CS 491/691 - Project 5 - SVM

Lee Easson[1]

leasson@nevada.unr.edu

1 Implementation - Binary Classification

1.1 Calculate Distance to Hyperplane

The helper function *distance_point_to_hyperplane(pt, w, b)* calculates the distance between a point *pt* and the decision boundary defined by w and b.

```
def distance_point_to_hyperplane(pt, w, b):
    p1=np.array([0.0, (w*0.0)+b])
    p2=np.array([15.0,(w*15.0) + b])
    p3=np.array([pt[0],pt[1]])

    d=np.cross(p2-p1,p3-p1)/np.linalg.norm(p2-p1)
    return d
```

The function gets two points from the decision boundary, *p1* and *p2*, and computes the distance between *p3* and the line defined by *p1* and *p2* by using the cross product and normalization features from the *numpy* library.

1.2 Compute Margin

Possibly the most integral helper function for SVM binary classification, *compute_margin(data, w, b)* reads in a set of data (*data*) and a separator defined by w and b and returns the margin.

```
def compute_margin(data, w, b):
    # Separate into (+) and (-) samples
    distances_pos = {}
    distances_neg = {}
    for d in data:
        if d[2] == 1.0:
            distances_pos.update({(d[0], d[1]) : 0})
        else:
            distances_neg.update({(d[0], d[1]) : 0})

    # Get distances for (+) and (-) samples
    for pos in distances_pos:
        dist = distance_point_to_hyperplane(pos, w, b)
        distances_pos[pos] = abs(dist)

    for neg in distances_neg:
```

```
            dist = distance_point_to_hyperplane(neg, w, b)
            distances_neg[neg] = abs(dist)

        # Get min distance from each list
        distances_pos = dict(sorted(distances_pos.items(),
            key=lambda x: x[1]))
        distances_neg = dict(sorted(distances_neg.items(),
            key=lambda x: x[1]))
        minimum_pos = min(distances_pos, key=distances_pos.
            get)
        minimum_neg = min(distances_neg, key=distances_neg.
            get)

        # Get perpendicular line
        w = np.cross(minimum_pos,minimum_neg)
        return w
```

The function first separates the *data* into positive- and negative-labeled samples into their respective dictionaries – *distances_pos* and *distances_neg*. For all samples in both dictionaries, the distance between the current point and the hyperplane are updated for each sample. The dictionaries are then sorted and the minimum distance is extracted from each. Finally, the line perpendicular to the line defined by the minimum distance positive point and the minimum distance negative point is calculated using cross product and is returned as the decision boundary.

1.3 SVM Binary Training

svm_train_brute(training_data) is the primary function of SVM binary classification that reads in a set of training data and returns a decision boundary defined by w and b and a *numpy* array of support vectors S.

```
    def svm_train_brute(training_data):
        w = 0.0
        b = 0.0
        S = []

        margin = compute_margin(training_data, w, b)
        w = margin
        S.append(margin)

        S = np.array(S)
        return w,b,S
```

Both w and b are initialized to zero and S is initialized as an empty list. The margin is calculated using the *compute_margin(data, w, b)* function from the previous section and appended to S. Finally, S is converted to a *numpy* array and is returned along with w and b.

1.4 SVM Binary Testing

svm_binary_brute(w, b, x) is a simple accuracy testing function for the SVM binary classification. It reads in the decision boundary defined by w and b and a test point x.

```
def svm_test_brute(w, b, x):
    pred = 0
    result = np.dot(x, w)

    if result[0] >= 0:
        pred = 1
    else:
        pred = -1
    return pred
```

This function simply calculates the dot product of the weight that defines the decision boundary and the test point. IF the results is positive or zero, it is classified as 1, otherwise it is classified as -1.

2 Implementation - Multiclass Classification

2.1 Computer Margin for Label

The helper function *compute_margin(data, w, b, label)* is a modification of the original *compute_margin(data, w, b)* function with the *label* parameter allowing for classification of more than two different labels.

```
def compute_margin(data, w, b, label):
    # Separate into label and not label samples
    distances_pos = {}
    distances_neg = {}

    for d in data:
        if d[2] == label:
            distances_pos.update({(d[0], d[1]) : 0})
        else:
            distances_neg.update({(d[0], d[1]) : 0})

    # Get distances for label and not label samples
    for pos in distances_pos:
        dist = distance_point_to_hyperplane(pos, w, b)
        distances_pos[pos] = abs(dist)

    for neg in distances_neg:
        dist = distance_point_to_hyperplane(neg, w, b)
        distances_neg[neg] = abs(dist)

    # Get min distance from each list
```

```
    distances_pos = dict(sorted(distances_pos.items(),
        key=lambda x: x[1]))
    distances_neg = dict(sorted(distances_neg.items(),
        key=lambda x: x[1]))
    minimum_pos = min(distances_pos, key=distances_pos.
        get)
    minimum_neg = min(distances_neg, key=distances_neg.
        get)

    # Get perpendicular line
    w = np.cross(minimum_pos,minimum_neg)
    return w,b
```

2.2 SVM Multiclass Training

svm_train_multiclass(training_data) is very similar to the *svm_train_brute(training_data)* function with a few differences. The function first iterates through all the labels in the training data and isolates the unique class labels (in the case of this project, the class labels are 1,2,3,4 as opposed to 1 and -1).

```
def svm_train_multiclass(training_data):
    W = []
    B = []
    labels = []
    for i in range(len(training_data)): labels.append(
        training_data[i][2])
    Y = len(set(labels))
    labels = list(set(labels))

    # Compute margin
    for i in range(Y):
        w = 0.0
        b = 0.0
        w,b = compute_margin(training_data,w,b,labels[i])
        W.append(w)
        B.append(b)

    W = np.array(W)
    B = np.array(B)
    return [W, B]
```

After extracting the unique class labels, the function iterates for every label treating every *compute_margin(training_data, w, b, label)* calculation as a classification of 'X' and 'not X'. Each decision boundary is added to their respective lists and returned as an array of *[W, B]*.

2.3 SVM Multiclass Testing

The final method in the SVM multiclass testing is *svm_test_multiclass(W, B, x)*, which reads in the list of decision boundaries defined by W and B and a test point x and returns the corresponding class (returns -1 if class cannot be determined).

```
def svm_test_multiclass(W, B, x):
    c = -1
    return c
```

3 Plots of Training Data and Decision Boundaries

3.1 Binary Classification

Four different plots, shown in figures 1, 2, 3, 4, were used to test binary classification. Each plot contains binary-labeled data; either positive (indicated with crosses) or negative (labeled with circles).

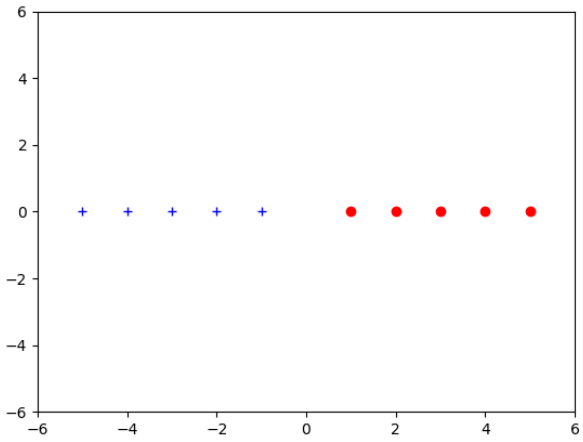

Fig. 1.

The four binary plots are shown in figures 5, 6, 7, 8 after applying binary classification SVM.

3.2 Multiclass Classification

Two different plots, shown in figures 9, 10, were used to test multiclass classification. Each plot contains data labeled with '+', 'o', '*', or '.'.

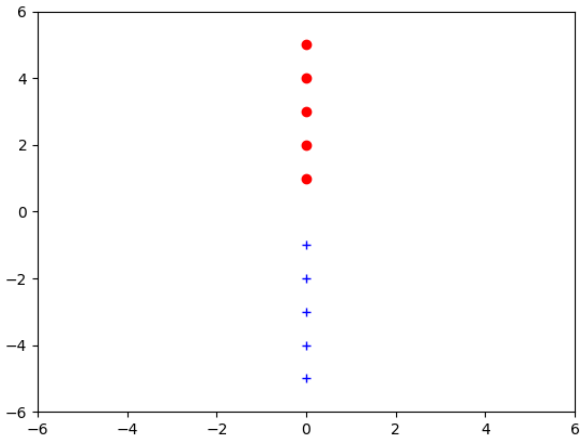

Fig. 2.

The two multiclass plots are shown in figures 11 and 12 after applying multiclass SVM classification.

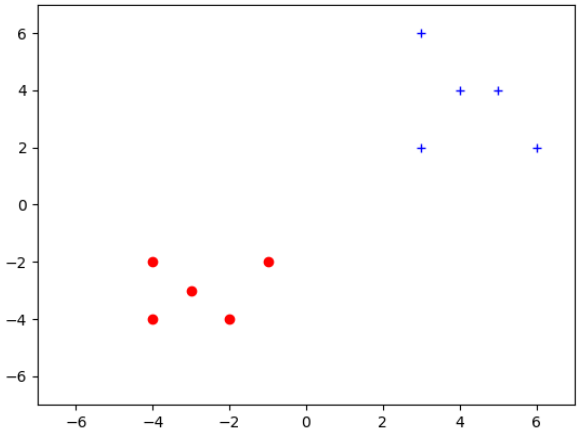
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

Fig. 3.

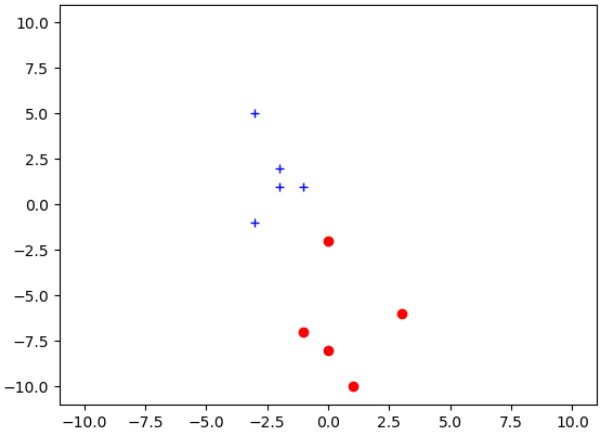
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Fig. 4.

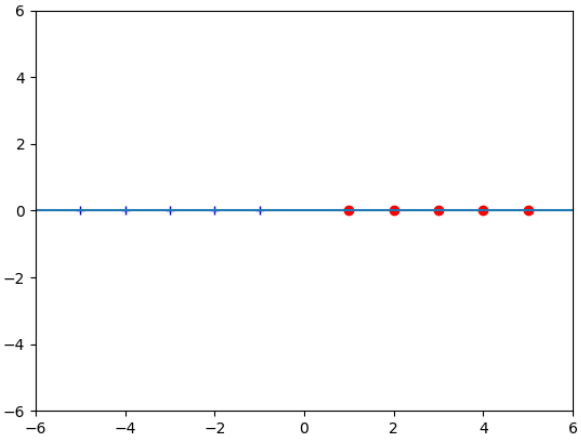

Fig. 5. w=0.0, b=0.0

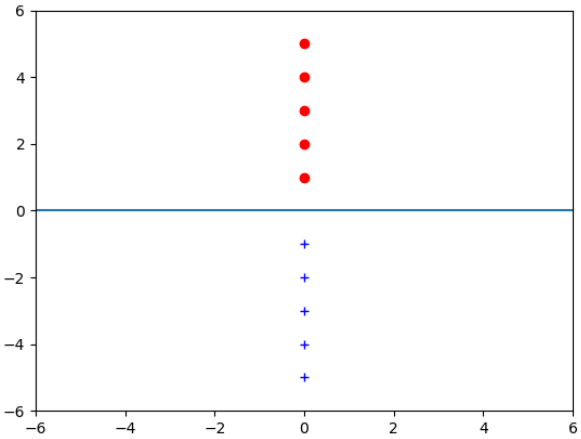

Fig. 6. w=0.0, b=0.0

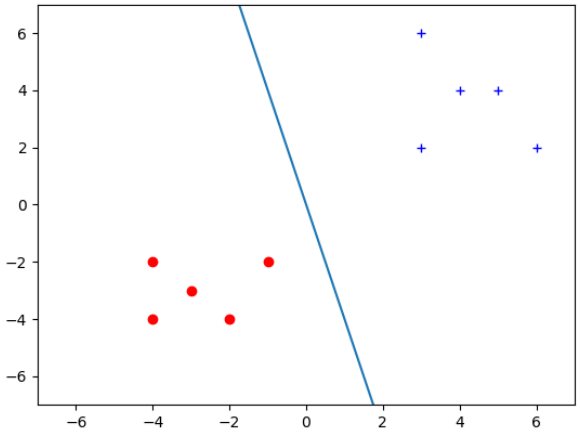

Fig. 7. w=-4.0, b=0.0

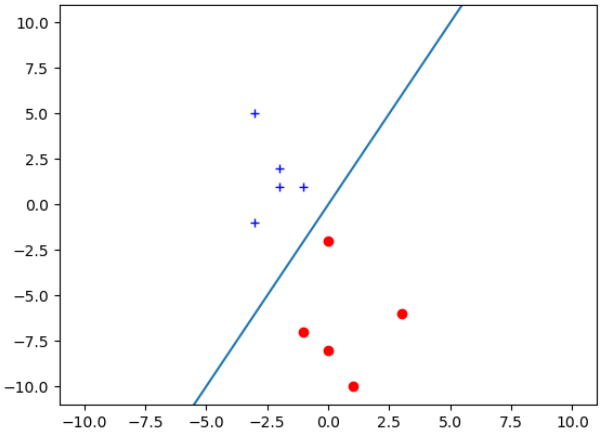

Fig. 8. w=2.0, b=0.0

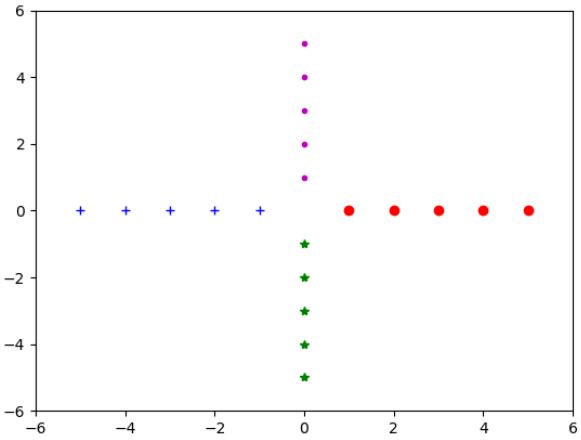

Fig. 9. Caption

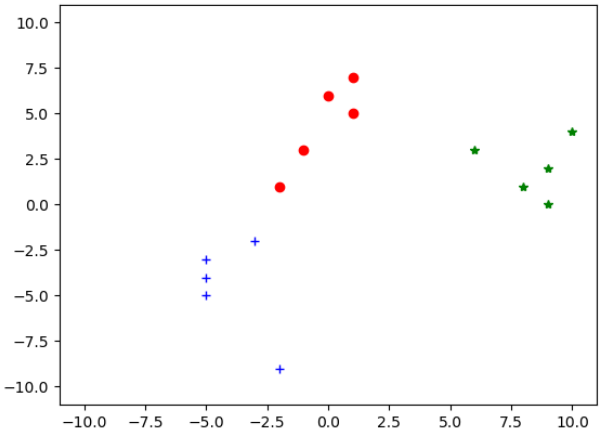

Fig. 10. Caption

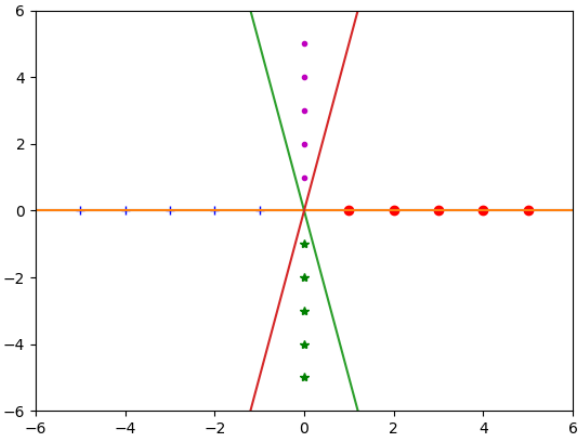

Fig. 11. Caption

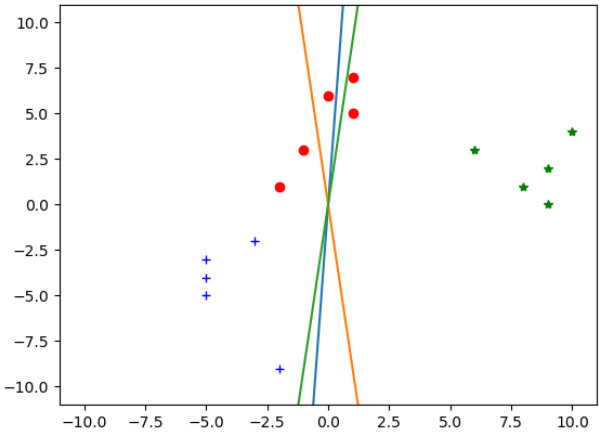

Fig. 12. Caption