Method of Least Squares

1. Test case generation.

```python
import random
import numpy as np


def makeRegression(coefficient, dataSize):
    X = np.empty((len(coefficient) - 1, dataSize))
    for i in range(len(X)):
        random.seed()
        step = random.randint(1, 20)
        X[i] = [x for x in range(0, dataSize * step, step)]
    Y = np.empty((dataSize, 1))

    for i in range(dataSize):
        Y[i] = coefficient[0]
        for j in range(0, N - 1):
            Y[i] += coefficient[j + 1] * X[j][i]

    return X, Y
```

Importing libraries + implementing a function that creates a set X and a corresponding set Y in N-dimensional space (dataSize is the number of points in the space, coefficient is the set of weights on which the array Y is built). The X array is filled with a random step.

```python
for N in range(10, 100 + 10, 10):
    for n in range(1, 100 + 1):
        random.seed()
        dataSize = random.randint(150, 200)  # generated dataset size

        random.seed()
        coefTrue = np.empty((N, 1))
        for i in range(len(coefTrue)):
            random.seed()
            coefTrue[i] = random.uniform(7.0, 45.0) # weighting coefficients
        xScale, yEst = makeRegression(coefTrue, dataSize)

        # Adding outliers
        nO = random.randint(10, 20) # number of outliers
        iO = np.arange(nO, dataSize, int(dataSize/nO) - random.randint(1, 5)) # indexes of outliers
        for i in range(0, nO):
            yEst[iO[i]] += random.randint(-256, 256)

        Y = np.empty([1, dataSize])
        for i in range(0, dataSize):
            Y[0][i] = yEst[i]

        np.save("x_scale_{}_{}".format(N, n), xScale)
        np.save("y_est_{}_{}".format(N, n), Y)
```

Generate the number of points randomly (but in the range of 150 to 200). Generate an array of weights randomly (but in the range of 7 to 45), use the previously written function to create an X and Y array. Next, we add a random number of outliers (deviation from -256 to 256 to the expected value) to the Y array. Convert the Y array to a string for further writing to a file. Create files x_scale_N_n and y_est_N_n. Repeat this operation for each value of dimension N = {10, 20, ..., 100} for n = 100 files.

2. Solving test examples with the help of solver. Drawing a graph of dependence of the average time of solving problems on the dimensionality N. Storing the value of global minimum and optimal value of the target function for each test case.

The solver CVX was used to solve the problem by minimizing the objective function and finding the weights.

```
import time
import numpy as np
import cvxpy as cvx
import matplotlib.pyplot as plt
```

Importing Libraries.

```
file = open("Global_min.txt", "w+")
```

Create a file to store the values of the global minimum and optimal value of the target function for each test case.

```
clock = [x for x in range(1, 10 + 1)]
N_array = [x for x in range(10, 100 + 10, 10)]
```

Creation of the clock array to store values of the calculated average running time of the program, and the N_array array with the dimension values.

```
for N in range(10, 100 + 10, 10):
    t0 = time.time()
    for n in range(1, 100 + 1):
        weight = [0 for x in range(N)]
```

Go through the values of dimensions and files. The variable t0 is responsible for storing the time value at the moment of program start for the current dimension. Also create weight array, which will store weights for the processed file.

```
x_scale_prob = np.load(f"x_scale_{N}_{n}.npy")
y_estimate_prob = np.load(f"y_est_{N}_{n}.npy")
x_scale = np.array(x_scale_prob, dtype=float)
y_estimate0 = np.array(y_estimate_prob, dtype=float)
y_estimate = [0 for i in range(len(y_estimate0[0]))]
for i in range(len(y_estimate)):
    y_estimate[i] = y_estimate0[0][i]
```

Create arrays x_scale and y_estimate, which store the values obtained from the file.

```
w0 = cvx.Variable(1)
wi = cvx.Variable(N - 1)

gamma = 1e-15
obj = cvx.Minimize(cvx.norm(y_estimate - w0 - wi @ x_scale) + gamma * cvx.norm(wi))

prob = cvx.Problem(obj)
prob.solve()
```

Create arrays w0 and wi (w0 will store the free coefficient, wi - vector of remaining weight coefficients). Create the target function and minimize it using the solver.

```
weight[0] = w0.value[0]
for i in range(1, N):
    weight[i] = wi.value[i - 1]
```

We fill in the arrays of model weights (global minimum values).

```
yy = [0 for x in range(len(y_estimate))]
for i in range(len(yy)):
    yy[i] = w0.value[0]
    for d in range(0, N - 1):
        yy[i] += wi.value[d]*x_scale[d][i]
```

Create and fill the array of function values calculated by the obtained coefficients (optimal values of the target function).

```
file.write(f"{N}_{n} {weight} {yy} \n")
```

Write the values of the global minimum, the optimal values of the target function, and the corresponding dimensionality value for each test case into the file.

```
t1 = time.time() - t0
clock[int(N/10 - 1)] = (t1 / 100)
```

Filling the clock array with the values of the average time of program operation for the current dimension.

```
file.close()

plt.plot(N_array, clock, "-om")
plt.show()
```

Plotting the dependence of average problem solving time on the dimensionality N.

2.1 The three-dimensional case.

```
import numpy as np
import cvxpy as cvx
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from mpl_toolkits.mplot3d import Axes3D


x_scale_prob = np.load('x_scale_3_1.npy')
y_estimate_prob = np.load('y_est_3_1.npy')
x_scale = np.array(x_scale_prob,dtype=float)
y_estimate0 = np.array(y_estimate_prob,dtype=float)
y_estimate = [0 for i in range(len(y_estimate0[0]))]
for i in range(len(y_estimate)):
    y_estimate[i] = y_estimate0[0][i]

N = len(x_scale) + 1

w0 = cvx.Variable(1)
wi = cvx.Variable(N-1)

gamma = 1e-14
obj = cvx.Minimize(cvx.norm(y_estimate-w0-wi*x_scale) + gamma*cvx.norm(wi))

prob = cvx.Problem(obj)
prob.solve()

print(prob.status)
```

Here is a similar code for the three-dimensional case.

```
yy = [0]*len(y_estimate)
koef0=w0.value[0]
koef1=wi.value[0]
koef2=wi.value[1]
for i in range(len(yy)):
  yy[i]=koef0+koef1*x_scale[0][i]+koef2*x_scale[1][i]

print("w0 = ", koef0, "w1 = ", koef1, "w2 = ", koef2)

fig = plt.figure(figsize=(7, 4))
ax_3d = fig.add_subplot(projection='3d')
ax_3d.plot(x_scale[0], x_scale[1], yy, 'green')
ax_3d.scatter(x_scale[0], x_scale[1], y_estimate)
plt.show()
```

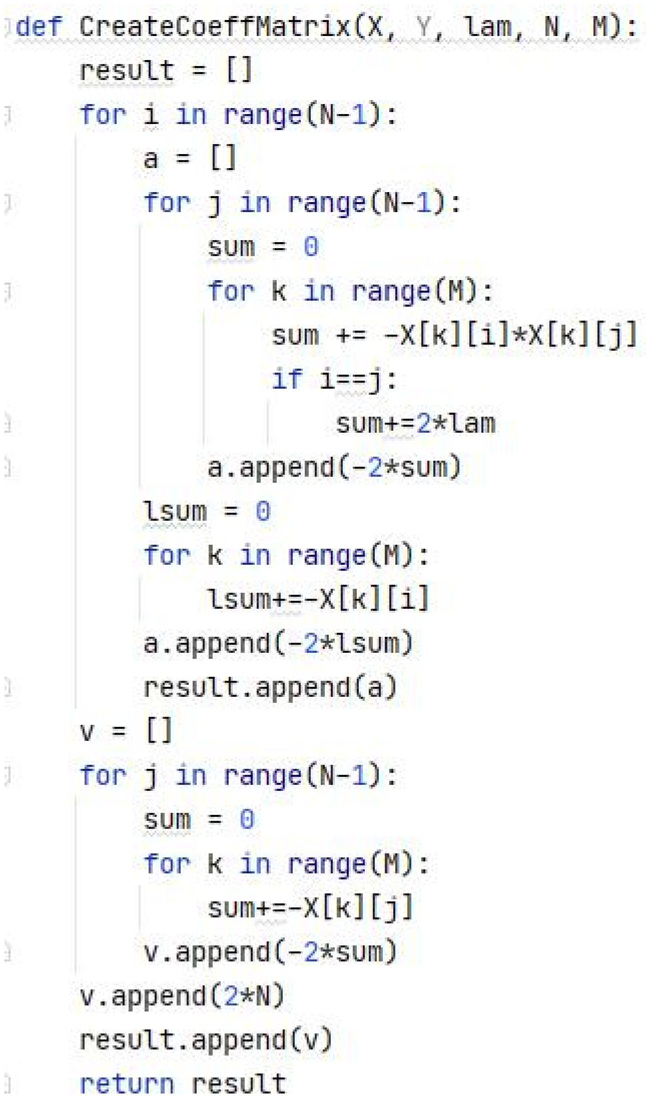
Draw a graph in three-dimensional space, on which the points show the initial values of the function and the line shows the calculated values of the function.

3. Solution of the problem by Cramer's method and inverse matrix method.

The solution of the linear regression problem can be reduced to the solution of a system of linear algebraic equations. The target function reaches its minimum when all components of the gradient are equal to zero. The system has the following form:

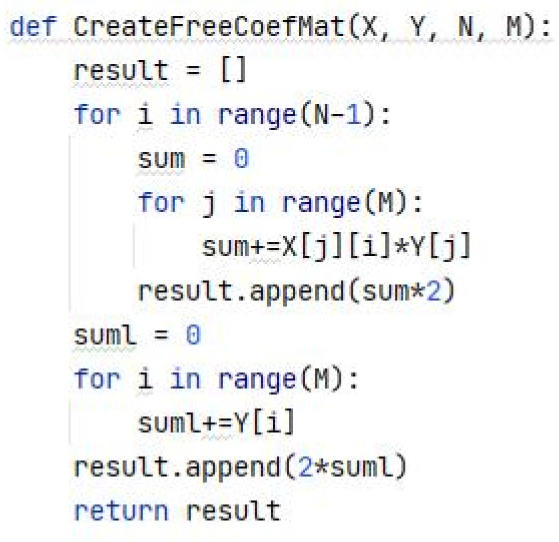
$$\begin{cases} \sum_{i=1}^{n} -2\,x_{i1}(y_i - b - k1x_{i1} - k2x_{i2} - \dots - k_{n-1}x_{in-1}) + 2\lambda k1 = 0 \\ \sum_{i=1}^{n} -2\,x_{i2}(y_i - b - k1x_{i1} - k2x_{i2} - \dots - k_{n-1}x_{in-1}) + 2\lambda k2 = 0 \\ . \\ . \\ . \\ \sum_{i=1}^{n} -2\,x_{in-1}(y_i - b - k1x_{i1} - k2x_{i2} - \dots - k_{n-1}x_{in-1}) + 2\lambda kn - 1 = 0 \\ \sum_{i=1}^{n} -2(y_i - b - kx_i) = 0 \end{cases}$$

To solve it, a matrix of coefficients in front of the variables (b, k1, k2, ...) was compiled



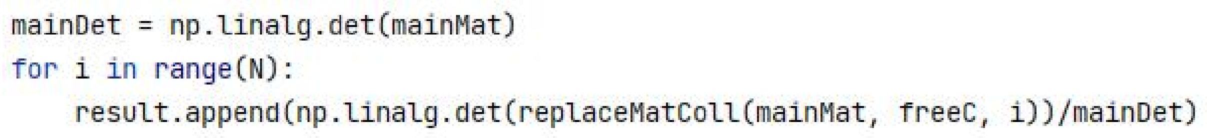
```python
def CreateCoeffMatrix(X, Y, lam, N, M):
    result = []
    for i in range(N-1):
        a = []
        for j in range(N-1):
            sum = 0
            for k in range(M):
                sum += -X[k][i]*X[k][j]
                if i==j:
                    sum+=2*lam
            a.append(-2*sum)
        lsum = 0
        for k in range(M):
            lsum+=-X[k][i]
        a.append(-2*lsum)
        result.append(a)
    v = []
    for j in range(N-1):
        sum = 0
        for k in range(M):
            sum+=-X[k][j]
        v.append(-2*sum)
    v.append(2*N)
    result.append(v)
    return result
```

Then the vector of free coefficients



```python
def CreateFreeCoefMat(X, Y, N, M):
    result = []
    for i in range(N-1):
        sum = 0
        for j in range(M):
            sum+=X[j][i]*Y[j]
        result.append(sum*2)
    suml = 0
    for i in range(M):
        suml+=Y[i]
    result.append(2*suml)
    return result
```

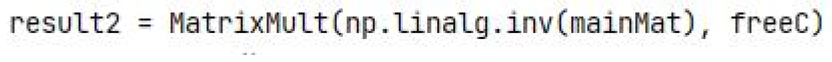
After that, Cramer's method was implemented to solve the linear system of equations



```
mainDet = np.linalg.det(mainMat)
for i in range(N):
    result.append(np.linalg.det(replaceMatColl(mainMat, freeC, i))/mainDet)
```

Where the determinant is calculated using the numpy library. ReplaceMatColl - function for generating auxiliary matrices to solve SLAEs.

Realization of the method of solving SLAU by inverse matrix



```
result2 = MatrixMult(np.linalg.inv(mainMat), freeC)
```

During the execution of the program, an array of the sought variables is output, where the last one is the coefficient of the line b offset.

```
[11.28853518232289, 19.351776089466266, 6.450591651351097, 30.6403135335568,
20.96442466821996, 30.640313945509824, 1.6126479037909773, 22.577067730130285,
19.35177307360338, -15.941068379713826]
```

4. Solution of the problem by the gradient descent method.

The solution to the linear regression problem can be found using the gradient descent method:

(Example of a system for ten-dimensional space)

$$K_{n+1}^{1} = K_n^1 - a_k * \nabla \mathrm{f}(K_n^1)$$

$$K_{n+1}^{2} = K_n^2 - a_k * \nabla \mathrm{f}(K_n^2)$$

$$K_{n+1}^{3} = K_n^3 - a_k * \nabla \mathrm{f}(K_n^3)$$

$$\dots$$

$$K_{n+1}^{8} = K_n^8 - a_k * \nabla \mathrm{f}(K_n^8)$$

$$K_{n+1}^{9} = K_n^9 - a_k * \nabla \mathrm{f}(K_n^9)$$

$$B_{n+1} = B_n - a_b * \nabla \mathrm{f}(B_n)$$

```
while gradient_norm() > eps:

    #count derivative
    s = 0
    for j in range(len(points)):
        s1 = points[j][-1]
        for k in range(n-1):
            s1 -= points[j][k] * coefficients[k]
        s1 -= coefficients[-1]
        s += s1
    for i in range(n):
        if i == n-1:
            derivative = -2 * s
        else:
            derivative = 0
            for j in range(len(points)):
                s2 = points[j][-1]
                for k in range(n-1):
                    s2 -= points[j][k] * coefficients[k]
                s2 -= coefficients[-1]
                derivative += -2 * points[j][i] * s2 + 2 * lamb * coefficients[i]

        #gradient descent
        coefficients[i] = coefficients[i] - a[i] * derivative
```

Where points is a list that stores the coordinates of the points,

coefficients - a list that stores the sought coefficients and looks as follows: [k1, k2, k3, k4, k5, k6, k7, k8, k9, b]),

gradient_norm() is a function that finds the Euclidean norm of the gradient.

$a_k = 7 * 10^{-10}$, $a_b = 7 * 10^{-7}$. The values a_k *и* a_b are constants and were selected manually.

Then, after executing the program, an array of sought variables is output, where the last element is the free coefficient B and the rest are K.

```
[39.259334344824104, 112.59262149705546, 2.3052930316117677,
11.937853769893287, 0.9401166804670176, 0.8870183293462036,
0.7924740885427194, 1.0486178696135755, 1.0017060385596253,
59.41110183828982]
```

5. Conclusions.

To measure accuracy, we added to each algorithm the function r2_score from the sklearn library, which finds the coefficient of determination (we found the mean for each dimension).

```python
from sklearn.metrics import r2_score

fileR = open("R2_score.txt", "w+")

R_array = [0 for x in range(1, 100 + 1)]

R_array[n - 1] = r2_score(y_estimate, yy)

fileR.write(f"{N} {sum(R_array) / 100}\n")
```

Based on the results obtained, the following conclusions were drawn:

- Solver is the fastest of all algorithms, the accuracy of solver is about the same as that of gradient descent.
- The second fastest method after solver is the inverse matrix method, followed by the Cramer method, the accuracy is about the same and worse than solver and gradient descent.