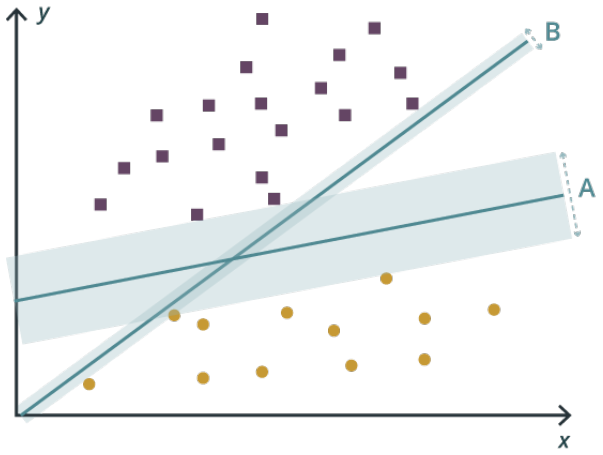
Support Vector Machines

MATH1900: Machine Learning

Location: http://people.sc.fsu.edu/~jburkardt/classes/ml_2019/support_vector_machines/support_vector_machines.pdf



What line maximizes the separation between two sets of data?

The general hyperplane separation problem

Given m items of n-dimensional data x, some of which belong to set P, find the coefficients w of a hyperplane

$$y(x_i) = w_0 + \sum_{j=1}^{n} w_j x_{i,j}$$

so that $y(x_i)$ indicates whether x_i belongs in set P.

In the logistic regression problem, we have seen a similar attempt to compute a separator for two sets of data. In that problem, the solution is determined by finding weights w that minimize the cost:

$$J(w) = \frac{1}{m} \sum_{i=1}^{m} -y_i \log(y(x_i)) + (1 - y_i) \log(1 - y(x_i))$$

Now we are going to look at two other classification methods which construct a hyperplane by different procedures.

The perceptron problem

Given m items (x_i, y_i) of n-dimensional data, each either in class 0 or class 1, find a hyperplane

$$y(x_i) = w_0 + \sum_{j=1}^{n} w_j x_{i,j}$$

so that

$$\begin{cases} y(x_i) < 0 \text{ if } x_i \in \text{class 0} \\ y(x_i) > 0 \text{ if } x_i \in \text{class 1} \end{cases}$$

1 The perceptron problem

A set of data, consisting of two classes, is *linearly separable* if we can find a line (or plane or hyperplane) that correctly splits the data. If we can find such a line, it allows us to easily determine where data belongs, possibly to guess why the two classes are different, and to predict the class of a new data item.

The perceptron problem assumes we have been given such a set of data, including the labels indicating which class each item belongs to, and asks us to find a suitable division line.

Suppose that we have m sets of data, with the i-th set denoted by (x_i, y_i). While y_i is a single number (a class that is either 0 or 1), the quantity x_i is an n-vector, with j-th component $x_{i,j}$. In the usual way, we will insert into x an extra, initial value $x_{i,0}$ which is always 1. Hence we can write our desired formula as

$$y(x_i) = \sum_{j=0}^{n} w_j x_{i,j} = w' x_i \approx y_i$$

For the i-th data item (x_i, y_i), we have an error e_i:

$$e_i = (y_i - y(x_i))$$

and our sum-of-squares error is

$$E = \sum_{i=1}^{m} e_i^2$$

To use gradient descent, we need the $n+1$ component gradient vector:

$$\nabla(E) = \left[\begin{array}{cccc} \frac{\partial E}{\partial w_0}, & \frac{\partial E}{\partial w_1}, & \cdots & \frac{\partial E}{\partial w_n} \end{array} \right]$$

and we can see that, for every index j, we have:

$$\frac{\partial E}{\partial w_j} = -2 * \sum_{i=1}^{m} (y - y_i) * x_{i,j}$$

We can ignore the factor of -2, and now use gradient descent to try to find values w which make the gradient zero, and hence minimize the sum-of-squares error E.

Once we have our weights w, we can classify any data item x. Of course, since all negative values of $y(x)$ will map to class 0 and all positives to class 1, we can assign a data item x based on the sign of $y(x)$:

$$\text{class}(x) = \begin{cases} 1, \text{if } 0 \le y(x) \\ 0, \text{otherwise} \end{cases}$$

2 A gradient descent code for the perceptron problem

Our program will continue as long as any classification is wrong. (We are assuming that a perfect separating line exists. Otherwise, our program will never stop!)

We use a learning rate `alpha` to slow down our descent. We should probably also normalize columns 1 through n of the x data before starting. We can set an initial guess for the weights as $\frac{1}{n+1}$.

```
def perceptron_gradient ( x_data, y_data, alpha, stepmax, w0 ):

  import numpy as np

  m = len ( y_data )

  w = np.copy ( w0 )
  e = 0
  step = 0

  while ( step < stepmax ):

    e = 0
    step = step + 1

    for i in range ( 0, m ):
      y = np.dot ( w[:], x_data[i,:] )
      f = ( 0 < y )
      e = e + ( y_data[i] != f )
      w = w + alpha * np.dot ( ( y_data[i] - f ), x_data[i,:] )

    w = w / np.linalg.norm ( w )
    if ( e == 0 ):
      break

  return w, e, step
```

Listing 1: Gradient descent for a perceptron problem.

3 Example: The perceptron algorithm for generator data

The Stark Industries data center relies for backup power on an array of 56 generators. During a recent blackout, the generators were turned on, and some of them were found to be defective. The main console measured the RPM and vibration rate of each generator. The file *generator_data.txt* contains these measurements, as $n = 56$ rows of 4 items of data:

1. **id**: the generator ID;
2. **rpm**: the rotational rate;
3. **vib**: the vibrational rate;
4. **ok**: the generator status: 0 = "failed", 1 = "working"

We want to determine the weights w_0, w_1 and w_2 for a formula:

$$y = w_0 * 1 + w_1 * rpm + w_2 * vib$$

so that

$$\text{ok}(x) = \begin{cases} 1, \text{(working) if } 0 \leq y(x) \\ 0, \text{(failed) otherwise} \end{cases}$$

that will allow us to predict, in the future, whether a generator will fail, based on the `rpm` and `vib` measurements.

To simplify our notation, we will create an extra variable,

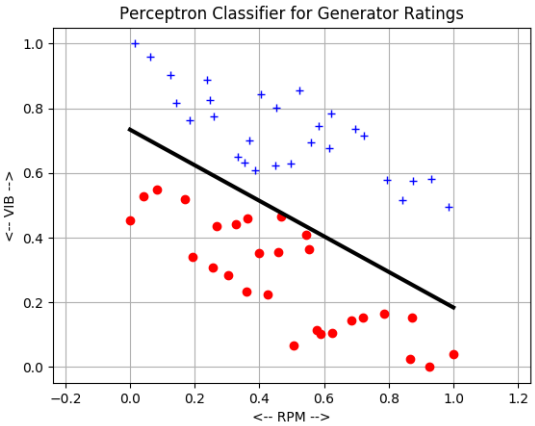
1. **one**: a vector of 1's;

We create a 3-column data matrix $x = |one|rpm|vib|$.

```
import numpy as np

data = np.loadtxt ( 'generator_data.txt' )
rpm = data[:,1]
vib = data[:,2]
act = data[:,3]
m = len ( rpm )

r = ( rpm - np.min ( rpm ) ) / ( np.max ( rpm ) - np.min ( rpm ) )
v = ( vib - np.min ( vib ) ) / ( np.max ( vib ) - np.min ( vib ) )

alpha = 0.01
w0 = np.ones ( 3 ) / 3.0
x = np.zeros ( [ m, 3 ] )
x[:,0] = 1.0
x[:,1] = r
x[:,2] = v

w = perceptron_gradient ( x, y, alpha, stepmax, w0 )
```

Listing 2: Solving the perceptron problem for generator data.

If we plot our generator data and the perceptron line, we see that we have successfully classified our data. On the other hand, the line could surely have been chosen more carefully, so that it ran down the middle of the empty space between the two classes. However, to be guaranteed of such a result, we must turn to support vector machines.



The dividing line produced by the perceptron algorithm

4 Solving a quadratic program with cvxopt

The support vector machine approach improves on the perceptron by producing the separating line that runs exactly halfway down the space that separates the two classes of data (assuming there is such space!). Naturally, an SVM problem is somewhat more complicated. One way to solve an SVM problem results in a formulation known as a quadratic program (like a linear program, but with a quadratic objective function). To solve such problems, we can use a quadratic programming package such as `cvxopt`.

The Python package `cvxopt` can be downloaded and installed by the command `pip3 install cvxopt`. `cvxopt()` assumes that we are trying to solve a general quadratic problem of the form:

$$\begin{aligned} \text{minimize: } & f(x) = \frac{1}{2}x'Qx + p'x \\ \text{subject to: } & G\,x \leq h \\ & A\,x = b \end{aligned}$$

As an example, consider a problem in which x is a vector of length 2, and we want to solve:

$$\begin{aligned} \text{minimize: } & f(x) = 2x_0^2 + x_1^2 + x_0x_1 + x_0 + x_1 \\ \text{subject to: } & x_0 \geq 0, x_1 \geq 0 \\ & x_0 + x_1 = 1 \end{aligned}$$

With some thought, we can set up the corresponding arrays that `cvxopt` expects:

$$f(x) = \frac{1}{2}\begin{bmatrix} x_0 \\ x_1 \end{bmatrix}' 2 * \begin{bmatrix} 2 & 0.5 \\ 0.5 & 1 \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix}$$

We can now set up our problem to be solved. Note that `cvxopt` uses a `matrix()` command to set up vectors and matrices. Once our solution vector `x` is computed, we can easily convert it into our familiar `np.array()` data type:

```
1  from cvxopt import matrix
2  from cvxopt import solvers
3
4  Q = 2*matrix([ [2, .5], [.5, 1] ])
5  p = matrix([1.0, 1.0])
6
7  G = matrix([[-1.0,0.0],[0.0,-1.0]])
8  h = matrix([0.0,0.0])
9
10 A = matrix([1.0, 1.0], (1,2))  # Makes this a (1,2) matrix instead of a vector
11 b = matrix(1.0)
12
13 sol = solvers.qp ( Q, p, G, h, A, b )
14
15 import numpy as np
16
17 x = np.array ( sol['x'] )
18 x = np.squeeze ( x )
19
20 print ( '' )
21 print ( '  Cost is minimized at x = [', x[0], ',', x[1], ']' )
22 cost = 2.0 * x[0]**2 + x[1]**2 + x[0]*x[1] + x[0] + x[1]
23 print ( '  Minimized cost is ', cost )
```

Running this program produces the output:

```
1    Cost is minimized at x = [ 0.2500000952702475 , 0.7499999047297524 ]
2    Minimized cost is  1.8750000000000178
```

We will be able to describe the support vector machine problem as a similar example of quadratic programming, so that we can use `cvxopt()` to get a solution.

The Support Vector Machine problem

Given m items of data x with each item having dimension n, some of which belong to a set P, find weights w of a function so that for each item x, we can compute

$$y(x_i) = \sum_{j=0}^{n} w_j x_{i,j}$$

so that

$$\begin{cases} y_i <= -1 \text{ if } x_i \notin P \\ y_i => +1 \text{ if } x_i \in P \end{cases} \quad (1)$$

and for which the hyperplane $y = 0$ creates the widest possible margin between the two sets of data.

5 The support vector machine problem

We are given m sets of data $\{x_i, y_i\}$, where each x_i is an n vector, and y_i is a classification, which, by SVM convention, will have the value -1 or +1. (Note that, for the perceptron, we used classifications of 0 and 1.)

We assume our data is linearly separable, that is, that at least one straight line can be drawn that splits the data into its two groups. If so, there will be many such lines. The SVM approach seeks the best such line, which maximizes the separation margin between the two data sets.

In general, the equation of a hyperplane can be represented as

$$f(x) = b + \sum_{j=1}^{n} w_j x_j = b + w' * x = 0$$

but, as in the perceptron problem, we want to simplify our notation by inserting a variable x_0 which is identically 1, so we can write:

$$f(x) = w_0\, 1 + \sum_{j=1}^{n} w_j x_j = \sum_{j=0}^{n} w_j x_j = w' * x = 0$$

Given data, our goal is to find weights w which produce the best separating line. The function $f(x)$ can be used to measure how far any data item is. These values will always be -1 or less, for points on the negative side of the line, and +1 or greater on the positive side.

If our dataset has spatial dimension 2, we can easily visualize the results as a highway separating the two sets of data, chosen in such a way that the highway is as wide as possible.

The SVM problem is similar to the Perceptron problem, but there are two changes. First, we require that y not just be on the right side of zero for each data value, but actually be at least 1 unit away from zero. It turns out that the second requirement can be rephrased as follows:

$$\text{find the weights } w_0, ...w_n \text{ of which minimize } \frac{1}{2}\sum_{j=1}^{n} w_j^2 \text{ and satisfy inequality (1)}$$

How can we find the appropriate weights w for a given set of data?

The SVM system can be rewritten as seeking weights w which:

$$\begin{aligned} \text{minimize: } & \frac{1}{2}\sum_{j=1}^{n} w_j^2 \\ \text{subject to: } & y_i * (w' x_i) \geq 1 \text{ for each data } i \end{aligned}$$

In other words, the sign of $w'x_i$ must always match the sign of our data label y_i.

Recall that `cvxopt()` assumes that we are trying to solve a general quadratic problem of the form:

$$\begin{aligned} \text{minimize: } & f(x) = \frac{1}{2} x' Q x + p' x \\ \text{subject to: } & G\,x \leq h \\ & A\,x = b \end{aligned}$$

In order to make our problem fit the pattern expected by `cvxopt`, we have to set up the following objects. Recall that we have m data items. We will let n indicate the dimension of our original data, and $N = n + 1$ is the dimension of the data after we insert the extra "1" variable.

$$\begin{aligned} x &\rightarrow w \\ Q &\rightarrow \begin{pmatrix} 0 & 0 \\ 0 & I_{n\times n} \end{pmatrix} \\ p &\rightarrow \begin{pmatrix} 0_{N\times 1} \end{pmatrix} \\ G &\rightarrow -\operatorname{diag}(y) * \begin{pmatrix} 1_{m\times 1} & | & X \end{pmatrix} \\ h &\rightarrow \begin{pmatrix} -1_{m\times 1} \end{pmatrix} \\ A &\rightarrow \text{not used!} \\ b &\rightarrow \text{not used!} \end{aligned}$$

Here, $\operatorname{diag}(y)$ is the $m \times m$ diagonal matrix whose diagonal entries are the values of the y data (the classifications), and X is the $m \times n$ matrix of data values before we insert the initial column of 1's.

We begin in the usual way, reading our data, but we won't need to normalize the data because our solver is well-written. We simply call a function `w = fit(rpm, vib, grade)` which will handle the `cvxopt()` solution procedure as a black box.

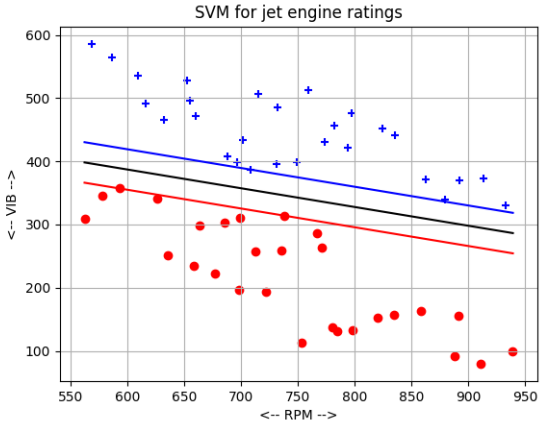
```
1   import numpy as np
2
3   data = np.loadtxt ( 'jet_data.txt' )
4
5   rpm = data[:,1]
6   vib = data[:,2]
7   grade = data[:,3]
8   m = len ( rpm )
9
10  w = fit ( rpm, vib, grade )
```

The `fit` function has to set up the arrays we need, convert them to the `matrix` format, and call the appropriate solver:

```
1  def fit ( rpm, vib, grade ):
2    from cvxopt import matrix, solvers
3    import numpy as np
4
```

```
5   m = len ( rpm )
6
7   Q = 0.5 * np.array ( [ \
8     [ 0.0, 0.0, 0.0 ], \
9     [ 0.0, 1.0, 0.0 ], \
10    [ 0.0, 0.0, 1.0 ] ] )
11
12  p = np.zeros ( 3 )
13
14  G = np.zeros ( [ m, 3 ] )
15  G[:,0] = 1
16  G[:,1] = rpm
17  G[:,2] = vib
18  G = - np.dot ( np.diag ( grade ), G )
19
20  h = -1.0 * np.ones ( m )
21
22  Q = matrix ( Q )
23  p = matrix ( p )
24  G = matrix ( G )
25  h = matrix ( h )
26
27  sol = solvers.qp ( Q, p, G, h )
28  w = np.array ( sol['x'] )
29
30  return w
```

To make a plot of the hyperline and its two margins, we simply want to use the weights w to plot $y(x) = -1, 0, 1$ as red, black, and blue lines. It takes a little work to figure out how to specify the portion of these lines to display:



The -1, 0, and +1 lines for the jet SVM.

6 Computing Assignment #8

The file *gopher_data.txt* contains 50 records; each record lists the skull width and skull length in centimeters, and a species label, which is either -1 (Golden Gopher) or +1 (Hopping Gopher).

Write a program *hw8.py* which computes the weight vector w for an SVM for this data.

Email a copy of your program to Dr Schneier **mhs64@pitt.edu** by Wednesday, 6 November.