index

March 24, 2022

1 Building an SVM from Scratch - Lab

1.1 Introduction

In this lab, you'll program a simple Support Vector Machine from scratch!

1.2 Objectives

In this lab you will:

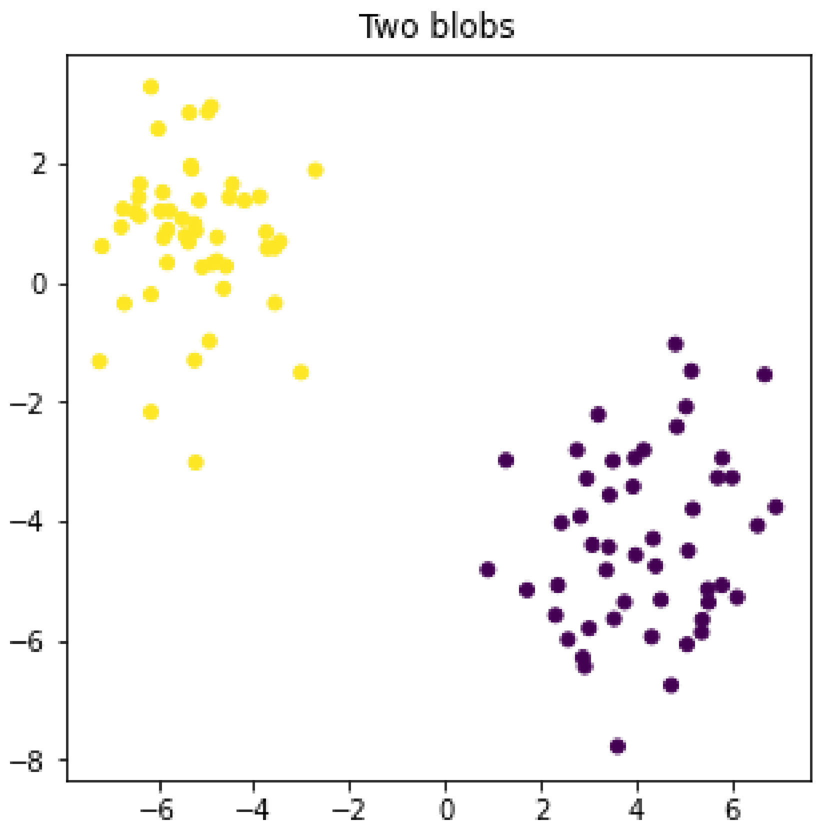
- Build a simple linear max margin classifier from scratch
- Build a simple soft margin classifier from scratch

1.3 The data

Support Vector Machines can be used for any n-dimensional feature space. However, for this lab, you'll focus on a more limited 2-dimensional feature space so that you can easily visualize the results.

Scikit-learn has an excellent set of dataset generator functions. One of them is `make_blobs()`. Below, you can find the code to create two blobs using the `make_blobs()` function. Afterward, you'll use this data to build your own SVM from scratch!

[159]:
```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

plt.figure(figsize=(5, 5))

plt.title('Two blobs')
X, labels = make_blobs(n_features=2, centers=2, cluster_std=1.25, ␣
 ↪random_state=123)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=25);
```



[160]:
```
import pandas as pd
df_X = pd.DataFrame(X, columns = ['x','y'])
df_y = pd.DataFrame(labels, columns = ["labels"])

df_XY = pd.concat([df_X, df_y], axis = 1)


df_XY_2 = pd.DataFrame(dict(x=X[:,0], y=X[:,1], labels=labels))
df_XY_2
```

[160]:
```
           x         y  labels
0   5.063265 -6.063064       0
1   6.109024 -5.274792       0
2   3.425176 -4.434750       0
3  -7.227832 -1.319790       1
4   5.088712 -4.494258       0
..       …         …       …
95 -4.930288 -0.980467       1
96  2.882488 -6.284667       0
97 -5.213493  0.868648       1
```

```
98 -3.741399  0.847325       1
99  3.206133 -2.212918       0

[100 rows x 3 columns]
```

1.4 Build a Max Margin classifier

Since you are aiming to maximize the margin between the decision boundary and the support vectors, creating a support vector machine boils down to solving a convex optimization problem. As such, you can use the Python library `cvxpy` to do so. More information can be found here.

You may have not used `cvxpy` before, so make sure it is installed on your local computer using `pip install cvxpy`.

The four important commands to be used here are:

- `cp.Variable()` where you either don't include anything between `()` or, if the variable is an array with multiple elements, the number of elements.
- `cp.Minimize()` or `cp.Maximize()`, with the element to be maximized passed in as a paramater.
- `cp.Problem(objective, constraints)`, the objective is generally a stored minimization or maximization objective, and the constraints are listed constraints. Constraints can be added by a "+" sign.
- Next, you should store your `cp.Problem` in an object and use `object.solve()` to solve the problem.

Recall that we're trying to solve this problem:

\$ w x^{(i)} + b 1\$ if $y^{(i)} = 1$

\$ w x^{(i)} + b -1\$ if $y^{(i)} = -1$

And, the objective function you're maximizing is $\frac{2}{\|w\|}$. To make things easier, you can instead minimize $\|w\|$

Note that $y^{(i)}$ is the class label. Take a look at the labels by printing them below.

[161]:
```python
# Print labels
labels
```

[161]:
```
array([0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1,
       1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1,
       1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1,
       0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1,
       1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0])
```

Before you start to write down the optimization problem, split the data in the two classes. Name them `class_1` and `class_2`.

[162]:
```python
# Assign label 0 to class_1
class_1 = df_XY[df_XY["labels"] == 0][["x", "y"]]
```

```
# Assign label 1 to class_2
class_2 = df_XY[df_XY["labels"] == 1][["x", "y"]]

class_1["x"]
```

```
[162]: 0     5.063265
       1     6.109024
       2     3.425176
       4     5.088712
       6     6.686796
       7     3.609859
       9     5.793621
       12    5.382139
       13    2.311777
       15    4.731952
       16    2.362033
       19    5.498430
       20    2.922676
       23    6.538275
       25    1.275508
       27    5.152804
       28    4.155678
       29    2.759591
       30    4.322224
       31    4.514933
       32    2.829963
       34    5.042767
       35    3.510620
       36    4.819715
       39    3.374406
       41    3.013806
       46    3.754298
       47    4.403634
       49    6.005574
       50    5.367141
       51    2.573674
       55    3.535936
       56    4.851094
       60    2.963498
       61    3.966488
       66    4.352620
       68    3.440759
       71    0.896035
       72    3.932941
       74    5.798939
       76    5.184451
```

```
78    3.986246
81    6.919840
84    3.080776
85    5.701008
89    2.431507
92    1.714967
93    5.511804
96    2.882488
99    3.206133
Name: x, dtype: float64
```

[163]:
```
## FromGitHub
G_class_1 = X[labels == 0]
G_class_2 = X[labels == 1]

G_class_1.shape
```

[163]:
```
(50, 2)
```

Next, you need to find a way to create a hyperplane (in this case, a line) that can maximize the difference between the two classes. Here's a pseudocode outline: - First, `import cvxpy as cp` - Next, define the variables. note that `b` and `w` are variables (What are the dimensions?) - Then, build the constraints (You have two constraints here) - After that, use "+" to group the constraints together - The next step is to define the objective function - After that, define the problem using `cp.Problem()` - Solve the problem using `.solve()` - Finally, print the problem status (however you defined the problem, and attach `.status`)

[166]:
```
# Import cvxpy
import cvxpy as cp
import warnings
warnings.filterwarnings('ignore')



# Define the variables

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





# Define the constraints
x_constraint = [w @ class_1.iloc[i, [0,1]] + b >= 1  for i in range(class_1.
 ↪shape[0])]
```

```
y_constraint = [w @ class_2.iloc[i, [0,1]] + b <= -1  for i in range(class_2.
↪shape[0])]

# x_constraint = [w.T @ G_class_1[i] + b >= 1  for i in range(class_1.shape[0])]
# y_constraint = [w.T @ class_2.iloc[i, [0,1]] + b <= 1  for i in range(class_2.
↪shape[0])]



# Sum the constraints
total_constraint = x_constraint + y_constraint

# Define the objective. Hint: use cp.norm

obj = cp.Minimize(cp.norm(w,2))

# Add objective and constraint in the problem

prob = cp.Problem(obj, total_constraint)

# Solve the problem
prob.solve()
print('Problem Status: %s'%prob.status)
```

```
Problem Status: optimal
```

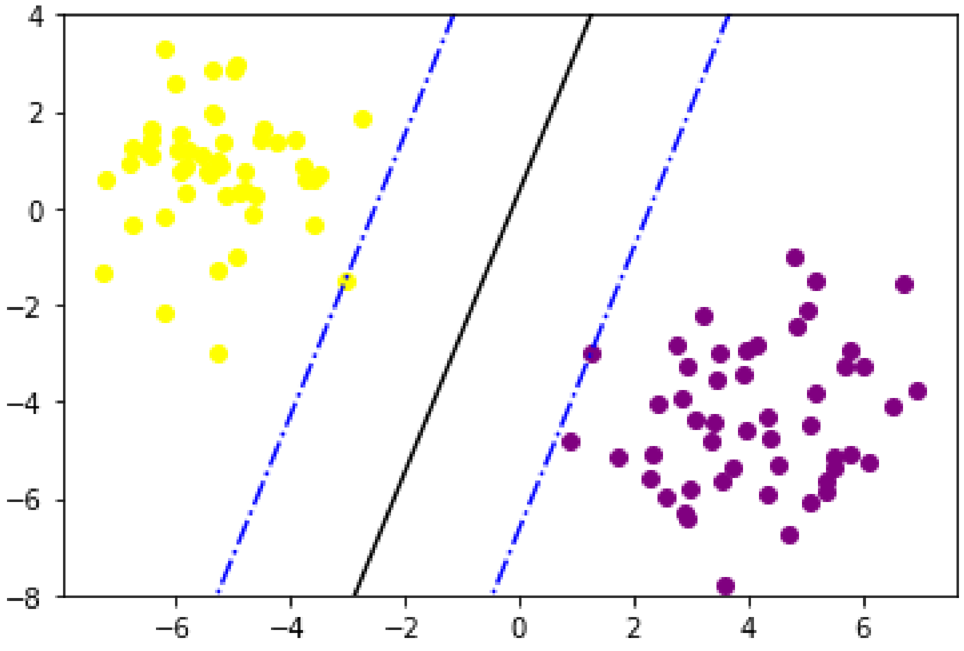
Great! Below is a helper function to assist you in plotting the result of your SVM classifier.

[167]:
```
## A helper function for plotting the results, the decision plane, and the␣
↪supporting planes

def plotBoundaries(x, y, w, b):
    # Takes in a set of datapoints x and y for two clusters,



    d1_min = np.min([x[:,0], y[:,0]])
    d1_max = np.max([x[:,0], y[:,0]])

    # Line form: (-a[0] * x - b ) / a[1]

    d2_at_mind1 = (-w[0]*d1_min - b ) / w[1]
    d2_at_maxd1 = (-w[0]*d1_max - b ) / w[1]
    sup_up_at_mind1 = (-w[0]*d1_min - b + 1 ) / w[1]
    sup_up_at_maxd1 = (-w[0]*d1_max - b + 1 ) / w[1]
    sup_dn_at_mind1 = (-w[0]*d1_min - b - 1 ) / w[1]
    sup_dn_at_maxd1 = (-w[0]*d1_max - b - 1 ) / w[1]

    # Plot the clusters!
    ### Original Code is here
    plt.scatter(x[:,0], x[:,1], color='purple')
    plt.scatter(y[:,0], y[:,1], color='yellow')




    plt.plot([d1_min,d1_max], [d2_at_mind1, d2_at_maxd1], color='black')
    plt.plot([d1_min,d1_max], [sup_up_at_mind1, sup_up_at_maxd1],'-.',␣
↪color='blue')
    plt.plot([d1_min,d1_max], [sup_dn_at_mind1, sup_dn_at_maxd1],'-.',␣
↪color='blue')
    plt.ylim([np.floor(np.min([x[:,1],y[:,1]])), np.ceil(np.max([x[:,1], y[:
↪,1]]))])
```

Use the helper function to plot your result. To get the values of `w` and `b`, use the `.value` attribute.

[168]:
```
w = w.value
b = b.value
```

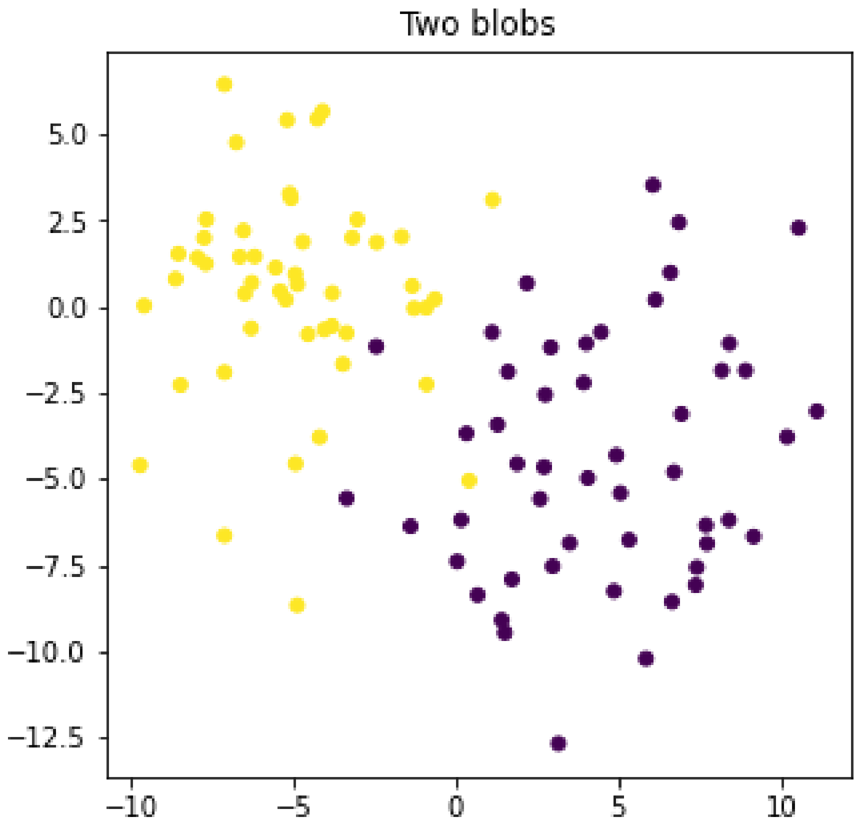
[171]:
```
# Plot
plotBoundaries(G_class_1, G_class_2, w, b)
```



1.5 A more complex problem

Now, take a look at another problem by running the code below. This example will be a little trickier as the two classes are not perfectly linearly separable.

[172]:
```
plt.figure(figsize=(5, 5))

plt.title('Two blobs')
X, labels = make_blobs(n_features=2, centers=2, cluster_std=3, ␣
 ↪random_state=123)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=25);
```



Copy your optimization code from the Max Margin Classifier and look at the problem status. What do you see?

[174]:
```
# Copy the optimization code from above

import warnings
warnings.filterwarnings('ignore')

class_1 = X[labels == 0]
class_2 = X[labels == 1]

d = 2
m = 50
n = 50
```

```
# Define the variables
w = cp.Variable(d)
b = cp.Variable()

# Define the constraints
x_constraints = [w @ class_1[i] + b >= 1  for i in range(class_1.shape[0])]
y_constraints = [w @ class_2[i] + b <= -1 for i in range(class_2.shape[0])]

# Sum the constraints
constraints = x_constraints +  y_constraints

# Define the objective. Hint: use cp.norm
obj = cp.Minimize(cp.norm(w,2))

# Add objective and constraint in the problem
prob = cp.Problem(obj, constraints)

# Solve the problem
prob.solve()
print('Problem Status: %s'%prob.status)
```

```
Problem Status: infeasible
```

1.5.1 What's happening?

The problem status is "infeasible". In other words, the problem is not linearly separable, and it is impossible to draw one straight line that separates the two classes.

1.6 Build a Soft Margin classifier

To solve this problem, you'll need to "relax" your constraints and allow for items that are not correctly classified. This is where the Soft Margin classifier comes in! As a refresher, this is the formulation for the Soft Margin classifier:

$$b + w_T x^{(i)} \geq 1 - \xi^{(i)} \text{ if } y^{(i)} = 1$$

$$b + w_T x^{(i)} \leq -1 + \xi^{(i)} \text{ if } y^{(i)} = -1$$

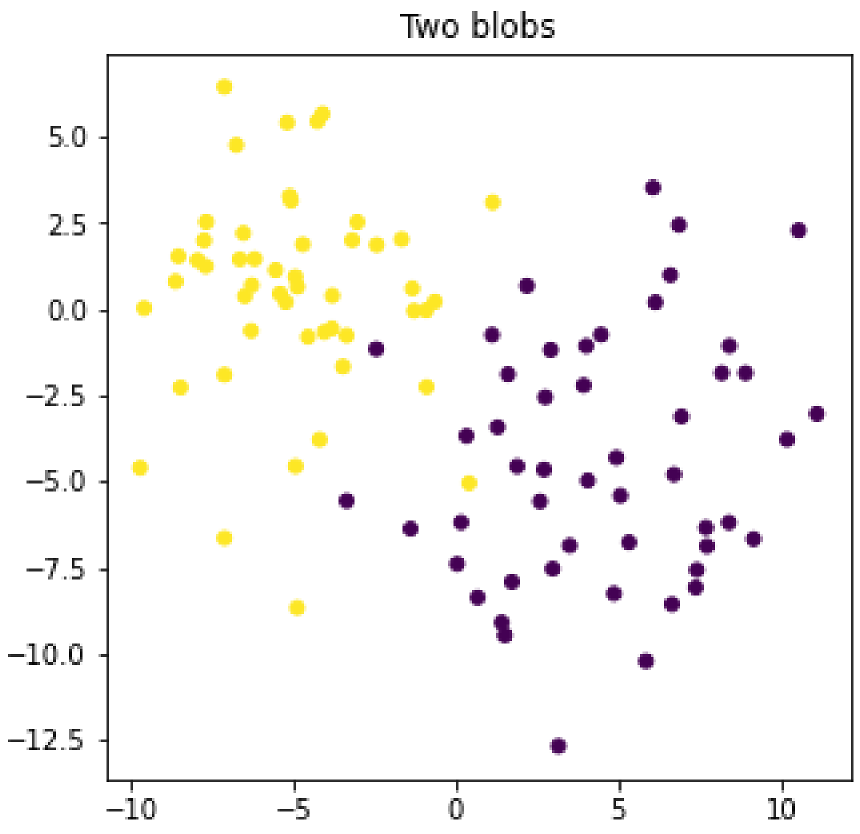
The objective function is

$$\frac{1}{2}\|w\|^2 + C(\sum_i \xi^{(i)})$$

Use the code for the SVM optimization again, but adjust for the slack parameters ξ (ksi or xi).

Some important things to note: - Every ξ needs to be positive, that should be added as constraints - Your objective needs to be changed as well - Allow for a "hyperparameter" C which you set to 1 at first and you can change accordingly. Describe how your result changes

[175]:
```
plt.figure(figsize=(5, 5))

plt.title('Two blobs')
X, labels = make_blobs(n_features=2, centers=2, cluster_std=3, ␣
 ↪random_state=123)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=25);
```



[177]:
```
# Reassign the class labels
class_1 = X[labels == 0]
class_2 = X[labels == 1]
```

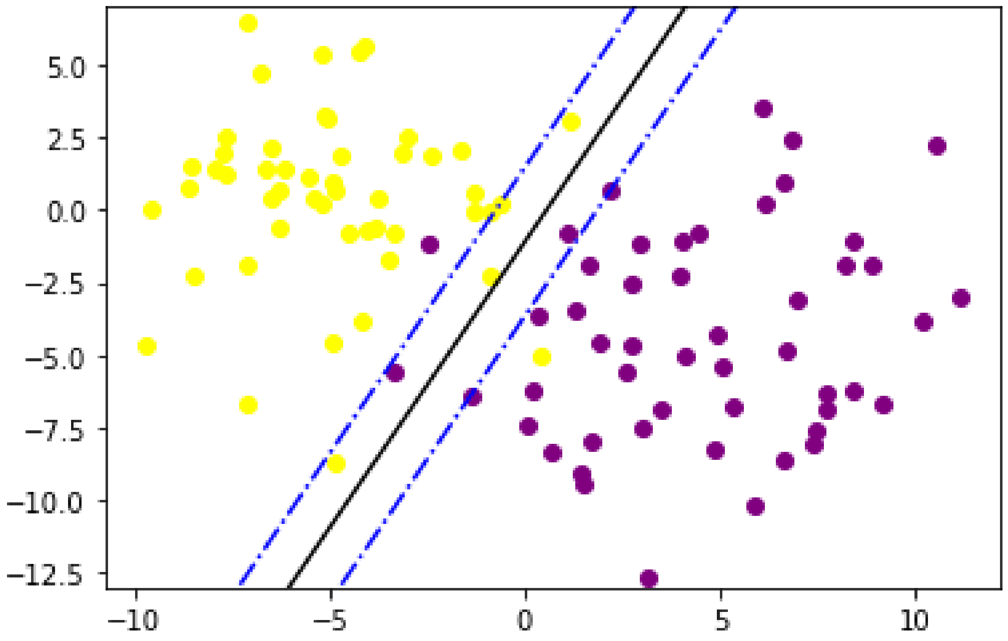
[189]:
```
import cvxpy as cp
import warnings
warnings.filterwarnings('ignore')
```

```
# Define the variables

w_d = 2
xi_1_d = class_1.shape[0]
xi_2_d = class_2.shape[0]
w = cp.Variable(w_d)
b = cp.Variable()

xi_1 = cp.Variable(xi_1_d)
xi_2 = cp.Variable(xi_2_d)


C = 1


# Define the constraints
x_constraint = [w @ class_1[i] + b >= 1 - xi_1[i]  for i in range(xi_1_d)]
y_constraint = [w @ class_2[i] + b <= -1 + xi_2[i] for i in range(xi_2_d)]
xi_1_constraint = [xi_1 >= 0 for i in range(xi_1_d)]
xi_2_constraint = [xi_2 >= 0 for i in range(xi_2_d)]

# Sum the constraints

constraints = x_constraint + y_constraint + xi_1_constraint + xi_2_constraint


# Define the objective. Hint: use cp.norm. Add in a C hyperparameter and assume␣
↪1 at first

obj = cp.Minimize(cp.norm(w,2) + C * (sum(xi_1) + sum(xi_2)))

# Add objective and constraint in the problem

prob = cp.Problem(obj, constraints)



# Solve the problem
prob.solve()
print('Problem Status: %s'%prob.status)
```

```
Problem Status: optimal
```

Plot your result again.

[190]:
```
# Your code here
w = w.value
b = b.value

plotBoundaries(class_1, class_2, w, b)
```



Now go ahead and experiment with the hyperparameter C (making it both larger and smaller than 1). What do you see?

1.7 Summary

Great! You now understand the rationale behind support vector machines. Wouldn't it be great to have a library that did this for you? Well, you're lucky: scikit-learn has an SVM module that automates all of this. In the next lab, you'll take a look at using this pre-built SVM tool!