Lecture topics:

- Boosting, margin, and gradient descent
- complexity of classifiers, generalization

Boosting

Last time we arrived at a boosting algorithm for sequentially creating an ensemble of base classifiers. Our base classifiers were *decision stumps* that are simple linear classifiers relying on a single component of the input vector. The stump classifiers can be written as $h(\mathbf{x};\theta) = \text{sign}(\, s(x_k - \theta_0)\,)$ where $\theta = \{s, k, \theta_0\}$ and $s \in \{-1, 1\}$ specifies the label to predict on the positive side of $x_k - \theta_0$. Figure 1 below shows a possible decision boundary from a stump when the input vectors $\mathbf{x}$ are only two dimensional.

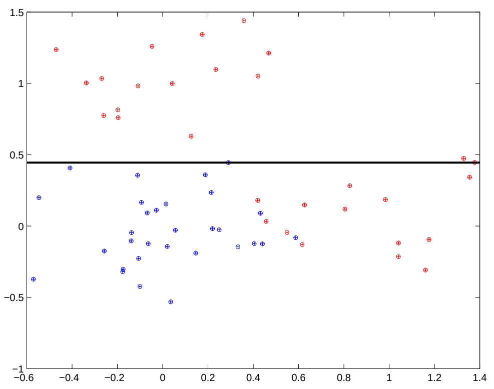

Figure 1: A possible decision boundary from a trained decision stump. The stump in the figure depends only on the vertical x_2-axis.

The boosting algorithm combines the stumps (as base learners) into an ensemble that, after m rounds of boosting, takes the following form

$$h_m(\mathbf{x}) = \sum_{j=1}^{m} \hat{\alpha}_j h(\mathbf{x}; \hat{\theta}_j) \tag{1}$$

where $\hat{\alpha}_j \geq 0$ but they do not necessarily sum to one. We can always normalize the ensemble by $\sum_{j=1}^{m} \hat{\alpha}_j$ after the fact. The simple Adaboost algorithm can be written in the following modular form:

(0) Set $W_0(t) = 1/n$ for $t = 1, \ldots, n$.

(1) At stage m, find a base learner $h(\mathbf{x}; \hat{\theta}_m)$ that approximately minimizes

$$- \sum_{t=1}^{m} \tilde{W}_{m-1}(t) y_t h(\mathbf{x}_t; \theta_m) = 2\epsilon_m - 1 \tag{2}$$

where ϵ_m is the weighted classification error (zero-one loss) on the training examples, weighted by the normalized weights $\tilde{W}_{m-1}(t)$.

(2) Given $\hat{\theta}_m$, set

$$\hat{\alpha}_m = 0.5 \log \left(\frac{1 - \hat{\epsilon}_m}{\hat{\epsilon}_m} \right) \tag{3}$$

where $\hat{\epsilon}_m$ is the weighted error corresponding to $\hat{\theta}_m$ chosen in step (1). For binary $\{-1, 1\}$ base learners, $\hat{\alpha}_m$ exactly minimizes the weighted training loss (loss of the ensemble):

$$J(\alpha_m, \hat{\theta}_m) = \sum_{t=1}^{n} \tilde{W}_{m-1}(t) \exp\Big(-y_t \alpha_m h(\mathbf{x}_t; \hat{\theta}_m) \Big) \tag{4}$$

In cases where the base learners are not binary (e.g., return values in the interval $[-1, 1]$), we would have to minimize Eq.(4) directly.

(3) Update the weights on the training examples based on the new base learner:

$$\tilde{W}_m(t) = c_m \cdot \tilde{W}_{m-1}(t) \exp\Big(-y_t \hat{\alpha}_m h(\mathbf{x}_t; \hat{\theta}_m) \Big) \tag{5}$$

where c_m is the normalization constant to ensure that $\tilde{W}_m(t)$ sum to one after the update. The new weights can be again interpreted as normalized losses for the new ensemble $h_m(\mathbf{x}_t) = h_{m-1}(\mathbf{x}) + \hat{\alpha}_m h(\mathbf{x}; \hat{\theta}_m)$.

Let's try to understand the boosting algorithm from several different perspectives. First of all, there are several different types of errors (errors here refer to zero-one classification losses, not the surrogate exponential losses). We can talk about the weighted error of base learner m, introduced at the m^{th} boosting iteration, relative to the weights $\tilde{W}_{m-1}(t)$ on the training examples. This is the weighted training error $\hat{\epsilon}_m$ in the algorithm. We can also measure the weighted error of the same base classifier relative to the updated weights, i.e., relative to $\tilde{W}_m(t)$. In other words, we can measure how well the current base learner would do at the next iteration. Finally, in terms of the ensemble, we have the unweighted training error and the corresponding generalization (test) error, as a function of boosting iterations. We will discuss each of these in turn.

Weighted error. The weighted error achieved by a new base learner $h(\mathbf{x}; \hat{\theta}_m)$ relative to $\tilde{W}_{m-1}(t)$ tends to increase with m, i.e., with each boosting iteration (though not monotonically). Figure 2 below shows this weighted error $\hat{\epsilon}_m$ as a function of boosting iterations. The reason for this is that since the weights concentrate on examples that are difficult to classify correctly, subsequent base learners face harder classification tasks.

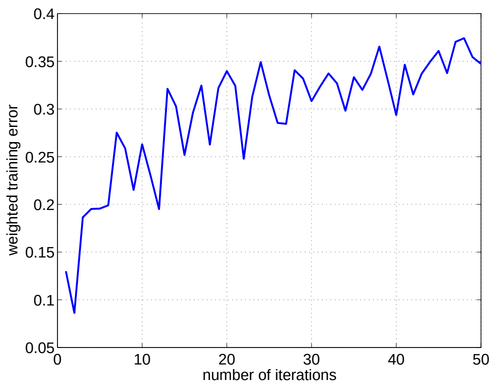

Figure 2: Weighted error $\hat{\epsilon}_m$ as a function of m.

Weighted error relative to updated weights. We claim that the weighted error of the base learner $h(\mathbf{x}; \hat{\theta}_m)$ relative to the updated weights $\tilde{W}_m(t)$ is exactly 0.5. This means that the base learner introduced at the m^{th} boosting iteration will be useless (at chance level) for the next boosting iteration. So the boosting algorithm would never introduce the same base learner twice in a row. The same learner might, however, reappear later on (relative to a different set of weights). One reason for this is that we don't go back and update $\hat{\alpha}_j$'s for base learners already introduced into the ensemble. So the only way to change the previously set coefficients is to reintroduce the base learners. Let's now see that the claim is indeed true. We can equivalently show that the *weighted agreement* relative to $\tilde{W}_m(t)$ is exactly zero:

$$\sum_{t=1}^{m} \tilde{W}_m(t) y_t h(\mathbf{x}_t; \hat{\theta}_m) = 0 \tag{6}$$

Consider the optimization problem for α_m after we have already found $\hat{\theta}_m$:

$$J(\alpha_m, \hat{\theta}_m) = \sum_{t=1}^{n} \tilde{W}_{m-1}(t) \exp\Big(-y_t \alpha_m h(\mathbf{x}_t; \hat{\theta}_m) \Big) \tag{7}$$

The derivative of $J(\alpha_m, \hat{\theta}_m)$ with respect to α_m has to be zero at the optimal value $\hat{\alpha}_m$ so that

$$\begin{aligned} \frac{d}{d\alpha_m} J(\alpha_m, \hat{\theta}_m)\big|_{\alpha_m = \hat{\alpha}_m} &= -\sum_{t=1}^{n} \tilde{W}_{m-1}(t) \exp\Big(-y_t \hat{\alpha}_m h(\mathbf{x}_t; \hat{\theta}_m) \Big) y_t h(\mathbf{x}_t; \hat{\theta}_m) && (8) \\ &= -c_m \sum_{t=1}^{n} \tilde{W}_m(t) y_t h(\mathbf{x}_t; \hat{\theta}_m) = 0 && (9) \end{aligned}$$

where we have used Eq.(5) to move from $\tilde{W}_{m-1}(t)$ to $\tilde{W}_m(t)$. So the result is an optimality condition for α_m.

Ensemble training error. The training error of the ensemble does not necessarily decrease monotonically with each boosting iteration. The exponential loss of the ensemble does, however, decrease monotonically. This should be evident since it is exactly the loss we are sequentially minimizing by adding the base learners. We can also quantify, based on the weighted error achieved by each base learner, how much the exponential loss decreases after each iteration. We will need this to relate the training loss to the classification error. In fact, the amount that the training loss decreases after iteration m is exactly c_m, the normalization constant for the updated weights (we have to normalize the weights precisely because the exponential loss over the training examples decreases). Note also that c_m is exactly $J(\hat{\alpha}_m, \hat{\theta}_m)$. Now,

$$\begin{aligned} J(\hat{\alpha}_m, \hat{\theta}_m) &= \sum_{t=1}^{n} \tilde{W}_{m-1}(t) \exp\Big(-y_t \hat{\alpha}_m h(\mathbf{x}_t; \hat{\theta}_m) \Big) && (10) \\ &= \sum_{t:\, y_t = h(\mathbf{x}_t; \hat{\theta}_m)} \tilde{W}_{m-1}(t) \exp\left(-\hat{\alpha}_m \right) + \sum_{t:\, y_t \neq h(\mathbf{x}_t; \hat{\theta}_m)} \tilde{W}_{m-1}(t) \exp\left(\hat{\alpha}_m \right) && (11) \\ &= (1 - \hat{\epsilon}_m) \exp\left(-\hat{\alpha}_m \right) + \hat{\epsilon}_m \exp\left(\hat{\alpha}_m \right) && (12) \\ &= (1 - \hat{\epsilon}_m) \sqrt{\frac{\hat{\epsilon}_m}{1 - \hat{\epsilon}_m}} + \hat{\epsilon}_m \sqrt{\frac{1 - \hat{\epsilon}_m}{\hat{\epsilon}_m}} && (13) \\ &= 2\sqrt{\hat{\epsilon}_m (1 - \hat{\epsilon}_m)} && (14) \end{aligned}$$

Note that this is always less than one for any $\hat{\epsilon}_m < 1/2$. The training loss of the ensemble after m boosting iterations is exactly the product of these terms (renormalizations). In other words,

$$\sum_{t=1}^{m} \exp\left(\, -y_t h_m(\mathbf{x}_t)\,\right) = \prod_{k=1}^{m} 2\sqrt{\hat{\epsilon}_k(1-\hat{\epsilon}_k)} \tag{15}$$

This and the observation that

$$\text{step}(z) \leq \exp(z) \tag{16}$$

for all z, where the step function $\text{step}(z) = 1$ if $z > 0$ and zero otherwise, suffices for our purposes. A simple upper bound on the training error of the ensemble, $\text{err}_n(h_m)$, follows from

$$\begin{aligned} \text{err}_n(h_m) &= \frac{1}{n}\sum_{t=1}^{n} \text{step}\left(\, -y_t h_m(\mathbf{x}_t)\,\right) &(17)\\ &\leq \frac{1}{n}\sum_{t=1}^{n} \exp\left(\, -y_t h_m(\mathbf{x}_t)\,\right) &(18)\\ &= \frac{1}{n}\prod_{k=1}^{m} 2\sqrt{\hat{\epsilon}_k(1-\hat{\epsilon}_k)} &(19) \end{aligned}$$

Thus the exponential loss over the training examples is an upper bound on the training error and this upper bound goes down monotonically with m provided that the base learners are learning something at each iteration (their weighted errors less than half). Figure 3 shows the training error as well as the upper bound as a function of the boosting iterations.

Ensemble test error. We have so far discussed only training errors. The goal, of course, is to generalize well. What can we say about the generalization error of ensemble generated by the boosting algorithm? We have repeatedly tied the generalization error to some notion of margin. The same is true here. Consider figure 5 below. It shows a typical plot of the ensemble training error and the corresponding generalization error as a function of boosting iterations. Two things are notable in the plot. First, the generalization error seems to decrease (slightly) even after the ensemble has reached zero training error. Why should this be? The second surprising thing seems to be the fact that the generalization error does not increase even after a large number of boosting iterations. In other words, the boosting algorithm appears to be somewhat resistant to overfitting. Let's try to explain these two (related) observations.

The votes $\{\hat{\alpha}_j\}$ generated by the boosting algorithm won't sum to one. We will therefore renormalize ensemble

$$\tilde{h}_m(\mathbf{x}) = \frac{\hat{\alpha}_1 h(\mathbf{x};\hat{\theta}_1) + \ldots \hat{\alpha}_m h(\mathbf{x};\hat{\theta}_m)}{\hat{\alpha}_1 + \ldots + \hat{\alpha}_m} \tag{20}$$

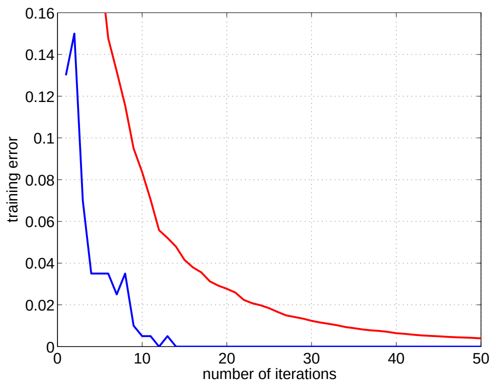

Figure 3: The training error of the ensemble as well as the corresponding exponential loss (upper bound) as a function of the boosting iterations.

so that $\tilde{h}_m(\mathbf{x}) \in [-1, 1]$. As a result, we can define a "voting margin" for training examples as margin$(t) = y_t \tilde{h}_m(\mathbf{x}_t)$. The margin is positive if the example is classified correctly by the ensemble. It represents the degree to which the base classifiers agree with the correct classification decision (negative value indicates disagreement). Note that margin$(t) \in [-1, 1]$. It is a very different type of margin (voting margin) than the geometric margin we have discussed in the context linear classifiers. Now, in addition to the training error $\text{err}_n(h_m)$ we can define a margin error $\text{err}_n(h_m; \rho)$ that is the fraction of example margins that are at or below the threshold ρ. Clearly, $\text{err}_n(h_m) = \text{err}_n(h_m; 0)$. We now claim that the boosting algorithm, even after $\text{err}_n(h_m; 0) = 0$ will decrease $\text{err}_n(h_m; \rho)$ for larger values of $\rho > 0$. Figure 4a-b provide an empirical illustration that this is indeed happening. This is perhaps easy to understand as a consequence of the fact that exponential loss, $\exp(-\text{margin}(t))$, decreases as a function of the margin, even after the margin is positive.

The second issue to explain is the apparent resistance to overfitting. One reason is that the complexity of the ensemble does not increase very quickly as a function of the number of base learners. We will make this statement more precise later on. Moreover, the boosting iterations modify the ensemble in sensible ways (increasing the margin) even after the training error is zero. We can also relate the margin, or the margin error $\text{err}_n(h_m; \rho)$ directly to generalization error. Another reason for resistance to overfitting is that the sequential procedure for optimizing the exponential loss is not very effective. We would overfit much more quickly if we reoptimized $\{\alpha_j\}$'s *jointly* rather than through the sequential procedure (see the discussion of boosting as gradient descent below).

Boosting as gradient descent. We can also view the boosting algorithm as a simple

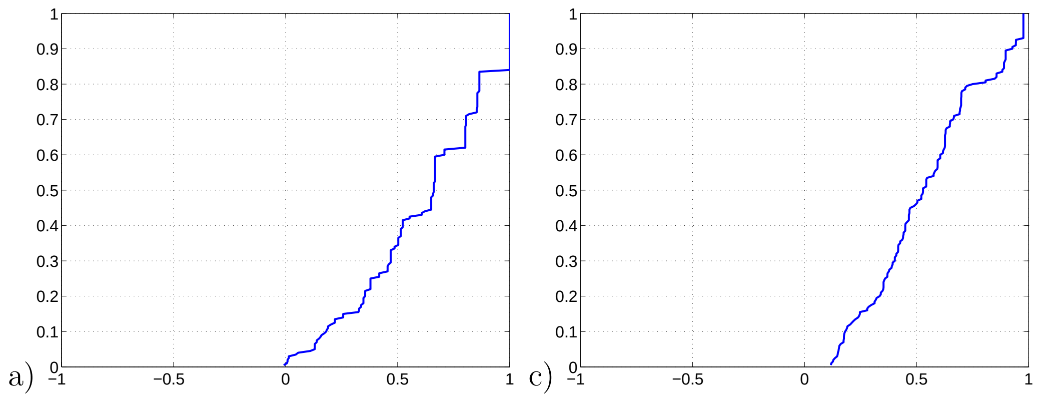

Figure 4: The margin errors $\text{err}_n(h_m;\rho)$ as a function of ρ when a) $m = 10$ b) $m = 50$.

gradient descent procedure (with line search) in the space of discriminant functions. To understand this we can view each base learner $h(\mathbf{x};\theta)$ as a vector based on evaluating it on each of the training examples:

$$\vec{h}(\theta) = \begin{bmatrix} h(\mathbf{x}_1;\theta) \\ \cdots \\ h(\mathbf{x}_n;\theta) \end{bmatrix} \tag{21}$$

The ensemble vector $\vec{h}_m$, obtained by evaluating $h_m(\mathbf{x})$ at each of the training examples, is a positive combination of the base learner vectors:

$$\vec{h}_m = \sum_{j=1}^{m} \hat{\alpha}_m \vec{h}(\hat{\theta}_m) \tag{22}$$

The exponential loss objective we are trying to minimize is now a function of the ensemble vector $\vec{h}_m$ and the training labels. Suppose we have $\vec{h}_{m-1}$. To minimize the objective, we can select a useful direction, $\vec{h}(\hat{\theta}_m)$, along which the objective seems to decrease. This is exactly how we derived the base learners. We can then find the minimum of the objective by moving in this direction, i.e., evaluating vectors of the form $\vec{h}_{m-1} + \alpha_m \vec{h}(\hat{\theta}_m)$. This is a line search operation. The minimum is attained at $\hat{\alpha}_m$, we obtain $\vec{h}_m = \vec{h}_{m-1} + \hat{\alpha}_m \vec{h}(\hat{\theta}_m)$, and the procedure can be repeated.

Viewing the boosting algorithm as a simple gradient descent procedure also helps us understand why it can overfit if we continue with the boosting iterations.

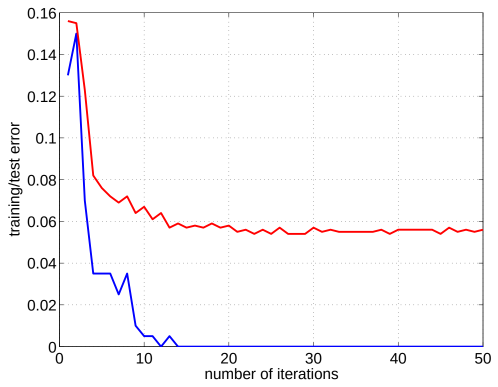

Figure 5: The training error of the ensemble as well as the corresponding generalization error as a function of boosting iterations.

Complexity and generalization

We have approached classification problems using linear classifiers, probabilistic classifiers, as well as ensemble methods. Our goal is to understand what type of performance guarantees we can give for such methods based on finite training data. This is a core theoretical question in machine learning. For this purpose we will need to understand in detail what "complexity" means in terms of classifiers. A single classifier is never complex or simple; complexity is a property of the set of classifiers or the model. Each model selection criterion we have encountered provided a slightly different definition of "model complexity".

Our focus here is on performance guarantees that will eventually relate the margin we can attain to the generalization error, especially for linear classifiers (geometric margin) and ensembles (voting margin). Let's start by motivating the complexity measure we need for this purpose with an example.

Consider a simple decision stump classifier restricted to x_1 coordinate of 2−dimensional input vectors $\mathbf{x} = [x_1, x_2]^T$. In other words, we consider stumps of the form

$$h(\mathbf{x}; \theta) = \text{sign}\,(\, s(x_1 - \theta_0)\,) \tag{23}$$

where $s \in \{-1, 1\}$ and $\theta_0 \in \mathcal{R}$ and call this set of classifiers $\mathcal{F}_1$. Example decision boundaries are displayed in Figure 6.

Suppose we are getting the data points in a sequence and we are interested in seeing when our predictions for future points become constrained by the labeled points we have already

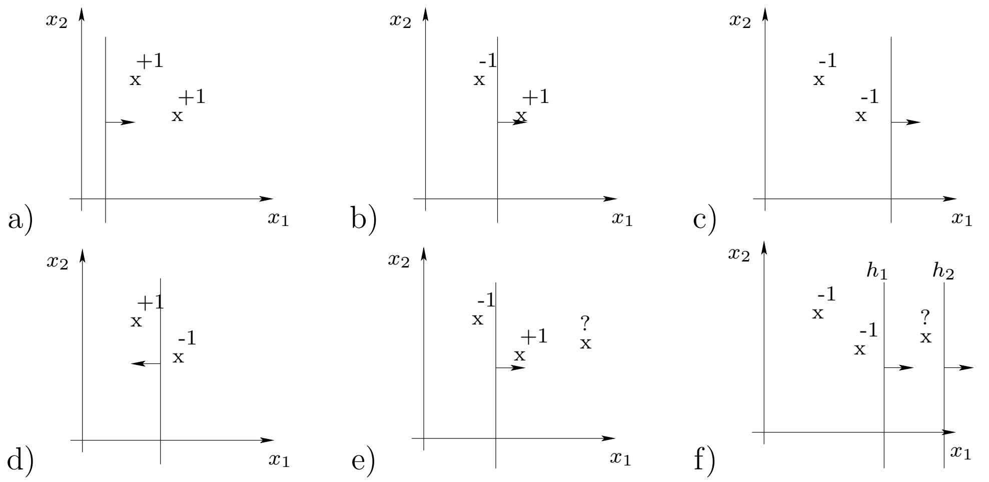

Figure 6: Possible decision boundaries corresponding to decision stumps that rely only on x_1 coordinate. The arrow (normal) to the boundary specifies the positive side.

seen. Such constraints pertain to both the data and the set of classifiers $\mathcal{F}_1$. See Figure 6e. Having seen the labels for the first two points, -1 and $+1$, all classifiers $h \in \mathcal{F}_1$ that are consistent with these two labeled points have to predict $+1$ for the next point. Since the labels we have seen force us to classify the new point in only one way, we can claim to have learned something. We can also understand this as a limitation of (the complexity of) our set of classifiers. Figure 6f illustrates an alternative scenario where we can find two classifiers $h_1 \in \mathcal{F}_1$ and $h_2 \in \mathcal{F}_1$, both consistent with the first two labels in the figure, but make different predictions for the new point. We could therefore classify the new point either way. Recall that this freedom is not available for all label assignments to the first two points. So, the stumps in $\mathcal{F}_1$ can classify any two points (in general position) in all possible ways (Figures 6a-d) but are already partially constrained in how they assign labels to three points (Figure 6e). In more technical terms we say that $\mathcal{F}_1$ *shatters* (can generate all possible labels over) two points but not three.

Similarly, for linear classifiers in $2-$dimensions, all the eight possible labelings of three points can be obtained with linear classifiers (Figure 7a). Thus linear classifiers in two dimensions shatter three points. However, there are labels over four points that no linear classifier can produce (Figure 7b).

VC-dimension. As we increase the number of data points, the set of classifiers we are considering may no longer be able to label the points in all possible ways. Such emerging constraints are critical to be able to predict labels for new points. This motivates a key

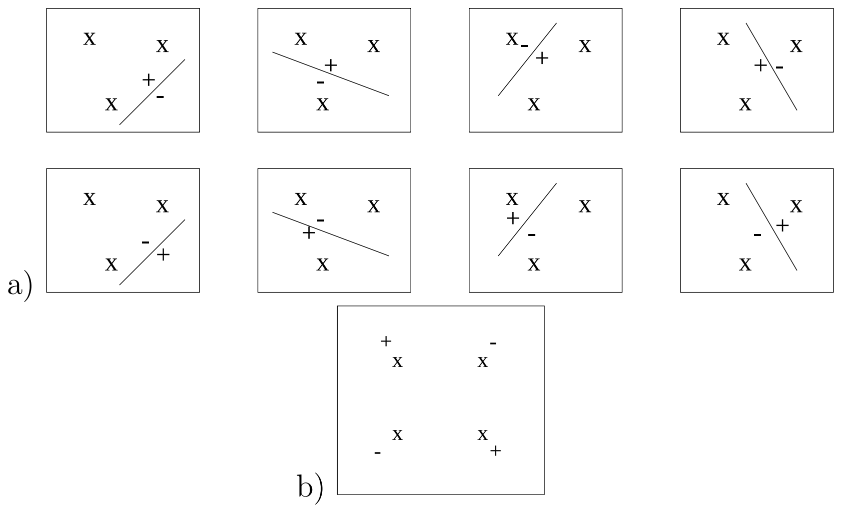

Figure 7: Linear classifiers on the plane can shatter three points a) but not four b).

measure of complexity of the set of classifiers, the *Vapnik-Chervonenkis dimension*. The VC-dimension is defined as the maximum number of points that a classifier can shatter. So, the VC-dimension of $\mathcal{F}_1$ is two, denoted as $d_{VC}(\mathcal{F}_1)$, and the VC-dimension of linear classifiers on the plane is three. Note that the definition involves a maximum over the possible points. In other words, we may find less than d_{VC} points that the set of classifiers cannot shatter (e.g., linear classifiers with points exactly on a line in $2-d$) but there cannot be any set of more than d_{VC} points that the classifier can shatter.

The VC-dimension of the set of linear classifiers in $d-$dimensions is $d+1$, i.e., the number of parameters. This is not a useful result for understanding how kernel methods work. For example, the VC-dimension of linear classifiers using the radial basis kernel is ∞. We can incorporate the notion of margin in VC-dimension, however. This is known as the V_γ dimension. The V_γ dimension of a set of linear classifiers that attain geometric margin γ when examples lie within an enclosing sphere of radius R is bounded by R^2/γ^2. In other words there are not that many points we can label in all possible ways if any valid labeling has to be with margin γ. This result is independent of the dimension of the classifier, and is exactly the mistake bound for the perceptron algorithm!