# Support Vector Machines (SVM) Introduction

# Support Vector Machines

- Popular machine learning algorithm
- Powerful and versatile:
  - Linear and non-linear classification
  - Regression
  - Outlier detection
- Uses kernels to transform features
- Good for classification of complex small to medium sized datasets

# Benefits

- Good for data with lots of features
- Remains solvable even when number of features is greater than number of samples
- Memory efficient since it only uses a subset of data called support vectors in the decision function
- Different kernels can be used to define decision function, and you can even specify custom ones

# Disadvantages/Challenges

- Sensitive to outliers
- Features must be scaled
- Overfitting can happen, especially when you have more features than samples
  - Pick your kernel function properly
  - Use regularization terms
- SVMs do not provide probability estimates

Support Vector Machines (SVM) Introduction

# The End

# Linear SVM Classification

# Separating Two Classes

# SVM Decision Boundary

# Margin Approaches

- There are two approaches to how we treat data on or near the road:
  - Hard margin classification
  - Soft margin classification
- Hard margin classification
  - Does not allow any points on the road
  - Insists different classes are on different sides of the road

# Hard Margin Issues

- The assumptions of the hard margin solution lead to some issues:
  - The data has to be linearly separable
  - Outliers near the road can make the margins of the hard margin solution small
  - Outliers near the other class can make it so the the hard margin solution is not solvable at all

# Soft Margin Classification

- Soft margin classification solves for the widest street but allows for some margin violations
- Margin violations
  - Instances on the road
  - Or even on the wrong side of the street
- The C hyperparameter in the scikit-learn implementation of SVM determines the amount of margin violations allowed

# SVM Classifiers in Scikit-learn

- LinearSVC
  - Based on the liblinear library
  - Performs better with a large # of samples
  - Has more penalty and loss functions built-in
- SVC with a linear kernel
  - Based on the libsvm library
- NuSVC
  - Based on libsvm
  - Has a Nu parameter to control the # of support vectors
- SGD class with loss=hinge and alpha = 1/(m+C)
  - Good for online and out-of-core training

# Linear SVC Implmentation

- Load your imports

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
```

- Load the iris dataset

```
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris-Virginica
```

# Implement an SVM Pipeline

- Build a pipeline to scale your features and then train a linear svm classifier

```
svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge", random_state=42)),
    ])

svm_clf.fit(X, y)
```

- This returns a trained model

```
Pipeline(memory=None,
         steps=[('scaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('linear_svc',
                 LinearSVC(C=1, class_weight=None, dual=True,
                           fit_intercept=True, intercept_scaling=1,
                           loss='hinge', max_iter=1000, multi_class='ovr',
                           penalty='l2', random_state=42, tol=0.0001,
                           verbose=0))],
         verbose=False)
```

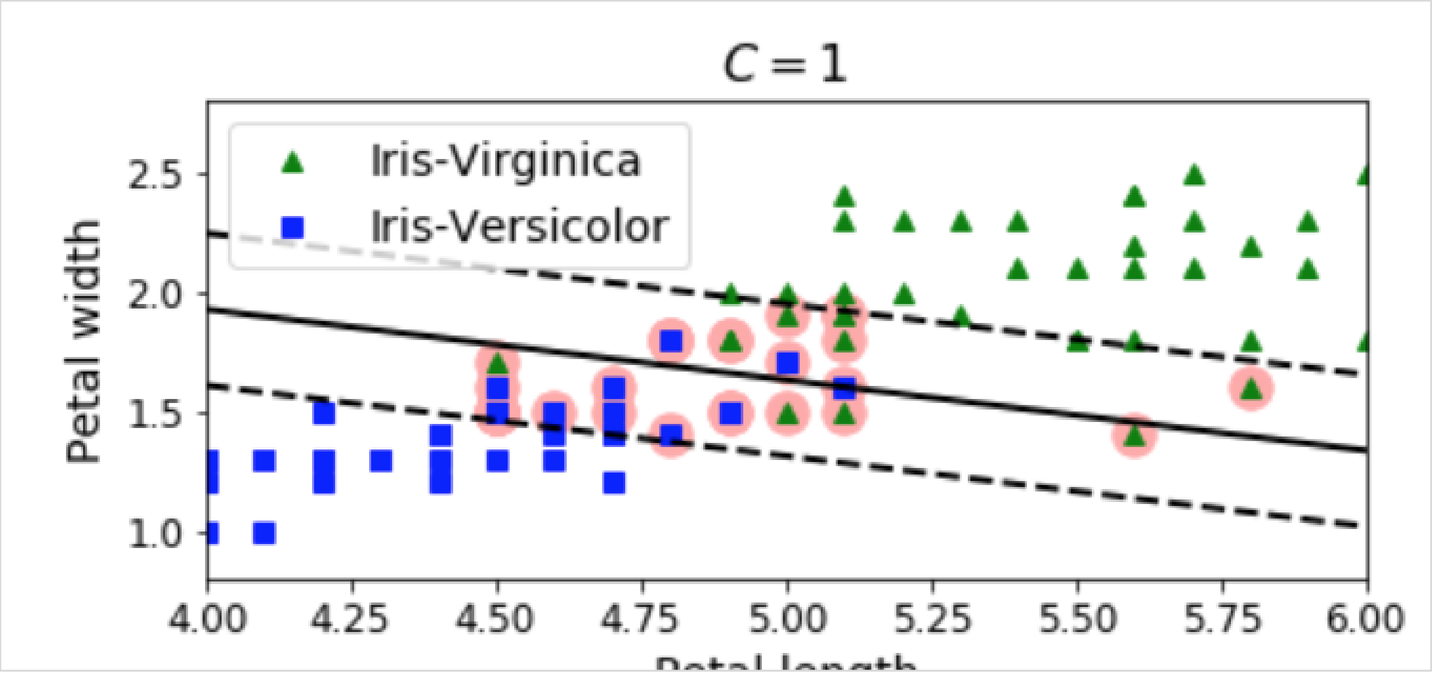
# Predict Using Your Model

```
svm_clf.predict([[5.5, 1.7]])
```

```
array([1.])
```

# What Does Your Decision Boundary Look Like?

- Here is the resulting decision boundary



# Exercise

- Implement an SVC with linear kernel (kernel=linear and C=1)
- Implement an SGDClassifier

Linear SVM Classification

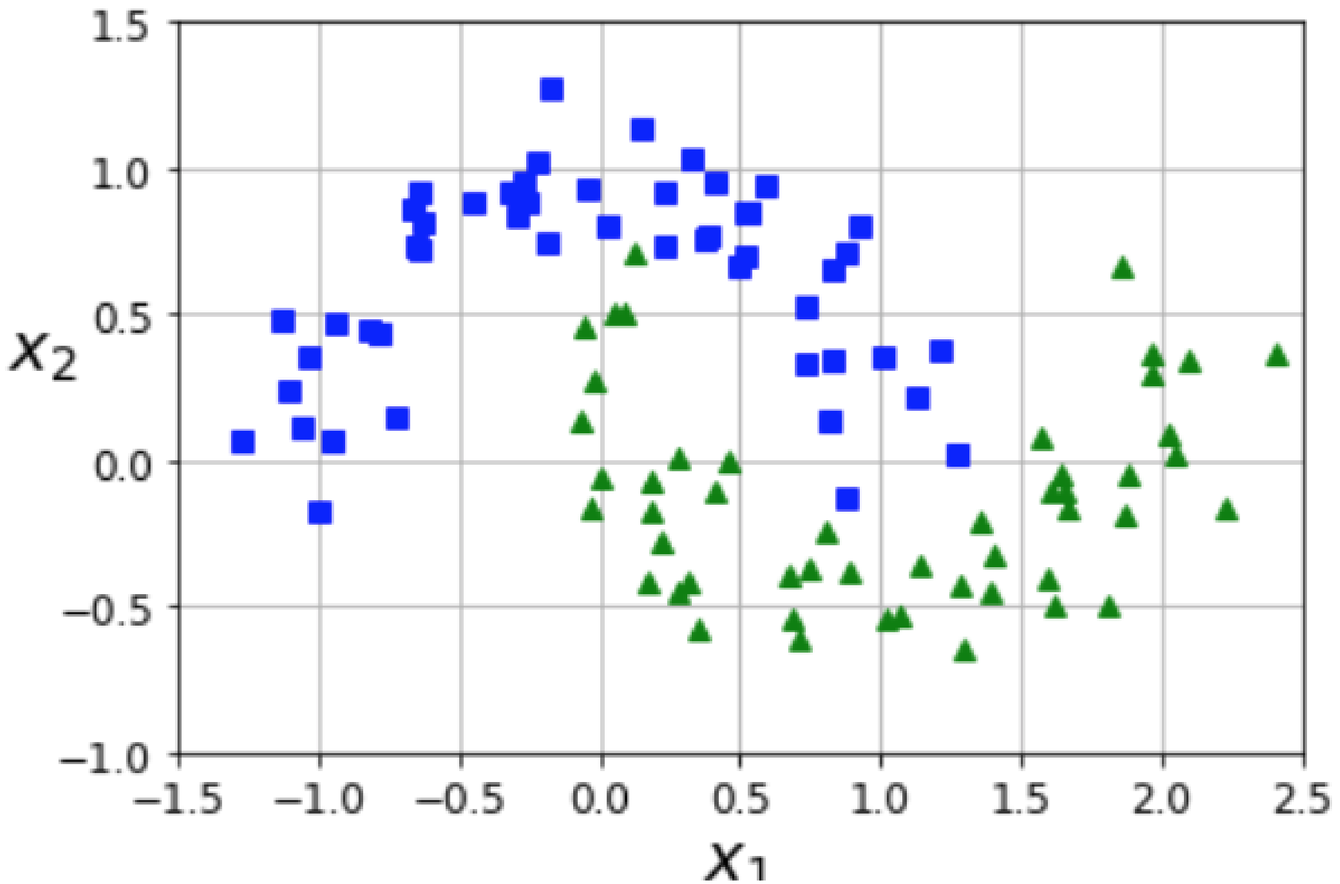
# The End

# Non-Linear SVM Classification

# Non-Linear SVM Classification

- What if your data is not linearly separable?
- Recall we solved this problem before by adding polynomial features
- This transformation into higher dimensional space can result in a linear separable solution

# Make Moons Example



# Implementation in scikit-learn

- In scikit-learn you can implement a pipeline:
- Start with a polynomial features transformer
- Feed that to a standard scaler
- and finally the last step of your pipeline is calling a LinearSVC with a hinge loss function and a C value of 10

# Example Code

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge", random_state=42))
    ])

polynomial_svm_clf.fit(X, y)
```

# Warning (Danger Will Robinson)

- Implementing the pipeline in this way can get very expensive as you compute and add new polynomial features to your input set
- This can make the model too slow
- But don’t worry…

# Solving the Transformation Conundrum

- How do you solve the expense of adding more features?
- Scikit-learn’s SVC class uses kernels to define the decision boundary
- There is a mathematical technique called the kernel trick
- It gives the same result as adding more features without actually producing them

# Implementing a Poly Kernel

- Instead of use a polynomial feature transformer in your pipeline feed the scaled data into an SVC estimator
- The hyperparameters for you SVC should include:
  - kernel=“poly”
  - degree=3
  - coef0=1  #controls how much influence high-degree polynomials have versus low-degree

# Example Code

```
from sklearn.svm import SVC

poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

# Similarity Functions

- Another way to transform data to make it separable in higher dimensions is by using a similarity function
- A similarity function maps each instance to a landmark
- A common function used for similarity is the gaussian radial basis function (RBF)

# RBF Function

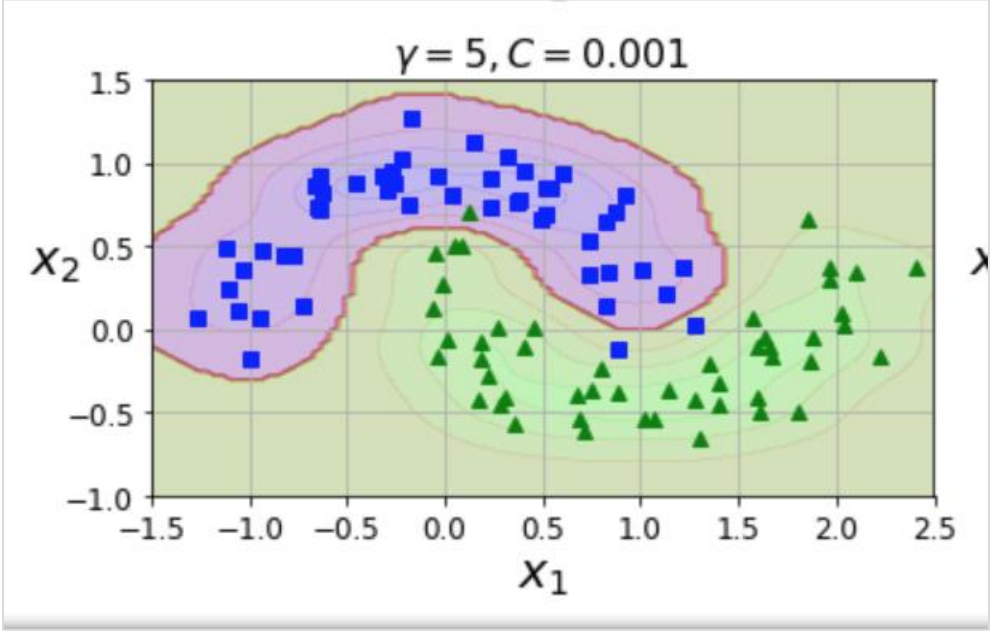
- An RBF computes a bell-shaped curve
  - The curve starts at 0 for points far from the landmark
  - Is equal to 1 at the landmark
- You use the calculated similarity scores for this transformation to plot a linear separable solution
- Again, actually producing these transformation could get expensive, but not to worry…

# SVC RBF Kernel

- RBF has been implemented as a kernel in SVC
  - It can utilize the kernel trick
- Change your pipeline once again:
  - Kernel=”rbf” and gamma=5
- Gamma affects how skinny or wide the bell-shaped curve is
  - High gamma makes the bell curve narrower
    - Individual instances have more effect
    - The resultant model wiggles around individual points
  - Lower gamma make the bell curve wider
    - Individual instances have less effect
    - Thus the model is smoother

# Example Code and Output

```
rbf_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ])
rbf_kernel_svm_clf.fit(X, y)
```



Non-Linear SVM Classification

# The End

# SVM Regression

# SVM Regression

- For SVM regression, we reverse the objective :
  - We try to fit as many instances as possible on the street
  - Margin violations are defined as points off the street
- A new parameter, epsilon, defines the width of the margin

# Epsilon Hyperparameter

- Recall that adding data points in SVM Classification off the street don’t affect the calculation for the decision boundary
- Similarly in an SVM regressor, adding points on the street don’t affect the decision boundary
- If you set epsilon to a large value, more data will fall on the street
- Small epsilon values will shrink your margin and result in more violations

# Scikit-learn SVR Classes

- Scikit-learn has a linearSVR class
  - Uses the same library as linearSVC
  - Is good for large datasets (scales linearly)
  - Implements the epsilon parameter
- For non-linear problems use the SVR class
  - Uses the same library as SVC
  - Supports the kernel trick
  - Implements epsilon
  - Can slow down when dataset gets large

# Linear SVR

- Plot some noisy linear data

```
np.random.seed(42)
m = 50
X = 2 * np.random.rand(m, 1)
y = (4 + 3 * X + np.random.randn(m, 1)).ravel()
```

- Train a linearSVR Regressor

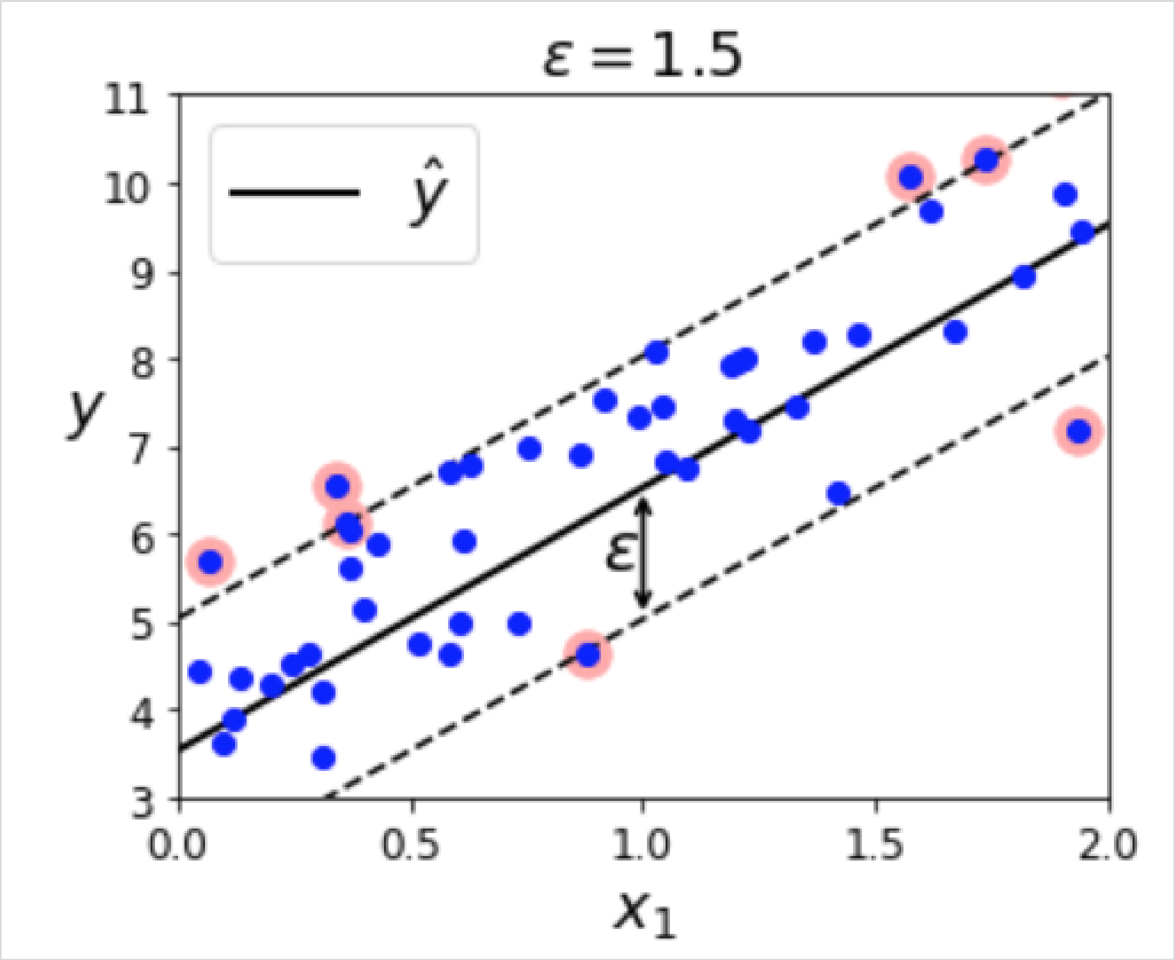
```
from sklearn.svm import LinearSVR

svm_reg = LinearSVR(epsilon=1.5, random_state=42)
svm_reg
svm_reg.fit(X, y)
```

# Linear SVR

- Plo

- Tra



# Solving a Polynomial

- Plot some noisy quadratic data

```
np.random.seed(42)
m = 100
X = 2 * np.random.rand(m, 1) - 1
y = (0.2 + 0.1 * X + 0.5 * X**2 + np.random.randn(m, 1)/10).ravel()
```

- Train an SVR with a polynomial kernel

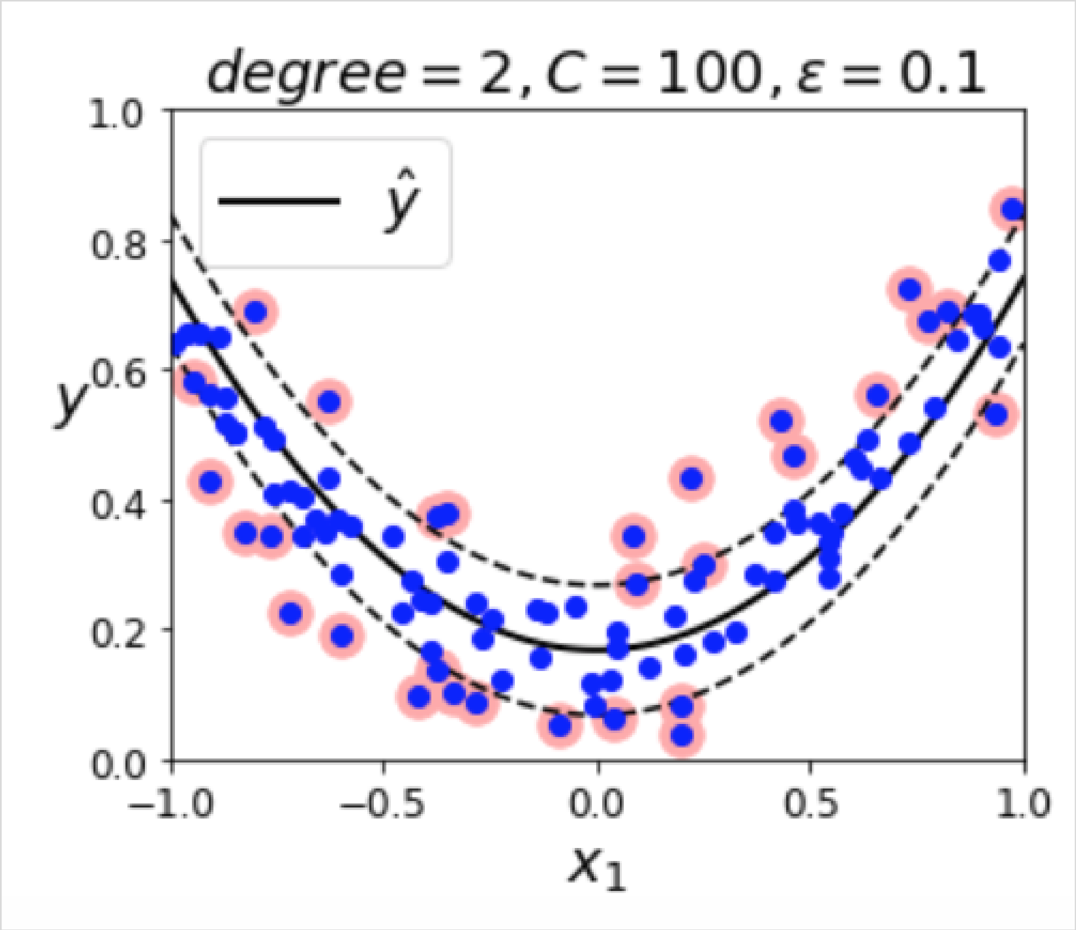
```
from sklearn.svm import SVR

svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1, gamma="auto")
svm_poly_reg.fit(X, y)
```

# Solving a Polynomial

- Plot s

```
np.ranc
m = 10(
X = 2 *
y = (0.                                    10).ravel()
```




- Train                                    ernel

```
from skl

svm_poly                                   gamma="auto")
svm_poly
```

# Exercise

- Implement a Support Vector Regressor with a linear kernel

# The End

# SVM Training

# SVM Prediction

- Predict the class of a new instance X by computing:

$$w^T x + b = w_1 x_{1+} w_2 x_2 + \ldots + w_n x_n$$

- If the above calculation is positive the prediction belongs to the positive class
- If the above calculation is 0 or negative the prediction belongs to the negative class

# Decision Function and Boundary

- The decision boundary in a two-dimensional space will be a line where the decision function is 0
- Our margins are defined where the decision function equals plus or minus one
- Training a linear SVM classifier means
  - finding the values of w and b that make the margin as wide as possible
  - While avoiding margin violations or limiting them

# Training Objective

- The slope of the decision function is equal to the norm of the weight vector
- If we lower slope the points where the decision function is equal to +-1 are going to be twice as far from the decision boundary
- The smaller the weight vector w, the larger the margin
- So if we want a large margin we have to minimize the norm of the weight vector

# Hard Margin Objective

- In plain terms we want to:
  - Minimize the square of the weight vector norm
  - Add a constraint to force the points to be greater than one or less than minus one
- In math terms
  - $\min\limits_{w,b} \frac{1}{2} w^t w$
  - Subject to $t^{(i)}\left(w^T x^{(i)} + b\right) \geq 1 \; for \; i = 1,2, \ldots, m$
  - Where t(i) = 1 for a positive training instance and -1 for a negative training instance

# Soft Margin Objective

- Soft margin adds a slack variable
  - Measures the distance of the point on the street to its marginal hyperplane
  - We have to add this to our objective
  - How do we do that?  (We’ve seen it before)
  - Something related to Alpha…
  - Got it yet?
  - Yes, it is the C hyperparameter

# Soft Margin Equation

- In plain terms:
  - We want to minimize the square of the weight vector
  - Add the sum of the slack variables and multiply that times C
  - A high C will force the slack variables to be smaller, allowing less margin violations
  - The above is constrained by
    - The equation being greater than 1,
    - Subtracted by the slack variable when the slack variable is greater than 0

# Soft Margin Mathematically

- Here is the mathematical formula for the softmax objective

$$\min_{w,b,\zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i$$

$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \zeta_i,$$

$$\zeta_i \geq 0, i = 1, \ldots, n$$

# Quadratic Programming

- Both the hard margin and soft margin objectives are quadratic optimization problems with linear constraints
- This type of problem is very common in mathematics and the solution is a quadratic programming solver
- All we need to do is present our objective to a QP solver in the correct way to come up with the minimum values

# General QP Solver

- The following is a general problem formulation for a quadratic programming problem:

$$\underset{\mathbf{p}}{\text{Minimize}} \quad \frac{1}{2}\mathbf{p}^T \cdot \mathbf{H} \cdot \mathbf{p} \quad + \quad \mathbf{f}^T \cdot \mathbf{p}$$

$$\text{subject to} \quad \mathbf{A} \cdot \mathbf{p} \le \mathbf{b}$$

$$\text{where} \quad \begin{cases} \mathbf{p} & \text{is an } n_p\text{-dimensional vector } (n_p = \text{number of parameters}), \\ \mathbf{H} & \text{is an } n_p \times n_p \text{ matrix}, \\ \mathbf{f} & \text{is an } n_p\text{-dimensional vector}, \\ \mathbf{A} & \text{is an } n_c \times n_p \text{ matrix } (n_c = \text{number of constraints}), \\ \mathbf{b} & \text{is an } n_c\text{-dimensional vector.} \end{cases}$$

# Solving the Hard Margin Problem

- To solve hard margin objective you set the QP parameters as follows:
  - $n_{p\,=}$ n+1 where n is the number of features
    - The plus one is for the bias term
  - $n_c$ = m , where n is the number of features
  - H is the $n_p$x $n_p$ identity matrix, with a zero in the top-left cell to ignore the bias
  - f = 0, an $n_p$ dimensional vector full of 0s.
  - b = -1, an $n_p$ dimensional vector full of -1s
  - $a^{(i)} = -t^{(i)} X^{(i)}$ which adds an extra bias feature

# The Dual Problem and the Kernel Trick

- The QP solution to the SVM objective we just covered is called the primal problem
- Every primal problem has a similar but different problem called its dual problem
- It turns out, for the SVM objective, both the primary and dual problem have the same solution

# The Linear SVM Dual Problem

- Solve for alpha using a QP

$$\underset{\alpha}{\text{minimize}} \ \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^{m} \alpha^{(i)}$$

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

# Then Solve for w^hat and b^hat

- Use the following equation to minimize the primal solution:

$$\widehat{\mathbf{w}} = \sum_{i=1}^{m} \widehat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^{m} \left(1 - t^{(i)}\left(\widehat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)}\right)\right)$$

# What Is the Kernel Trick?

- Notice the highlighted portion of the dual problem:
- These alpha vectors contain the transformed data that you present to your SVM classifier for training

$$\underset{\alpha}{\text{minimize}} \ \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \boxed{\alpha^{(i)} \alpha^{(j)}} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^{m} \alpha^{(i)}$$

$$\text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

# Using A Combination of the Original Data

- It turns out you can use a dot product of the original data and the result will be the same
- You don't need to do the transformations
- Mercer's theorem describes the conditions required for a function to use it as a kernel
- For example, K must be continuous and symmetric
- What you want is $K(a,b) = \phi(a)^T\phi(b)$

# Online SVM Classification

- One solution to an SVM for online learning is to use the SGDClassifier
- SGD does converge more slowly than QP solutions

$$J(\mathbf{w}, b) = \boxed{\frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}} + \boxed{C}\sum_{i=1}^{m} \boxed{max\left(0, 1 - t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right)\right)}$$

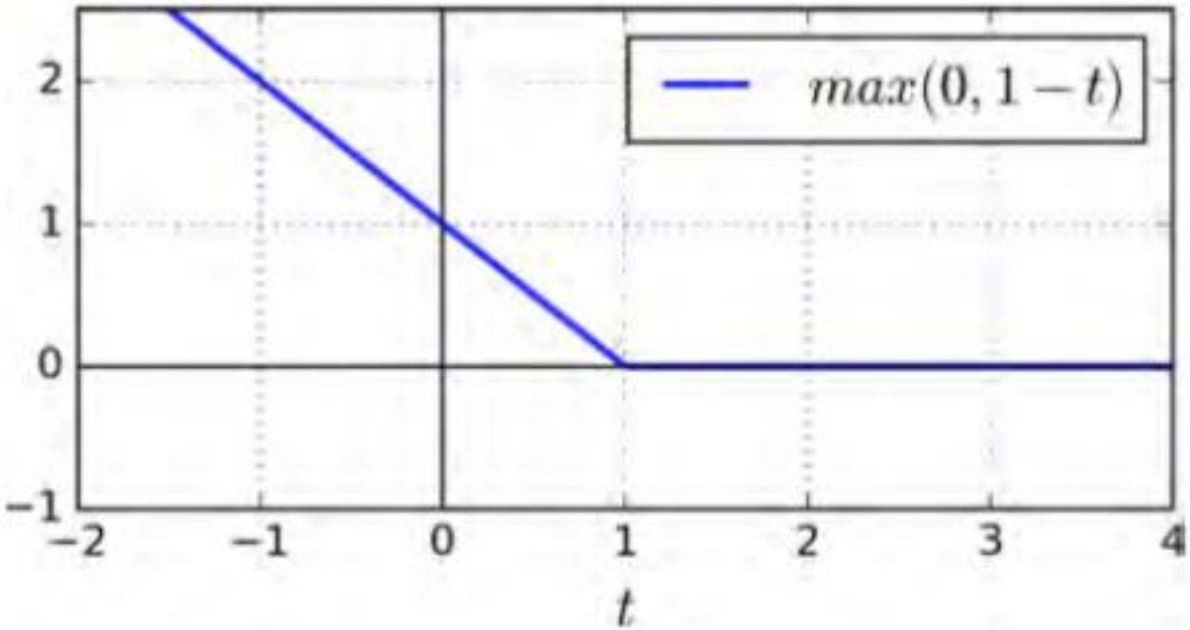
*We want small weights*

C=0 wide margins, lots of violations
C=∞ narrow margins, less violations

*0 if is it off the street*
*Or measure distance from correct side*

# Hinge loss

- Max(0,1-$t^{(i)}$) is called hinge loss and the graph of it looks like this:



# The End