GHENT UNIVERSITY

Lecture Notes

E016350: Artificial Intelligence

Logistic Regression and Optimization Algorithms

Aleksandra Pizurica

Spring 2025

Contents

Disclaimer: *These lecture notes were written by Prof. Aleksandra Pizurica to accompany the slides of the course E016350: Artificial Intelligence, facilitating their understanding. The lecture notes are not meant to be self-contained, and do not cover all the study material in the course. They are by no means meant to replace the recommended textbook and do not necessarily cover all the relevant aspects that are presented in the slides and explained in the lectures. Some sections are adapted from the book of S. Russel and P. Norvig: Artificial Intelligence: A Modern Approach.*

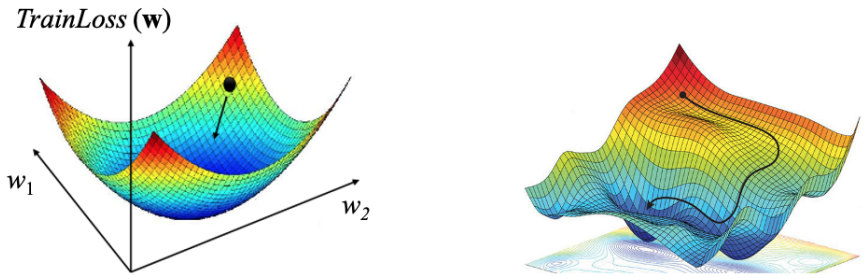


Figure 1: The optimization in ML aims at finding the weights that minimize the training loss. **Left**: A convex loss function (e.g., in the case of linear regression under the L_2 loss); **Right**: in general, the "loss landscape" is much more complex, non-convex with many local minima.

1 Optimization in machine learning

Our learning task is to determine the parameters (weights) $\mathbf{w}$ of a hypothesis $h_{\mathbf{w}}(\mathbf{x})$ that approximates the true, unknown function $y = f(\mathbf{x})$ that generated the data. We find the optimal $\mathbf{w}$ by minimizing the training loss

$$TrainLoss(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}) \tag{1}$$

where $L(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w})$ is some loss function. Formally, we solve a minimization problem:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} TrainLoss(\mathbf{w}) \tag{2}$$

This is typically done by applying some variant of the **gradient descent** algorithm.

When the loss function is convex (like in the left of Fig. 1), the gradient descent, unless applied with a very wrong step size, is guaranteed to find the global optimum. In general, the training loss will have a much more complex landscape with many local minima, especially in deep learning. The illustration on the right of Fig. 1 gives some idea about such more complex training loss functions with only two weights, since we cannot visualize higher dimensional ones. The gradient descent algorithm will in these cases likely end up in a local optimum, but its variants, like the so-called **stochastic gradient descent** will in practice find good solutions even for very complex loss functions.

1.1 Gradient descent algorithm

We can minimize an arbitrary loss function by applying **iterative optimization**. The idea is to start with some $\mathbf{w}$ and keep on tweaking it to make the loss go down until we reach the minimum. To make the best "move" in the weight space at each step, we can use the gradient of the function. The gradient of a scalar-valued differentiable function of several variables is the vector field whose value at each point gives the **direction and the rate** of the **fastest increase** of the function at that point. Hence, moving along the direction of the **negative gradient** decreases the loss function. This iterative optimization procedure is called **gradient descent**. If the goal of the optimization procedure is to maximize an objective function, then we move in the direction

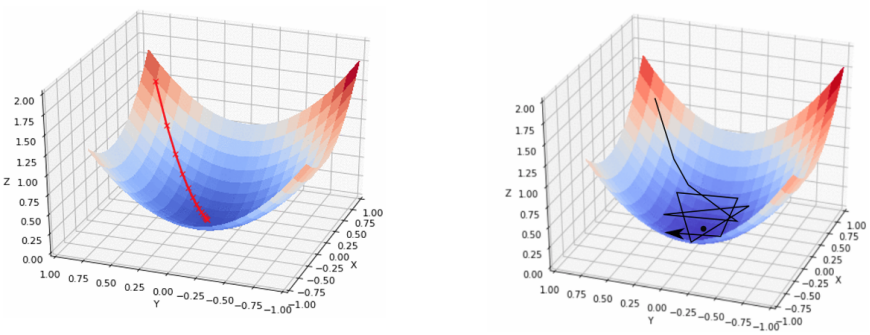


Figure 2: An illustartion of the gradient descent procedure with a good learning rate (left) and with a too large learning rate (right).

of the gradient to reach the maximum – this is known as the **gradient ascent** algorithm. We can use either of these two algorithms for the same problem if we can flip the objective function.

Thus, to minimize the training loss by the gradient descent, we will first initialize $\mathbf{w}$ to some value (say, all zeros) and then take a number of steps in the weight space, each time in the direction of the negative gradient. This means that we will each time subtract from $\mathbf{w}$ the gradient at that point $\nabla_{\mathbf{w}} TrainLoss(\mathbf{w})$ multiplied by some positive constant α that determines the step size. Concretely, the algorithm is as follows.

Algorithm: Gradient Descent (GD)

initialize $\mathbf{w} = [0, \ldots, 0]$

for iter $1, 2, \ldots$
$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} TrainLoss(\mathbf{w})$

Observe that in each iteration *all* the training data are used. Therefore, each iteration here is an **epoch**, the term which refers to *using all the training data at once*. The step size $\alpha \geq 0$, also called the **learning rate**, specifies how aggressively we want to pursue the descent direction. The step size and the number of epochs are two hyperparameters of the optimization algorithm.

The loss minimization by the gradient descent procedure is illustrated in Fig. 2. In the case where the learning rate is well chosen, the algorithm steadily steps towards the minimum, while with a too large learning rate it will take too large sweeps, therefore "overshooting" and possibly even completely failing to reach the optimum (see also an illustration in Fig. 3). Generally, larger steps sizes are like driving fast: you can get faster convergence, but you might also get very unstable results and "crash". On the other hand, smaller step sizes give more stability , but the destination is reached more slowly. Note that when $\alpha = 0$, the weights don't change.

Some general strategies for choosing the learning rate include:

- set α such that update changes of $\mathbf{w}$ are about 0.1–1%
- decreasing: start with $\alpha = 1$ and then let $\alpha = 1/\sqrt{\#\text{updates made so far}}$
- more sophisticated – adapt α based on the data
  - e.g., *AdaGrad* and *Adam* optimizer

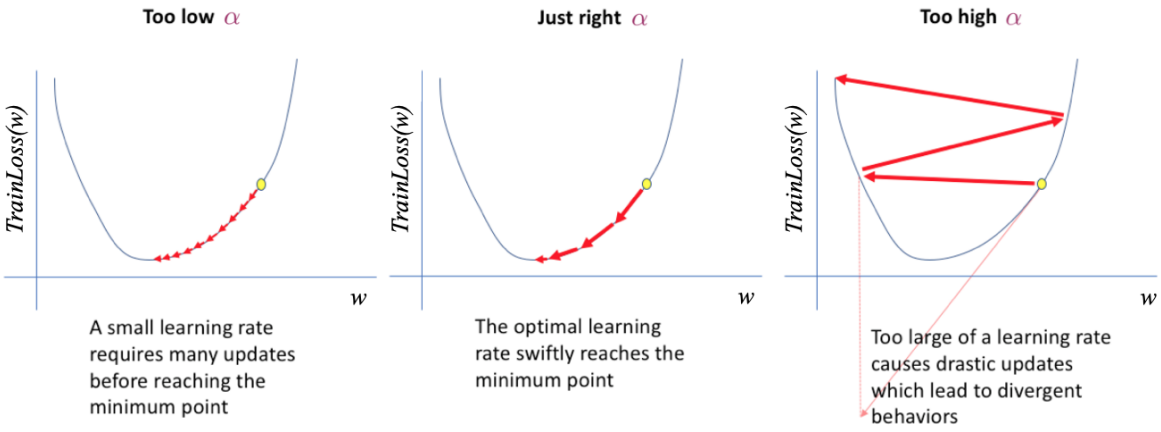


Figure 3: The influence of the learning rate. Illustration Credit: E. Duchesnay.

1.2 Stochastic gradient descent

While gradient descent is a powerful general-purpose algorithm to optimize the training loss, one problem with it is that it's very slow. It is because it requires in each step the gradient of the full training loss, and the training loss is a sum over all the training data, see Eq (1). Thus, if we have millions of the training examples, each gradient computation requires going through those millions of examples, before we can make any small update of the weights.

The natural question is then – *Can we make progress before seeing all the data?* The answer to this question is – *yes*: rather than looping through all the training examples to compute a single gradient, we can make an update of the weights based on **each** example. This way the procedure will be much less stable and we will need many more steps, but each of these steps will be very cheap! This method is called the **stochastic gradient descent** (**SGD**).

Algorithm: Stochastic Gradient Descent (SGD)

- init $\mathbf{w} = [0, \ldots, 0]$
- for iter $1, 2, \ldots$
  - For $(x, y) \in \mathcal{D}_{train}$:
    $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} L(\mathbf{x}, y, \mathbf{w})$

Each update now is not as good as with the (standard) gradient descent algorithm because we are only looking at one example at a time rather than taking all the examples. But the advantage is that each of these updates we compute very quickly so we can make many more steps this way.

There is a version between SGD and GD called **minibatch SGD**, where each update is made based on a **batch** of B examples. There are other variants of SGD. E.g., we can randomize the order in which we loop over the training data in each iteration. This is important, e.g., if in the training data we had all the positive examples first and the negative examples after that [1].

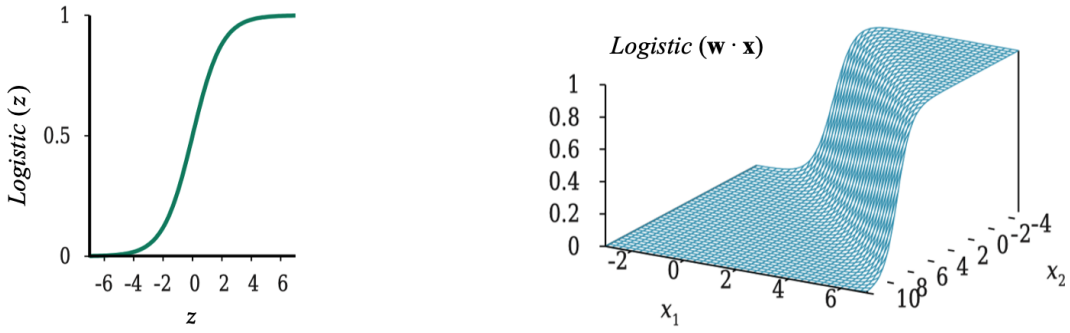


Figure 4: The logistic (sigmoid) function $Logistic(z) = 1/(1 + e^{-z})$ and an example of a logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x})$ for some weight vector $\mathbf{w} \in \mathbb{R}^2$. Figure from [4].

2 Logistic regression

Now we return to the task of binary classification. Previously we have seen that for some weight vector $\mathbf{w} \in \mathbb{R}^d$, the logistic regression hypothesis is

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}} = g(\mathbf{w} \cdot \mathbf{x}) \tag{3}$$

and we derived the update rule for the weights using (stochastic) gradient descent under the L_2 loss. Note that the loss function under the L_2 loss: $\sum_i (y^{(i)} - h_{\mathbf{w}}(\mathbf{x}^{(i)}))^2$ was convex for *linear* regression where $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$. But with the nonlinear logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x})$ this loss is nonconvex with many local minima. So, although we could derive the update rule for logistic regression under the L_2 loss, the optimization with the gradient descent will be difficult (gradient descent may not find the global optimum – it may get stuck in a local minimum).

2.1 Logistic loss

For the reasons explained above, we will rarely use the logistic regression with square-error loss, but rather with the so-called **logistic loss**:

$$L(h_{\mathbf{w}}(\mathbf{x}), y) = \begin{cases} -\log(h_{\mathbf{w}}(\mathbf{x})) & \text{if } \; y = 1 \\ -\log(1 - h_{\mathbf{w}}(\mathbf{x})) & \text{if } \; y = 0 \end{cases} \tag{4}$$

which has nice properties for optimization and which can also be derived using the principle of **maximum likelihood estimation** as we will show next.

The logistic loss is illustrated schematically in Fig. 5. Note that $h_{\mathbf{w}}(\mathbf{x})$ from Eq (3) is between 0 and 1 and the logistic loss is a monotonic decreasing function with respect to the hypothesis when $y = 1$, and monotonically increasing when $y = 0$. Moreover, the loss is exactly zero when we are 100% confident while making the correct hypothesis and tends to infinity when we are 100% confident while making the wrong hypothesis.

For binary classification with $y \in \{0, 1\}$, the logistic loss function from Eq (4) can be written more compactly as:

$$L(h_{\mathbf{w}}(\mathbf{x}), y) = -y \log(h_{\mathbf{w}}(\mathbf{x})) - (1 - y) \log(1 - h_{\mathbf{w}}(\mathbf{x})) \tag{5}$$

We will show now how we can derive this loss function using maximum-likelihood estimation.

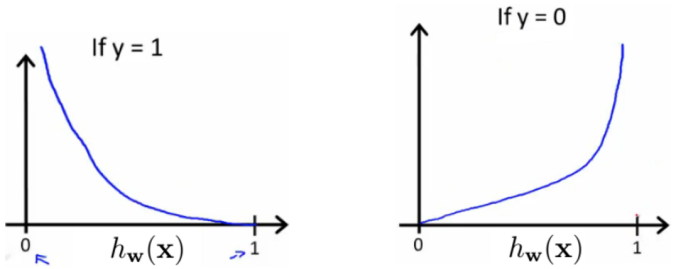


Figure 5: A schematic plot of the logistic loss from the machine learning course of Andrew Ng, with adapted notation.

2.2 Logistic regression under the maximum likelihood optimization

We already said earlier that the logistic regression $h_{\mathbf{w}}(\mathbf{x}) = \mathit{Logistic}(\mathbf{w} \cdot \mathbf{x})$ given in Eq (3) can be interpreted as the **probability** that $y = 1$. Let us now write this statement formally:

$$\begin{aligned} P(y = 1|\mathbf{x}, \mathbf{w}) &= h_{\mathbf{w}}(\mathbf{x}) \\ P(y = 0|\mathbf{x}, \mathbf{w}) &= 1 - h_{\mathbf{w}}(\mathbf{x}) \end{aligned} \tag{6}$$

Since y is always 1 or 0, we can write this more compactly as

$$P(y|\mathbf{x}, \mathbf{w}) = (h_{\mathbf{w}}(\mathbf{x}))^y (1 - h_{\mathbf{w}}(\mathbf{x}))^{(1-y)} \tag{7}$$

If the training examples were generated independently, the **likelihood** of the weights is:

$$\mathcal{L}(\mathbf{w}) = \prod_{i=1}^{N} P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) = \prod_{i=1}^{N} \left(h_{\mathbf{w}}(\mathbf{x}^{(i)}\right)^{y^{(i)}} \left(1 - h_{\mathbf{w}}(\mathbf{x}^{(i)}\right)^{1-y^{(i)}} \tag{8}$$

In the maximum-likelihood philosophy, the optimal weights are those that are most likely given the data, i.e., those that yield the maximum likelihood. It is easier to maximize the logarithm of this likelihood and it will yield exactly the same solution as maximizing the likelihood itself, since the logarithm is a monotonic function. Therefore, we express first the **log likelihood**:

$$\ell(\mathbf{w}) = \log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} y^{(i)} \log h_{\mathbf{w}}(\mathbf{x}^{(i)}) \; + \; (1 - y^{(i)}) \log(1 - h_{\mathbf{w}}(\mathbf{x}^{(i)})$$

Observe that this is in fact the *logistic loss* from Eq (5) which was there written for one example only.

Now we can determine the update rule for the logistic regression by maximizing the log-likelihood of the weights. This is **the most common form of the logistic regression**.

Note that now $\mathit{TrainLoss}(\mathbf{w}) = -\ell(\mathbf{w})$, so we are applying the gradient descent algorithm to $-\ell(\mathbf{w})$, or equivalently, we are applying the **gradient ascent** to $\ell(\mathbf{w})$:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \nabla_{\mathbf{w}} \ell(\mathbf{w}) \tag{9}$$

We start with **one** training example $(\mathbf{x}, y)$:

$$\begin{aligned}
\frac{\partial}{\partial w_j}\ell(\mathbf{w}) &= \Big(y\frac{1}{g(\mathbf{w}\cdot\mathbf{x})} - (1-y)\frac{1}{1-g(\mathbf{w}\cdot\mathbf{x})}\Big)\frac{\partial}{\partial w_j}g(\mathbf{w}\cdot\mathbf{x}) \\
&= \Big(y\frac{1}{g(\mathbf{w}\cdot\mathbf{x})} - (1-y)\frac{1}{1-g(\mathbf{w}\cdot\mathbf{x})}\Big)g(\mathbf{w}\cdot\mathbf{x})(1-g(\mathbf{w}\cdot\mathbf{x}))\frac{\partial}{\partial w_j}(\mathbf{w}\cdot\mathbf{x}) \\
&= (y(1-g(\mathbf{w}\cdot\mathbf{x})) \;-\; (1-y)g(\mathbf{w}\cdot\mathbf{x}))x_j \\
&= (y - h_{\mathbf{w}}(\mathbf{x}))x_j
\end{aligned}$$

In the derivation above, we used the fact that $g'(z) = g(z)(1-g(z))$. Hence, the maximum-likelihood update rule for the logistic regression, with one example, is

$$w_j \leftarrow w_j + \alpha(y - h_{\mathbf{w}}(\mathbf{x}))x_j$$

and with all training examples

$$w_j \leftarrow w_j + \alpha\sum_{i=1}^{N}(y^{(i)} - h_{\mathbf{w}}(\mathbf{x}^{(i)}))x_j^{(i)}$$

Note that this update looks exactly the same as for the least-squares linear regression but, of course, $h_{\mathbf{w}}$ is different. We followed here the derivation from [3], where you can find more details about the logistic regression, including an alternative algorithm for the maximization of $\ell(\mathbf{w})$.

3 Multiclass linear classification

So far we considered only binary linear classification. Now we turn to a more general case where we can have more than two classes. For example, we want to predict the value of a handwritten digit or to classify newspaper articles into categories culture, science, sports, politics etc. We still want to define the decision boundaries based on **linear functions** of the input, i.e., based on linear combinations of input features. This task is called **multiclass linear classification**.

Let our input be a d-dimensional vector as before $\mathbf{x} \in \mathbb{R}^d$. We now have a weight vector $\mathbf{w}_y \in \mathbb{R}^d$ for each output class $y \in \{1, \ldots, K\}$, and a new input $\mathbf{x}$ is classified based on the **scores** $\mathbf{w}_y \cdot \mathbf{x}$ that are computed for every class. We will put all the weight vectors together in one long vector $\mathbf{w} = [(\mathbf{w}_1)^\top, \ldots, (\mathbf{w}_K)^\top]^\top \in \mathbb{R}^{Kd}$ and we'll denote the prediction same as before by $h_{\mathbf{w}}(\mathbf{x})$.

3.1 Multiclass perceptron

Given the setup above, the prediction rule "the highest score wins":

$$h_{\mathbf{w}}(\mathbf{x}) = \arg\max_i \; \mathbf{w}_i \cdot \mathbf{x} \tag{10}$$

extends directly the linear binary classification with a **hard threshold** to multiple classes. This classification approach is illustrated in Fig. 6 and is often referred to as the **multiclass perceptron**. The scores $z_i = \mathbf{w}_i \cdot \mathbf{x}$ are also called **activations** (this is the terminology that we will use commonly with neural networks).

Often it is convenient to represent multiclass classification with **one-hot encoding**. This means that the target output (the correct classification result) is represented as a vector $\mathbf{t}$ with

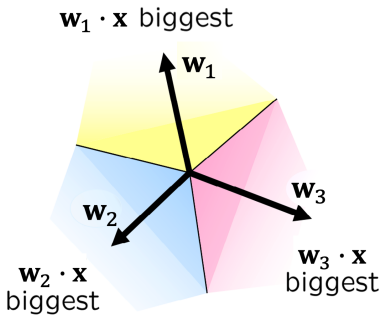


Figure 6: The concept of multiclass linear classification illustrated on a case with three classes. The input data point $\mathbf{x}$ is assigned to the class that gives the biggest score. Credit: D. Klein & P. Abbeel [2].

all zeroes except one entry "1", which indicates the correct class. For example, if the correct class out of K possibles classes is the k-th class, the one-hot encoded target output is

$$\mathbf{t} = \underbrace{[0, \ldots, 0, 1, 0, \ldots, 0]^\top}_{\text{entry } k \text{ is one}} \in \mathbb{R}^K$$

We can represent the multiclass perceptron prediction with one-hot encoding as follows. Let $\mathbf{o} \in \mathbb{R}^K$ be the one-hot encoded output vector. Then for the multiclass perceptron

$$o_k = \begin{cases} 1 & \text{if } k = \arg\max\limits_i \ \mathbf{w}_i \cdot \mathbf{x} \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

3.2 Multiclass logistic regression and the softmax rule

The question now is how to turn the *hard* multiclass classification that we defined above into a soft one. In the binary case we replaced the hard threshold with the sigmoid function and we called the resulting model logistic regression. The nice property of the sigmoid (logistic) function was that it provided a probabilistic interpretation of the output as the probability of belonging to class "1".

We can equivalently turn the scores for multiclass classification into probabilities for belonging to the corresponding classes by using the **softmax** rule:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{k=1}^K e^{z_k}}, \quad i = 1, \ldots, K \tag{12}$$

The original activations z_i are transformed this way to **softmax activations**. The resulting approach is **multiclass logistic regression** (also called **multinomial logistic regression** or **softmax regression**) where the hypothesis is defined as:

$$h_{\mathbf{w}}(\mathbf{x}) = \begin{bmatrix} P(y=1|\mathbf{x}, \mathbf{w}) \\ P(y=2|\mathbf{x}, \mathbf{w}) \\ \vdots \\ P(y=K|\mathbf{x}, \mathbf{w}) \end{bmatrix} = \frac{1}{\sum_{k=1}^K e^{\mathbf{w}_k \cdot \mathbf{x}}} \begin{bmatrix} e^{\mathbf{w}_1 \cdot \mathbf{x}} \\ e^{\mathbf{w}_2 \cdot \mathbf{x}} \\ \vdots \\ e^{\mathbf{w}_K \cdot \mathbf{x}} \end{bmatrix} \tag{13}$$

where $\mathbf{w} = [(\mathbf{w}_1)^\top, \ldots, (\mathbf{w}_K)^\top]^\top$. If we denote the output vector by $\mathbf{o} = h_\mathbf{w}(\mathbf{x})$ we can write

$$o_k = softmax(\mathbf{w}_k \cdot \mathbf{x}), \quad k = 1, \ldots, K \tag{14}$$

Note how this "softens" the hard classification rule in Eq (11).

4 Learning weights for multiclass linear classification

Let us now see how we learn the weights from the training data for the two above presented multiclass classification methods.

4.1 Multiclass perceptron learning rule

Remember the perceptron learning rule for binary classification with $y \in \{0, 1\}$, which can be written in a vector form as $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y - h_\mathbf{w}(\mathbf{x}))\mathbf{x}$. It did nothing if the output was correct, and otherwise the weights were either increased or decreased by $\alpha\mathbf{x}$ to nudge them in the right direction (increasing if $y = 1$ and $h_\mathbf{w}(\mathbf{x}) = 0$ and decreasing in $y = 0$ and $h_\mathbf{w}(\mathbf{x}) = 1$). This is simply extended to the case with multiple classes as follows:

- If $h_\mathbf{w}(\mathbf{x}) = y$ do nothing
- If $h_\mathbf{w}(\mathbf{x}) \neq y$ update the weights for the true class y and for the predicted class $y^* = h_\mathbf{w}(\mathbf{x})$
  - Update the **correct** class vector as $\mathbf{w}_y \leftarrow \mathbf{w}_y + \alpha\mathbf{x}$
  - Update the **wrong** class vector as $\mathbf{w}_{y^*} \leftarrow \mathbf{w}_{y^*} - \alpha\mathbf{x}$
  - Do **not** change the weights of any other class

4.2 Optimization for multiclass logistic regression

For multiclass logistic regression we optimize the weights similarly as we did with the logistic regression in the binary case: by maximizing the likelihood of the weights given the training data:

$$\mathbf{w}^* = \arg\max_\mathbf{w} \mathcal{L}(\mathbf{w})$$

Assuming as before that the training examples were generated independently, the likelihood is:

$$\mathcal{L}(\mathbf{w}) = \prod_{i=1}^{N} P(y^{(i)}|\mathbf{x}^{(i)}, \underbrace{\mathbf{w}_1, \ldots, \mathbf{w}_K}_{\mathbf{w}}) = \prod_{i=1}^{N} P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) \tag{15}$$

where

$$P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) = \frac{e^{\mathbf{w}_{y^{(i)}} \cdot \mathbf{x}^{(i)}}}{\sum_y e^{\mathbf{w}_{y^{(i)}} \cdot \mathbf{x}^{(i)}}}$$

Again, as was the case with the binary logistic regression, we will perform the desired optimization easier on =the logarithm of the likelihood:

$$\ell(\mathbf{w}) = \log \mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} \log P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) \tag{16}$$

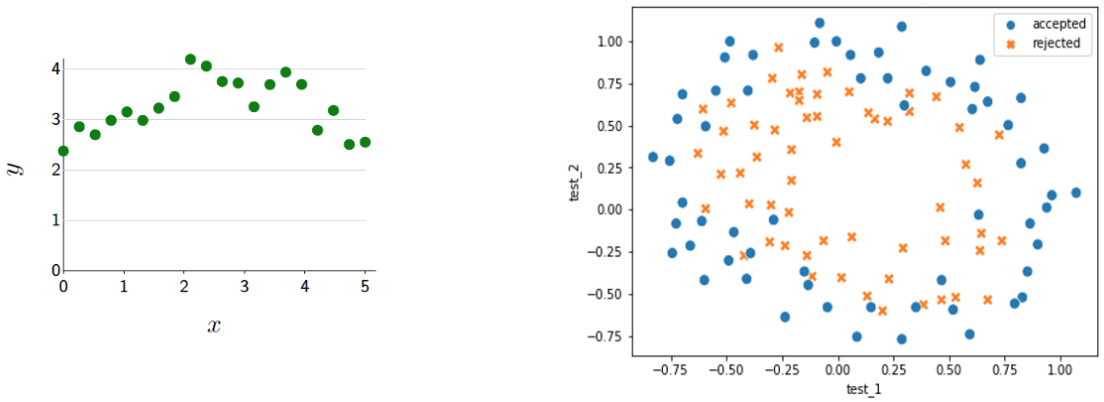


Figure 7: Examples of more complex data where a non-linear predictor is needed for regression (left) or classification (right). Figures from [1].

The optimization objective is now equivalently expressed as maximizing the likelihood or minimizing the negative log-likelihood, i.e., the training loss is now the negative log-likelihood and we have that:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \; -\ell(\mathbf{w}) = \arg\min_{\mathbf{w}} \; -\sum_{i=1}^{N} \log P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) \tag{17}$$

Thus the update rule with the stochastic gradient descent is

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \sum_{i=1}^{N} \nabla \log P(y^{(i)}|\mathbf{x}^{(i)}, \mathbf{w}) \tag{18}$$

It is possible to express this update rule analytically and to show that it is a direct extension of the update rule for the weights in the case of binary logistic regression. Let $\mathbf{t}^{(i)}$ and $\mathbf{o}^{(i)}$ denote **one-hot** encoded target and predicted output for the ith example. The update rule for multiclass logistic regression is:

$$\mathbf{w}_k \leftarrow \mathbf{w}_k + \alpha \sum_{i=1}^{N} (t_k^{(i)} - o_k^{(i)}))\mathbf{x}^{(i)}, \qquad k = 1, \ldots, K$$

The log-likelihood loss in the logistic regression, which is often called the **logistic loss** or just **log loss**) is in the literature often called also **cross-entropy loss** (although strictly speaking the logistic loss is an approximation of the true cross-entropy loss, which would require the actual (unknown) distribution of the examples, and we are approximating this unknown distribution by its samples contained in the training set). Nevertheless, these terms are now often used interchangeably and the term cross-entropy loss is common in the machine learning community.

5 Linear predictors with nonlinear features

So far we were dealing with linear regression and linear classification. However, in real life data are often more complex and a linear predictor may not be a satisfactory fit (see examples in

Fig. 7). In this case, we can turn to more advanced models like decision trees and neural networks (that we will study next). Before doing so, let's see how we can tackle these tasks still with the machinery of linear predictors but then feeding them with nonlinear features. You will see that in some cases this can work pretty well!

The main idea is to extract a vector of **nonlinear features** $\phi(\mathbf{x}) \in \mathbb{R}^n$ from the input $\mathbf{x} \in \mathbb{R}^d$ and to feed these nonlinear features to a linear predictor. The prediction will be non-linear in $\mathbf{x}$! With appropriately selected nonlinear features we can fit the data as illustrated in Fig. 8.

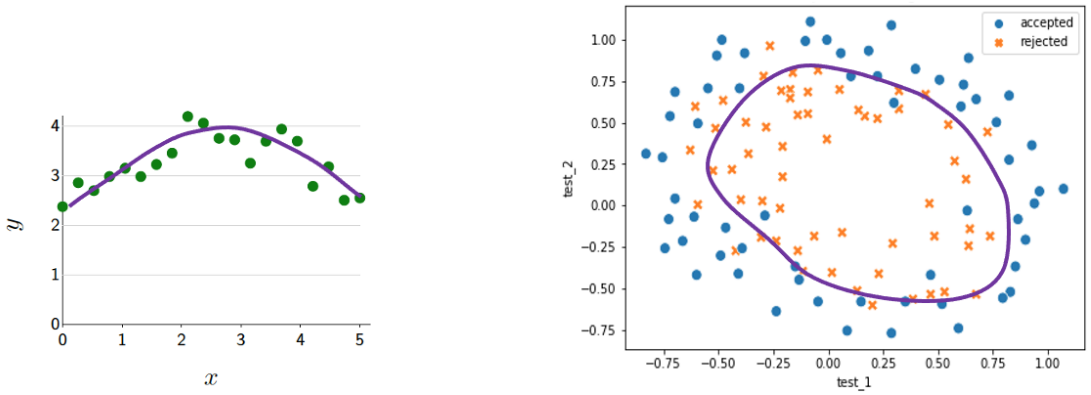


Figure 8: By extracting nonlinear features $\phi(\mathbf{x})$ from the input $\mathbf{x}$ and feeding those to linear regression as $h_{\mathbf{w}}(\mathbf{x}) = \phi(\mathbf{x}) \cdot \mathbf{w}$ or to logistic regression as $h_{\mathbf{w}}(\mathbf{x}) = Logistic(\phi(\mathbf{x}) \cdot \mathbf{w})$, we obtain predictions that are nonlinear in $\mathbf{x}$. Illustrations from [1].

5.1 Regression with nonlinear features

We generalize linear regression $\mathbf{x} \cdot \mathbf{w}$ by replacing the "raw" input $\mathbf{x}$ by some feature vector $\phi(\mathbf{x})$. The resulting predictor is

$$h_{\mathbf{w}}(\mathbf{x}) = \phi(\mathbf{x}) \cdot \mathbf{w} \tag{19}$$

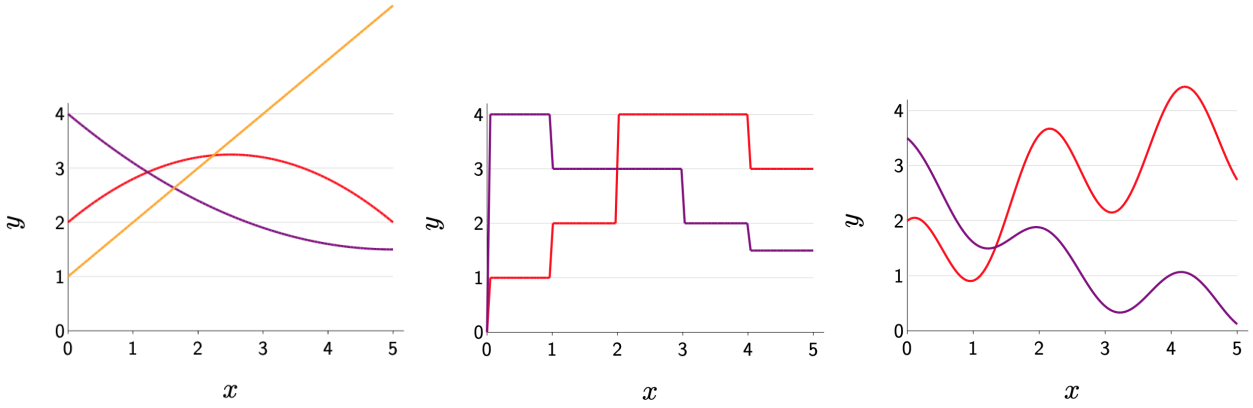
The feature vector $\phi(\mathbf{x})$ can be arbitrary. We will illustrate the use of nonlinear features for univariate regression only, i.e., for the case where the input is scalar x from which we will construct a n-dimensional feature vector $\phi(x)$. Fig. 9 illustrates three classes of nonlinear predictors that are obtained with different feature vectors.

Note that with $\phi(x) = [1, x]^\top$ the predictor in Eq (19) would simply be univariate linear regression (the dummy variable $x_0 = 1$ allows us to include the intercept term w_0 in the vector $\mathbf{w}$). Now, if we construct a nonlinear feature vector by adding a quadratic term x^2:

$$\phi(x) = [1, x, x^2]^\top$$

we obtain **quadratic predictors** illustrated in Fig. 8(a). The different curves there correspond to different weight vectors $\mathbf{w}$. The line corresponds to $\mathbf{w} = [1, 1, 0]^\top$ which sets the quadratic term to zero.

The **piecewise constant** predictors in Fig. 8(b) are obtained with feature extractors that divide the input space into regions and allow the predicted value of each region to vary independently. Specifically, each component of the feature vector corresponds to one region, e.g., (0, 1],



(a) (b) (c)

Figure 9: Examples of predictors with (a) quadratic features; (b) piece-wise constant features and (c) features with periodicity structure. Illustrations from [1].

and is 1 if x lies in that region and 0 otherwise:

$$\phi(x) = [\mathbf{1}[0 < x \leq 1], \mathbf{1}[1 < x \leq 2], \mathbf{1}[2 < x \leq 3], \mathbf{1}[3 < x \leq 4], \mathbf{1}[4 < x \leq 5]]^\top$$

Assuming the regions are disjoint, the weight associated with a component/region is exactly the predicted value. E.g., the predictor shown in red corresponds to $\mathbf{w} = [1, 2, 4, 4, 3]^\top$. As we make the regions smaller, we get more features, and the expressiveness of our hypothesis class increases. In the limit, we can essentially capture any predictor we want.

This sounds very nice but think what happens if x were not a scalar, but a d-dimensional vector $\mathbf{x}$? Then if each com p onent g ets broken u p into B bins, then there will be B d features! For each feature, we need to fit its wei g ht, and there will in g enerall y be too few exam p les to fit all the features.

Fig. 8(c) shows yet another family of the predictors, and these have some **periodicity structure**. In particular, these were obtained with

$$\phi(x) = [1, x, x^2, \cos(3x)]^\top$$

We showed three examples but there is an unboundedly large design space of possible feature extractors. In practice, the choice of features is informed by the prediction task that we wish to solve (either prior knowledge or preliminary data exploration) [1].

References

[1] M. Charikar and S. Koyejo. *Artificial Intelligence: Principles and Techniques (CS221).* Stanford University, 2024.

[2] D. Klein and P. Abbeel. *Intro to AI (ai.berkeley.edu).* UC Berkeley, 2023.

[3] A. Ng and T. Ma. *Machine Learning CS229: Lecture Notes.* Stanford University, 2023.

[4] S Russel and Norvig. P. *Artificial Intelligence: A Modern Approach, 4th Edition.* Pearson, 2021.