Machine Learning Excercise 1

MtrNr. 6329857

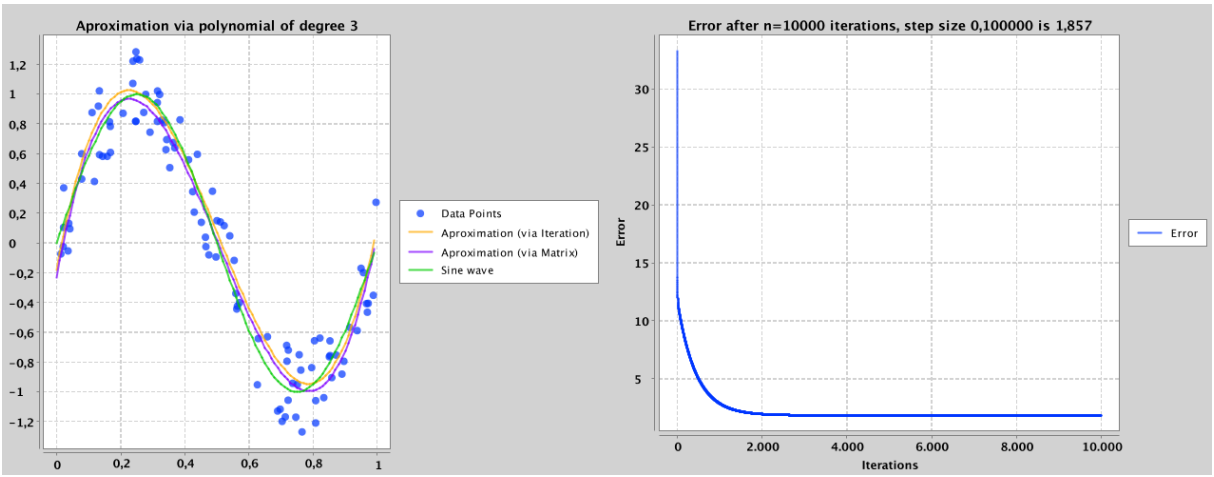
Universität Hamburg — 22. Oktober 2019

Hypothesis

The goal is to find a polynomial to aproximate our data points. We know that the source of our data points is one period of the sine function. One sine period has one maximum and one minumum. So we can presume that a polynomial of degree 3 is sufficent to fit our points.

$$y = f(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

Result



The polynomial

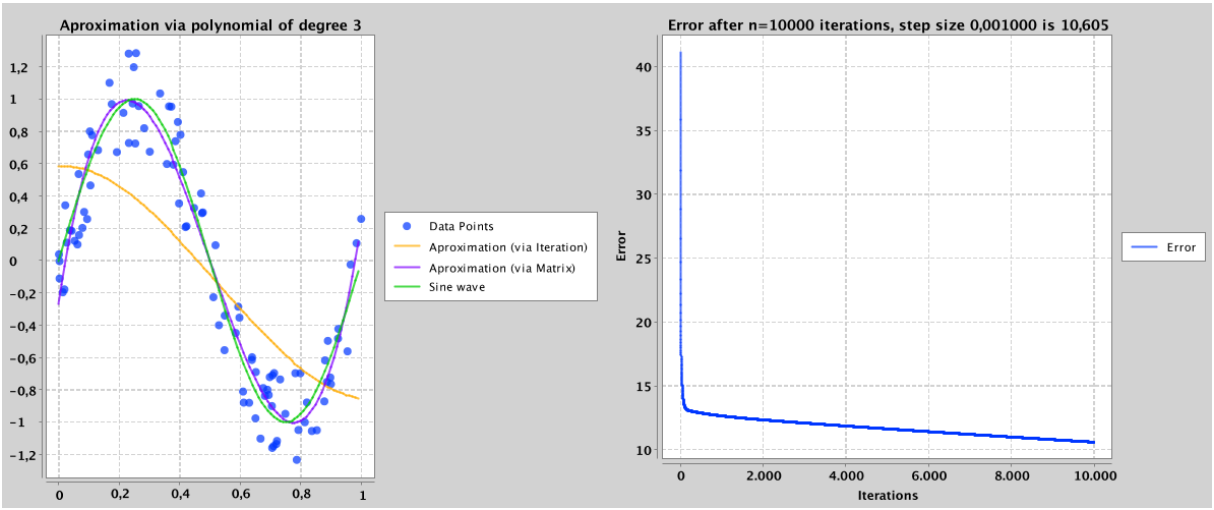
$$f(x) = \begin{bmatrix} -0.18389300723650462 \\ 11.861463633653237 \\ -33.86021075015673 \\ 22.307252966654328 \end{bmatrix}^T \begin{bmatrix} x^0 \\ x^1 \\ x^2 \\ x^3 \end{bmatrix}$$

has been computed using stochastic gradient descent algorithm using a learning rate of $\alpha = 0.2$ yielding a final error of 1.857

Optimizing the learnign rate

Using the gradient gradient descent algorithm with learning rate $\alpha = 0.001$ and wating for a maximum of 10.000 iterations we get

$$\theta = \left[0.5836921783142958, 0.13027164529834673, -4.315198280227572, 2.746957039499795\right]$$

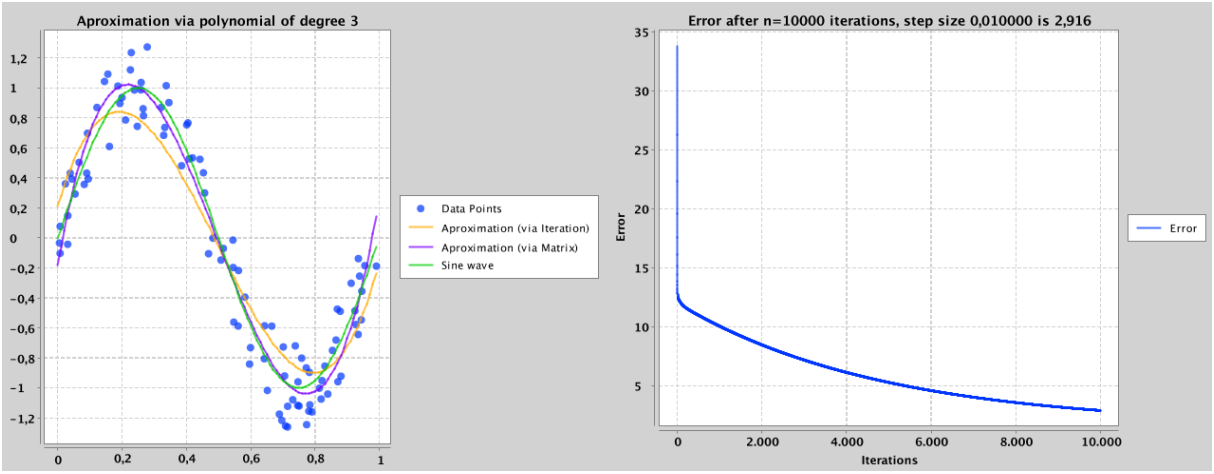


But as we can see the resulting curve does not really fit the points and the final error is above > 10. Plotting the error we see that indeed it decreases with each iteration and would decrease further if we allow the algorithm to iterate further. But the slope of the error plot is already getting pretty low and we do not want to wait much longer for the error to decrease.

Instead of increasing the maximum iterations we try to increase the learning rate α to let the algorithm progress faster.

Using the gradient gradient descent algorithm with learning rate $\alpha = 0.01$ and wating for a maximum of 10.000 iterations we get

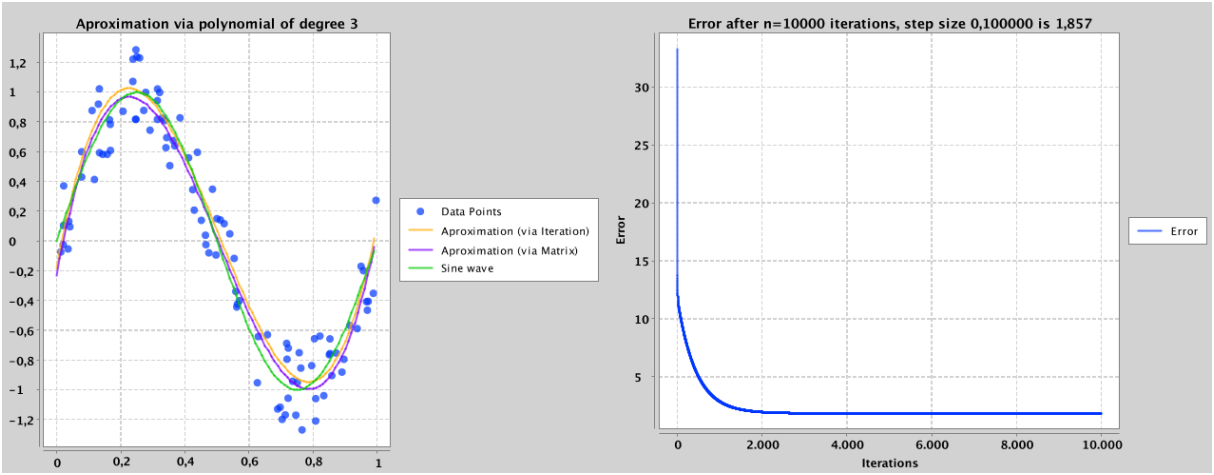
$$\theta = \left[0.2124157957316986, 7.179268724738586, -23.335012658458066, 15.78218638956732\right]$$



Plotting the polynom we can see it fits the points much better. Plotting the error we can see that it's dropping much faster below 10 and just below 3 after the same number of iterations as before. Ee can also see that increasing the number of iterations would further decrease the error. But instead we further increase the learning rate.

Using the gradient gradient descent algorithm with learning rate $\alpha = 0.0$ and wating for a maximum of 10.000 iterations we get

$$\theta = [-0.18389300723650462, 11.861463633653237, -33.86021075015673, 22.307252966654328]$$



Now the function fits the points pretty well and plotting the error we can see that it already converges after 4000 iterations at about 1.8.

Increasing the learning rate to much also increases the final error again. We could now do for example a binary search for the optimal learning rate and then increase the iterations until we are satisfied with a low low enough final error.

But just looking at the plot of the function matching the points we can already settle for the polynomial

$$f(x) = \begin{bmatrix} -0.18389300723650462 \\ 11.861463633653237 \\ -33.86021075015673 \\ 22.307252966654328 \end{bmatrix}^T \begin{bmatrix} x^0 \\ x^1 \\ x^2 \\ x^3 \end{bmatrix}$$

and an error of 1.857

Implementation

```
(defn build-polynomial [pairs degree]
  "convert [x y] pairs into {:x :y} map where"
  ":x becomes a vector of powers of x"
  (->> pairs
       (map
        (fn [[x y]] {:x (powers x degree)
```

```
                    :y y}))
       (vec)))

(defn gradient-descent [data alpha theta]
  "gradient descent algorithm"
  "alpha is the the learning rate"
  "theta is the initial vector of coefficients"
  (->> data
       (reduce
        (fn [t d]
          (m/add-scaled t
                        (:x d)
                        (* alpha
                           (- (:y d)
                              (m/dot t (:x d))))))
        theta)))

(defn find-polynomial [points degree learning-rate]
  "given a list of points find a polynomial of degree"
  "using gradient descent algorithm with learning-rate"
  (let [data (build-polynomial points degree)]
    (->> #(noise 0.5)
         ; initialize coefficients
         (repeatedly)
         (take (inc degree))
         (vec)
         ; apply gradient descent
         (iterate #(gradient-descent data learning-rate %1)))
```