Aprendizagem 2024
Homework III - Group 92
Maria Maló - ist102994
André Feliciano - ist100286

Part I: Pen and paper

1.

$\phi = y_1 \times y_2$	y_{num}
1	1.25
3	7.0
6	2.7
9	3.2
8	5.5

Least squares gives that the weights must be:

$$w = (X^T X)^{-1} X^T z$$

$$X = \begin{pmatrix} 1 & 1 \\ 1 & 3 \\ 1 & 6 \\ 1 & 9 \\ 1 & 8 \end{pmatrix} \quad z = \begin{pmatrix} 1.25 \\ 7.0 \\ 2.7 \\ 3.2 \\ 5.5 \end{pmatrix}$$

$$X^T = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 6 & 9 & 8 \end{pmatrix}$$

As suggested, we've used numpy to compute w by performing the multiplication:

$$w = (X^T X)^{-1} X^T z$$

We obtained the weight vector:

$$w = \begin{pmatrix} 3.31593 \\ 0.11372 \end{pmatrix}$$

$y_{\text{num}} = 3.31593 + 0.11372\phi$

We can estimate the y_{num} for the data-set using our linear-regression model.

$$\begin{aligned}
x_1 :&\quad y_{\text{num}} = 3.31593 + 0.11372 \times 1 = 3.42965 \\
x_2 :&\quad y_{\text{num}} = 3.31593 + 0.11372 \times 3 = 3.65709 \\
x_3 :&\quad y_{\text{num}} = 3.31593 + 0.11372 \times 6 = 3.99825 \\
x_4 :&\quad y_{\text{num}} = 3.31593 + 0.11372 \times 9 = 4.33941 \\
x_5 :&\quad y_{\text{num}} = 3.31593 + 0.11372 \times 8 = 4.22569
\end{aligned}$$

2. Using Riddle regularization one gets:

$$w = \left(X^T X + \lambda I\right)^{-1} X^T z$$

$$\lambda = 1 \quad \lambda I_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Using once more numpy, we obtain the weight vector:

$$w_{\text{ridge}} = \begin{pmatrix} 1.81809 \\ 0.32376 \end{pmatrix}$$

$y_{\text{num}} = 1.81809 + 0.32376\phi$

We can estimate the y_{num} for the data-set using our linear-regression + Ridge regularization model.

$$\begin{aligned}
x_1 : &\quad y_{\text{num}} = 1.81809 + 0.32376 \times 1 = 2.14185 \\
x_2 : &\quad y_{\text{num}} = 1.81809 + 0.32376 \times 3 = 2.78937 \\
x_3 : &\quad y_{\text{num}} = 1.81809 + 0.32376 \times 6 = 3.76065 \\
x_4 : &\quad y_{\text{num}} = 1.81809 + 0.32376 \times 9 = 4.73193 \\
x_5 : &\quad y_{\text{num}} = 1.81809 + 0.32376 \times 8 = 4.40817
\end{aligned}$$

We can indeed check that the weights change when we add a regularization term to the weight calculation. Without regularization, the model is ”free” to choose its best coefficient to fit the data and minimize the squared error, so we obtain the weight vector:

$$\begin{pmatrix} 3.31593 & 0.11372 \end{pmatrix}^T$$

whose $w_0 = 3.31593$ is large, which might be a sign of data overfitting.

However, when we add a regularization term, large as $\lambda = 1$, it discourages large coefficients, finding a balance between minimizing the error and keeping the coefficients small. We obtain a weight vector:

$$\begin{pmatrix} 1.81809 & 0.32376 \end{pmatrix}^T$$

whose first coordinate is noticeably smaller than without the regularization. The regularization acts in the weights so as to keep the model as generalized as it can, preventing overfitting of the data set. Regularization is tremendously useful when one is working with large data sets that may contain lots of ”noise” observations. Unlike OLS, ridge regularization prioritizes model generalization over fitting perfectly to training data.

3. We have three new observations:

- $x_6 = (2, 2, 0.7)$
- $x_7 = (1, 2, 1.1)$
- $x_8 = (5, 1, 2.2)$

OLS

$$
\begin{aligned}
x_6: &\quad y_{\text{out}}(4) = 3.31593 + 4 \times 0.11372 = 3.77081 \\
x_7: &\quad y_{\text{out}}(2) = 3.31593 + 2 \times 0.11372 = 3.54337 \\
x_8: &\quad y_{\text{out}}(5) = 3.31593 + 5 \times 0.11372 = 3.88453
\end{aligned}
$$

Ridge

$$
\begin{aligned}
x_6: &\quad y_{\text{out}}(4) = 1.81809 + 0.32376 \times 4 = 3.11313 \\
x_7: &\quad y_{\text{out}}(2) = 1.81809 + 0.32376 \times 2 = 2.46561 \\
x_8: &\quad y_{\text{out}}(5) = 1.81809 + 0.32376 \times 5 = 3.43689
\end{aligned}
$$

RMSE

$$
\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{N}(z_i - \hat{z}_i)^2)}
$$

Keeping in mind all values obtained previously for both models (OLS and Ridge regularization) in the training and test data sets one can compute the following RMSE. We used a python code to compute these values more easily.

$RMSE_{training+OLS} = 2.02650$

$RMSE_{training+Ridge} = 2.15354$

$RMSE_{test+OLS} = 2.46560$

$RMSE_{test+Ridge} = 1.75290$

The results indeed show what we were expecting when performing the Ridge regularization. The regularization, as stated in the previous exercise, acts in the weights so as to keep the model as generalized as possible, avoiding overfitting. Overfitting means high accuracy when using the training data-set and poor accuracy when one explores further observations. The goal is to find a balanced model that fits well our training set but also behaves well when working with new observations.

For the training set, we can check that using the OLS model gives us a smaller Root Mean Squared Error than using the Ridge regularization. However, when we explore a new data-set the RMSE increases for the OLS model, whereas with the Ridge regularization it decreases and has a noticeable smaller RMSE than OLS. The Ridge regularization costs a slightly higher RMSE for the training set in order to keep the model generalized, avoiding noisy observations, to obtain good results for new observations. OLS, in the contrary, keeps the model too dependant on the training set (overfitted), performing poorly and worse when working with new observations.

4.

$$
W^{(1)} = \begin{pmatrix} 0.1 & 0.1 \\ 0.1 & 0.2 \\ 0.2 & 0.1 \end{pmatrix}, \quad b^{(1)} = \begin{pmatrix} 0.1 \\ 0 \\ 0.1 \end{pmatrix}, \quad W^{(2)} = \begin{pmatrix} 1 & 2 & 2 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}
$$

$$b^{(2)} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

Activation function: Softmax $\left(z_c^{(out)}\right) = \frac{e^{z_c^{(out)}}}{\sum_{l=1}^{|c|} e^{z_l^{(out)}}}$

$$\eta = 0.1, \quad x_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Forward propagation:

$$Z^{(1)} = W^{(1)}x_1 + b^{(1)} = \begin{pmatrix} 0.1 & 0.1 \\ 0.1 & 0.2 \\ 0.2 & 0.1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0 \\ 0.1 \end{pmatrix} = \begin{pmatrix} 0.3 \\ 0.3 \\ 0.4 \end{pmatrix}$$

$X^{(1)} = Z^{(1)}$ as there is no activation in the hidden layer.

$$Z^{(2)} = W^{(2)}X^{(1)} + b^{(2)} = \begin{pmatrix} 1 & 2 & 2 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0.3 \\ 0.3 \\ 0.4 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2.7 \\ 2.3 \\ 2.0 \end{pmatrix}$$

$$X^{(2)} = \text{softmax}(Z^{(2)})$$

$$\text{softmax}(z_c^{(out)}) = \frac{e^{z_c^{(out)}}}{\sum_{j=1}^{L} e^{z_j^{(out)}}}$$

$$\sum_{j=1}^{|c|} e^{z_c^{(out)}} = e^{2.7} + e^{2.3} + e^{2.0} = 14.87973 + 9.97418 + 7.38906 = 32.24297$$

$$X^{(2)} = \begin{pmatrix} \frac{e^{2.7}}{32.24297} \\ \frac{e^{2.3}}{32.24297} \\ \frac{e^{2.0}}{32.24297} \end{pmatrix} = \begin{pmatrix} 0.46149 \\ 0.30934 \\ 0.22917 \end{pmatrix}$$

Backpropagation:

$$b_{\text{new}}^{(2)} = b_{\text{old}}^{(2)} - \eta \frac{\partial E}{\partial b^{(2)}}, \quad w_{\text{new}}^{(2)} = w_{\text{old}}^{(2)} - \eta \frac{\partial E}{\partial w^{(2)}}$$

$$\frac{\partial E}{\partial b^{(2)}} = \delta^{(2)}, \quad \frac{\partial E}{\partial w^{(2)}} = \delta^{(2)} X^{(1)T}$$

$$w_{\text{new}}^{(2)} = w_{\text{old}}^{(2)} - \eta \delta^{(2)} X^{(1)T}$$

$$\eta = 0.1$$

For a softmax activation and cross-entropy error:

$$\delta^{(2)} = x^{(2)} - t = \begin{pmatrix} 0.46149 \\ 0.30934 \\ 0.22917 \end{pmatrix} - \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.46149 \\ -0.69066 \\ 0.22917 \end{pmatrix}$$

$$b^{(1)}_{\text{new}} = b^{(1)}_{\text{old}} - \eta \frac{\partial E}{\partial b^{(1)}}, \quad w^{(1)}_{\text{new}} = w^{(1)}_{\text{old}} - \eta \frac{\partial E}{\partial w^{(1)}}$$

$$\frac{\partial E}{\partial b^{(1)}} = \delta^{(1)}, \quad \frac{\partial E}{\partial w^{(1)}} = \delta^{(1)} (x^{(0)})^T$$

$$\delta^{(1)} = \left(\frac{\partial z^{(2)}}{\partial x^{(1)}} \right)^T \delta^{(2)} \circ \frac{\partial x^{(1)}}{\partial z^{(1)}} = (w^{(2)})^T \delta^{(2)} \circ 1$$

$$= \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 1 \\ 2 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0.46149 \\ -0.69066 \\ 0.22917 \end{pmatrix} = \begin{pmatrix} 0 \\ -0.22917 \\ 0.46149 \end{pmatrix}$$

Updates:

Knowing all of these terms, one can compute the new weights after one step of stochastic gradient descent:

$$w^{(1)}_{\text{new}} = w^{(1)}_{\text{old}} - 0.1 \cdot \delta^{(1)} (x^{(0)})^T = \begin{pmatrix} 0.1 & 0.1 \\ 0.12292 & 0.22292 \\ 0.15385 & 0.05385 \end{pmatrix},$$

$$b^{(1)}_{\text{new}} = b^{(1)}_{\text{old}} - 0.1 \cdot \delta^{(1)} = \begin{pmatrix} 0.1 \\ 0.02292 \\ 0.05385 \end{pmatrix}$$

$$w^{(2)}_{\text{new}} = w^{(2)}_{\text{old}} - 0.1 \cdot \delta^{(2)} (x^{(1)})^T = \begin{pmatrix} 0.98616 & 1.98616 & 1.98154 \\ 1.02072 & 2.02072 & 1.02763 \\ 0.99312 & 0.99312 & 0.99083 \end{pmatrix}$$

$$b^{(2)}_{\text{new}} = b^{(2)}_{\text{old}} - 0.1 \cdot \delta^{(2)} = \begin{pmatrix} 0.95385 \\ 1.06907 \\ 0.97708 \end{pmatrix}$$

Part II: Programming

5. Resulting plot

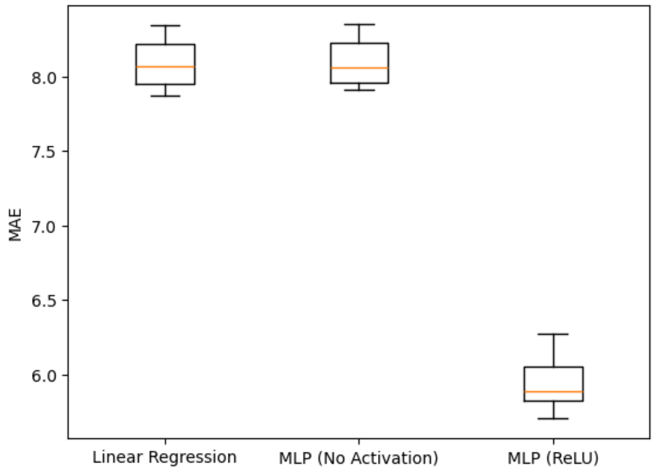

Figure 1: Plot of the Mean Absolute Error for the models indicated.

6. The results from the MLP No Activation and the Linear Regression are very similar, and this is because without an activation function the MLP produces only a Linear Regression. The results that come out from the MLP with no activation starting with a vector X[0] and multiple b[#] and W[#]:

$$\begin{aligned}
Z^{[1]} &= W^{[1]}X^{[0]} + b^{[1]} \\
X^{[1]} &= Z^{[1]} \\
Z^{[2]} &= W^{[2]}X^{[1]} + b^{[2]} \\
X^{[2]} &= Z^{[2]} = W^{[2]}W^{[1]}X^{[0]} + W^{[2]}b^{[1]} + b^{[2]} \\
\textbf{Which if we use } WW &= W^{[2]}X^{[1]} \\
\textbf{and } bb &= W^{[2]}b^{[1]} + b^{[2]} \\
\textbf{becomes } X^{[2]} &= W^{[2]}W^{[1]}X^{[0]} + W^{[2]}b^{[1]} + b^{[2]}
\end{aligned}$$

And this last equation is the same as a linear regression. Activation functions are crucial in making MLPs powerful tools for modeling non-linear relationships in data. Without them, an MLP behaves no differently from a linear model. The boxplots visually support this by showing that an MLP with no activation functions performs similarly to linear regression, having a larger Mean absolute error, whereas an MLP with activation functions outperforms both models, showing a smaller MAE. This happens because MLP's with activation functions enable the study of non-linear features in the data-sets that MLP's with no activation and linear regression do not, as they only capture linear relations.

7. The exercise asks for studying the following case using a 20-80 train-test split random_state = 0

   Grid Search of hyperparameters from the model of MultiLayer Perceptron 2 hidden layers 10 neurons each

- (i) L2 penalty [0.001,0.01,0.1]
- (ii) learning rate [0.001,0.01,0.1]
- (iii) batch size [32,64,128]

After doing that we obtained the following results:

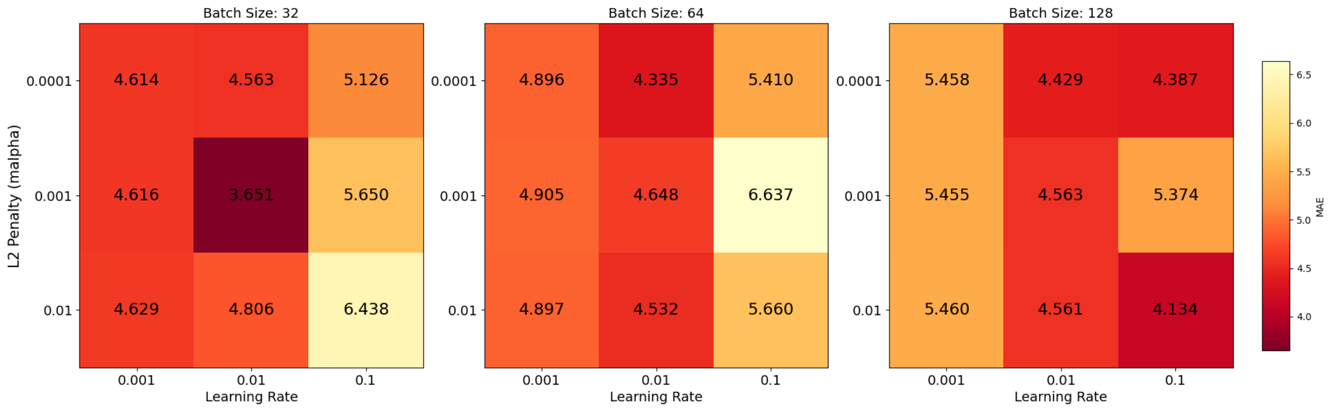

Figure 2: Grid Search of the hyperparameters L2 penalty, learning rate and batch size. Same color scale for all of them.

The parameters changed were the learning rate, the batch size and the L2 penalty. The learning rate defines the rate at which the matrices in the weights and biases in the MLP change according to the results given by the loss function, increasing it too much may lead to overshooting the minimum of the loss function, leading to divergence, while keeping it too small may lead to a slow convergence, in extreme cases this may result in the outcome being a local minima instead of the global minimum. The batch size determines the number of training examples used to train one iteration of the model. Having larger batch sizes can give more stable gradient estimates, but lead to poor generalization. Having a smaller batch size leads to an higher number of iterations which can be computationally intensive, but lead to better results as we can see in the table above.

We can also distinguish batch size from epoch, which is another hyperparameter, that defines the number of times that we have run the model over all the training data, in order to obtain more valid results we could use more epochs in order to better fit the model to the outcomes desired, although we should be aware of the computationally intensive task that it may lead to. The L2 Penalty is also known as weight decay or ridge regularization, it is a regularization technique that adds a term to the loss function that penalizes large weigths using, for example, the modulus of the weights, to determine the penalization, this leads to reducing overfitting. Increasing L2 penalty leads to simpler models, reduces overfitting, but may increase bias, on the other side decreasing L2 penalty allows for more complex models potentially leading to overfitting.

In our grid search the best combination was the batch size of 32, an L2 Penalty of 0.001 and a learning rate of 0.01 resulting in a MAE of 3.651.

Part III: Extra Simulated Annealing after grid search

In hyperparameter tuning, grid search is a commonly used method where every combination of a predefined set of hyperparameters is systematically tested to identify the best model configuration. While this method is thorough, it has limitations. Since it only explores values within the specified grid, it may miss better combinations that lie between those grid points. To address this, one could use a random search to explore a broader range of values, however, more advanced optimization techniques, such as simulated annealing, offer a more sophisticated approach.

Simulated annealing is inspired by the process of crystal formation in physics, where a material is heated and then slowly cooled to reduce defects. In this algorithm, the ”temperature” controls how much the hyperparameters can change. At higher temperatures, the algorithm explores the search space more broadly, allowing for larger changes in hyperparameters. As the temperature decreases, the changes become smaller and more refined. One of the key features of simulated annealing is that it occasionally accepts worse solutions early in the process to escape local minima, increasing the chances of finding a global minimum as the temperature cools.

For this task, we combined both techniques. We first performed a broad grid search to identify promising hyperparameters and then used simulated annealing to fine-tune them further. Starting with the best hyperparameters from the grid search, we ran 100 iterations of simulated annealing, with an initial temperature of 8.0 and a cooling rate of 0.95. This approach allowed us to achieve a lower mean absolute error (MAE) of 3.6166 with the final hyperparameters: ’L2 penalty / alpha’: 0.00292, ’learning_rate_init’: 0.0101, and ’batch_size’: 32.

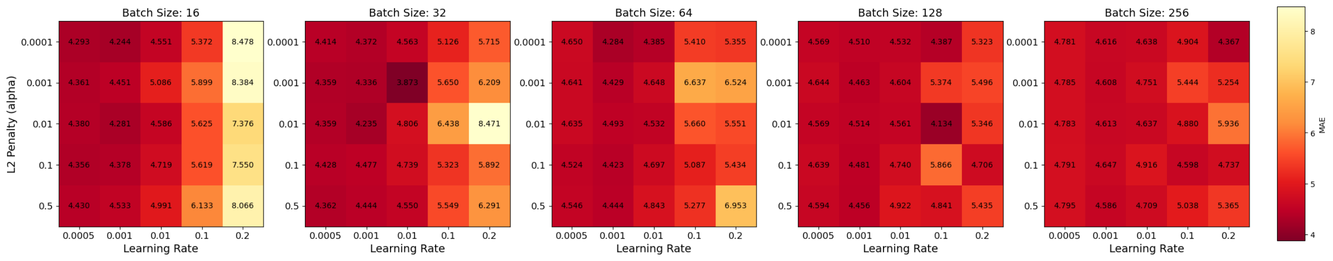

Figure 3: Broader grid Search of the hyperparameters L2 penalty, learning rate and batch size. Same color scale for all of them.

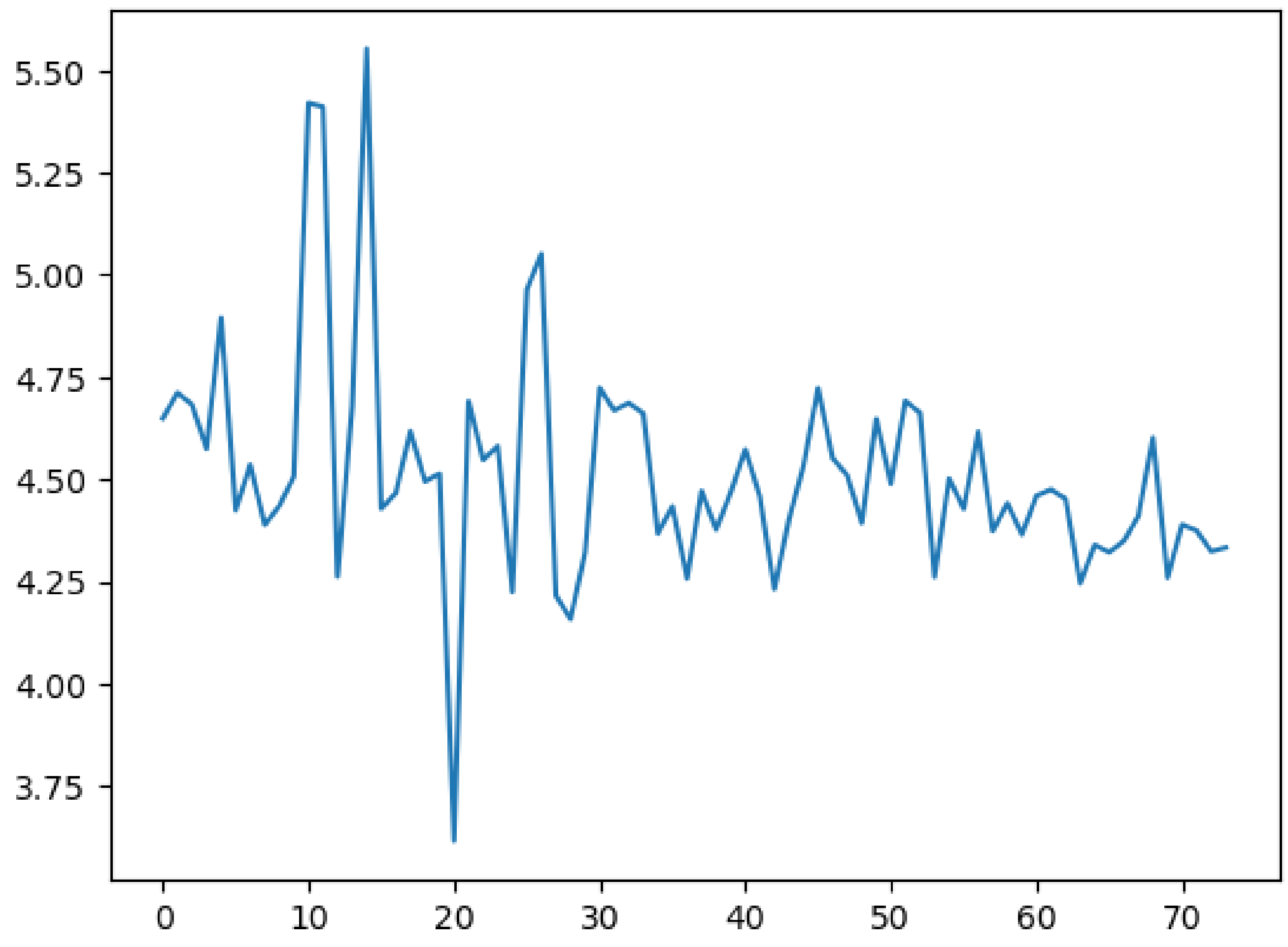

Figure 4: Graph showing the evolution of the MAE as the iterations of the Simulated Annealing go through.

A Code

A.1 Python Script

```
# %% [markdown]
# # import libraries

# %%
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib
import matplotlib.pyplot as plt
import scipy
from scipy import stats
import itertools
```

```python
# Sklearn
from sklearn.model_selection import train_test_split, \
                                    GridSearchCV, cross_val_score
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import accuracy_score, mean_absolute_error
from sklearn.linear_model import LinearRegression
from sklearn.neural_network import MLPRegressor

# %% [markdown]
# # Read data

# %%
park_data = pd.read_csv("parkinsons.csv")

park_inp = park_data.drop("target", axis=1)
park_tar = park_data["target"]

# %% [markdown]
# # Basic data visualisation and scale analysis

# %% [markdown]
# ### Target visualisation for curiosity
#
# It's interesting to note that the scale varies from 0 to 260.
# Although as we will see in the target visualisation we only
# have scores below

# %%
plt.hist(park_tar)
#plt.xlim([0,260])

# %%
# Assuming park_data is already loaded

# Calculate min, max, and difference for each column
min_values = park_data.min()
max_values = park_data.max()
diff_values = max_values - min_values

# Create a DataFrame with min, max, and difference
summary_df = pd.DataFrame({
    'Min': min_values,
    'Max': max_values,
    'Difference': diff_values
})

# Display the summary
```

```
print("Min, Max, and Difference values for each column:")
print(summary_df)

# Create histograms for all columns
fig, axes = plt.subplots(5, 4, figsize=(20, 25))
fig.suptitle("Histograms of all features", fontsize=16)

for i, column in enumerate(park_data.columns):
    row = i // 4
    col = i % 4
    sns.histplot(park_data[column], ax=axes[row, col], kde=True)
    axes[row, col].set_title(column)
    axes[row, col].set_xlabel('')

plt.tight_layout()
plt.subplots_adjust(top=0.95)
plt.show()

# %% [markdown]
# # 5) Model comparison
# - Linear regression Model (LR)
# - MLP Regressor with 2 hidden layers of 10 neurons each
#           + no activation functions (MLP_NA)
# - MLP Regressor with 2 hidden layers of 10 neurons each
#           + ReLU activation functions (MLP_ReLU)
#
#
#
# MLP random_state=0
# Boxplot of Mean Absolute Error (MAE) for each model

# %% [markdown]
# ### Data splitting

# %%
rand_st = list(range(1,11))

train_test_data = []
for rand in rand_st:
    inp_train_temp, inp_test_temp, tar_train_temp, tar_test_temp =\
    train_test_split(park_inp,park_tar,test_size=0.2,random_state=rand)
    train_test_data.append(
        [inp_train_temp, inp_test_temp, tar_train_temp, tar_test_temp])

# %% [markdown]
# Model Definition
# %%
# Linear Regression Model
linear_model = LinearRegression()

# MLP Regressor with 2 hidden layers, no activation
mlp_no_activation = MLPRegressor(hidden_layer_sizes=(10, 10),
                                 activation='identity',
                                 random_state=0)

# MLP Regressor with 2 hidden layers, ReLU activation
mlp_relu = MLPRegressor(hidden_layer_sizes=(10, 10),
                        activation='relu',
                        random_state=0)

# List of models
models = [linear_model, mlp_no_activation, mlp_relu]
model_names = ['Linear Regression', 'MLP (No Activation)', 'MLP (ReLU)']

# %%
def evaluate_model(model, X_train, y_train, X_test):
    # Train the model
    model.fit(X_train, y_train)

    # Make predictions on the test set
    return model.predict(X_test)

# Perform cross-validation and calculate MAE for each model
mae_scores = []

for model in models:
    mae_score_temp = []
    for run in range(10):
        y_pred = evaluate_model(model,
                                train_test_data[run][0],
                                train_test_data[run][2],
                                train_test_data[run][1])
        mae = mean_absolute_error(train_test_data[run][3], y_pred)
        mae_score_temp.append(mae)
        # Negate because sklearn returns negative MAE
    mae_scores.append(mae_score_temp)

# %% [markdown]
# Plot the results from the training.

# %%
plt.boxplot(mae_scores, labels=model_names)
plt.ylabel("MAE")
```

```
# %% [markdown]
# # 6)
# The results from the MLP_NA and the Linear Regression
# are very similar, this is because without an activation
# function the MLP produces only a Linear Regression.
# From We can think of the Results that come up from the MLP Like
#
#
# Z^{[1]} = W^{[1]}X^{[0]} + b^{[1]}
#
# X^{[1]} = Z^{[1]}
#
# Z^{[2]} = W^{[2]}X^{[1]} + b^{[2]}
#
# X^{[2]} = Z^{[2]} = W^{[2]}W^{[1]}X^{[0]} + W^{[2]}b^{[1]} + b^{[2]}
#
# Where the left half could also be represented as
# W and the right half a general constant

# %% [markdown]
# # 7.
# 20-80 train-test split random_state = 0
#
# Grid Search of hyperparameters from the model
# of MultiLayer Perceptron 2 hidden layers 10 neurons each
# - (i)   L2 penalty    [0.001,0.01,0.1]
# - (ii)  learning rate [0.001,0.01,0.1]
# - (iii) batch size    [32,64,128]

# %% [markdown]
# # 7)  Normal version

# %%
# Split the data
X_train_notscaled, X_test_notscaled, y_train_notscaled, y_test_notscaled =\
        train_test_split(park_inp,park_tar,test_size=0.2,random_state=0)



# Define the model and parameter grid
param_grid_notscaled = {
    'alpha': [0.0001, 0.001, 0.01],
    'learning_rate_init': [0.001, 0.01, 0.1],
    'batch_size': [32, 64, 128]
}
# Generate all combinations of hyperparameters
```

```python
param_combinations = list(itertools.product(*param_grid_notscaled.values()))


# Initialize matrix to store results
mae_matrix_notscaled = np.zeros((3, 3, 3))

# Iterate through all combinations
for i, (alpha, learning_rate, batch_size) in enumerate(param_combinations):
    # Create and train the model
    model = MLPRegressor(hidden_layer_sizes=(10, 10),
                         random_state=0,alpha=alpha,
                         learning_rate_init=learning_rate, batch_size=batch_size)
    model.fit(X_train_notscaled, y_train_notscaled)

    # Make predictions on the test set
    y_pred = model.predict(X_test_notscaled)

    # Calculate MAE
    mae = mean_absolute_error(y_test_notscaled, y_pred)

    # Store MAE in the matrix
    mae_matrix_notscaled[
        param_grid_notscaled['alpha'].index(alpha),
        param_grid_notscaled['learning_rate_init'].index(learning_rate),
        param_grid_notscaled['batch_size'].index(batch_size)] = mae



# %%
# Find the best combination
best_idx = np.unravel_index(np.argmin(mae_matrix_notscaled),
                            mae_matrix_notscaled.shape)
best_params_notscaled = {
  'alpha': param_grid_notscaled['alpha'][best_idx[0]],
  'learning_rate_init': param_grid_notscaled['learning_rate_init'][best_idx[1]],
  'batch_size': param_grid_notscaled['batch_size'][best_idx[2]]
}
best_score_notscaled = mae_matrix_notscaled[best_idx]

# Plot the results
fig_notscaled, axes_notscaled = plt.subplots(1, 3, figsize=(20, 6))
batch_sizes_notscaled = [32, 64, 128]
# Find global min and max for consistent color scaling
value_min_notscaled = np.min(mae_matrix_notscaled)
value_max_notscaled = np.max(mae_matrix_notscaled)

for i, batch_size in enumerate(batch_sizes_notscaled):
    ax = axes_notscaled[i]
    im = ax.imshow(mae_matrix_notscaled[:, :, i], cmap='YlOrRd_r',
                   vmin=value_min_notscaled, vmax=value_max_notscaled)
    ax.set_xticks(np.arange(3))
    ax.set_yticks(np.arange(3))
    ax.set_xticklabels([0.001, 0.01, 0.1], fontsize=14)
    ax.set_yticklabels([0.0001, 0.001, 0.01], fontsize=14)
    ax.set_xlabel('Learning Rate', fontsize=14)
    if i == 0: ax.set_ylabel('L2 Penalty (malpha)', fontsize=15)
    ax.set_title(f'Batch Size: {batch_size}', fontsize=14)

    for j in range(3):
        for k in range(3):
            text = ax.text(k, j, f'{mae_matrix_notscaled[j, k, i]:.3f}',
                           ha="center", va="center", color="black",
                           fontsize=17)

plt.tight_layout()
fig_notscaled.subplots_adjust(right=0.9)

# Add a colorbar to the right of the subplots
cbar_ax = fig_notscaled.add_axes([0.92, 0.15, 0.02, 0.7])
fig_notscaled.colorbar(im, cax=cbar_ax, label='MAE')

plt.show()

print("Best parameters:", best_params_notscaled)
print("Best test MAE:", best_score_notscaled)
```

A Extra Code - Simulated annealing

A.1 Python Script

```
# %% [markdown]
# # import libraries

# %%
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib
import matplotlib.pyplot as plt
import scipy
from scipy import stats
```

```python
import itertools
import random
import time
import math

# Sklearn
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import accuracy_score, mean_absolute_error
from sklearn.linear_model import LinearRegression
from sklearn.neural_network import MLPRegressor

# %% [markdown]
# # Read data

# %%
park_data = pd.read_csv("parkinsons.csv")

park_inp = park_data.drop("target", axis=1)
park_tar = park_data["target"]

# %% [markdown]
# # Basic data visualisation and scale analysis

# %% [markdown]
# ### Target visualisation for curiosity
#
# It's interesting to note that the scale varies from 0 to 260. Although as we will see i

# %%
plt.hist(park_tar)
#plt.xlim([0,260])

# %%
# Assuming park_data is already loaded

# Calculate min, max, and difference for each column
min_values = park_data.min()
max_values = park_data.max()
diff_values = max_values - min_values

# Create a DataFrame with min, max, and difference
summary_df = pd.DataFrame({
    'Min': min_values,
    'Max': max_values,
    'Difference': diff_values
})
```

```
# Display the summary
print("Min, Max, and Difference values for each column:")
print(summary_df)

# Create histograms for all columns
fig, axes = plt.subplots(5, 4, figsize=(20, 25))
fig.suptitle("Histograms of all features", fontsize=16)

for i, column in enumerate(park_data.columns):
    row = i // 4
    col = i % 4
    sns.histplot(park_data[column], ax=axes[row, col], kde=True)
    axes[row, col].set_title(column)
    axes[row, col].set_xlabel('')

plt.tight_layout()
plt.subplots_adjust(top=0.95)
plt.show()

# %% [markdown]
# # 5) Model comparison
# - Linear regression Model (LR)
# - MLP Regressor with 2 hidden layers of 10 neurons each + no activation functions (MLP_
# - MLP Regressor with 2 hidden layers of 10 neurons each + ReLU activation functions (MI
#
#
#
# MLP random_state=0
# Boxplot of Mean Absolute Error (MAE) for each model

# %% [markdown]
# ### Data splitting

# %%
rand_st = list(range(1,11))

train_test_data = []
for rand in rand_st:
    inp_train_temp, inp_test_temp, tar_train_temp, tar_test_temp =\
    train_test_split(park_inp,park_tar,test_size=0.2,random_state=rand)
    train_test_data.append(
        [inp_train_temp, inp_test_temp, tar_train_temp, tar_test_temp])

# %% [markdown]
# Model Definition
```

```python
# %%
# Linear Regression Model
linear_model = LinearRegression()

# MLP Regressor with 2 hidden layers, no activation
mlp_no_activation = MLPRegressor(hidden_layer_sizes=(10, 10),
                                 activation='identity',
                                 random_state=0)

# MLP Regressor with 2 hidden layers, ReLU activation
mlp_relu = MLPRegressor(hidden_layer_sizes=(10, 10),
                        activation='relu',
                        random_state=0)

# List of models
models = [linear_model, mlp_no_activation, mlp_relu]
model_names = ['Linear Regression', 'MLP (No Activation)', 'MLP (ReLU)']

# %%
def evaluate_model(model, X_train, y_train, X_test):
    # Train the model
    model.fit(X_train, y_train)

    # Make predictions on the test set
    return model.predict(X_test)

# Perform cross-validation and calculate MAE for each model
mae_scores = []

for model in models:
    mae_score_temp = []
    for run in range(10):
        y_pred = evaluate_model(model, train_test_data[run][0], train_test_data[run][2],
                               train_test_data[run][1])
        mae = mean_absolute_error(train_test_data[run][3], y_pred)
        mae_score_temp.append(mae)  # Negate because sklearn returns negative MAE
    mae_scores.append(mae_score_temp)

# %% [markdown]
# Plot the results from the training.

# %%
plt.boxplot(mae_scores, labels=model_names)
plt.ylabel("MAE")

# %% [markdown]
# # 6)
# The results from the MLP_NA and the Linear Regression are very similar, this is becaus
# From We can think of the Results that come up from the MLP Like
#
#
# Z^{[1]} = W^{[1]}X^{[0]} + b^{[1]}
#
# X^{[1]} = Z^{[1]}
#
# Z^{[2]} = W^{[2]}X^{[1]} + b^{[2]}
#
# X^{[2]} = Z^{[2]} = W^{[2]}W^{[1]}X^{[0]} + W^{[2]}b^{[1]} + b^{[2]}
#
# Where the left half could also be represented as W and the right half a general consta

# %% [markdown]
# # 7.
# 20-80 train-test split random_state = 0
#
# Grid Search of hyperparameters from the model of MultiLayer Perceptron 2 hidden layers
# - (i)   L2 penalty    [0.001,0.01,0.1]
# - (ii)  learning rate [0.001,0.01,0.1]
# - (iii) batch size    [32,64,128]

# %% [markdown]
# # 7)  Normal version

# %%
# Split the data
X_train_notscaled, X_test_notscaled, y_train_notscaled, y_test_notscaled =\
        train_test_split(park_inp,park_tar,test_size=0.2,random_state=0)



# Define the model and parameter grid
mlp_notscaled = MLPRegressor(hidden_layer_sizes=(10, 10),
                   max_iter=1000, random_state=0)
param_grid_notscaled = {
    'alpha': [0.0001, 0.001, 0.01],
    'learning_rate_init': [0.001, 0.01, 0.1],
    'batch_size': [32, 64, 128]
}
# Generate all combinations of hyperparameters
param_combinations = list(itertools.product(*param_grid_notscaled.values()))


# Initialize matrix to store results
mae_matrix_notscaled = np.zeros((3, 3, 3))
```

```
# Iterate through all combinations
for i, (alpha, learning_rate, batch_size) in enumerate(param_combinations):
    # Create and train the model
    model = MLPRegressor(hidden_layer_sizes=(10, 10),
                         random_state=0,alpha=alpha,
                         learning_rate_init=learning_rate, batch_size=batch_size)
    model.fit(X_train_notscaled, y_train_notscaled)

    # Make predictions on the test set
    y_pred = model.predict(X_test_notscaled)

    # Calculate MAE
    mae = mean_absolute_error(y_test_notscaled, y_pred)

    # Store MAE in the matrix
    mae_matrix_notscaled[
                param_grid_notscaled['alpha'].index(alpha),
                param_grid_notscaled['learning_rate_init'].index(learning_rate),
                param_grid_notscaled['batch_size'].index(batch_size)] = mae



# %%
# Find the best combination
best_idx = np.unravel_index(np.argmin(mae_matrix_notscaled), mae_matrix_notscaled.shape)
best_params_notscaled = {
    'alpha': param_grid_notscaled['alpha'][best_idx[0]],
    'learning_rate_init': param_grid_notscaled['learning_rate_init'][best_idx[1]],
    'batch_size': param_grid_notscaled['batch_size'][best_idx[2]]
}
best_score_notscaled = mae_matrix_notscaled[best_idx]

# Plot the results
fig_notscaled, axes_notscaled = plt.subplots(1, 3, figsize=(20, 6))
batch_sizes_notscaled = [32, 64, 128]
# Find global min and max for consistent color scaling
value_min_notscaled = np.min(mae_matrix_notscaled)
value_max_notscaled = np.max(mae_matrix_notscaled)

for i, batch_size in enumerate(batch_sizes_notscaled):
    ax = axes_notscaled[i]
    im = ax.imshow(mae_matrix_notscaled[:, :, i], cmap='YlOrRd_r',
                   vmin=value_min_notscaled, vmax=value_max_notscaled)
    ax.set_xticks(np.arange(3))
    ax.set_yticks(np.arange(3))
    ax.set_xticklabels([0.001, 0.01, 0.1], fontsize=14)
    ax.set_yticklabels([0.0001, 0.001, 0.01], fontsize=14)
    ax.set_xlabel('Learning Rate', fontsize=14)
    if i == 0: ax.set_ylabel('L2 Penalty (malpha)', fontsize=15)
    ax.set_title(f'Batch Size: {batch_size}', fontsize=14)

    for j in range(3):
        for k in range(3):
            text = ax.text(k, j, f'{mae_matrix_notscaled[j, k, i]:.3f}',
                           ha="center", va="center", color="black",
                           fontsize=17)

plt.tight_layout()
fig_notscaled.subplots_adjust(right=0.9)

# Add a colorbar to the right of the subplots
cbar_ax = fig_notscaled.add_axes([0.92, 0.15, 0.02, 0.7])
fig_notscaled.colorbar(im, cax=cbar_ax, label='MAE')
plt.show()

print("Best parameters:", best_params_notscaled)
print("Best test MAE:", best_score_notscaled)

# %%

# Split the data
X_train, X_test, y_train, y_test = train_test_split(park_inp, park_tar, test_size=0.2, ra

# Define the model and parameter grid for Grid Search
param_grid = {
    'alpha': [0.0001, 0.001, 0.01, 0.1, 0.5],
    'learning_rate_init': [0.0005, 0.001, 0.01, 0.1, 0.2],
    'batch_size': [16,32, 64, 128, 256]
}
param_combinations = list(itertools.product(*param_grid.values()))

# Initialize matrix to store results
mae_matrix = np.zeros((5, 5, 5))

# Grid Search to find the best parameters
best_mae_gs = float('inf')
best_params_gs = None
for alpha, learning_rate, batch_size in param_combinations:
    model = MLPRegressor(hidden_layer_sizes=(10, 10), max_iter=1000, random_state=0,
                         alpha=alpha, learning_rate_init=learning_rate, batch_size=batch_
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    mae = mean_absolute_error(y_test, y_pred)
    idx_alpha = param_grid['alpha'].index(alpha)
    idx_lr = param_grid['learning_rate_init'].index(learning_rate)
    idx_batch = param_grid['batch_size'].index(batch_size)
    mae_matrix[idx_alpha, idx_lr, idx_batch] = mae

    if mae < best_mae_gs:
        best_mae_gs = mae
        best_params_gs = {'alpha': alpha, 'learning_rate_init': learning_rate, 'batch_siz
# Plot the results
fig, axes = plt.subplots(1, 5, figsize=(25, 6))
batch_sizes = [16,32, 64, 128, 256]
value_min = np.min(mae_matrix)
value_max = np.max(mae_matrix)

for i, batch_size in enumerate(batch_sizes):
    ax = axes[i]
    im = ax.imshow(mae_matrix[:, :, i], cmap='YlOrRd_r', vmin=value_min, vmax=value_max)
    ax.set_xticks(np.arange(5))
    ax.set_yticks(np.arange(5))
    ax.set_xticklabels([0.0005, 0.001, 0.01, 0.1, 0.2], fontsize=12)
    ax.set_yticklabels([0.0001, 0.001, 0.01, 0.1, 0.5], fontsize=12)
    ax.set_xlabel('Learning Rate', fontsize=14)
    if i == 0: ax.set_ylabel('L2 Penalty (alpha)', fontsize=14)
    ax.set_title(f'Batch Size: {batch_size}', fontsize=14)

    for j in range(5):
        for k in range(5):
            ax.text(k, j, f'{mae_matrix[j, k, i]:.3f}', ha="center", va="center", color='

plt.tight_layout()
fig.subplots_adjust(right=0.9)
cbar_ax = fig.add_axes([0.92, 0.15, 0.02, 0.7])
fig.colorbar(im, cax=cbar_ax, label='MAE')
plt.show()

# %%
# Use the best parameters from Grid Search to initialize Simulated Annealing
initial_params = best_params_gs
param_bounds = {'alpha': [0.0001, 0.5], 'learning_rate_init': [0.001, 0.1], 'batch_size':

random.seed(1)

# Simulated Annealing settings
n_iterations = 100
temperature = 8.0
cooling_rate = 0.95
```

```
def smooth_bound(x, lower, upper):
    """
    A function that smoothly approaches the bounds using a sigmoid-like function.
    It behaves linearly in the middle but smoothly saturates towards the bounds.
    """
    range_span = upper - lower
    return lower + (range_span / (1 + np.exp(-0.05 * (x - lower - 0.5 * range_span))))

# Helper function to generate a neighbour solution
def generate_neighbour(params, param_bounds,temperature):
    new_params = params.copy()

    # Generate new alpha value in log scale
    log_alpha = math.log10(params['alpha'])
    log_alpha_perturb = random.gauss(log_alpha,temperature/10)  # Adjust the stddev for p
    new_alpha = 10**log_alpha_perturb
    if param_bounds['alpha'][0] <= new_alpha <= param_bounds['alpha'][1]:
        new_params['alpha'] = new_alpha

    # Generate new learning_rate_init value in log scale
    log_learning_rate = math.log10(params['learning_rate_init'])
    log_learning_rate_perturb = random.gauss(log_learning_rate,temperature/10)  # Adjust
    new_learning_rate = 10**log_learning_rate_perturb
    if param_bounds['learning_rate_init'][0] <= new_learning_rate <= param_bounds['learni
        new_params['learning_rate_init'] = new_learning_rate

    return new_params

# Helper function to calculate acceptance probability
def acceptance_probability(old_mae, new_mae, temperature):
    if new_mae < old_mae:
        return 1.0
    else:
        acceptance_probability = np.exp((old_mae - new_mae) / temperature)
        print(acceptance_probability)
        return acceptance_probability

# Initialise with the best parameters from Grid Search
current_params = initial_params
model = MLPRegressor(hidden_layer_sizes=(10, 10), max_iter=1000, random_state=0,
                     alpha=current_params['alpha'],
                     learning_rate_init=current_params['learning_rate_init'],
                     batch_size=current_params['batch_size'])
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
current_mae = mean_absolute_error(y_test, y_pred)
```

```python
best_params = current_params
best_mae = current_mae

start_mae = 10
# Simulated Annealing loop
best_mae = start_mae
new_mae_list, alpha_list, learning_list, batch_list = [],[],[],[]

for iteration in range(n_iterations):
    if (iteration + 1) % 50 == 0:
        print("Best parameters found:", best_params)
        print("Best MAE:", best_mae)
        time.sleep(30)
    new_params = generate_neighbour(current_params,param_bounds,temperature)

    model = MLPRegressor(hidden_layer_sizes=(10, 10), max_iter=1000, random_state=0,
                         alpha=new_params['alpha'],
                         learning_rate_init=new_params['learning_rate_init'],
                         batch_size=32)
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    new_mae = mean_absolute_error(y_test, y_pred)

    print(f"Iteration {iteration + 1}: New  MAE = {new_mae:.4f} with params ['alpha': {ne
    print("current_mae = ", current_mae)
    print("new_mae = ", new_mae)
    if acceptance_probability(current_mae, new_mae, temperature) > random.random():
        current_params = new_params
        current_mae    = new_mae
        new_mae_list.append(new_mae)
        alpha_list.append(new_params['alpha'])
        learning_list.append(new_params['learning_rate_init'])
        batch_list.append(new_params['batch_size'])

    if new_mae < best_mae:
        best_params = new_params
        best_mae = new_mae

    temperature *= cooling_rate

    print(f"Iteration {iteration + 1}: Best MAE = {best_mae:.4f} with params {best_params

plt.figure()
plt.plot(new_mae_list)
plt.figure()
plt.plot(alpha_list)
plt.figure()
plt.plot(learning_list)
#plt.plot(batch_list)


# %%
plt.figure(75)
plt.plot(new_mae_list)
plt.figure(76)
plt.plot(alpha_list)
plt.figure(77)
plt.plot(learning_list)
plt.figure(78)
plt.plot(batch_list)


# %%
import re
import matplotlib.pyplot as plt

def parse_log_file(filename):
    # Regular expression pattern to match the desired lines
    pattern = r"Iteration (\d+): New. MAE = (\d.\d+) with params \['alpha': (\d.\d+), 'le
#Iteration 500: New  MAE = 5.3479 with params ['alpha': 0.4375, 'learning_rate_init': 0.042
    # List to store parsed results
    results = []

    # Open the log file and read line by line
    with open(filename, 'r') as file:
        for line in file:
            # Search for the pattern in the current line
            match = re.search(pattern, line)
            if match:
                iteration = int(match.group(1))
                mae = float(match.group(2))
                alpha = float(match.group(3))
                learning_rate_init = float(match.group(4))
                batch_size = float(match.group(5))

                # Append the parsed values to the results list
                results.append({
                    'iteration': iteration,
                    'mae': mae,
                    'alpha': alpha,
                    'learning_rate_init': learning_rate_init,
                    'batch_size': batch_size
                })
    return results

def plot_results(parsed_results):
    # Extract values for plotting
    iterations = [result['iteration'] for result in parsed_results]
    mae_values = [result['mae'] for result in parsed_results]
    alpha_values = [result['alpha'] for result in parsed_results]
    learning_rate_values = [result['learning_rate_init'] for result in parsed_results]
    batch_size_values = [result['batch_size'] for result in parsed_results]

    # Plot MAE over iterations
    plt.figure(figsize=(10, 6))
    plt.plot(mae_values, label='MAE', marker='o')
    plt.xlabel('Iteration')
    plt.ylabel('Mean Absolute Error (MAE)')
    plt.title('MAE over Iterations')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Plot Alpha over iterations
    plt.figure(figsize=(10, 6))
    plt.plot(alpha_values, label='Alpha', marker='o', color='r')
    plt.xlabel('Iteration')
    plt.ylabel('Alpha')
    plt.title('Alpha over Iterations')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Plot Learning Rate Init over iterations
    plt.figure(figsize=(10, 6))
    plt.plot(learning_rate_values, label='Learning Rate Init', marker='o', color='g')
    plt.xlabel('Iteration')
    plt.ylabel('Learning Rate Init')
    plt.title('Learning Rate Init over Iterations')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Plot Batch Size over iterations
    plt.figure(figsize=(10, 6))
    plt.plot(batch_size_values, label='Batch Size', marker='o', color='m')
    plt.xlabel('Iteration')
    plt.ylabel('Batch Size')
    plt.title('Batch Size over Iterations')
    plt.legend()
    plt.grid(True)
    plt.show()

def main():
    # Parse the log file
    filename = "output.txt"  # Replace with your log file name
    parsed_results = parse_log_file(filename)

    # Display the parsed results
    for result in parsed_results:
        print(f"Iteration: {result['iteration']}, MAE: {result['mae']}, Alpha: {result['a

    # Plot the parsed results
    plot_results(parsed_results)

if __name__ == "__main__":
    main()


# %%
current_mae_test = 5
for i in np.arange(5,10):
    acceptance_probability(current_mae, i , 1)

print(list(np.arange(5,10)))

# %% [markdown]
# # 7) Scaled version

# %%
# Split the data
X_train_scaled, X_test_scaled, y_train_scaled, y_test_scaled = train_test_split(park_inp

# Scale the features
scaler_scaled = StandardScaler()
X_train_scaled = scaler_scaled.fit_transform(X_train_scaled)
X_test_scaled = scaler_scaled.fit_transform(X_test_scaled)

# Define the model and parameter grid
mlp_scaled = MLPRegressor(hidden_layer_sizes=(10, 10),
                   max_iter=1000, random_state=0)
param_grid_scaled = {
    'alpha': [0.0001, 0.001, 0.01],
    'learning_rate_init': [0.001, 0.01, 0.1],
    'batch_size': [32, 64, 128]
}
```

```python
# Generate all combinations of hyperparameters
param_combinations = list(itertools.product(*param_grid_scaled.values()))

# Initialize matrix to store results
mae_matrix_scaled = np.zeros((3, 3, 3))

# Iterate through all combinations
for i, (alpha, learning_rate, batch_size) in enumerate(param_combinations):
    # Create and train the model
    model = MLPRegressor(hidden_layer_sizes=(10, 10),
                         random_state=0,alpha=alpha,
                         learning_rate_init=learning_rate, batch_size=batch_size)
    model.fit(X_train_scaled, y_train_scaled)

    # Make predictions on the test set
    y_pred = model.predict(X_test_scaled)

    # Calculate MAE
    mae = mean_absolute_error(y_test_scaled, y_pred)

    # Store MAE in the matrix
    mae_matrix_scaled[
                param_grid_scaled['alpha'].index(alpha),
                param_grid_scaled['learning_rate_init'].index(learning_rate),
                param_grid_scaled['batch_size'].index(batch_size)] = mae

# %% [markdown]
# Plot the results and print the outcomes

# %%
# Plot the results
fig_scaled, axes_scaled = plt.subplots(1, 3, figsize=(20, 6))
batch_sizes_scaled = [32, 64, 128]
# Find global min and max for consistent color scaling
value_min_scaled = np.min(mae_matrix_scaled)
value_max_scaled = np.max(mae_matrix_scaled)

for i, batch_size in enumerate(batch_sizes_scaled):
    ax = axes_scaled[i]
    im = ax.imshow(mae_matrix_scaled[:, :, i], cmap='YlOrRd_r',
                   vmin=value_min_scaled, vmax=value_max_scaled)
    ax.set_xticks(np.arange(3))
    ax.set_yticks(np.arange(3))
    ax.set_xticklabels([0.001, 0.01, 0.1], fontsize=14)
    ax.set_yticklabels([0.0001, 0.001, 0.01], fontsize=14)
    ax.set_xlabel('Learning Rate', fontsize=14)
    if i == 0: ax.set_ylabel('L2 Penalty (alpha)', fontsize=15)
    ax.set_title(f'Batch Size: {batch_size}', fontsize=14)

    for j in range(3):
        for k in range(3):
            text = ax.text(k, j, f'{mae_matrix_scaled[j, k, i]:.3f}',
                           ha="center", va="center", color="black",
                           fontsize=17)

plt.tight_layout()
fig_scaled.subplots_adjust(right=0.9)

# Add a colorbar to the right of the subplots
cbar_ax = fig_scaled.add_axes([0.92, 0.15, 0.02, 0.7])
fig_scaled.colorbar(im, cax=cbar_ax, label='MAE')
plt.show()

# Find the best combination
best_idx = np.unravel_index(np.argmin(mae_matrix_scaled), mae_matrix_scaled.shape)
best_params_scaled = {
    'alpha': param_grid_scaled['alpha'][best_idx[0]],
    'learning_rate_init': param_grid_scaled['learning_rate_init'][best_idx[1]],
    'batch_size': param_grid_scaled['batch_size'][best_idx[2]]
}
best_score_scaled = mae_matrix_scaled[best_idx]

print("Best parameters:", best_params_scaled)
print("Best test MAE:", best_score_scaled)

# %% [markdown]
# # Test the statistical significance of scaling the inputs beforehand

# %%
# Flatten the mae matrices
mae_scaled = mae_matrix_scaled.flatten()
mae_notscaled = mae_matrix_notscaled.flatten()

# Perform paired t-test
t_statistic, p_value = stats.ttest_rel(mae_scaled, mae_notscaled)

print(f"Paired t-test results:")
print(f"t-statistic: {t_statistic}")
print(f"p-value: {p_value}")

# Calculate mean difference
mean_difference = np.mean(mae_notscaled - mae_scaled)
print(f"\nMean difference (Not Scaled - Scaled): {mean_difference}")
```

```
# Calculate percentage of cases where scaled version performs better
scaled_better = np.sum(mae_scaled < mae_notscaled) / len(mae_scaled) * 100
print(f"Percentage of cases where scaled version performs better: {scaled_better:.2f}%")

# Visualize the differences
plt.figure(figsize=(10, 6))
plt.scatter(range(len(mae_scaled)), mae_notscaled - mae_scaled, alpha=0.5)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel("Hyperparameter combination")
plt.ylabel("Difference in MAE (Not Scaled - Scaled)")
plt.title("Difference in MAE between Not Scaled and Scaled versions")
plt.show()
```