worksheet_16

April 2, 2024

1 Worksheet 16

Name: **Bowen Li**
UID: **U79057147**

1.0.1 Topics

- Support Vector Machines (Non-linear case)

1.1 Support Vector Machines

Follow along in class to implement the perceptron algorithm and create an animation of the algorithm.

a) As we saw in class, the form

$$w^T x + b = 0$$

while simple, does not expose the inner product `<x_i, x_j>` which we know `w` depends on, having done the math. This is critical to applying the "kernel trick" which allows for learning non-linear decision boundaries. Let's modify the above algorithm to use the form

$$\sum_i \alpha_i < x_i, x > +b = 0$$

[2]:
```
def polynomial(x_i, x_j, c, n):
    return (np.dot(x_i, x_j) + c) ** n
```

[7]:
```
import numpy as np
from PIL import Image as im
import matplotlib.pyplot as plt
import sklearn.datasets as datasets

TEMPFILE = "temp.png"
CENTERS = [[0, 1], [1, 0]]

epochs = 100
learning_rate = .01
expanding_rate = .99
retracting_rate = 1.1
```

```python
#X, labels = datasets.make_blobs(n_samples=10, centers=CENTERS, cluster_std=0.
↪2, random_state=0)
X = np.array([[0,0], [0,1], [1,0], [1,1]])
labels = np.array([1, 0, 0, 1])

Y = np.array(list(map(lambda x : -1 if x == 0 else 1, labels.tolist())))



alpha_i = np.zeros((len(X),))
b = 0

def snap(x, alpha_i, b, error):
    # create a mesh to plot in
    h = .01  # step size in the mesh
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    meshData = np.c_[xx.ravel(), yy.ravel()]
    cs = np.array([x for x in 'gb'])
    fig, ax = plt.subplots()
    ax.scatter(X[:,0],X[:,1],color=cs[labels].tolist(), s=50, alpha=0.8)

    if error:
        ax.add_patch(plt.Circle((x[0], x[1]), .12, color='r',fill=False))
    else:
        ax.add_patch(plt.Circle((x[0], x[1]), .12, color='y',fill=False))

    Z = predict_many(alpha_i, b, meshData)
    Z = np.array([0 if z <=0 else 1 for z in Z]).reshape(xx.shape)
    ax.contourf(xx, yy, Z, alpha=.5, cmap=plt.cm.Paired)
    fig.savefig(TEMPFILE)
    plt.close()
    return im.fromarray(np.asarray(im.open(TEMPFILE)))

def predict_many(alpha_i, b, Z):
    res = []
    for i in range(len(Z)):
        res.append(predict(alpha_i, b, Z[i]))
    return np.array(res)

def predict(alpha_i, b, x):
    kernels = np.array([polynomial(X[j], x, 5, 4) for j in range(X.shape[0])])
    return alpha_i.T @ kernels + b
```

```python
images = []
for epoch in range(epochs):
    # pick a point from X at random
    i = np.random.randint(0, len(X))
    error = False
    x, y = X[i], Y[i]

    y_pred = predict(alpha_i, b, x)
    if y_pred * y > 0:
        if -1 < y_pred < 1:
            alpha_i[i] += y * learning_rate
            alpha_i *= retracting_rate
            b += y * learning_rate * retracting_rate
        else:
            alpha_i *= expanding_rate
            b *= expanding_rate
    else:
        error = True
        alpha_i[i] += y * learning_rate
        alpha_i *= expanding_rate
        b += y * learning_rate * expanding_rate

    images.append(snap(x, alpha_i, b, error))
    ''' TESTING CODE
    num_correct = 0
    for j in range(len(X)):
        x_test, y_test = X[j], Y[j]
        y_test_pred = predict(alpha_i, b, x_test)
        if y_test_pred * y_test > 0:
            num_correct += 1
    acc = num_correct / len(X)
    print(f"EPOCH: {epoch+1}\t Acc: {acc}")
    '''


images[0].save(
    'svm_dual.gif',
    optimize=False,
    save_all=True,
    append_images=images[1:],
    loop=0,
    duration=100
)
```

Write a configurable kernel function to apply in lieu of the dot product. Try it out on a dataset that is not linearly separable.

b) Assume we fit an SVM using a polynomial Kernel function and it seems to overfit the data.

How would you adjust the tuning parameter `n` of the kernel function?

Decrease `n` to reduce model complexity and potentially reduce overfitting.

c) Assume we fit an SVM using a RBF Kernel function and it seems to underfit the data. How would you adjust the tuning parameter `sigma` of the kernel function?

Decreasing `sigma` gives each point more influence in its local neighborhood which can increase model complexity to prevent underfitting.

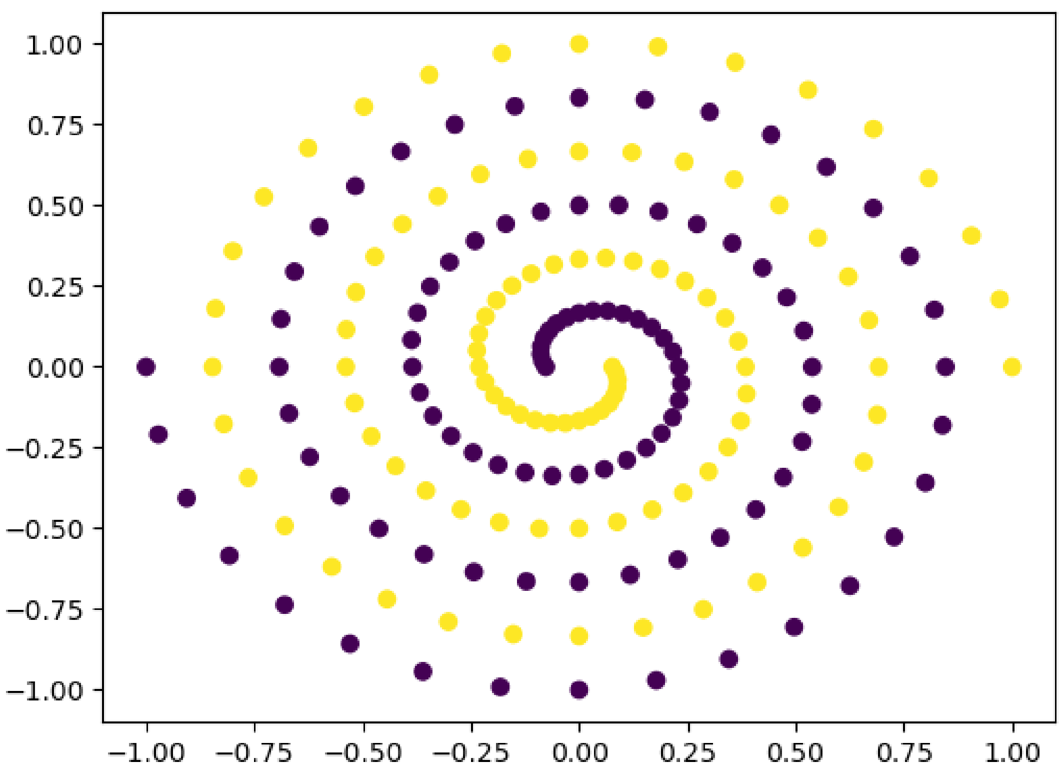
d) Tune the parameter of a specific Kernel function, to fit an SVM (using your code above) to the following dataset:

[8]:
```
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt("spiral.data")
X, Y = data[:, :2], data[:, 2]

plt.scatter(X[:,0], X[:,1], c=Y)
```

[8]: <matplotlib.collections.PathCollection at 0x142c56b77f0>



[24]:
```python
def rbf(x_i, x_j, sigma):
    diff = x_i - x_j
    return np.exp(-diff.T @ diff / (2*sigma*sigma))
```

[33]:
```python
import numpy as np
from PIL import Image as im
import matplotlib.pyplot as plt
import sklearn.datasets as datasets



epochs = 1000
learning_rate = .01
expanding_rate = .995
retracting_rate = 1.01

labels = (Y == 1).astype(int)

alpha_i = np.zeros((len(X),))
b = 0

def snap(x, alpha_i, b, error):
    # create a mesh to plot in
    h = .1  # step size in the mesh
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    meshData = np.c_[xx.ravel(), yy.ravel()]
    cs = np.array([x for x in 'gb'])
    fig, ax = plt.subplots()
    ax.scatter(X[:,0],X[:,1],color=cs[labels].tolist(), s=50, alpha=0.8)

    if error:
        ax.add_patch(plt.Circle((x[0], x[1]), .12, color='r',fill=False))
    else:
        ax.add_patch(plt.Circle((x[0], x[1]), .12, color='y',fill=False))

    Z = predict_many(alpha_i, b, meshData)
    Z = np.array([0 if z <=0 else 1 for z in Z]).reshape(xx.shape)
    ax.contourf(xx, yy, Z, alpha=.5, cmap=plt.cm.Paired)
    fig.savefig(TEMPFILE)
    plt.close()
    return im.fromarray(np.asarray(im.open(TEMPFILE)))

def predict_many(alpha_i, b, Z):
    res = []
```

```python
    for i in range(len(Z)):
        res.append(predict(alpha_i, b, Z[i]))
    return np.array(res)

def predict(alpha_i, b, x):
    kernels = np.array([rbf(X[j], x, 0.05) for j in range(X.shape[0])])
    return alpha_i.T @ kernels + b

images = []
for epoch in range(epochs):
    # pick a point from X at random
    i = np.random.randint(0, len(X))
    error = False
    x, y = X[i], Y[i]

    y_pred = predict(alpha_i, b, x)
    if y_pred * y > 0:
        if -1 < y_pred < 1:
            alpha_i[i] += y * learning_rate
            alpha_i *= retracting_rate
            b += y * learning_rate * retracting_rate
        else:
            alpha_i *= expanding_rate
            b *= expanding_rate
    else:
        error = True
        alpha_i[i] += y * learning_rate
        alpha_i *= expanding_rate
        b += y * learning_rate * expanding_rate

    if (epoch + 1) % 10 == 0:
        images.append(snap(x, alpha_i, b, error))
        num_correct = 0
        for j in range(len(X)):
            x_test, y_test = X[j], Y[j]
            y_test_pred = predict(alpha_i, b, x_test)
            if y_test_pred * y_test > 0:
                num_correct += 1
        acc = num_correct / len(X)
        print(f"EPOCH: {epoch+1}\t Acc: {acc}")


images[0].save(
    'svm_spiral.gif',
    optimize=False,
    save_all=True,
    append_images=images[1:],
```

```
    loop=0,
    duration=100
)
```

```
EPOCH: 10        Acc: 0.6082474226804123
EPOCH: 20        Acc: 0.5
EPOCH: 30        Acc: 0.5
EPOCH: 40        Acc: 0.5
EPOCH: 50        Acc: 0.5
EPOCH: 60        Acc: 0.5
EPOCH: 70        Acc: 0.5
EPOCH: 80        Acc: 0.5
EPOCH: 90        Acc: 0.5
EPOCH: 100       Acc: 0.5
EPOCH: 110       Acc: 0.5
EPOCH: 120       Acc: 0.5
EPOCH: 130       Acc: 0.5
EPOCH: 140       Acc: 0.5
EPOCH: 150       Acc: 0.5
EPOCH: 160       Acc: 0.5
EPOCH: 170       Acc: 0.5
EPOCH: 180       Acc: 0.5
EPOCH: 190       Acc: 0.5
EPOCH: 200       Acc: 0.5
EPOCH: 210       Acc: 0.5
EPOCH: 220       Acc: 0.5
EPOCH: 230       Acc: 0.5
EPOCH: 240       Acc: 0.5
EPOCH: 250       Acc: 0.5
EPOCH: 260       Acc: 0.5
EPOCH: 270       Acc: 0.5
EPOCH: 280       Acc: 0.5
EPOCH: 290       Acc: 0.5
EPOCH: 300       Acc: 0.5
EPOCH: 310       Acc: 0.5
EPOCH: 320       Acc: 0.5
EPOCH: 330       Acc: 0.5
EPOCH: 340       Acc: 0.5
EPOCH: 350       Acc: 0.5
EPOCH: 360       Acc: 0.5
EPOCH: 370       Acc: 0.5
EPOCH: 380       Acc: 0.5
EPOCH: 390       Acc: 0.5
EPOCH: 400       Acc: 0.5
EPOCH: 410       Acc: 0.5
EPOCH: 420       Acc: 0.5154639175257731
EPOCH: 430       Acc: 0.5
EPOCH: 440       Acc: 0.5
```

```
EPOCH: 450        Acc: 0.5
EPOCH: 460        Acc: 0.5
EPOCH: 470        Acc: 0.5
EPOCH: 480        Acc: 0.5103092783505154
EPOCH: 490        Acc: 0.520618556701031
EPOCH: 500        Acc: 0.5309278350515464
EPOCH: 510        Acc: 0.520618556701031
EPOCH: 520        Acc: 0.5103092783505154
EPOCH: 530        Acc: 0.520618556701031
EPOCH: 540        Acc: 0.520618556701031
EPOCH: 550        Acc: 0.5567010309278351
EPOCH: 560        Acc: 0.5515463917525774
EPOCH: 570        Acc: 0.5721649484536082
EPOCH: 580        Acc: 0.5876288659793815
EPOCH: 590        Acc: 0.5773195876288659
EPOCH: 600        Acc: 0.634020618556701
EPOCH: 610        Acc: 0.5773195876288659
EPOCH: 620        Acc: 0.5515463917525774
EPOCH: 630        Acc: 0.5567010309278351
EPOCH: 640        Acc: 0.5567010309278351
EPOCH: 650        Acc: 0.5515463917525774
EPOCH: 660        Acc: 0.5463917525773195
EPOCH: 670        Acc: 0.5670103092783505
EPOCH: 680        Acc: 0.5567010309278351
EPOCH: 690        Acc: 0.5721649484536082
EPOCH: 700        Acc: 0.6082474226804123
EPOCH: 710        Acc: 0.654639175257732
EPOCH: 720        Acc: 0.6701030927835051
EPOCH: 730        Acc: 0.711340206185567
EPOCH: 740        Acc: 0.7268041237113402
EPOCH: 750        Acc: 0.6701030927835051
EPOCH: 760        Acc: 0.6855670103092784
EPOCH: 770        Acc: 0.7268041237113402
EPOCH: 780        Acc: 0.7371134020618557
EPOCH: 790        Acc: 0.7216494845360825
EPOCH: 800        Acc: 0.7474226804123711
EPOCH: 810        Acc: 0.7216494845360825
EPOCH: 820        Acc: 0.7216494845360825
EPOCH: 830        Acc: 0.7989690721649485
EPOCH: 840        Acc: 0.8762886597938144
EPOCH: 850        Acc: 0.9072164948453608
EPOCH: 860        Acc: 0.8762886597938144
EPOCH: 870        Acc: 0.8762886597938144
EPOCH: 880        Acc: 0.8762886597938144
EPOCH: 890        Acc: 0.9072164948453608
EPOCH: 900        Acc: 0.9329896907216495
EPOCH: 910        Acc: 0.9484536082474226
EPOCH: 920        Acc: 0.9845360824742269
```

```
EPOCH: 930       Acc: 0.9896907216494846
EPOCH: 940       Acc: 0.9948453608247423
EPOCH: 950       Acc: 1.0
EPOCH: 960       Acc: 0.9948453608247423
EPOCH: 970       Acc: 0.9948453608247423
EPOCH: 980       Acc: 0.9948453608247423
EPOCH: 990       Acc: 0.9948453608247423
EPOCH: 1000      Acc: 0.9948453608247423
```

[ ]:

```

```