ML Commando Course 2018

Image Recognition with Support Vector Machines

Russell Moore

ALTA / Computer Laboratory

August 2018

Get the notebook

`https://github.com/rjm49/ml_commando_2018/blob/master/session_3_images.ipynb`

Computer Vision

Have nightmares with the 'Olivetti Faces' dataset!

Faces - what we'll do

In this example the tools we'll be looking at are:

- Lots of stuff we've seen before (pre-processing)
- Some image display code
- Support Vector Machine (SVM) Classification

Faces - data

Passport-style mono photos of Olivetti team members, cropped tightly to remove identifying peripheral features.

- Images (400, 64, 64) .. actual pixel images (these are just to look at, the classifier doesn't use them!)
- Image data (400, 4096) .. pixel luminosity scores, images are unravelled into a single row of 'pixels'
- target data (400,) .. this is the Person ID.

Loads of features

- For our Titanic analysis we had 5 features.
- This time we have 4096!
- Large numbers of features can be a good thing - they allow us to specify a complex decision boundary between our classes.
- Note that humans are not wired to work this way - we seem to use 'logon' based packets of information and recognise eyes, mouth etc... the machine doesn't even know what a nose is.
- So it's not obvious that facial recognition based on a big sequence of numbers should even work at all (maybe it won't).

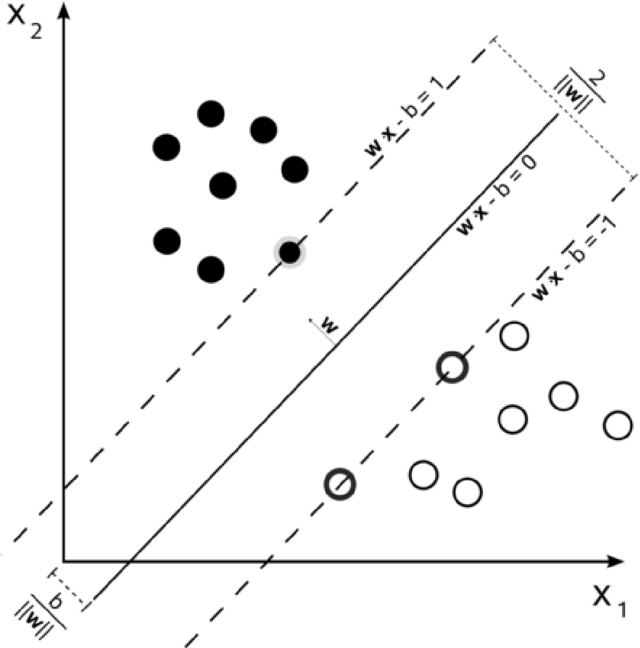
Support Vector Machines

- Support Vector Machines (SVMs) were invented in 1963 by Vladimir Vapnik, and extensively improved in the 1990s.
- They are slightly similar to the linear classifier we implemented for iris data, except...
- They try to find the maximum separation between classes (and so use a different cost function).
- They only care about edge ('supporting') cases of classes
- Usually target class $y_i \in \{-1, +1\}$
- They are very effective and can be trained on fairly small datasets, even when there are large numbers of features.
- Basic form is a linear classifier but they can be extended to non-linear cases.

The margin

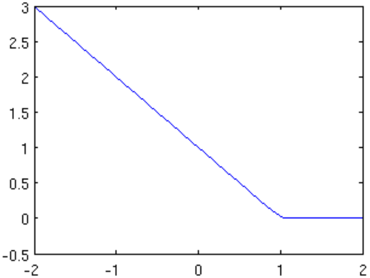
- We have 2 classes that we want to separate
- Assuming they are linearly separable we can draw a line between them
- But let's assume we want to separate them with more than just a line - we want a kind of 'buffer' between them, as wide as possible. This is called the *margin*.
- Assuming we have a decision line D_0, let's see how the margin relates to it...

Obligatory SVM diagram



Cost pt1 - cost of misclassification

- We want misclassified samples within the margin to incur some kind of cost (but correct or non-marginal samples should not).
- Use a 'hinge loss' function (in each direction):



- same as $max(0, 1 - y_i(\vec{w} \cdot \vec{x_i} - b))$ when $y_i \in \{-1, +1\}$
- So $loss = 0$ if $|y_{pred}| \geq 1$ and $sign(y_{pred}) == sign(y_{true})$ else $loss > 0$

Cost pt2 - cost of narrow margin

- Take points $\vec{x_+}$ and $\vec{x_-}$ on each margin boundary
- The difference $(\vec{x_+} - \vec{x_-})$ is itself a vector pointing from one boundary to the other (possibly at an angle!). The component of this vector in direction $\vec{w}$, describes the size of the margin:

$$M = \frac{(x_+ - x_-) \cdot \vec{w}}{|\vec{w}|}$$

- Using what we already know:

$$\vec{w} \cdot \vec{x_+} - b = +1$$
$$\vec{w} \cdot \vec{x_-} - b = -1$$

so..

$$\vec{w} \cdot (\vec{x_+} - \vec{x_-}) = 2$$

- And so:

$$M = \frac{2}{|\vec{w}|}$$

- Minimise $|\vec{w}|$ to maximise margin!

Linear SVM cost function

- Combining the two previous steps, a linear SVM has the following cost function. We try to minimise (by gradient descent):

$$Q(\vec{x}; \vec{w}, b) = C \left(\sum_{i} max(0, 1 - y_i(\vec{w} \cdot x_i - b)) \right) + \frac{1}{2}|\vec{w}|^2$$

- The C-term controls how much we penalise misclassified points. The w-term makes sure our margin is minimal.
- The w-term is so formulated to make it easier to differentiate.
- The *C hyperparameter* allows you to alter the behaviour to avoid overfitting. If C is large, the SVM will fit training data very closely, but may not generalise well to unknown data.

Refresher: Vectors

- Take two vectors, $\vec{u} = [u_1, u_2]$ and $\vec{v} = [v_1, v_2]$
- 'Dot product' gives us a scalar from two vectors:

$$\vec{u} \cdot \vec{v} = |\vec{u}||\vec{v}| \cos\theta \text{ (geometric definition)}$$
$$\vec{u} \cdot \vec{v} = u_1 v_1 + u_2 v_2 \text{ (arithmetic definition)}$$

- Because of the *cosine* term, a projection of vector u onto vector v can be written: $proj(u, v) = |u| \cos\theta = \frac{(u \cdot v)}{|v|}$
- Important: this is a number that tells us the length of u in the direction of v...
- Also, if $\theta > 90^o$ then $\cos\theta < 0$

Refresher: Distances from a hyperplane

- If we have a hyperplane (line, plane, etc) described by a normal vector $\vec{w}$ then we can measure the distance from that plane of any sample x_i using a projection onto $\vec{w}$, namely $d(x_i) = \frac{x_i \cdot \vec{w}}{|\vec{w}|}$
- Note that $d(x_i)$ will give a negative value if it describes a point on 'reverse' side of the hyperplane.
- So now we know how far any point is from any decision boundary we might choose ... (or, how far a decision boundary is from any and all points.)