Chapter 3

Parametric Learning and Perceptrons

In the previous chapter, we saw how a model can draw decision boundaries and make predictions simply by looking at the nearest neighbors of test points. The basic idea behind these *instance-based models* is that they can be *lazy* and just store the training data to define our decision boundary.

There are downsides to lazy, instance-based approaches such as k-NNs, however. One issue is that these approaches can be memory intensive. We need to store the entire training set in order to compute the nearest neighbors during inference (i.e., at test time), which can be quite memory intensive. Another issue is that instance-based approaches tend to have large time complexities for inference. If we assume that we have m-dimensional features and that it takes time $\mathcal{O}(m)$ to compute the distance between two points, then the inference cost for a k-NN model is $\mathcal{O}(mn)$, where n is the number of training points. In other words, we need to iterate over the entire training set every time we want to make a prediction!

The key issue with instance-based methods is that they pay a price for not doing a lot of work up front. Indeed, there is no cost for training these models; we simply store the training data. However, the cost we pay is that these lazy models are very expensive during inference.

At the other end of the spectrum—opposite from instance-based learning—we have *parametric models*, which can be classified as *eager* approaches. The key idea in eager (i.e., parametric) approaches is that we try to do a lot of work up front to summarize our training idea. In particular, our goal is to summarize our training data $\mathcal{D}_{\text{trn}}$ into a small set of parameters Θ, so that these Θ parameters define our decision boundary. Again, the key distinction is that instance-based models define the decision boundary *implicitly* based on the training data, whereas parametric models summarize the training data into an *explicit* parametric decision boundary.

3.1 Linear Decision Boundaries

Parametric models define their decision boundaries based on a set of parameters, Θ, which are learned from the training data. But what are these parameters? And what do these decision boundaries actually look like?

The simplest and by far the most popular kind of decision boundary is a *linear decision boundary.* In the case of binary classification between a positive class 1 and a negative class -1, a linear decision boundary means that the decision criteria is a linear function of the input features.[1] In other words, assuming that we have a m-dimensional feature vector $\mathbf{x}$ for each data point, we can write the prediction function for a basic linear model as

$$f_{\text{LM}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} + b > 0 \\ -1 & \text{if } \mathbf{w}^\top \mathbf{x} + b < 0 \\ \text{undefined} & \text{otherwise} \end{cases} \tag{3.1}$$

$$= \text{sign}\left(\mathbf{w}^\top \mathbf{x} + b\right), \tag{3.2}$$

which means that we classify the point as positive if the expression $\mathbf{w}^\top \mathbf{x}_i + b$ is positive and negative if the expression is negative. (The predictions for points right on the decision boundary are undefined.) Note that the *parameter vector* $\mathbf{w}$ and the *bias term* b are the parameters of our linear model (i.e., $\Theta = \{\mathbf{w}, b\}$).

Dot products, coefficients, and bias terms In Equation 3.1, we used dot-product vector notation to specify the linear decision boundary. We can expand this notation to as follows:

$$\mathbf{w}^\top \mathbf{x} + b = \mathbf{w}[0]\mathbf{x}[0] + \mathbf{w}[1]\mathbf{x}[1] + ... + \mathbf{w}[m-1]\mathbf{x}[m-1] + b, \tag{3.3}$$

where we array index notation to denote different entries in the m-dimensional vectors. In other words, we can use the dot product between a feature vector $\mathbf{x}$ and a parameter vector $\mathbf{w}$ as a shorthand for a linear function.

Sometimes we will refer to the individual entries in the parameter vector $\mathbf{w}[j]$ as *feature coefficients*, since each feature coefficient specifies the impact of a particular feature on the prediction outcome. For example, suppose we are classifying spam email and the fourth feature (i.e., $\mathbf{x}[4]$) encodes the number of spelling errors in the email. If we have a positive coefficient for this feature (i.e., $\mathbf{w}[4] > 0$), then this would mean that increasing the number of spelling errors in an email would increase our likelihood of classifying an email as spam.

The bias term b (which is also called the *intercept* term) is a parameter in our linear model that does not depend on the features. Intuitively, the bias term allows us to shift the overall predictions to be more positive or negative; e.g., a large negative bias term $b << 0$ would bias our model

[1] We use -1 to denote the negative class—rather than 0—in this chapter because it simplifies notation in many of our derivations. This change has no meaningful impact on the learning problem.

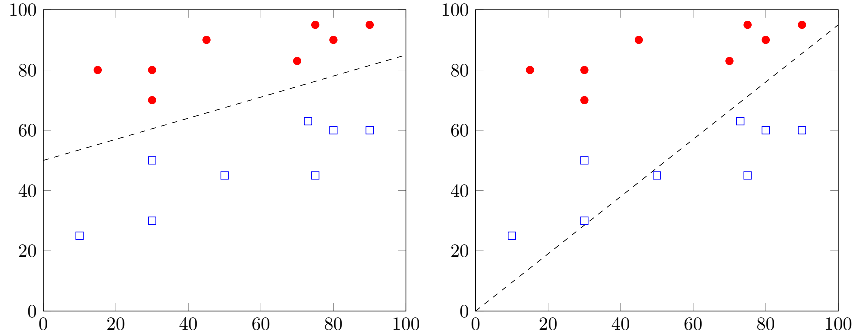

Figure 3.1: Example of dataset of positive points (red circles) and negative points (blue squares) that can be perfectly separated by a linear decision boundary. Note that having a bias term is essential to define a correct linear decision boundary (right figure). Without a bias term, the decision boundary must go through the origin, which makes perfect classification impossible on this dataset (right figure).

towards making negative predictions overall. Including a bias term is essential. For example, there are datasets that can be perfectly separable using a linear decision boundary, but where it is impossible to do so without a bias term (see Figure 3.1).

In geometric terms, a linear decision boundary means that the feature space is partitioned into two regions based on a linear boundary: a positive region and a negative region. In two dimensions, it is easy to visualize this idea as a line separating space into two regions (Figure 3.1). In higher dimensions, we must use a *hyperplane*, which generalizes the notion of a straight line, to separate the different regions (Figure 3.2).

Note that this kind of decision boundary is very different from the partitioning we obtained using k-nearest neighbors in the previous chapter. Whereas k-NNs partitioned space into regions around each training point, the linear decision boundary approach summarizes all the training data into a single decision boundary, specified by the feature coefficients $\mathbf{w}$ and the intercept term b.

3.2 Perceptron Learning Algorithm

In the previous section, we introduced the idea of making predictions based on a linear decision boundary. But how can we actually learn this decision boundary? In this section, we will introduce the simplest parametric learning algorithm: *the perceptron algorithm*.

[2] Image credit: http://www.cs.cornell.edu/courses/cs4758/2012sp/materials/

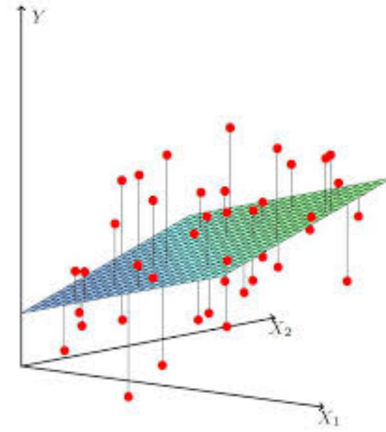

Figure 3.2: Visualization of a hyperplane in a 3D space.[2]

Intuition: Learning from mistakes

The basic intuition behind the perceptron algorithm is that we want to learn a decision boundary from our training data by iteratively correcting the predictions that our model makes.

Suppose we start with a random guess for our decision boundary; e.g., we start by guessing that $\mathbf{w} = \mathbf{1}$ and that $b = 1$. If we use this naive decision boundary to make a prediction on a training point $(\mathbf{x}_i, y_i)$, then we will likely end up making an incorrect prediction. In particular, we will have that

$$\text{sign}(\mathbf{w}^\top \mathbf{x}_i + b) \neq y_i, \tag{3.4}$$

which is equivalent to the having that

$$y_i(\mathbf{w}^\top \mathbf{x}_i + b) < 0, \tag{3.5}$$

i.e., an incorrect prediction implies that the sign of the prediction and true value are opposite.

Given that we have made such an incorrect prediction, how can we improve our model? The logic behind the perceptron algorithm is as follows.

- If we incorrectly classify a negative training point as positive, then we need to shift our weights to make the prediction more negative for that point.
- If we incorrectly classify a positive training point as negative, then we need to shift our weights to make the prediction more positive for that point.
- If we correctly classify a training point, then we should keep our parameters as they are.

One simple way to guarantee these properties is to update our model on incorrect classifications using the rules

$$\mathbf{w}^{(\text{new})} = \mathbf{w}^{(\text{old})} + y_i \mathbf{x}_i \tag{3.6}$$

$$b^{(\text{new})} = b^{(\text{old})} + y_i. \tag{3.7}$$

If we do such an update, we can guarantee that

$$y_i \left(\left(\mathbf{w}^{(\text{new})} \right)^\top \mathbf{x}_i + b^{(\text{new})} \right) \geq y_i \left(\left(\mathbf{w}^{(\text{old})} \right)^\top \mathbf{x}_i + b^{(\text{old})} \right), \tag{3.8}$$

which means that our model is never getting worse (i.e., always monotonically improving) on this training point after such an update. Intuitively, we can think that the update moves our parameters in the direction so that the prediction and target have the same sign. This improvement guarantee follows from the fact that

$$\begin{aligned} y_i \left(\left(\mathbf{w}^{(\text{new})} \right)^\top \mathbf{x}_i + b^{(\text{new})} \right) &= y_i \left((\mathbf{w}^{(\text{old})} + \mathbf{x}_i y_i)^\top \mathbf{x}_i + (b^{(\text{old})} + y_i) \right) \\ &= y_i \left(\left(\mathbf{w}^{(\text{old})} \right)^\top \mathbf{x}_i + b^{(\text{old})} + \mathbf{x}_i^\top \mathbf{x}_i y_i + y_i \right) \\ &= y_i \left(\left(\mathbf{w}^{(\text{old})} \right)^\top \mathbf{x}_i + b^{(\text{old})} \right) + \mathbf{x}_i^\top \mathbf{x}_i y_i^2 + y_i^2 \\ &\geq y_i \left(\left(\mathbf{w}^{(\text{old})} \right)^\top \mathbf{x}_i + b^{(\text{old})} \right), \end{aligned} \tag{3.9}$$

where the last inequality holds due to the fact that $\mathbf{x}^\top \mathbf{x}$ and y_i^2 are always positive. Note that if we want to further divide the y_i term from the inequality in Equation 3.9 then we need to account for the fact that the the inequality would flip in the case where y_i is negative, which gives rise to the natural outcome that

$$\begin{cases} \left(\left(\mathbf{w}^{(\text{new})} \right)^\top \mathbf{x}_i + b^{(\text{new})} \right) \geq \left(\left(\mathbf{w}^{(\text{old})} \right)^\top \mathbf{x}_i + b^{(\text{old})} \right) & \text{if } y_i = 1 \\ \left(\left(\mathbf{w}^{(\text{new})} \right)^\top \mathbf{x}_i + b^{(\text{new})} \right) \leq \left(\left(\mathbf{w}^{(\text{old})} \right)^\top \mathbf{x}_i + b^{(\text{old})} \right) & \text{if } y_i = -1. \end{cases} \tag{3.10}$$

In other words, our prediction increases if the label is positive and decreases if the label is negative.

Simplifying notation for the intercept term You may have noticed that the notation for the linear model can become a bit cumbersome, since we need to keep track of both the parameter vector $\mathbf{w}$ as well as the intercept term b. Indeed, for this reason it is actually quite common for researchers to simply ignore the intercept term in their mathematical expressions.

Formally speaking, we do not simply ignore the intercept term. Instead, we can assume that the data has a *dummy feature* added to account for the

intercept. To do this we simply add a new *constant* feature to all our training points, i.e., every training point has a dummy feature $\mathbf{x}[0] = 1$ prepended to its feature vector. Assuming that we have added this dummy feature, we do not need to explicitly add a intercept term to the model, since the coefficient $\mathbf{w}[0]$ now plays this role, i.e.,

$$\begin{aligned}\mathbf{w}^\top \mathbf{x} &= \mathbf{w}[0]\mathbf{x}[0] + \mathbf{w}[1]\mathbf{x}[1] + ... + \mathbf{w}[m]\mathbf{x}[m] \\ &= \mathbf{w}[0] + \mathbf{w}[1]\mathbf{x}[1] + ... + \mathbf{w}[m]\mathbf{x}[m],\end{aligned}$$

since $\mathbf{x}[0] = 1$ by design for all our training points. Note that adding this dummy feature means that the dimension of our parameter vector is now $m + 1$, assuming that our original dataset had m features. **Unless otherwise specified, it is always safe to assume that a dummy feature has been added to the data in order to account for the intercept term.** Of course, sometimes we will explicitly write out the intercept term (when it is useful for calculations), but we will much more often use the simplification discussed above.

Formalizing the perceptron algorithm

The idea of correcting the models prediction on a single training point can be extended to a general training algorithm. Instead simply making an update on a single training point, we loop over the entire training data and iteratively update our model. The full perceptron algorithm is summarized below. (Following the discussion above, we omit the explicit intercept term for notational simplicity.)

Algorithm 1: Basic Perceptron Algorithm

input : training set $\mathcal{D}_{\text{trn}} = \{(\mathbf{x}_i, y_i), i = 0, ..., n-1\}$;
max updates K
output: Parameters of a linear decision boundary $\Theta = \{\mathbf{w}\}$
$\mathbf{w}^{(0)} = \mathbf{0}$
$misclassifiedPoint = True$
$k = 0$
while *misclassifiedPoint* ***and*** $k < K$ **do**
  $misclassifiedPoint = False$
  for $i = 0$ **to** $n - 1$ **do**
    if $y_i(\mathbf{w}^{(k)})^\top \mathbf{x}_i \leq 0$ **then**
      $misclassifiedPoint = True$
      $\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + y_i \mathbf{x}_i$ `// update parameters`
      k = k + 1
      break
    end
  end
end
Return $\mathbf{w}^{(k)}$

As an input to this algorithm, we must provide the training data, as well as one algorithm *hyperparameter*. In the case of the perceptron algorithm, our single hyperparameter is the maximum number of updates we should do before stopping the algorithm. This parameter ensures that the algorithm does not end up in an infinite loop and allows us to specify an upper bound on the computation time.

3.3 Analyzing the Perceptron Algorithm

One important attribute of the perceptron algorithm is that it comes with some convergence guarantees, and this convergence analysis also reveals interesting perspectives on how hard it is to separate a particular dataset using a linear decision boundary.

Linear separability

Several times throughout this chapter, we have mentioned the idea of "separating" a dataset using a linear decision boundary. Here, we formalize this notion.

Definition 1. *A dataset $\mathcal{D}$ is linearly separable if and only if there there exists some $\gamma > 0$ and some set of parameters $\mathbf{w}^*$ such that*

$$y_i(\mathbf{w}^*)^\top \mathbf{x}_i \geq \gamma \tag{3.11}$$

for all $(\mathbf{x}_i, y_i) \in \mathcal{D}$

Intuitively, a dataset is linearly separable if there exists a linear decision boundary that can correctly classify all the points in the datasets. Of course, not all datasets are linearly separable, and many datasets require *non-linear decision boundaries* (see Figure 3.3 for an illustration).

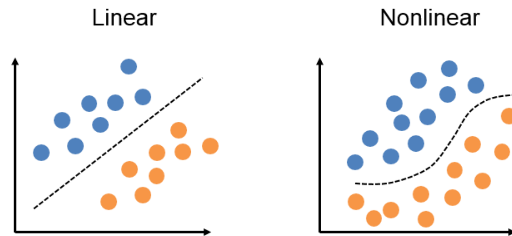

Figure 3.3: Illustration of linear and non-linear decision boundaries.[3]

[3]Image credit: https://jtsulliv.github.io/perceptron/

Perceptron convergence theorem

A key property of the perceptron algorithm is that it is guaranteed to converge in a finite number of iterations *on linearly separable data*. Note that by *converged*, we mean that the algorithm is able to correctly classify all points in the training set and reach the stopping condition for the while loop *before* timing-out due to the maximum iterations hyperparameter. In other words, if we have a linearly separable training dataset $\mathcal{D}_{\text{trn}}$, then we can guarantee that the perceptron algorithm will be able to perfectly classify this dataset after running for a finite number of iterations. However, if the dataset is not linearly separable, then we cannot make any guarantees.

Formally, we can prove the following theorem.

Theorem 1. *Assume that there exists some $\gamma > 0$ and some set of optimal parameters $\mathbf{w}^*$ such that $y_i(\mathbf{w}^*)^\top \mathbf{x}_i \geq \gamma$ for all $(\mathbf{x}_i, y_i) \in \mathcal{D}_{trn}$ then there exists some finite k^* such that perceptron algorithm will converge after k^* parameter updates.*

Proof. The proof of Theorem 1 relies on two lemmas.

Lemma 1. *The inner product $(\mathbf{w}^*)^\top \mathbf{w}^{(k)}$ increases at least linearly with each update. In particular, we have that $(\mathbf{w}^*)^\top \mathbf{w}^{(k)} \geq \gamma k$, where k denotes the number of updates in Algorithm 1 and γ and $\mathbf{w}^*$ are defined as in Theorem 1.*

Lemma 2. *The squared norm of the weight vector $\|\mathbf{w}^{(k)}\|^2$ increases at most linearly with each update. In particular, if we assume that $\|\mathbf{x}_i\| < R, \forall i \in \mathcal{D}_{trn}$, then $\|\mathbf{w}^{(k)}\|^2 \leq R^2 k$, where k denotes the number of updates in Algorithm 1 and $\mathbf{w}^*$ is defined as in Theorem 1.*

We defer the proof of these lemmas as exercises to the student (but note that solutions will eventually be provided). Given these two lemmas, we can prove Theorem 1 by showing that the cosine of the angle between $\mathbf{w}^{(k)}$ and $\mathbf{w}^*$ must increase with each update. In particular,

$$\begin{aligned}
\cos(\mathbf{w}^{(k)}, \mathbf{w}^*) &= \frac{(\mathbf{w}^{(k)})^\top \mathbf{w}^*}{\|\mathbf{w}^{(k)}\| \|\mathbf{w}^*\|} && (3.12) \\
&\geq \frac{k\gamma}{\|\mathbf{w}^{(k)}\| \|\mathbf{w}^*\|} \qquad \text{(by Lemma 1)} && (3.13) \\
&\geq \frac{k\gamma}{\sqrt{kR^2} \|\mathbf{w}^*\|} \qquad \text{(by Lemma 2)} && (3.14)
\end{aligned}$$

Thus, we know that:

1. The cosine between the updated weight vector $\mathbf{w}^{(k)}$ and the separating weight vector $\mathbf{w}^*$ must increase by a finite amount at each update (Equation 3.14).
2. Cosines are bounded above by 1 (by definition).

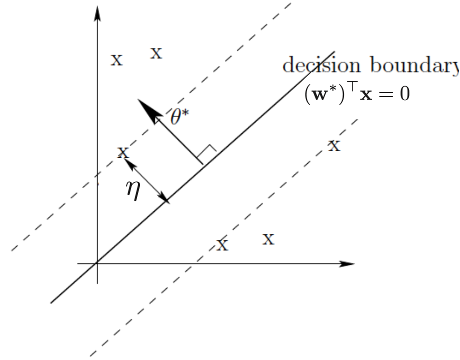

Figure 3.4: Visual illustration of the geometric margin.[4]

Together, these two facts imply that the algorithm must converge in a finite number of updates. The cosine must hit the upper bound of 1 after a finite number of updates. Moreover, we can bound the number of iterations required for convergence by combining Equation 3.14 and the fact that cosine is bounded by 1:

$$1 \geq \frac{k\gamma}{\sqrt{kR^2}\|\mathbf{w}^*\|} \Leftrightarrow k \leq \frac{R^2\|\mathbf{w}^*\|^2}{\gamma^2} \tag{3.15}$$

□

The geometric margin and the difficulty of classification

One interesting side result in the proof of Theorem 1 is the fact that we can bound the number of iterations required for the perceptron algorithm to converge (Equation 3.15). We found that the number of iterations is inversely proportionally to the ratio $\frac{\gamma}{\|\mathbf{w}^*\|}$, which suggests that this ratio might somehow relate to the difficulty of the classification task. In fact, we can interpret this ratio as specifying the distance between the separating hyperplane and the closest point to this hyperplane. We will call this ratio the geometric margin, denoted by $\eta = \frac{\gamma}{\|\mathbf{w}^*\|}$.

The key insight is that η gives the distance between the decision boundary and the points $\mathbf{x}_i$ that are closest to the decision boundary. These points are the points x_i where $\mathbf{w}^\top \mathbf{x}_i = \gamma$, meaning that the inequality in Equation 3.11 reaches its lower bound.

The geometric interpretation of $\eta = \frac{\gamma}{\|\mathbf{w}^*\|}$ holds based on the following reasoning. First, we know that $\mathbf{w}^*$ specifies the *normal* to the decision boundary, since the expression $(\mathbf{w}^*)^\top \mathbf{x} = 0$ defines the separating hyperplane. Now, given a point $\mathbf{x}_i$ that satisfies $\mathbf{w}^\top \mathbf{x}_i = \gamma$, we can compute the distance between this point and the decision boundary by drawing a line that (a) goes between this

[4]Figure adapted from Tommi Jaakkola's course materials for 6.867 Machine Learning, Fall 2006.

point and the decision boundary and (b) is parallel to $\mathbf{w}^*$. We will use $\mathbf{x}_\gamma$ to denote the point where this line intersects the decision boundary, and η will correspond to the length of this line. Now, we have that

$$\mathbf{x}_\gamma = \mathbf{x}_i - \frac{\eta \mathbf{w}^*}{\|\mathbf{w}^*\|} \tag{3.16}$$

because subtracting $\frac{\eta \mathbf{w}^*}{\|\mathbf{w}^*\|}$ translates us by a distance of η towards the decision boundary (since $\frac{\mathbf{w}^*}{\|\mathbf{w}^*\|}$ is a unit vector normal to the decision boundary). Finally, we have assumed that $\mathbf{x}_\gamma$ is on the decision boundary, which implies that $\mathbf{x}_\gamma^\top \mathbf{w}^* = 0$. Taking these facts together, we can solve for η:

$$\mathbf{x}_\gamma^\top \mathbf{w}^* = 0 \tag{3.17}$$

$$\left(\mathbf{x}_i - \frac{\eta \mathbf{w}^*}{\|\mathbf{w}^*\|}\right)^\top \mathbf{w}^* = 0 \tag{3.18}$$

$$\mathbf{x}_i^\top \mathbf{w}^* - \eta \frac{(\mathbf{w}^*)^\top \mathbf{w}^*}{\|\mathbf{w}^*\|} = 0 \tag{3.19}$$

$$\gamma - \eta \frac{\|\mathbf{w}^*\|^2}{\|\mathbf{w}^*\|} = 0 \tag{3.20}$$

$$\eta = \frac{\gamma}{\|\mathbf{w}^*\|} \tag{3.21}$$

Thus, $\eta = \frac{\gamma}{\|\mathbf{w}^*\|}$ can be geometrically interpreted as the distance between the decision boundary and the closest point to this decision boundary. This geometric margin determines the upper bound on the number of iterations we need to find a separating hyperplane using the perceptron algorithm and gives a natural notion of difficulty for a classification. Many more sophisticated notions of classification difficulty in machine learning build upon this idea.

Convergence and generalization

There are some important points to caveat regarding the above analysis. First, it is important to note that all our analysis above was with regards to the *training data*. We showed that the algorithm will converge on a linearly separable training set, but this does not guarantee that we will perfectly classify *test examples*. We can, however, ensure generalization in an *infinite data* scenario. In particular, suppose we have an infinite source or stream of data points $\mathbf{x}_i$, all of which are assumed to satisfy $(\mathbf{w}^*)^\top \mathbf{x}_i \geq \gamma$ for some optimal hyperplane $\mathbf{w}^*$ and have bounded norm $\|\mathbf{x}_i\| \leq R$. The perceptron convergence theorem guarantees that under these assumptions, we will only need to update our model a finite number of times, after which it will correctly classify any point it is given.

Convergence, uniqueness, and support vector machines

A second important caveat is the issue of uniqueness. In all our discussions, we assumed that there is *some* optimal parameter vector $\mathbf{w}^*$ and that we will converge to *some* separating hyperplane in a finite number of iterations. However,

there may be an infinite number of such separating hyperplanes, and we can never guarantee that we will converge to a particular, unique value. This is a fundamental issue in the perceptron approach.

In fact, there is an entire branch of machine learning dedicated to understanding how we can select the *best* separating hyperplane, under the assumption that we have many different choices. The idea of selecting the best hyperplane is the key goal behind machine learning models such as *support vector machines (SVMs)*, which, for example, try to find a separating hyperplane with the largest possible geometric margin. The details of such approaches rely heavily on advanced optimization techniques and are beyond the mathematical scope of this course.