Massachvsetts Institvte of Technology
Department of Electrical Engineering and Computer Science
6.034 Recitation, Thursday, November 17

SVMs & Boosting, II — Prof. Bob Berwick, 32D-728

Part I. SVMs: solving for support vectors using the kernel function directly
Question 1. Instead of using the 'eyeball' technique, we can use the constraints of the SVM optimization problem to solve for the support vector 'alpha' values *directly*, and so find the decision boundary line. This is useful when we have no geometric picture, especially when we move to non-linear kernel functions. But first, let's do this for a simple case, to gain practice. Let us consider solving for the following SVM classifier:

$$h(\vec{x}) = \mathrm{sgn}\left(\sum_i \alpha_i y_i K(\vec{x}_i, \vec{x}) + b\right)$$

where we will use first a **linear** kernel function $K(\vec{u},\vec{v}) = \vec{u}\cdot\vec{v}$, i.e., the normal dot product, on the following data set with just 3 training data points, *A, B,* and *C*:

data point *A*: (–1, 1); classifier output: +1
data point *B*: (0, 0); classifier output: –1
data point *C*: (1, 0); classifier output: +1

Part A. Instead of doing this geometrically, we now explicitly compute the kernel function values for *each* of the 3 points, pairwise against one another (these are just dot product computations!)

i	j	$K(\vec{x}_i,\vec{x}_j)$
A	A	$(-1,+1)\cdot(-1,+1) = 2$
A	B	$(-1,+1)\cdot(0,0) = 0$
A	C	$(-1,+1)\cdot(1,0) = -1$
B	B	$(0,0)\cdot(0,0) = 0$
B	C	$(0,0)\cdot(+1,0) = 0$
C	C	$(+1,0)\cdot(+1,0) = +1$

Part B. Now we can find the system of linear equations governing the support vector alphas, α_A, α_B, and α_C that can be derived given the two key constraints given by the Lagrangian formulation of the SVM optimization problem:
Constraint 1. $\sum_i \alpha_i y_i = 0$
Which for the 3 points **with support** (ie, that are support vectors), *A, B, C,* we therefore have the equation:

$$\alpha_A(1) + \alpha_B(-1) + \alpha_C(1) = 0$$

Constraint 2. For all points **with support**, we have:

$$y = \sum_i \alpha_i y_i K(\vec{x}_i, \vec{x}) + b$$

For points *A, B, C,* we therefore have the summation over 3 terms implies 3 equations as follows (where we have switched the terms of $\vec{x}_i$ and $\vec{x}$ from the equation above, don't be deceived – the inner products are commutative):

$$+1 = \alpha_A K(x_A, x_A) - \alpha_B K(x_A, x_B) + \alpha_C K(x_A, x_C) + b$$
$$-1 = \alpha_A K(x_B, x_A) - \alpha_B K(x_B, x_B) + \alpha_C K(x_B, x_C) + b$$
$$+1 = \alpha_A K(x_C, x_A) - \alpha_B K(x_C, x_B) + \alpha_C K(x_C, x_C) + b$$

Part C. But we can now simplify these 3 equations, by plugging in the actual values for *K* as computed above, to get these 3 equations:

$$+1 = 2\alpha_A - \alpha_C + b$$
$$-1 = b$$
$$+1 = -\alpha_A + \alpha_C + b$$

Thus we immediately can solve for b=–1. Now we can solve for α_A by adding the first and third equations together, getting:

$$+2 = \alpha_A + 2(-1)$$
$$+4 = \alpha_A$$

Using this value for α_A, we can now find the value for α_C:

$$+1 = -4 + \alpha_C - 1$$
$$6 = \alpha_C$$

Finally, using *Constraint 1*, we can solve for α_B, since the sum from *Constraint 1* must be 0. So $\alpha_A + \alpha_C = \alpha_B$, therefore, α_B= 4 + 6 = 10, and we have: α_A=4; α_B=10; α_C =6; b=–1.

Part D. Now, what is the SVM classifier equation for h given this answer? This follows directly from the boundary line definition in the SVM Winston notes:

$$\begin{aligned} h(x_1,x_2) &= \text{sgn}(\alpha_A K(x_A,x) + \alpha_B K(x_B,x) + \alpha_C K(x_C,x) + b) \\ &= \text{sgn}(4(-1,+1)\cdot(x_1,x_2) + 10(0,0)\cdot(x_1,x_2) + 6(+1,0)\cdot(x_1,x_2) - 1) \\ &= \text{sgn}(4(-1,+1)\cdot(x_1,x_2) + 6(+1,0)\cdot(x_1,x_2) - 1) \\ &= \text{sgn}(2x_1 + 4x_2 - 1) \end{aligned}$$

Note: the middle term with α_B has dropped out here because $K(x_B,x)$ is always 0 no matter what x is. This equation defines the boundary line midway between the gutters as well, all without having to draw anything:

$$2x_1 + 4x_2 - 1 > 0$$
$$x_2 = -\frac{1}{2}x_1 + \frac{1}{4}$$

Part E. Let us see what happens when we *add* new training data points.
If we added a new point x_D at (2,0), with output y_D= +1, nothing would change, because this point would not have any support vector weight, and it would be correctly classified with what we have so far.
If instead y_D= –1, then everything would change, because this would force us to redefine where the decision boundary line goes.

Question 2. Non-linear kernels.
Now let's repeat this game with the *same* three data points, *but* with a **non-linear** kernel function, the **quadratic form**:

$$K(\vec{u},\vec{v}) = (1 + \vec{u}\cdot\vec{v})^2$$

Part A. Recall that this kernel will give parabolic shaped contour lines. So, first, we just have to go through the kernel computation as we did with the linear kernel, but now of course we are computing a different 'similarity' function, using the new kernel function. Note that we can make use of our old values of $u{\cdot}v$ from our previous table to speed up the computation, i.e., for point *A,* dot product of *A* with itself is just 2, from our table above:

i	j	$K(\vec{x}_i,\vec{x}_j) = (1+\vec{u}\cdot\vec{v})^2$
A	A	$(1+2)^2 = 9$
A	B	$(1+0)^2 = 1$
A	C	$(1+-1)^2 = 0$
B	B	$(1+0)^2 = 1$
B	C	$(1+0)^2 = 1$
C	C	$(1+1)^2 = 4$

Part B. As before, we can now set up a system of equations from the Lagrangian constraint solution.

Constraint 1. $\sum_i \alpha_i y_i = 0$

Constraint 2. For all points **with support**, we have:

$$y = \sum_i \alpha_i y_i K(\vec{x}_i, \vec{x}) + b$$

Just as before, these constraints work out to imply:

$$+1 = \alpha_A K(x_A, x_A) - \alpha_B K(x_A, x_B) + \alpha_C K(x_A, x_C) + b$$
$$-1 = \alpha_A K(x_B, x_A) - \alpha_B K(x_B, x_B) + \alpha_C K(x_B, x_C) + b$$
$$+1 = \alpha_A K(x_C, x_A) - \alpha_B K(x_C, x_B) + \alpha_C K(x_C, x_C) + b$$

Part C. But this time the *kernel* values are different! We plug in the *K* values from the quadratic 'inner product' as defined above. When we do this, and solve for the three alpha values (I shall leave this algebra to you this time), we get:

$$\alpha_A = \frac{8}{23}; \alpha_B = \frac{3}{23}; \alpha_C = \frac{18}{23}; b = -\frac{49}{23}$$

Part D. Now we can define the classifier function as we did before, with these new values for the alphas:

$$h(\vec{x} = (x_1, x_2)) = \text{sgn}(\alpha_A K(x_A, x) + \alpha_B K(x_B, x) + \alpha_C K(x_C, x) + b)$$
$$= \text{sgn}(\frac{8}{23}(1 + x_A \cdot x)^2 + \frac{3}{23}(1 + x_B \cdot x)^2 + \frac{18}{23}(1 + x_C \cdot x)^2 - \frac{49}{23})$$
$$= \text{sgn}(\frac{8}{23}(1 + \begin{bmatrix} -1 \\ +1 \end{bmatrix} \cdot x)^2 + \frac{3}{23}(1 + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \cdot x)^2 + \frac{18}{23}(1 + \begin{bmatrix} +1 \\ 0 \end{bmatrix} \cdot x)^2 - \frac{49}{23})$$
$$= \text{sgn}\left(\frac{8}{23}(1 - x_1 + x_2)^2 + 0 + \frac{18}{23}(1 + \cdot x_1)^2 - \frac{49}{23}\right)$$

3. Other spaces, other kernels

OK, now the other big win with SVMs has to do with the ease with which you can transform from one space to another, where the data may be more easily separable. The remaining questions all ask you about that, for **different** kinds of kernels (= different ways to compute inner products, or 'distance', aka, 'similarity' of two points). That is, we need to define *K*(*u,v*) = φ(*u*)·φ(*v*), the dot product in the transformed space. (You should try these out in Winston's demo program to see how the decision boundaries change.)
The basic kernels we consider are these:

1. Single linear kernel. These are just straight lines in the plane (or in higher dimensions). You should remember what perceptrons can and cannot 'separate' via cuts, and this tells you what linear kernels can do. (But see below under linear combination of kernels!!!)
$K(\vec{u}, \vec{v}) = (\vec{u} \cdot \vec{v}) + b$, e.g., $K(\vec{u}, \vec{v}) = (\vec{u} \cdot \vec{v})$ (ordinary dot product)

2. Polynomial kernel.
$K(\vec{u}, \vec{v}) = (\vec{u} \cdot \vec{v} + b)^n$, $n > 1$

eg., Quadratic kernel: $K(\vec{u}, \vec{v}) = (\vec{u} \cdot \vec{v} + b)^2$

In 2-D the resulting decision boundary can look parabolic, linear, or hyperbolic depending on which terms in the expansion dominate.

3. Radial basis function (RBF) or Gaussian kernel.

$$K(\vec{u},\vec{v}) = -\exp\left(-\frac{\|\vec{u}-\vec{v}\|^2}{2\sigma^2}\right)$$

In 2-D, the decision boundaries for RBFs resemble contour circles around clusters of positive and negative points Support vectors are generally positive or negative points that are closest to the opposing cluster. The contour space that results is drawn from the sum of support vector Gaussians. Try the demo to see.
When the variance or spread of the Gaussian curve σ^2 ('sigma-squared') is large, you get 'wider' or 'flatter' Gaussians. When it is small, you get sharper Gaussians. Hence, when using a small sigma-squared, the contour density will appear closer, or tighter, around the support vector points. In 2-D, as a point gets closer to a support vector, it will approach *exp*(0)=1, and as it gets farther away, it approaches *exp*(–infinity)= 0.
Note that you can **combine** several radial basis function kernels to get a perfect fit around **any** set of data points, but this will usually amount to a typical case of over-fitting – there are 2 free parameters for every RBF kernel function.

4. Sigmoidal (tanh) kernel. This allows for a combination of linear decision boundaries, like neural nets.

$$K(\vec{u},\vec{v}) = \tanh(k\vec{u}\cdot\vec{v}+b)$$

$$K(\vec{u},\vec{v}) = \frac{e^{k\vec{u}\cdot\vec{v}+b}+1}{e^{k\vec{u}\cdot\vec{v}+b}-1}$$

The properties of this kernel function: it is similar to the sigmoid function; it ranges from –1 to +1; it approaches +1 when $x \gg 0$; and it approaches –1 when $x \ll 0$. The resulting decision boundaries are logical combinations of linear boundaries, not that different from second-layer neurons in neural nets.

5. Linear combinations of kernels (scaling or general linear combination).
Kernel functions are closed under addition and scaling by a positive factor.

Part II. Boosting and the Adaboost algorithm

0. The idea behind **boosting** is to find a weighted combination of *s* "weak" classifiers (classifiers that underfit the data and still make mistakes, though as we will see they make mistakes on less than ½ the data), h_1, h_2...,h_s, into a **single strong** classifer, $H(x)$. This will be in the form:

$$H(\vec{x}) = sign(\alpha_1 h_1(\vec{x}) + \alpha_2 h_2(\vec{x}) + \cdots + \alpha_s h_s(\vec{x})$$

$$H(\vec{x}) = sign\left(\sum_{i=1}^{s} a_i h_i(\vec{x})\right)$$

where: $H(\vec{x}) \in \{-1,+1\}, h_i(\vec{x}) \in \{-1,+1\}$

Recall that the *sign* function simply returns +1 if weighted sum is positive, and –1 if the weighted sum is negative (i.e., it classifies the data point as + or –).
Each training data point is **weighted.** These weights are denoted w_i for i=1, ..., n. **Weights** are *like* probabilities, from the interval (0, 1], with their **sum equal to 1. BUT** weights are **never 0**. This implies that **all data points have some vote** on what the classification shuld be, at all times. (You might contrast that with SVMs.)

The general idea will be to pick a single 'best' classifier *h* (one that has the lowest error rate when acting all alone), as an initial 'stump' to use. Then, we will **boost** the weights of the data points that this classifier **mis-classifies (makes mistakes on),** so as to focus on the next classifier *h* that does best on the re-weighted data points. This will have the effect of trying to fix up the errors that the first classifier made. Then, using this next classifier, we repeat to see if we can now do better than in the first round, and so on. In computational practice, we use the same sort of entropy-lowering function we used with ID/classifier trees: the one to pick is the one that lowers entropy the most. But usually we will give you a set of classifiers that is easier to 'see', or will specify the order.

In Boosting we always pick these initial 'stump' classifiers so that the error rate is strictly < ½. Note that if a stump gives an error rate greater than ½, this can always be 'flipped' by reversing the + and – classification outputs. (If the stump said –, we make it +, and vice-versa.) Classifiers with error exactly equal to ½ are useless because they are no better than flipping a fair coin.

1. Here are the definitions we will use.
Errors:
The error rate of a classifier *s*, E^s, is simply the sum of all the *weights* of the training points classifier h_s gets **wrong**.
($1–E^s$) is 1 minus this sum, the sum of all the *weights* of the training points classifier h_s gets **correct.**
By assumption, we have that:
$E^s < ½$ and $(1– E^s) > ½$, so $E^s < (1– E^s)$, which implies that $(1– E^s)/E^s > 1$

Weights:
α_s is **defined** to be ½ ln[(1– E^s)/ E^s)], so from the definition of weights, the quantity inside the ln term is > 1, so all alphas must be positive numbers.

Let's write out the Adaboost algorithm and then run through a few iterations of an example problem.

2. Adaboost algorithm

Input: training data, $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$

1. **Initialize data point weights.**

   Set $w_i^1 = \frac{1}{n} \quad \forall i \in (1, \ldots, n)$

2. **Iterate over all 'stumps':** for s=1, ..., T

   a. **Train base** learner using distribution w^s on training data.
   Get a base (stump) classifier $h_s(x)$ that achieves the lowest error rate E^s .
   (In examples, these are picked from pre-defined stumps.)

   b. **Compute the stump weight**: $\alpha_s = \frac{1}{2} \ln \frac{(1 - E^s)}{E^s}$

   c. **Update weights** (3 ways to do this; we pick Winston's method)

   For points that the classifier **gets correct,** $w_i^{s+1} = \left[\frac{1}{2} \cdot \frac{1}{1 - E^s} \right] \cdot w_i^s$

   (Note from above that $1 - E^s > ½$, so the fraction $1/(1 - E^s)$ must be < 2, so the total factor scaling the old weight must be < 1, i.e., the **weight of correctly classified points must go DOWN in the next round)**

   For points that the classifier **gets incorrect,** $w_i^{s+1} = \left[\frac{1}{2} \cdot \frac{1}{E^s} \right] \cdot w_i^s$

   (Note from above that $E^s < ½$, so the fraction $1/E^s$) must be > 2, so the total factor scaling the old weight must be > 1, i.e., the **weight of incorrectly classified points must go UP in the next round**)

3. **Termination condition:**

   If $s > T$ or if $H(x)$ has error 0 on training data or < some error threshold, exit;
   If there are no more stumps h where the weighted error is < ½, exit (i.e., all stumps now have error exactly equal to ½)

4. **Output final classifier:**

$H(\vec{x}) = sign\left(\sum_{i=1}^{s} a_i h_i(\vec{x}) \right)$ **[this is just the weighted sum of the original stump classifiers]**

Note that test stump classifiers that are **never** used are ones that make more errors than some pre-existing test stump. In other words, if the set of mistakes stump X makes is a **superset** of errors stump Y makes, then Error(X) > Error(Y) is **always** true, no matter weight distributions we use. Therefore, we will **always** pick Y over X because it makes fewer errors. So X will **never** be used!

3. Let's try a boosting problem from an exam (on the other handout).

4. Food for thought questions.

1. How does the weight α^s given to classifier h_s relate to the performance of h_s as a function of the error E^s?
2. How does the error of the classifier E^s affect the new weights on the samples? (How does it raise or lower them?)
3. How does AdaBoost end up treating outliers?
4. Why is not the case that new classifiers "clash" with the old classifiers on the training data?
5. Draw a picture of the training error, theoretical bound on the true error, and the typical test error curve.
6. Do we expect the error of new weak classifiers to increase or decrease with the number of rounds of estimation and re-weighting? Why or why not?

Answers to these questions:

1. How does the weight α^s given to classifier h_s relate to the performance of h_s as a function of the error E^s?

Answer: The lower the error the better the classifier h is on the (weighted) training data, and the larger the weight α^t we give to the classifier output when classifying new examples.

2. How does the error of the classifier E^s affect the new weights on the samples? (How does it raise or lower them?)

Answer: The lower the error, the better the classifier h classifies the (weighted) training examples, hence the larger the increase on the weight of the samples that it classifies incorrectly and similarly the larger the decrease on those that it classifies correctly. More generally, the smaller the error, the more significant the change in the weights on the samples. Note that this dependence can be seen indirectly in the AdaBoost algorithm from the weight of the corresponding classifier α_t. The lower the error E^t, the larger α_t, the better h_t is on the (weighted) training data.

3. How does AdaBoost end up treating outliers?

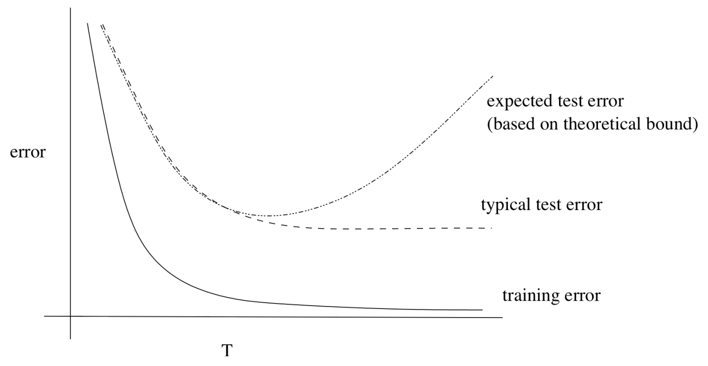
Answer: AdaBoost can help us identify outliers since those examples are the hardest to classify and therefore their weight is likely to keep increasing as we add more weak classifiers. At the same time, the theoretical bound on the training error implies that as we increase the number of base/weak classifiers, the final classifier produced by AdaBoost will classify all the training examples. This means that the outliers will eventually be "correctly" classified from the standpoint of the training data. Yet, as expected, this might lead to overfitting.

4. Why is not the case that new classifiers "clash" with the old classifiers on the training data?

Answer: The intuition is that, by varying the weight on the examples, the new weak classifiers are trained to perform well on different sets of examples than those for which the older weak classifiers were trained on. A similar intuition is that at the time of classifying new examples, those classifiers that are not trained to perform well in such examples will cancel each other out and only those that are well trained for such examples will prevail, so to speak, thus leading to a weighted majority for the correct label.

5. Draw a picture of the training error, theoretical bound on the true error, and the typical test error curve.

Answer:



6. Do we expect the error of new weak classifiers to increase or decrease with the number of rounds of estimation and re-weighting? Why?

Answer: We expect the error of the weak classifiers to increase in general since they have to perform well in those examples for which the weak classifiers found earlier did not perform well. In general, those examples will have a lot of weight yet they will also be the hardest to classify correctly.