learn-co-curriculum / **dsc-building-an-svm-from-scratch-lab**

No description, website, or topics provided.

10 commits | 3 branches | 0 packages | 0 releases | 3 contributors | View license

Branch: solution | New pull request | Create new file | Upload files | Find file | Clone or download

This branch is 6 commits ahead, 7 commits behind master. Pull request | Compare

sumedh10 update readme — Latest commit 16e6481 21 days ago

index_files	update readme	21 days ago
.gitignore	Create .gitignore	last year
.learn	updating title	11 months ago
CONTRIBUTING.md	review done master	last year
LICENSE.md	review done master	last year
README.md	update readme	21 days ago
index.ipynb	update readme	21 days ago

README.md

Building an SVM from Scratch - Lab

Introduction

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

Objectives

In this lab you will:

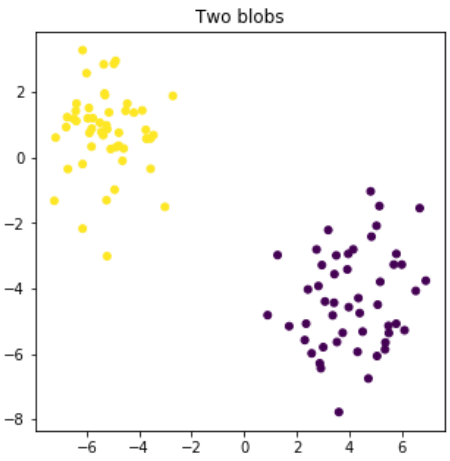
- Build a simple linear max margin classifier from scratch
- Build a simple soft margin classifier from scratch

The data

Support Vector Machines can be used on 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 dataset generator. One of them is `make_blobs()`. Below, you can find the code to create two blobs using the `make_blobs()` function. Afterwards, you'll use this data to build your own SVM from scratch!

```
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);
```



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 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 antything between `()` or, if the variable is an array with multiple elements, the number of elements.
- `cp.Minimize()` or `cp.Maximize()`, with between the parentheses the element to be maximized.
- `cp.Problem(objective, constraints)`, the objective is generally a stored minimization or maximization objective, 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 optimization problem.

Recall that we're trying to solve this problem:

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

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

And as an objective function you're maximizing $\dfrac{2}{\lVert w \rVert}$. To make things easier, you can instead minimize $\lVert w \rVert$

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

```
labels
```

```
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`.

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

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`)

```
import cvxpy as cp

d = 2
m = 50
n = 50

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

# Define the constraints
x_constraints = [w.T * class_1[i] + b >= 1  for i in range(m)]
y_constraints = [w.T * class_2[i] + b <= -1 for i in range(n)]

# 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)
```

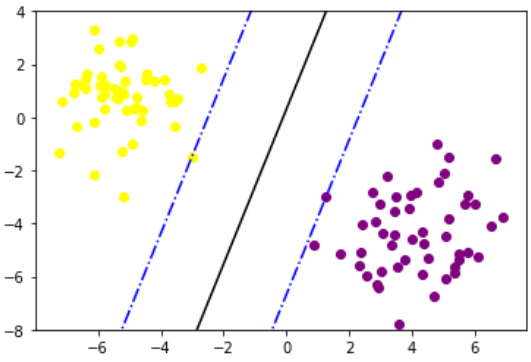
```
Problem Status: optimal
```

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

```
## 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!
    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.

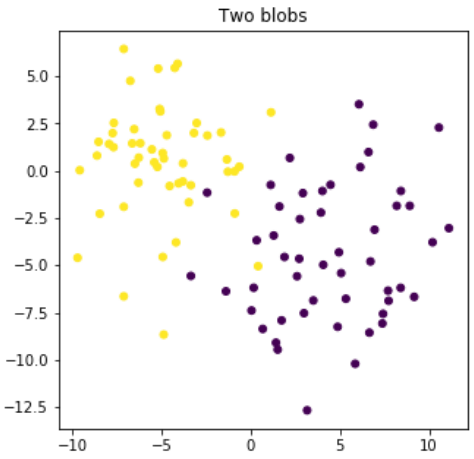
```
w = w.value
b = b.value


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



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.

```
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?

```
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.T * class_1[i] + b >= 1  for i in range(m)]
y_constraints = [w.T * class_2[i] + b <= -1 for i in range(n)]

# 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
```

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.

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:

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

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

The objective function is

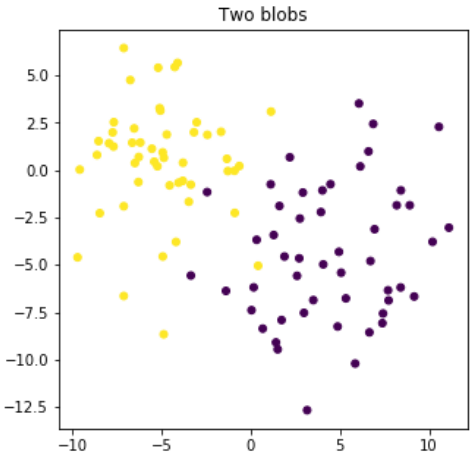
$$\dfrac{1}{2}\lVert w \rVert^2+ C(\sum_i \xi^{(i)})$$

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

```
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);
```



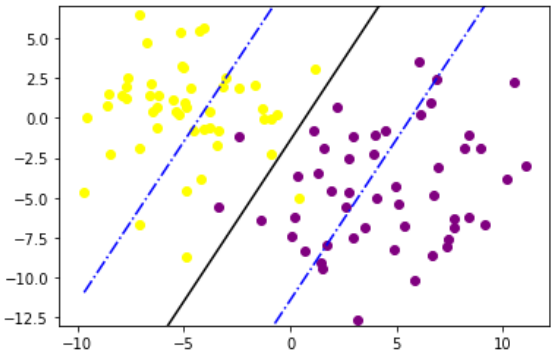

```
# Reassign the class labels
class_1 = X[labels == 0]
class_2 = X[labels == 1]



d = 2
m = 50
n = 50

# Define the variables
w = cp.Variable(d)
b = cp.Variable()
ksi_1 = cp.Variable(m)
ksi_2 = cp.Variable(n)

C=0.01

# Define the constraints
x_constraints = [w.T * class_1[i] + b >= 1 - ksi_1[i]  for i in range(m)]
y_constraints = [w.T * class_2[i] + b <= -1 + ksi_2[i] for i in range(n)]
ksi_1_constraints = [ksi_1 >= 0  for i in range(m)]
ksi_2_constraints = [ksi_2 >= 0  for i in range(n)]

# Sum the constraints
constraints = x_constraints +  y_constraints + ksi_1_constraints + ksi_2_constraints

# 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(ksi_1) + sum(ksi_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

```
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?

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 which automates all of this. In the next lab, you'll take a look at using this pre-built SVM tool!