $k = 1$ is $\frac{10}{14}$, while it is $\frac{4}{14}$ for $k = 13$ (see figures in (a), the figure for $k \geq 5$ also applies to $k = 13$). For $k = 1$, "overfitting" of e.g. the red crosses in the upper right lead to the blue dashes being missclassified in the process, even though there are generally blue dashes in the area. For $k = 13$, $k$ is close to the number of points, and all points are predicted to be in the majority class in this case.

## 3

As $Tr$ is linear and $Tr(A) = Tr(A^T)$, we can rewrite $J$ as

$$\begin{aligned}
J(\tilde{W}) &= \frac{1}{2}Tr((\tilde{X}\tilde{W}T)^T(\tilde{X}\tilde{W}T)) \\
&= \frac{1}{2}Tr((\tilde{W}^T\tilde{X}^T - T^T)(\tilde{X}\tilde{W}T)) \\
&= \frac{1}{2}Tr(\tilde{W}^T\tilde{X}^T\tilde{X}\tilde{W} - \tilde{W}^T\tilde{X}^T T - T^T\tilde{X}\tilde{W} + T^T T) \\
&= \frac{1}{2}\left[Tr(\tilde{W}^T\tilde{X}^T\tilde{X}\tilde{W}) - \underbrace{Tr(\tilde{W}^T\tilde{X}^T T)}_{=Tr(T^T\tilde{X}\tilde{W})} - Tr(T^T\tilde{X}\tilde{W}) + Tr(T^T T)\right]
\end{aligned}$$

Using the gradient rules for $Tr$, we can write the gradient $\nabla J(\tilde{W})$ as such:

$$\begin{aligned}
\nabla J(\tilde{W}) &= \frac{1}{2}\left[\nabla_w Tr(\tilde{W}^T\tilde{X}^T\tilde{X}\tilde{W}) - 2\nabla_w Tr(T^T\tilde{X}\tilde{W}) + \underbrace{\nabla_w Tr(T^T T)}_{=0}\right] \\
&= \frac{1}{2}\left[(\tilde{X}^T\tilde{X} + \tilde{X}^T\tilde{X})\tilde{W} - 2\tilde{X}^T T\right] \\
&= \tilde{X}^T\tilde{X}\tilde{W} - \tilde{X}^T T
\end{aligned}$$

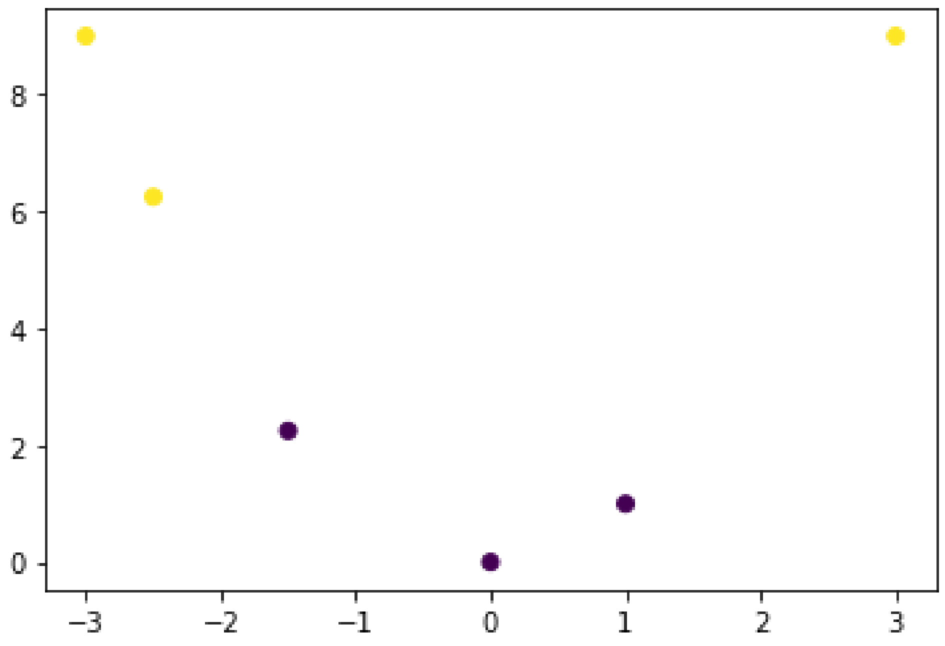
We optimize $J$ by solving $\nabla J = 0$:

$$\begin{aligned}
\nabla J(\tilde{W}) &= 0 && \Leftrightarrow \\
\tilde{X}^T\tilde{X}\tilde{W} &= \tilde{X}^T T && \Leftrightarrow \\
\tilde{W} &= (\tilde{X}^T\tilde{X})^{-1}\tilde{X}^T T
\end{aligned}$$

## 4

### 4.1 (a)

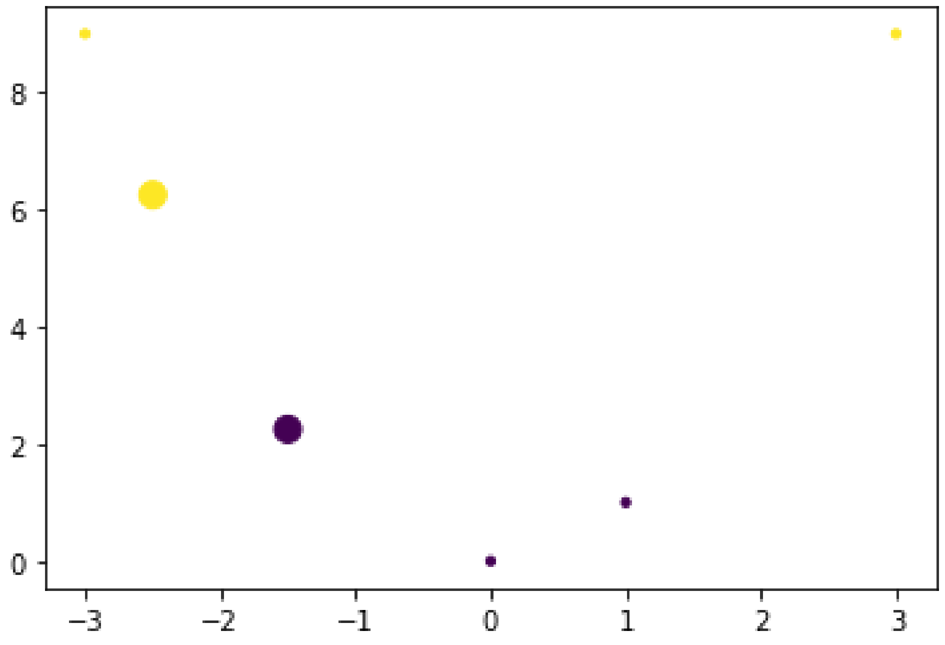
```
[5]: x = np.array([
         [-3,9],
         [-2.5,6.25],
         [3,9],
         [-1.5,2.25],
         [0,0],
         [1,1]
     ])
     y = np.array([1,1,1,-1,-1,-1])

     N = len(x)

     xt = x.transpose()

     plt.scatter(xt[0], xt[1], c = y)
     plt.show()
```



Yes it is linearly separable

## 4.2 (b)

[6]:
```
plt.scatter(xt[0], xt[1], c = y, s = [10,100,10,100,10,10])
plt.show()
```



The support vectors are marked as the bigger dots above. These are the points closest to the "boundary" of the classes.

The lowest margin separating hyperplane is found as the line going perpendicular to the vector between the support vectors, right in the middle between them.

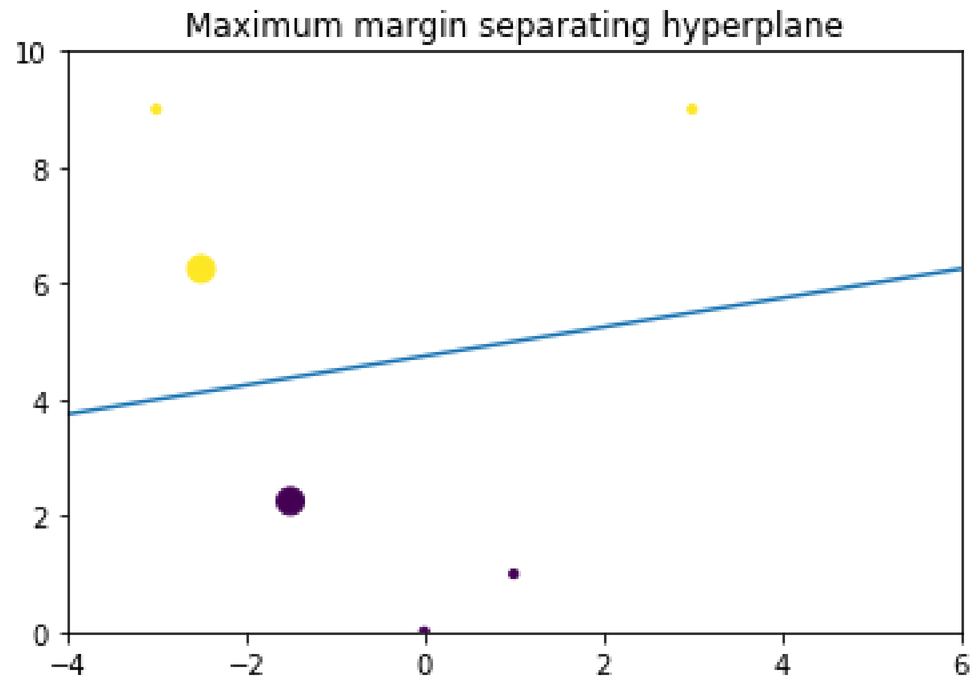
A normalized such $w$ may be found by taking $\frac{s_1 - s_2}{||s_1 - s_2||}$. $b$ is calculated by solving the equation (the distance being the same to both support vectors):

$$w^T x_1 + b = -(w^T x_2 + b) \Leftrightarrow$$
$$b = \frac{1}{2}(-w^T x_2 - w^T x_1)$$

[7]:
```
s1, s2 = np.array([
    [-2.5, 6.25],
    [-1.5, 2.25],
])

w = (s1 - s2) / np.linalg.norm(s1 - s2)
b = 1/2*(-w.dot(s1)-w.dot(s2))

print(f'w = {w}')
print(f'b = {b}')
```

```
plt.scatter(xt[0], xt[1], c = y, s = [10,100,10,100,10,10])

x1 = np.array([-4, 6])
x2 = (-b - x1 * w[0]) / w[1]

plt.plot(x1, x2)
plt.xlim(-4,6)
plt.ylim(0,10)
plt.title("Maximum margin separating hyperplane")
plt.show()
```

```
w = [-0.24253563  0.9701425 ]
b = -4.608176875690327
```



## 4.3 (c)

Let $i \in S = \{1,2\}$ be the indices of the support vectors. We then have $\alpha_i > 0$ for $i \in S$ and $\alpha_i = 0$ else. We have $x_1 = (-2.5, 6.25)^T$, $x_2 = (-1.5, 2.25)^T$, $y_1 = 1$ and $y_2 = -1$.

The dual optimization problem

$$\begin{aligned}\max_\alpha W(\alpha) &= \sum_{i=1}^m \alpha_i - \frac{1}{2}\sum_{i,j=1}^m y^{(i)}y^{(j)}\alpha_i\alpha_j(x^{(i)} \cdot x^{(j)}) \\ \text{s.t. } &\alpha_i \geq 0, i = 1, \dots, m \\ &\sum_{i=1}^m \alpha_i y^{(i)} = 0\end{aligned}$$

simplifies to

$$\begin{aligned}\max_\alpha W(\alpha) &= \alpha_1 + \alpha_1 - \frac{1}{2}(\alpha_1^2 \underbrace{||x_1||^2}_{=2.5^2+6.25^2} + \alpha_2^2 \underbrace{||x_2||^2}_{=1.5^2+2.25^2} - 2\alpha_1\alpha_2 \underbrace{x_1^T x_2}_{=2.5\cdot1.5+6.25\cdot2.25}) \\ \text{s.t. } &\alpha_i \geq 0, i = 1, \dots, m \\ &\alpha_1 = \alpha_2\end{aligned}$$

let $a = \alpha_1 = \alpha_2$, we have $W(a) = 2a - \frac{a^2}{2}(||x_1||^2 + ||x_2||^2 - 2x_1^T x_2)$. We can then find the minimum by a stationary point of $W(\alpha)$, as it is of a single variable and unconstrained.

$$\begin{aligned}\frac{dW}{da} &= 0 & \Leftrightarrow \\ 2 - (||x_1||^2 + ||x_2||^2 - 2x_1^T x_2)a &= 0 & \Leftrightarrow \\ a &= \frac{2}{||x_1||^2 + ||x_2||^2 - 2x_1^T x_2} & \implies \\ a &\approx 0.118\end{aligned}$$

Further, we have $w = \sum_n \alpha_n y^{(n)} x_n = a(-2.5, 6.25)^T - a(-1.5, 2.25)^T \approx (-0.118, 0.472)^T$ by the optimum critera.

Finally, we can calculate $b$ by

$$\begin{aligned}b &= \frac{1}{|S|}\sum_{n \in S}[y^{(n)} - (\sum_{k \in S}\alpha_n y^{(n)} x_n)^T x_n] \\ &= \frac{1}{2}[1 - \underbrace{w^T x_1}_{\approx 3.245} - 1 - \underbrace{w^T x_2}_{\approx 1.239}] \\ &\approx -2.242\end{aligned}$$

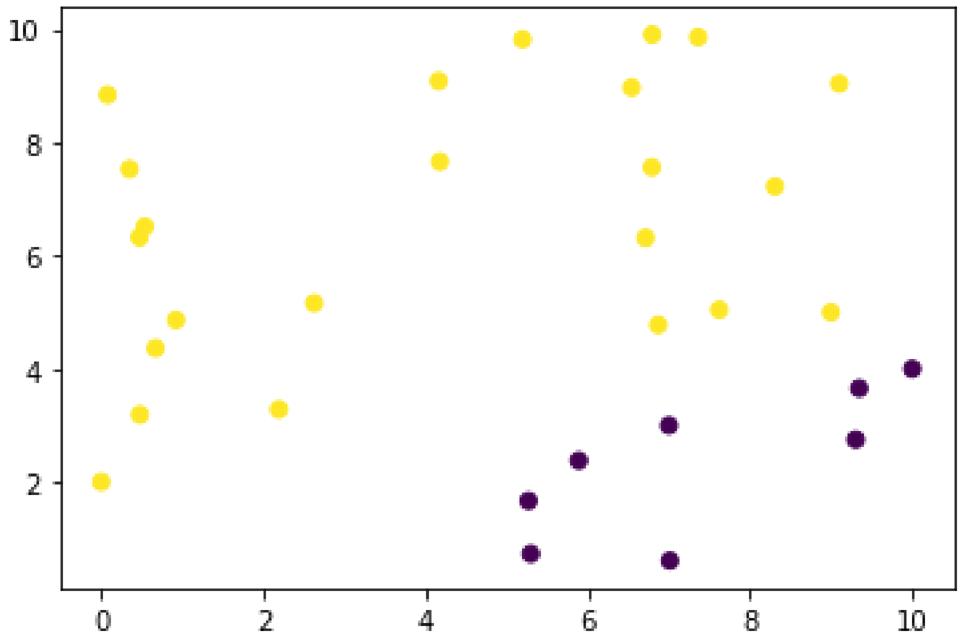
We have that the $w$ aquired from this approach is $\frac{1}{2}$ times the $w$ aquired in (b), and the same goes for $b$. As such, the solution in (b) is a scaled version of this solution, both being valid solutions to the problem.

## 5

### 5.1 (a)

[8]:
```
data = pd.read_csv("Data.csv", header=None)
x = np.array(data[[0, 1]])
y = np.array(data[2])

N = len(x)

xt = x.transpose()
```

[9]:
```
plt.scatter(xt[0], xt[1], c = y)
plt.show()
```

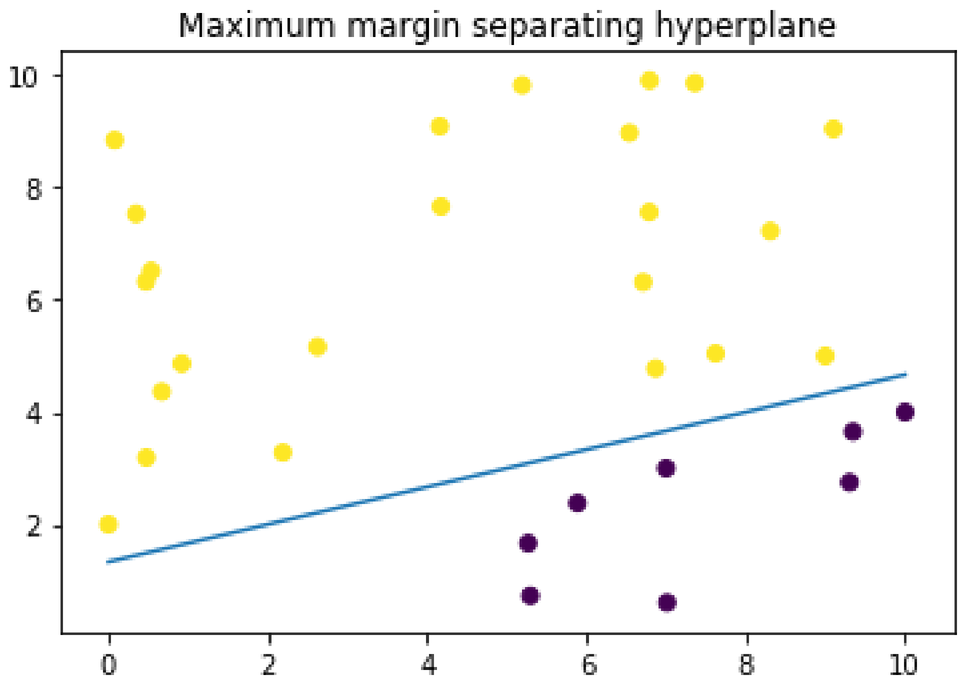


Yes it is linearly separable.

### 5.2 (b)

[10]:
```
import cvxpy as cp

w = cp.Variable(2)
b = cp.Variable()

objective = cp.Minimize(1/2*cp.sum_squares(w))
constraints = [y[i]*(w.T @ x[i] + b) >= 1 for i in range(N)]
prob = cp.Problem(objective, constraints)

prob.solve()

print(f'w = {w.value}')
print(f'b = {b.value}')

x1 = np.array([0, 10])
x2 = (-np.array(b.value) - x1 * w.value[0]) / w.value[1]

plt.plot(x1, x2)
plt.scatter(xt[0], xt[1], c = y)
plt.title("Maximum margin separating hyperplane")
plt.show()
```

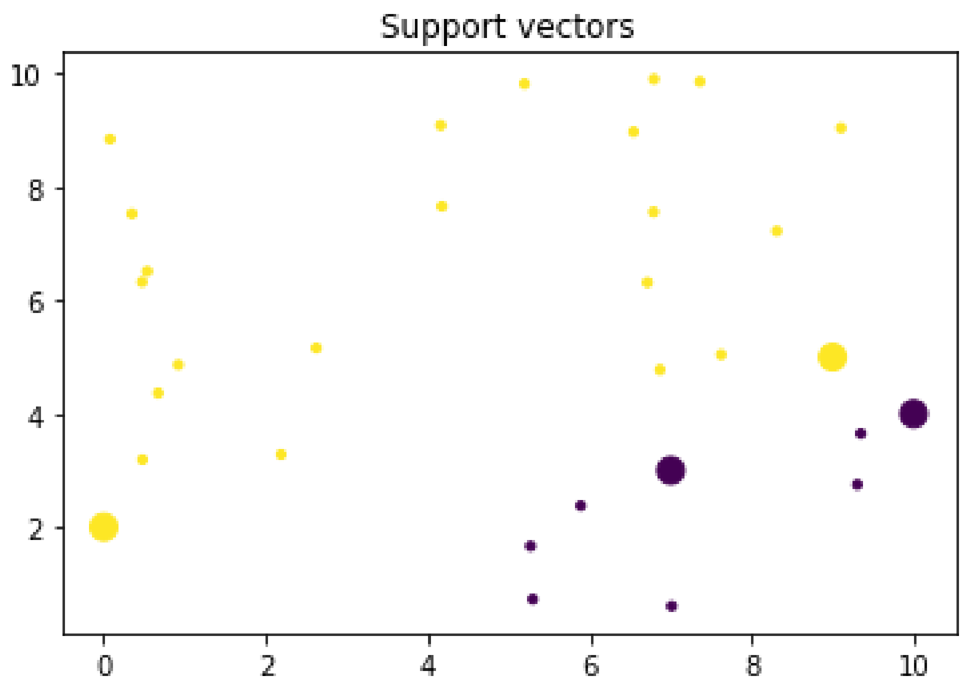
```
w = [-0.5  1.5]
b = -1.9999999999999993
```



### 5.3 (c)

[11]:
```python
alpha = cp.Variable(N)

P = []
for i in range(N):
    P.append([])
    for j in range(N):
        P[i].append(y[i]*y[j]*np.dot(x[i], x[j]))

# P += 1e-13 * np.eye(N)

objective = cp.Maximize(cp.sum(alpha) - 1/2*cp.quad_form(alpha, P))

constraints = [alpha >= 0, alpha.T @ y == 0]

prob = cp.Problem(objective, constraints)
prob.solve()

a = (abs(alpha.value) > 1e-9) * alpha.value

print(f'alpha = {a}')

plt.scatter(xt[0], xt[1], c = y, s = [100 if a[i] > 0 else 10 for i in␣
 ↪range(len(a))])
plt.title("Support vectors")
plt.show()
```

```
alpha = [-0.          -0.          -0.          -0.          -0.          -0.
 -0.          -0.          -0.          -0.          -0.          -0.
 -0.          -0.          -0.          -0.          -0.          -0.
 -0.          -0.          -0.          -0.          -0.          -0.
 -0.          -0.          -0.           1.03178447   0.21821553   0.26059482
  0.98940518]
```



There are 4 support vectors, marked as large dots in the plot above.