CSE 5523: Lecture Notes 19
Support Vector Machines

Contents

19.1 Support Vector Machines 1
19.2 Sample code 3
19.3 Kernel functions 4
19.4 Sample 'weightless' code 5
19.5 Slack variables 6

We can define complex separators with a closed-form-ish solution (but it doesn't scale well).

19.1 Support Vector Machines

We want to find a line $\mathbf{w}, b$ that separates the data by 1 'unit' (which is just scaled into $\mathbf{w}$ and b):

$$\forall_{\langle 1,\mathbf{x}_n\rangle\in\mathcal{D}}\ \mathbf{w}^\top\mathbf{x}_n + b \geq 1 \text{ and } \forall_{\langle -1,\mathbf{x}_n\rangle\in\mathcal{D}}\ \mathbf{w}^\top\mathbf{x}_n + b \leq -1$$

$$\forall_{\langle 1,\mathbf{x}_n\rangle\in\mathcal{D}}\ \mathbf{w}^\top\mathbf{x}_n + b \geq 1 \text{ and } \forall_{\langle -1,\mathbf{x}_n\rangle\in\mathcal{D}}\ -(\mathbf{w}^\top\mathbf{x}_n + b) \geq 1 \qquad \text{additive inverse in inequality}$$

$$\forall_{\langle y_n,\mathbf{x}_n\rangle\in\mathcal{D}}\ y_n\,(\mathbf{w}^\top\mathbf{x}_n + b) \geq 1 \qquad \text{group conjuncts}$$

And we want that separation 'unit' to be maximal (i.e. to have minimal rescaling by $\mathbf{w}$):

$$\underset{\mathbf{w}\text{ s.t. }\forall_{\langle y_n,\mathbf{x}_n\rangle\in\mathcal{D}}\ y_n(\mathbf{w}^\top\mathbf{x}_n+b)\geq 1}{\text{argmin}} \overbrace{\|\mathbf{w}\|_2}^{\text{distance by which to divide 'unit'}} = \underset{\mathbf{w}\text{ s.t. }\forall_{\langle y_n,\mathbf{x}_n\rangle\in\mathcal{D}}\ y_n(\mathbf{w}^\top\mathbf{x}_n+b)\geq 1}{\text{argmin}} \frac{1}{2}\|\mathbf{w}\|_2^2$$

We also want to be able to express this only in terms of our training data, to allow richer models.

We model separation as extra arbitrarily awful cost α_n for each example violating the constraint.

$$\underset{\mathbf{w},b\text{ s.t. }\forall_{\langle y_n,\mathbf{x}_n\rangle\in\mathcal{D}}\ y_n(\mathbf{w}^\top\mathbf{x}_n+b)\geq 1}{\text{argmin}} \frac{1}{2}\|\mathbf{w}\|_2^2 = \underset{\mathbf{w},b}{\text{argmin}} \max_{\alpha_1,\ldots,\alpha_{|\mathcal{D}|}\geq 0} \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_n \alpha_n \left(y_n\,(\mathbf{w}^\top\mathbf{x}_n + b) - 1\right) \qquad \text{constraint as cost}$$

$$= \underset{\mathbf{w},b}{\text{argmin}} \max_{\alpha_1,\ldots,\alpha_{|\mathcal{D}|}\geq 0} \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_n \alpha_n\, y_n\,(\mathbf{w}^\top\mathbf{x}_n + b) + \sum_n \alpha_n \qquad \text{distrib. axiom}$$

(Arbitrary awfulcy keeps us above the constraint when perpendicular to arbitrarily steep gradients.)

These awful α_n's are called **Lagrange multipliers**. The function with them in it is a **Lagrangian**.

Now we differentiate the Lagrangian to optimize $\mathbf{w}$ (slope of cost is zero at minimum $\mathbf{w}$):

$$0 = \frac{\partial}{\partial \mathbf{w}_{[v]}} \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_n \alpha_n\, y_n\,(\mathbf{w}^\top\mathbf{x}_n + b) + \sum_n \alpha_n$$

$$
\begin{aligned}
&= \frac{\partial}{\partial \mathbf{w}_{[v]}} \frac{1}{2} \sum_m (\mathbf{w}_{[m]})^2 - \sum_n \alpha_n y_n (\mathbf{w}^\top \mathbf{x}_n + b) + \sum_n \alpha_n && \text{def. of 2-norm} \\
&= \frac{\partial}{\partial \mathbf{w}_{[v]}} \frac{1}{2} \sum_m (\mathbf{w}_{[m]})^2 - \frac{\partial}{\partial \mathbf{w}_{[v]}} \sum_n \alpha_n y_n (\mathbf{w}^\top \mathbf{x}_n + b) + \frac{\partial}{\partial \mathbf{w}_{[v]}} \sum_n \alpha_n && \text{sum rule} \\
&= \sum_m \frac{\partial}{\partial \mathbf{w}_{[v]}} \frac{1}{2} (\mathbf{w}_{[v]})^2 - \sum_n \frac{\partial}{\partial \mathbf{w}_{[v]}} \alpha_n y_n (\mathbf{w}^\top \mathbf{x}_n + b) + \sum_n \frac{\partial}{\partial \mathbf{w}_{[v]}} \alpha_n && \text{sum rule} \\
&= \mathbf{w}_{[v]} - \sum_n \alpha_n y_n \frac{\partial}{\partial \mathbf{w}_{[v]}} (\mathbf{w}^\top \mathbf{x}_n + b) && \text{product rule} \\
&= \mathbf{w}_{[v]} - \sum_n \alpha_n y_n \left(\frac{\partial}{\partial \mathbf{w}_{[v]}} \mathbf{w}^\top \mathbf{x}_n + \frac{\partial}{\partial \mathbf{w}_{[v]}} b \right) && \text{sum rule} \\
&= \mathbf{w}_{[v]} - \sum_n \alpha_n y_n (\mathbf{x}_n)_{[v]} && \text{product rule} \\
\mathbf{0} &= \mathbf{w} - \sum_n \alpha_n y_n \mathbf{x}_n && \text{apply to all } v \\
\mathbf{w} &= \sum_n \alpha_n y_n \mathbf{x}_n && \text{subtract } \sum_n \alpha_n y_n \mathbf{x}_n
\end{aligned}
$$

and to optimize b (slope of cost is zero at minimum b):

$$
\begin{aligned}
0 &= \frac{\partial}{\partial b} \frac{1}{2} \|\mathbf{w}\|_2^2 - \sum_n \alpha_n y_n (\mathbf{w}^\top \mathbf{x}_n + b) + \sum_n \alpha_n \\
&= -\frac{\partial}{\partial b} \sum_n \alpha_n y_n (\mathbf{w}^\top \mathbf{x}_n + b) && \text{sum rule} \\
&= -\sum_n \frac{\partial}{\partial b} \alpha_n y_n (\mathbf{w}^\top \mathbf{x}_n + b) && \text{sum rule} \\
&= -\sum_n \alpha_n y_n \frac{\partial}{\partial b} (\mathbf{w}^\top \mathbf{x}_n + b) && \text{product rule} \\
&= -\sum_n \alpha_n y_n && \text{sum rule}
\end{aligned}
$$

This lets us reformulate the Lagrangian entirely in terms of our training data:

$$
\begin{aligned}
&\frac{1}{2} \|\mathbf{w}\|_2^2 - \sum_n \alpha_n y_n (\mathbf{w}^\top \mathbf{x}_n + b) + \sum_n \alpha_n \\
&\quad = \frac{1}{2} \mathbf{w}^\top \mathbf{w} - \sum_n \alpha_n y_n (\mathbf{w}^\top \mathbf{x}_n + b) + \sum_n \alpha_n && \text{def. of 2-norm} \\
&\quad = \frac{1}{2} \left(\sum_n \alpha_n y_n \mathbf{x}_n \right)^\top \left(\sum_m \alpha_m y_m \mathbf{x}_m \right) - \sum_n \alpha_n y_n \left(\left(\sum_m \alpha_m y_m \mathbf{x}_m \right)^\top \mathbf{x}_n + b \right) + \sum_n \alpha_n && \text{subst. opt. of } \mathbf{w} \\
&\quad = \frac{1}{2} \sum_{n,m} \alpha_n \alpha_m y_n y_m \mathbf{x}_n^\top \mathbf{x}_m - \sum_{n,m} \alpha_n \alpha_m y_n y_m \mathbf{x}_n^\top \mathbf{x}_m + b \sum_n \alpha_n y_n + \sum_n \alpha_n && \text{distributive axiom}
\end{aligned}
$$

$$= -\frac{1}{2} \sum_{n,m} \alpha_n \alpha_m y_n y_m \mathbf{x}_n^\top \mathbf{x}_m + b \sum_n \alpha_n y_n + \sum_n \alpha_n \qquad \text{add like terms}$$

$$= -\frac{1}{2} \sum_{n,m} \alpha_n \alpha_m y_n y_m \mathbf{x}_n^\top \mathbf{x}_m + \sum_n \alpha_n \qquad \text{apply opt. of } b$$

This is called the Lagrangian **dual**. It is expressed entirely in the space of N inputs.

We still have constraints, specifically that the α's be non-negative, but solvers exist for this.

These constraints fit the form of **quadratic programming** optimizers, so we use those to find α's.

The solver wants a matrix for our dual, indexed by n and m above, called a **Hessian**:

$$\mathbf{H} = \text{diag}(\mathbf{y})\, \mathbf{X}^\top \mathbf{X}\, \text{diag}(\mathbf{y})$$

The resulting α vector will be mostly zero with a few positive values, called **support vectors**.

Support vectors are those points closest to the separator, which serve to *define* the separator.

Once we have the optimum α's, we can plug them in to get weights $\mathbf{w}$, using the equation above:

$$\mathbf{w} = \sum_n \alpha_n y_n \mathbf{x}_n$$

Then we choose any item $\mathbf{x}_n$ on the support vector, say that with the highest Lagrangian value:

$$n = \underset{n}{\text{argmax}}\, \boldsymbol{\alpha}$$

and use it to define b:

$$b = y_n - \mathbf{w}^\top \mathbf{x}_n$$

19.2 Sample code

Sample SVM code using cvxopt solver:

```
import sys
import numpy
import pandas
import cvxopt
cvxopt.solvers.options['show_progress'] = False

YX = pandas.read_csv( sys.argv[1] )            ## read data
N = len(YX)

y = YX[YX.columns[0]].to_frame()               ## transform data
```

```
X = YX[YX.columns[1:]]

H = (numpy.diagflat(y.values) @ X @ X.T @ numpy.diagflat(y.values)).values    ## Hessian

a = numpy.array( cvxopt.solvers.qp( cvxopt.matrix( H, tc='d'          ),
                                    cvxopt.matrix( -numpy.ones((N,1)) ),
                                    cvxopt.matrix( -numpy.eye(N)      ),
                                    cvxopt.matrix( numpy.zeros(N)     ),
                                    cvxopt.matrix( y.T.values, tc='d' ),
                                    cvxopt.matrix( numpy.zeros(1)     ) )['x'] )

w = X.T @ (y*a)                     ## weights are points averaged by Lagrangians
n = numpy.argmax( a * y )           ## find a support vector x_n
b = (y.T)[n] - w.T @ (X.T)[n]       ## bias is difference between value and estimate of x_n

yhat = numpy.sign( X @ w + numpy.ones((N,1)) * b.values )    ## estimate including bias

print( yhat )                                                ## print estimate
```

Run on simple dataset with three support vectors (the last three points):

```
y,x1,x2
1,1,5
1,2,4
-1,2,2
-1,4,4
```

It correctly produces a separator:

```
     y
0  1.0
1  1.0
2 -1.0
3 -1.0
```

19.3 Kernel functions

We can also make 'weightless' SVM's, with no weight vector, to allow wigglier separators:

$$b = y_n - (\boldsymbol{\alpha} \odot \mathbf{y})^\top \overbrace{\mathbf{X}\,\mathbf{x}_n}^{\text{inner prod.}} \qquad \text{where } n = \operatorname*{argmax}_n \boldsymbol{\alpha}$$

$$\hat{y} = \operatorname{sign}\Big((\boldsymbol{\alpha} \odot \mathbf{y})^\top \underbrace{\mathbf{X}\,\mathbf{x}}_{\text{inner prod.}} + b \Big)$$

Inner products in these models can be replaced with functions on vectors, called **kernel functions**.

E.g. the **radial basis function (RBF) kernel** uses distances to support vectors as coordinates:

$$\mathsf{K}_{\text{RBF}}(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|_2^2}{2\sigma^2} \right)$$

So the inner products in the Hessian, bias and expectation equations can be replaced with:

$$\left(\sum_m \delta_m \mathsf{K}(\mathbf{x}_m, \mathbf{x}') \right)$$

Also, since non-support vectors have zero α, they can be ignored here and in $\mathbf{y}$ to save time.

19.4 Sample 'weightless' code

Sample SVM code with no weight vector (NOTE: this does not use the kernel function):

```
import sys
import numpy
import pandas
import cvxopt
cvxopt.solvers.options['show_progress'] = False

YX = pandas.read_csv( sys.argv[1] )           ## read data
N = len(YX)

y = YX[YX.columns[0]].to_frame()              ## transform data
X = YX[YX.columns[1:]]

H = (numpy.diagflat(y.values) @ X @ X.T @ numpy.diagflat(y.values)).values    ## Hessian

a = numpy.array( cvxopt.solvers.qp( cvxopt.matrix( H, tc='d'          ),
                                    cvxopt.matrix( -numpy.ones((N,1)) ),
                                    cvxopt.matrix( -numpy.eye(N)      ),
                                    cvxopt.matrix( numpy.zeros(N)     ),
                                    cvxopt.matrix( y.T.values, tc='d' ),
                                    cvxopt.matrix( numpy.zeros(1)     ) )['x'] )

n = numpy.argmax( a * y )              ## find a support vector x_n
b = (y.T)[n] - (a*y).T @ X @ (X.T)[n]  ## bias is difference between value and estimate of x_n

yhat = numpy.sign( X @ X.T @ (a*y) + numpy.ones((N,1)) * b.values )   ## estimate

print( yhat )                                                         ## print estimate
```

On the same input:

```
y,x1,x2
1,1,5
1,2,4
-1,2,2
-1,4,4
```

This produces the same result:

```
     y
0   1.0
1   1.0
2  -1.0
3  -1.0
```

19.5 Slack variables

We can also make the SVM less brittle by introducing a 'slack' variable ξ_n for each data point:

$$\underset{\mathbf{w} \text{ s.t. } \forall_{\langle y_n, \mathbf{x}_n \rangle \in \mathcal{D}} \; y_n (\mathbf{w}^\top \mathbf{x}_n + b) \geq 1 - \xi_n}{\operatorname{argmin}} \frac{1}{2} \|\mathbf{w}\|_2^2 + \sum_n \xi_n \quad \xi_n \geq 0$$