Support Vector Machines

In their simplest form, SVMs are **linear classifiers** that do **risk minimization**.

$$R(\Theta) \leq J(\Theta) = R_{emp}(\Theta) + \sqrt{\frac{h \times (log(\frac{2N}{h}) + 1) - \log(\frac{\eta}{4})}{N}} \tag{1}$$

h: capacity of a clasifier
N: number of training points

- Vapnik and Chernovenkis proved that with probability $1-\eta$ previous equation holds true.

- Vapnik and Chernovenkis proved that with probability $1-\eta$ previous equation holds true.

- SVM algorithm minimizes both h and empirical risk at the same time by increasing seperation margin (less flexibility)

- Vapnik and Chernovenkis proved that with probability $1-\eta$ previous equation holds true.
- SVM algorithm minimizes both h and empirical risk at the same time by increasing seperation margin (less flexibility)

- Let's derive the equations

Derivation

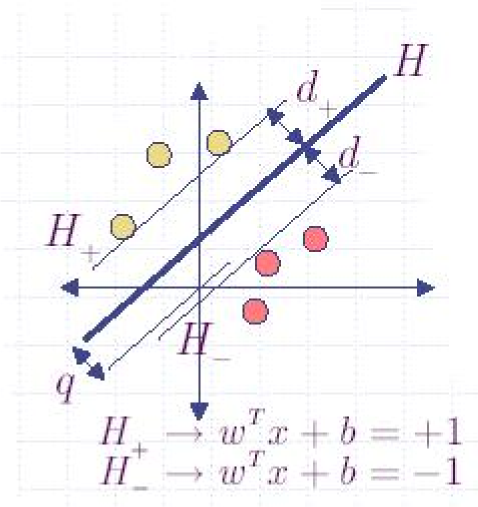

Figure 1:

Decision plane: $w^T x + b = 0$
Let's define $q = \min_x \|x - 0\|$

- We will later use the formula for q on H^+ and H^-.
- For H: $q = \min_x \|x - 0\|$ subject to $w^T x + b = 0$
- Lagrange: $min_x \frac{1}{2}\|x - 0\|^2 + \lambda(w^T x + b)$
- Take gradient $(\frac{\partial}{\partial x})$ set to 0
- After some algebra: $q = \frac{|b|}{||w||}$
- Define:
  - $H^+ = w^T x + b = +1$
  - $H^- = w^T x + b = -1$
- This is without loss of generality; We can still adjust b & w
- Calculate q^+ and q^-
  - $q^+ = \frac{|b-1|}{||w||}$
  - $q^- = \frac{|-b-1|}{||w||}$
- The margin then is
  - $m = q^+ + q^- = \frac{|b-1-b-1|}{||w||} = \frac{|-2|}{||w||} = \frac{2}{||w||}$

For maximal margin, increase m (maximize $\frac{2}{||w||}$) or minimize $||w||$!

Constraints

We want points classified so that + and - points are in the correct side of the hyperplanes;

$w^T x + b \geq +1, \forall y_i = +1$
$w^T x + b \leq -1, \forall y_i = -1$

Combine the two

$$y_i(w^T x + b) - 1 \geq 0 \tag{2}$$

Putting it all together

$$\min \frac{1}{2}||w||^2 \text{ subject to } y_i(w^T x_i + b) - 1 \geq 0 \tag{3}$$

This is a quadratic program!

qp

- Python language has cvxopt package
- Matlab Optimization Toolbox has qp() function.

- Or Steve Gunn's SVM Toolbox has another qp written in C
- SVMLight has its own qp
- qp functions usually expect a problem in $\frac{1}{2}x^T P x + q^T x$ format
- We can massage previous equation to fit the equation above

Dual

- For SVM purposes, working with the dual is easier.
- Form the Lagrange (again), take derivative, set equal to zero
- This gives us the KKT point

$$
\begin{aligned}
L_p &= \frac{1}{2}||w||^2 - \sum_i \alpha_i (y_i (w^T x_i + b) - 1) && (4) \quad \text{(eq:primal)} \\
\frac{\partial}{\partial w} L_p &= w - \sum_i \alpha_i y_i x_i = 0 \\
w &= \sum_i \alpha_i y_i x_i && (5) \quad \text{(eq:wdual)} \\
\frac{\partial}{\partial b} L_p &= -\sum_i \alpha_i y_i = 0 && (6) \quad \text{(eq:cdual)}
\end{aligned}
$$

Plugging equation eq:wdual ?? and eq:cdual ?? into primal equation eq:primal ??:

$$
\text{Maximize } L_D = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^T x_j \qquad (7) \quad \text{(eq:svm)}
$$

constraints

$$
\begin{aligned}
\sum_i \alpha_i y_i = 0 \\
\alpha_i \geq 0
\end{aligned}
$$

qp

- qp() again! But now the variable(s) we are solving for are α_i's, not x's.
- Massage eq:svm ?? into $\frac{1}{2}x^T P x + q^T x$ form
- This can be achieved by setting $P_{i,j}$ to be $-y_i y_j x_i^T x_j$
- Call qp
- The solution is a list of α's

Calculating b

- Due to KKT condition, for each nonzero α_i, the corresponding constraint in the primal problem is tight (an equality)
- Then for each non-zero α_i, calculate b using $w^T x_i + b = y_i$.
- Each b from non-zero α_i will be approximately equal to other b's. It is numerically safer to average all b's for final b.

Classifier Done

For each new point x, we can use $sign(x^T w + b)$ as our classifier. The result, -1 or $+1$ will tell us which class this new point belongs to.

Sample Output

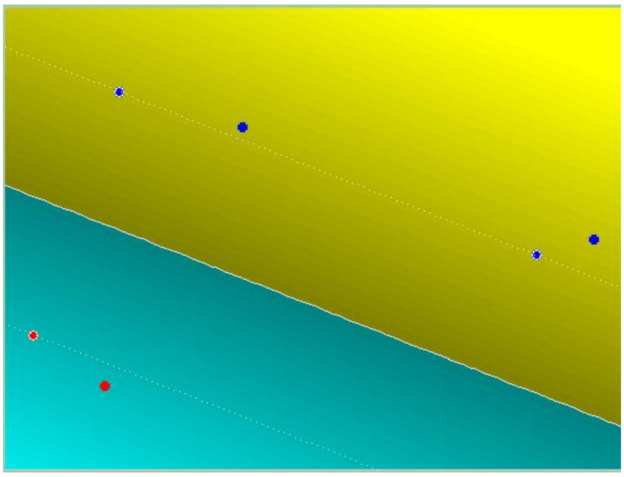

Figure 2:

Kernels

- We talked about linear boundaries so far
- SVMs can also form non-linear boundaries
- Simple: Just preprocess input data with a basis function into higher dimensions
- Rest of the algorithm is unchanged

Nonlinear Kernel

Slack

- Sometimes the problem might be inseperable

- A few points might throw off the classifier
- We can introduce "slack" into a classifier
- For example, allow data to fall on the wrong side with $w^T + b \geq -0.03$ for $y_i = +1$
- But we don't want too many of such points, hence penalize the "quantity" of suck slack points

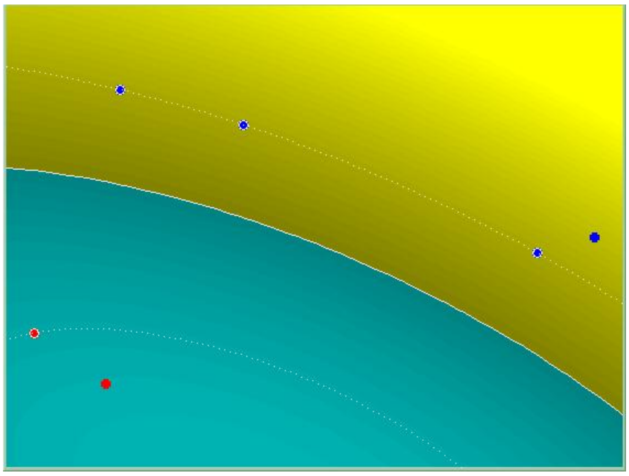

Figure 3:

```
import numpy as np
from numpy import linalg
import cvxopt
import cvxopt.solvers

def svm(X, y):
    n_samples, n_features = X.shape

    # Gram matrix
    K = np.zeros((n_samples, n_samples))
    for i in range(n_samples):
        for j in range(n_samples):
            K[i,j] = np.dot(X[i], X[j])

    P = cvxopt.matrix(np.outer(y,y) * K)
    q = cvxopt.matrix(np.ones(n_samples) * -1)
    A = cvxopt.matrix(y, (1,n_samples))
    b = cvxopt.matrix(0.0)

    G = cvxopt.matrix(np.diag(np.ones(n_samples) * -1))
    h = cvxopt.matrix(np.zeros(n_samples))

    # solve QP problem
    solution = cvxopt.solvers.qp(P, q, G, h, A, b)

    print solution

    # Lagrange multipliers
    a = np.ravel(solution['x'])

    print "a", a

    # Support vectors have non zero lagrange multipliers
    ssv = a > 1e-5
    ind = np.arange(len(a))[ssv]
    a = a[ssv]
    sv = X[ssv]
    sv_y = y[ssv]
    print "%d support vectors out of %d points" % (len(a), n_samples)
    print "sv", sv
    print "sv_y", sv_y

    # Intercept
    b = 0
    for n in range(len(a)):
        b += sv_y[n]
        b -= np.sum(a * sv_y * K[ind[n],ssv])
    b /= len(a)

    # Weight vector
    w = np.zeros(n_features)
    for n in range(len(a)):
        w += a[n] * sv_y[n] * sv[n]

    print "a", a
    return w, b, sv_y, sv, a

if __name__ == "__main__":

    def test():
        X = np.array([[3.,3.],[4.,4.],[7.,7.],[8.,8.]])
        y = np.array([1.,1.,-1.,-1.])
        w, b, sv_y, sv, a = svm(X, y)
        print "w", w
        print "b", b
        print 'test points'
        print np.dot([2.,2.], w) + b # > 1
        print np.dot([9.,9.], w) + b # < -1

    test()
```

Note: We are maximizing the dual L_d, but we are still calling the minimizer `qp()` function. Therefore the q's, which represent the summation of all α's are negated as seen above in `np.ones(n_samples) * -1`. The quadratic part already has the negated statement $-\frac{1}{2}$ in the beginning, so the rest of does not have to change.

References

http://www.mblondel.org/journal/2010/09/19/support-vector-machines-in-python

Jebara, T., Machine Learning Lecture, Columbia University