A Guide to Support Vector Machines

Vivian Lee and Sheryl Deakin

Table of Contents

Intro: Data is Everywhere

Every transaction, interaction, and scenario generates new information, or *data*. The creation of data is a continuous process. In light of this fact and the advances in computing technology over the past few decades, machine learning has emerged as a popular field of study. With machine learning, we can analyze and make use of historical data collected from our pasts to provide new, useful information that can influence our futures.

Machine learning models can be divided into two broad categories: supervised and unsupervised learning. With supervised learning models, we start with a collection of n observations that each include a set of i independent variables ($x_i \in X$) and the actual value of the independent variable Y we are interested in predicting. This collection of observations with known Y values is used to

train and validate a learning model that can be used to predict Y for new observations where it is not known. In unsupervised learning, we seek to achieve a good representation of the domain space covered by our training samples and learn about the structure of our data, rather than predict some variable Y or label for a new observation.

This guide provides an introduction to supervised machine learning models with a focus on a particular SML model, Support Vector Machines (SVMs).

Important Background Knowledge

Types of Learning Models

1. Regression

A regression analysis model attempts to determine the relationship between one or more independent variables $x_1, x_2, \ldots x_n \in X$ and a dependent continuous variable Y. Once trained and validated, a regression model can be used to predict the value of Y for a new observation where Y is not known.

2. Classification

A classification model (classifier) is used to predict the class, or label, of a new observation.

For example, given $Y \in \{Fruit, Vegetable\}$, our classification model would predict whether a new given produce item is a fruit or a vegetable.

Classification can be binary (0 or 1) or multiclass (1, 2, ... K).

Supervised Learning Models

To fully understand SVMs, we need a basic understanding of some simpler supervised learning models: linear regression and logistic regression.

1. Linear Regression

Taking a step back to high school algebra and geometry, we revisit the well-known formula for a line:

$$y = mx + b$$

The coefficients m and b, the slope and y-intercept respectively, are our weights. We can rewrite this formula as follows:

$$\begin{aligned} w_0 = b, w_1 = m \\ x_0 = 1, x_1 = x_1 \\ y = w_1(x_1) + w_0(1) \\ = w_1(x_1) + w_0 \\ = w_0 + w_1(x_1) \end{aligned}$$

If we had two independent variables, our line (2-D) would become a plane (3-D) and we would write our equation as:

$$y = w_0 + w_1(x_1) + w_2(x_2)$$

A line in n-dimensional space would have $n - 1$ independent variables. Our formula would then be:

$$y = w_0 + w_1(x_1) + \ldots + w_{n-1}(x_{n-1})$$

These weight coefficients $w_0, w_1 \ldots w_n$ can be stored in a vector of size $n + 1$ known as θ where $\theta_0 = w_0$, the y-intercept or *bias* term. Given a set of observations, a linear regression model will determine the best values of $\theta_0, \theta_1 \ldots \theta_n$ that produce the line of best fit. Then, given a new observation X with independent variables $a_1, a_2 \ldots a_n$, the predicted Y value will be the y value on the line of best fit where $x_1 = a_1, x_2 = a_2 \ldots = x_n = a_n$.

The predicted value Y is also represented by $h_\theta(x)$, known as the model's hypothesis. The two terms will be used interchangeably going forward.

$y = w_0(x_1) + w_1(x_1) + \ldots + w_n(x_n)$ is equivalent to $h_\theta(x) = \theta^T x$

Optimization Objective: Cost Function and Finding θ

Our goal in building a predictive model is to ensure that our model can make predictions that are as accurate as possible. This is also known as minimizing error. We use $J(\theta)$ to represent the average error (cost) of each prediction made using the given set of weights θ.

In linear regression, we use the following function, the mean squared error, to compute the error for our model, given that we tested on n observations.

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Therefore, our goal is to find the value of θ which will minimize $J(\theta)$ by bringing it as close to 0 as possible. This can be done by a process known as **gradient descent**.

Notice that $J(\theta)$ here is a convex function. By taking the first derivative of $J(\theta)$ and setting it equal to 0 we can determine the global minimum. The θ value at the global minimum represents the set of weights that minimize $J(\theta)$. Taking the second derivative of $J(\theta)$ will help us determine which direction (positive or negative) to make steps in to reach the global minimum.

First derivative:

$\frac{d}{d\theta_0} J(\theta) = \sum_{i=1}^{n} \frac{2}{n} (h_\theta(x^{(i)}) - y^{(i)})$

$\frac{d}{d\theta_j} J(\theta) = \sum_{i=1}^{n} \frac{2}{n} (h_\theta(x^{(i)}) - y^{(i)})(x_j^{(i)})$

Second derivative:

$\frac{d^2}{d\theta_0^2} J(\theta) = 0$

$\frac{d^2}{d\theta_j^2} J(\theta) = \sum_{i=1}^{n} \frac{2}{n}(x_j^{(i)})^2$

Example: Linear Regression

Since the focus of this guide is on SVMS, we will use the linear_model package available in the sklearn library to construct our linear and logistic regression models. We will demonstrate how to create an SVM model from scratch in the section on SVMs later on.

Given the following data set of 5 observations (assumes $x_0 = 1$).

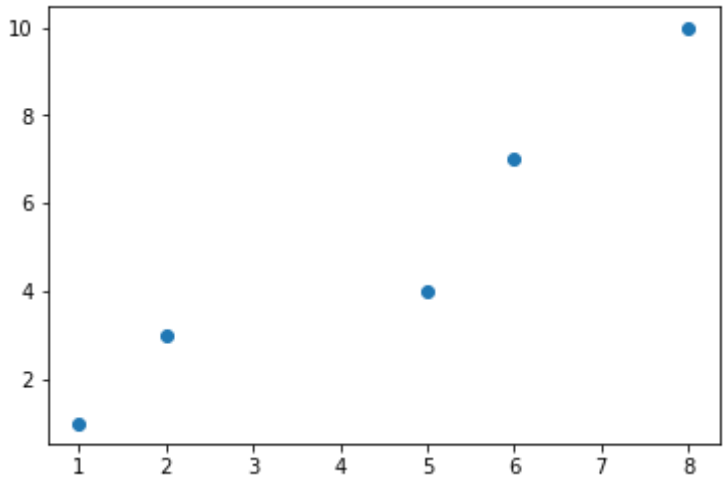
x_1	y
1	1
2	3
5	4
6	7
8	10

In [1]:
```
# Import necessary libraries
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import LinearRegression, LogisticRegression
import math
from IPython.display import Image
from utils import plot_data, plot_decision_function
from sklearn.datasets.samples_generator import make_blobs
```

Try changing the values of x and y in the code below afterwards to see how they alter the line of best fit determined by the linear regression model.

In [2]:
```
# create the dataset
x = np.array([1,2,5,6,8]).reshape(-1,1)
y = np.array([1,3,4,7,10]).reshape(-1,1)
```

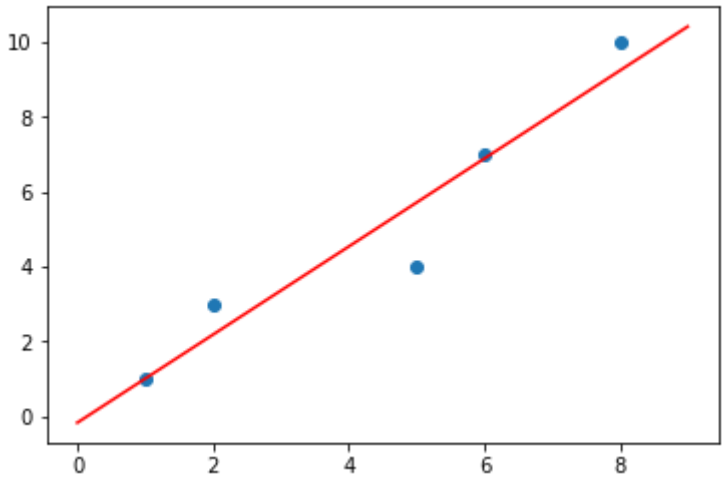
In [3]:
```
# visualization of the data points
plt.figure()
plt.scatter(x,y)
plt.show()
```



In [4]:
```
# construct a linear regression model
reg = LinearRegression().fit(x,y)
w0 = reg.intercept_[0]
w1 = reg.coef_[0][0]

print("y = w0 + w1x")
print("y = " + str(w0) + " + " + str(w1) + "x")
```

```
y = w0 + w1x
y = -0.16867469879518016 + 1.1746987951807226x
```

In [5]:
```
# plot the line of best fit found with the determined weights
start = 0
end = np.amax(x) + 2 # add 2 or any number > 1 so line will extend on plot

best_fit_x = np.array([range(start,end)]).reshape(-1,1)
best_fit_y = w0 + w1 * best_fit_x

plt.scatter(x,y)
plt.plot(best_fit_x, best_fit_y, color = 'red')
plt.show()
```



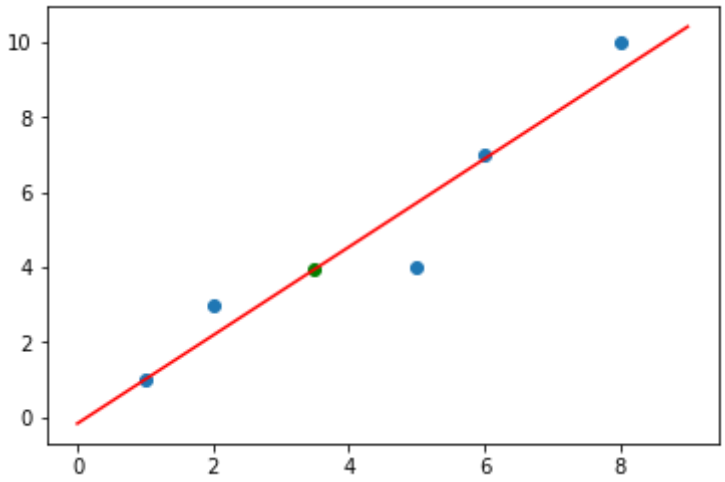
Use the constructed linear regression model to predict y for a new observation given a specified value of x

In [6]:
```
new_x = 3.5
new_y = reg.predict(np.array([new_x]).reshape(-1,1))[0][0]

print("Given x = " + str(new_x) + " the model predicts y = " + str(new_y))
```

```
Given x = 3.5 the model predicts y = 3.942771084337349
```

The new observation is represented by the green point on the plot, which will always be on the line of best fit.

In [7]:
```
plt.scatter(x,y)
plt.plot(best_fit_x, best_fit_y, color = 'red')
plt.scatter([new_x], [new_y], color = 'green')
plt.show()
```



2. Logistic Regression

Despite its misleading name, logistic regression is actually a classification model. To keep things simple, we will limit our focus to binary classification in this review. In binary classification, $h_\theta(x)$ can only be one of two predefined values. For example: true or false, blue or white, Yanny or Laurel.

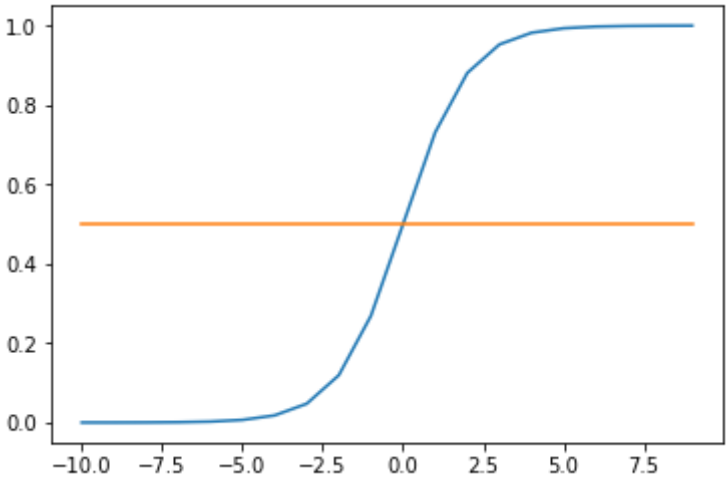
The values can be represented as 0 and 1.

Similarly to linear regression, logistic regression will determine the optimal weights vector θ, the coefficients of the independent variables. The weights vector is used to draw a decision boundary that separates the given data observations.

In 2-D space, this decision boundary is a line. In 3-D space, it is a hyperplane.

Recall the sigmoid or "logistic" function: $g(a) = 1/(1 + e^{-a})$

In [8]:
```
def sigmoid(x):
    return 1.0 / (1.0 + math.exp(-x))

x = range(-10,10)
y = [sigmoid(a) for a in x]

plt.plot(x,y)
plt.plot(x,[0.5]*len(x))
plt.show()
```



In logistic regression for a binary classifier, we are predicting the probability that $h_\theta(x) = 1$

$h_\theta(x) = 1 = P(y = 1|x; \theta)$

Since $h_\theta(x) \in \{0, 1\}$, y can only be 0 or 1, $P(y = 1|x; \theta) + P(y = 0|x; \theta) = 1|$

Our hypothesis function is $h_\theta(x) = g(\theta^T x) = 1/(1 + e^{-\theta^T x})$

Therefore, as can be seen in the plot above, $0 \leq h_\theta(x) \leq 1$

If $h_\theta(x) \geq 0.5$ we predict $y = 1$

If $h_\theta(x) < 0.5$ we predict $y = 0$

Notice in the sigmoid function plot $g(a)$ that if $a \geq 0$ then $g(a) \geq 0.5$

From this we can conclude:

$h_\theta(x) = g(\theta^T x) \geq 0.5$ when $\theta^T x \geq 0$ => therefore if $\theta^T x \geq 0$ we predict $y = 1$

$h_\theta(x) = g(\theta^T x) < 0.5$ when $\theta^T x < 0$ => therefore if $\theta^T x < 0$ we predict $y = 0$

Logistic Regression: Cost Function

Recall $J(\theta)$ in linear regression, the mean squared error of the predicted test observations:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)})^2$$

In logistic regression the cost for our i^{th} sample is also represented by the squared difference between its predicted y value and its actual y value

$$cost(h_\theta(x^{(i)}), y^{(i)}) = (h_\theta(x^{(i)}) - y^{(i)})^2$$

We use this same form as the base of our cost function $J(\theta)$ in logistic regression.

In binary logistic regression, the method to calculate cost for a sample varies based on which of the two class options (0 and 1) was predicted for that sample.

$$cost(h_\theta(x^{(i)}), y^{(i)}) = \begin{cases} -\log(h_\theta(x^{(i)})), & \text{if y=1} \\ -\log(1 - h_\theta(x^{(i)})), & \text{if y=0} \end{cases}$$

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y\log(h_\theta(x) + (1-y)\log(1 - h\theta(x))]$$

Notice that if $y = 1$ the term when on the right side of $+$ for when $y = 0$ will be 0

Similarly, if $y = 0$ the term on the left side of $+$ for when $y = 1$ will be 0

This neat observation allows us to combine the two cases to a single equation for $J(\theta)$. This equation is known as the log loss.

As before in linear regression, we can take the first derivative of $J(\theta)$ and set it equal to 0 to minimize error and determine the optimal weights θ that will maximize the classification accuracy of our logistic regression model.

Example: Logistic Regression

Here we provide an example of building a logistic regression model using sklearn's LogisticRegression class.

To ensure we have enough samples in our training and test datasets, we make use of a function *make_blobs* and set a seed to control the randomness of the generated dataset.

In [9]:
```
# create the training and test datasets
X, y = make_blobs(n_samples=100, centers = 2, n_features = 2, random_state=
train_X_1 = X[0:80,]
train_y_1 = y[0:80]
test_X_1 = X[80:,]
test_y_1 = y[80:]
```

In [10]:
```
# build the logistic regression model and train with the training set
logreg = LogisticRegression(solver="liblinear")
logreg.fit(train_X_1, train_y_1)
```

Out[10]:
```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=Tr
ue,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='liblinea
r',
          tol=0.0001, verbose=0, warm_start=False)
```

In [11]:
```
# test the logistic regression model
pred_log_reg = logreg.predict(test_X_1)

# add column labels to print for visualization purposes
col_labels = "   x1             x2          actual y     predicted y"
print(col_labels)

# print the test set with the actual and expected y values
test_set_with_pred = np.column_stack((test_X_1, test_y_1, pred_log_reg))
print(test_set_with_pred)
```

```
   x1             x2          actual y     predicted y
[[ -0.10657046 -11.82507855   0.           0.         ]
 [ -1.3196722   -9.15547193   0.           0.         ]
 [  3.54368779  -2.7667254    1.           1.         ]
 [  4.27299496  -0.32269083   1.           1.         ]
 [ -0.58498235 -11.51494191   0.           0.         ]
 [ -0.15775339  -1.72007462   1.           1.         ]
 [  1.1180732   -0.53705      1.           1.         ]
 [ -0.66876118  -9.43350477   0.           0.         ]
 [  0.39949397  -0.14405158   1.           1.         ]
 [ -1.29923245  -8.30647414   0.           0.         ]
 [  5.10194218  -0.47243127   1.           1.         ]
 [ -3.32042501  -9.43521984   0.           0.         ]
 [ -1.54915892  -7.25010857   0.           0.         ]
 [ -0.42556099  -1.54532627   1.           1.         ]
 [ -2.41323523  -9.04838281   0.           0.         ]
 [ -1.61892392  -9.71765939   0.           0.         ]
 [  0.5249033   -2.54827245   1.           1.         ]
 [ -1.68399423 -10.86599403   0.           0.         ]
 [ -1.2800922   -8.93912279   0.           0.         ]
 [  1.59476537  -1.28989266   1.           1.         ]]
```

In [12]:
```
# get accuracy rate on test set predictions
num_correct_log_reg = sum(pred_log_reg == test_y_1)
num_classified = len(test_y_1) * 1.0
print("accuracy rate: " + str(num_correct_log_reg / num_classified))
```

```
accuracy rate: 1.0
```

Support Vector Machines

At last! Now we have the necessary base knowledge to start learning about support vector machines.

Support vector machines (SVMs) can be used to construct both regression and classification models. For this guide, we will focus on support vector machines for classification.

Like logistic regression, SVMs can be used to classify a given data observation. However, they are a non-probabilistic model. Rather than computing the probability that the observation belongs to a specific class Y based on learning from a training set, SVMs plot each observation with n-features

in the training set in n-D space and divide the feature space into different regions. New samples from the test set are classified by which region they are in when plotted on the graph.

Maximizing the Margin

Similar to logistic regression, the feature space is divided by a decision boundary (line in 2-D, hyperplane in 3-D).

However, the SVM's goal is to determine the θ value which produces a decision boundary that:

1. separates observations belonging to different classes
2. maximizes the margin (the distance between the decision boundary and nearest data point of a class on either side)

The distance from a data point to the decision boundary, the *margin*, represents the confidence of our prediction. The farther away a data point is from the decision boundary, the more confident we are that it belongs to one class over the other.

New Notation for SVMs

For our discussion on SVMs we will introduce some new notation.

Our weights vector previously represented by θ will now be represented by w and b where $w = \theta_1 \dots \theta_m$ and $b = \theta_0$

Our predicted value transitions from $y \in \{0, 1\}$ to $y \in \{-1, 1\}$

Our decision boundary is represented by the equation of the hyperplane $w \cdot x + b = 0$

The distance between a data point x_i from the decision boundary is represented by

$$\gamma = (w \cdot x_i + b)y_i$$

This distance is also known as the margin between x_i and the decision boundary.

Our prediction for a data point x_i is dependent on the sign of $w \cdot x_i + b$

Maximizing the margin between each data point in our feature space to the decision boundary is equivalent to increasing the confidence of our prediction for each point.

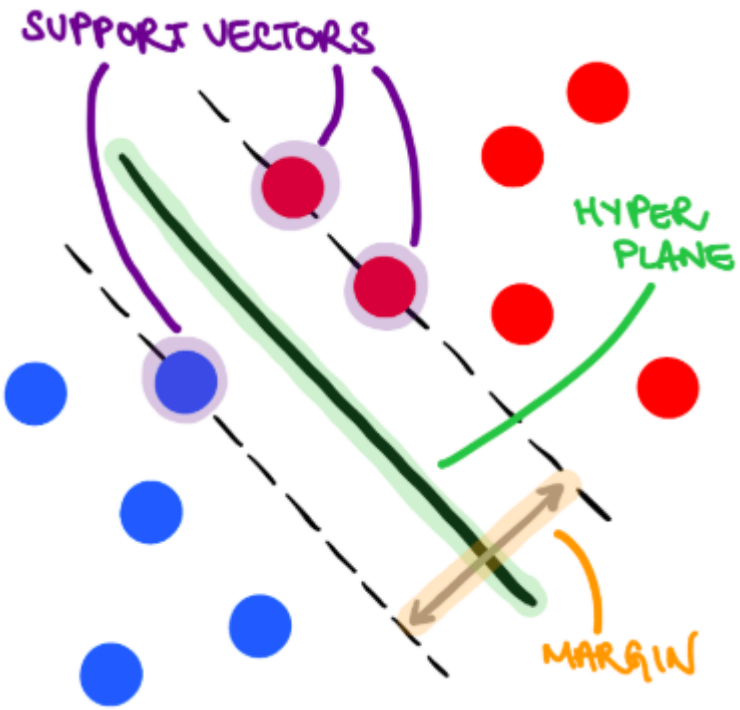
Therefore, our optimization goal is defined by

$$max_{w,\gamma} s.t. \forall i, y_i(w \cdot x_i + b) \geq \gamma$$

"finding w to maximizing the margin γ such the distance between each of our data points in our feature space is at least γ (gamma)"

Defining the Support Vectors

The support vectors are the data points closest to the decision boundary and their distance to the decision boundary is the margin we seek to maximize.



Solving for the Margin γ

Using the image above, let us call the blue support vector x_1 and one of the red support vectors x_2

We define the dashed lines that run through the support vectors and are parallel to the decision boundary by

$w \cdot x_1 + b = -1$ and $w \cdot x_2 + b = +1$

As a reminder, the distance between a support vector and the decision boundary is denoted by γ

We can define x_1 in terms of x_2 by:

$$x_1 = x_2 + 2\gamma \frac{w}{||w||}$$

Preventing Scaling and Normalizing w

We normalize w in the equation above to prevent scaling from influencing the w value determined by the optimization function.

Given $\gamma = (w \cdot x + b)y_i$, notice that if we scale w and b by a factor of 2, $(2w \cdot x + 2b)$, we can increase our margin γ. However, even though this will increase our margin, our prediction is based on the *sign* of γ and not the magnitude.

Therefore, though scaling would increase our margin, it would have no affect on our prediction y_i for x_i. We must normalize w to prevent scaling; though it would help maximize the margin, it would have no affect on our model's prediction accuracy.

$$\gamma = (\frac{w}{||w||} \cdot x + b)y$$

Using this we can solve for γ as follows:

$$w \cdot x_1 + b = 1$$

$$w \cdot (x_2 + 2\gamma\frac{w}{||w||}) + b = 1$$

$$w \cdot x_2 + b + 2\gamma\frac{w \cdot w}{||w||} = 1$$

$$(w \cdot x_2 + b) + 2\gamma\frac{w \cdot w}{||w||} = 1$$

$$-1 + 2\gamma\frac{w \cdot w}{||w||} = 1$$

$$\gamma\frac{w \cdot w}{||w||} = 1$$

$$\gamma = \frac{||w||}{w \cdot w}$$

$$\gamma = \frac{||w||}{||w||^2}$$

$$\gamma = \frac{1}{||w||}$$

Optimization Problem for Hard Margin SVMs

Maximizing the margin is equivalent to finding a set of weights w which minimizes the value $\frac{1}{2}||w||^2$

$max\ \gamma \approx max_w\ \frac{1}{||w||} \approx min_w\ ||w|| \approx min_w \frac{1}{2}||w||^2$

If we add the constraint that the distance between each of our data points must be at least 1, we have an SVM with hard constraints or hard margins

$min_w \frac{1}{2}||w||^2 s.t. \forall i, y_i(w \cdot x_i + b) \geq 1$

Hard margin SVMs assume that the given data points are perfectly linearly separable

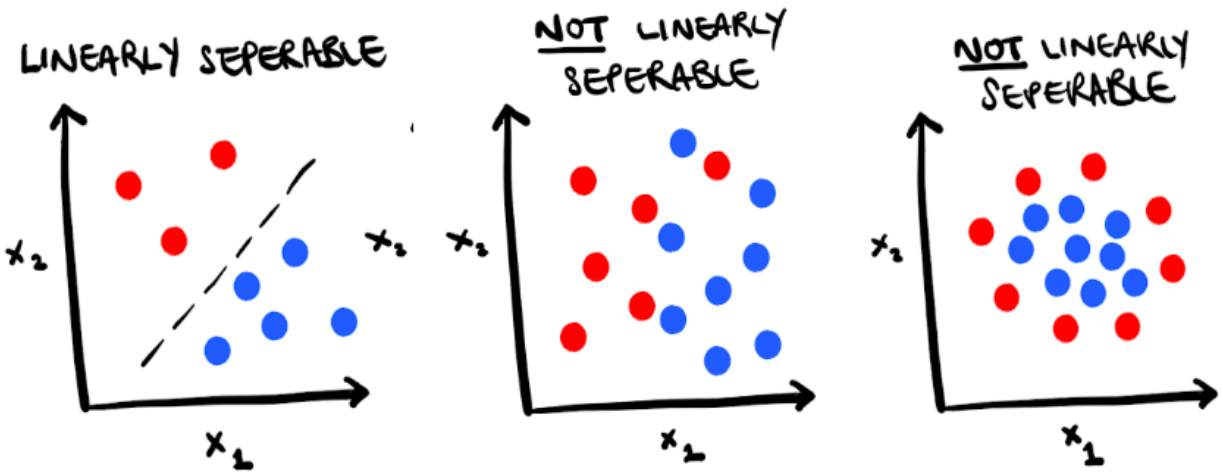
Linearly and Non-Linearly Separable Data

SVMs can be used to classify linearly and non-linearly separable data.

Linearly separable data can be separated by a line, whereas a line would be insufficient for separating non-linearly separable data and result in some misclassifications.

For the scope of this guide, we will focus on the first two cases in the image below:

1. data that is perfectly linearly separable
2. data that can be separated with a line but would involve some classification error



Optimization Problem for Soft Margin SVMs with Slack Variables

If we have data that is not linearly separable, as seen in the middle image, we can still create a fairly accurate SVM classifier with a linear decision boundary by updating our optimization problem to include a penalty C for incorrect classifications. This is known as SVMs with soft margins.

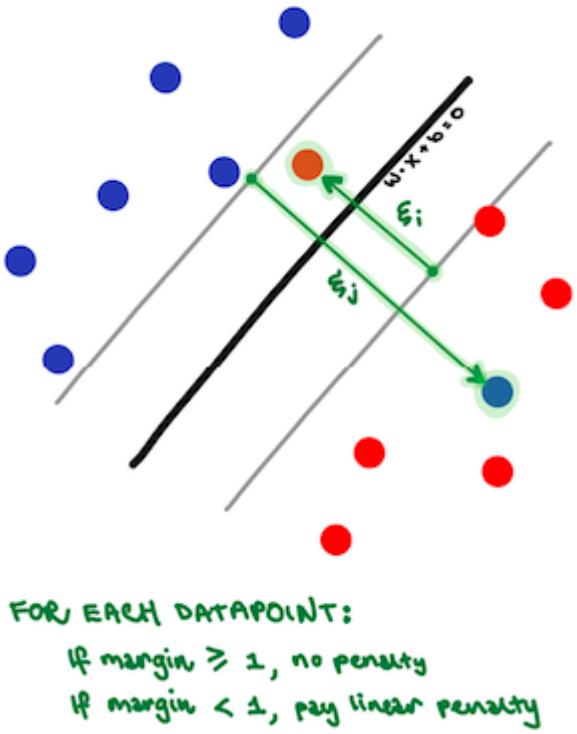
We define the penalty by introducing slack variables. The slack variable for a data point x_i is denoted by $\mathcal{E}_i$.

slack of x_i = $\mathcal{E}_i$ = the distance from x_i to the correct support vector for its class

penalty for misclassification of $x_i = \mathcal{E}_i$

Our new optimization problem is defined by

$$min_w \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \mathcal{E}_i$$
$$s.t.\ \forall i, y_i(w \cdot x_i + b) \geq 1 - \mathcal{E}_i$$



Regularization Parameter C

Notice now that there will be a tradeoff between finding w which minimizes $\frac{1}{2}\|w\|^2$ vs $C\sum_{i=1}^{n}\mathcal{E}_i$

$\frac{1}{2}\|w\|^2$ controls the magnitude of our margin (minimizing this value maximizes our margin) while $C\sum_{i=1}^{n}\mathcal{E}_i$ controls how well we fit our model to the data.

C a hyperparameter which we explicitly assign ourselves to balance between this tradeoff.

If $C = \infty$, we assign an enormous penalty for each misclassified data point. Our decision boundary may be very good at separating the data, but the margin will be very small. A smaller margin reduces our confidence about the classification of a new data point.

If $C = 0$, we assign no penalty for misclassified data, and the optimization function will set $w = 0$ to minimize $min_w \frac{1}{2}\|w\|^2$. By setting $w = 0$ our SVM will determine a decision boundary that completely ignore the data points (our training data).

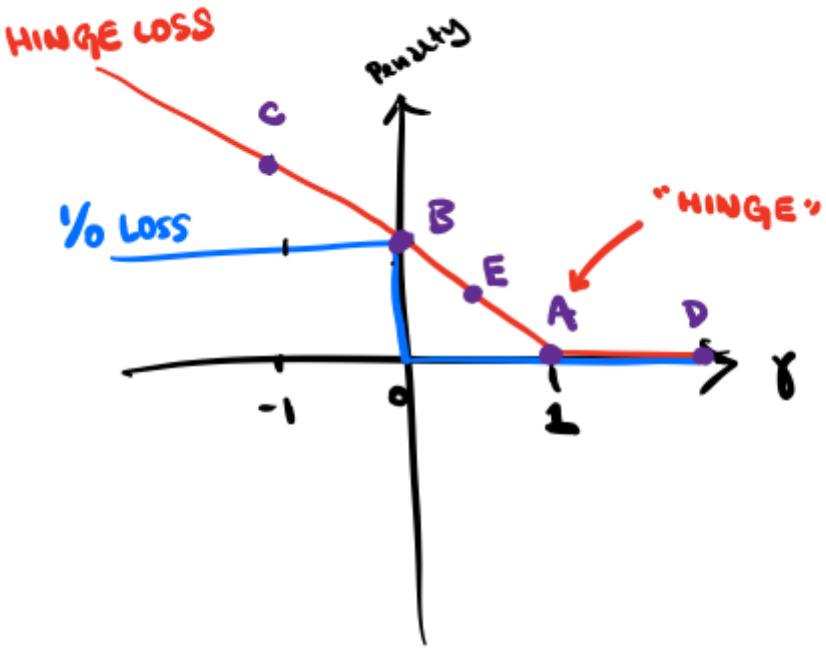
Ideally we want to be somewhere in the middle where we find a sweet spot value for C that will separate the data points fairly accurately and maximize the margin, allowing for some small amount of misclassification.

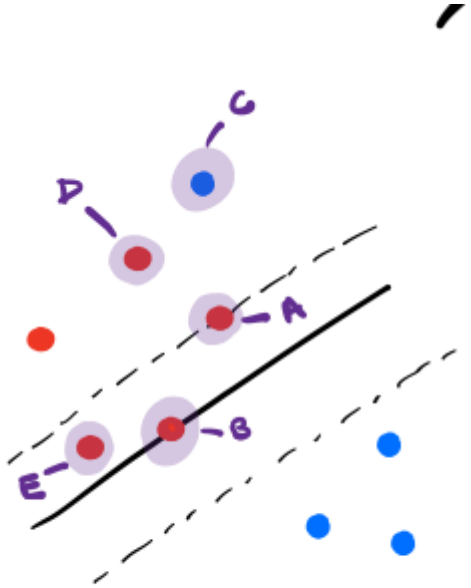
Balancing between these two tradeoffs to find the best value for C which achieves this objective is known as **regularization**.

Hinge Loss Function

Recall $C\sum_{i=1}^{n}\mathcal{E}_i$ defines how well we fit our model to the training data. It is also known as the empirical loss of our model.

In SVMs, we use the *hinge loss* function to represent this loss.





The hinge loss for the predicted y_i for data point x_i is $\ell(y_i)$

$$\ell(y_i) = max(0, 1 - \gamma_i) = max(0, 1 - y_i(w \cdot x_i + b))$$

Subgradient of Hinge Loss

The hinge loss is a non-differentiable function, meaning it is not differentiable everywhere. However, we can still perform gradient descent to learn the optimal weights by taking taking the subgradient of the hinge loss.

Taking the subgradient means we account for the two cases from $max(0, 1 - y_i(w \cdot x_i + b))$ where:

1. 0 is returned:
    - we predicted y_i accurately (y_i and $w \cdot x_i + b$ have the same sign) and $\gamma_i \geq 1$
2. value > 0 is returned:
    - we predicted y_i incorrectly (y_i and $w \cdot x_i + b$ have different signs)
    - we predicted y_i accurately but $\gamma_i < 1$

Reminder: γ_i = distance of x_i from the decision boundary

Notice that even though we predict y_i correctly, we still incur a small penalty for being too close ($\gamma < 1$) to the decision boundary. This penalty occurs because we want our data points to be at least 1 unit away from the decision boundary to increase the confidence of our prediction.

Recall the value we are trying to minimize in our optimization problem for SVMs:

$$min_w \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \mathcal{E}_i$$

$\sum_{i=1}^{n} \mathcal{E}_i$ represents the hinge loss for each sample in our test set

We solve our optimization problem by taking the regular first derivative of the value to minimize and taking subgradients for the hinge loss function

$$J(w) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \mathcal{E}_i$$

$$J'(w) = \begin{cases} w - Cy_i x_i & \text{if } max(0, 1 - y_i(w \cdot x_i + b)) > 0 \\ w & \text{otherwise} \end{cases}$$

SVM Code Examples

Create the Datasets

Note that we use 0 and 1 to differentiate between the two classes here instead of -1 and 1. This is intentional, so that our datasets can be plotted with the plot_data function imported from utils.

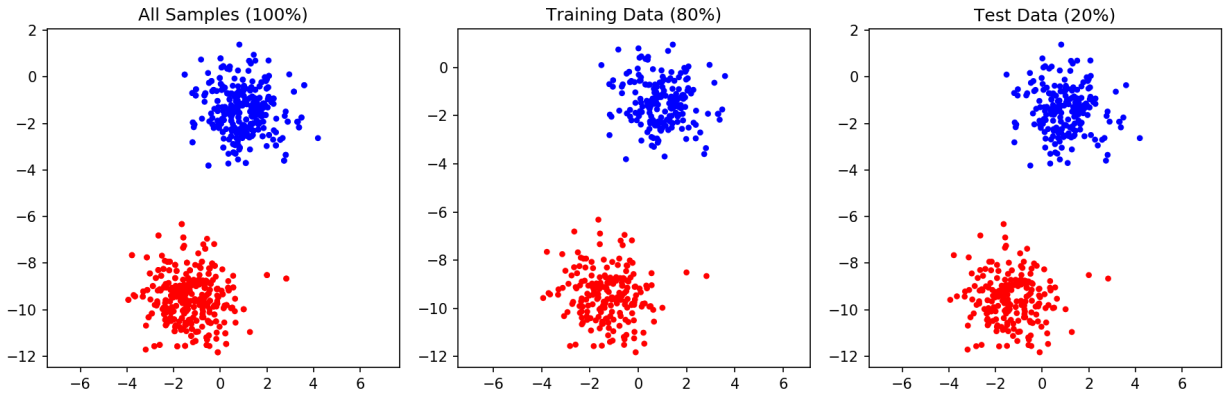
When we do the actual calculations in our implementation of the optimization "fit" function, which does gradient descent on the hinge loss function, we convert the 0's to -1's.

We also pass in an int for the random_state argument to set a seed and control randomness to ensure that we get the same datasets from the make_blob function calls (given that specific int) for testing and validating the model that uses our determined optimal weights θ.

The make_blob function returns a shuffled dataset by default, so we can simply split up the rows in the returned dataset by their index numbers and use the first 80% of the samples for our training set and the remaining 20% of the samples for our test set. We do this for each of the 5 generated datasets.

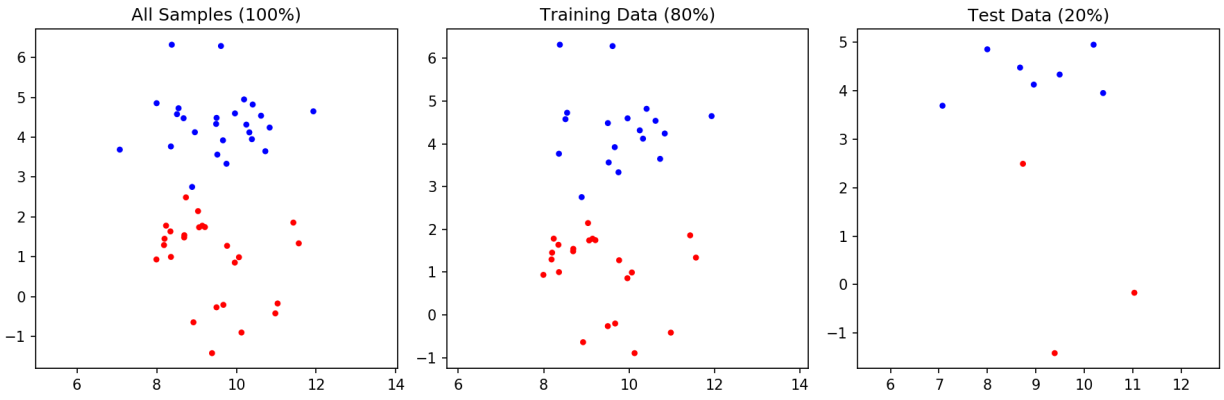
Dataset 1 (linearly separable, medium distance between clusters)

In [13]:
```
X, y = make_blobs(n_samples=500, centers = 2, n_features = 2, random_state=
train_X_1 = X[0:400,]
train_y_1 = y[0:400]
test_X_1 = X[100:,]
test_y_1 = y[100:]

plot_data(train_X_1, train_y_1, test_X_1, test_y_1)
```



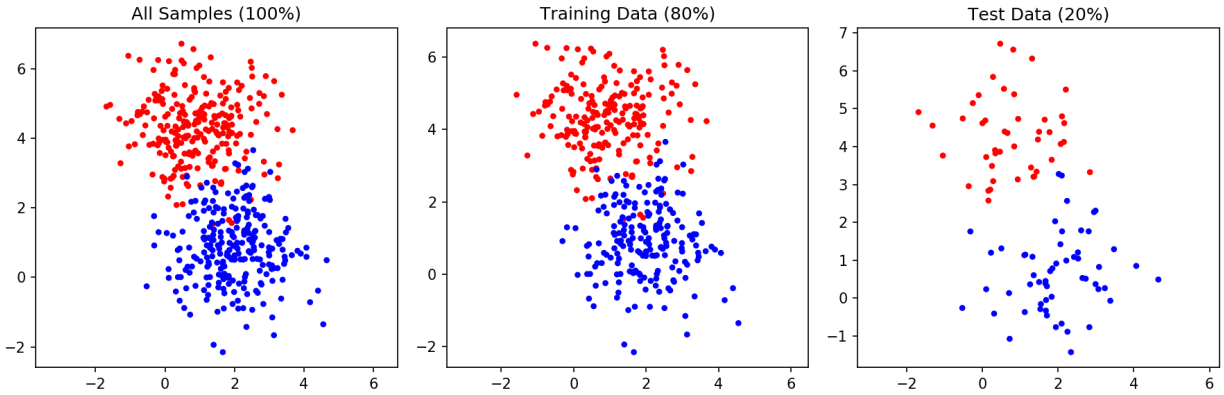
Dataset 2 (linearly separable, small distance between clusters)

In [14]:
```
X, y = make_blobs(n_samples=50, centers = 2, n_features = 2, random_state=4
train_X_2 = X[0:40,]
train_y_2 = y[0:40]
test_X_2 = X[40:,]
test_y_2 = y[40:]

plot_data(train_X_2, train_y_2, test_X_2, test_y_2)
```



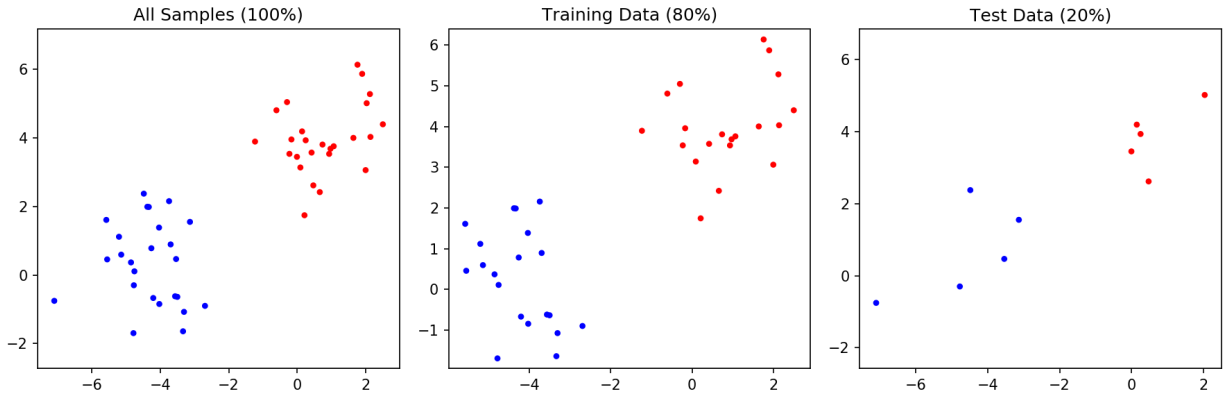
Dataset 3 (nonlinearly separable, some noise)

In [15]:
```
X, y = make_blobs(n_samples=500, centers = 2, n_features = 2, random_state=
train_X_3 = X[0:400,]
train_y_3 = y[0:400]
test_X_3 = X[400:,]
test_y_3 = y[400:]

plot_data(train_X_3, train_y_3, test_X_3, test_y_3)
```



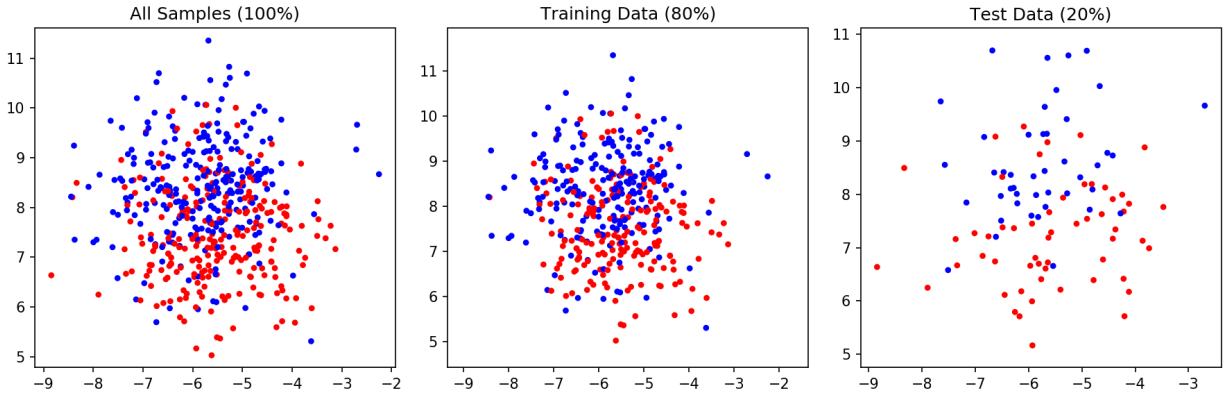
Dataset 4 (linearly separable, large distance between clusters)

In [16]:
```
X, y = make_blobs(n_samples=50, centers = 2, n_features = 2, random_state=3
train_X_4 = X[0:40,]
train_y_4 = y[0:40]
test_X_4 = X[40:,]
test_y_4 = y[40:]

plot_data(train_X_4, train_y_4, test_X_4, test_y_4)
```



Dataset 5 (nonlinearly separable, a lot of noise)

In [17]:
```
X, y = make_blobs(n_samples=500, centers = 2, n_features = 2, random_state=
train_X_5 = X[0:400,]
train_y_5 = y[0:400]
test_X_5 = X[400:,]
test_y_5 = y[400:]

plot_data(train_X_5, train_y_5, test_X_5, test_y_5)
```



1. Example with sklearn.linear_model.SGDClassifier

In [18]:
```
from sklearn import linear_model
```

Construct model using training and test samples from Dataset 1.

Try it Out: Change the datasets used for training and testing the model to see how the classifier algorithm behaves with different datasets.

For example, to switch from dataset 1 to dataset 2, change the following in the code below:

- train_X_1 => train_X_2
- train_y_1 => train_y_2
- test_X_1 => test_X_2
- test_y_1 => test_y_2

The full documentation for the SGDClassifier class is available here: [https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html)](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html)

In [19]:
```
# fit model using SGDClassifier with hinge loss
clf_sgd = linear_model.SGDClassifier(max_iter=10000, tol=1e-3, random_state
clf_sgd.fit(train_X_1, train_y_1)
```

Out[19]:
```
SGDClassifier(alpha=0.01, average=False, class_weight=None,
       early_stopping=False, epsilon=0.1, eta0=0.0, fit_intercept=True,
       l1_ratio=0.15, learning_rate='optimal', loss='hinge',
       max_iter=10000, n_iter=None, n_iter_no_change=5, n_jobs=None,
       penalty='l2', power_t=0.5, random_state=0, shuffle=True, tol=0.00
1,
       validation_fraction=0.1, verbose=0, warm_start=False)
```

In [20]:
```
# get model weights
theta_sgd = np.insert(clf_sgd.coef_[0], 0, clf_sgd.intercept_[0]).tolist()
print("weights: " + str(theta_sgd))
```

```
weights: [10.850021154191168, 0.42030788283277093, 1.7058616279586047]
```

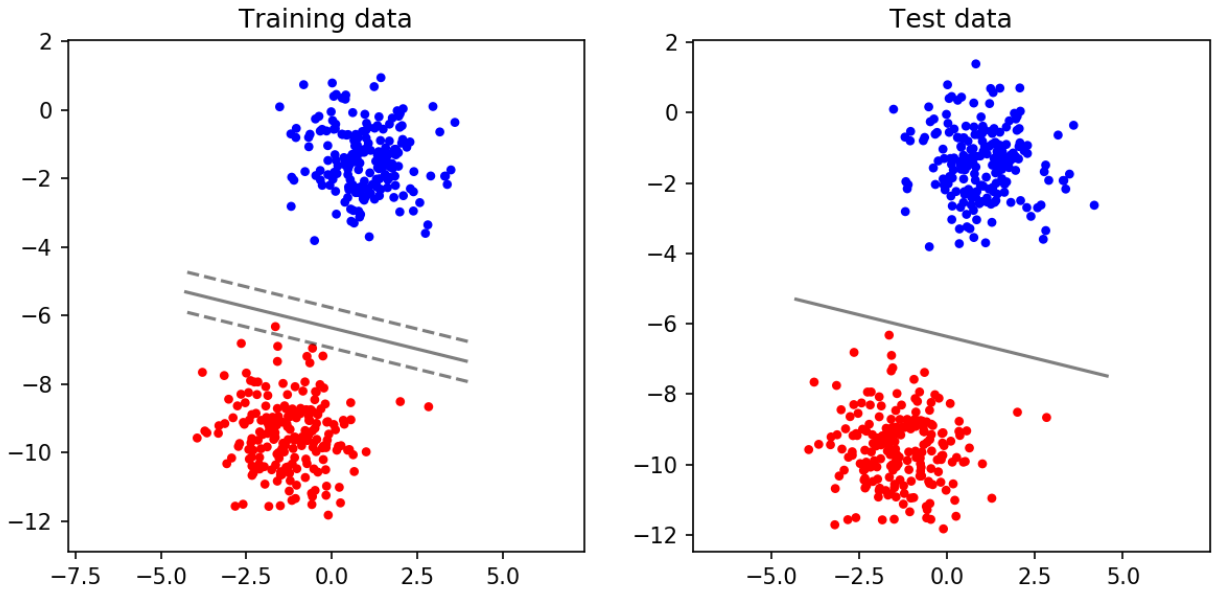
In [21]:
```
# make predictions on test set
pred_sgd = clf_sgd.predict(test_X_1)
test_set_with_pred_sgd = np.column_stack((test_X_1, pred_sgd))
print(test_set_with_pred_sgd)
```

```
[[  2.04163822  -0.96078962   1.        ]
 [ -2.29456502 -10.59673933   0.        ]
 [  2.82859067  -8.66035449   0.        ]
 ...
 [  4.1851508   -2.62977901   1.        ]
 [ -1.68399423 -10.86599403   0.        ]
 [  0.53411463  -0.38180428   1.        ]]
```

In [22]:
```
# get accuracy on test set predictions
num_correct_sgd = sum(pred_sgd == test_y_1)
num_classified = len(test_y_1) * 1.0
print("accuracy rate: " + str(num_correct_sgd / num_classified))
```

```
accuracy rate: 1.0
```

In [23]:
```
# plot visualizations with function from utils
plot_decision_function(train_X_1, train_y_1, test_X_1, test_y_1, clf_sgd)
```



2. Example with sklearn.svm.SVC

Construct model using training and test samples from Dataset 1.

Try it Out: Change the datasets used for training and testing the model to see how the classifier algorithm behaves with different datasets.

For example, to switch from dataset 1 to dataset 2, change the following in the code below:

- train_X_1 => train_X_2
- train_y_1 => train_y_2
- test_X_1 => test_X_2
- test_y_1 => test_y_2

Note: The SVC classifier constructor has a large number of optional arguments, many of which are not specified here; we assume the default values for most of them. The *kernel* argument specifies which kernel method to use. SVC offers the ability to construct an SVM model that uses the kernel trick, which we briefly touch upon later in this guide. For this code example, we use the *linear* kernel to stay consistent with our examples for SGDClassifier and our own implementation and construct a linear decision boundary. We include this example using SVC to compare the results of the SVM model generated by SVC with the models from SGDClassifier and our own implementation.

The full documentation for the SVC class is available here: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html (https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html)

In [24]:
```
# fit model using svm.SVC class
from sklearn.svm import SVC
clf_svc = SVC(C=1.0, kernel='linear', random_state=0, max_iter=10000)
clf_svc.fit(train_X_1, train_y_1)
```

Out[24]:
```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='linear', max_iter=10000, probability=False, random_state=0,
  shrinking=True, tol=0.001, verbose=False)
```

In [25]:
```
# get model weights
theta_svc = np.concatenate([clf_svc.intercept_, clf_svc.coef_[0]])
print(theta_svc)
```

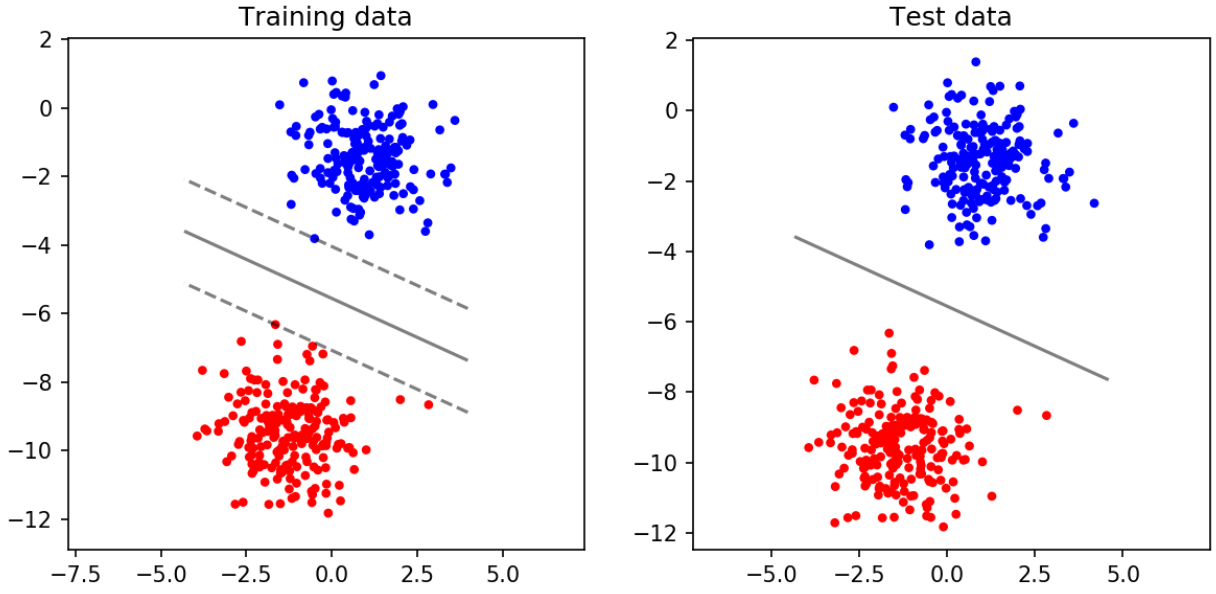
```
[3.66467415 0.30037281 0.65953095]
```

In [26]:
```
# make predictions on test set
pred_svc = clf_svc.predict(test_X_1)
test_set_with_pred_svc = np.column_stack((test_X_1, pred_svc))
#print(test_set_with_pred_svc)
```

In [27]:
```
# get accuracy on test set predictions
num_correct_svc = sum(pred_svc == test_y_1)
num_classified = len(test_y_1) * 1.0
print("accuracy rate: " + str(num_correct_svc / num_classified))
```

```
accuracy rate: 1.0
```

In [28]:
```
# plot visualizations with function from utils
plot_decision_function(train_X_1, train_y_1, test_X_1, test_y_1, clf_svc)
```



3. Example with an implementation of gradient descent on hinge loss from scratch

In [29]:
```
# classify based on which side of the decision boundary X is on
# returns 0 or 1 instead of -1 or 1 for plotting function purposes
def svm_gd_hinge_predict(w, X):
    predictions = []

    for test_sample in X:
        # insert 1 for the intercept b
        x = np.insert(test_sample, 0, 1)
        pred = np.dot(w, x)
        classification = 1 if pred >= 0 else 0
        predictions += [classification]

    return np.array(predictions)
```

In [30]:
```
def svm_gd_hinge_fit(X, y, alpha=0.01, max_iter=10000, C=1.0):

    num_samples = X.shape[0]
    num_features = X.shape[1]
    learning_rate = alpha

    # initialize weights vector +1 for b
    w = np.array([0.0] * (num_features + 1))

    for i in range(max_iter):

        total_hinge_loss = np.array([0.0] * (num_features + 1))

        for j in range(num_samples):

            x = np.insert(X[j], 0, 1)

            label = -1 if y[j] == 0 else 1

            # incur penalty if margin < 1.0
            if label * np.dot(w, x) < 1.0:
                total_hinge_loss += learning_rate * (w - C * label * x)
                w = w - learning_rate * (w + C * -label * x)
            else:
                total_hinge_loss += (learning_rate * w)

        # update weights
        avg_total_hinge_loss = total_hinge_loss / num_samples
        w = w - avg_total_hinge_loss

        # update learning rate
        learning_rate = learning_rate * 0.95

    return w
```

Ex 1: Train and Test on Dataset 1 (linearly separable, medium distance between clusters)

In [31]:
```
# fit model using self-defined implementation of SGD on hinge loss
weights = svm_gd_hinge_fit(train_X_1, train_y_1)
print(weights)

# make predictions on test set
pred = svm_gd_hinge_predict(weights, test_X_1)
test_set_with_pred = np.column_stack((test_X_1, pred))

# get accuracy on test set predictions
num_correct_self = sum(pred == test_y_1)
num_classified = len(test_y_1) * 1.0
print("accuracy rate: " + str(num_correct_self / num_classified))
```

```
[0.66137505 0.40115971 0.17319011]
accuracy rate: 0.985
```

Ex 2: Train and Test on Dataset 2 (linearly separable, small distance between clusters)

In [32]:
```
# fit model using self-defined implementation of SGD on hinge loss
weights = svm_gd_hinge_fit(train_X_2, train_y_2)

# make predictions on test set
pred = svm_gd_hinge_predict(weights, test_X_2)
test_set_with_pred = np.column_stack((test_X_2, pred))

# get accuracy on test set predictions
num_correct_self = sum(pred == test_y_2)
num_classified = len(test_y_2) * 1.0
print("accuracy rate: " + str(num_correct_self / num_classified))
```

```
accuracy rate: 1.0
```

Ex 3: Train and Test on Dataset 3 (nonlinearly separable, some noise)

In [33]:
```
# Note: When the data has some noise and is therefore not perfectly linearl
# Verify this for yourself by altering parameter values in the code example

# fit model using self-defined implementation of SGD on hinge loss
weights = svm_gd_hinge_fit(train_X_3, train_y_3, C=6.0)

# make predictions on test set
pred = svm_gd_hinge_predict(weights, test_X_3)
test_set_with_pred = np.column_stack((test_X_3, pred))

# get accuracy on test set predictions
num_correct_self = sum(pred == test_y_3)
num_classified = len(test_y_3) * 1.0
print("accuracy rate: " + str(num_correct_self / num_classified))
```

```
accuracy rate: 0.96
```

Ex 4: Train and Test on Dataset 4 (linearly separable, large distance between clusters)

In [34]:
```
# fit model using self-defined implementation of SGD on hinge loss
weights = svm_gd_hinge_fit(train_X_4, train_y_4)

# make predictions on test set
pred = svm_gd_hinge_predict(weights, test_X_4)
test_set_with_pred = np.column_stack((test_X_4, pred))

# get accuracy on test set predictions
num_correct_self = sum(pred == test_y_4)
num_classified = len(test_y_4) * 1.0
print("accuracy rate: " + str(num_correct_self / num_classified))
```

```
accuracy rate: 1.0
```

Ex 5: Train and Test on Dataset 5 (nonlinearly separable, a lot of noise)

In [35]:
```
# fit model using self-defined implementation of SGD on hinge loss
weights = svm_gd_hinge_fit(train_X_5, train_y_5)

# make predictions on test set
pred = svm_gd_hinge_predict(weights, test_X_5)
test_set_with_pred = np.column_stack((test_X_5, pred))

# get accuracy on test set predictions
num_correct_self = sum(pred == test_y_5)
num_classified = len(test_y_5) * 1.0
print("accuracy rate: " + str(num_correct_self / num_classified))
```

```
accuracy rate: 0.44
```

Summary of Model Accuracy on Datasets 1-5

Continue on to the next section for explanations on our process of tuning the hyperparameters to improve the models' accuracies

Before Tuning: using the default hyperparameter values

Model	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5
SGDClassifier	1.0	0.9	0.98	1.0	0.80
SVC	1.0	1.0	0.98	1.0	0.79
Our Implementation	0.985	1.0	0.96	1.0	0.44

After Tuning: using the tuned hyperparameter values

Model	Dataset 1	Dataset 2	Dataset 3	Dataset 4	Dataset 5
SGDClassifier	1.0	1.0 (B)	0.98(C)	1.0	0.81 (F)
SVC	1.0	1.0	0.98(D)	1.0	0.80 (G)
Our Implementation	1.0 (A)	1.0	0.96(E)	1.0	0.65 (H)

Plotting Learning Curves and Tuning Hyperparameters

Notice that when we use the default hyperparameter values, our optimization function performs fairly well and has a similar accuracy to sklearn's SGDClassifier and SVC models for datasets 1-4.

Tuning the hyperparameters means finding the optimal range for the hyperparameter values that lead to the highest model accuracy. Our hyperparameters in this example are the learning rate α and regularization parameter C.

We performed grid search and plotted learning curves for SGDClassifier, SVC, and our optimization function on Datasets 1-5. Grid search involves testing the model on all possible combinations (the cartesian product) of the hyperparameters.

We plotted learning curves for the different classifier models (SGDClassifier, SVC, our implementation) on all combinations of (α, C) using the following list of possible values:

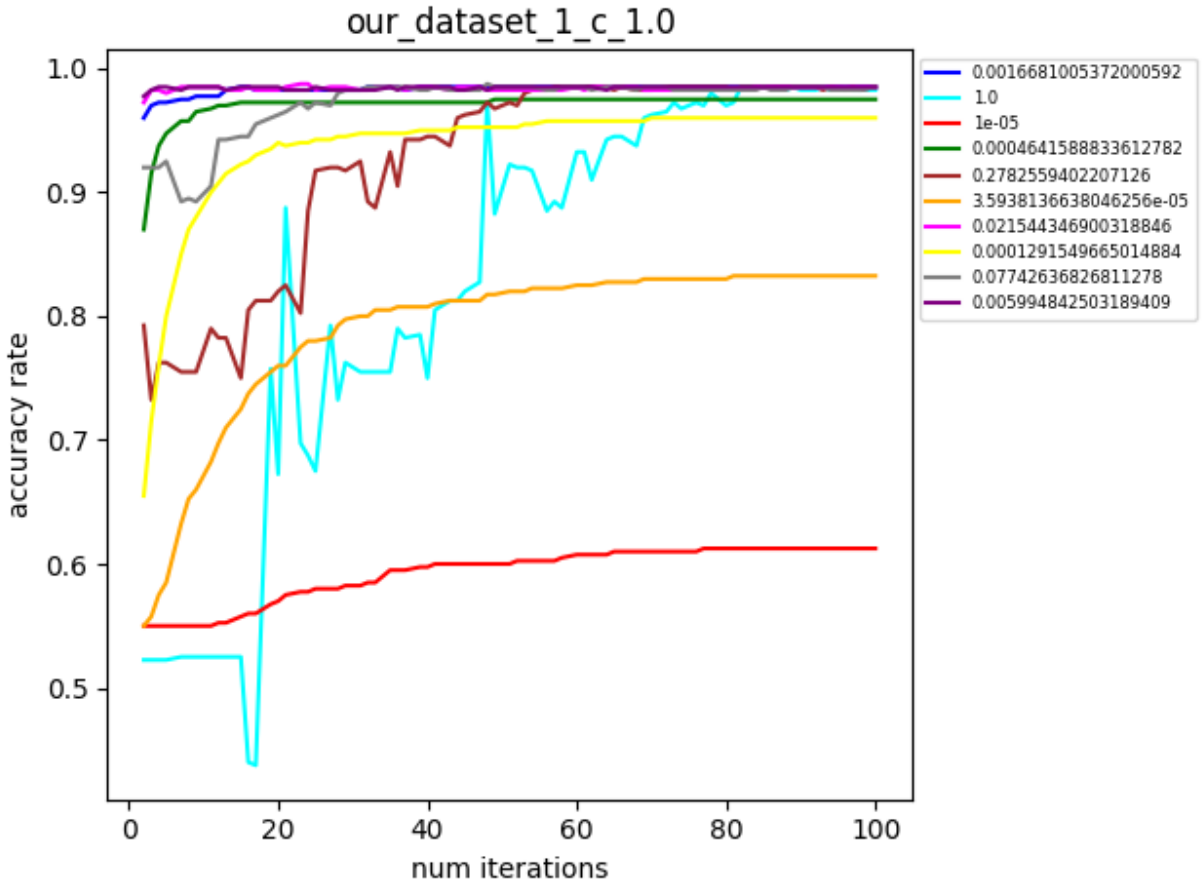
C = [1.0, 2.0, 5.0, 10.0]
α = np.logspace(-5, 0, num=10)

In [36]:
```
alphas = np.logspace(-5, 0, num=10)
print("learning rates: " + str(alphas))
```

```
learning rates: [1.00000000e-05 3.59381366e-05 1.29154967e-04 4.64158883e
-04
 1.66810054e-03 5.99484250e-03 2.15443469e-02 7.74263683e-02
 2.78255940e-01 1.00000000e+00]
```

Each graph has a specified C value that was held constant and shows the learning curve for the 10 different alpha values, presented by the different colored learning curves which match with the key shown to the right of the graph. For example, plotting learning curves for Dataset 1 using SGDClassifier would result in 4 graphs (1 for each C value tested) where each graph contains 10 learning curves (1 for each alpha value tested).

The python script that we wrote and used to create the graphs and save them to png images is available in our repository, 'plot_learning_curves.py'. We show an example of one of the generated learning curve graphs below, but do not include all of the images in this guide to conserve space. The learning curve graphs for each of the 3 classifiers evaluated are available for you to view separately in the subfolder 'learning_curves' in this repository.



Discussion: Tuning Hyperparameters to Improve Model Accuracy

Dataset 1 (linearly separable, medium distance between clusters)

(A): We see from our plotted learning curves that using a smaller learning rate than the default 0.01 and a C value greater than 1.0 improves the accuracy of our optimization function from 0.985 at the default values to 1.0 with tuned hyperparameter values.

The following combinations result in models with an accuracy rate of 1.0

- alpha = 4.64158883e-04, C = 5.0, 10.0
- alpha = 1.66810054e-03, C = 5.0, 10.0
- alpha = 5.99484250e-03, C = 2.0, 5.0, 10.0
- alpha = 2.15443469e-02, C = 2.0, 5.0, 10.0
- alpha = 7.74263683e-02, C = 2.0, 5.0, 10.0
- alpha = 2.78255940e-01, C = 5.0, 10.0
- alpha = 1.00000000e+00, C = 5.0, 10.0

These results demonstrate that increasing C (the penalty for misclassification) helps regulate our model and prevent it from overfitting to the training data. The alpha values below 4.64158883e-04 would most likely also reach an accuracy rate of 1.0 with a higher C value if we had let them run for more iterations. We did not do this for plotting the learning curves as that would have been very slow and inefficient.

In [37]:
```
# using our optimization fn
weights = svm_gd_hinge_fit(train_X_1, train_y_1, alpha=4.64158883e-04, C =
pred = svm_gd_hinge_predict(weights, test_X_1)
num_correct_self = sum(pred == test_y_1)
num_classified = len(test_y_1) * 1.0
print("accuracy rate: " + str(num_correct_self / num_classified))
```

```
accuracy rate: 1.0
```

Dataset 2 (linearly separable, small distance between clusters)

(B): The SGDClassfier model does not take in a C value. Our learning curves help us to tune alpha to improve the SGDClassifier model accuracy for Dataset 2. Our original chosen alpha = 0.01 resulted in an accuracy of 0.9. The model accuracy can be improved to 1.0 when alpha is tuned to one of:

- 1.66810054e-03
- 5.99484250e-03
- 2.15443469e-02
- 1.00000000e+00

In [38]:
```
# using SGDClassifier
clf_sgd = linear_model.SGDClassifier(max_iter=100, tol=1e-3, random_state=0
clf_sgd.fit(train_X_2, train_y_2)
pred_sgd = clf_sgd.predict(test_X_2)
num_correct_sgd = sum(pred_sgd == test_y_2)
num_classified = len(test_y_2) * 1.0
print("accuracy rate: " + str(num_correct_sgd / num_classified))
```

```
accuracy rate: 1.0
```

Dataset 3 (nonlinearly separable, some noise)

Dataset 3 contains nonlinearly separable data and has some noise, meaning that regardless of where the linear decision boundary lies on the graph, there will always be at least one misclassified test sample and we cannot achieve a 1.0 accuracy rate.

(C): The SGDClassifier model reports a maximum accuracy of 0.98 for Dataset 3 for all of the alpha values tested except 1.29154967e-04 (where the accuracy is 0.90)

(D): The SVC model does not take in an alpha value, but does take in a C value. The model reports a maximum accuracy of 0.98 for all of the C values tested

(E): The presence of noise in Dataset 3 results in high variations in our learning curves. The maximum accuracy for a model trained with our optimization function is 0.96.

We see that when C is small (ie. 1.0) and the alpha value is small, we do not reach the global minimum of the cost function in the allotted maximum number of iterations and our model accuracy is less than 0.96 (ie. alpha values of 1.00000000e-05 and 3.59381366e-05 result in accuracy rates of 0.7 and 0.83 respectively).

When C is larger (ie. 2.0 or 5.0), we achieve an accuracy rate above 0.9 for all of the alpha values *except* the smallest value tested, 1.00000000e-05.

When C is very large (10.0), the accuracy rate will be above 0.9 for *any* of the alpha values tested after at least (approximately) 65 iterations.

Though we cannot achieve an accuracy rate of 1.0 using a linear decision boundary, we see that increasing the misclassification penalty C also increases the SVM model's accuracy rate and allows it to learn well with various learning rates.

In [39]:
```
# using our optimization fn
weights = svm_gd_hinge_fit(train_X_3, train_y_3, alpha=1.00000000e-05  , C
pred = svm_gd_hinge_predict(weights, test_X_3)
num_correct_self = sum(pred == test_y_3)
num_classified = len(test_y_3) * 1.0
print("accuracy rate: " + str(num_correct_self / num_classified))
```

```
accuracy rate: 0.94
```

Dataset 5

Dataset 5 contains nonlinearly separable data with a lot of noise. Any line chosen to separate the data samples by their respective classes will result in a large number of misclassifications.

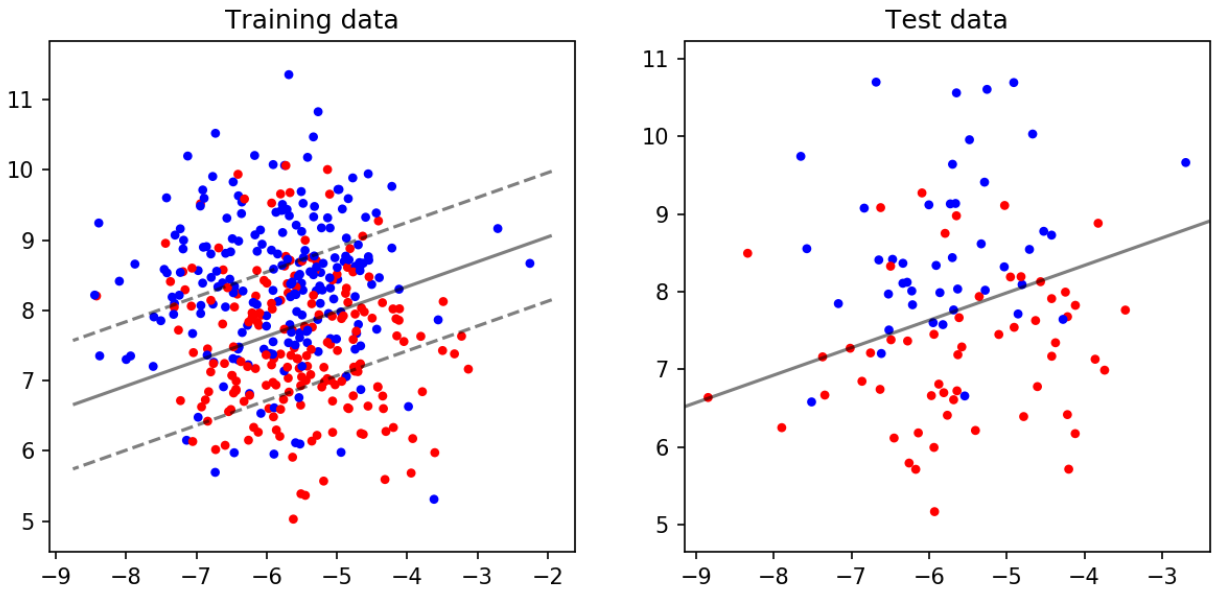
See the next section for new methods to improve model accuracy for very noisy datasets.

The learning curves for all 3 classifiers tested (SGDClassifier, SVC, our optimization function) all show a large amount of volatility for Dataset 5.

(F): We can marginally improve the accuracy of our SGDClassifier model for Dataset 5 from 0.80 to 0.81 by lowering alpha from 0.01 to 5.99484250e-03

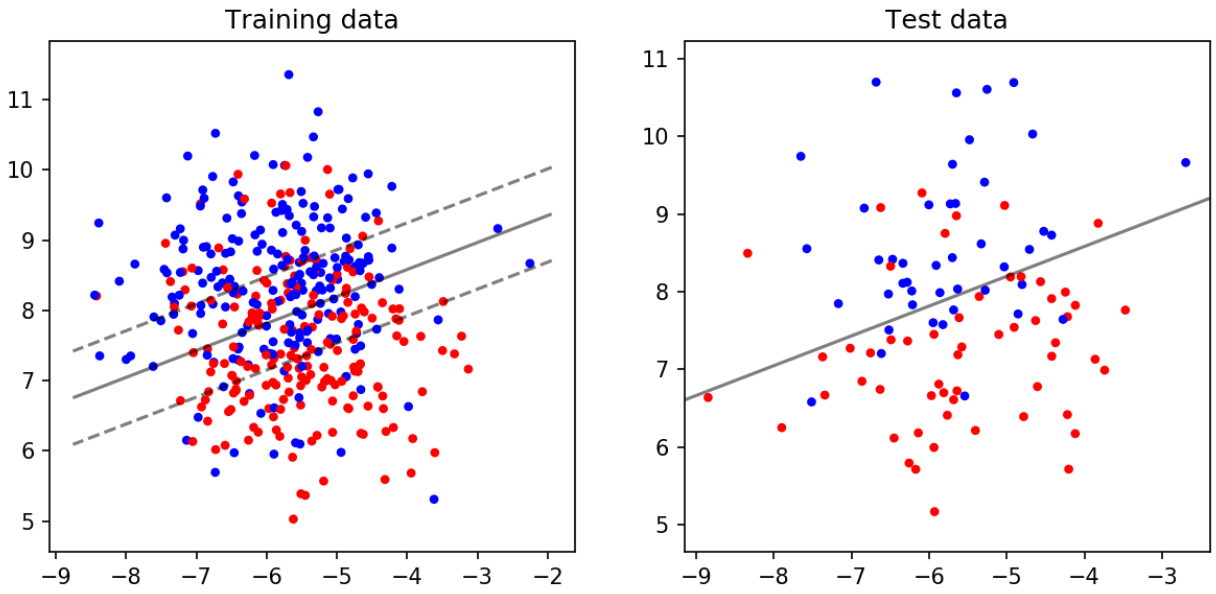
In [40]:
```
# using SGDClassifier - before tuning
clf_sgd = linear_model.SGDClassifier(max_iter=10000, tol=1e-3, random_state
clf_sgd.fit(train_X_5, train_y_5)
pred_sgd = clf_sgd.predict(test_X_5)
num_correct_sgd = sum(pred_sgd == test_y_5)
num_classified = len(test_y_5) * 1.0
print("accuracy rate: " + str(num_correct_sgd / num_classified))
plot_decision_function(train_X_5, train_y_5, test_X_5, test_y_5, clf_sgd)
```

```
accuracy rate: 0.8
```



In [41]:
```
ing SGDClassifier - after tuning
sgd = linear_model.SGDClassifier(max_iter=100, tol=1e-3, random_state=0, alp
sgd.fit(train_X_5, train_y_5)
_sgd = clf_sgd.predict(test_X_5)
correct_sgd = sum(pred_sgd == test_y_5)
classified = len(test_y_5) * 1.0
t("accuracy rate: " + str(num_correct_sgd / num_classified))
_decision_function(train_X_5, train_y_5, test_X_5, test_y_5, clf_sgd)
```

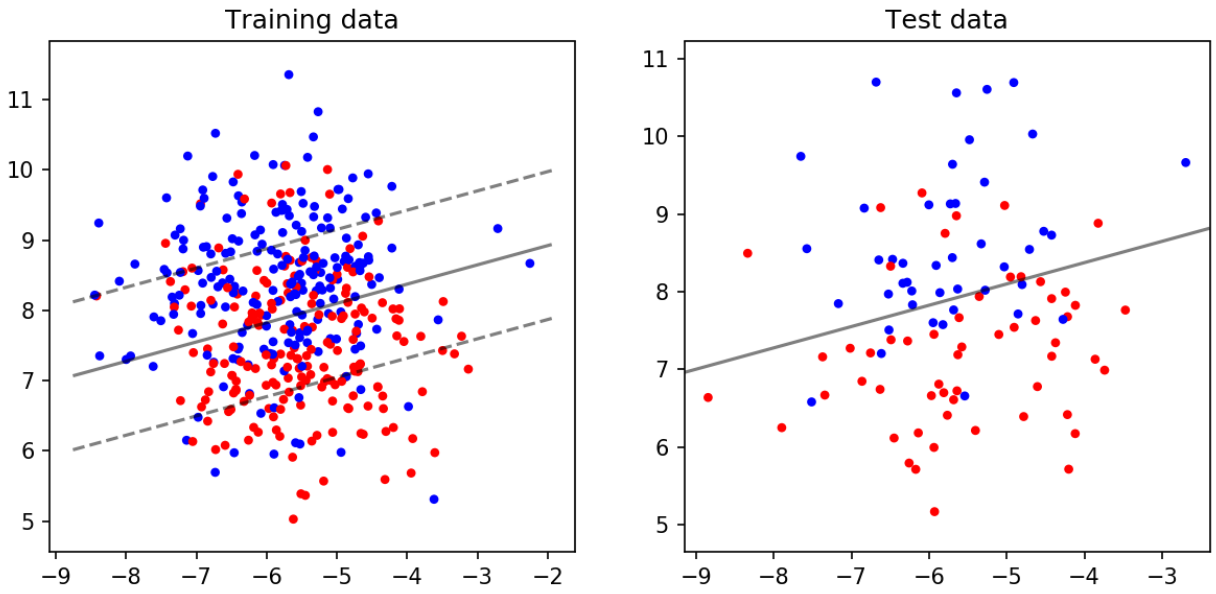
```
accuracy rate: 0.81
```



(G): We can marginally improve the accuracy for our SVC model for Dataset 5 from 0.79 to 0.80 by increasing the misclassification penalty parameter C from 1.0 to 5.0.

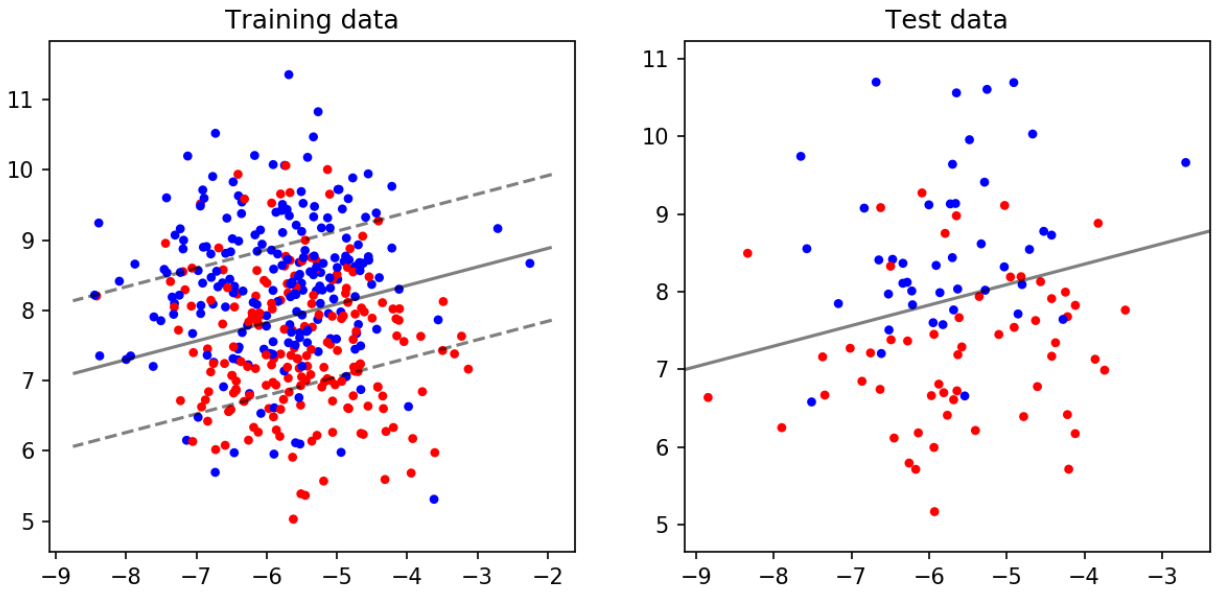
In [42]:
```
# using SVC - before tuning
clf_svc = SVC(C=1.0, kernel='linear', random_state=0, max_iter=10000)
clf_svc.fit(train_X_5, train_y_5)
pred_svc = clf_svc.predict(test_X_5)
num_correct_svc = sum(pred_svc == test_y_5)
num_classified = len(test_y_5) * 1.0
print("accuracy rate: " + str(num_correct_svc / num_classified))
plot_decision_function(train_X_5, train_y_5, test_X_5, test_y_5, clf_svc)
```

```
accuracy rate: 0.79
```



In [43]:
```
# using SVC - after tuning
clf_svc = SVC(C=5.0, kernel='linear', random_state=0, max_iter=10000)
clf_svc.fit(train_X_5, train_y_5)
pred_svc = clf_svc.predict(test_X_5)
num_correct_svc = sum(pred_svc == test_y_5)
num_classified = len(test_y_5) * 1.0
print("accuracy rate: " + str(num_correct_svc / num_classified))
plot_decision_function(train_X_5, train_y_5, test_X_5, test_y_5, clf_svc)
```

```
accuracy rate: 0.8
```



(H): The learning curves for our optimization function show a large amount of volatility for Dataset 5. The highest accuracy we can achieve for Dataset 5 is 0.65 where the misclassification penalty is very large (C = 10.0) and very specific alpha and max_iter combinations are used (alpha=5.99484250e-03, max_iter=58) and (alpha=2.15443469e-02, max_iter=65)

In [44]:
```
# using our optimization fn - before tuning
weights = svm_gd_hinge_fit(train_X_5, train_y_5)
pred = svm_gd_hinge_predict(weights, test_X_5)
num_correct_self = sum(pred == test_y_5)
num_classified = len(test_y_5) * 1.0
print("weights: " + str(weights))
print("accuracy rate: " + str(num_correct_self / num_classified))
```

```
weights: [-0.02963268  0.01067057  0.11467975]
accuracy rate: 0.44
```

In [45]:
```
# using our optimization fn - after tuning
weights = svm_gd_hinge_fit(train_X_5, train_y_5, alpha=5.99484250e-03, C =
pred = svm_gd_hinge_predict(weights, test_X_5)
num_correct_self = sum(pred == test_y_5)
num_classified = len(test_y_5) * 1.0
print("using hyperparameter values: alpha=5.99484250e-03, max_iter=58")
print("weights: " + str(weights))
print("accuracy rate: " + str(num_correct_self / num_classified))

weights = svm_gd_hinge_fit(train_X_5, train_y_5, alpha=2.15443469e-02, C =
pred = svm_gd_hinge_predict(weights, test_X_5)
num_correct_self = sum(pred == test_y_5)
num_classified = len(test_y_5) * 1.0
print("using hyperparameter values: alpha=2.15443469e-02, max_iter=65")
print("weights: " + str(weights))
print("accuracy rate: " + str(num_correct_self / num_classified))
```

```
using hyperparameter values: alpha=5.99484250e-03, max_iter=58
weights: [-0.44330591  0.1918906   0.21102427]
accuracy rate: 0.65
using hyperparameter values: alpha=2.15443469e-02, max_iter=65
weights: [-0.44750188  0.22984551  0.23969303]
accuracy rate: 0.65
```

Nonlinearly Separable Noisy Data - Higher Order Features and The Kernel Trick

Dataset 5 contains nonlinearly separable data where there is a lot of noise. Looking at the plot of Dataset 5, we see that there is no way to separate the red and blue samples with a straight line without having a significant number of misclassifications.

There are two methods that can be used to separate the data samples and achieve higher model accuracy rates. We will not go into detail about them here, but will simply introduce them:

1. Manipulating the features we do have, x_1 and x_2, to create new features to add to our learning model. Some examples may include, $x_1 x_2$, x_1^2, x_2^2, $(x_1 x_2)^2$, $(x_1 + x_2)^2$
2. Utilising the **kernel trick**. The kernel trick would involve the implementation of new kernel methods, algorithms that can be used to transform the dataset into higher dimensions where there is a clear decision boundary between samples belonging to different classes.

Takeaways from Implementing our Optimization Function Using GD on Hinge Loss

- tuning and finding the optimal range of hyperparameters can have a significant impact on model accuracy
- setting a seed to control the "randomness" of datasets generated from make_blobs is helpful for validating constructed models
- having a large enough sample size per dataset used is important to test the model's accuracy (we initially started with 18 samples to test our optimization function and experienced major improvements in accuracy when we increased to 50 samples later on)
- it is important to test a model on multiple datasets to evaluate how good it is and verify that it is not overfitting to one particular dataset

References:

1. Carpuat, Marine. “(Sub)gradient Descent.” Intro to Machine Learning CMSC 422, Spring 2017, University of Maryland. PowerPoint presentation.
2. Leskovec, Rajaraman, and Ullman. Lecture 70 - Soft Margin SVMs | Mining of Massive Datasets | Stanford University. Youtube, YouTube, 13 Apr. 2016, https://www.youtube.com/watch?v=8xbnLHn4jjQ&list=PLLssT5z_DsK9JDLcT8T62VtzwyW9LNepV&index=70 (https://www.youtube.com/watch?v=8xbnLHn4jjQ&list=PLLssT5z_DsK9JDLcT8T62VtzwyW9LNepV&index=70).
3. Ng, Andrew. Lecture 6.1-6.6 Logistic Regression. Youtube, YouTube, 31 Dec. 2016, https://www.youtube.com/watch?v=-la3q9d7AKQ (https://www.youtube.com/watch?v=-la3q9d7AKQ).
4. Ng, Andrew. Lecture 7.1-7.4 Regularization. Youtube, YouTube, 31 Dec. 2016, https://www.youtube.com/watch?v=u73PU6Qwl1I&list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN&index=40&t=0s (https://www.youtube.com/watch?v=u73PU6Qwl1I&list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN&index=40&t=0s).
5. Ng, Andrew. Lecture 12.1-12.6 Support Vector Machines. Youtube, YouTube, 31 Dec. 2016, https://www.youtube.com/watch?v=hCOIMkcsm_g&list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN&index=70 (https://www.youtube.com/watch?v=hCOIMkcsm_g&list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN&index=70).
6. Ng, Andrew. Part V Support Vector Machines, lecture notes, Machine Learning CS229, Stanford University.
7. Skrikumar, Vivek. “Support Vector Machine: Training with Stochastic Gradient Descent.” Machine Learning, Fall 2018, The University of Utah. PowerPoint presentation.