IDNM 680: HOMEWORK 3

JAMES DELLA-GIUSTINA

Task 1

You have studied various methods to deal with data: interpolation, linear regression, and curve fitting. Try to make a "smart" program to deal with a randomly generated dataset, so that it can automatically determine the best way to make a prediction with a specific data point. That is, if the data shows trend close to linear, then the program will use linear regression to make the prediction. Otherwise, it will use curve fitting or interpolation to make prediction. This would be considered a simple deep learning experiment.

Solution 1. This program uses polynomials p_d of degree $d \in [1, 5]$ to accurately describe a linear relationship between the x and y values of the data. We can employ some statistical analysis to determine which polynomial p_d most accurately fits the data by examining the R^2 score which can be calculated in the following way:

- n is the number of observed data points.
- $\bar{y}$ is the mean of the observed data.
- y_i is the i^{th} observed data point.
- $\hat{y}_i$ is the predicted value of the i^{th} data point by p_d.

Then, we can define the total sum of squares (TSS), the regression sum of squares (RSS), and the residual sum of squares (ESS) as follows:

$$TSS = \sum_{i=1}^{n} (y_i - \bar{y})^2 \quad \text{and} \quad RSS = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$$

Then:

$$R^2 = \frac{RSS}{TSS}$$

Due to the technical difficulty in generating truly random data that can be fit by any one of these 5 polynomials p_d, I have simply slightly spaced polynomials to generate data and fitted the appropriate model. Note that in my first example I had designed an if statement that would choose the polynomial based on the maximum R^2 score and then find the 'smallest max' for computationally efficiency. However, because of the data generation problem, I did not include this functionality for anything besides the linear case.

```
import numpy as np

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import r2_score

# Generate some random data
x = np.linspace(1, 10, 100)
y = np.random.randn(100) * 11
order = np.random.randint(1, 3)

for i in range(1, order + 1):
  coefficient = np.random.randint(-10, 10) * x**(i*np.random.rand())
  y += coefficient
```

Date: Spring 2023.

```
# Fit a linear regression model
model_lin = LinearRegression().fit(x.reshape(-1, 1), y)
y_pred_lin = model_lin.predict(x.reshape(-1, 1))
r2_lin = r2_score(y, y_pred_lin)

# Fit k degree polynomials for k \in [2,5]
poly2 = PolynomialFeatures(degree=2)
X_poly2 = poly2.fit_transform(x.reshape(-1, 1))
model_poly2 = LinearRegression().fit(X_poly2, y)
y_pred_poly2 = model_poly2.predict(X_poly2)
r2_poly2 = r2_score(y, y_pred_poly2)

poly3 = PolynomialFeatures(degree=3)
X_poly3 = poly3.fit_transform(x.reshape(-1, 1))
model_poly3 = LinearRegression().fit(X_poly3, y)
y_pred_poly3 = model_poly3.predict(X_poly3)
r2_poly3 = r2_score(y, y_pred_poly3)

poly4 = PolynomialFeatures(degree=4)
X_poly4 = poly4.fit_transform(x.reshape(-1, 1))
model_poly4 = LinearRegression().fit(X_poly4, y)
y_pred_poly4 = model_poly4.predict(X_poly4)
r2_poly4 = r2_score(y, y_pred_poly4)

poly5 = PolynomialFeatures(degree=5)
X_poly5 = poly5.fit_transform(x.reshape(-1, 1))
model_poly5 = LinearRegression().fit(X_poly5, y)
y_pred_poly5 = model_poly5.predict(X_poly5)
r2_poly5 = r2_score(y, y_pred_poly5)

print("Linear model R2:", r2_lin)
print("Quadratic model R2:", r2_poly2)
print("Cubic model R2:", r2_poly3)
print("Quartic model R2:", r2_poly4)
print("Quintic model R2:", r2_poly5)

R_scores = [r2_lin, r2_poly2, r2_poly3, r2_poly4, r2_poly5]
first_choice = max(R_scores)
choice = None
for value in R_scores:
    if (first_choice - value) <= 0.08 and (choice is None or value < choice):
        choice = value

plt.scatter(x, y, label='Data')

if choice == r2_lin:
    plt.plot(x, y_pred_lin, label='Linear model',color='red')
elif choice == r2_poly2:
    plt.plot(x, y_pred_poly2, label='Quadratic model',color='orange')
elif choice == r2_poly3:
    plt.plot(x, y_pred_poly3, label='Cubic model',color='orange')
elif choice == r2_poly4:
    plt.plot(x, y_pred_poly4, label='Quartic model',color='orange')
else:
```

```
    plt.plot(x, y_pred_poly5, label='Quintic model',color='orange')

plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend()
plt.show()
```

This produces text output:

```
Linear model R2: 0.3407038825209183
Quadratic model R2: 0.36240593316236414
Cubic model R2: 0.38253058233409143
Quartic model R2: 0.38360193828474365
Quintic model R2: 0.38360196777160027
```

and Figure 1.

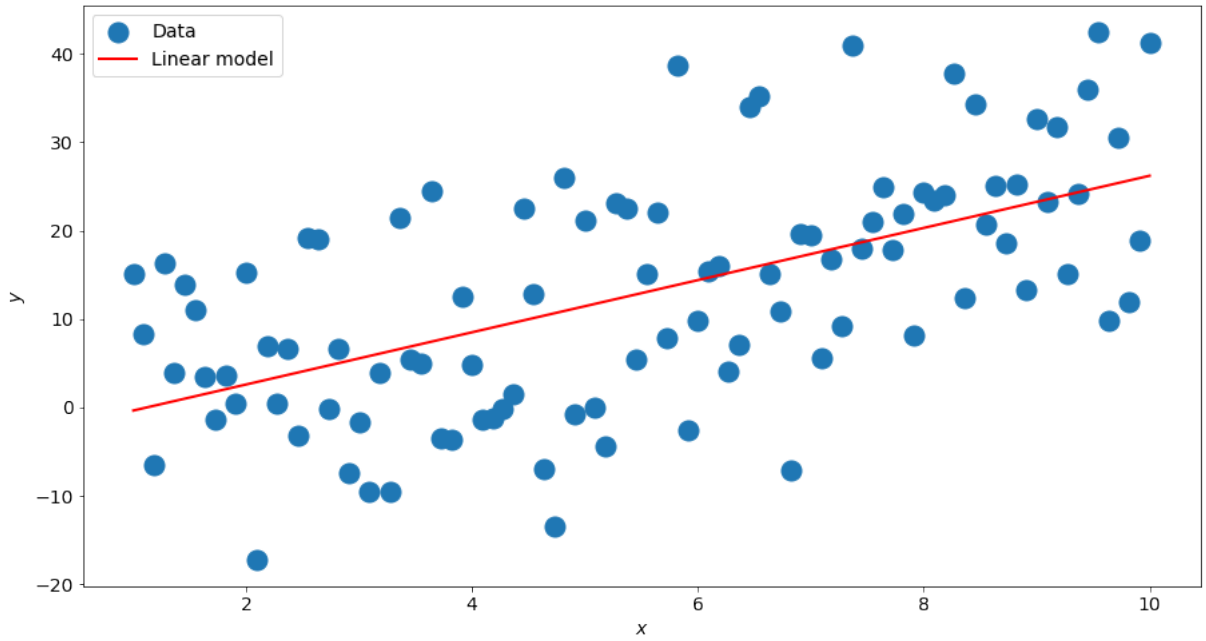

FIGURE 1

```
    # Quadratic
x = np.linspace(-10, 10, 100)
y = x**2 + np.random.normal(0, 5, 100)

# Fit a linear regression model
model_lin = LinearRegression().fit(x.reshape(-1, 1), y)
y_pred_lin = model_lin.predict(x.reshape(-1, 1))
r2_lin = r2_score(y, y_pred_lin)

# Fit k degree polynomials for k \in [2,5]
poly2 = PolynomialFeatures(degree=2)
X_poly2 = poly2.fit_transform(x.reshape(-1, 1))
model_poly2 = LinearRegression().fit(X_poly2, y)
y_pred_poly2 = model_poly2.predict(X_poly2)
r2_poly2 = r2_score(y, y_pred_poly2)

print("Linear model R2:", r2_lin)
print("Quadratic model R2:", r2_poly2)
```

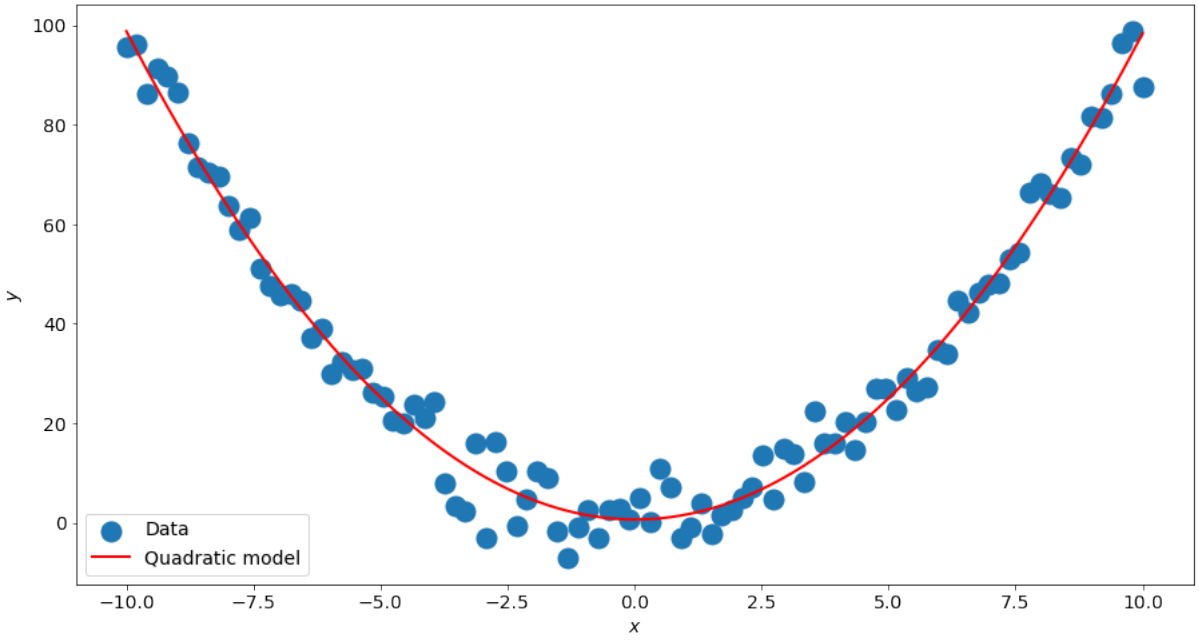

FIGURE 2

```
R_scores = [r2_lin, r2_poly2]
choice = max(R_scores)

plt.scatter(x, y, label='Data')

if choice == r2_lin:
    plt.plot(x, y_pred_lin, label='Linear model',color='red')
else:
    plt.plot(x, y_pred_poly2, label='Quadratic model',color='red')

plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend()
plt.show()
```

With text output:

```
Linear model R2: 1.8242323457928755e-05
Quadratic model R2: 0.9780947703007071
```

and Figure 2.

```
# Cubic
x = np.linspace(-10, 10, 100)
y = x**3 + np.random.normal(0, 50, 100)

# Fit a linear regression model
model_lin = LinearRegression().fit(x.reshape(-1, 1), y)
y_pred_lin = model_lin.predict(x.reshape(-1, 1))
r2_lin = r2_score(y, y_pred_lin)

# Fit k degree polynomials for k \in [2,5]
poly2 = PolynomialFeatures(degree=2)
X_poly2 = poly2.fit_transform(x.reshape(-1, 1))
model_poly2 = LinearRegression().fit(X_poly2, y)
y_pred_poly2 = model_poly2.predict(X_poly2)
r2_poly2 = r2_score(y, y_pred_poly2)
```

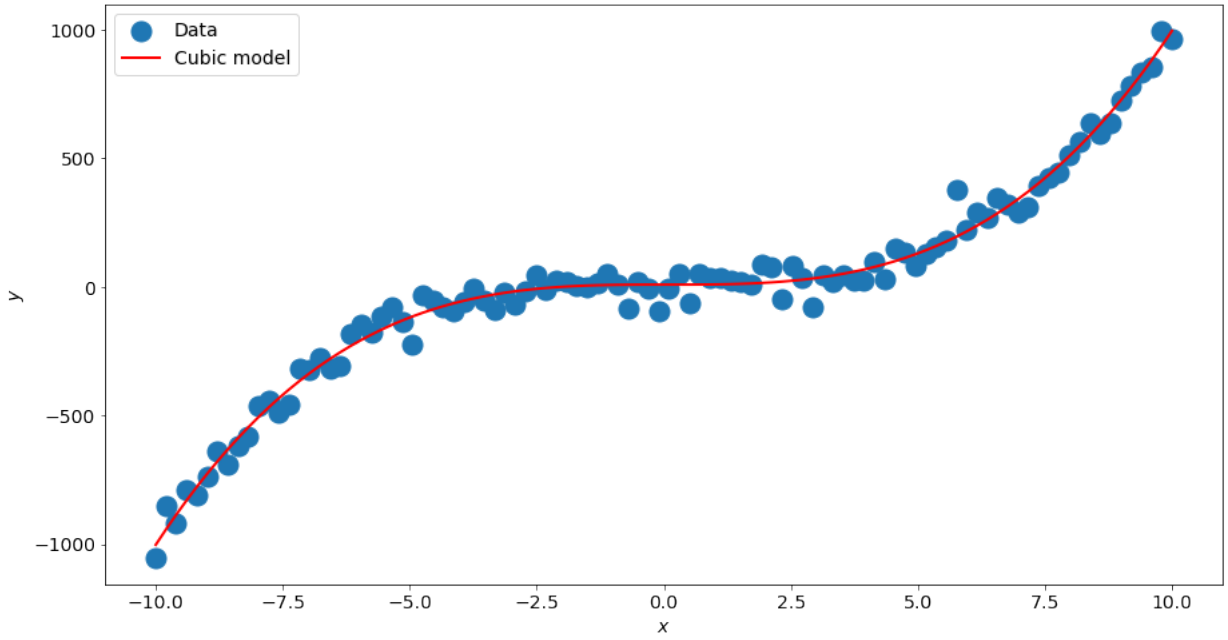

FIGURE 3

```
poly3 = PolynomialFeatures(degree=3)
X_poly3 = poly3.fit_transform(x.reshape(-1, 1))
model_poly3 = LinearRegression().fit(X_poly3, y)
y_pred_poly3 = model_poly3.predict(X_poly3)
r2_poly3 = r2_score(y, y_pred_poly3)


print("Linear model R2:", r2_lin)
print("Quadratic model R2:", r2_poly2)
print("Cubic model R2:", r2_poly3)

R_scores = [r2_lin, r2_poly2, r2_poly3]
choice = max(R_scores)

plt.scatter(x, y, label='Data')

if choice == r2_lin:
    plt.plot(x, y_pred_lin, label='Linear model',color='red')
elif choice == r2_poly2:
    plt.plot(x, y_pred_poly2, label='Quadratic model',color='red')
else:
    plt.plot(x, y_pred_poly3, label='Cubic model',color='red')

plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend()
plt.show()
```

with text output:

```
Linear model R2: 0.8279256898526317
Quadratic model R2: 0.8280180807052759
Cubic model R2: 0.9859121661101606
```

and Figure 3.

```
# Quartic
x = np.linspace(-10, 10, 100)
y = x**4 + np.random.normal(0, 500, 100)

# Fit a linear regression model
model_lin = LinearRegression().fit(x.reshape(-1, 1), y)
y_pred_lin = model_lin.predict(x.reshape(-1, 1))
r2_lin = r2_score(y, y_pred_lin)

# Fit k degree polynomials for k \in [2,5]
poly2 = PolynomialFeatures(degree=2)
X_poly2 = poly2.fit_transform(x.reshape(-1, 1))
model_poly2 = LinearRegression().fit(X_poly2, y)
y_pred_poly2 = model_poly2.predict(X_poly2)
r2_poly2 = r2_score(y, y_pred_poly2)

poly3 = PolynomialFeatures(degree=3)
X_poly3 = poly3.fit_transform(x.reshape(-1, 1))
model_poly3 = LinearRegression().fit(X_poly3, y)
y_pred_poly3 = model_poly3.predict(X_poly3)
r2_poly3 = r2_score(y, y_pred_poly3)

poly4 = PolynomialFeatures(degree=4)
X_poly4 = poly4.fit_transform(x.reshape(-1, 1))
model_poly4 = LinearRegression().fit(X_poly4, y)
y_pred_poly4 = model_poly4.predict(X_poly4)
r2_poly4 = r2_score(y, y_pred_poly4)



print("Linear model R2:", r2_lin)
print("Quadratic model R2:", r2_poly2)
print("Cubic model R2:", r2_poly3)
print("Quartic model R2:", r2_poly4)



R_scores = [r2_lin, r2_poly2, r2_poly3, r2_poly4]
choice = max(R_scores)


plt.scatter(x, y, label='Data')

if choice == r2_lin:
    plt.plot(x, y_pred_lin, label='Linear model',color='red')
elif choice == r2_poly2:
    plt.plot(x, y_pred_poly2, label='Quadratic model',color='red')
elif choice == r2_poly3:
    plt.plot(x, y_pred_poly3, label='Cubic model',color='red')
else:
    plt.plot(x, y_pred_poly4, label='Quartic model',color='red')


plt.xlabel('$x$')
```

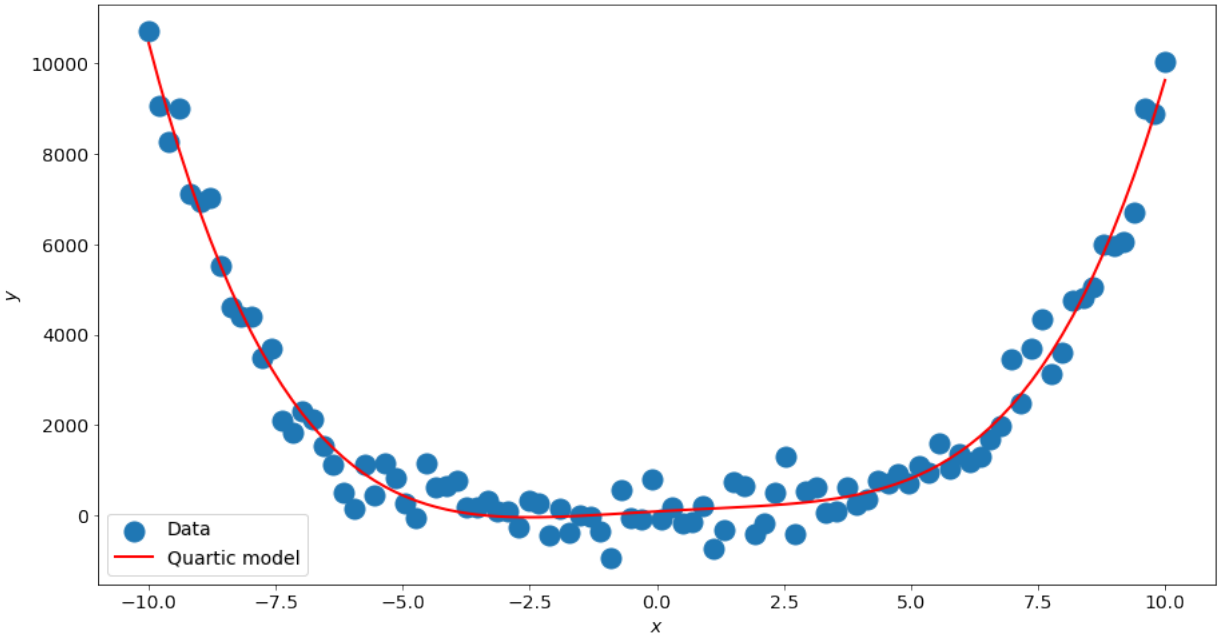

FIGURE 4

```
plt.ylabel('$y$')
plt.legend()
plt.show()
```

which has text output:

```
Linear model R2: 3.92361034551314e-07
Quadratic model R2: 0.8823810772737075
Cubic model R2: 0.8857650082722868
Quartic model R2: 0.9719587957730005
```

and Figure 4 Finally for the quintic case:

```
    # Quintic
# Generate some random data
x = np.linspace(-10, 10, 100)
y = x**5 + np.random.normal(0, 5000, 100)

# Fit a linear regression model
model_lin = LinearRegression().fit(x.reshape(-1, 1), y)
y_pred_lin = model_lin.predict(x.reshape(-1, 1))
r2_lin = r2_score(y, y_pred_lin)

# Fit k degree polynomials for k \in [2,5]
poly2 = PolynomialFeatures(degree=2)
X_poly2 = poly2.fit_transform(x.reshape(-1, 1))
model_poly2 = LinearRegression().fit(X_poly2, y)
y_pred_poly2 = model_poly2.predict(X_poly2)
r2_poly2 = r2_score(y, y_pred_poly2)

poly3 = PolynomialFeatures(degree=3)
X_poly3 = poly3.fit_transform(x.reshape(-1, 1))
model_poly3 = LinearRegression().fit(X_poly3, y)
y_pred_poly3 = model_poly3.predict(X_poly3)
r2_poly3 = r2_score(y, y_pred_poly3)

poly4 = PolynomialFeatures(degree=4)
```

```
X_poly4 = poly4.fit_transform(x.reshape(-1, 1))
model_poly4 = LinearRegression().fit(X_poly4, y)
y_pred_poly4 = model_poly4.predict(X_poly4)
r2_poly4 = r2_score(y, y_pred_poly4)

poly5 = PolynomialFeatures(degree=5)
X_poly5 = poly5.fit_transform(x.reshape(-1, 1))
model_poly5 = LinearRegression().fit(X_poly5, y)
y_pred_poly5 = model_poly5.predict(X_poly5)
r2_poly5 = r2_score(y, y_pred_poly5)

print("Linear model R2:", r2_lin)
print("Quadratic model R2:", r2_poly2)
print("Cubic model R2:", r2_poly3)
print("Quartic model R2:", r2_poly4)
print("Quintic model R2:", r2_poly5)


R_scores = [r2_lin, r2_poly2, r2_poly3, r2_poly4, r2_poly5]
choice = max(R_scores)


plt.scatter(x, y, label='Data')

if choice == r2_lin:
    plt.plot(x, y_pred_lin, label='Linear model',color='red')
elif choice == r2_poly2:
    plt.plot(x, y_pred_poly2, label='Quadratic model',color='red')
elif choice == r2_poly3:
    plt.plot(x, y_pred_poly3, label='Cubic model',color='red')
elif choice == r2_poly4:
    plt.plot(x, y_pred_poly4, label='Quartic model',color='red')
else:
    plt.plot(x, y_pred_poly5, label='Quintic model',color='red')

plt.xlabel('$x$')
plt.ylabel('$y$')
plt.legend()
plt.show()
```

which has text output:

```
Linear model R2: 0.6728978130389058
Quadratic model R2: 0.6731601448312439
Cubic model R2: 0.9668176036246945
Quartic model R2: 0.9668209706005049
Quintic model R2: 0.9801693316062714
```

and Figure 5.

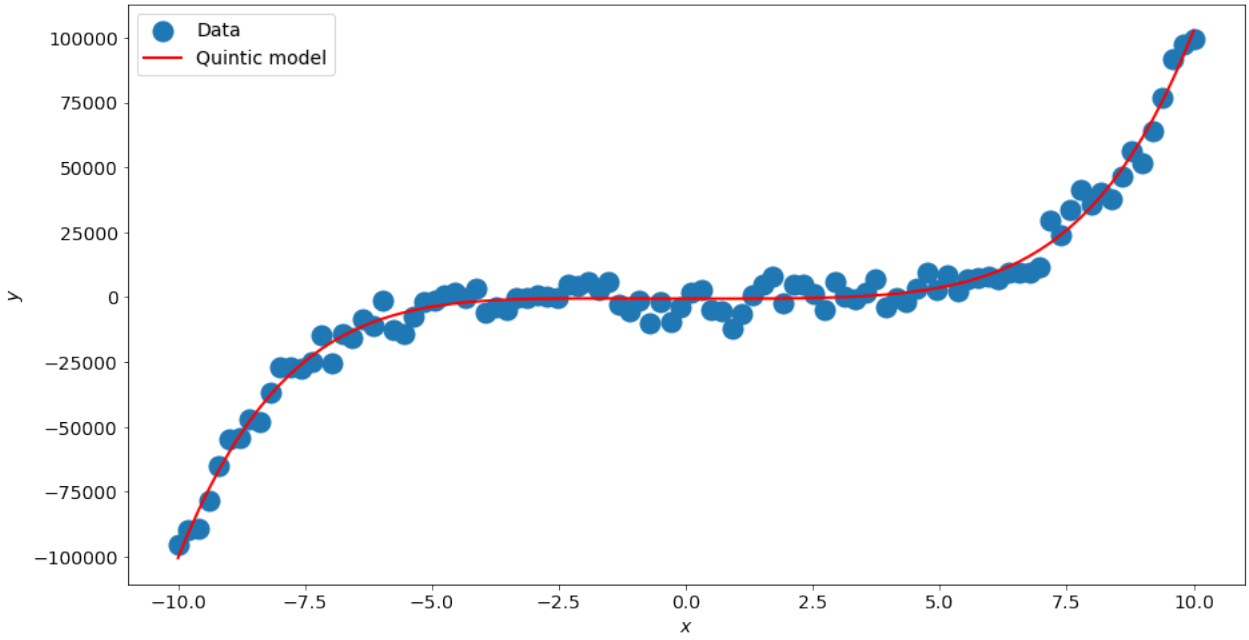
Data
Quintic model
100000
75000
50000
25000
0
−25000
−50000
−75000
−100000
y
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
x

FIGURE 5

Task 2

In this task, I will ask you to continue with a physics problem: simulating the pitched baseball. The pitched ball will feel the forces of gravity

$$F = mg$$

The air resistance force,

$$\vec{F}_{drag} = \frac{1}{2} C_D \rho A v^2$$

where C_D is the drag coefficient, A is the cross-sectional area of the ball; and the Magnus force

$$F_{Magnus} = \frac{1}{2} C_L A v^2$$

with the direction determined by $\vec{\omega} \times \vec{v}$. Setting up a coordinate system, and solve the equations of motion for the base ball. Specifically, produce a few situations such as curve ball, drop ball, and rise ball. A useful comprehensive discussion can be found here `http://baseball.physics.illinois.edu/ClarkGreerS` Regarding the Magnus effect, you may want to check the Wiki page here `https://en.wikipedia.org/wiki/Mag`

Solution 2. The three forces acting on the baseball in flight are given as:

Gravity (in the negative y-direction)

$$F_g = -mg \cdot \hat{e}_y$$

Drag force given as

$$\vec{F}_{drag} = -\frac{1}{2} C_D \rho A v^2 \hat{v}$$

Magnus Force

$$F_{Magnus} = \frac{1}{2} C_L A v^2 \hat{\alpha}$$

Which points perpendicular to

$$\hat{\alpha} = \vec{\omega} \times \vec{v}$$

Combining these with Newton's second law yields:

$$\dot{v_x} = -\frac{1}{2m} C_D \rho A v_x^2 \sqrt{v_x^2 + v_y^2 + v_z^2}$$

$$\dot{v_y} = -\frac{1}{2m} C_D \rho A v_y^2 \sqrt{v_x^2 + v_y^2 + v_z^2} + \frac{1}{2m} C_L \rho A (v_x^2 + v_y^2 + v_z^2) cos(\alpha)$$

$$\dot{v_z} = -g - \frac{1}{2m} C_D \rho A v_z^2 \sqrt{v_x^2 + v_y^2 + v_z^2} + \frac{1}{2m} C_L \rho A (v_x^2 + v_y^2 + v_z^2) sin(\alpha)$$

Where the initial conditions are given as

$$v_{x,i} = v_i \sin(\phi) \cos(\theta)$$

$$v_{y,i} = v_i \cos(\phi)$$

$$v_{y,i} = v_i \sin(\phi) \sin(\theta)$$

The following code defines a function `pitch` that takes angle arguments α, θ, ϕ and a lift coefficient CL and gives a 3-D plot of the trajectories. Note that for the curveball, x and z positions were swapped in the argument of `ax.plot()`.

```
%matplotlib nbagg
%matplotlib notebook
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
from mpl_toolkits.mplot3d import Axes3D
import ipywidgets as ipw
import math
import plotly.graph_objects as go
```

```
def pitch(alpha, theta, phi, CL):
    # Parameters
    m = 0.1875 # kg (mass of softball)
    r = 0.0366 # m (radius of softball)
    A = np.pi * r**2 # A (cross sectional area of the ball)
    rho = 1.2 # kg/m^3 (air density)
    g = 9.81 # m/s^2 (acceleration due to gravity)

    # Lift and drag coefficients for a softball
    CD = 0.35 # Assume spin rate of 3000 RPM

    # Differential equations
    def baseball(state, t):
        vx, vy, vz, x, y, z = state
        if y <= 0:
            return [0, 0, 0, 0, 0, 0]
        v = np.sqrt(vx**2 + vy**2 + vz**2)
        F_mag = 0.5 * rho * CL * A
        F_drag = 0.5 * rho * CD * A

        ax = -F_drag * vx * v/m
        ay = -g - F_drag * vy * v/m + F_mag * v**2 * np.sin(np.deg2rad(alpha))/m
        az = -F_drag * vz * v/m + F_mag * v**2 * np.cos(np.deg2rad(alpha))/m

        return [ax, ay, az, vx, vy, vz]

    # Initial conditions
    v0 = 30

    vx0 = v0 * np.cos(np.deg2rad(theta)) * np.sin(np.deg2rad(phi))
    vy0 = v0 * np.sin(np.deg2rad(theta)) * np.sin(np.deg2rad(phi))
    vz0 = v0 * np.cos(np.deg2rad(phi))
    x0 = 0
    y0 = 5
    z0 = 0

    # Pack initial conditions into a tuple
    state0 = [vx0, vy0, vz0, x0, y0, z0]

    # Time array
    t = np.linspace(0, 5, 1000)

    # Solve the differential equations
    sol = odeint(baseball, state0, t)

    # Extract the components of the solution
    vx = sol[:,0]
    vy = sol[:,1]
    vz = sol[:,2]
    x = sol[:,3]
    y = sol[:,4]
    z = sol[:,5]

    # Plot the trajectory of the ball in 3D
    fig = plt.figure()
```

```
    ax = fig.add_subplot(111, projection='3d')
    ax.plot(z, x, y, 'r--', label='Trajectory')
    ax.set_xlabel('x')
    ax.set_ylabel('z')
    ax.set_zlabel('y')
    ax.set_xlim(0, 50)
    ax.set_ylim(-.8,.8)
    ax.set_zlim(0, 6)
    ax.legend()
    plt.show()
```

Using this function call for the associated pitches:

```
# Rise Ball
# Parameters: alpha = 90, theta= 90, phi = 9, CL = .2
pitch(90, 90, 9, .2)
```

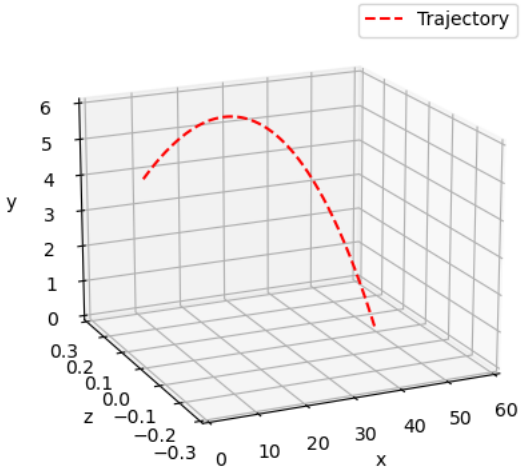

FIGURE 6. The rise ball with $\alpha = 90°, \theta = 90°, \phi = 9°$, & lift coefficient $CL = .2$.

```
# Drop Ball
# Parameters: alpha = -90, theta= 6, phi=0, CL \in [0,.16],
pitch(-90,6,0,.15)
```

```
# Curve Ball
# Parameters: alpha = 180, theta= 3, phi=9 CL = .15
pitch(170,3,88,.15)
```

Achieving accurate results proved to be a challenging task as the parameter values provided in the paper did not accurately reflect the pitches, in addition to the unconventional geometrical interpretation of $\mathbb{R}^3$ with y representing the vertical component. While I did not specifically utilize the Runge-Kutta 4 method for numerical solutions as described in the paper, I employed SciPy's `odeint` function which uses the `Fortran` LSODE solver. Further research revealed that LSODE offers comparable or superior accuracy to Runge-Kutta 4, albeit with a higher computational overhead cost.

I would love to discuss how something so easily verifiab In my opinion, the results of a research paper should be replicable in their entirety to uphold the integrity of the scientific method.I would love to discuss any shortcomings of my code and identify potential discrepancies between my implementation and the original work.

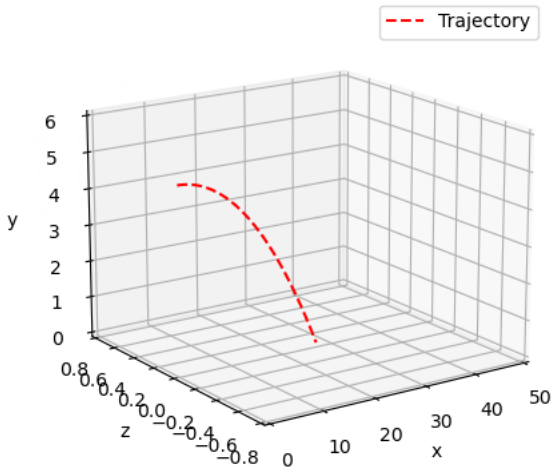

FIGURE 7. The drop ball with $\alpha = -90°, \theta = 6°, \phi = 0°$, & lift coefficient $CL = .15$.

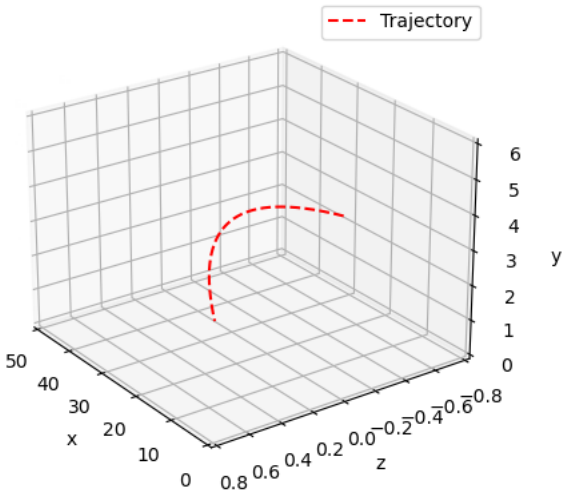

FIGURE 8. The curve ball with $\alpha = 180°, \theta = 3°, \phi = 9°$, & lift coefficient $CL = .15$.

Task 3

Gradient-descent is key in deep learning. In this task, you will need to develop a gradient descent program that are able to find minimum or maximum for any given functions (e.g., $y = 3x^2 - 5x + 2$, $y = 4x^5 + 6x^3 - 3x^2 + 5x - 2$, etc).

Solution 3. Gradient descent involves finding the minimum of a function f. Let the vector of partial derivatives of f with respect to each component of x be defined as:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]^T$$

The gradient of a vector is the direction of steepest ascent, and taking the negative of the gradient allows us to approach some minima (if a function is convex, i.e. it is uniform continuous or Lipschitz, then any minima is a global minimum). But sometimes, we cannot find exactly what value of x yields $\nabla f = 0$. Therefore, we employ an iterative approach, by first choosing a random x_0 and evaluating $\nabla f(x_0)$. With this in hand, we find:

$$x_1 = x_0 - \eta f(x_0)$$

for a small learning rate $\eta \in (0, 1)$ usually chosen ~ 0.001 (η may also be initially larger and decrease with each iteration). In general we have;

$$x_i = x_{i-1} - \eta f(x_{i-1})$$

We can stop iterating and be safe in assuming that we have reached a minima when $|\eta \nabla f(x_i)|$ falls below some specified threshold.

The following code implements gradient descent for two different functions.

```
# Function to optimize (change this to whatever function you want to minimize)
def function(x):
    return 3*x**2-5*x+2

# Gradient descent with decreasing learning rate
def gradient_descent(x0, lr, iterations, threshold):
    history = [x0]
    x = x0
    for i in range(iterations):
        grad = 6*x-5 # derivative of the function
        lr = lr/(1 + i) # decreasing learning rate
        x = x - lr*grad
        history.append(x)
        if np.abs(lr*grad) < threshold: # stopping condition
            break
    return history

# Define initial parameters
x0 = 4 # starting point
lr = 1 # initial learning rate
iterations = 100 # number of iterations
threshold = 1e-12 # threshold for stopping condition

# Run gradient descent
history = gradient_descent(x0, lr, iterations, threshold)

# Convert the history list to a numpy array
history = np.array(history)

# Plot the function and the history of gradient descent
x = np.linspace(-50,50, 1000)
y = function(x)
```

```
plt.plot(x, y, label='Function')
plt.plot(history, function(history), 'ro-', label='Gradient descent')
plt.legend()
plt.show()
```

which produces text output:

```
Stopping condition reached:  f(x) = 5.921189464667501e-16 < 1e-12
```

and Figure 9.

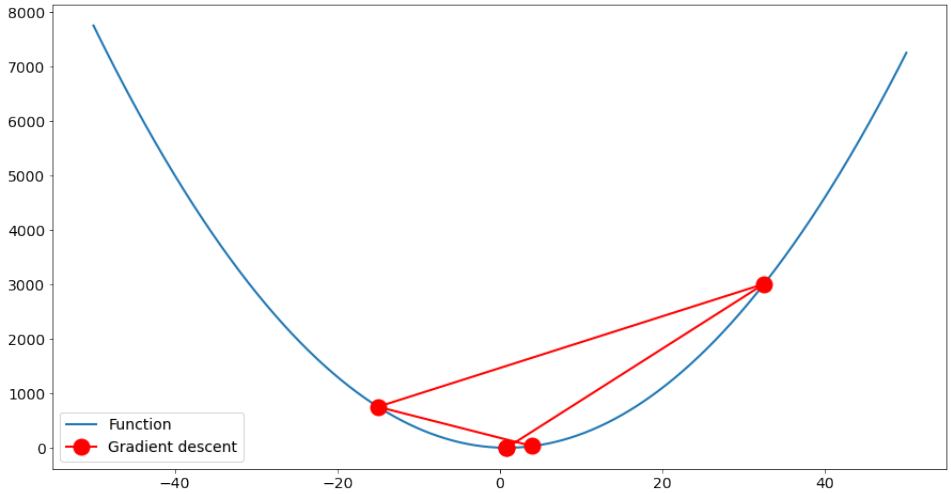

FIGURE 9

For functions like $f(x) = 4x^5 + 6x^3 - 3x^2 - 2$, the choice of a starting point was crucial. Since the function has no global minimum and diverges to $-\infty$, then our local minima is always $x \to -\infty$.

```
    import numpy as np # FINAL
import matplotlib.pyplot as plt
import mpmath as mp

def function(x):
    return 4*x**5+6*x**3-3*x**2+5*x-2

#Gradient descent with fixed multiplier inversely proportional to the gradient norm

def gradient_descent(x0, lr, iterations, threshold):
    history = [x0]
    x = x0
    for i in range(iterations):
        grad = 20*x**4+18*x**2+5 # derivative of the function
        grad_norm = np.linalg.norm(grad)
        lr = lr/grad_norm # fixed multiplier
        x = x - lr*grad
        history.append(x)
        if np.abs(lr*grad) < threshold: # stopping condition
            print('Stopping condition reached:  f(x) = ', lr*grad, '<', threshold)
            break
    return history



x0 = 0 # starting point
lr = 1 # initial learning rate
```

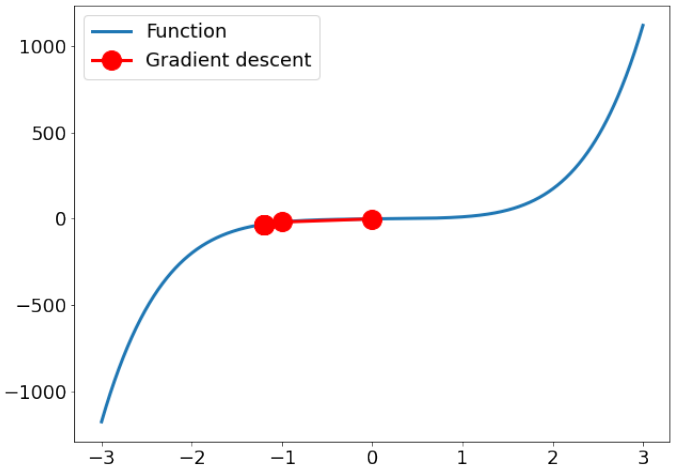

FIGURE 10

```
iterations = 100 # number of iterations
threshold = 1e-12 # threshold for stopping condition
history = gradient_descent(x0, lr, iterations, threshold)
history = np.array(history)
x = np.linspace(-3, 3, 1000)
y = function(x)
plt.plot(x, y, label='Function')
plt.plot(history, function(history), 'ro-', label='Gradient descent')
plt.legend()
plt.show
```

which produces text output:

```
Stopping condition reached:  f(x) = 3.046813525296221e-14 < 1e-12
```

and Figure 10.

TASK 4

Use google finance data api (`https://pypi.org/project/googlefinance/`), do a curve fitting of the index of SPY, QQQ, or IWM in real time in one day. Are you able to make a real-time predictor (e.g., provide a 15-min earlier prediction of the trend of SPY)?

Further reading can be found from here:
`https://serpapi.com/blog/scrape-google-finance-markets-in-python/`
`https://www.makeuseof.com/stock-price-data-using-python/`

Solution 4. I am still working on using `R` for a LSTM recurrent neural network to perform this task. Rather than turning something half done in, I hope you will accept a late submission; especially if it profitable =)