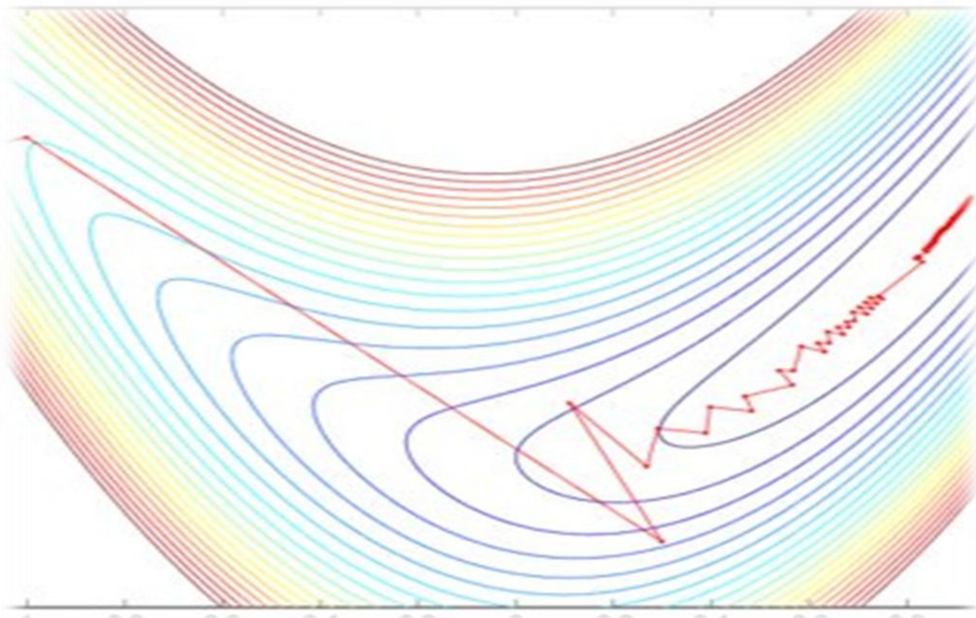
Optimization Techniques

Project report



PRESENTED BY:

Monique Ehab	4928
Hania Mohamed Hani	4697
Hana Hesham Nazmy	4952
Menna Farouk	5088
Ola Ahmed	4607
Noran Amra	4702
Cloudia Kamal	4948

Problem Statement:

The Matlab function "arrayfun" helps you develop a function that can be applied to all array elements without loops. This saves time at the expense of more intense memory usage. For very large arrays (very large depends on your computer's specifications), it is desirable to save time while minimizing memory usage that deteriorates your computer performance. Thus, applying "arrayfun" to blocks of the array, instead of it as a whole, is advantageous. First, develop a model that predicts the code run-time and some measure of your computer performance (that reflects memory utilization) for a given input block size. Second, use a multi-objective optimization procedure, to choose a suitable block size that achieves the best run-time with an acceptable memory usage.

Decision variables	**Block size**
Pre-specified parameters	**1- Time** **2- Memory**
Constraints	There are no constraints (except that the time and memory are positive, any negative values are rejected).
Optimization functions	- The following are the contradicting objective functions to be optimized. - The goal is to **obtain the optimum block size** that will <u>save time in addition to minimizing the memory utilization.</u> 1- **F1(x)=** time*(array_size/x), where x is the block size 2- **F2(x)=**memory - Where time and memory are obtained from the NN using **model.predict(x)**.
The method	- The method used to optimize the objective function is the **weighted method**, where we added both functions multiplying them by unity weight (we used equal weights for both optimization functions) - Then we solved them using classical method by minimize_scalar function. **scipy.optimize.minimize_scalar(***fun, bounds=None, args=(), method='bounded'***)** - We used the **method ="bounded"** as we have the **minimum block size=1 and the maximum block_size=array_size=10000000.** **res=minimize_scalar(obj_fun, bounds=(1, 10000000), method='bounded')**

The following snippets show the code of training a NN which is part 1 of the project:

Steps:

- We created a dataset of 10K samples using a code we implemented.
- We plotted our dataset to know the relationship between the input and the output
    - There is a linear relationship between input array size as input and time as output
    - There is a linear relationship between input array size as input and memory utilized as output
- We trained 4 models (fitting and prediction):
  1. Linear Regression Model for input array size as input and time as output
  2. Linear Regression Model for input array size as input and memory utilized as output
  3. Multioutput Linear Regression Model for input array size as input, and time and memory utilized as output
  4. Neural Network for input array size as input, and time and memory utilized as output
- We used our neural network model in part 2 of the project which is the optimization problem
- We used the mean squared error as our loss function.

THE CODE:

Imports

```
[ ] import numpy as np
    import pandas as pd
    from numpy import random
    import time
    import matplotlib.pyplot as plt
    import matplotlib
    import keras
    from keras import layers
    from keras.models import Sequential
    from keras.layers import Dense
    from sklearn.model_selection import train_test_split
    from sklearn.linear_model import LinearRegression
    from sklearn.multioutput import MultiOutputRegressor
    from sklearn.linear_model import Ridge
    from sklearn.metrics import mean_squared_error
```

PART 1:

1- Reading the data from the file 'myCompleteData' and saving it as a dataframe:

Save dataset as a dataframe

```
[ ] header_list = ['size','time','memory']
    data = pd.read_csv('/myCompleteData.txt', sep=',', names=header_list)
    print(data)
```

```
         size      time    memory
0           0  0.000000     98304
1        1000  0.001026      8192
2        2000  0.000000         0
3        3000  0.000000         0
4        4000  0.000000         0
...       ...       ...       ...
9995  9995000  0.442401  79962112
9996  9996000  0.413388  79970304
9997  9997000  0.431589  79978496
9998  9998000  0.437489  79986688
9999  9999000  0.419379  79994880

[10000 rows x 3 columns]
```

2- Editing the data units (time in millisec, memory in KB).

Edit Columns of Data

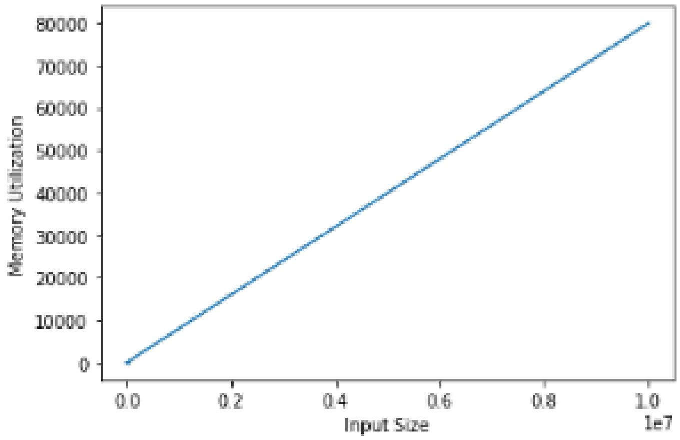
- Changed the time unit from 1 sec to 1 millisec
- Changed the memory unit from byte to KB

```
[ ] data['time'] = data['time'] *1000
    data['memory'] = data['memory'] /1000
```

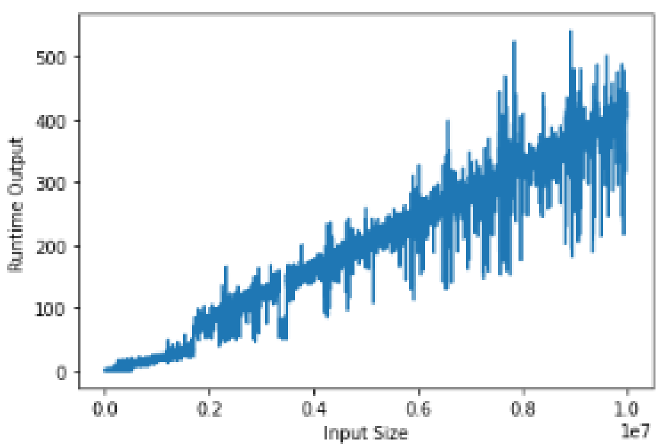
3- Plotting the dataset.

Plotting Dataset

```
[ ] plt.plot(data['size'], data['memory'])
    plt.xlabel('Input Size')
    plt.ylabel('Memory Utilization')
    plt.show()
```




```
[ ] plt.plot(data['size'], data['time'])
    plt.xlabel('Input Size')
    plt.ylabel('Runtime Output')
    plt.show()
```



4- Loading the data of the dataset.

Load Data

Loading Input Data (X)

```
[ ] # Input X
    X = data['size'].to_frame()
    print(type(X))
    print(X.shape)
    # print(X.dtype)
```

```
<class 'pandas.core.frame.DataFrame'>
(10000, 1)
```

Loading Output Data (y)

```
[ ] # Output y
    y = data[['time','memory']]
    print(y.dtypes)
    print(type(y))
    print(y.shape)
```

```
time      float64
memory    float64
dtype: object
<class 'pandas.core.frame.DataFrame'>
(10000, 2)
```

5- Splitting our dataset into train and test for training and testing our model.

Split data into train and test

```
[ ] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

```
[ ] print(X_train.shape)
    print(y_train.shape)
    print(X_test.shape)
    print(y_test.shape)
```

```
(6700, 1)
(6700, 2)
(3300, 1)
(3300, 2)
```

```
[ ] print(type(X_train))
    print(type(y_train))
```

```
<class 'pandas.core.frame.DataFrame'>
<class 'pandas.core.frame.DataFrame'>
```

Splitting y into smaller dataframes

Splitting y_train into

1. y_train_time
2. y_train_mem

```
[ ] # y_train_time
    y_train_time = y_train.iloc[:,0].to_frame()
    print('Y_train_time type is ' + str(type(y_train_time)))
    print('Y_train_time shape is ' + str(y_train_time.shape))

    # y_train_mem
    y_train_mem = y_train.iloc[:,1].to_frame()
    print('Y_train_mem type is ' + str(type(y_train_mem)))
    print('Y_train_mem shape is ' + str(y_train_mem.shape))
```

```
Y_train_time type is <class 'pandas.core.frame.DataFrame'>
Y_train_time shape is (6700, 1)
Y_train_mem type is <class 'pandas.core.frame.DataFrame'>
Y_train_mem shape is (6700, 1)
```

Splitting y_test into

1. y_test_time
2. y_test_mem

```
[ ] # y_test_time
    y_test_time = y_test.iloc[:,0].to_frame()
    print('Y_test_time type is ' + str(type(y_test_time)))
    print('Y_test_time shape is ' + str(y_test_time.shape))

    # y_test_mem
    y_test_mem = y_test.iloc[:,1].to_frame()
    print('Y_test_mem type is ' + str(type(y_test_mem)))
    print('Y_test_mem shape is ' + str(y_test_mem.shape))
```

```
Y_test_time type is <class 'pandas.core.frame.DataFrame'>
Y_test_time shape is (3300, 1)
Y_test_mem type is <class 'pandas.core.frame.DataFrame'>
Y_test_mem shape is (3300, 1)
```

6- Linear models:

- Linear Regression model for time:

Fitting linear regression model

```
[ ] # define model for output time
    model1 = LinearRegression()
    # fit model
    model1.fit(X_train, y_train_time)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Prediction of linear regression model

```
yhat_time = model1.predict(X_test)
# summarize prediction
# print(yhat[0])
```

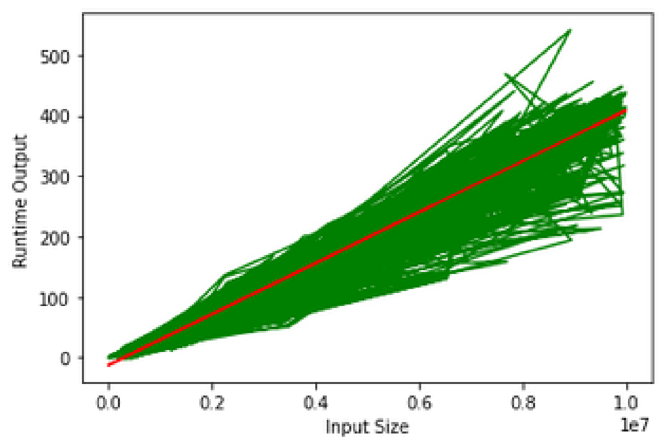
Calculating mean squared error of predicted output

```
[ ] print(mean_squared_error(y_test_time, yhat_time))
```

```
552.8957690638919
```

Plotting predicted time values (in red) vs ground truth (in green)

```
[ ] plt.plot(X_test, y_test_time, 'g' , label = 'ground truth')
    plt.plot(X_test, yhat_time, 'r', label = 'predicted output')
    plt.xlabel('Input Size')
    plt.ylabel('Runtime Output')
    plt.show()
```



- Linear Regression model for memory:

Fitting linear regression model

```
[ ] # define model for output memory
    model2 = LinearRegression()
    # fit model
    model2.fit(X_train, y_train_mem)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

Prediction of linear regression model

```
[ ] yhat_mem = model2.predict(X_test)
```

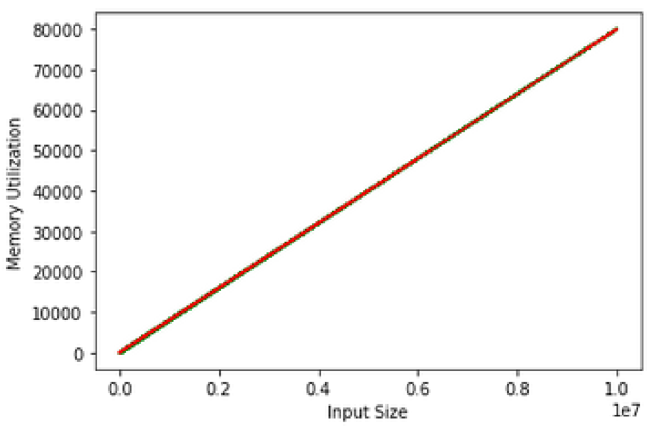
Calculating mean squared error of predicted output

```
[ ] print(mean_squared_error(y_test_mem, yhat_mem))
```

```
460.4849710663497
```

Plotting predicted memory values (in red) vs ground truth (in green)

```
[ ] plt.plot(X_test, y_test_mem, 'g' , label = 'ground truth')
    plt.plot(X_test, yhat_mem, 'r', label = 'predicted output')
    plt.xlabel('Input Size')
    plt.ylabel('Memory Utilization')
    plt.show()
```



- Multioutput Linear Regression model:

Fitting the model

```
[ ] # define model
    model3 = MultiOutputRegressor(Ridge(random_state=123))
    # fit model
    model3.fit(X_train, y_train)
```

```
MultiOutputRegressor(estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
                                     max_iter=None, normalize=False,
                                     random_state=123, solver='auto',
                                     tol=0.001),
                     n_jobs=None)
```

Predicting the test values

```
[ ] yhat = model3.predict(X_test)
```

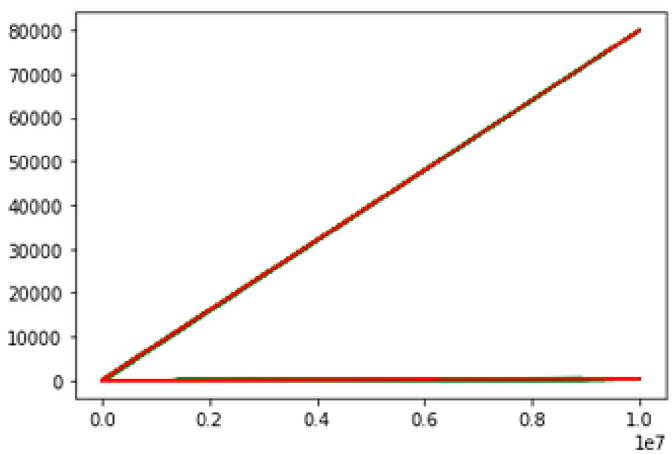
Calculating mean squared error of predicted output

```
[ ] print(mean_squared_error(y_test, yhat))
```

```
506.69037006511616
```

Plotting predicted output values (in red) vs ground truth (in green)

```
[ ] plt.plot(X_test, y_test, 'g' , label = 'ground truth')
    plt.plot(X_test, yhat, 'r', label = 'predicted data')
    plt.show()
```



7- <u>Neural Network:</u>

Define NN Model

```
[ ] # define the keras model
    model = Sequential()
    model.add(Dense(10, input_dim=1, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(20, input_dim=10, activation='relu',kernel_initializer='he_uniform'))
    model.add(Dense(2, activation='linear'))
```

Compile NN Model

```
[ ] # compile the keras model
    opt = keras.optimizers.Adam(learning_rate=0.01)
    model.compile(loss = 'mean_squared_error' , optimizer=opt)
```

Fit NN Model

```
[ ] # fit the keras model on the dataset
    model.fit(X_train, y_train, epochs=150, batch_size=64)
```

```
Epoch 1/150
105/105 [==============================] - 1s 955us/step - loss: 10677642767843.0195
Epoch 2/150
105/105 [==============================] - 0s 981us/step - loss: 95963820.3019
Epoch 3/150
105/105 [==============================] - 0s 1ms/step - loss: 2308.7844
Epoch 4/150
105/105 [==============================] - 0s 1ms/step - loss: 485.8555
Epoch 5/150
105/105 [==============================] - 0s 960us/step - loss: 494.2585
Epoch 6/150
105/105 [==============================] - 0s 960us/step - loss: 524.3301
Epoch 7/150
105/105 [==============================] - 0s 1ms/step - loss: 736.1956
Epoch 8/150
105/105 [==============================] - 0s 1ms/step - loss: 753.8583
Epoch 9/150
105/105 [==============================] - 0s 1ms/step - loss: 813.3837
Epoch 10/150
105/105 [==============================] - 0s 1ms/step - loss: 747.2876
Epoch 11/150
105/105 [==============================] - 0s 1ms/step - loss: 699.1210
Epoch 12/150
105/105 [==============================] - 0s 1ms/step - loss: 2767.5702
```

```
Epoch 145/150
105/105 [==============================] - 0s 1ms/step - loss: 544.2473
Epoch 146/150
105/105 [==============================] - 0s 1ms/step - loss: 670.3976
Epoch 147/150
105/105 [==============================] - 0s 1ms/step - loss: 553.2144
Epoch 148/150
105/105 [==============================] - 0s 1ms/step - loss: 671.5436
Epoch 149/150
105/105 [==============================] - 0s 1ms/step - loss: 856.2715
Epoch 150/150
105/105 [==============================] - 0s 1ms/step - loss: 687.7796
```

Make Predictions

```
[ ] # make probability predictions with the model
    predictions = model.predict(X_test)
```

```
[ ] print(predictions.shape)
```

```
(3300, 2)
```

Splitting predictions of time and memory

```
predictions_time = y_test.iloc[:,0].to_frame()
predictions_mem = y_test.iloc[:,1].to_frame()
# print(predictions_mem.shape)
```

Calculating Mean Squared Error for time

```
[ ] mean_squared_error(y_test_time, predictions_time)
```

```
0.0
```

Calculating Mean Squared Error for memory

```
[ ] mean_squared_error(y_test_mem, predictions_mem)
```

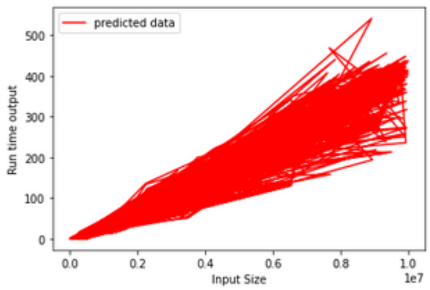
```
0.0
```

8- Plots:

- Time plots:

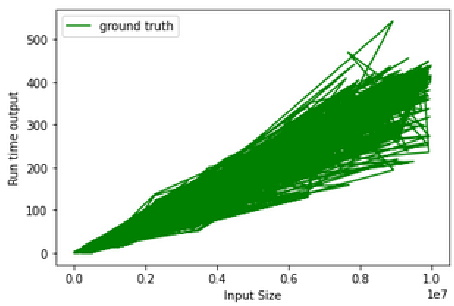
Plotting predictions of time

```
[ ] plt.plot(X_test, predictions_time, 'r', label = 'predicted data')
    plt.xlabel('Input Size')
    plt.ylabel('Run time output')
    plt.legend()
    plt.show()
```



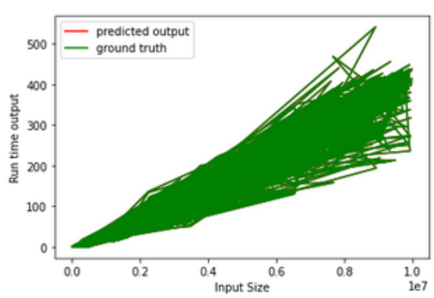
Plotting ground truth values of time

```
[ ] plt.plot(X_test, y_test_time, 'g', label = 'ground truth')
    plt.xlabel('Input Size')
    plt.ylabel('Run time output')
    plt.legend()
    plt.show()
```



Plotting predicted output values of time (in red) vs ground truth (in green)

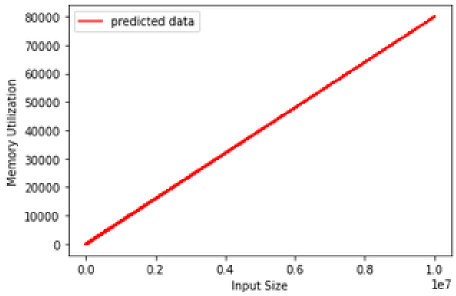
```
[ ] plt.plot(X_test, predictions_time, 'r', label = 'predicted output')
    plt.plot(X_test, y_test_time, 'g' , label = 'ground truth')
    plt.xlabel('Input Size')
    plt.ylabel('Run time output')
    plt.legend()
    plt.show()
```



- ## **Memory plots:**

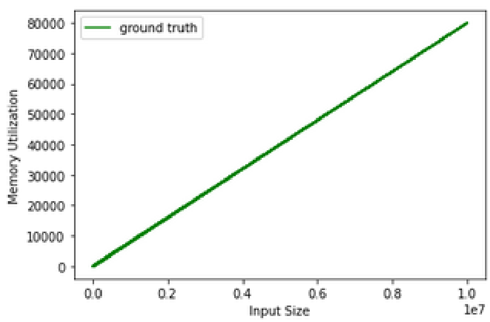
Plotting predictions of memory

```
plt.plot(X_test, predictions_mem, 'r', label = 'predicted data')
plt.xlabel('Input Size')
plt.ylabel('Memory Utilization')
plt.legend()
plt.show()
```



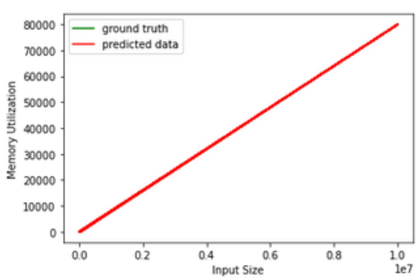
Plotting ground truth values of memory

```
[ ] plt.plot(X_test, y_test_mem, 'g', label = 'ground truth')
    plt.xlabel('Input Size')
    plt.ylabel('Memory Utilization')
    plt.legend()
    plt.show()
```



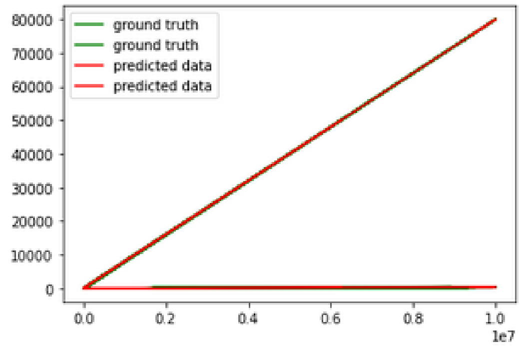
Plotting predicted output values of memory (in red) vs ground truth (in green)

```
[ ] plt.plot(X_test, y_test_mem, 'g', label = 'ground truth')
    plt.plot(X_test, predictions_mem, 'r', label = 'predicted data')
    plt.xlabel('Input Size')
    plt.ylabel('Memory Utilization')
    plt.legend()
    plt.show()
```



- **Time and memory plots:**

```
[ ] plt.plot(X_test, y_test, 'g', label = 'ground truth')
    plt.plot(X_test, predictions, 'r', label = 'predicted data')
    plt.legend()
    plt.show()
```



PART 2:

- **Optimization of the objective functions using the NNs and the weighted function method in association.**

```
from scipy.optimize import minimize_scalar
import sympy as sym
import sys
from sympy import Symbol
array_size=10000000

#equal weights
def obj_fun(x):
  ret=model.predict(x.reshape(1, 1))
  time=ret[0][0]
  memory=ret[0][1]
  return (time*(array_size/x))+memory

res=minimize_scalar(obj_fun, bounds=(1, 10000000), method='bounded')
print('The optimum value of the block size:')
print(res.x)
print('The corresponding time predicted using the NN:')
print(model.predict(res.x.reshape(1, 1))[0][0])
print('The corresponding time predicted using the NN:')
print(model.predict(res.x.reshape(1, 1))[0][1])
```

RESULT

```
The optimum value of the block size:
13915.409443614575
The corresponding time predicted using the NN:
1.6544288
The corresponding memory predicted using the NN:
108.53957
```