Mathematics of the Perceptron learning algorithm
Excerpts from Python Machine Learning
Second Edition
By Sebastian Raschka and Vahid Mirjalili[1]
and other sources

Stepan Oskin

July 11, 2019

Abstract

1 Introduction

1.1 Classification

Classifying data is a common task in machine learning. Suppose some given data points each belong to one of two classes, and the goal is to decide which class a new data point will be in.

In the case of the perceptron learning algorithm, a data point is viewed as a p-dimensional vector (a list of p numbers), and we want to know whether we can separate such points with a $(p-1)$-dimensional hyperplane. This is called a linear classifier. More formally, we can put the idea behind artificial neurons into the context of a binary classification task where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity. There are many hyperplanes that might classify the data. We can then define a decision function ($\phi(z)$) that takes a linear combination of certain input values x and a corresponding weight vector w, where z is the so-called net input.

1.2 McCulloch-Pitts (MCP) neuron

Trying to understand how the biological brain works, in order to design AI, *Warren McCulloch and Walter Pitts* published the first concept of a simplified brain cell, the so-called McCulloch-Pitts (MCP) neuron, in 1943[2]. Neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in figure 1:

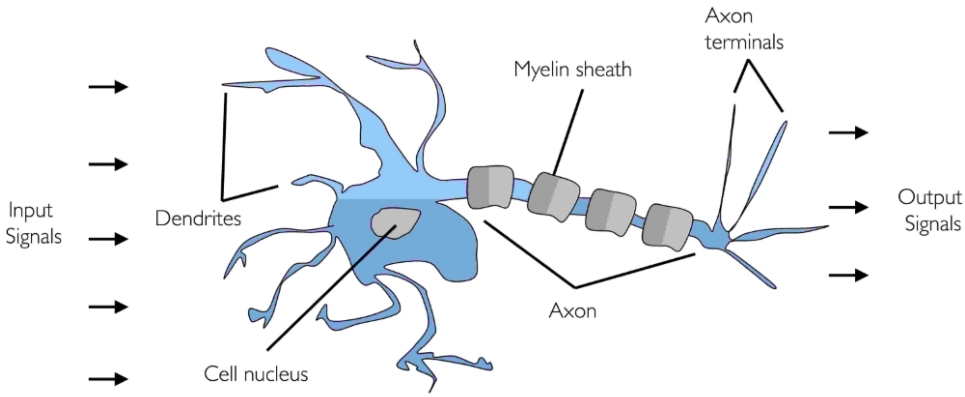


Figure 1: Neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals. According to the simplified MCP model proposed by McCulloch and Pitts, if the signal accumulated by the dendrites exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

McCulloch and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

1.3 Rosenblatt's Perceptron

Only a few years later, *Frank Rosenblatt* published the first concept of the perceptron learning rule based on the MCP neuron model[3]. With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that are then multiplied with the input features in order to make the decision of whether a neuron fires or not. In the context of supervised learning and classification, such an algorithm could then be used to predict if a sample belongs to one class or the other.

2 Mathematics of the perceptron

2.1 The formal definition of an artificial neuron

More formally, we can put the idea behind artificial neurons into the context of a binary classification task where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity. There are many hyperplanes that might classify the data. We can then define a decision function ($\phi(z)$) that takes a linear combination of certain input values x and a corresponding weight vector w, where z is the so-called net input:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}, \quad z = w_1 x_1 + \cdots + w_m x_m \tag{1}$$

Now, if the net input of a particular sample $\boldsymbol{x}^{(i)}$ is greater than a defined threshold θ, we predict class 1, and class -1 otherwise. In the perceptron algorithm, the decision function $\phi(.)$ is a variant of a **unit step function**:

$$\phi(z) = \begin{cases} 1 & \text{if } \ z \geq \theta, \\ -1 & \text{otherwise} \end{cases} \tag{2}$$

For simplicity, we can bring the threshold θ to the left side of the equation and define a weight-zero as $w_0 = -\theta$ and $x_0 = 1$ so that we write z in a more compact form:

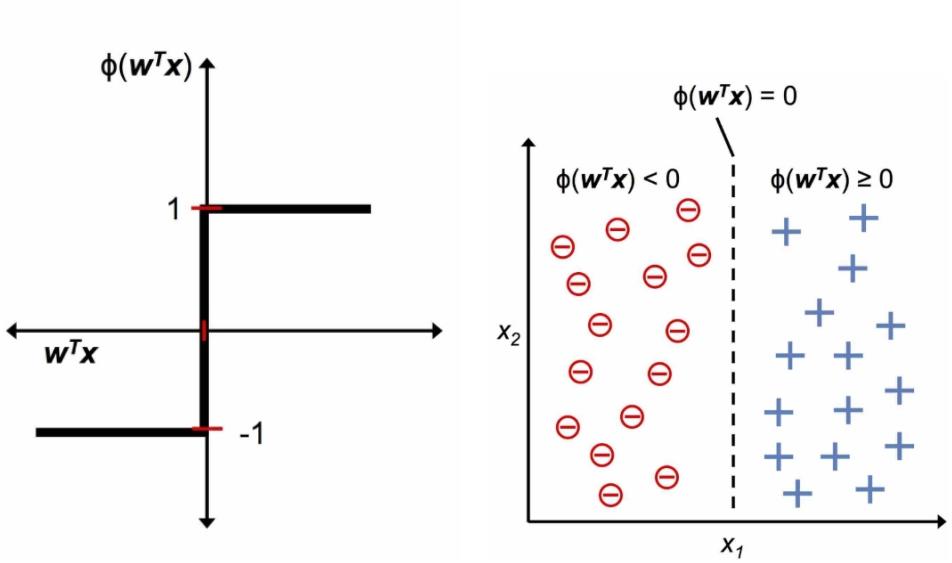
$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} \boldsymbol{x}_j \boldsymbol{w}_j = \boldsymbol{w}^T \boldsymbol{x} \tag{3}$$

In machine learning literature, the negative threshold, or weight, $w_0 = -\theta$, is usually called the **bias unit**.

Figure 2 illustrates how the net input $z = \boldsymbol{w}^T \boldsymbol{x}$ is squashed into a binary output (`-1` or 1) by the decision function of the perceptron (subfigure 2a) and how it can be used to discriminate between two linearly separable classes (subfigure 2b):

2.2 The perceptron learning rule

The whole idea behind the MCP neuron and Rosenblatt's **thresholded** perceptron model is to use a reductionist approach to mimic how a single neuron in the brain works: it either **fires** or it doesn't. Thus, Rosenblatt's initial



(a) $\gamma = 0.2$ (b) $\gamma = 100$

Figure 2: .

perceptron rule is fairly simple and can be summarized by the following steps:

1. Initialize the weights to 0 or small random numbers.
2. For each training sample:
   (a) Compute the output value.
   (b) Update the weights.

Here, the output value is the class label predicted by the unit step function that we defined earlier, and the simultaneous update of each weight w_j in the weight vector $\boldsymbol{w}$ can be more formally written as:

$$w_j := w_j + \Delta w_j \tag{4}$$

The value of Δw_j, which is used to update the weight w_j, is calculated by the perceptron learning rule:

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)} \tag{5}$$

Where η is the **learning rate** (typically a constant between 0.0 and 1.0), $y^{(i)}$ is the **true class label** of the i^{th} training sample, and $\hat{y}^{(i)}$ is the **predicted class label**. It is important to note that all weights in the weight vector are being updated simultaneously, which means that we don't recompute the $\hat{y}^{(i)}$ before all of the weights Δw_j are updated.

A simple diagram presented on figure 3 illustrates the general concept of the perceptron:

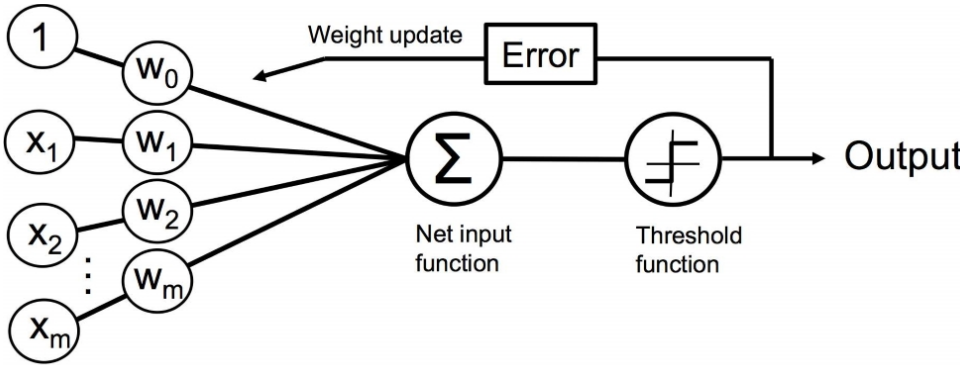


Figure 3: Rosenblatt's thresholded perceptron model uses a reductionist approach to mimic how a single neuron in the brain works: it either fires or it doesn't. If the net input of a particular sample $\boldsymbol{x}^{(i)}$ is greater than a defined threshold θ, we predict class 1, and class -1 otherwise.

The preceding diagram illustrates how the perceptron receives the inputs of a sample $\boldsymbol{x}$ and combines them with the weights $\boldsymbol{w}$ to compute the net input. The net input is then passed on to the threshold function, which generates a binary output `-1` or `+1` —the predicted class label of the sample. During the learning phase, this output is used to calculate the error of the prediction and update the weights.

2.3 Examples of the perceptron learning rule

Concretely, for a two-dimensional dataset, we would write the weight update using the perceptron learning rule as:

$$\begin{aligned} \Delta w_0 &= \eta \left(y^{(i)} - \text{output}^{(i)} \right) \\ \Delta w_1 &= \eta \left(y^{(i)} - \text{output}^{(i)} \right) x_1^{(i)} \\ \Delta w_2 &= \eta \left(y^{(i)} - \text{output}^{(i)} \right) x_2^{(i)} \end{aligned} \tag{6}$$

Let us make a simple thought experiment to illustrate how beautifully simple this learning rule really is. In the two scenarios where the perceptron predicts the class label correctly, the weights remain unchanged:

$$\begin{aligned} \Delta w_j &= \eta\,(-1-(-1))\,x_j^{(i)} = 0 \\ \Delta w_j &= \eta\,(1-1)\,x_j^{(i)} = 0 \end{aligned} \tag{7}$$

However, in the case of a wrong prediction, the weights are being pushed towards the direction of the positive or negative target class:

$$\begin{aligned} \Delta w_j &= \eta\,(1-(-1))\,x_j^{(i)} = \eta\,(2)\,x_j^{(i)} \\ \Delta w_j &= \eta\,(-1-1))\,x_j^{(i)} = \eta\,(-2)\,x_j^{(i)} \end{aligned} \tag{8}$$

To get a better intuition for the multiplicative factor $x_j^{(i)}$, let us go through another simple example, where:

$$y^{(i)} = +1,\ \hat{y}_j^{(i)} = -1,\ \eta = 1 \tag{9}$$

Let's assume that $x_j^{(i)} = 0.5$, and we mis-classify this sample as `-1`. In this case, we would increase the corresponding weight by 1 so that the net input $x_j^{(i)} \times w_j^{(i)}$ would be more positive the next time we encounter this sample, and thus be more likely to be above the threshold of the unit step function to classify the sample as `+1`:

$$\Delta w_j^{(i)} = (1-(-1))\,0.5 = (2)\,0.5 = 1 \tag{10}$$

The weight update is proportional to the value of $x_j^{(i)}$. For example, if we have another sample $x_j^{(i)} = 2$ that is incorrectly classified as `-1`, we'd push the decision boundary by an even larger extent to classify this sample correctly the next time:

$$\Delta w_j^{(i)} = (1-(-1))\,2 = (2)\,2 = 4 \tag{11}$$

2.4 Convergence of the perceptron

It is important to note that the **convergence of the perceptron is only guaranteed** if the two **classes are linearly separable** and the **learning rate is sufficiently small**. As can be seen in figure 4, in some cases the two classes can't be separated by a linear decision boundary. In this situation we can set a maximum number of passes over the training dataset

(**epochs**) and/or a threshold for the number of tolerated mis-classifications —the perceptron would never stop updating the weights otherwise.

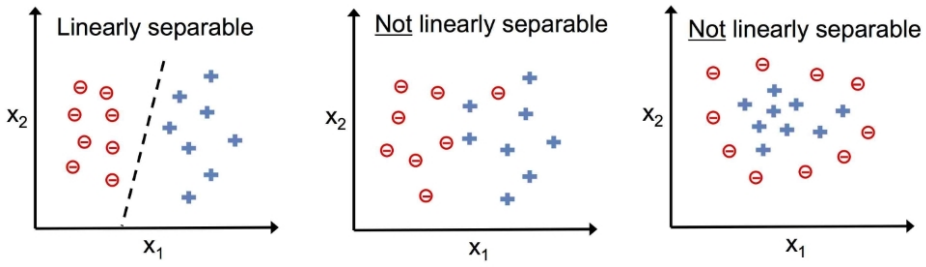


Figure 4: Convergence of the perceptron is only guaranteed if the two classes are linearly separable. However, in some cases the two classes can't be separated by a linear decision boundary.

3 An object-oriented perceptron API using NumPy

For class definition see `src.classifiers.Perceptron`. Code taken from the book by Sebastian Raschka and Vahid Mirjalili[1].

3.1 Description of the implementation

In `src.classifiers.Perceptron`, an object-oriented approach was taken to define the perceptron interface as a Python class, which allows us to initialize new `Perceptron` objects that can learn from data via a `fit` method, and make predictions via a separate `predict` method. As a convention, authors of the code[1] append an underscore (_) to attributes that are not being created upon the initialization of the object but by calling the object's other methods, for example, `self.w_`.

Using this perceptron implementation, new `Perceptron` objects can now be initialized with a given learning rate `eta` and `n_iter`, which is the number of epochs (passes over the training set). Via the `fit` method, we initialize the weights in `self.w_` to a vector $\mathbb{R}^{m+1}$, where m stands for the number of dimensions (features) in the dataset, where we add 1 for the first element in this vector that represents the bias unit. Under this convention, the first element in this vector, `self.w_[0]`, represents the so-called **bias unit**.

3.2 Initialization of weights

Vector of weights in initialized to small random numbers drawn from a normal distribution with standard deviation `0.01` via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, where `rgen` is a `NumPy` random number generator that is seeded with a user-specified random seed so that previous results can be reproduced if desired.

Now, the reason we don't initialize the weights to zero is that the learning rate η (`eta`) only has an effect on the classification outcome if the weights are initialized to non-zero values. If all the weights are initialized to zero, the learning rate parameter `eta` affects only the scale of the weight vector, not the direction.

The dot product of two Euclidean vectors $\boldsymbol{v1}$ and $\boldsymbol{v2}$ is defined as:

$$\boldsymbol{v1} \cdot \boldsymbol{v2} = \|\boldsymbol{v1}\| \, \|\boldsymbol{v2}\| \cos\theta \tag{12}$$

where θ is the angle between $v1$ and $v2$.

We can consider vectors $\boldsymbol{v1} = \begin{bmatrix}1 & 2 & 3\end{bmatrix}$ $\boldsymbol{v2} = 0.5 \times \boldsymbol{v1}$. Using trigonometry, we can show that the angle between $\boldsymbol{v1}$ and $\boldsymbol{v2}$ would be exactly zero:

$$\theta = \arccos \frac{\boldsymbol{v1}\boldsymbol{v2}}{\|\boldsymbol{v1}\| \, \|\boldsymbol{v2}\|} = 0 \tag{13}$$

3.3 Fitting the model and making predictions

After the weights have been initialized, the `fit` method loops over all individual samples in the training set and updates the weights according to the perceptron learning rule that we discussed in the previous section. The class labels are predicted by the `predict` method, which is called in the `fit` method to predict the class label for the weight update, but `predict` can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the `self.errors_` list so that we can later analyze how well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product $\boldsymbol{w}^T\boldsymbol{x}$.

3.4 Vectorization

Instead of using NumPy to calculate the vector dot product between two arrays `a` and `b` via `a.dot(b)` or `np.dot(a, b)`, we could also perform the calculation in pure Python via `sum([i * j for i, j in zip(a, b)])`. How-

ever, the advantage of using `NumPy` over classic Python for loop structures is that its arithmetic operations are **vectorized**. Vectorization means that an elemental arithmetic operation is automatically applied to all elements in an array. By formulating our arithmetic operations as a sequence of instructions on an array, rather than performing a set of operations for each element at the time, we can make better use of our modern CPU architectures with **Single Instruction, Multiple Data (SIMD)** support. Furthermore, `NumPy` uses highly optimized linear algebra libraries such as *Basic Linear Algebra Subprograms (BLAS) and Linear Algebra Package (LAPACK)* that have been written in C or Fortran. Lastly, `NumPy` also allows us to write our code in a more compact and intuitive way using the basics of linear algebra, such as vector and matrix dot products.

4 Training a perceptron model on two classes from the Iris dataset

To test the perceptron implementation, Iris flower dataset for classification was used, from which two flower classes Setosa and Versicolor were loaded. Although the perceptron rule is not restricted to two dimensions, only two features were considered (`sepal length` and `petal length` for visualization purposes. Though we only chose the two flower classes Setosa and Versicolor for practical reasons, the perceptron algorithm can be extended to multi-class classification —for example, the **One-versus-All (OvA)** technique.

4.1 Extracting `X` and `y` from Iris dataset

index	0	1	2	3	4
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Table 1: The first 5 rows from the Iris flower dataset for classification. The four features include `petal_width`, `petal_length`, `sepal_width`, and `sepal_length`. The three class labels are `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica`.

Next, we extract the first 100 class labels that correspond to the 50

`Iris-setosa` and 50 `Iris-versicolor` flowers, and convert the class labels into the two integer class labels `1` (`versicolor`) and `-1` (`setosa`) that we assign to a vector `y`, where the values method of a `pandas DataFrame` yields the corresponding `NumPy` representation.

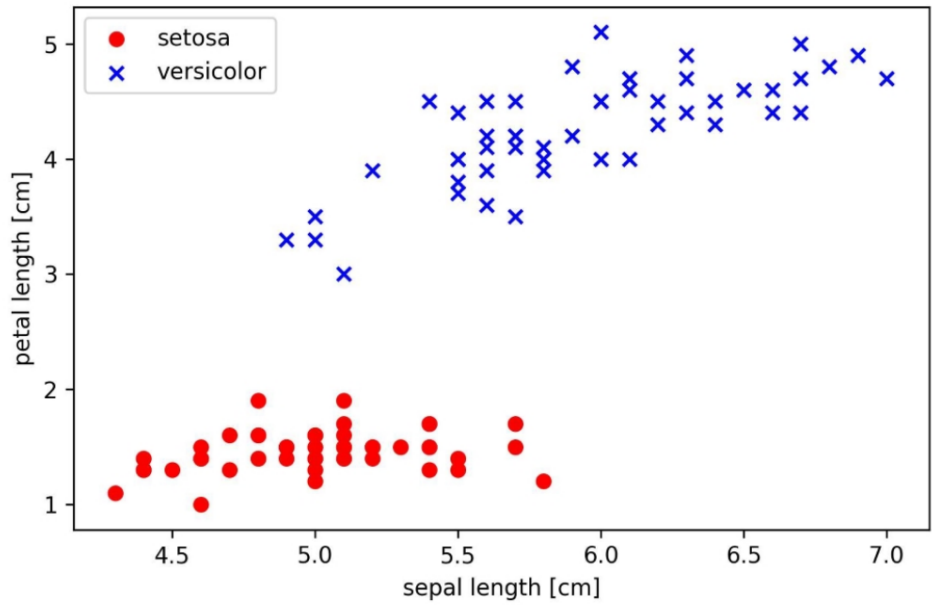


Figure 5: We extract the first 100 class labels from Iris dataset that correspond to the 50 `Iris-setosa` and 50 `Iris-versicolor` flowers, and convert the class labels into the two integer class labels `1` (`versicolor`) and `-1` (`setosa`) that we assign to a vector `y`. Similarly, we extract the first feature column (sepal length) and the third feature column (petal length) of those 100 training samples and assign them to a feature matrix `X`.

Similarly, we extract the first feature column (sepal length) and the third feature column (petal length) of those 100 training samples and assign them to a feature matrix `X`, which we can visualize via a two-dimensional scatter plot presented in figure 5. The scatterplot shows the distribution of flower samples in the Iris dataset along the two feature axes, petal length and sepal length. In this two-dimensional feature subspace, we can see that a linear decision boundary should be sufficient to separate Setosa from Versicolor flowers. Thus, a linear classifier such as the perceptron should be able to classify the flowers in this dataset perfectly.

4.2 Performance of the perceptron on two linearly separable classes

Figure 6 shows the misclassification error for each epoch and can be used to check whether the algorithm converged and found a decision boundary that separates the two Iris flower classes:

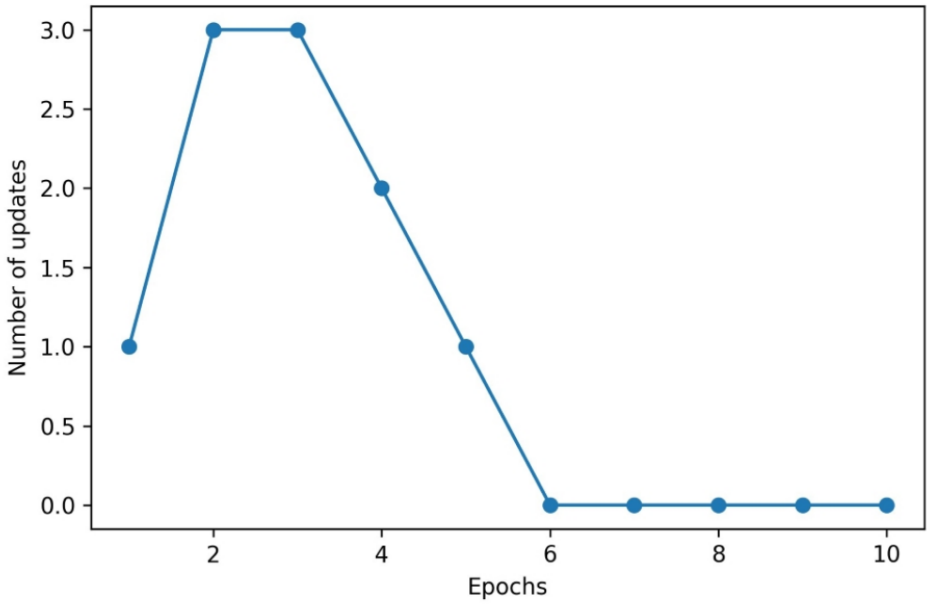


Figure 6: Plotting misclassification error by each epoch shows that the perceptron converged after the sixths epoch and can classify the two classes of Iris flowers perfectly.

As we can see in the preceding plot, our perceptron converged after the sixth epoch and should be able to classify the training samples perfectly. In addition to the number of updates per epoch, decision boundaries can be visualized for two-dimensional datasets. Decision boundary learned by the perceptron on the two classes of Iris flowers is presented on figure 7:

As we can see in the plot, the perceptron learned a decision boundary that is able to classify all flower samples in the Iris training subset perfectly. Although the perceptron classified the two Iris flower classes perfectly, convergence is one of the biggest problems of the perceptron. Frank Rosenblatt proved mathematically that the perceptron learning rule converges if the two classes can be separated by a linear hyperplane. However, if classes cannot

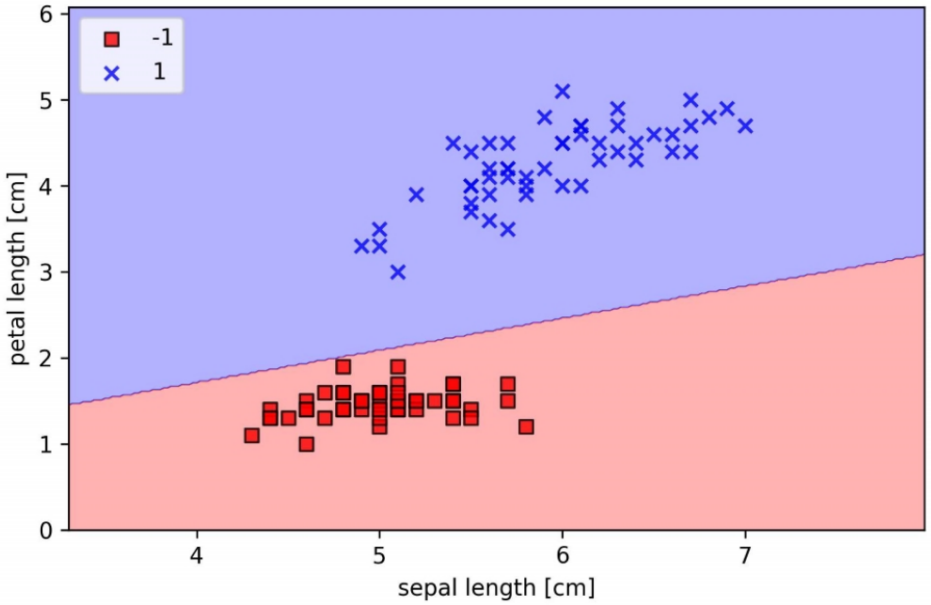


Figure 7: Plotting the decision boundary for two features shows that the perceptron learned a decision boundary that is able to classify all flower samples in the Iris training subset perfectly.

be separated perfectly by such a linear decision boundary, the weights will never stop updating unless we set a maximum number of epochs.

5 Training a perceptron model on three classes from the Iris dataset

5.1 OvA Technique for multi-class classification

OvA, or sometimes also called **One-versus-Rest (OvR)**, is a technique that allows us to extend a binary classifier to multi-class problems. Using OvA, we can train one classifier per class, where the particular class is treated as the positive class and the samples from all other classes are considered negative classes. If we were to classify a new data sample, we would use our n classifiers, where n is the number of class labels, and assign the class label with the highest confidence to the particular sample. In the case of the perceptron, we would use OvA to choose the class label that is associated with the largest absolute net input value.

5.2 Data preprocessing

The `scikit-learn` library offers not only a large variety of learning algorithms, but also many convenient functions to preprocess data and to fine-tune and evaluate our models. Conveniently, the Iris dataset is already available via `scikit-learn`, since it is a simple yet popular dataset that is frequently used for testing and experimenting with algorithms. We will only use two features from the Iris dataset for visualization purposes.

Petal length and petal width of the 150 flower samples was assigned to the feature matrix `X` and the corresponding class labels of the flower species to the vector `y`.

5.2.1 Encoding target variable

In `scikit-learn`'s version of the Iris dataset, the Iris flower class names `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica` are already stored as integers (here: '0', '1', '2'). Although many `scikit-learn` functions and class methods also work with class labels in string format, using integer labels is a recommended approach to avoid technical glitches and improve computational performance due to a smaller memory footprint; furthermore, encoding class labels as integers is a common convention among most machine learning libraries.

5.2.2 Train-test split

To evaluate how well a trained model performs on unseen data, the dataset was split into separate training and test datasets using the `train_test_split` function from `scikit-learn`'s `model_selection` module. `X` and `y` arrays were randomly split into 30 percent test data (45 samples) and 70 percent training data (105 samples).

Note that the `train_test_split` function already shuffles the training sets internally before splitting; otherwise, all class `0` and class `1` samples would have ended up in the training set, and the test set would consist of 45 samples from class 2. Via the `random_state` parameter, we provided a fixed random seed (`random_state=1`) for the internal pseudo-random number generator that is used for shuffling the datasets prior to splitting. Using such a fixed `random_state` ensures that our results are reproducible.

Lastly, we took advantage of the built-in support for stratification via `stratify=y`. In this context, stratification means that the `train_test_split` method returns training and test subsets that have the same proportions of class labels as the input dataset. NumPy's `bincount` function, which counts

the number of occurrences of each value in an array, can be used to verify that this is indeed the case.

5.2.3 Feature scaling

Many machine learning and optimization algorithms also require feature scaling for optimal performance. Features were standardized using the `StandardScaler` class from `scikit-learn`'s `preprocessing` module: Using the `fit` method, `StandardScaler` estimated the parameters μ (sample mean) and σ (standard deviation) for each feature dimension from the training data. By calling the `transform` method, we then standardized the training data using those estimated parameters μ and σ. The same scaling parameters were used to standardize the test set so that both the values in the training and test dataset are comparable to each other.

5.3 Training a perceptron on a multi-class dataset using `scikit-learn`

Having standardized the training data, a perceptron model can now be trained on all three classes on Iris flowers. Most algorithms in `scikit-learn` already support multi-class classification by default via the **One-versus-Rest (OvR)** method, which allows us to feed the three flower classes to the perceptron all at once. The `scikit-learn` interface resembles object-oriented perceptron implementation provided by *Raschka and Mirjalili*[1]: after loading the `Perceptron` class from the `linear_model` module, a new `Perceptron` object is initialized and the model is trained via the `fit` method. Here, the model parameter `eta0` is equivalent to the learning rate `eta` that was used in Raschka and Mirjalili's perceptron implementation, and the `max_iter` parameter defines the number of epochs (passes over the training set).

5.4 Finding an appropriate learning rate

Finding an appropriate learning rate requires some experimentation. If the learning rate is too large, the algorithm will overshoot the global cost minimum. If the learning rate is too small, the algorithm requires more epochs until convergence, which can make the learning slow —especially for large datasets. `random_state` parameter was used to ensure the reproducibility of the initial shuffling of the training dataset after each epoch.

5.5 Model accuracy

Having trained a model in `scikit-learn`, predictions can be made via the `predict` method, just like in Raschka and Mirjalili's perceptron implementation. We see that the perceptron misclassifies one out of the 45 flower samples. Thus, the misclassification error on the test dataset is approximately 0.0222 or 2.2 percent (1 / 45 = 0.0222).

Instead of the misclassification error, many machine learning practitioners report the classification accuracy of a model, which is simply calculated as follows:

Classification accuracy = 1 - error = 0.978 or 97.8 percent.

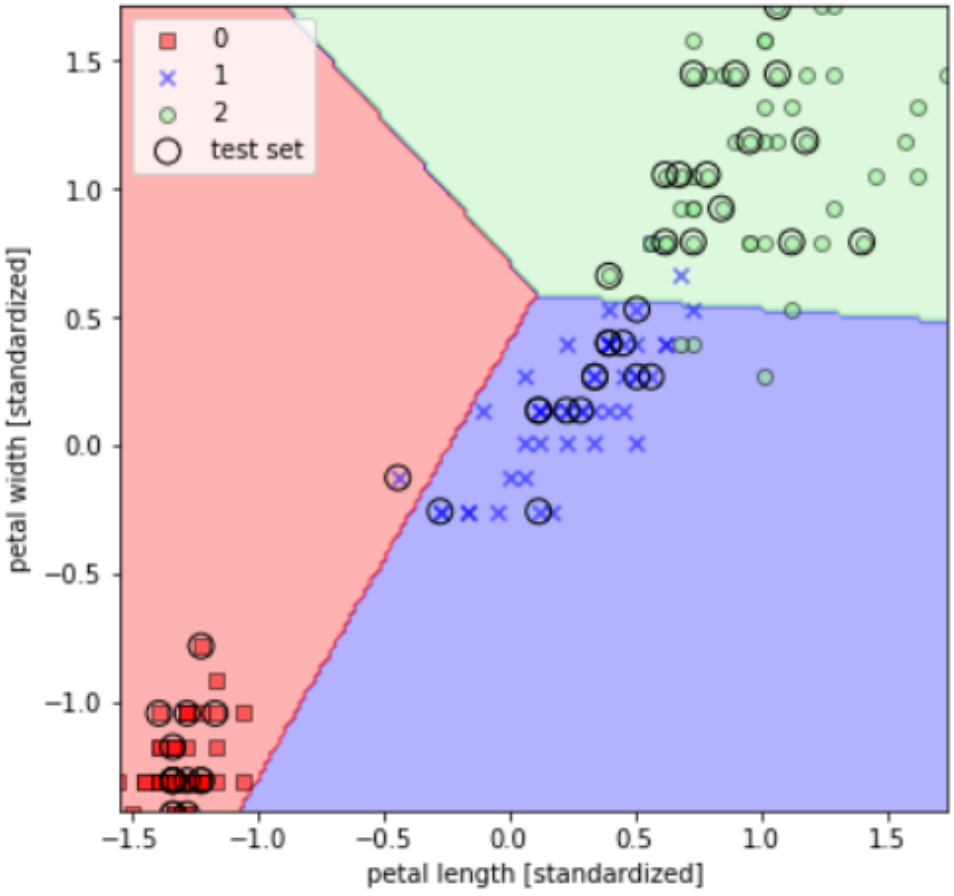


Figure 8: For three classes of Iris flowers, the perceptron is able to achieve an accuracy of 98% on the test set. However, since the three classes are not perfectly linearly separable, the perceptron never converges.

5.6 Decision boundary learned by the perceptron for three classes

For two-dimensional datasets, decision boundary learned by the algorithm can be visualized. Decision boundary learned by the perceptron for three classes of Iris flowers is presented on figure 8

References

[1] S. Raschka and V. Mirjalili, *Python Machine Learning, 2nd Ed.* . Birmingham, UK: Packt Publishing, 2 ed., 2017.

[2] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity (reprinted from bulletin of mathematical biophysics, vol 5, pg 115-133, 1943)," *Bulletin of Mathematical Biology*, vol. 52, no. 1–2, pp. 99–115, 1990.

[3] F. Rosenblatt, "The Perceptron: a Percieving and Recognizing Automation," tech. rep., Cornell Aeronautical Laboratory, Inc., Buffalo, NY, 1957.