PYCON INDIA

Chennai, 2019

A Talk by Arjun Kathuria

How I hand-coded core Machine learning Algorithms from scratch in python, what i learned, and why you should too.

About the Speaker (Me)

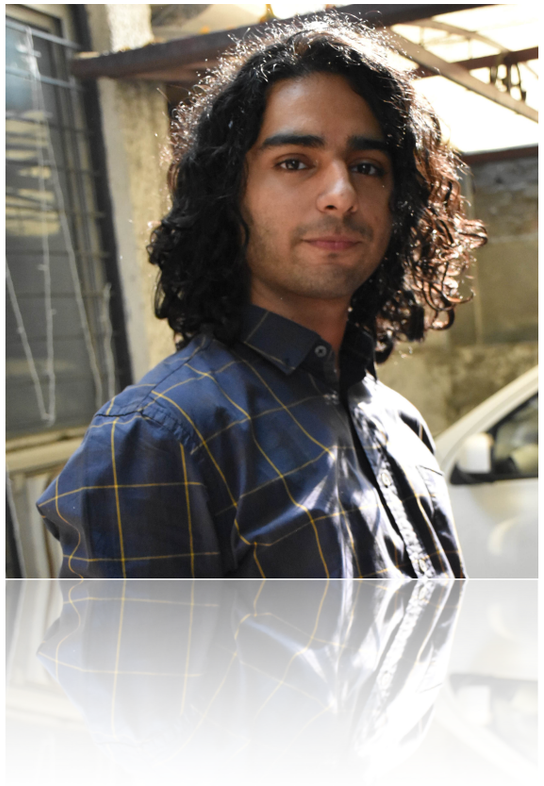
Hi,

I am Arjun, an independent software developer and hacker from New Delhi, India.

I was a Google summer of code student In 2016 with jQuery Foundation.

I held the position of the sole lead frontend software developer for a high growth startup in Bangalore, which i left to follow and pursue my other interests in computer science and music.

My current interests include python, machine learning, full-stack development, lower level Kernel stuff, Rust and Lisp.



- Github: **github/arjunkathuria**
- Twitter: **@arjunslays**
- email: **arjun.kathuria8@gmail.com**

The premise
The WHY

The premise

Machine Learning too, just like any field of science, has certain basic concepts and algorithms that act as the fundamental building blocks of everything wonderful you see around you. From your Youtube recommendations to your phone cameras, its everywhere.

Fortunately, these algorithms have been made available to us for use as wonderfully optimized, properly packaged and easy-to-use abstractions, think scikit-learn and friends.

We essentially treat and use them as BLACKBOXES!

Import a classifier already made for you from the library, throw your training data at it and BOOM, you can make predictions on new data now and well be on your way..

Believe it or not, that is a MAGNIFICENT thing. This is a massive win for practicality, pragmatism and performance.

BUT to Me, it seemed a bit too much of magic.

So I decided to pop the hood and look inside. This talk takes gives you a crisp take on what followed and what I found and presents you with what I learned along the way.

Chapter 1

Demystifying Machine learning.

WHAT is machine learning ?

I’ll do you one better

WHO is machine learning ?

Yea? I’ll do you one even better…

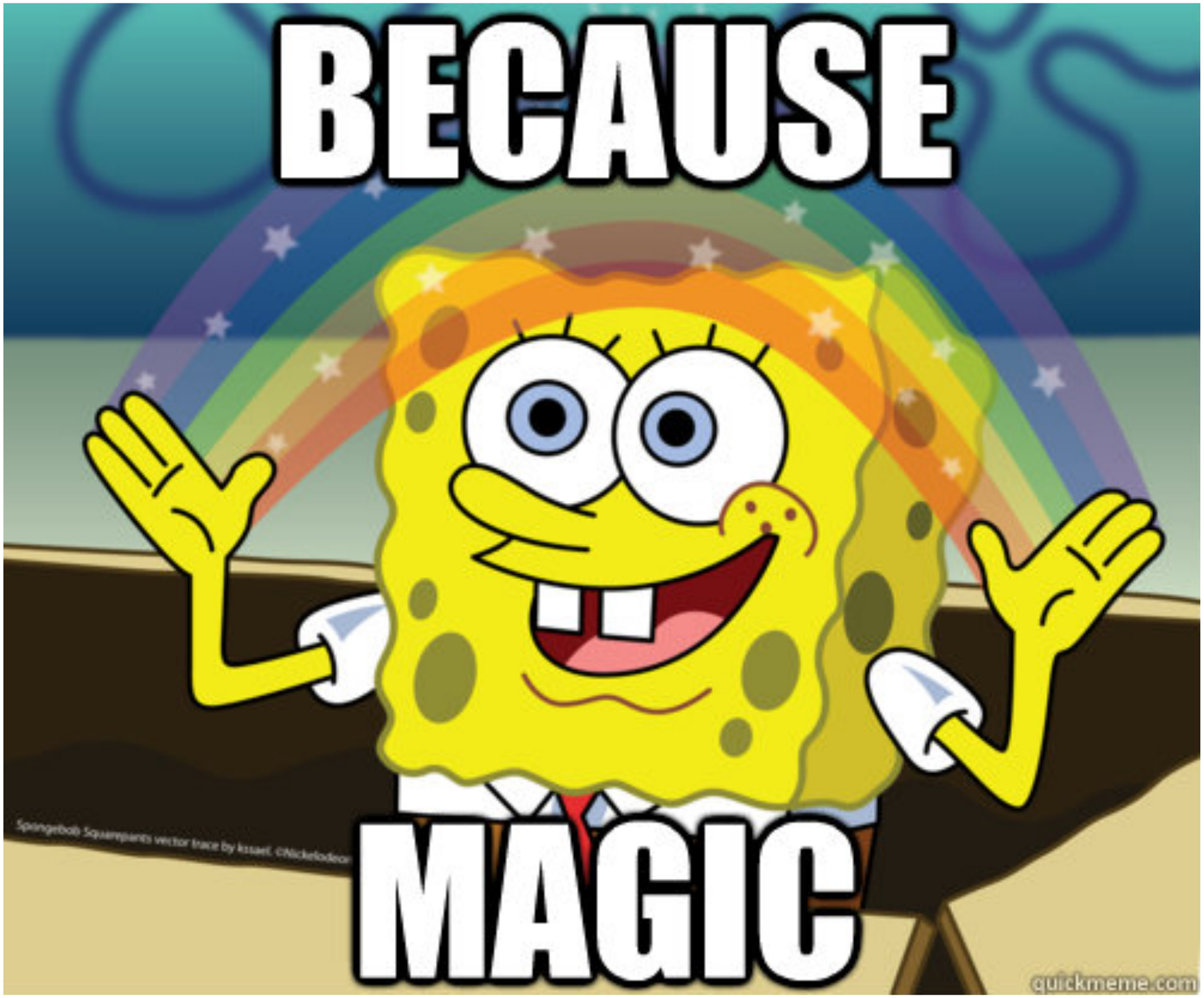
WHY is machine learning ?!

What one extreme of the spectrum believes what Machine Learning is

.

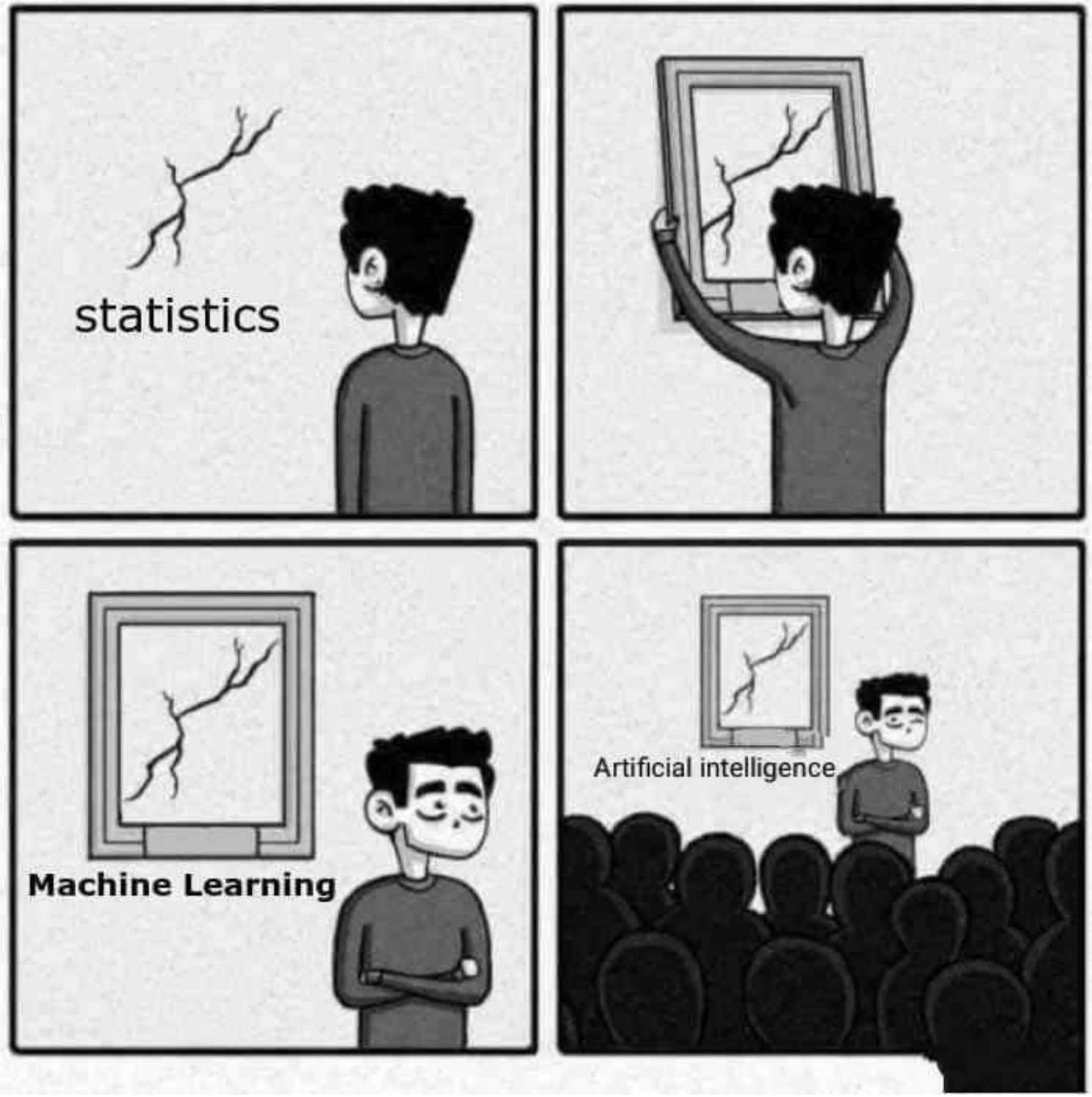
.

.


BECAUSE
MAGIC
quickmeme.com

What the other extreme of the spectrum believes what Machine Learning is

-
-
-


statistics
Machine Learning
Artificial intelligence

Well, Yes and No.

Its somewhere in the middle

The fundamentals of machine learning are still true to core statistics, but in practice, as of today, it has grown into something much bigger (as we will see).

In its simplest form -

When you zoom out and tune out the noise
Machine learning pertains to extraction of knowledge from data by building intelligent (often self-learning) agents / systems in order to make predictions.

Machine Learning, a broad field.

Machine Learning

Supervised Learning

- Labeled data
- Predicts outcome
- Eg: Classifiers like spam filters, text / character recognition

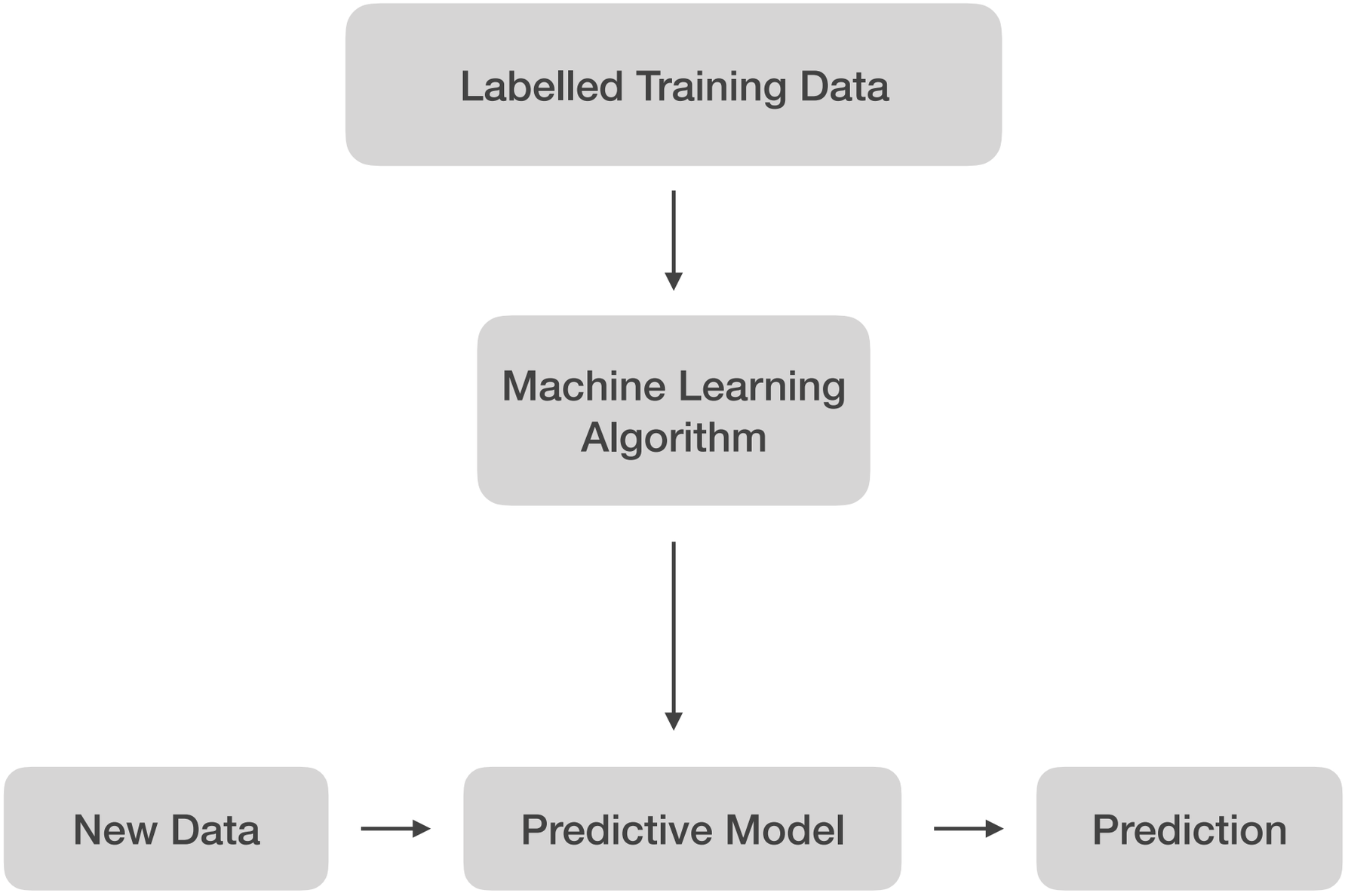
Unsupervised Learning

- Unlabelled data
- Finds hidden structure in data
- Eg: Clustering, user segmentation etc.

Reinforcement Learning

- Interacts with environment.
- Decision process based on a reward system
- Eg: Game playing agents.

Supervised learning in a



Chapter 2: The Algorithms

The classification algorithms to be covered

1. The Perceptron classifier algorithm
   +
   The Adaline algorithm

2. Logistic Regression

The Dataset

The ubiquitous Iris dataset is used for all code and examples in the talk

The dataset consists of 150 rows or measurements of the Iris flowers of 3 different species, namely - Virginica, Setosa and Versicolor - (50 of each) and it measures 4 of the flower features for each measurement.

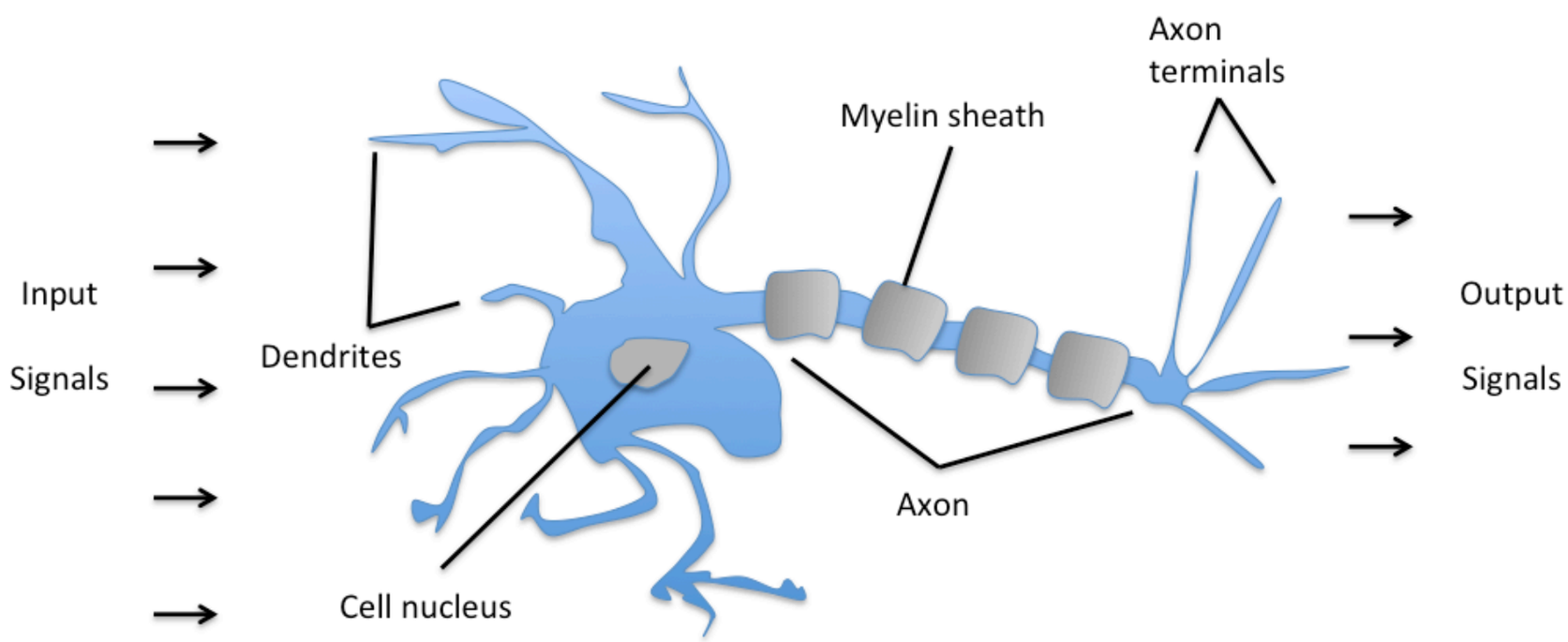
- The Sepal length [cm]
- The Sepal Width [cm]
- Petal Length [cm]
- Petal Width [cm]

Data Shape

	Sepal Length	Sepal Width	Petal length	petal width	Class Label
1	5.1	3.5	1.4	0.2	Iris-setosa
2	4.9	3.0	1.4	0.2	iris-setosa
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
57	6.5	3.4	4.5	1.2	iris-versicolor
.	.	.	.	.	.
.	.	.	.	.	.
149	6.2	3.4	5.4	2.3	iris-virginica
150	5.9	3.0	5.1	1.8	iris-virginica

Incidently, in terms of $x_{(m)}^{(n)}$, $x_{(1)}^{(150)}$ would refer to the 1st column/feature of sample Number 150.

Modelling the artificial neuron after a biological one



Schematic of a biological neuron.

Neurons (also called neurones or nerve cells) are the fundamental units of the brain and nervous system. They are electrically excitable and responsible for sending messages to and from the brain.

So, how a biological Neuron works is, it receives multiple signals from its extremities called *dendrites*, which is then propagated through the neuron's body and Now if the total strength of signal exceeds a certain threshold, an output signal is generated and passed on by the axon terminals

Artificial Neuron applied to a classification task.

The Earliest attempt at artificially modelling this behaviour for AI was done by two gentlemen named **Warren McCulloch and Walter Pitts** and thus, their model came to be known as the **McCulloch-Pitts** neuron.

They modelled the artificial neuron as a logic gate with a binary output, indicating that if the input exceeds the given threshold then the output was 1 or else a 0.

The Perceptron Algorithm

One of the earliest classification algorithms known for binary classification. It was Published by Frank Rosenblatt and was amongst the first algorithmically described ML Algorithm for classification. It was based on the MCP(McCulloch–Pitts) model of a neutron.

Later, based on the MCP neuron, a few years hence, the notion of Perceptron learning rule was published by **Frank Rosenblatt.** With Perceptron, Rosenblatt **put forward and algorithm that would automatically learn the weight coefficients** which are then multiplied by the input features to decide wether the neuron fires or not. In the context of Machine learning (more specifically supervised learning) and binary classification, this behaviour can be used to classify or predict wether a given samples belongs to class 1 or the other.

*A refresher on vectors

So, the vectors have about 2 major understanding systems / ideas associated with them which we primarily know as:

1. The Physics Perspective

2. The Computer Science perspective

The Physics Perspective

The Physics Perspective treats vectors as essentially Arrows pointing in space. What defines a given vector is its length(magnitude) and the direction it is pointing. As long as those two are same, the arrow can be moved around in space and still be the same vector. The vectors in a flat plane are 2-dimensional and vectors residing in a 3-d space are 3-D vectors and so on. The ones you studied in school.

The CS Perspective

The CS Perspective defines vectors as ORDERED list of numbers. To oversimplify, vectors in CS are just a fancy word for a list. A 2 dimensional vector simply means the length of the list is 2. Mind == Blown right ?
For Eg:- (see next page)

Suppose you are modelling houses as vectors and all you care about are the house no, sq. footage, and the price.

$$= \left.\begin{bmatrix} 15 \text{ h. no} \\ ₹100 \\ 2500\text{ft}^2 \end{bmatrix}\right\} = 3 \text{ Dimensional}$$

$$\begin{bmatrix} 15 \text{ h. no} \\ ₹100 \\ 2500\text{ft}^2 \end{bmatrix} \neq \begin{bmatrix} 15 \text{ h. no} \\ ₹2500 \\ 100\text{ft}^2 \end{bmatrix}$$

*Since order is important

Operations on vectors

Vector Addition:-
Vectors can be added to other vectors and the result is a new vector

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 2+5 \\ 3+6 \end{bmatrix} = \begin{bmatrix} 7 \\ 9 \end{bmatrix}$$

Vector scalar multiplication:-
Multiplying vectors with numbers, SCALES the vector, thus the name SCALAR.
It is equivalent to multiplying each element of the vector with the number individually.

$$2 \times \begin{bmatrix} 3 \\ 5 \end{bmatrix} = \begin{bmatrix} 2*3 \\ 2*5 \end{bmatrix} = \begin{bmatrix} 6 \\ 10 \end{bmatrix}$$

Overview of How it works without getting into the math.

- Initialise a set of weights (numbers, often random) equal to the number of features on the training set. i.e 1:1 mapping from a feature vector to a weight vector.

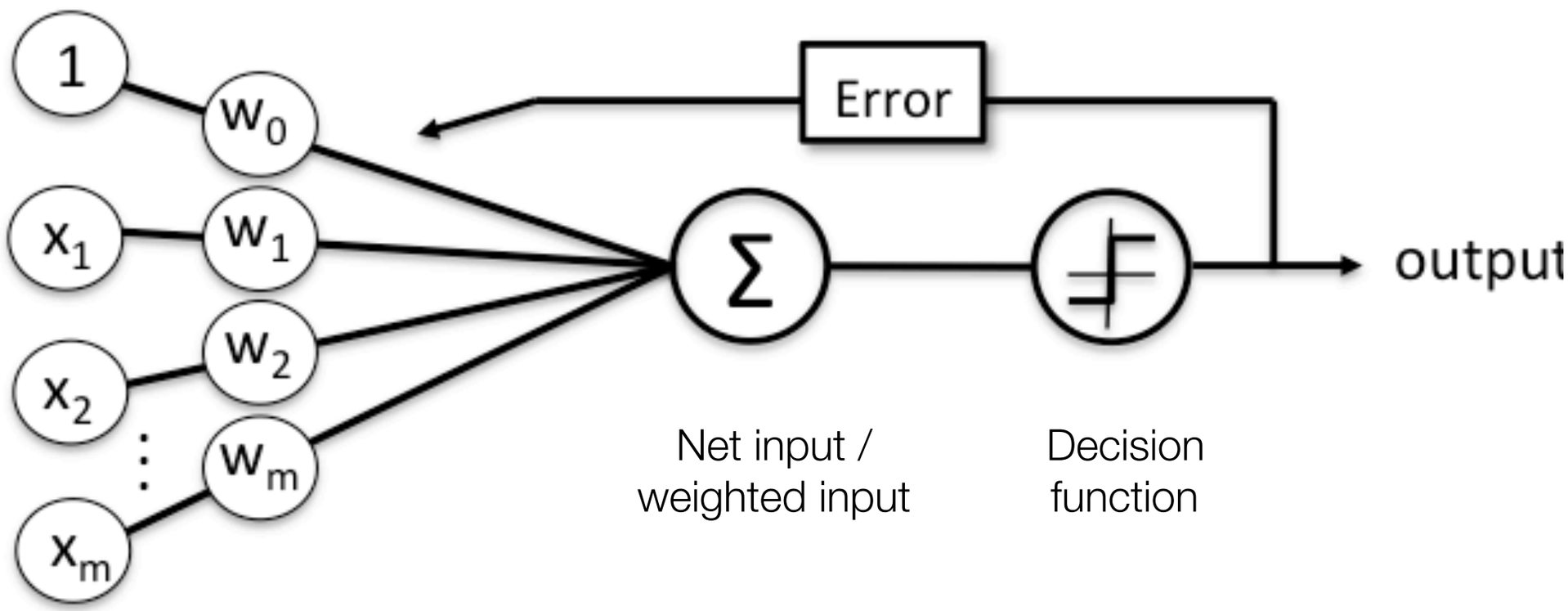
- For each training sample $\mathbf{x}^{(i)}$

  - Calculate the net input by linear combination of the weight and feature vectors. net input is also sometimes known as weighted input.

  - If the net input is greater than a threshold, predict class A else predict class B (or not of class A), often numerically coded as 1 and -1. computationally, this is done by a decision function.

  - Update the weights based on the perceptron learning rule, so that it is more accurate the next time.

The Schematic



Schematic of a perceptron classifier.

Terminology and Components of Perceptron Learning Rule.

1. **Input Vector / Input matrix / Input values**:- the data values as a vector/matrix/rows

2. **Weight Vector**:- a column vector containing weights for each dimension of input value.

3. **Net input**:- The linear combination of the input values (x) and the weight vector (w)

$$w = \begin{bmatrix} w_1 \\ \cdot \\ \cdot \\ \cdot \\ w_m \end{bmatrix} \; and \; x = \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_m \end{bmatrix},$$ then the net input $z = w_1x_1 + w_2x_2 + \ldots + w_mx_m$

more concisely, we can say $\mathbf{z} = \mathbf{w^T x}$ Where $w^T = [w_1 \ldots w_m] \; and \; x = \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_m \end{bmatrix},$

* A Refresher on Matrices and Matrix Multiplications

$$w^{n \times m} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_m^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_m^{(2)} \\ & & \vdots & \\ x_1^{(n)} & x_2^{(n)} & \dots & x_m^{(n)} \end{bmatrix}$$

Is a (n x m) matrix with n rows and m columns (features)

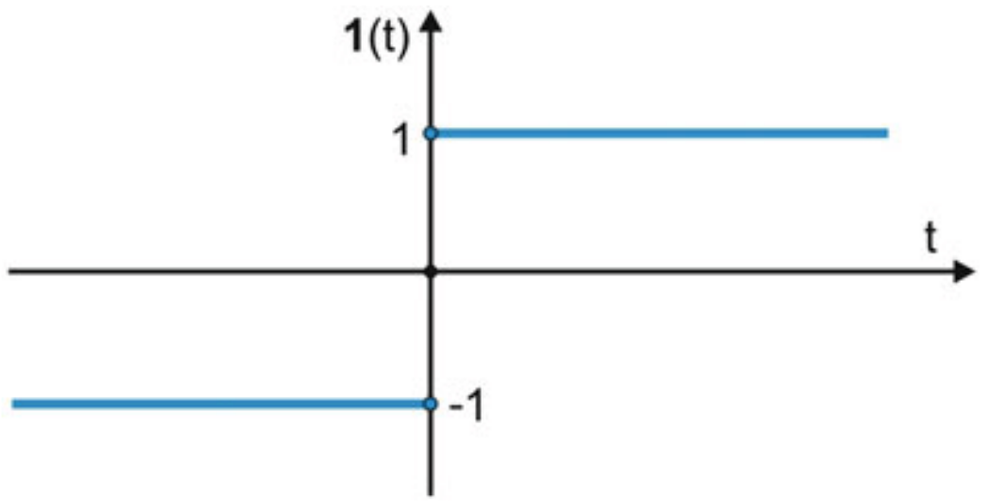
Matrices with a single row (1 x m) are called row vectors, $w = [w_1 \dots w_m]$

Matrices with a single columns (n x 1) are called column vectors $w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

The rule for matrix multiplication, however, is that two matrices can be multiplied only when the number of columns in the first equals the number of rows in the second (i.e., the inner dimensions are the same, m for an **(n×m)-matrix times an (m×p)-matrix, resulting in an (n×p)-matrix**

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^{(2\times 3)} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix}^{(3\times 1)} = \begin{bmatrix} 1\times 7 + 2\times 8 + 3\times 9 \\ 4\times 7 + 5\times 8 + 6\times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

In case of perceptron, the decision function is a variant of **the step function.**



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

Where $z = w^T x$ (Linear combination)

- **INTRODUCING THE BIAS UNIT**

Now, for simplicity, if we take the Threshold θ to the left side of the inequality in the equation, we can define a **weight-zero** where $w_0 = -\theta$ and $x_0 = 1$, such that $w_0 x_0 = -\theta$, we can write Z more succinctly as:-

$z = \mathbf{w_0 x_0} + w_1 x_1 + \ldots w_m x_m = w^T x$ AND $\phi(z) = \begin{cases} 1 & if\ z \geq \mathbf{0} \\ -1 & otherwise \end{cases}$

In Machine Learning, this weight-zero or negative threshold $w_0 = -\theta$ is called the **BIAS UNIT.** This makes the weight vector size = 1 + sizeof(x features)

Perceptron learning rule

The value for updating the weights at each increment is calculated by the learning rule

$$w = w + \Delta w_j$$

WHERE

Δw_j = Weight Update

$$\Delta w_j = \eta \; (\; target^{(i)} - output^{(i)} \;) \; x_j^{(i)}$$

η = Learning Rate

η, the learning rate (a constant between 0.0 and 1.0), “target” is the true class label, and the “output” is the predicted class label.

In the cases where the perceptron predicts the class label correctly, the weights remain unchanged:

when true class label is 1 and the prediction is also 1

$$\Delta w_j = \eta \; (\; 1^{(i)} - 1^{(i)} \;) \; x_j^{(i)} = 0$$

when true class label is -1 and the prediction is also -1

$$\Delta w_j = \eta \; (\; -1^{(i)} - -1^{(i)} \;) \; x_j^{(i)} = 0$$

In the case of wrong prediction, the weights are "**PUSHED**" towards the direction of the positive or negative target class, respectively:

when true class label is 1 and the prediction is -1 $\Delta w_j = \eta \; (\; 1^{(i)} \; - \; -1^{(i)} \;) \; x_j^{(i)} = \eta \; (2) \; x_j^{(i)}$

when true class label is -1 and the prediction is 1 $\Delta w_j = \eta \; (\; -1^{(i)} - 1^{(i)} \;) \; x_j^{(i)} = \eta \; (-2) \; x_j^{(i)}$

NOTE

it is important to note that all weights in the weight vector are being updated simultaneously for each sample i.e, we don't recompute the output before all the weights are updated. thus, for a 2-dimensional dataset, the updates would update each weight individually in each epoch:

$$\Delta w_1 = \eta \; (\; target^{(i)} - output^{(i)} \;) \; x_1^{(i)} \qquad \Delta w_2 = \eta \; (\; target^{(i)} - output^{(i)} \;) \; x_2^{(i)}$$

Example for Intuition:-

Let us assume that $x_j^{(i)}$ = 0.5 is misclassified as -1, which means, the net input DID NOT cross the threshold. We would increase the corresponding weight by a positive factor, so that the next time around the net output is more likely to be above the threshold of the unit step function and be classified correctly as 1

$$\Delta w_j = \; (\; 1 \; - \; -1 \;) \; 0.5 = (2) \; 0.5 = 1$$

Also, the weight is proportional to the value of $x_j^{(i)}$, I.e bigger the misclassification, bigger the correction. For eg if $x_j^{(i)}$ were = 2 that was incorrectly classified as -1, the decision boundary will be pushed even further. $\Delta w_j = \; (\; 1 \; - \; -1 \;) \; 2 = (2) \; 2 = 4$

Python Code to train a perceptron model ***imported*** form sci-kit learn library

```python
from sklearn.linear_model import Perceptron

# Create a perceptron object with the parameters:
# 40 iterations (epochs) over the data, and a learning rate of 0.1
ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)

# Train the perceptron
# X_train_std is the standardized training set.
ppn.fit(X_train_std, y_train)

# Apply the trained perceptron on the X data to make predicts for the y test data
y_pred = ppn.predict(X_test_std)

# Then you find out the accuracy of the classifier model,
# which is: 1 - (observations predicted wrong / total observations)
#
```

What we need:-

Look at the sci-kit api , we need to initialize a Perceptron class instance with some given parameters

1. The learning rate (eta (η)).

2. The number of iterations after which to stop.

3. A random state as a seed to the random number generator for weights, for consistent reproducibility.

That instance must contain:-

- A **Fit** method :- which fits the model on training data.

- A **Predict** method :- which predicts the classification of new data.

- Some sort of metric to **calculate efficiency / Accuracy** of model.

Writing your own Perceptron Classifier in Python

The init method

```python
import numpy as np


class Perceptron:

    """
    parameters:-
    ----------

    eta: float
         learning rate (number b/w 0.0 and 1.0)

    n_iter: int
         iterations over training set.

    random_state: int
         Random number generator seed.
    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
```

The net_input method

```
    def net_input(self, X):
        """Calculates the net input"""

        return np.dot(X, self.w_[1:]) + self.w_[0]
```

The predict method

```
    def predict(self, X):
        """Returns the class label after applying unit step"""

        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

The fit method

```python
def fit(self, X, Y):
    """Fits the training data onto the model.

    parameters:-
    ----------

    X : {feature matrix with training samples, array like (numpy object)},
        shape = [n_samples, n_features]

    y : {output vector with target values, array like (numpy object)}
        shape = [n_samples]
    """

    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1+X.shape[1])

    self.errors_ = []

    for _ in range(self.n_iter):
        errors = 0
        for xi, target in zip(X, Y):
            update = self.eta * (target - self.predict(xi))
            self.w_[1:] += update * xi
            self.w_[0] += update
            errors += int(update != 0.0)
        self.errors_.append(errors)
    return self
```

Training and plotting the Perceptron

```
import pandas as pd
import numpy as np
from perceptron import Perceptron
import matplotlib
matplotlib.use("TkAgg")

from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap

df = pd.read_csv('../datasets/iris.data', header=None)
```

```
ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, Y)

plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Number of Misclassifications')

print('Errors: ', ppn.errors_)
print('see the lovely converging graph now : )')

plt.show()
```

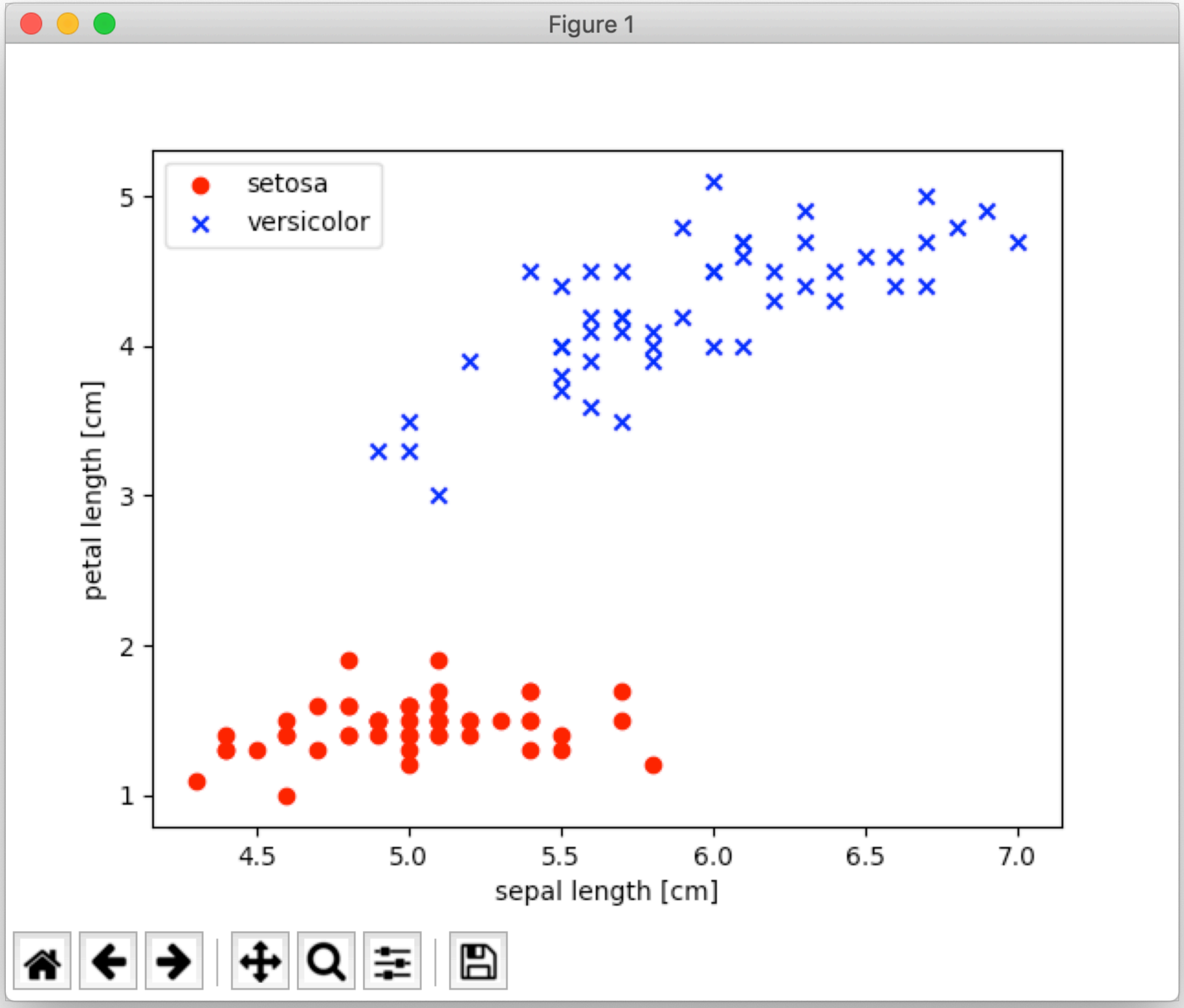
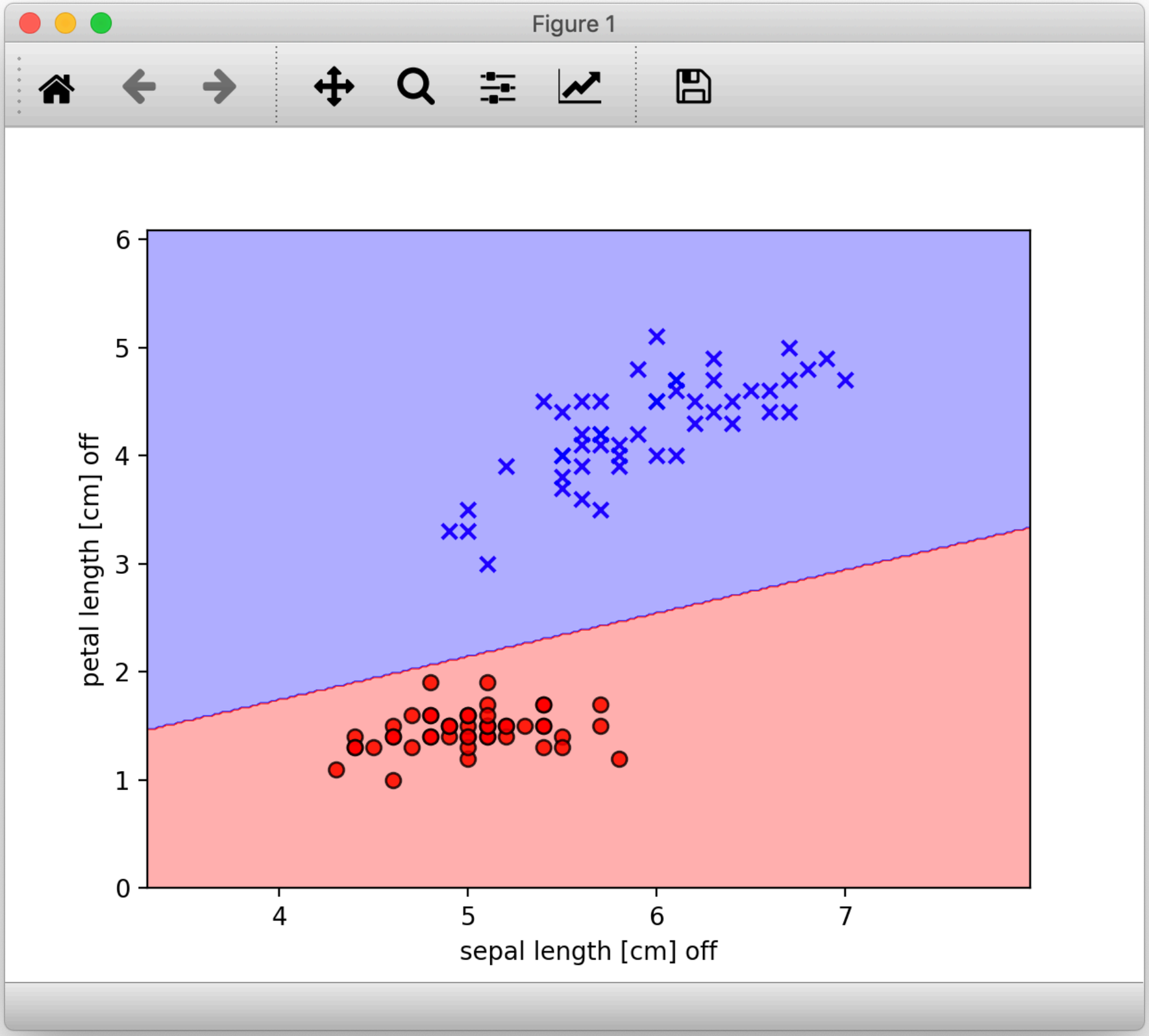
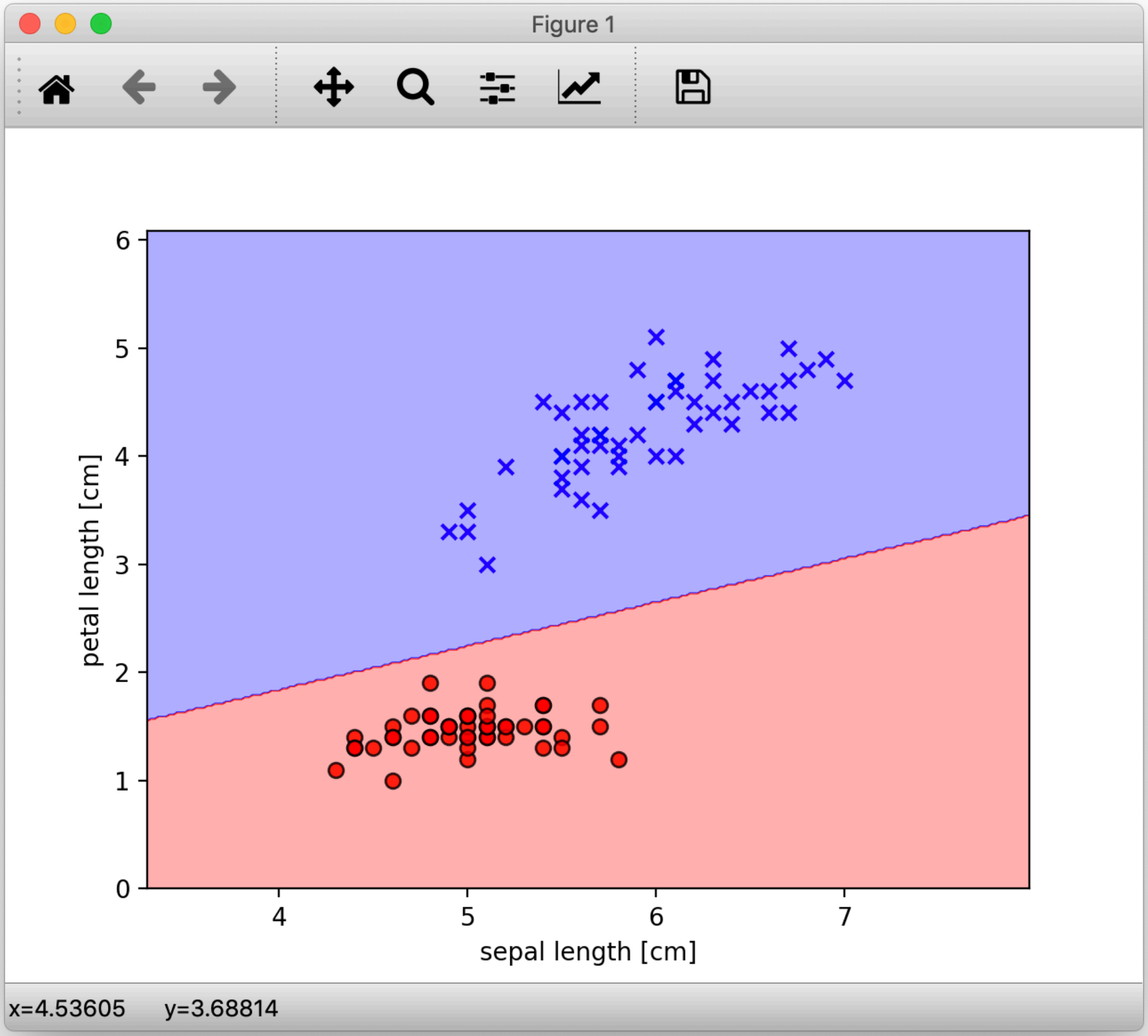


Figure 1
setosa
versicolor
petal length [cm]
sepal length [cm]
1
2
3
4
5
4.5
5.0
5.5
6.0
6.5
7.0

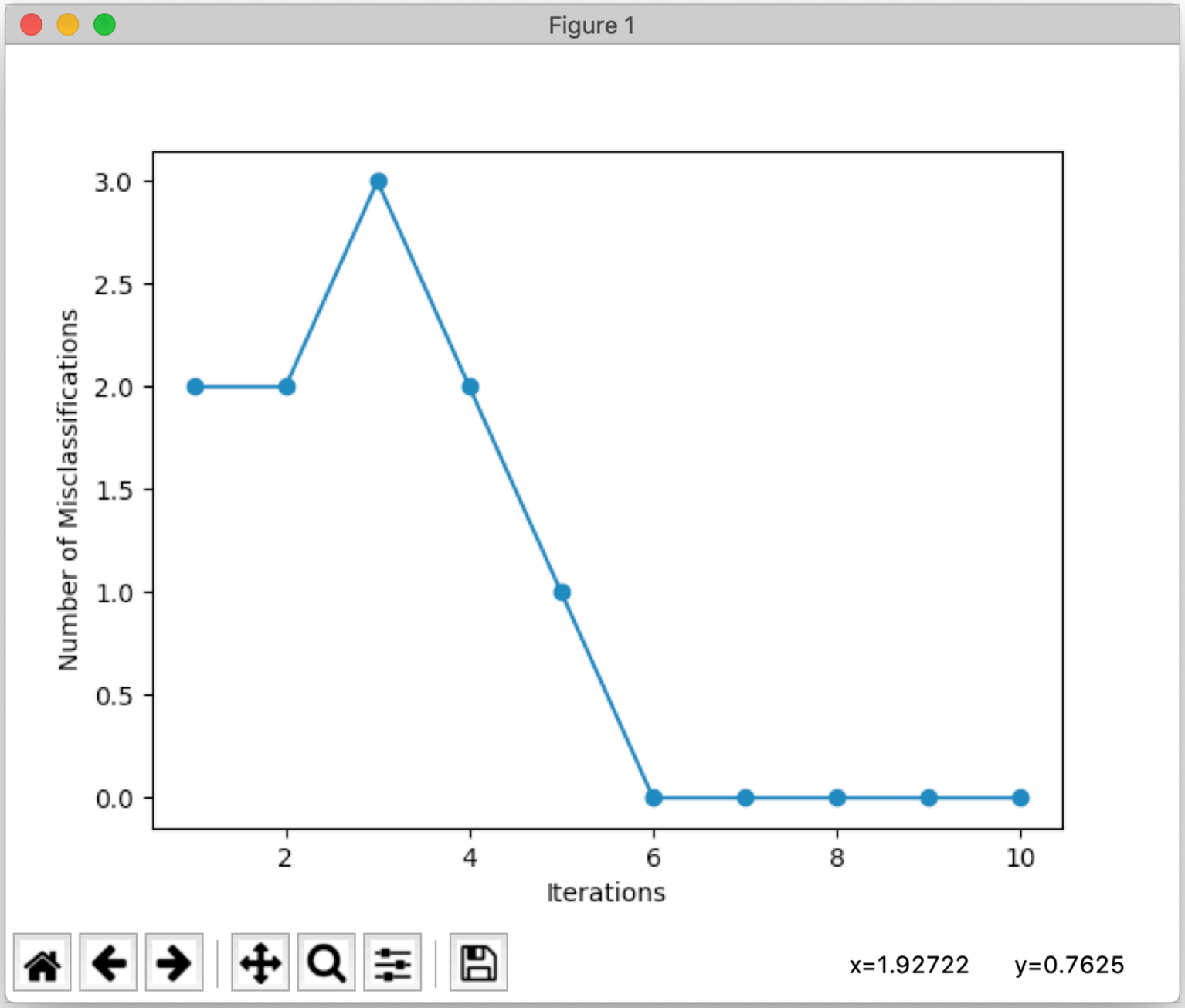
How the scikit-learn Model performed



How our handmade model performed



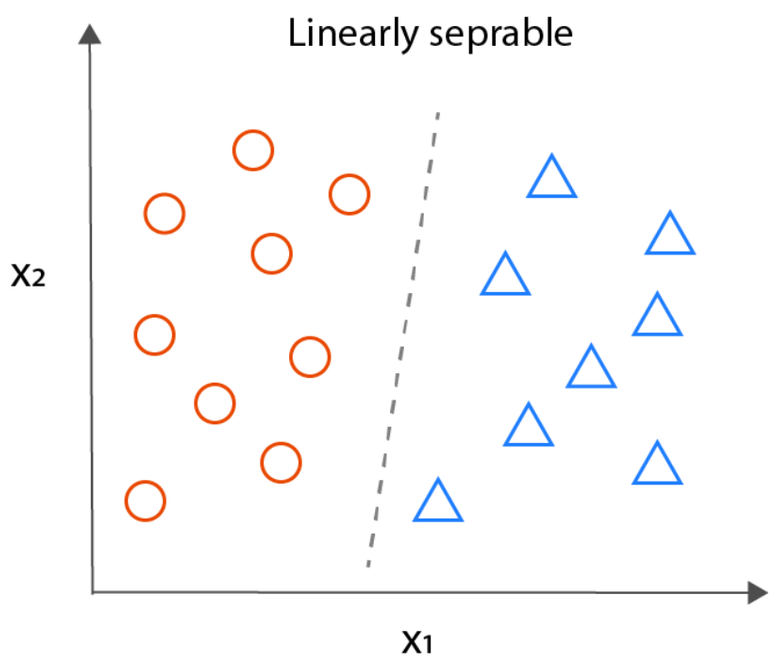
```
Python 3.7.3 (default, Mar 27 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> python.el: native completion setup loaded
>>> Errors:  [2, 2, 3, 2, 1, 0, 0, 0, 0, 0]
see the lovely converging graph now : )
>>>
```

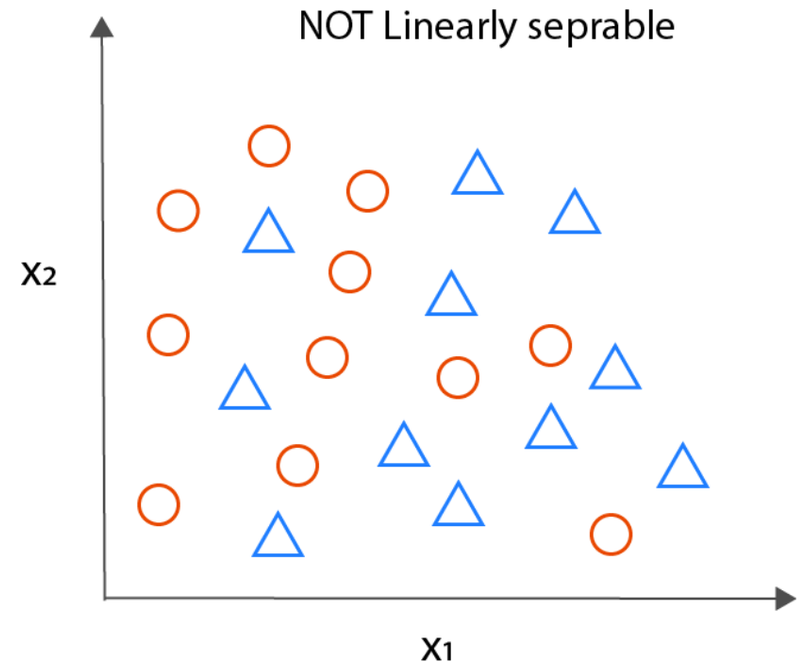


Drawbacks

The Perceptron, even though simple in design and elegant in its working has glaring drawback. IT Only seems to work when

- THE CLASSES ARE LINEARLY SEPARABLE i.e a clear linear separation / boundary
- Learning rate is not too high





If these conditions are not satisfied, the learning rule will never stop updating since some samples will always be mis-classified and would need weight updates, changing the decision boundary.

Since most of the data in the wild is rarely linearly separable, so, to overcome these, we set

- The limit the max number of passes over dataset (epochs)

  And / or

- Threshold for number of misclassifications.

The Adaptive Linear Neuron or AdaLiNe algorithm

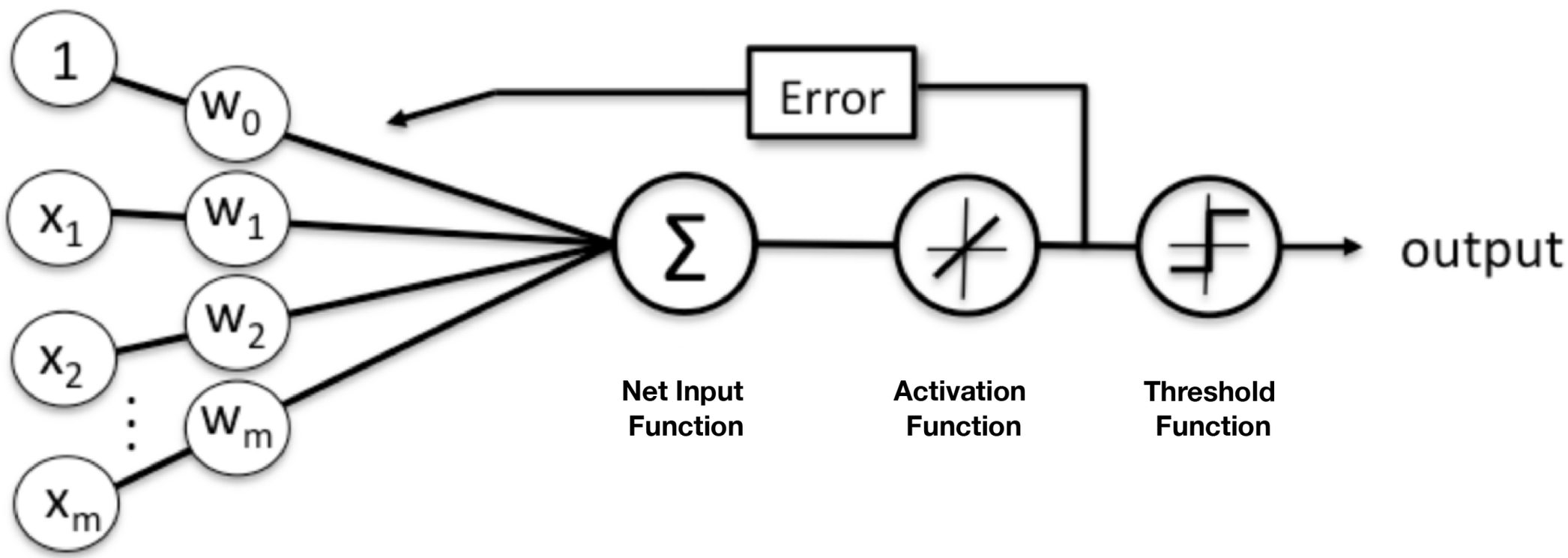
Importance.

Adaline is similar to perceptron, with a few differences and addition of a few key new concepts

- The Adaptive Linear Neuron or ADALINE is a binary classification algorithm and a single layer neural network which was published by **Bernard Widrow** and his doctoral student **Tedd Hoff** (also known as *Widrow-Hoff* rule) shortly after Rosenblatt's perceptron algorithm and can be thought of as an improvement on the later.

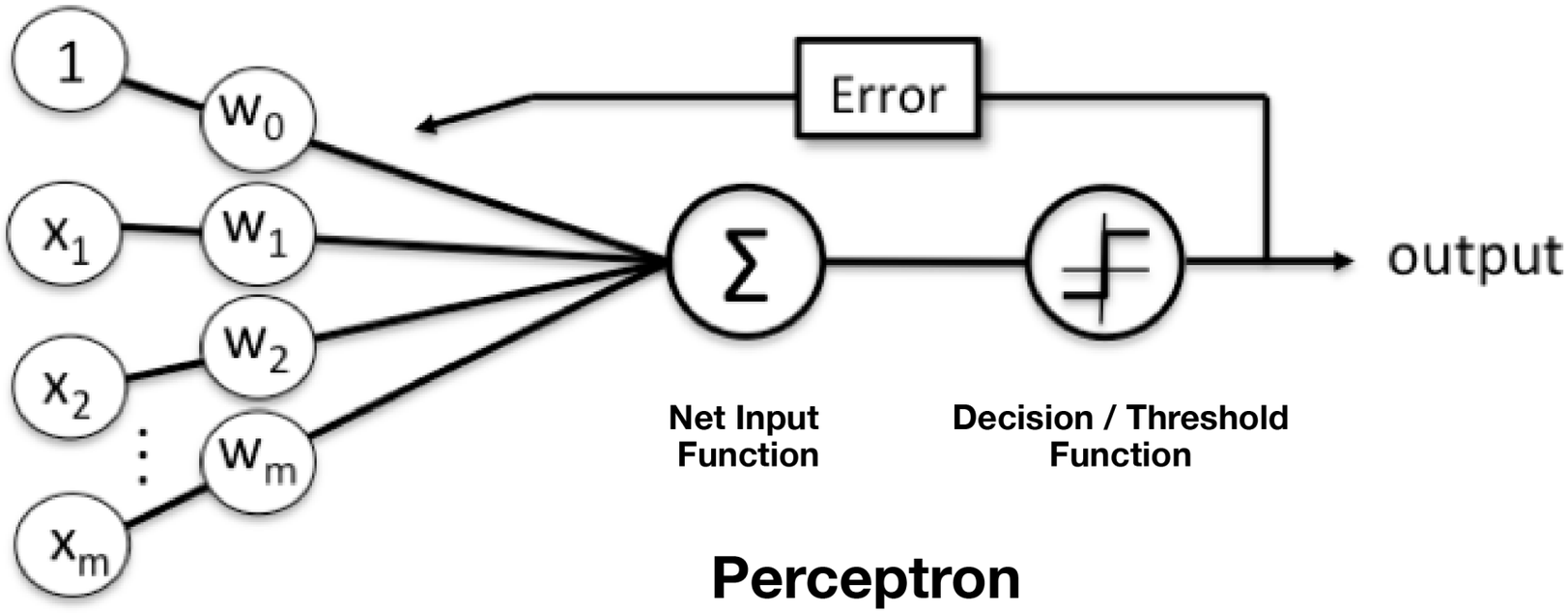
- Illustrates the **key concept of defining and optimising a continuous cost function** which really lays the foundational groundwork for understanding more sophisticated Machine learning algorithms like Logistic Regression, Support - Vector Machines etc.

The Schematic

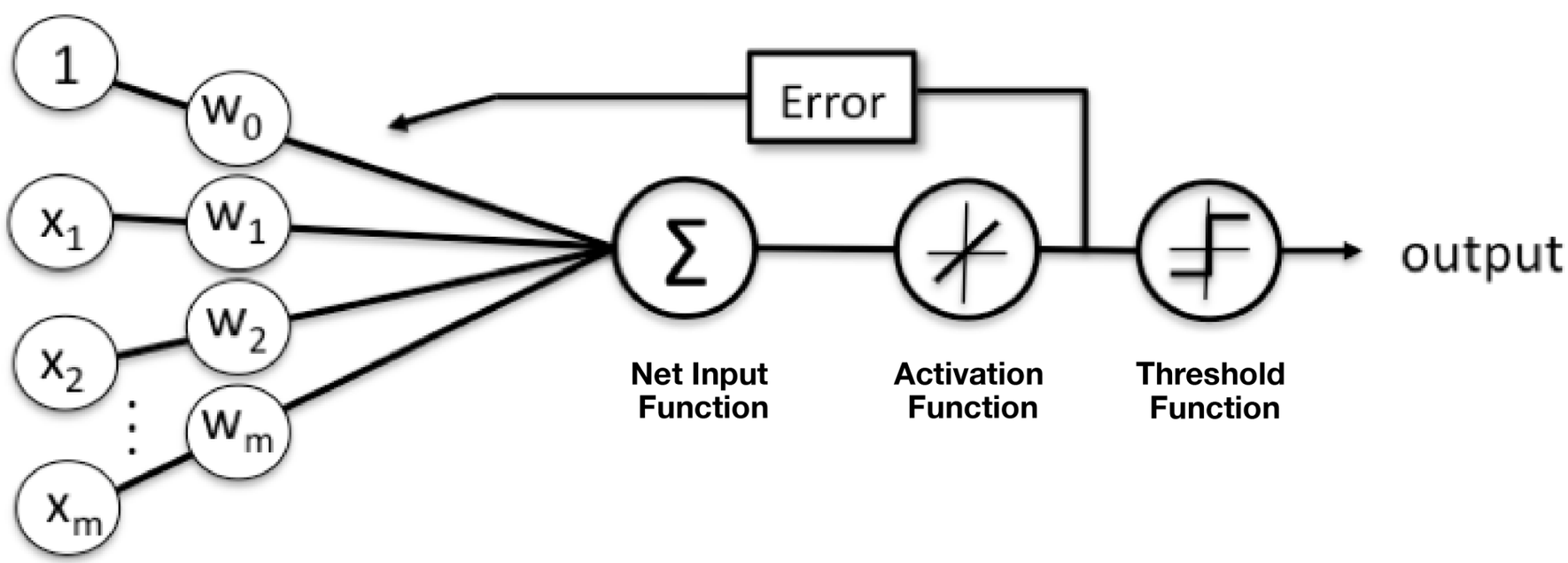


ADALINE

The Difference



Perceptron



ADALINE

Key Differences

- The Weights in Adaline are updated via the activation function and not the threshold function like Perceptron

- The Activation function in Adaline is simply the identity function of the net input

  $\phi(z) = z,$ where $z = w^T x$

  which is real-valued i.e not necessarily an integer like 0 or 1.

- Perceptron compares the true-class labels against the predicted class labels and then calculates the error, whereas, Adaline compares this real valued output to TRUE-CLASS LABLES and the error is decided based on the calculation.

- Adaline uses COST-FUNCTION MINIMIZATION and GRADIENT DESCEND

- In Perceptron, the Weights are updated incrementally after each data sample is processed, whereas in case of Adaline, the weight updates are calculated based on all training samples in the training set.

Cost Function Minimization

One of the central features of machine learning algorithms, esp. supervised ML algorithms is a **well defined objective function that is to be optimized while the algorithm learns.** This objective function often present as a cost function, which, as the name suggests, needs to be minimized.
The Algorithm then learns in a way such as to optimise (in our case, as we'll see, minimize the cost) for this objective function.

For example:-

When travelling, you could Optimize for:-

- Time:- you would then take actions that'll take you to your destination the fastest, no matter what they cost.
- Cost:- you would probably take cheaper means of transport even if they take more time.

The cost function for Adaline is defined as **sum of squared errors (SSE),** which in turn would be used to learn the weights by the algorithm.

$$J(w) = \frac{1}{2} \sum_{i} \left(y^{(i)} - \phi(z^{(i)}) \right)^2$$

$y^{(i)}$ = true class label, $\phi(z^{(i)})$ = net output from linear activation function (real - valued)

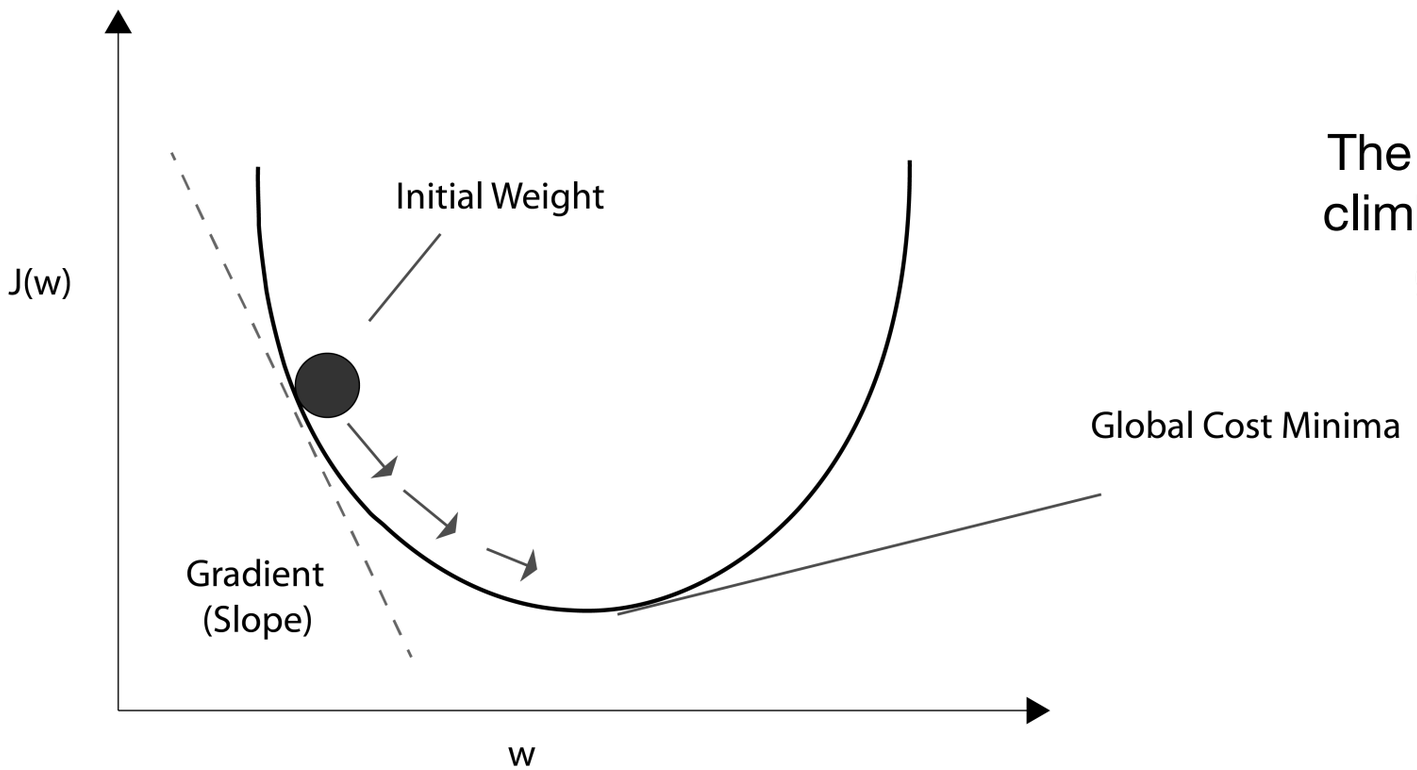
Benefits of this cost function

- The continuous linear activation function makes the cost function differentiable, a property which will come in handy when make use of the gradient (slope).

- The Cost function is convex (an upward opening parabola).

Both of these above factors, when combined, enable us to use a powerful optimisation technique called **GRADIENT DESCENT**, which really helps us find weights that optimally minimise the cost function for our classification task.

Gradient Decent



The Main idea here can be thought of as climbing or going downhill until a local or global cost minimum is reached.

In each pass over data, we **take a step in the opposite direction** of the gradient (which could either be positive or negative).

(thus ensuring moving towards minimum direction in a convex function)

(if the learning rate is too high, we WILL overshoot, thus a well-suited rate is required)

The step size is determined by the learning rate we decide upon when starting the classifier.

Adaline Learning Rule

The Adaline learning rule

$$w := w + \Delta w$$

Seems pretty similar to Perceptron learning rule

Here, Using Gradient Decent technique, we update the weights by taking a step in the opposite direction of the gradient $\nabla J(w)$ of our cost function $J(w)$

$$\Delta w = -\eta \nabla J(w)$$

where, η is the learning rate.

To Compute the Gradient of the cost function, we need to compute the partial derivative of the cost function w.r.t each weight w_j

IF $$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

(This is what we get when we partially differentiate our S.S.E cost function, notice the minus sign)

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi\left(z^{(i)}\right)\right) x_j^{(i)}$$

Since we update all weights Simultaneously, our a Adaline rule is Still

$$w := w + \Delta w$$

-> this Can be expressed as the dot product of feature matrix X and the error vector.

Writing your own AdaLiNe Classifier in Python

The init method

```python
class AdalineGD:
    """
    The Adaline Classifier

    parameters:-
    ----------

    eta: float
         learning rate (number b/w 0.0 and 1.0)

    n_iter: int
         iterations over training set.

    random_state: int
         Random number generator seed for consistent seeds
    """

    """
    Attributes

    w_: 1d-array,
        weights after fitting

    cost_: list
           sum-of-squares cost function values in each iteration
    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):

        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
```

The net_input and activation methods

```
    def net_input(self, X):
        """
        calculate net_input by linear combination of weights and feature matrix X
        """
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, net_input):
        """Compute linear activation"""
        return net_input
```

The predict method

```
    def predict(self, X):
        """Return class label the model predicts after the unit-step threshold fn"""
        return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)
```

The Fit method

```
    def fit(self, X, y):
        """Fits the training data onto the model.

        parameters:-
        ----------

        X : {feature matrix with training samples, array like (numpy object)},
            shape = [n_samples, n_features]

        y : {output vector with target label values, array like (numpy object)}
            shape = [n_samples]
        """

        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1+X.shape[1])

        self.cost_ = []

        for _ in range(self.n_iter):
            cost = 0
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0

            self.cost_.append(cost)

        return self
```

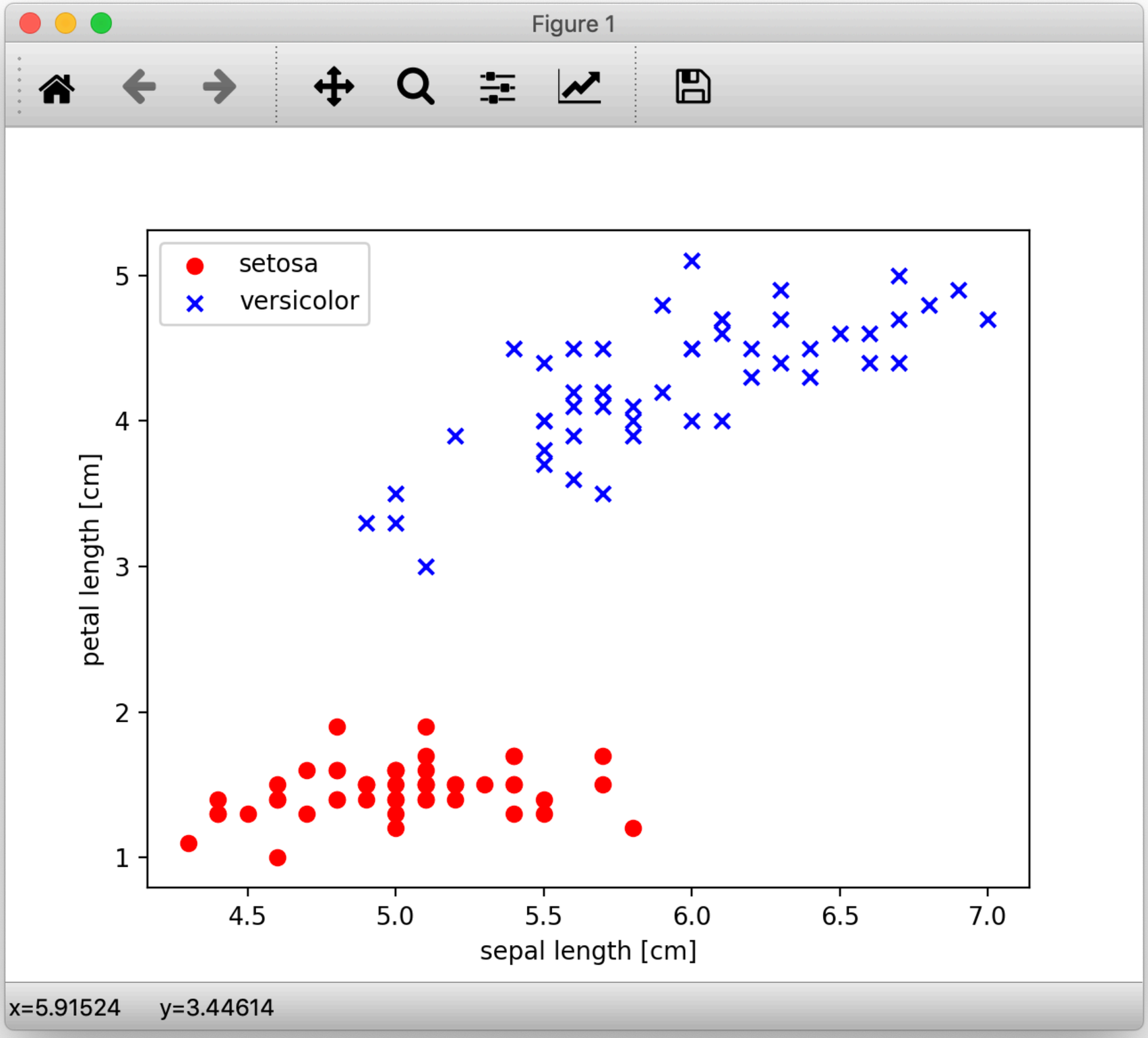
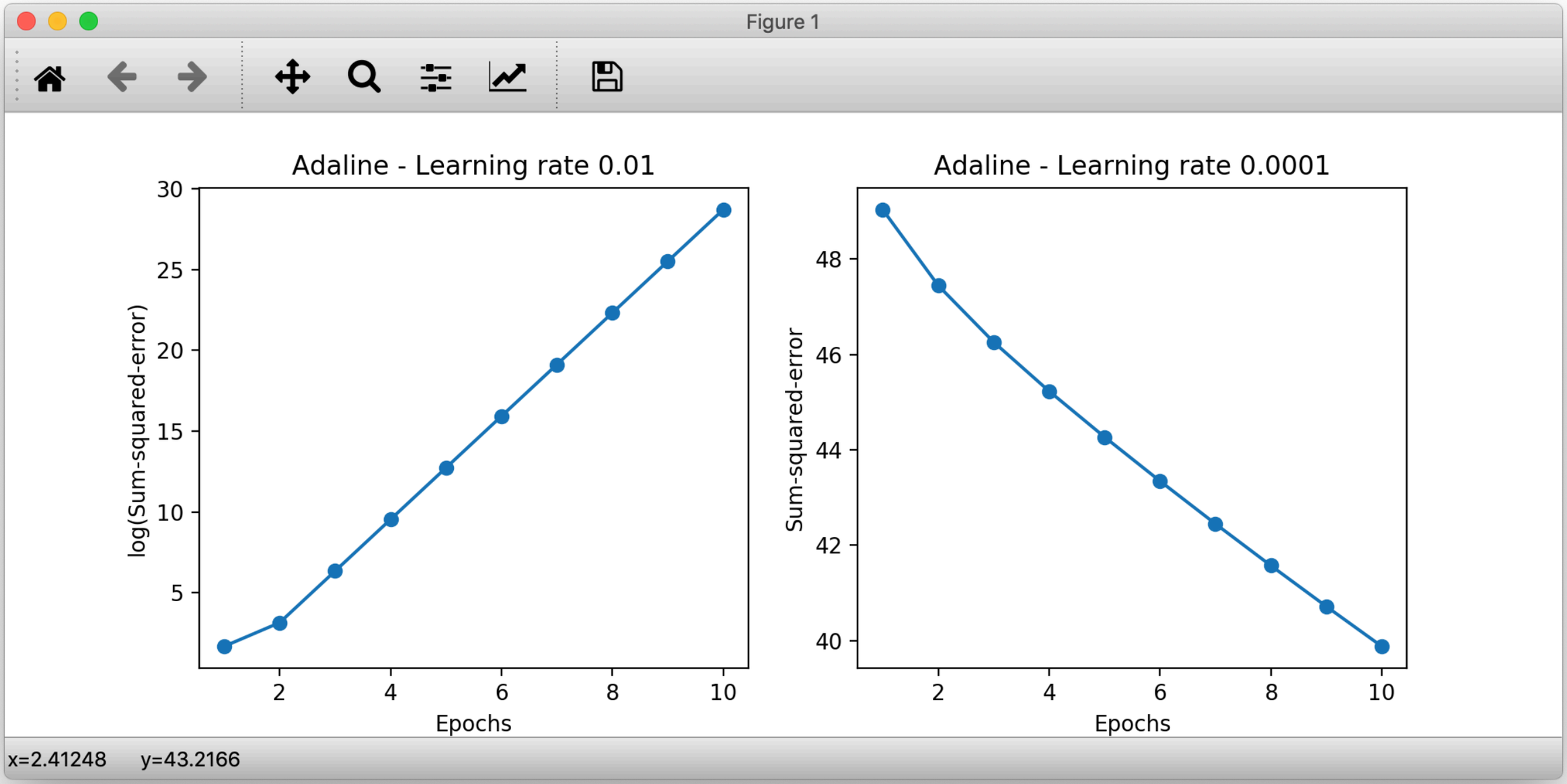


Figure 1
setosa
versicolor
petal length [cm]
sepal length [cm]
1
2
3
4
5
4.5
5.0
5.5
6.0
6.5
7.0
x=5.91524 y=3.44614

```python
import numpy as np
import pandas as pd
from adaline import AdalineGD
import matplotlib
matplotlib.use("Qt5Agg")

from matplotlib import pyplot as plt

df = pd.read_csv('../datasets/iris.data', header=None)

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
ax[0].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('log(Sum-squared-error)')
ax[0].set_title('Adaline - Learning rate 0.01')

print('cost1: ', ada1.cost_, '\n')

ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
ax[1].plot(range(1, len(ada2.cost_) + 1), ada2.cost_, marker='o')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Sum-squared-error')
ax[1].set_title('Adaline - Learning rate 0.0001')

print('cost2: ', ada2.cost_)

plt.show()
```

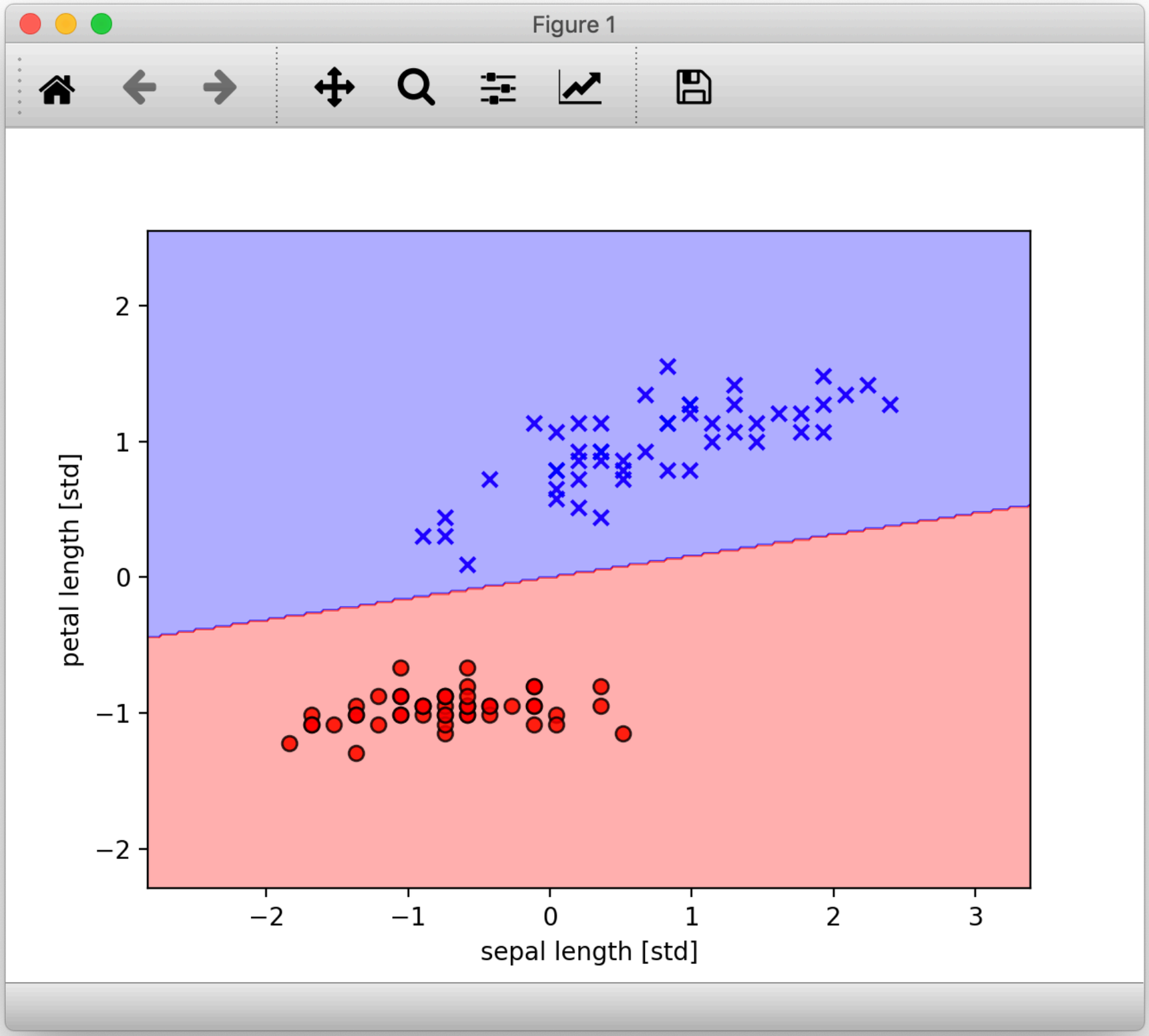
* Notice the two instances of Adaline classifier with same data but different learning rates.



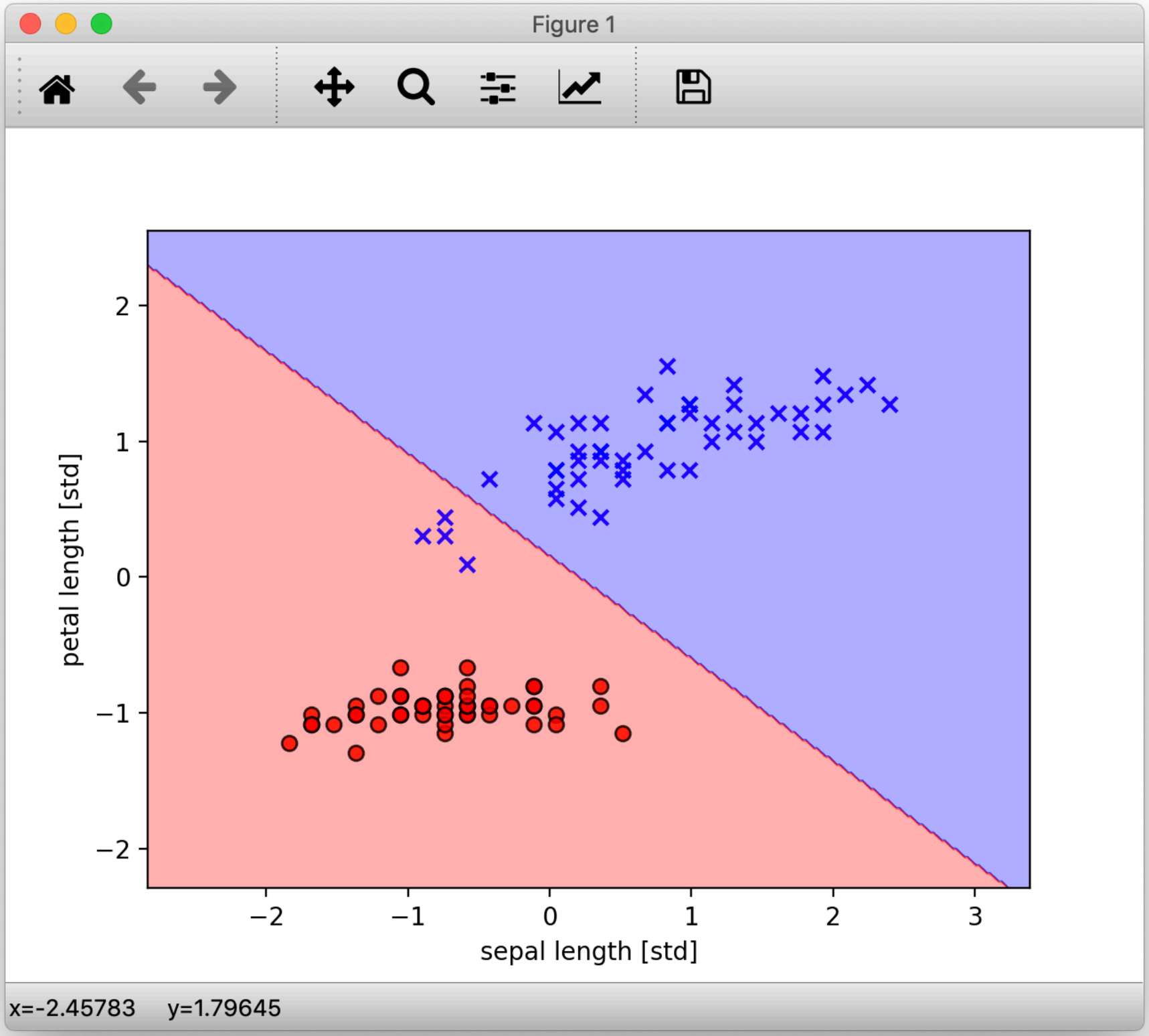

```
cost1:  [49.035525413753035, 1428.229565506252, 2223702.737494616, 3469517688.3901963, 5413298509749.252, 84460733
18799870.0, 1.3177945826948438e+19, 2.056082745948309e+22, 3.2079933501785183e+25, 5.005256405691528e+28]

cost2:  [49.035525413753035, 47.44671338997408, 46.26095419934007, 45.23154116965617, 44.27083462863981, 43.347515
26669241, 42.45020904163226, 41.57460931914628, 40.71891764198594, 39.882229510658284]
```

* Notice the Costs of two Adaline classifiers with same data but different learning rates.

How the model performed with learning rate 1



How the model performed with learning rate 2



The Logistic Regression Algorithm.

Logistic Regression

Logistic Regression is one of the most widely used algorithms for classification in the industry today.

Inspite of the name, Logistic Regression, just like the previously discussed Adaline model is a linear model for **binary classification and NOT regression.**

It just as-well can be extended to work as a multi-class classifier via using techniques like the OneVersesRest(OvR) technique.

The reasons for Logistic Regression's popularity?

- It is an relatively easy model to implement and understand

- it deals more in the probabilities that a given sample belongs to a certain class than binary yes or No's.

Logistic Regression is a lot similar to Adaline model but, **with a different activation and cost functions**, as we will see, this is where it all comes together.

New Terms and Functions

- **Odds ratio:**

The odds in favour of a particular event, the odds ratio can be written as $\frac{p}{1-p}$ where p is the probability of the positive* event.

(* positive = the event we are looking for, not necessarily in a good context, eg: presence of a disease)

- **LOGIT Function:-**

$logit(p) = log\frac{p}{1-p}$, **or more clearly as the natural logarithm of the odds ratio.**

Where again p is the probability of positive event.

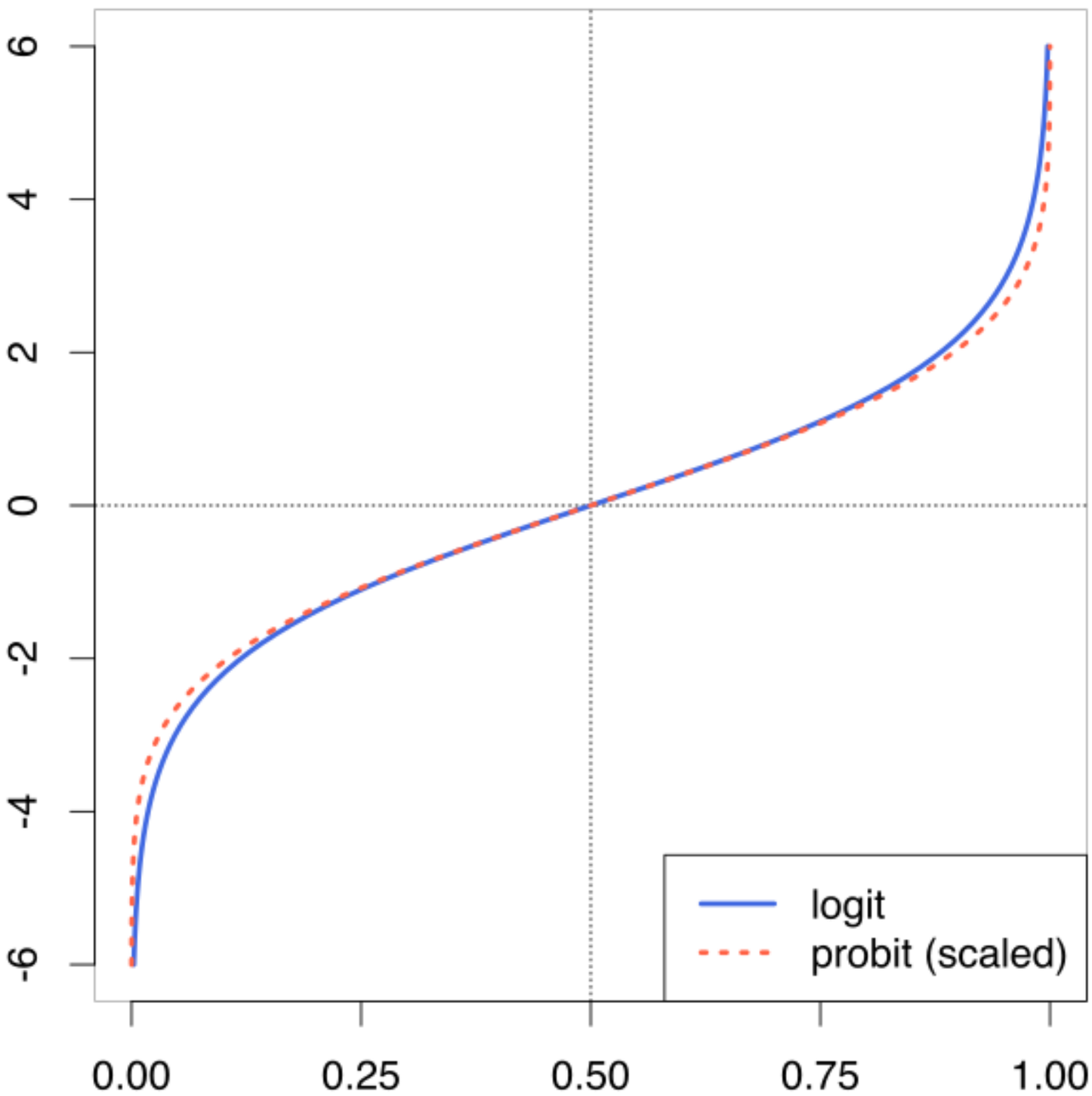
The logit function takes input values in the range 0 to 1 (THE PROBABILITY) and transforms them to values over the entire real-number range. i.e $[0,1] -> (-\infty, +\infty)$,

which we can use to express a linear relationship between feature values and log-odds:

$$logit(p(y=1|x)) = w_1x_1 + w_2x_2 + \ldots + w_mx_m = \sum_{i=0}^{m} w_ix_i = w^Tx$$

here, $p(y=1|x)$ is the conditional probability that a particular sample belongs to class 1 given features x.

What LOGIT function looks like



Input Domain (z):- [0, 1]

Output Range $\phi(z)$:- $(-\infty, +\infty)$

- **SIGMOID Function:-**

we are actually interested in predicting the probability that a certain sample belongs to a class, **which is the inverse form of logit function.** It is also called the **logistic sigmoid function**, often referred to as simply the *sigmoid function* because of its characteristic S-shape:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

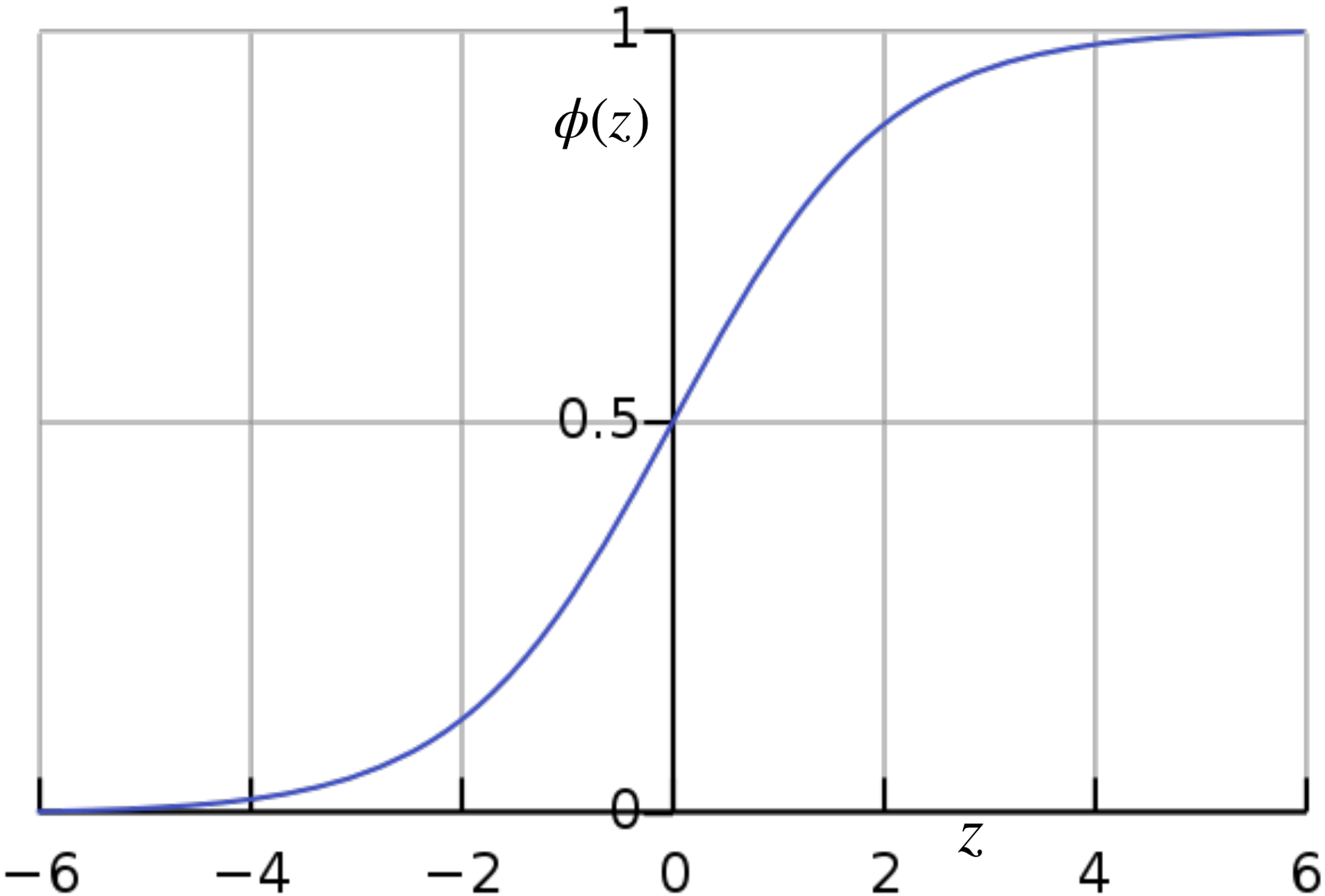
Where z is the net input, i.e. the linear combination of the weights and sample features, $z = w^T x = w_0 x_0 + w_1 x_1 \ldots w_m x_m$

its input domain, z is from $(-\infty, +\infty)$ and output Range is [0, 1], Exactly the opposite of the logit function that we've seen earlier. This is perfect as the function takes in real number values and outputs a value between [0, 1] inclusive which can be used as a probability measure.

Input Domain:- $(-\infty, +\infty)$

Output Range:- [0, 1]

What SIGMOID function looks like



We can see from the definition that :-

$\phi(z) -> 1$, if
$z -> \infty$, as

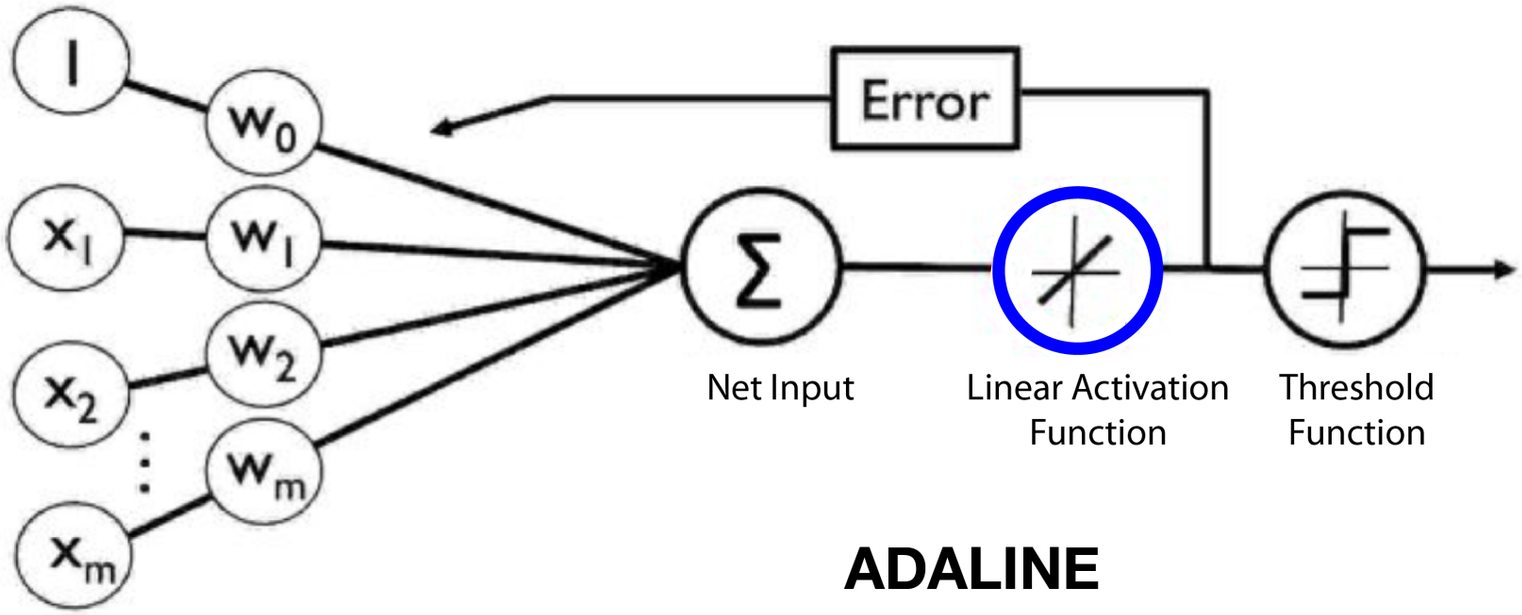
e^{-z} becomes dismally small and in the denominator

AND

$\phi(z) -> 0$, when
$z -> -\infty$

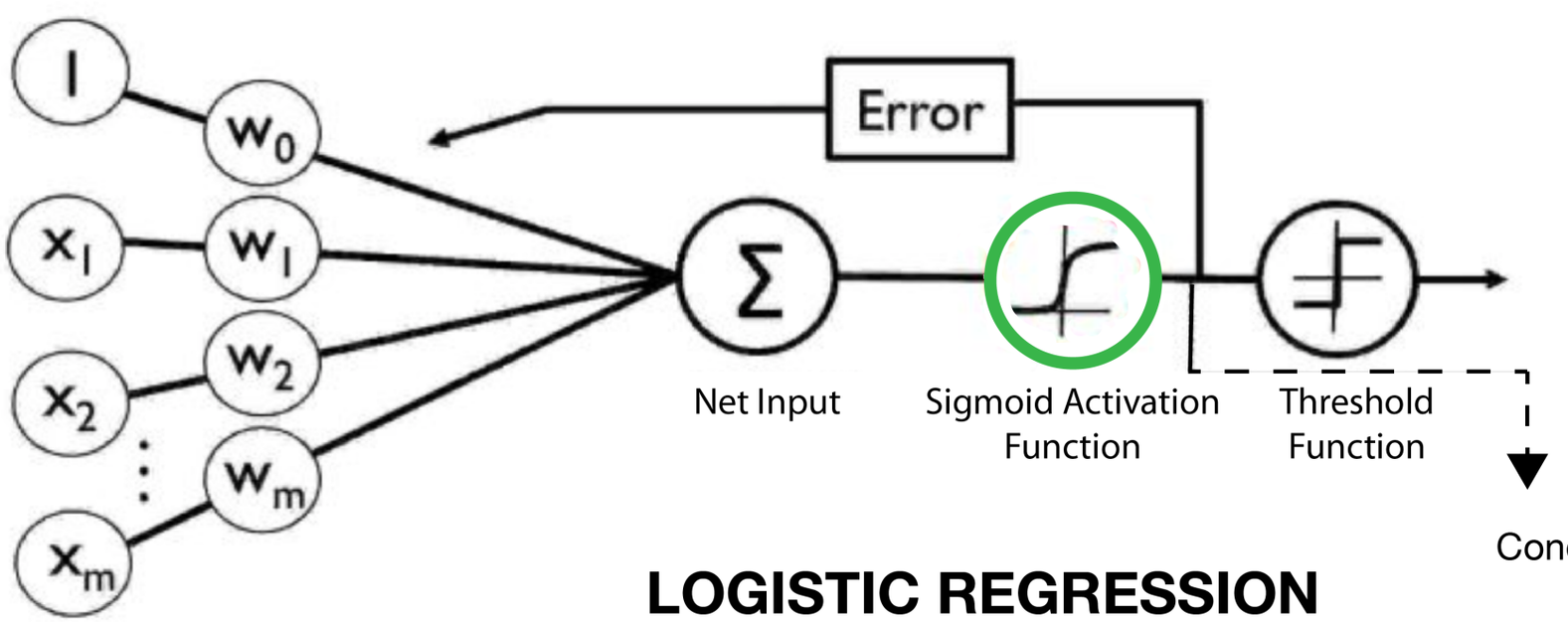
- There's an intercept at $\phi(0) = 0.5$

The Schematic



Prediction

ADALINE



Prediction

Conditional probalbility that a sample belongs to class 1, given its features x and weights w

LOGISTIC REGRESSION

The Cost Function

Remember in Adaline, we used the cost function S.S.E, which we minimized in order to learn weights.

For logistic regression, we want to **maximise the likelihood that a sample belongs to a certain class y, given features x** and with learned weights w. Let us call this likelihood function $L(w) = P(y \mid x; w)$.

In practice, it is easier to maximise the natural log of this function, which we call the log likelihood $l(w) = logL(w)$.

Now, we define the cost function $J(w)$ as -$l(w)$, **which conversely, we want to minimize.**
it would actually be equal to maximizing the log likelihood $l(w)$ but with added benefits like being able to **use gradient decent** and more mathematical ease.

Long story short, the cost function comes out to be.

- $$J(w) = \sum_{i=1}^{n} \left[-y^{(i)} \log(\hat{y}) - \left(1 - y^{(i)}\right) \log(1 - \hat{y}) \right]$$

$$J(w) = \sum_{i=1}^{n} \left[-y^{(i)} \log(\hat{y}) - \left(1 - y^{(i)}\right) \log(1 - \hat{y}) \right]$$

Where:-

- $y^{(i)}$ is the i_{th} class label in the desired true **target** (1 or 0)
- $\hat{y} = \phi(z)$, the **probability output** by the activation function, [0, 1]

To get a better intuition of the function, lets look at cost calculation of a single sample instance:

$$J(\phi(z), y; w) = -y \log(\phi(z)) - (1 - y)(\log(1 - \phi(z)))$$

$$j(\phi(z), y; w) = \begin{cases} -\log(\phi(z)) & if\ y = 1 \\ -\log(1 - \phi(z)) & if\ y = 0 \end{cases}$$

Note:- the gradient and weight updates for logistic regression comes out to be the same as Adaline after doing the math.

Converting the AdaLiNe Classifier implementation to Logistic Regression

The __init__ method

```python
import numpy as np


class LogisticRegressionGD:
    """
    A class for Logistic Regression binary classifier

    parameters:-
    ----------

    eta: float
         learning rate (number b/w 0.0 and 1.0)

    n_iter: int
         iterations over training set.

    random_state: int
         Random number generator seed for consistent seeds

    Attributes:-
    ----------

    w_: 1d-array,
        weights after fitting

    cost_: list
           cost function values in each iteration
    """

    def __init__(self, eta=0.05, n_iter=5, random_state=1):

        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
```

The net_input and activation in Logistic Regression.

```
def activation(self, z):
    # Notice the Sigmoidal Activation function
    return 1. / (1. + np.exp(-z))

def net_input(self, X):
    #The net input from linear combination of features and weights with a bias unit.
    return np.dot(X, self.w_[1:]) + self.w_[0]
```

The predict method

```
def predict(self, X):
    # Notice that the model predicts the sample to be in class 1 if
    # the probability(from the activation fn) is >= 0.5
    # else the prediction is class 0
    return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```

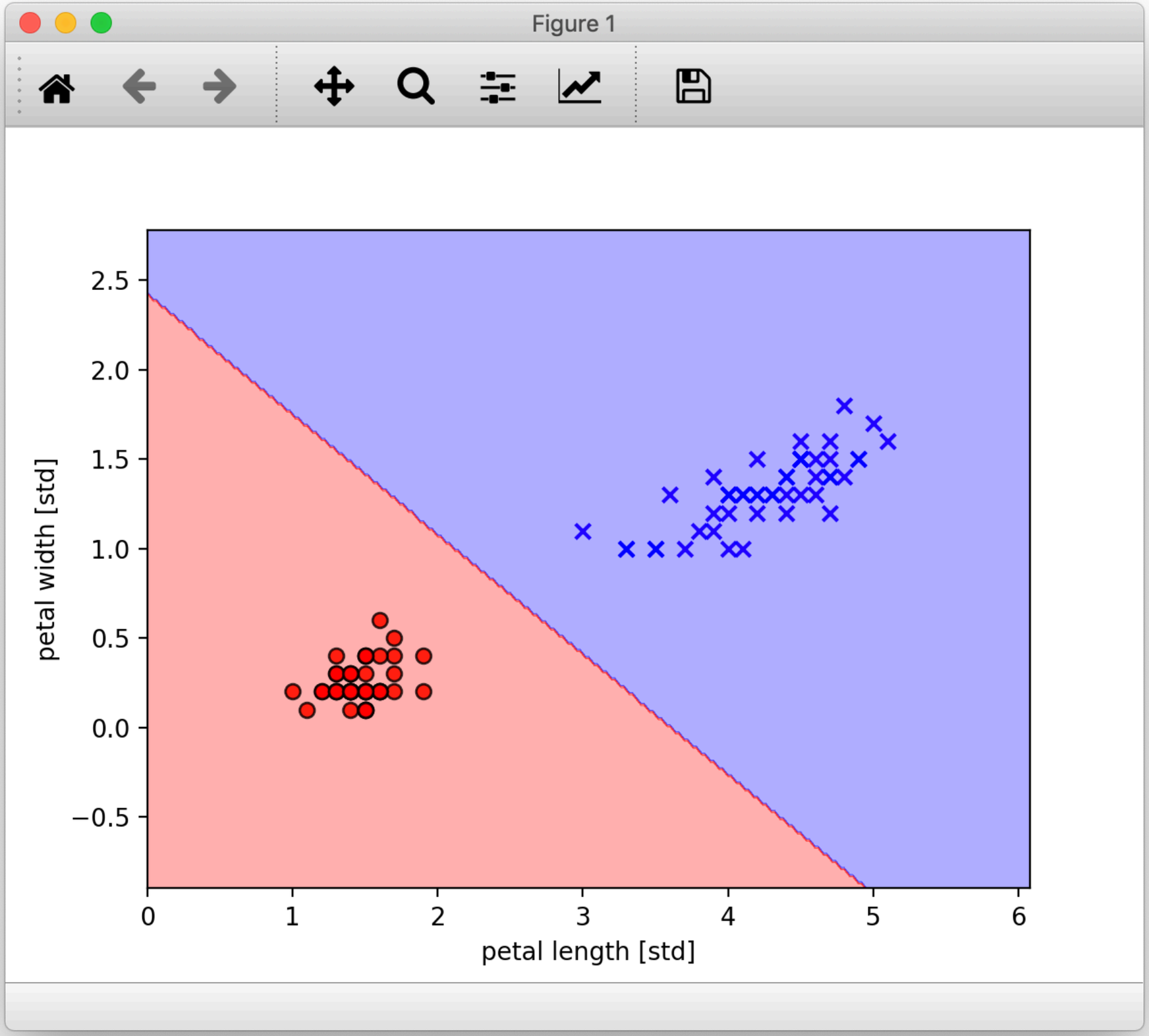
The fit method in Logistic Regression

```
def fit(self, X, y):
    """Fits the training data onto the model.

    parameters:-
    ----------

    X : {feature matrix with training samples, array like (numpy object)},
        shape = [n_samples, n_features]

    y : {output vector with target label values, array like (numpy object)}
        shape = [n_samples]
    """
    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])

    self.cost_ = []

    for _ in range(self.n_iter):
        cost = 0
        net_input = self.net_input(X)
        output = self.activation(net_input)
        errors = (y - output)
        self.w_[1:] += self.eta * X.T.dot(errors)
        self.w_[0] += self.eta * errors.sum()

        # Notice, the cost is now logistic cost instead of
        # the sum-of-squared errors cost
        cost = -y.dot(np.log(output)) - ((1-y).dot(np.log(1-output)))

        self.cost_.append(cost)

    print('costs: ', self.cost_)
```

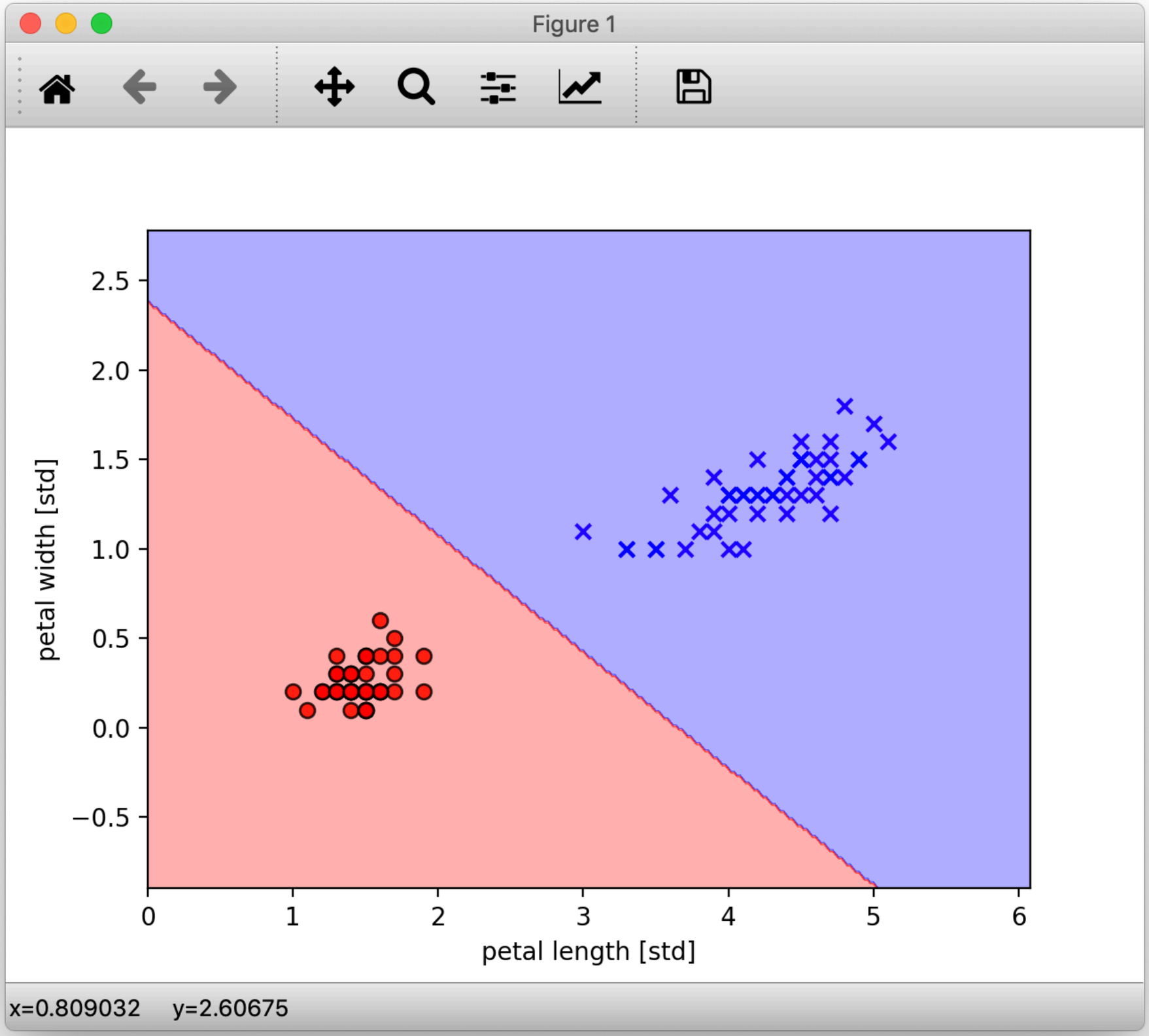
Python Code to train a perceptron model ***imported*** form sci-kit learn library

```python
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Notice the new parameter C implemented in the scikit-learn model
# This parameter has to do with a convention that comes
# from Support Vector Machines and the concept of Regularization parameter
# which are outside the scope of this talk due to time limitaions.
LR = Logisticregression(C=100.0, randomstate=1)


# Train the Logistic Regression model
# on the training set.
LR.fit(X_train, y_train)

# Use the Trained model on TEST data to get predictions
y_pred = LR.predict(X_test)

# Then you find out the accuracy of the classifier model.
print('Accuracy: %.2f' % accuracy_score(y_test, y_pred))
```

How the Scikit-Learn model performed



How our handmade model performed



Chapter 3

Summary

YOU Learned

- **YOU Learned** The Very Core fundamental Machine Learning algorithms and what happens under-the-hood when you use these classifier algorithms and how they work.
    - The Perceptron
    - The Adaline
    - The Logistic Regression

    With Powerful Techniques like Gradient Decent

    YOU Learned:-

    - The Necessary Math behind these algorithms and how they build upon each other.
    - How to implement them in python
    - How the data that you give them is utilised by these algorithms to LEARN and make a prediction.

Learnings and Outro

WHY YOU SHOULD TOO.

1. Machine Learning is much less of magic than it is made out to be by the media and is backed by hard science.

2. Data is hard.

3. Even though black-boxes exist to save you time and are often optimized and more efficient than ad hoc implementations, It's ALRIGHT to sometimes want to look inside these black boxes and learn how they work, even though you don't really NEED to.

People often give the example of driving a car and not knowing how the engine, that they use daily, works. Because they don't need to. Which is fair enough.

But, think about the days that you car won't start, or breaks down in the middle of the commute. If you know how the engine works, you can probably figure out or at least narrow down what went wrong and get it fixed and working that much faster. Which, in my humble opinion is always a better position to be in.

Libraries like Scikit-learn and others have done a tremendous job with giving us these highly optimized and easy to use tools. But when you really understand what is going on in there, it will help you in those tricky days where your model won't behave as you'd like or flat out not work. But now as you know what goes on in there and how the data flows and is utilized, you'd be that much more faster and surer to guess or narrow down what could have gone sideways and that ,i think, makes us better Engineers for it.

THANK YOU FOR LISTENING

ACKNOWLEDGEMENTS

1. Dr. Sebastian Raschka, for his wonderful Blog.

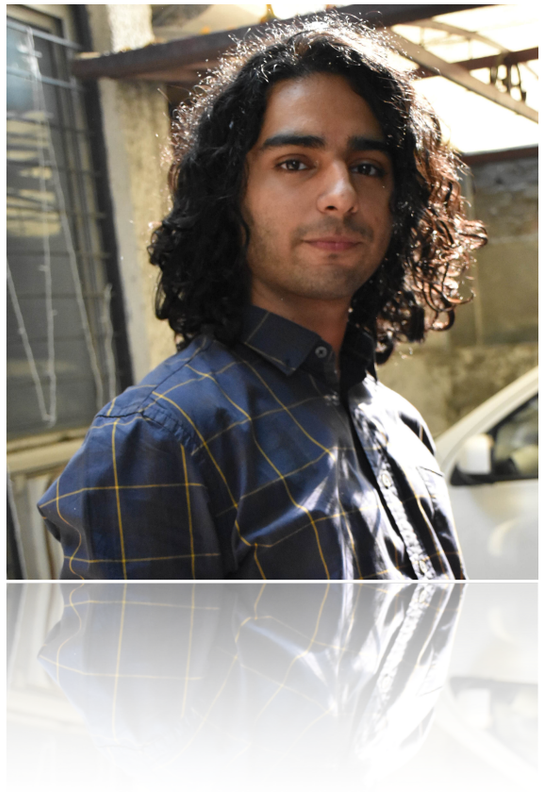
2. Dr. Sebastian Raschka and Dr. Vahid Mirjalili, for their insightful book “python machine learning.”

3. Wikipedia, for their countless articles on the topic.

4. 3Blue1Brown’s fantastic Youtube channel.

5. Karl Rosaen, for getting me started.

6. Pycon India 2019 Team. Way to go guys.



- Github: **github/arjunkathuria**
- Twitter: **@arjunslays**
- Email: **arjun.kathuria8@gmail.com**

Cue to Q/A session with the audience