handin1_done

September 19, 2024

0.0.1 Handin 1

1 Info

Everything should be completed and approved in person. Groups are fine.

The objectives for this handin is: * Playing with numpy. * Plotting with Plotly (as a preparation for using dash (https://plotly.com/dash/) later on) * Train a more complex model using the Fortuna Algorithm. * Route planning for Knut Knut Transport AS.

1.1 How to solve?

There should be a comment/section stating the objective of a task.
And there should be a commented section labeled # – CODE –
that shows where to put your code. For tasks that are not answered by code you can either write the answer
as a markup cell or write it on a seperate piece of paper.

[4]:
```
import numpy as np


# TASK: Create a numpy array containing the whole numbers between 1, 10␣
↪(inclusive)

# -- CODE --
a = np.arange(1,11,1,dtype=int)

assert isinstance(a, np.ndarray)
assert np.isclose(a, [1,2,3,4,5,6,7,8,9,10]).all()
print("<ok>")
```

```
<ok>
```

[5]:
```
# TASK: Reshape the array a so that it is a 2 by 4 array

a = np.array(range(0, 8))

# -- CODE --
a = a.reshape((2,4))
```

```
assert( a.shape == (2,4) )
print("<ok>")
```

```
<ok>
```

[9]:
```
# TASK: multiply all the numbers in a by 2.


a = np.array([[1,2,3,4], [5,6,7,8]])

# -- CODE --
a = a * 2

assert a.shape == (2,4)
assert a.sum() == 72
print("<ok>")
```

```
<ok>
```

[13]:
```
# TASK:  create a numpy array b that contains the sum of each row (axis 1) in a

a = np.array([[1,2,3,4], [5,6,7,8]])

# -- CODE --
b = np.sum(a,axis=1)

assert b.shape == (2,)
assert (b == [10, 26]).all()
print("<ok>")
```

```
<ok>
```

[15]:
```
# TASK:  create a numpy array b that contains the mean value of each column␣
↪(axis 0) in a
a = np.array([[1,2,3,4], [1,2,3,8]])

# -- CODE --
b = np.mean(a,axis=0)

assert b.shape == (4,)
assert (b == [1, 2, 3, 6]).all()
print("<ok>")
```

```
<ok>
```

[16]:
```
# TASK:  multiply each number that is equal to 2 by 10
a = np.array([[1,2,3,4], [1,2,3,8]])

# -- CODE --
```

```
b = np.where(a==2,10*a,a)

a_after_mul = np.array([[1,20,3,4], [1,20,3,8]])
assert (a_after_mul == b).all()
print("<ok>")
```

```
<ok>
```

[18]:
```
# TASK:  stack the arrays a, b and c into an array called s (vertically)
# s should be equal to s_true (hint, look at the numpy function vstack )


a = np.array([1, 3])
b = np.array([2, 4])
c = np.array([3, 5])

# -- CODE --
s = np.vstack((a,b,c))

s_true = np.array([[1, 3], [2, 4], [3, 5]])
assert (s_true == s).all()

print("<ok>")
```

```
<ok>
```

[20]:
```
# TASK:  Flatten the array a into a 1d array called f (hint: numpy has a␣
 ↪function named flatten)

a = np.random.randn(2,2,2)
print(a)

# -- CODE --
f = np.ndarray.flatten(a)

print("-"*50)
print(f)
assert f.ndim == 1
print("<ok>")
```

```
[[[-0.05064257  0.54998121]
  [ 0.37918823  0.36720715]]

 [[ 0.53721276 -0.33831552]
  [-0.56115702 -1.77865016]]]
--------------------------------------------------
[-0.05064257  0.54998121  0.37918823  0.36720715  0.53721276 -0.33831552
 -0.56115702 -1.77865016]
<ok>
```

1.2 Task 1 – Plot functions and scatter plots using Plotly

Change the code to perform the two following tasks:

(See Plotly: https://plotly.com/python/plotly-express/ for more info.)

Line Plot)

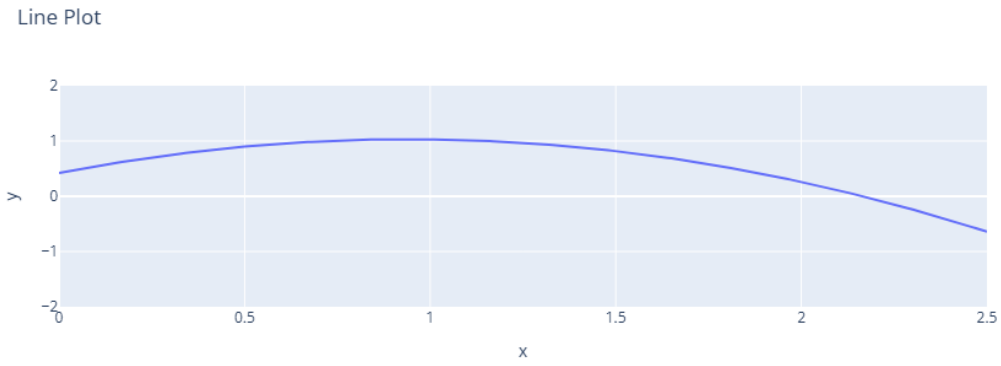
Plot the function f(x) = -0.69x^2 + 1.3x + 0.42 over the interval [0, 2.5], with 0.01 increments in x.

Scatter Plot)

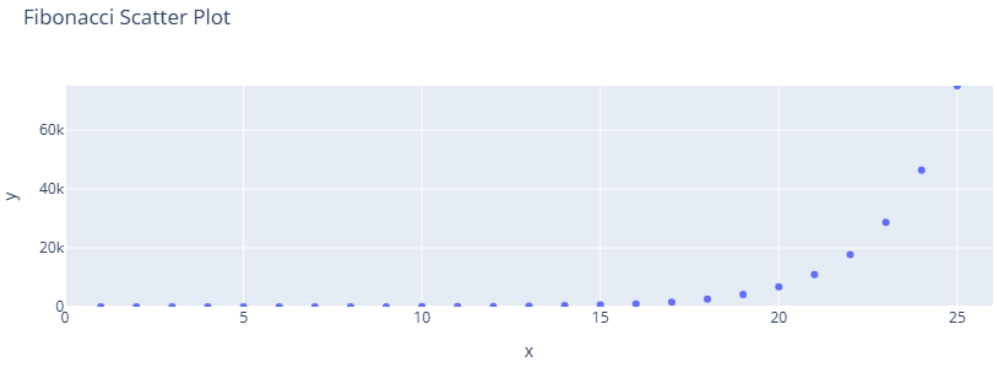
Plot the first 25 Fibbonachi numbers using a scatter plot. (The example shows the first 5).

[ ]:
```
# installs plotly
!pip install plotly
!pip install "jupyterlab>=3" "ipywidgets>=7.6"

print("You may have to restart jupyter to see graphs in the notebook.")
```

[4]:
```
import numpy as np
import plotly.express as px


# Change this code into the correct Line Plot
xs = np.arange(0,2.51,0.01)
#xs = np.array([1, 1.5, 2, 2.5, 6.0])
ys = -0.69*xs*xs+1.3*xs+0.42
#ys = np.array([1, 2, 3, 3.1, 2.5])


fig = px.line(x=xs, y=ys, title="Line Plot")
fig.update_layout(xaxis_range=[0,2.5], yaxis_range=[-2,2])
fig.show()
```



[10]:
```
# Change this code into the correct Scatter Plot (please take care of the axis)
phi = (1+np.sqrt(5))/2
neg_1_over_phi = (1-np.sqrt(5))/2

xs = np.arange(1,26)
ys = (phi**xs - neg_1_over_phi**xs)/np.sqrt(5)


fig = px.scatter(x=xs, y=ys, title="Fibonacci Scatter Plot")
fig.update_layout(xaxis_range=[0,26], yaxis_range=[0,ys[24]])
fig.show()
```



2 Task 2 – Numpy and curve fitting

Fill in the function get_2polynomal_from_3_points(). You are required to use the function: np.linalg.solve() for the actual computation, i.e. the task is mainly about shaping the input so it fits the np.linalg.solve() function. Note: dont use polyfit().

The function should return a 1d numpy array, with shape = (3,) in the format a, b,c. Such that axˆ2 + bx + c = y, goes through the points p0, p1, p2

HINT: How should a matrix equation be formed, so solving it gives us the coefficients for a polynomial?

Limitations: * There should be NO loops or ifs! (for/while/if). * Numpy functions that might come handy: reshape, hstack, flatten, ones

[40]:
```
import numpy as np

def get_2polynomal_from_3_points(p0: np.array, p1: np.array, p2: np.array) -> np.
 ↪array:
    #a*x0^2 + b*x0 + c = y0
    #a*x1^2 + b*x1 + c = y1
```

```
    #a*x2^2 + b*x2 + c = y2
    ps = np.hstack([p0, p1, p2]).reshape((2,3),order='F')
    xs = ps[0].reshape((3,-1))
    ys = ps[1].reshape((3,-1))
    a = np.hstack((xs*xs,xs,np.ones(3).reshape((3,-1))))
    b = ys.flatten()
    return np.linalg.solve(a,b)



point0 = np.array([0.147, 0.596])
point1 = np.array([0.7, 0.992])
point2 = np.array([2.06, 0.17])

polynomial_coefficients = get_2polynomal_from_3_points(point0, point1, point2)
assert(polynomial_coefficients.shape[0] == 3 and polynomial_coefficients.ndim ==␣
 ↪1)

print("Polynomial: {:2f}x^2 + {:.2f}x + {:.2f}".format(*polynomial_coefficients))


if not np.isclose(polynomial_coefficients[0], -0.69, atol=0.01):
    print("x^2 coeff is wrong (a), should be close to -0.69")

if not np.isclose(polynomial_coefficients[1], 1.3, atol=0.01):
    print("x coeff is wrong (b), should be close to 1.3")

if not np.isclose(polynomial_coefficients[2], 0.42, atol=0.01):
    print("Constant factor (c) is wrong, should be close to 0.42")

if np.isclose(polynomial_coefficients, [-0.69, 1.3, 0.42], atol=0.01).all():
    print("Correct polynomial found.")
```

```
Polynomial: -0.690280x^2 + 1.30x + 0.42
Correct polynomial found.
```

3 Task 3 – Simple random search

Find the triplet $a, b, c \in \{x \mid x \in \mathbb{Z} \text{ and } 450 > x > 0\}$

Using a random search in the parameter space. Such that the following relations is satisfied:

3.0.1 a

$$a = \begin{cases} c + 11, & \text{if } b \text{ is even} \\ 2c - 129, & \text{if } b \text{ is odd} \end{cases}$$

3.0.2 b

$b = (a \times c) \mod 2377$

3.0.3 c

$c = \left(\sum_{k=0}^{a-1} b - 7k\right) + 142$

Also how many guesses were needed?

Note that in math notation $\sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5$

[48]:
```
import random
#Solving Task a.
def loss(a,b,c):
    if(b%2==0):
        if(a==c+11):
            return True
        else:
            return False
    else:
        if(a==2*c-129):
            return True
        else:
            return False

def search():
    ct = 1
    a = random.randrange(0,451,1)
    b = random.randrange(0,451,1)
    c = random.randrange(0,451,1)
    while(loss(a,b,c)==False):
        ct += 1
        a = random.randrange(0,451,1)
        b = random.randrange(0,451,1)
        c = random.randrange(0,451,1)
    print("a = {}, b = {}, c = {}. Number of attempts to satisfy: {}.".
 ↪format(a,b,c,ct))
    return

search()
```

```
a = 365, b = 436, c = 354. Number of attempts to satisfy: 776.
```

[47]:
```
import random
#Solving Task b.
def loss(a,b,c):
    return b == (a*c)%2377
```

```
def search():
    ct = 1
    a = random.randrange(0,451,1)
    b = random.randrange(0,451,1)
    c = random.randrange(0,451,1)
    while(loss(a,b,c)==False):
        ct += 1
        a = random.randrange(0,451,1)
        b = random.randrange(0,451,1)
        c = random.randrange(0,451,1)
    print("a = {}, b = {}, c = {}. Number of attempts to satisfy: {}.".
 ↪format(a,b,c,ct))
    return

search()
```

```
a = 302, b = 78, c = 16. Number of attempts to satisfy: 2115.
```

[49]:
```
import random
#Solving Task c.
def loss(a,b,c):
    return c == a*b - 7 * (a-1) * a /2 + 142

def search():
    ct = 1
    a = random.randrange(0,451,1)
    b = random.randrange(0,451,1)
    c = random.randrange(0,451,1)
    while(loss(a,b,c)==False):
        ct += 1
        a = random.randrange(0,451,1)
        b = random.randrange(0,451,1)
        c = random.randrange(0,451,1)
    print("a = {}, b = {}, c = {}. Number of attempts to satisfy: {}.".
 ↪format(a,b,c,ct))
    return

search()
```

```
a = 56, b = 198, c = 450. Number of attempts to satisfy: 15178.
```

4 Task 4 – Revisit the Fortuna Algorithm

Below is a implementation of Fortuna that uses fits linear function $f_\theta = ax + b$ where $\theta = \{a, b\}$ to a function $g(x)$.
However, as is evidently from the graph, $g(x)$ is not a linear function.

1) Change the code to instead use $f_{\vartheta}(x) = \sum_{k=1}^{3}\Psi_k \sin(\gamma_k(x+\omega_k))$. *Such that* $|\theta| = 9$, where each parameter c in θ is in $[-4.4]$. That is, $f_\theta(x)$ is a sum of three sin terms. **Do not change the range of the sample_theta function.**

2) However, it seems Fortuna (on average) struggles to find the optimal parameters θ.
Therefore you will have to innovate and change the Fortuna algorithm so that it faster finds "better solutions". What changes did you make and **why** did you make them, and how did you measure how efficient these changes were?
A excellent solution here will have an expected best loss of less than 5 using 100000 guesses. (take the average over 100 runs). **But ANY improvment is sufficient to pass!**

3) Using your newly made modified Fortuna Algorithm optimize the function: $h(x) = \mu - (\zeta sin(\kappa x))(\tau(x+\lambda))$.
The y values for this function can be found in the numpy array ys_h (in the code below).
Does your new and improved Fortuna outperform the regular fortuna on this function as well? Why?
Remember to change your model to match $h(x)$

4) [**Optional**] Develop a multiprocces implemention of the Fortuna algorithm using python's multiprocessing library (https://docs.python.org/3/library/multiprocessing.html).
How are the speed ups? Are Fortuna really suited to parallel execution?

[ ]:
```
!pip install tqdm
```

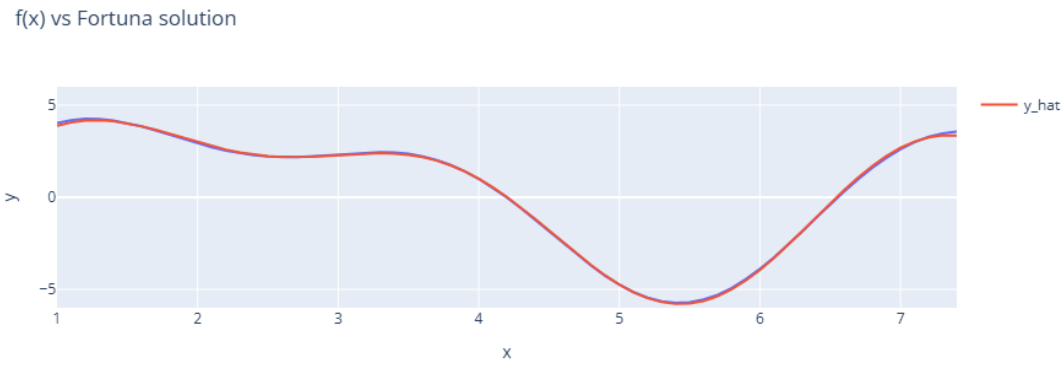
[87]:
```
%%time
# the magic %%time has to be the first line in the cell, and reports the total␣
 ↪exec time of the cell.

import random
import numpy as np
import plotly.express as px
import tqdm


def predict(x, theta):
    # change to sum of 3 sin() terms.
    # use np.sin() and not math.sin().
    return [np.sum([theta[3*i]*np.sin(theta[3*i+1]*(x_val+theta[3*i+2])) for i␣
 ↪in range(3)]) for x_val in x] # this is the model


def sample_theta(size_of_theta):
    # Do NOT CHANGE.
    theta = np.random.uniform(-4, 4, size=size_of_theta)
    return theta


def get_loss(y_hat, ys):
```

```
    # No change needed, returns quadratic loss.
    loss = ((y_hat - ys)**2).sum()
    return loss

xs = np.array([1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.1, 2.2,␣
 ↪2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.
 ↪8, 3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5.0, 5.1, 5.2, 5.3,␣
 ↪5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.
 ↪9, 7.0, 7.1, 7.2, 7.3, 7.4])
ys = np.array([4.03, 4.19, 4.26, 4.25, 4.17, 4.03, 3.85, 3.63, 3.40, 3.16, 2.93,␣
 ↪2.72, 2.53, 2.39, 2.28, 2.21, 2.18, 2.19, 2.22, 2.27, 2.33, 2.39, 2.44, 2.45,␣
 ↪2.43, 2.36, 2.22, 2.02, 1.75, 1.41, 1.00, 0.52, -0.01, -0.60, -1.22, -1.86, -2.
 ↪50, -3.13, -3.72, -4.27, -4.75, -5.15, -5.45, -5.65, -5.74, -5.70, -5.55, -5.
 ↪29, -4.92, -4.44, -3.89, -3.26, -2.58, -1.86, -1.12, -0.39, 0.32, 0.98, 1.60,␣
 ↪2.14, 2.61, 2.99, 3.28, 3.47, 3.57])
ys_h = np.array([15.98, 21.42, 24.1, 23.87, 21.0, 16.11, 10.06, 3.79, -1.8, -6.
 ↪01, -8.39, -8.82, -7.47, -4.77, -1.3, 2.31, 5.49, 7.81, 9.04, 9.18, 8.44, 7.
 ↪15, 5.72, 4.54, 3.89, 3.9, 4.52, 5.54, 6.63, 7.39, 7.49, 6.7, 4.97, 2.44, -0.
 ↪54, -3.48, -5.8, -6.98, -6.61, -4.51, -0.79, 4.2, 9.8, 15.21, 19.57, 22.09, 22.
 ↪2, 19.63, 14.52, 7.41, -0.83, -9.07, -16.11, -20.83, -22.36, -20.27, -14.59,␣
 ↪-5.91, 4.72, 15.9, 26.06, 33.69, 37.58, 36.94, 31.64])


# change to the size of theta ( 9 ) (for h(x) how many parameters does it have?)
n_params = 9

best_loss = float('inf')
best_theta = sample_theta(n_params)

for _ in tqdm.tqdm(range(100000)):
    curr_theta = sample_theta(n_params)
    thetas = curr_theta
    #Only challenge one parameter at a time in our hyperparameter
    k = random.randrange(n_params)
    curr_theta = [best_theta[j] if j!=k else thetas[j] for j in range(n_params)]
    y_hat = predict(xs, curr_theta)
    curr_loss = get_loss(y_hat, ys)

    if best_loss > curr_loss:
        best_loss = curr_loss
        best_theta = curr_theta


print("best loss:", best_loss)
print("theta:", best_theta)
```

```
fig = px.line(x=xs, y=ys, title="f(x) vs Fortuna solution")
fig.add_scatter(x=xs, y=predict(xs, best_theta), mode='lines', name="y_hat")
fig.update_layout(xaxis_range=[xs.min(),xs.max()], yaxis_range=[-6,6])
fig.show()


# to get a solid estimate -> you should train at least 100 models and take the␣
 ↪average performance.
```

```
100%|
| 100000/100000
[02:01<00:00, 824.01it/s]
```

```
best loss: 0.2531531701548082
theta: [-2.191817449149112, -1.9724058775206057, -0.004069416941780091,
-3.0431274224054343, -1.1731057663273372, -0.9181489887411542,
1.6897554413738822, 0.6114163805749877, 0.9392309968137722]
```




```
CPU times: total: 1min 25s
Wall time: 2min 1s
```

Solution: TL;DR: Change the Fortuna algorithm so that it only challenges one value in theta at a time.

Problem: The current Forrtuna algorithm updates the hyperparameter in a make-it-or-break-it fashion. Consider the loss function (a 9D 'surface' in a 10-dimensional space). We want to find the (global) or, mo reasonably, a good local minimum. The Fortuna algorithm 'teleports' each time at random to a point in this 9-dimensional parameter space, and the chances of getting a good theta decrease exponentially with the dimension of the hyperparameter.

Improvement Pick one value of the newly sampled ones at random and only challenge the corresponding one in the best_theta. Like this, instead of randomly teleporting to a point, 'losing' all of the previous state/training, we take a step along one of the directions. This way, **if the loss function smooth enough**, is the 'guess' will be gradually improved in one dimension each iteration, until the result should be better for this approach considering a high enough number of iterations.

$\frac{\delta L}{\delta \Psi_k} = 2\sum_{i=1}^{n}(f_\theta(x_i) - y_i)sin(\gamma_k(x + \omega_k))$

$\frac{\delta L}{\delta \gamma_k} = 2\sum_{i=1}^{n}(f_\theta(x_i) - y_i)\Psi_k(x + \omega_k)cos(\gamma_k(x + \omega_k))$

$\frac{\delta L}{\delta \omega_k} = 2\sum_{i=1}^{n}(f_\theta(x_i) - y_i)\Psi_k\gamma_k cos(\gamma_k(x + \omega_k))$

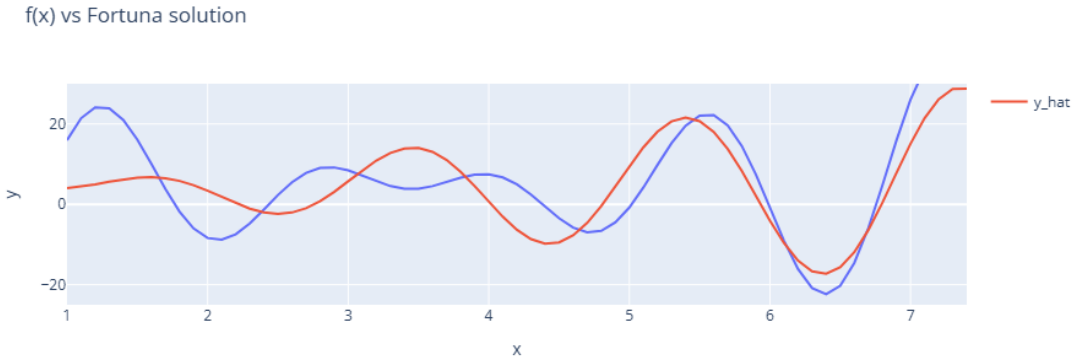
[86]:
```
%%time
# the magic %%time has to be the first line in the cell, and reports the total␣
 ↪exec time of the cell.

import random
import numpy as np
import plotly.express as px
import tqdm


def predict(x, theta):
    # change to sum of 3 sin() terms.
    # use np.sin() and not math.sin().
    mu = theta[0]
    zetta = theta[1]
    kappa = theta[2]
    tau = theta[3]
    lamda = theta[4]
    return [mu-zetta*np.sin(kappa*x_val)*(tau*(x_val+lamda)) for x_val in x] #␣
 ↪this is the model


def sample_theta(size_of_theta):
    # Do NOT CHANGE.
    theta = np.random.uniform(-4, 4, size=size_of_theta)
    return theta


def get_loss(y_hat, ys):
    # No change needed, returns quadratic loss.
    loss = ((y_hat - ys)**2).sum()
    return loss
```

```
xs = np.array([1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.1, 2.2,␣
 ↪2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.
 ↪8, 3.9, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 5.0, 5.1, 5.2, 5.3,␣
 ↪5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.
 ↪9, 7.0, 7.1, 7.2, 7.3, 7.4])
ys = np.array([4.03, 4.19, 4.26, 4.25, 4.17, 4.03, 3.85, 3.63, 3.40, 3.16, 2.93,␣
 ↪2.72, 2.53, 2.39, 2.28, 2.21, 2.18, 2.19, 2.22, 2.27, 2.33, 2.39, 2.44, 2.45,␣
 ↪2.43, 2.36, 2.22, 2.02, 1.75, 1.41, 1.00, 0.52, -0.01, -0.60, -1.22, -1.86, -2.
 ↪50, -3.13, -3.72, -4.27, -4.75, -5.15, -5.45, -5.65, -5.74, -5.70, -5.55, -5.
 ↪29, -4.92, -4.44, -3.89, -3.26, -2.58, -1.86, -1.12, -0.39, 0.32, 0.98, 1.60,␣
 ↪2.14, 2.61, 2.99, 3.28, 3.47, 3.57])
ys_h = np.array([15.98, 21.42, 24.1, 23.87, 21.0, 16.11, 10.06, 3.79, -1.8, -6.
 ↪01, -8.39, -8.82, -7.47, -4.77, -1.3, 2.31, 5.49, 7.81, 9.04, 9.18, 8.44, 7.
 ↪15, 5.72, 4.54, 3.89, 3.9, 4.52, 5.54, 6.63, 7.39, 7.49, 6.7, 4.97, 2.44, -0.
 ↪54, -3.48, -5.8, -6.98, -6.61, -4.51, -0.79, 4.2, 9.8, 15.21, 19.57, 22.09, 22.
 ↪2, 19.63, 14.52, 7.41, -0.83, -9.07, -16.11, -20.83, -22.36, -20.27, -14.59,␣
 ↪-5.91, 4.72, 15.9, 26.06, 33.69, 37.58, 36.94, 31.64])


# change to the size of theta ( 9 ) (for h(x) how many parameters does it have?)
n_params = 5

best_loss = float('inf')
best_theta = sample_theta(n_params)

for _ in tqdm.tqdm(range(100000)):
    curr_theta = sample_theta(n_params)
    thetas = curr_theta
    k = random.randrange(n_params)
    curr_theta = [best_theta[j] if j!=k else thetas[j] for j in range(n_params)]
    y_hat = predict(xs, curr_theta)
    curr_loss = get_loss(y_hat, ys_h)

    if best_loss > curr_loss:
        best_loss = curr_loss
        best_theta = curr_theta


print("best loss:", best_loss)
print("theta:", best_theta)


fig = px.line(x=xs, y=ys_h, title="f(x) vs Fortuna solution")
fig.add_scatter(x=xs, y=predict(xs, best_theta), mode='lines', name="y_hat")
fig.update_layout(xaxis_range=[xs.min(),xs.max()], yaxis_range=[-25,30])
fig.show()
```

```
# to get a solid estimate -> you should train at least 100 models and take the␣
↪average performance.
```

```
100%|
| 100000/100000
[00:16<00:00, 6234.51it/s]
```

```
best loss: 4354.405469143712
theta: [3.9997499698275467, -0.9600502742179247, 3.2108504980277806,
-3.9997581609147606, -0.8131565626332984]
```




```
CPU times: total: 11.5 s
Wall time: 16.1 s
```

Answer: Function h(x) cannot be improved by only performing descent along one direction at a time That is because the derivative of the loss function for h(x) does not behave in a way to allow for small learning rates.

$$\frac{\delta L}{\delta \mu} = 2\sum_{i=1}^{n}(h(x_i) - y_i)$$

$$\frac{\delta L}{\delta \zeta} = -2\sum_{i=1}^{n}(h(x_i) - y_i)(sin(\kappa x))(\tau(x + \lambda))$$

$$\frac{\delta L}{\delta \kappa} = 2\sum_{i=1}^{n}(h(x_i) - y_i)\zeta x cos(\kappa x)(\tau(x + \lambda))$$

$$\frac{\delta L}{\delta \tau} = -2\sum_{i=1}^{n}(h(x_i) - y_i)\zeta(sin(\kappa x))(x + \lambda)$$

$$\frac{\delta L}{\delta \lambda} = -2\sum_{i=1}^{n}(h(x_i) - y_i)\zeta\lambda(sin(\kappa x))$$

5 Task 5 – Fortuna For Decision Trees

Implement a Decision Tree for solving the XOR problem.
Here: there are 2 real valued inputs, x0, x1 (found in xs).

And the DT should take these two as input and predict an output: 0 or 1.
The true answer can be found in ys_true.

[108]:
```
import numpy as np
rng = np.random.default_rng(42)
n_examples = 40

xs = rng.uniform(size=(n_examples, 2))

# make a true y
b = (xs>0.5).astype(int)
ys_true = np.logical_xor(b[:, 0], b[:, 1]).astype(int)


##################################
# here you implement a Decision Tree that is built using the fortuna algorithm.
# The depth of the tree should be less than 4
# The DT should reach 100% accuracy.
def predict(xs, thresh):
    ys = []
    for x in xs:
        if x[0]<thresh[0]:
            if x[1]<thresh[1]:
                if x[0]>thresh[2]:
                    if x[1]>thresh[3]:
                        ys.append(0)
                    else:
                        ys.append(1)
                else:
                    if x[1]>thresh[3]:
                        ys.append(1)
                    else:
                        ys.append(0)
            else:
                if x[0]>thresh[2]:
                    if x[1]>thresh[3]:
                        ys.append(1)
                    else:
                        ys.append(0)
                else:
                    if x[1]>thresh[3]:
                        ys.append(0)
                    else:
```

```
                        ys.append(1)
        else:
            if x[1]<thresh[1]:
                if x[0]>thresh[2]:
                    if x[1]>thresh[3]:
                        ys.append(1)
                    else:
                        ys.append(0)
                else:
                    if x[1]>thresh[3]:
                        ys.append(0)
                    else:
                        ys.append(1)
            else:
                if x[0]>thresh[2]:
                    if x[1]>thresh[3]:
                        ys.append(0)
                    else:
                        ys.append(1)
                else:
                    if x[1]>thresh[3]:
                        ys.append(1)
                    else:
                        ys.append(0)
    return ys
    #return [1 if x[1]>thresh[1] else 0 if x[0]>thresh[2] else 0 if␣
↪x[1]>thresh[0] else 1 for x in xs]

def sample_theta(size_of_theta):
    # Do NOT CHANGE.
    theta = np.random.uniform(0, 1, size=size_of_theta)
    return theta

def get_loss(y_hat, ys):
    # No change needed, returns quadratic loss.
    loss = ((y_hat - ys)**2).sum()
    return loss

# change to the size of theta ( 9 ) (for h(x) how many parameters does it have?)
n_params = 4

best_loss = float('inf')
best_theta = sample_theta(n_params)

for _ in tqdm.tqdm(range(100000)):
    curr_theta = sample_theta(n_params)
    thetas = curr_theta
```

```
    #Only challenge one parameter at a time in our hyperparameter
    #k = random.randrange(n_params)
    #curr_theta = [best_theta[j] if j!=k else thetas[j] for j in range(n_params)]
    y_hat = predict(xs, curr_theta)
    curr_loss = get_loss(y_hat, ys_true)

    if best_loss > curr_loss:
        best_loss = curr_loss
        best_theta = curr_theta


print("best loss:", best_loss)
print("theta:", best_theta)
```

```
100%|
| 100000/100000
[00:06<00:00, 16290.36it/s]

best loss: 0
theta: [2.06579904e-02 5.08579131e-01 5.33865200e-01 1.75125891e-04]
```

6 Task 6 – Best Route App

A company called Knut Knut Transport AS is using one of the 4 routes for delivery (see Pitch_knutknut_transport.pdf for more info). * A -> C -> D
* A -> C -> E
* B -> C -> D
* B -> C -> E

They have discovered that they can transport the goods faster by picking the right route given only the depature time of the transport.

In the file "traffic.jsonl" on Canvas is the collected data up to the current point in time. So far, they have just selected a route at random, now they want to implement a simple web-api that can help the drivers select the best route.

- Using the data found in traffic.jsonl, create a ML model that given an time (hour:min) can select the fastest route to travel. **Be sure to document in a google doc/word/notebook/... how YOU created your ML model.** The ML model should be trained using the Fortuna algorithm.
- Download the knut_knut_app.py found on canvas an implement the function: get_the_best_route_as_a_text_informatic(dep_hour, dep_min) such that the web application works and gives a good estimated time.
- The CEO (Knut) want you to estimate how much time&money they can save from using this new app.

- Prepare a VERY short presentation (pdf) on the gains and tech behind the app (random subsample will present for the class)

Some hints: * Data Exploration: Look closely at the data using both numerical analysis and visual (plotting) * Use the python library datetime to handle time calculations. * Make sure the features are scaled in a way that make sense. * Once you have a model working, use the python library pickle to save/load the model.