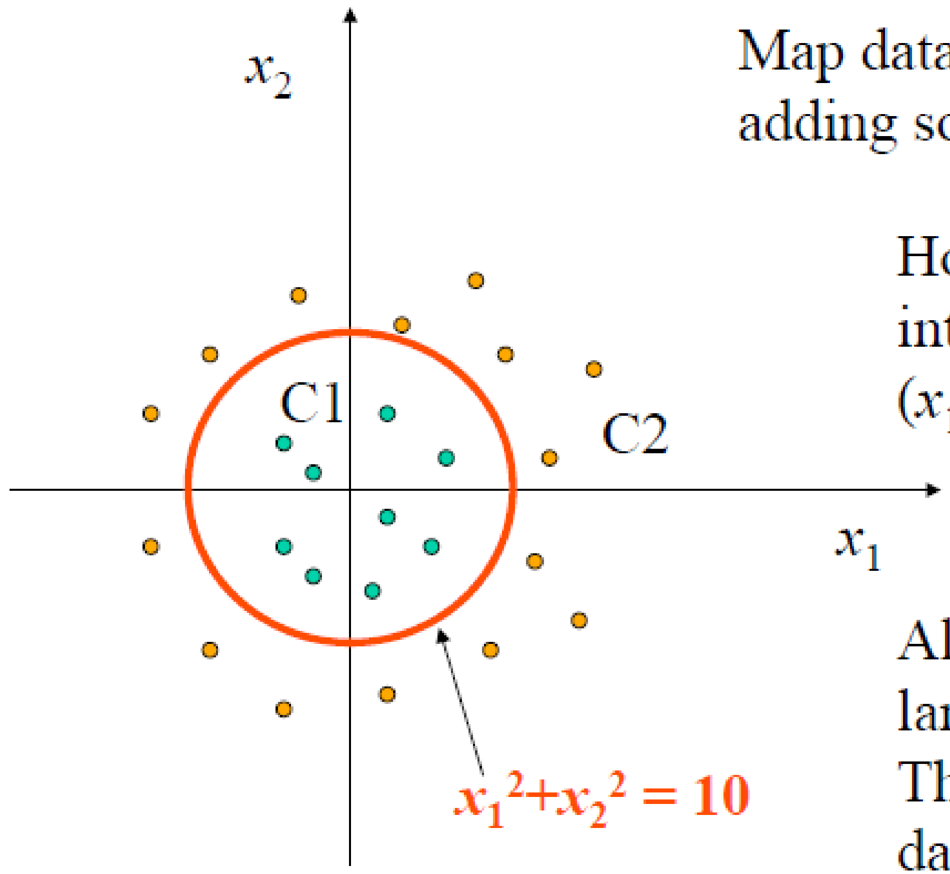
Support Vector Machines

SVM—Support Vector Machines

- A classification method for both linear and nonlinear data
- SVM searches for the linear optimal separating hyperplane (i.e., “decision boundary”)
- For nonlinear data, it maps the original training data into a higher dimension where data is linearly separable
  - Ex: XOR
- SVM finds this hyperplane using support vectors and margins (defined by the support vectors)

Support Vector Machine Approach



Map data point into high dimension, e.g. adding some non-linear features.

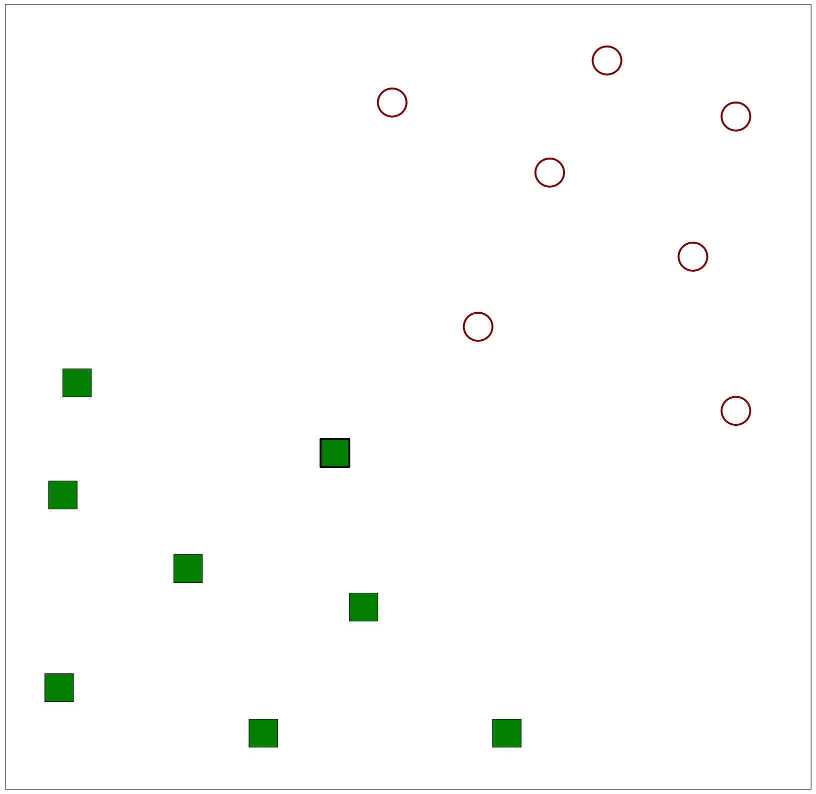
How about we augument feature into three dimension $(x_1, x_2, x_1^2+x_2^2)$.

All data points in class C2 have a larger value for the third feature Than data points in C1. Now data is linearly separable.

SVM—Characteristics and Applications

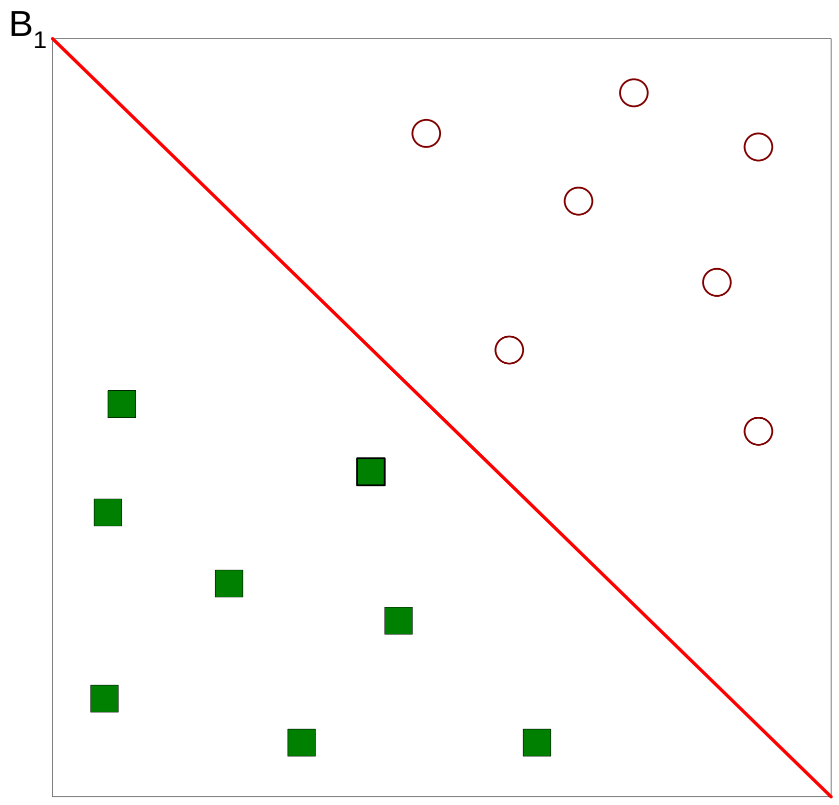
- Training can be slow but accuracy is high
- They are less prone to overfitting than other methods
- The support vectors found also provide a compact description of the learned model
- Finds optimal solution
- Can be used both for classification and prediction
- Applications:
  - handwritten digit recognition, object recognition, speaker identification, text classification

Support Vector Machines



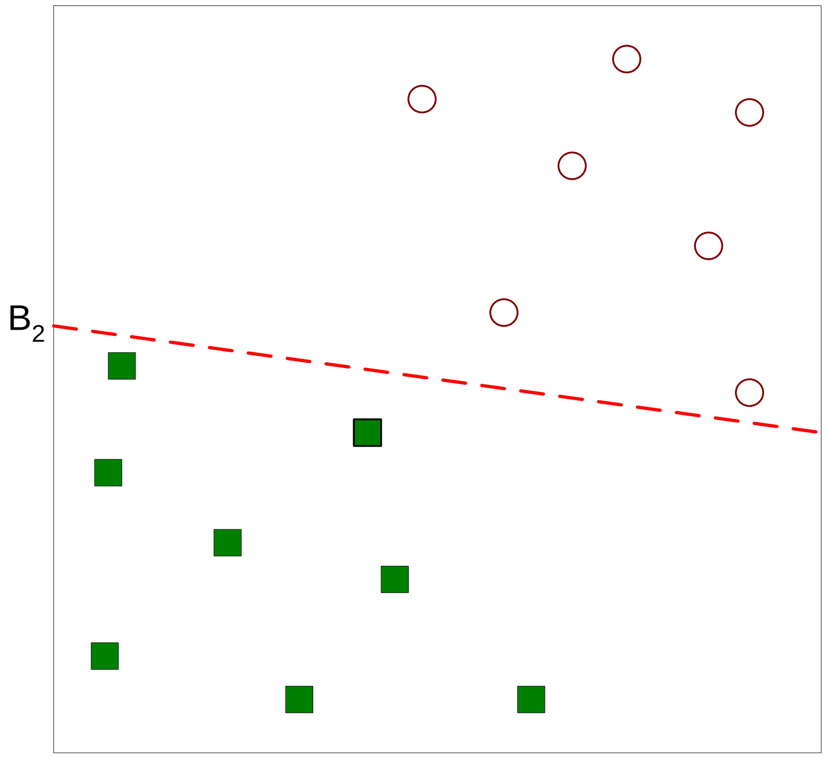
- Find a linear hyperplane (decision boundary) that will separate the data

A Separating Hyperplane



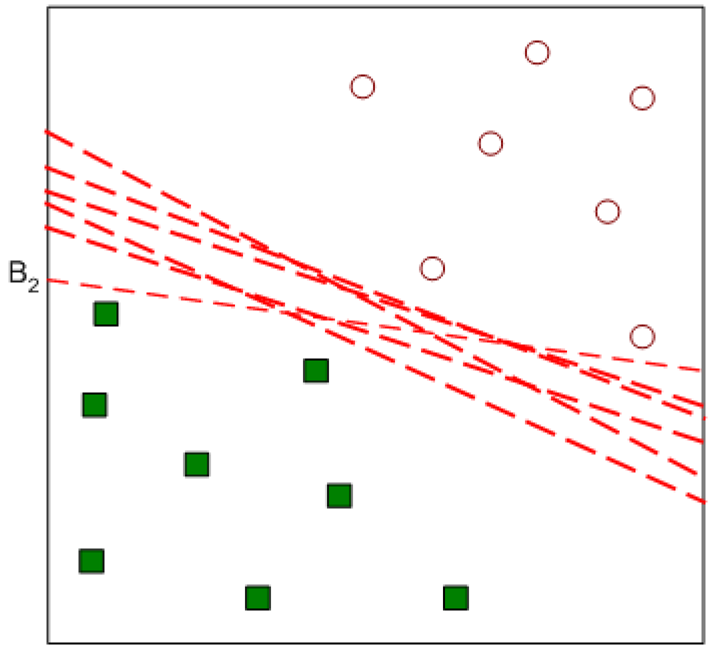
- One Possible Solution

Another Separating Hyperplane



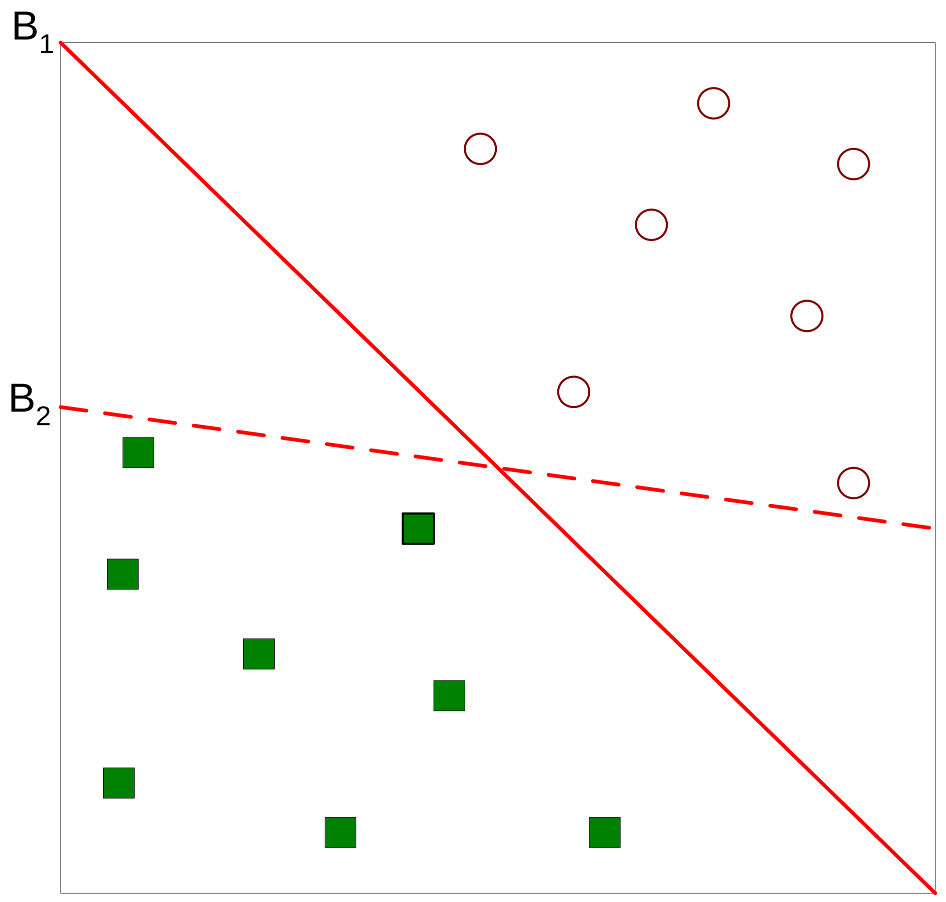
- Another possible solution

Many Separating Hyperplanes



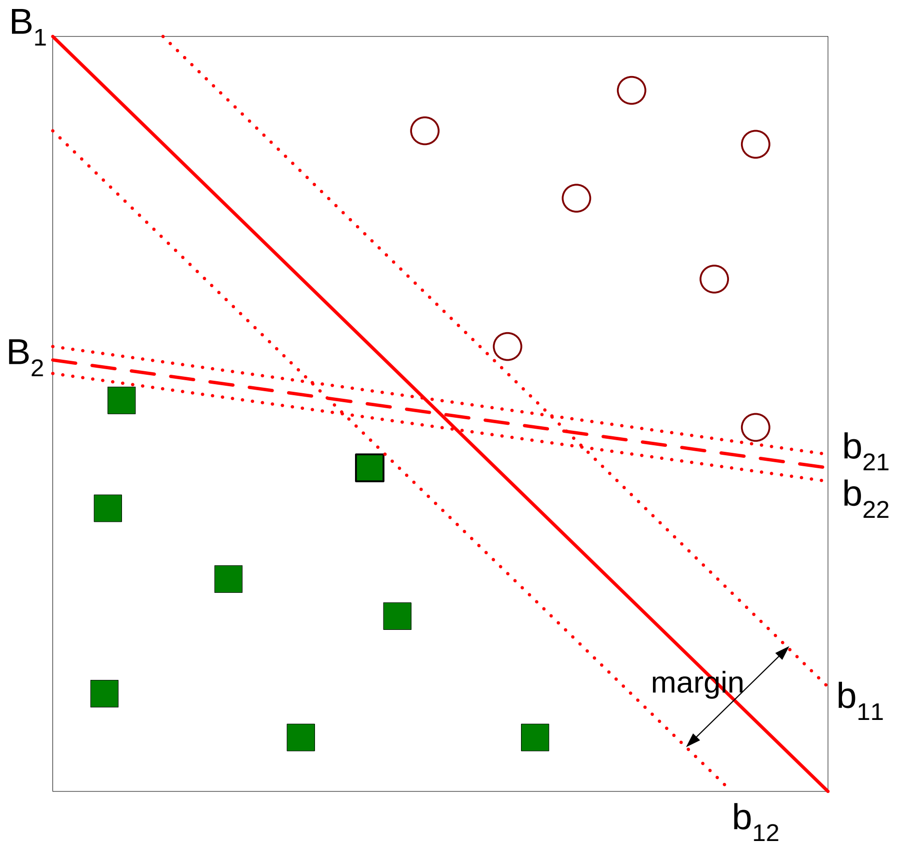
- Other possible solutions

Which Hyperplane to Use?



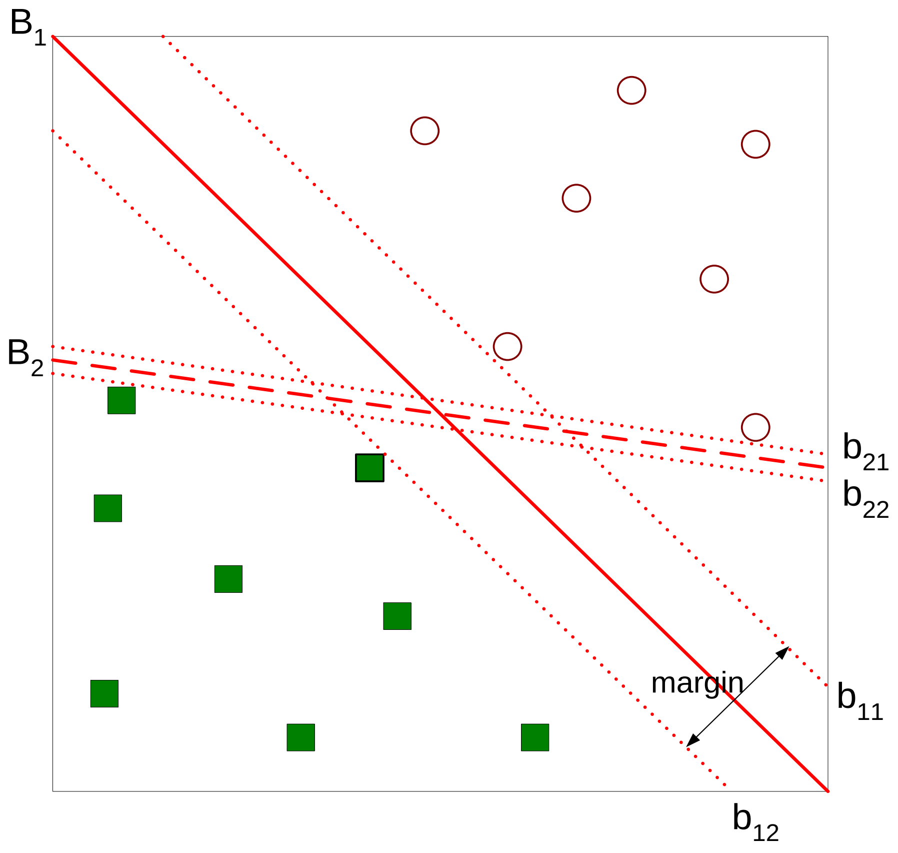
- Which one is better? B1 or B2?
- How do you define better?

Margin Hyperplanes



- **Margin Hyperplanes** are parallel hyperplanes that touch the closest training samples form the different classes
- Distance between margin hyperplanes is called the **margin** of the separating hyperplane Bi

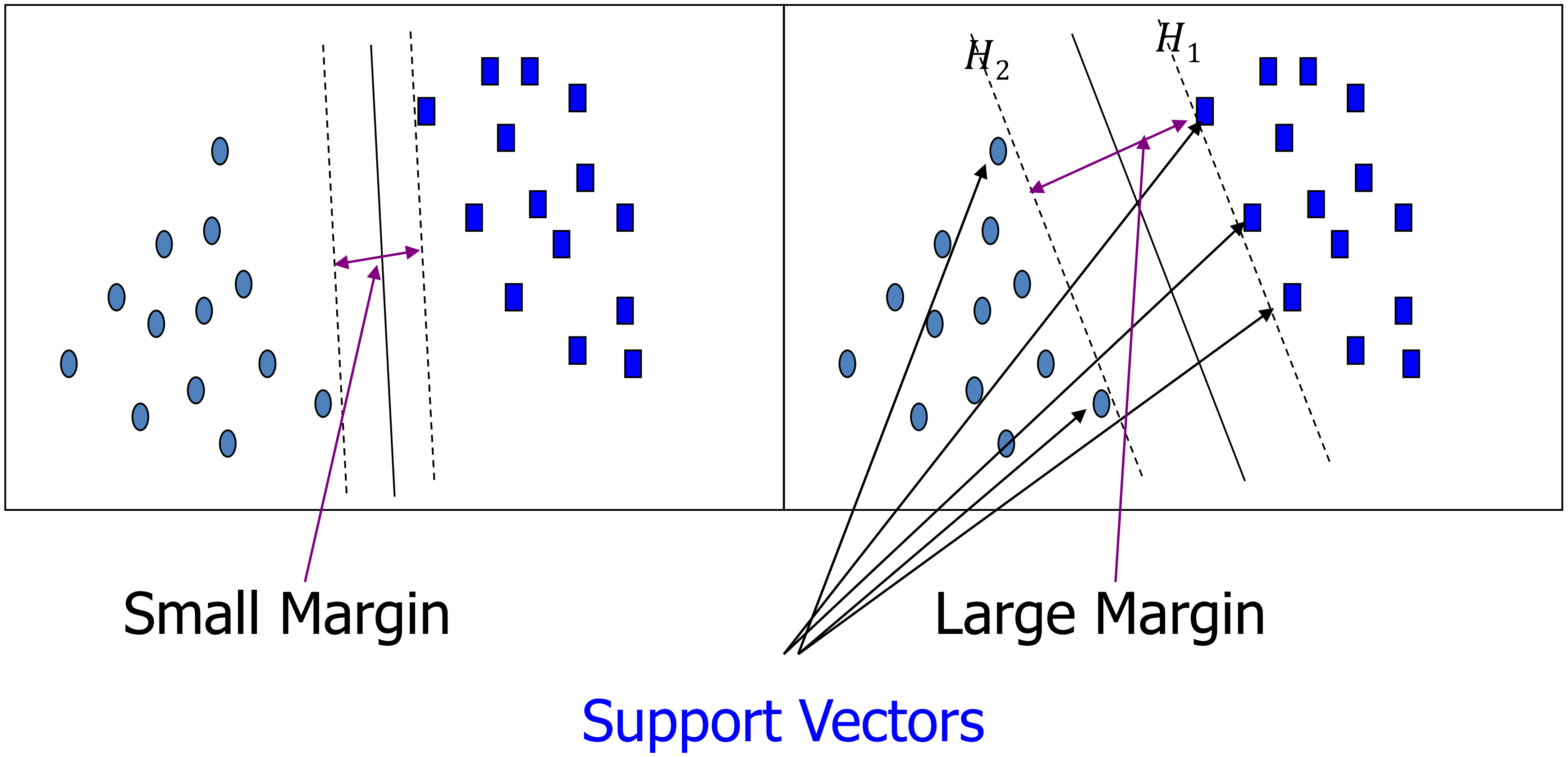
Maximum Margin Hyperplane



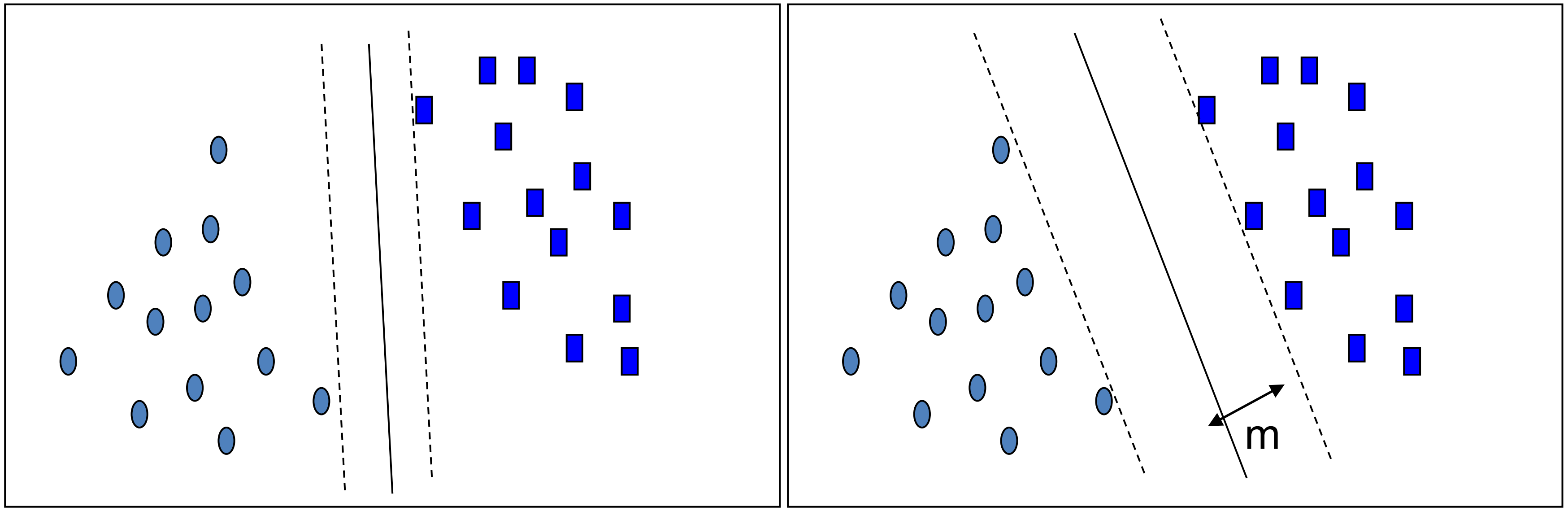
- Find hyperplane that maximizes the margin => B1 is better than B2

Support Vectors

- Any training tuples that fall on hyperplanes H1 or H2 (i.e., the sides defining the margin) are support vectors



SVM – Linearly Separable Data



- Let data D be ($\mathbf{X}_1$, y_1), …, ($\mathbf{X}_{|D|}$, $y_{|D|}$), where the $\mathbf{X}_i$'s is the set of training tuples associated with the class labels y_i 's
- There are infinite lines (hyperplanes) separating the two classes but we want to find the best one
- SVM searches for the hyperplane with the largest margin, i.e., maximum marginal hyperplane (MMH)

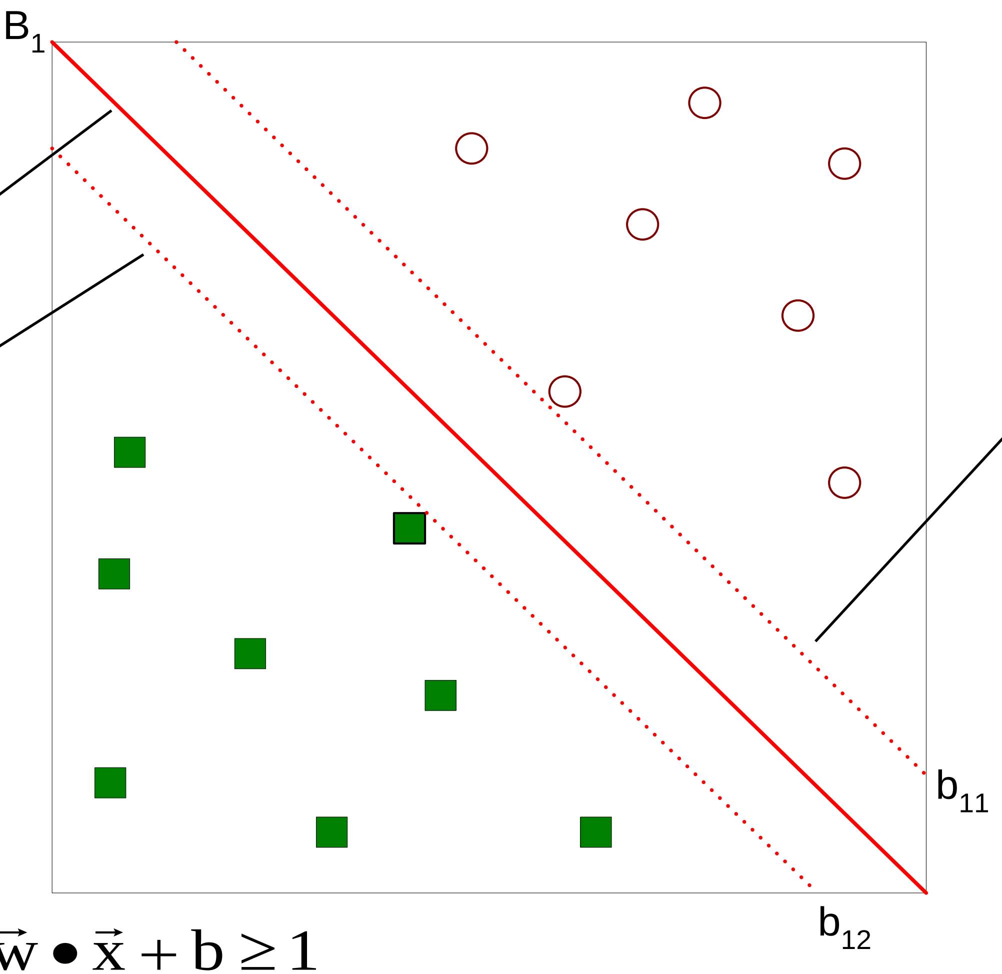
SVM – Linearly Separable Data

B1: x1 + x2 – 2 = 0, can be written as **w.x** + b = 0.
B1: (x1, x2).(1, 1) – 2 = 0.

$$\vec{w} \bullet \vec{x} + b = 0$$

$$\vec{w} \bullet \vec{x} + b = -1$$

$$\vec{w} \bullet \vec{x} + b = +1$$



$$f(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \bullet \vec{x} + b \geq 1 \\ -1 & \text{if } \vec{w} \bullet \vec{x} + b \leq -1 \end{cases}$$

$$\text{Margin} = \frac{2}{\|\vec{w}\|^2}$$

SVM – Linearly Separable Data

$$f(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \bullet \vec{x} + b \geq 1 \\ -1 & \text{if } \vec{w} \bullet \vec{x} + b \leq -1 \end{cases}$$

can be written as $y_i(w_1x_1 + w_2x_2 + \cdots + w_0) \geq 1 \; \forall i$

- This is an optimization problem
- Solving the optimization, the MMH can be rewritten as a decision boundary

$$d(z) = \sum_{i=1}^{n} \alpha_i y_i x_i^T z + b$$

- y_i is the class label of support vector x_i with transpose x_i^T
- α_i and b are parameters determined by the algorithm
- n is the number of support vectors

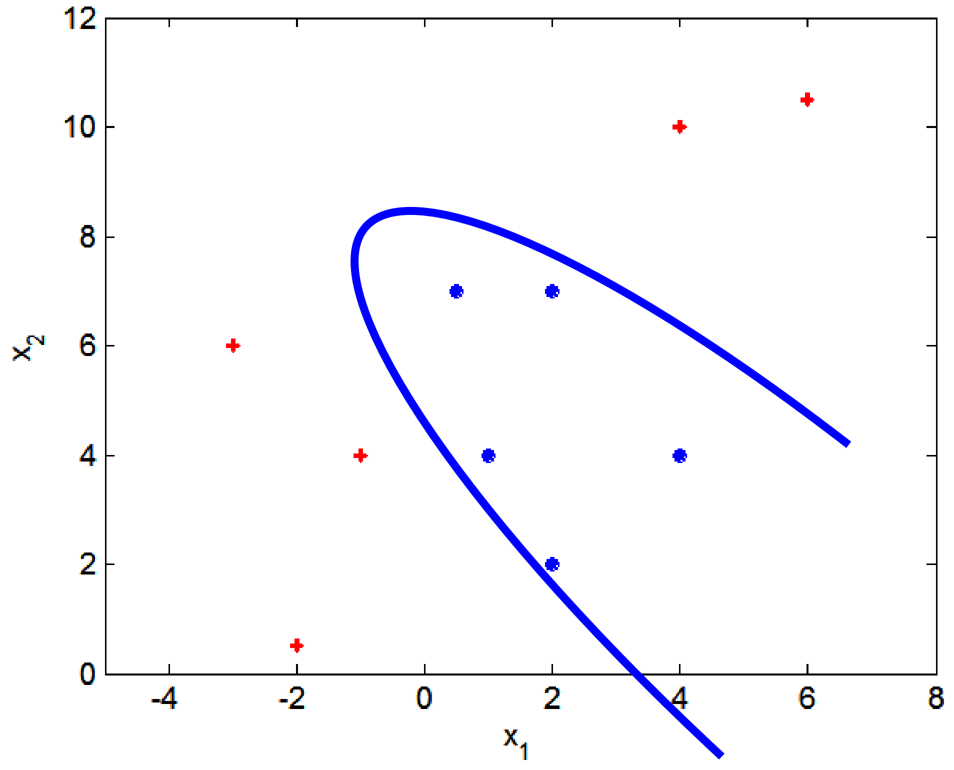
- To classify a new object z, plug in d(z)
  - if sign is +ve (or $\geq$0) assign to the class +1
  - if sign is -ve, assign to the class -1

Role of Support Vectors for SVM

- The complexity of trained classifier is characterized by the # of support vectors
- The support vectors are the essential/critical training examples —they lie closest to the decision boundary
- If all other training examples are removed and the training is repeated, the same separating hyperplane would be found
- An SVM with a small number of support vectors can have good generalization, even when the dimensionality of the data is high

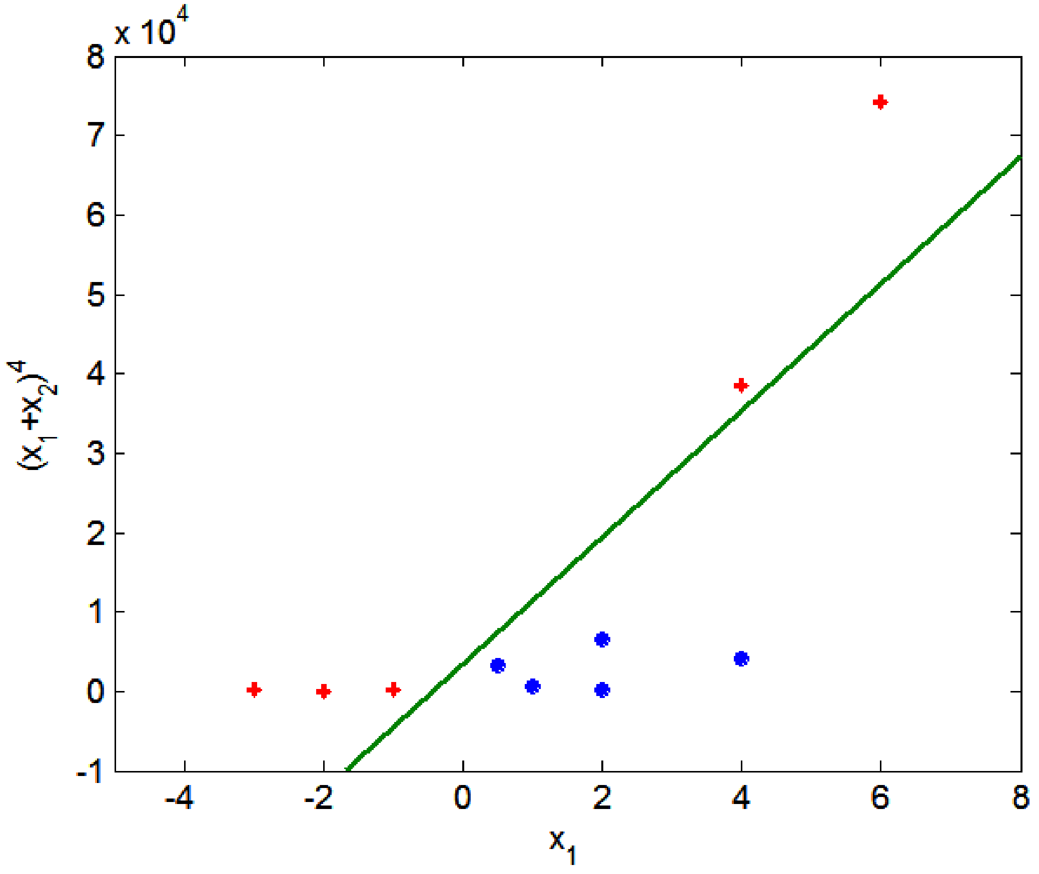
Nonlinear Support Vector Machines

- What if the data are not linearly separable?



Nonlinear Support Vector Machines

- Transform data into higher dimensional space
- Search for a linear separating hyperplane in the new space



Nonlinear Support Vector Machines

- Since data is transformed to a higher dimensional space, computations will be more expensive in the new space
  - To classify a test sample, we need to compute its dot product with every support vector

$$d(z) = \sum_{i=1}^{n} \alpha_i y_i \phi(x_i)^T \phi(z) + b$$

    where ϕ is the nonlinear mapping function
  - During training, a similar dot product is computed several times to find the support vectors and the MMH

The Kernel Trick

- Tuples appear only in the form of dot products $\phi(\boldsymbol{X}_i).\phi(\boldsymbol{X}_j)$
- So we do not need the exact tuples, we only need their dot product
- Instead of computing the dot product on the transformed data tuples, it is mathematically equivalent to apply a kernel function K to the original data tuples
  - i.e., $\mathrm{K}(\boldsymbol{X}_i, \boldsymbol{X}_j) = \phi(\boldsymbol{X}_i).\phi(\boldsymbol{X}_j)$

$$d(z) = \sum_{i=1}^{n} \alpha_i y_i \phi(x_i)^T \phi(z) + b = \sum_{i=1}^{n} \alpha_i y_i \mathrm{K}(xi, z) + b$$

  - this is known as the kernel trick

Kernel Functions

- Typical kernel functions

$$\text{Polynomial kernel of degree } h: \quad K(X_i, X_j) = (X_i \cdot X_j + 1)^h$$

$$\text{Gaussian radial basis function kernel}: \quad K(X_i, X_j) = e^{-\|X_i - X_j\|^2/2\sigma^2}$$

$$\text{Sigmoid kernel}: \quad K(X_i, X_j) = \tanh(\kappa X_i \cdot X_j - \delta)$$

- Another way to write the Gaussians RBF kernel function:

$$K(x_i, x_j) = \exp(-\gamma||x_i - x_j||^2), \gamma > 0$$

Kernel Trick Example

- **Polynomial Kernel:**

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) = (1 + \mathbf{x}_i^T \mathbf{x}_j)^q \qquad (27.58)$$

where q is the degree of the polynomial.

For example, consider the following feature transformation from $\phi : \mathbb{R}^2 \to \mathbb{R}^6$,

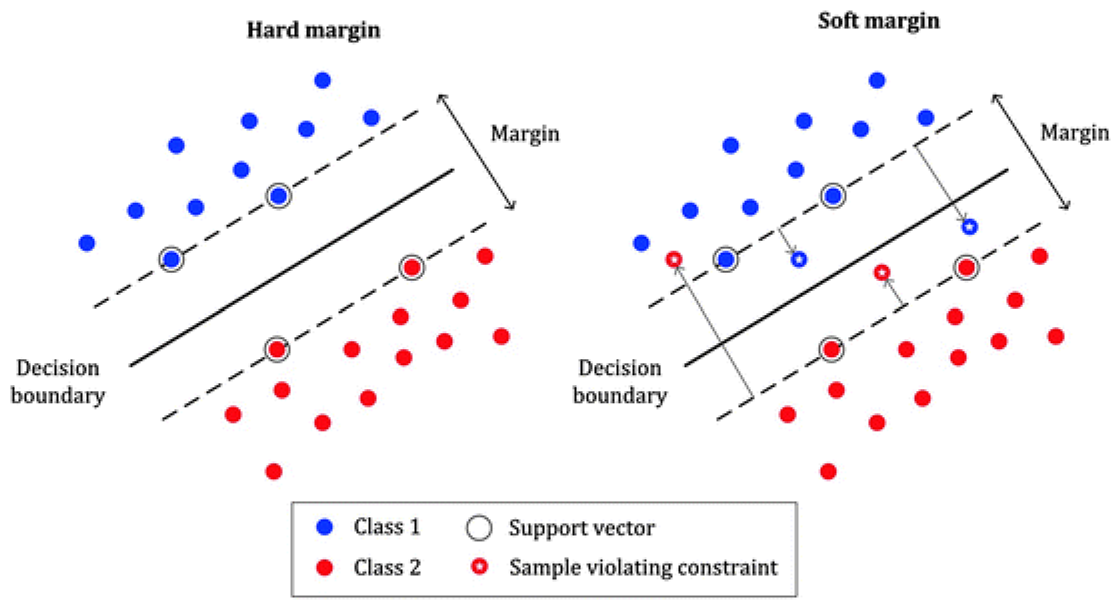
$$\phi(\mathbf{x} = (x_1, x_2)) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2) \qquad (27.59)$$

The kernel in this case is given as:

$$\begin{aligned} K(\mathbf{x}, \mathbf{y}) &= \phi(\mathbf{x})^T \phi(\mathbf{y}) \\ &= (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2)(1, \sqrt{2}y_1, \sqrt{2}y_2, {y_1}^2, {y_2}^2, \sqrt{2}y_1y_2)^T \\ &= 1 + 2x_1y_1 + 2x_2y_2 + x_1^2{y_1}^2 + x_2^2y_2^2 + 2x_1x_2y_1y_2 \\ &= 1^2 + 2(x_1y_1 + x_2y_2) + (x_1y_1 + x_2y_2)^2 \\ &= (1 + (x_1y_1 + x_2y_2))^2 = (1 + \mathbf{x}^T\mathbf{y})^2 \qquad (27.60) \end{aligned}$$

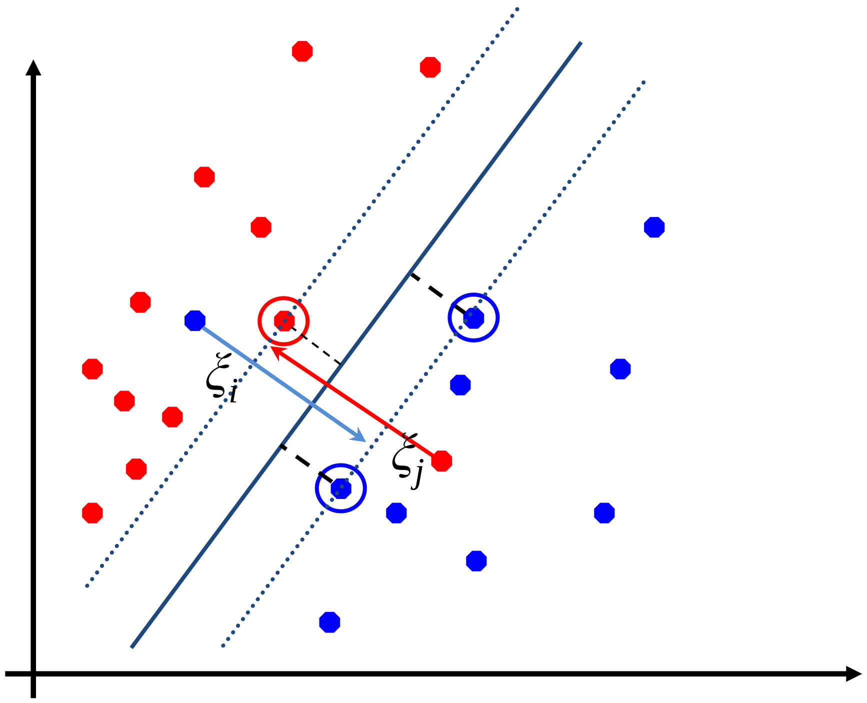
In other words, the dot product in the transformed space, can be expressed as a polynomial kernel in the original space.

Hard Margin vs. Soft Margin Classification



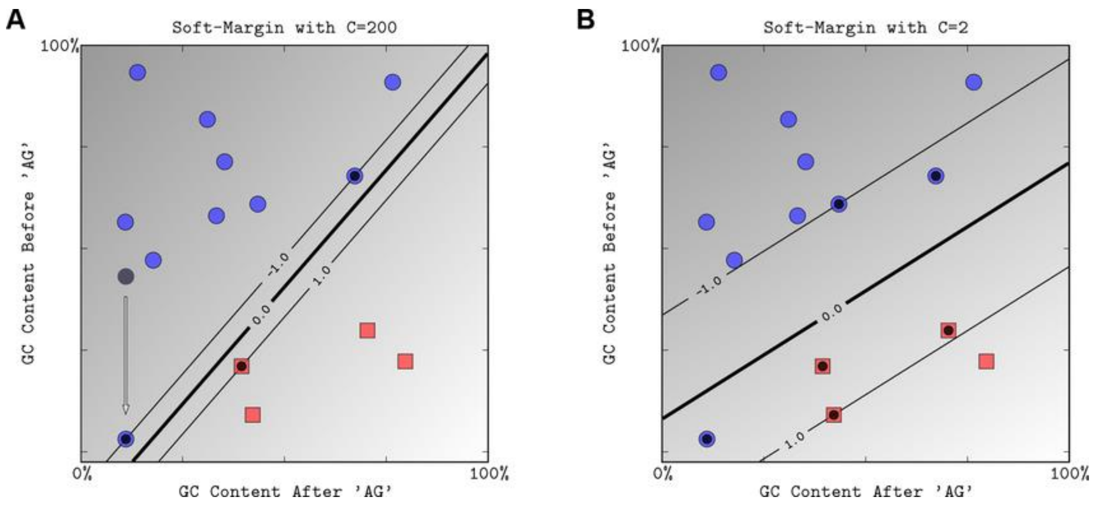
Soft Margin Classification

- A *slack variables* ξ_i is added to allow misclassification of noisy examples or to simplify nonlinear SVM
- Allow some errors
- Still, try to minimize training errors, and
- Place hyperplane "far" from each class (large margin)



Soft Margin Classification (cont.)

- A hyperparameter, C, is used to control trade off
- C controls to what extent we are willing to allow points to sit within the margin or on the wrong side of the decision boundary
- The higher the value of C, the harder the margin and vice versa



Scikit-Learn Zone
Training a Linear SVM Classifier

- We can use either sklearn.svm.**LinearSVC** or sklearn.**svm.SVC**(kernel="linear")
- It is also recommended to standardize data for SVM
  - We can use sklearn.preprocessing.**StandardScaler**

Training a Linear SVM Classifier

- from sklearn.svm import LinearSVC
- from sklearn import datasets
- from sklearn.preprocessing import StandardScaler

- # Load data with only two classes and two features
- iris = datasets.load_iris()
- features = iris.data[:100,:2]
- target = iris.target[:100]

- scaler = StandardScaler() # Standardize features
- features_standardized = scaler.fit_transform(features)

- svc = LinearSVC(C=1.0) # Create support vector classifier

- model = svc.fit(features_standardized, target) # Train model

Using the Classifier

- # Create new observation
- new_observation = [[-2, 3]]

- # Predict class of new observation
- model.predict(new_observation)
- # array([0])

Visualizing the Data Set and the MMH

- A linear decision boundary, a hyperplane, is uniquely specified by a vector normal to the hyperplane, **w,** and an intercept, b.
- The normal vector is contained in the classifier's coef_ attribute
- The intercept is contained in the classifier's intercept_ attribute

- View these two attributes:
- model.coef_
  #array([[1.69023986, -1.23469804]])
- mode.intercept_
  #array([0.27239508])

Visualizing the MMH

- Equation of the hyperplane

$$\vec{w} \bullet \vec{x} + b = 0$$

- $(w1, w2).(x1, x2) + b = 0$ # assume 2D
- $w1x1 + w2x2 + b = 0$
- $w2x2 = -w1x1 - b$
- $x2 = -\frac{w1}{w2}x1 - \frac{b}{w2}$
- i.e., $y = -\frac{w1}{w2}x - \frac{b}{w2}$

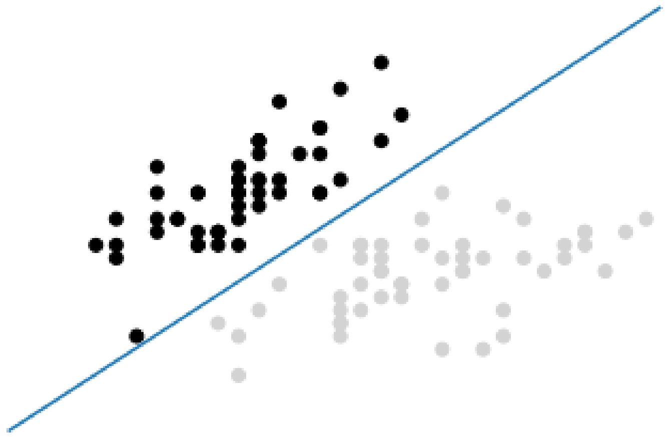
Graph the Dataset and Separating MMH

- import numpy as np
- from matplotlib import pyplot as plt
- %matplotlib inline

- # Plot data points and color using their class
- color = ["black" if c == 0 else "lightgrey" for c in target]
- plt.scatter(features_standardized[:,0], features_standardized[:,1], c=color)

- # Create the hyperplane
- w = svc.coef_[0]
- slope = -w[0] / w[1] # $y = -\frac{w1}{w2}x - \frac{b}{w2}$
- x = np.linspace(-2.5, 2.5)
- y = slope * x - (svc.intercept_[0]) / w[1]

- # Plot the hyperplane
- plt.plot(x, y)
- plt.axis("off")
- plt.show()



Handling Linearly Inseparable Classes Using Kernels

- Use sklearn.svm.**SVC**()
- Parameters include:
  - **kernel**="rbf " | "sigmoid " | "poly" | "linear " (default='rbf')
  - **C**: float, (default=1.0) Penalty parameter C of the error term
  - **degree**: int, optional (default=3) Degree of the polynomial kernel function (‘poly’). Ignored by other kernels
  - **gamma**: float, optional (default=’auto’) Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’.
    - Current default is 'auto' which uses 1 / n_features
    - if gamma='scale', then it uses 1 / (n_features * X.var())
    - The current default of gamma, ‘auto’, will change to ‘scale’ in version 0.22.

Applying SVM to XOR Data Set

- from sklearn.svm import SVC
- import numpy as np
- np.random.seed(0) # Set randomization seed

- features = np.random.randn(200, 2) # Generate two features

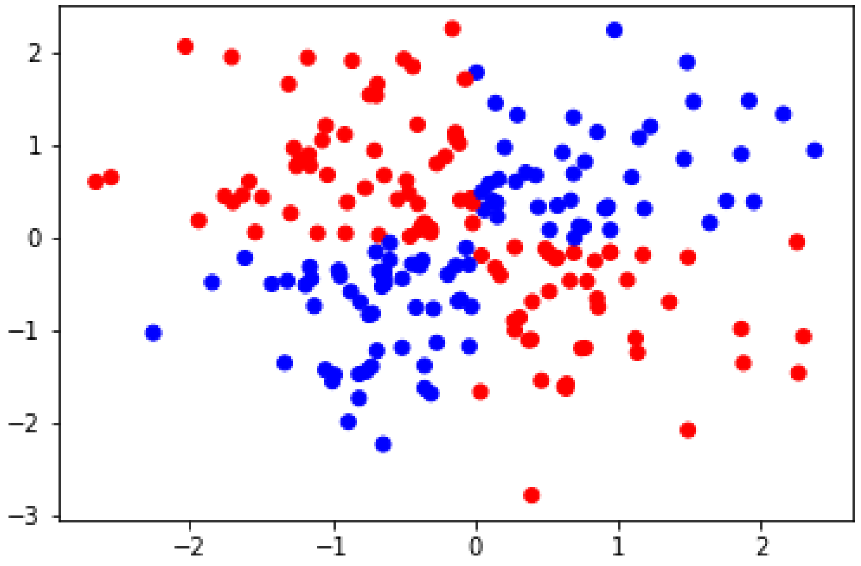
- # Use a XOR to generate linearly inseparable classes
- target_xor = np.logical_xor(features[:, 0] > 0, features[:, 1] > 0)
- target = np.where(target_xor, 0, 1)

- # Create a support vector machine with a radial basis function kernel
- svc = SVC(kernel="rbf", random_state=0, gamma=1, C=1)

- # Train the classifier
- model = svc.fit(features, target)

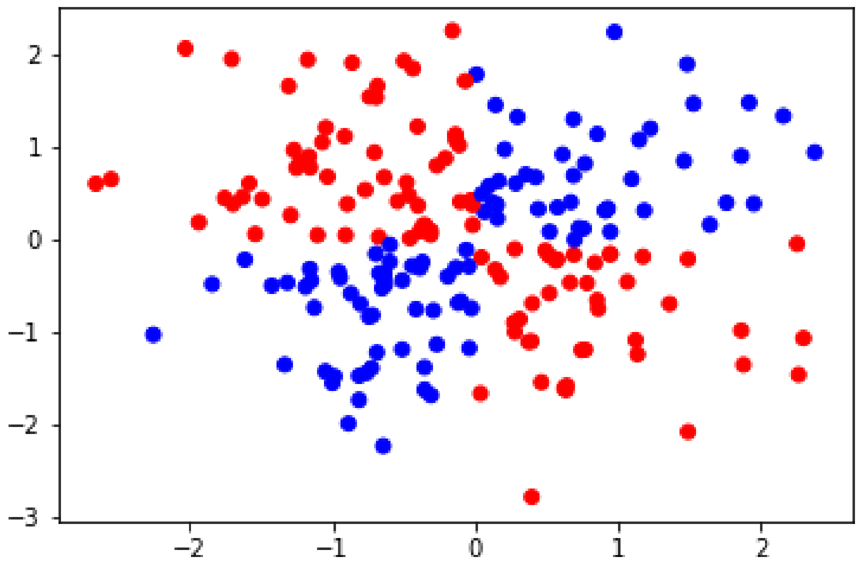
Visualize the Data Set

- # Plot the data set
- color = ['red' if c == 0 else 'blue' for c in target]
- plt.scatter(features[:,0], features[:,1], c = color)
- plt.show()



Classify New Observations

- model.predict([[-1, 1], [1, 1]])
  # array([0, 1])



Hyperparameter Tuning with Grid Search

- Finding good values for hyperparameters is tricky
- Scikit-learn provides techniques to help
- A commonly used technique is grid search, which tries all possible combinations of the values of interest
- Example: for a SVM with RBF Kernel, we want to find good values for gamma and C. We want to use grid search to try the values 0.001, 0.01, 0.1, 1, 10, and 100 for the parameter C, and the same for gamma.
- There are 36 combinations of parameters in total

Grid Search

- Grid search with cross-validation is a commonly used method to tune hyperparameters
- Scikit-learn provides the GridSearchCV class, which implements it in the form of an estimator
- You need to specify the parameters to search over as a dictionary
  - keys are the names of parameters to adjust
  - values are the parameter settings we want to try out

    ```
    param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
                  'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
    ```
- GridSearchCV will then perform all the necessary model fits

Grid Search

- **from sklearn.model_selection import** GridSearchCV
- **from sklearn.svm import** SVC
- grid_search = GridSearchCV(SVC(), param_grid, cv=5)

Grid Search

- To avoid overfitting, the cross-validation used for parameter tuning should not include the test set

  ```
  X_train, X_test, y_train, y_test =
  train_test_split(iris.data, iris.target, random_state=0)
  ```

Grid Search

- The grid_search created behaves just like a classifier
  - it provides the methods fit, predict, and score
  - when we call fit, it will run cross-validation for each combination of parameters we specified in param_grid:

```
grid_search.fit(X_train, y_train)
```

- Fitting the GridSearchCV object searches for the best parameters and fits a new model on the whole training dataset with the best tuned parameters

- We can evaluate model on the test set:
- print("Test set score: %.2f" % grid_search.score(X_test, y_test))
- # Test set score: 0.97

Grid Search

- Best parameters are stored in the best_params_ attribute
- Best cross-validation accuracy is stored in the best_score_ attribute
- print("Best parameters:", grid_search.best_params_)
  # Best parameters: {'C': 100, 'gamma': 0.01}
- print("Best cross-validation score: %.2f" %
  grid_search.best_score_)
  # Best cross-validation score: 0.97

Grid Search

- We can access the model with the best parameters trained on the whole training set using the best_estimator_ attribute:

```
print("Best estimator:\n", grid_search.best_estimator_)
# Best estimator:
  SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
   decision_function_shape='ovr', degree=3, gamma=0.01, kernel='rbf',
   max_iter=-1, probability=False, random_state=None, shrinking=True,
   tol=0.001, verbose=False)
```

- See notebook: Support Vector Machines.ipynb