Aprendizagem 2022

Homework III – Group 019

Diogo Gaspar 99207, Rafael Oliveira 99311

Part I: Pen and paper

1. **Consider the basis function, $\phi_j(x) = x^j$, for performing a 3-order polynomial regression,**

$$\hat{z}(x, w) = \sum_{j=0}^{3} w_j \phi_j(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^3.$$

Learn the Ridge regression (l_2 regularization) on the transformed data space using the closed-form solution with $\lambda = 2$.

We have in hands a **supervised learning** problem, with a given training dataset as shown below:

	y_1	z
x_1	0.8	24
x_2	1	20
x_3	1.2	10
x_4	1.4	13
x_5	1.6	12

Table 1: Training dataset: y_1 as the input's (only) variable, z as the target variable

We can note that in the statement's estimation function, $\hat{z}(x, w)$, x is a single-element vector (with its only entry being each sample's y_1 value). Therefore, it makes sense to "expand" the table above as follows, in order to have a broader representation of the values we'll be using in the estimation function:

	y_1	y_1^2	y_1^3	z
x_1	0.8	0.64	0.512	24
x_2	1	1	1	20
x_3	1.2	1.44	1.728	10
x_4	1.4	1.96	2.744	13
x_5	1.6	2.56	4.096	12

Table 2: Training dataset with additional information

The equation below shows the closed-form solution for the Ridge regression problem, with $\lambda = 2$:

$$w = (\Phi^T \Phi + \lambda I)^{-1} \Phi^T z = (\Phi^T \Phi + 2I)^{-1} \Phi^T z$$

Here, Φ is the result of applying the basis function to our training dataset's inputs, such that:

$$\Phi = \begin{bmatrix} 1 & \phi_1(x_1) & \phi_2(x_1) & \phi_3(x_1) \\ 1 & \phi_1(x_2) & \phi_2(x_2) & \phi_3(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \phi_1(x_5) & \phi_2(x_5) & \phi_3(x_5) \end{bmatrix} = \begin{bmatrix} 1 & 0.8 & 0.64 & 0.512 \\ 1 & 1 & 1 & 1 \\ 1 & 1.2 & 1.44 & 1.728 \\ 1 & 1.4 & 1.96 & 2.744 \\ 1 & 1.6 & 2.56 & 4.096 \end{bmatrix}$$

We are now able to learn the given polynomial regression model, with $\lambda = 2$:

$$(\Phi^T\Phi + \lambda I)^{-1} = \left(\begin{bmatrix} 1 & 0.8 & 0.64 & 0.512 \\ 1 & 1 & 1 & 1 \\ 1 & 1.2 & 1.44 & 1.728 \\ 1 & 1.4 & 1.96 & 2.744 \\ 1 & 1.6 & 2.56 & 4.096 \end{bmatrix}^T \begin{bmatrix} 1 & 0.8 & 0.64 & 0.512 \\ 1 & 1 & 1 & 1 \\ 1 & 1.2 & 1.44 & 1.728 \\ 1 & 1.4 & 1.96 & 2.744 \\ 1 & 1.6 & 2.56 & 4.096 \end{bmatrix} + \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \right)^{-1}$$

$$= \begin{bmatrix} 0.341688 & -0.121426 & -0.0749023 & -0.00932537 \\ -0.121426 & 0.389208 & -0.0966772 & -0.0744562 \\ -0.0749023 & -0.0966772 & 0.372578 & -0.17135 \\ -0.00932537 & -0.0744562 & -0.17135 & 0.179988 \end{bmatrix}$$

$$\Phi^T z = \begin{bmatrix} 1 & 0.8 & 0.64 & 0.512 \\ 1 & 1 & 1 & 1 \\ 1 & 1.2 & 1.44 & 1.728 \\ 1 & 1.4 & 1.96 & 2.744 \\ 1 & 1.6 & 2.56 & 4.096 \end{bmatrix}^T \begin{bmatrix} 24 \\ 20 \\ 10 \\ 13 \\ 12 \end{bmatrix} = \begin{bmatrix} 79 \\ 88.6 \\ 105.96 \\ 134.392 \end{bmatrix}$$

$$w = (\Phi^T\Phi + \lambda I)^{-1}\Phi^T z = \begin{bmatrix} 7.04508 \\ 4.64093 \\ 1.96734 \\ -1.30088 \end{bmatrix}$$

Having learned the regression model, we can now use it to predict labels z for new samples!

2. **Compute the training RMSE for the learnt regression model.**

   We know that the Root Mean Squared Error (RMSE) for a given regression model is defined as

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

   where N is the number of samples in the dataset, z_i is the true label for the i-th sample, and $\hat{z}_i$ is the predicted label for the i-th sample. As stated in the previous question's statement, $\hat{z}$ is given by the matrix product $\Phi \cdot w$. We can, then, compute the RMSE for the training dataset as follows:

$$\hat{z} = \Phi \cdot w = \begin{bmatrix} 1 & 0.8 & 0.64 & 0.512 \\ 1 & 1 & 1 & 1 \\ 1 & 1.2 & 1.44 & 1.728 \\ 1 & 1.4 & 1.96 & 2.744 \\ 1 & 1.6 & 2.56 & 4.096 \end{bmatrix} \cdot \begin{bmatrix} 7.04508 \\ 4.64093 \\ 1.96734 \\ -1.30088 \end{bmatrix} = \begin{bmatrix} 11.3509 \\ 12.3525 \\ 13.1992 \\ 13.8287 \\ 14.1785 \end{bmatrix}$$

$$\begin{aligned} \text{RMSE} &= \sqrt{\frac{1}{5}\sum_{i=1}^{5}(z_i - \hat{z}_i)^2} \\ &= \sqrt{\frac{1}{5}\left((24 - 11.35086463)^2 + \ldots + (12 - 14.17854143)^2\right)} \\ &= 6.84329 \end{aligned}$$

3. **Consider a multi-layer perceptron characterized by one hidden layer with 2 nodes. Using the activation function $f(x) = e^{0.1x}$ on all units, all weights initialized as 1 (including biases), and the half squared error loss, perform one batch gradient descent update (with learning rate $\eta = 0.1$) for the first three observations (0.8), (1) and (1.2).**

As a side-note, we'll be using the following notation in order to represent the resulting column matrix X' of the sum of all entries X_{ji} in a given line j, for all lines of the original matrix X:

$$\Sigma_X = X' = \begin{bmatrix} X_{11} + X_{12} + \ldots + X_{1n} \\ X_{21} + X_{22} + \ldots + X_{2n} \\ \vdots \\ X_{m1} + X_{m2} + \ldots + X_{mn} \end{bmatrix}$$

For example,

$$\Sigma_{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}} = \begin{bmatrix} 1+2+3 \\ 4+5+6 \end{bmatrix} = \begin{bmatrix} 6 \\ 15 \end{bmatrix}$$

This notation will be particularly useful in the latter section of this question's answer.

Our multi-layer perceptron, considering the parameters stated above, should only have one output-node, since we're considering a regression problem aiming to predict a single output variable. Ergo, our MLP will look as follows:

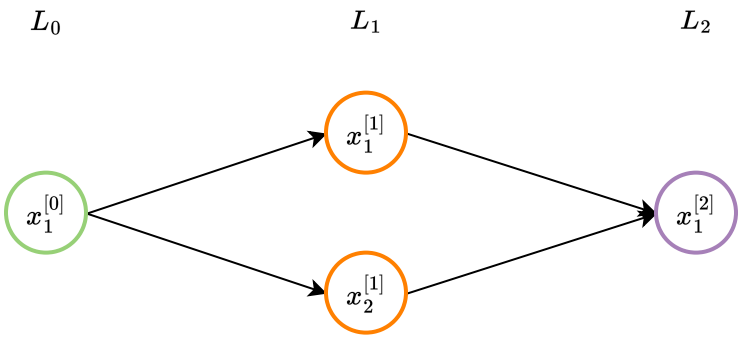

Figure 1: Visual representation of the multi-layer perceptron in question **(bias is implicit)**.

Each node in the hidden layer has an activation function $f(x) = e^{0.1x}$. Moreover, we know that the learning rule for the weights of each layer l is given by $\Delta w^{[l]} = -\eta \frac{\partial E}{\partial w^{[l]}}$ (with an analogous logic associated to biases), with $\eta = 0.1$ and E being the half squared error loss: $E = \frac{1}{2} \sum_{i=1}^{N} (z_i - \hat{z}_i)^2$.

A gradient descent update will require us to go through 3 phases: *forward propagation*, *back propagation* and *updates* (via gradient updates, updating biases and weights).

(i) **Forward Propagation**

Starting with the forward propagation, and considering l as a given layer, we have (where i matches the i-th sample):

$$z_i^{[l]} = w^{[l]} x_i^{[l-1]} + b^{[l]}, \quad x_i^{[l]} = f\left(z_i^{[l]}\right)$$

We also know, from the question's statement, both the input nodes' values and initial weight/bias matrices (considering a column per sample for x):

$$x^{[0]} = \begin{bmatrix} 0.8 & 1 & 1.2 \end{bmatrix}$$

$$w^{[1]} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad b^{[1]} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad w^{[2]} = \begin{bmatrix} 1 & 1 \end{bmatrix}, \quad b^{[2]} = \begin{bmatrix} 1 \end{bmatrix}$$

Applying the aforementioned equations for layer $l = 1$ (note that, just like with x, we'll also have one column per sample for $z^{[l]}$):

$$z^{[1]} = w^{[1]} x^{[0]} + b^{[1]} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0.8 & 1 & 1.2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.8 & 2 & 2.2 \\ 1.8 & 2 & 2.2 \end{bmatrix}$$

$$x^{[1]} = f(z^{[1]}) = f\left(\begin{bmatrix} 1.8 & 2 & 2.2 \\ 1.8 & 2 & 2.2 \end{bmatrix}\right) = \begin{bmatrix} 1.19722 & 1.2214 & 1.24608 \\ 1.19722 & 1.2214 & 1.24608 \end{bmatrix}$$

In the same manner, we can compute the values for layer $l = 2$:

$$z^{[2]} = \begin{bmatrix} 3.39443 & 3.44281 & 3.49215 \end{bmatrix}$$

$$x^{[2]} = \begin{bmatrix} 1.40417 & 1.41097 & 1.41795 \end{bmatrix}$$

Note that the value above, $x^{[2]}$, is the output of our MLP - the model's prediction of the output variable, for each sample (in each column).

(ii) **Back Propagation**

Now we'll want to propagate information backwards; for that, we'll need to use the chain rule, multiplying successive derivatives as we go backwards. We know the following:

$$\frac{\partial z^{[l]}}{\partial x^{[l-1]}} = w^{[l]} \qquad \frac{\partial z^{[l]}}{\partial w^{[l]}} = x^{[l-1]} \qquad \frac{\partial z^{[l]}}{\partial b^{[l]}} = 1$$

and also, considering L (= 2 in this case) as being the **output layer** (which will be useful for reusing previously calculated δ's):

$$\delta^{[L]} = \frac{\partial E}{\partial x^{[L]}} \circ \frac{\partial x^{[L]}}{\partial z^{[L]}} \qquad \frac{\partial E}{\partial w^{[l]}} = \delta^{[l]} \left(\frac{\partial z^{[l]}}{\partial w^{[l]}} \right)^T$$

$$\delta^{[l]}_{l \neq L} = \left(\frac{\partial z^{[l+1]}}{\partial x^{[l]}} \right)^T \cdot \delta^{[l+1]} \circ \frac{\partial x^{[l]}}{\partial z^{[l]}} \qquad \frac{\partial E}{\partial b^{[l]}} = \delta^{[l]} \underbrace{\cancel{\left(\frac{\partial z^{[l]}}{\partial b^{[l]}} \right)^T}}_{=1} = \delta^{[l]}$$

Note that, this way, we'll be able to create a matrix $\delta^{[l]}$ for each layer l, where each column i has the calculated $\delta_i^{[l]}$. We'll now be able to start propagating backwards, considering the following equalities (derived both in class and in the course's textbook):

$$\frac{\partial E}{\partial x_i^{[L]}} = \frac{\partial \frac{1}{2} \sum_{i=1}^{N} (z_i - x_i^{[L]})^2}{\partial x_i^{[L]}} = \sum_{i=1}^{N} (x_i^{[L]} - z_i), \quad \frac{\partial x_i^{[l]}}{\partial z_i^{[l]}} = \frac{\partial e^{0.1 z_i^{[l]}}}{\partial z_i^{[l]}} = 0.1 e^{0.1 z_i^{[l]}}$$

$$\frac{\partial z_i^{[l]}}{\partial x_i^{[l-1]}} = w^{[l]}, \quad \frac{\partial z_i^{[l]}}{\partial b^{[l]}} = 1, \quad \frac{\partial z_i^{[l]}}{\partial w^{[l]}} = x_i^{[l-1]}$$

With $L = 2$:

$$\begin{aligned}
\delta^{[2]} &= \frac{\partial E}{\partial x^{[2]}} \circ \frac{\partial x^{[2]}}{\partial z^{[2]}} \\
&= \left[\sum_{i=1}^{N} (x_i^{[2]} - z_i) \right] \circ 0.1 e^{0.1 z^{[2]}} \\
&= \begin{bmatrix} -22.5958 & -18.589 & -8.58205 \end{bmatrix} \circ \begin{bmatrix} 0.140417 & 0.141097 & 0.141795 \end{bmatrix} \\
&= \begin{bmatrix} -3.17283 & -2.62286 & -1.2169 \end{bmatrix}
\end{aligned}$$

$$
\begin{aligned}
\delta^{[1]} &= \left(\frac{\partial z^{[2]}}{\partial x^{[1]}}\right)^T \cdot \delta^{[2]} \circ \frac{\partial x^{[1]}}{\partial z^{[1]}} \\
&= w^{[2]^T} \cdot \delta^{[2]} \circ 0.1e^{0.1z^{[1]}} \\
&= \begin{bmatrix} -3.17283 & -2.62286 & -1.2169 \\ -3.17283 & -2.62286 & -1.2169 \end{bmatrix} \circ \begin{bmatrix} 0.119722 & 0.12214 & 0.124608 \\ 0.119722 & 0.12214 & 0.124608 \end{bmatrix} \\
&= \begin{bmatrix} -0.379857 & -0.320357 & -0.151634 \\ -0.379857 & -0.320357 & -0.151634 \end{bmatrix}
\end{aligned}
$$

(iii) **Updates**

In the last phase, we'll be updating our model: after computing the gradients, we'll be able to update weights and biases!

Starting with weight matrices:

$$
\begin{aligned}
\frac{\partial E}{\partial w^{[1]}} &= \delta^{[1]} \cdot \left(\frac{\partial z^{[1]}}{\partial w^{[1]}}\right)^T = \delta^{[1]} \cdot x^{[0]^T} = \begin{bmatrix} -0.806204 \\ -0.806204 \end{bmatrix} \\
\frac{\partial E}{\partial w^{[2]}} &= \delta^{[2]} \cdot \left(\frac{\partial z^{[2]}}{\partial w^{[2]}}\right)^T = \delta^{[2]} \cdot x^{[1]^T} = \begin{bmatrix} -8.51849 & -8.51849 \end{bmatrix} \\
w^{[1]} = w^{[1]} - \eta \cdot \frac{\partial E}{\partial w^{[1]}} &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 0.1 \cdot \begin{bmatrix} -0.806204 \\ -0.806204 \end{bmatrix} = \begin{bmatrix} 1.08062 \\ 1.08062 \end{bmatrix} \\
w^{[2]} = w^{[2]} - \eta \cdot \frac{\partial E}{\partial w^{[2]}} &= \begin{bmatrix} 1 & 1 \end{bmatrix} - 0.1 \cdot \begin{bmatrix} -8.51849 & -8.51849 \end{bmatrix} \\
&= \begin{bmatrix} 1.85185 & 1.85185 \end{bmatrix}
\end{aligned}
$$

Finally, we'll update biases:

$$
\frac{\partial E}{\partial b^{[1]}} = \Sigma_{\delta^{[1]}} = \begin{bmatrix} -0.851849 \\ -0.851849 \end{bmatrix} \qquad \frac{\partial E}{\partial b^{[2]}} = \Sigma_{\delta^{[2]}} = \begin{bmatrix} -7.01259 \end{bmatrix}
$$

$$
\begin{aligned}
b^{[1]} = b^{[1]} - \eta \cdot \frac{\partial E}{\partial b^{[1]}} &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 0.1 \begin{bmatrix} -0.851849 \\ -0.851849 \end{bmatrix} \\
&= \begin{bmatrix} 1.08518 \\ 1.08518 \end{bmatrix} \\
b^{[2]} = b^{[2]} - \eta \cdot \frac{\partial E}{\partial b^{[2]}} &= \begin{bmatrix} 1 \end{bmatrix} - 0.1 \begin{bmatrix} -7.01259 \end{bmatrix} \\
&= \begin{bmatrix} 1.70126 \end{bmatrix}
\end{aligned}
$$

Part II: Programming and critical analysis

The code utilized to answer the following questions is available in this report's appendix.

4. **Compute the MAE of the three regressors: linear regression, MLP_1 and MLP_2.**

   We opted to utilize `sklearn`'s `mean_absolute_error` function to compute the MAE of the three regressors. The regressors were created as shown in the appendix (using `Ridge` and `MLPRegressor` with the respective parameters).

   We gathered the following results:

Regressor	**MAE**
Linear Regression (Ridge)	0.16283
MLP_1	0.06804
MLP_2	0.09781

Table 3: Collected Mean Absolute Errors for each specified regressor

5. **Plot the residues (in absolute value) using two visualizations: boxplots and histograms.**

   Each regressor's residues, calculated as the absolute difference between the predicted and actual values, were plotted using both boxplots and histograms (using, respectively, `seaborn`'s `boxplot` and `histplot` functions), as shown in this report's appendix (figure after the code).

6. **How many iterations were required for MLP_1 and MLP_2 to converge?**

   Calling the `print_regressor` method for each regressor shows us not only the MAE, but also the number of iterations required for each of the MLP regressors to converge. In this case:

Regressor	**Early Stopping?**	**# Iters. Req. Converge**
MLP_1	Yes	452
MLP_2	No	77

Table 4: Number of iterations required for each specified regressor to converge

7. **What can be motivating the unexpected differences on the number of iterations? Hypothesize one reason underlying the observed performance differences between the MLPs.**

   As it has been noted in the previous question's answer, MLP_1 takes many more iterations to converge than MLP_2 - almost six times as many. It's probably worth emphasizing that the number of iterations in a batch gradient descent algorithm matches the number of epochs ran - the amount of times the algorithm goes through the entire dataset.

   With `MLPRegressor`'s early stopping implementation, the training stops after a given number of epochs have passed without the validation score improving by at least a given tolerance (in order to avoid overfitting). The validation set, with `MLPRegressor`'s `shuffle` parameter set by default to `True`, may contain samples from both the training and test sets: with differing training sets (which forcefully seems like the case here), the training process may converge at a different amount of epochs; it's also likely that a training set with a lesser amount of samples may take more epochs to converge, as the algorithm will have to go through the entire dataset more times to reach the "same amount of samples seen" in order to better fit the data.

   The **different number of iterations** could, then, be associated with the differing training sets used to train each regressor, plus the fact that MLP_1 stops when the validation phase tells it do so, while MLP_2 strictly looks at convergence regarding training data.

   Regarding performance, MLP_1 seems to perform better than MLP_2, with a lower (and thus better) MAE, as stated in question 4.'s answer. Although both regressors end up converging in training, MLP_1 has the advantage of stopping right where the validation score starts to stagnate/decrease: this means that the regressor ends up not overfitting the data, as it would if it kept training for more epochs until reaching "regular" convergence, like MLP_2 (which appears to be a bit more overfitted, in comparison to MLP_1).

Appendix

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.io.arff import loadarff
from sklearn.linear_model import Ridge
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error
from sklearn.neural_network import MLPRegressor

sns.set_style("darkgrid")

# Load data
data = loadarff("data/kin8nm.arff")
df = pd.DataFrame(data[0])

X = df.drop("y", axis=1).values
y = df["y"].values

X_train, X_test, y_train, y_test = train_test_split(
  X, y,
  test_size=0.3, random_state=0
)


def predict(regressor):
  regressor.fit(X_train, y_train)
  return regressor.predict(X_test)


def plot_regressors_residues():
  """Utilized for answering question 5."""
  global regressors
  predictions = []
  descriptions = []
  for description, regressor in regressors.items():
    predictions.append(predict(regressor))
    descriptions.append(description)

  fig, (ax1, ax2) = plt.subplots(2, figsize=(8, 12))

  data = pd.DataFrame(
    {
      description: np.abs(y_test - prediction)
      for description, prediction in zip(descriptions, predictions)
    }
  )

  sns.histplot(
    data=data,
    bins=30,
    ax=ax1,
    multiple="dodge",
  )
```

```
  ax1.set_title("Residues histogram for each regressor")
  ax1.set_xlabel("Residue")
  ax1.set_ylabel("Count")
  ax1.legend(descriptions)

  sns.boxplot(
    data=data,
    ax=ax2,
    orient="h",
  )
  ax2.set_title("Residues boxplot for each regressor")
  ax2.set_xlabel("Residue")
  ax2.set_ylabel("Regressor")
  ax2.legend(descriptions)
  plt.savefig("assets/residues.png")


def print_regressor(regressor, description, y_pred):
  """Utilized for answering questions 4. and 6."""
  print(description)
  print("MAE: {:.5f}".format(mean_absolute_error(y_test, y_pred)))
  if "MLP" in description:
    print("Iterations: {}".format(regressor.n_iter_))


regressors = {
  "Ridge": Ridge(alpha=0.1),
  "MLP 1": MLPRegressor(
    hidden_layer_sizes=(10, 10),
    activation="tanh",
    max_iter=500,
    random_state=0,
    early_stopping=True,
  ),
  "MLP 2": MLPRegressor(
    hidden_layer_sizes=(10, 10),
    activation="tanh",
    max_iter=500,
    random_state=0
  ),
}

for description, regressor in regressors.items():
  y_pred = predict(regressor)
  print("---")
  print_regressor(regressor, description, y_pred)

plot_regressors_residues()
```

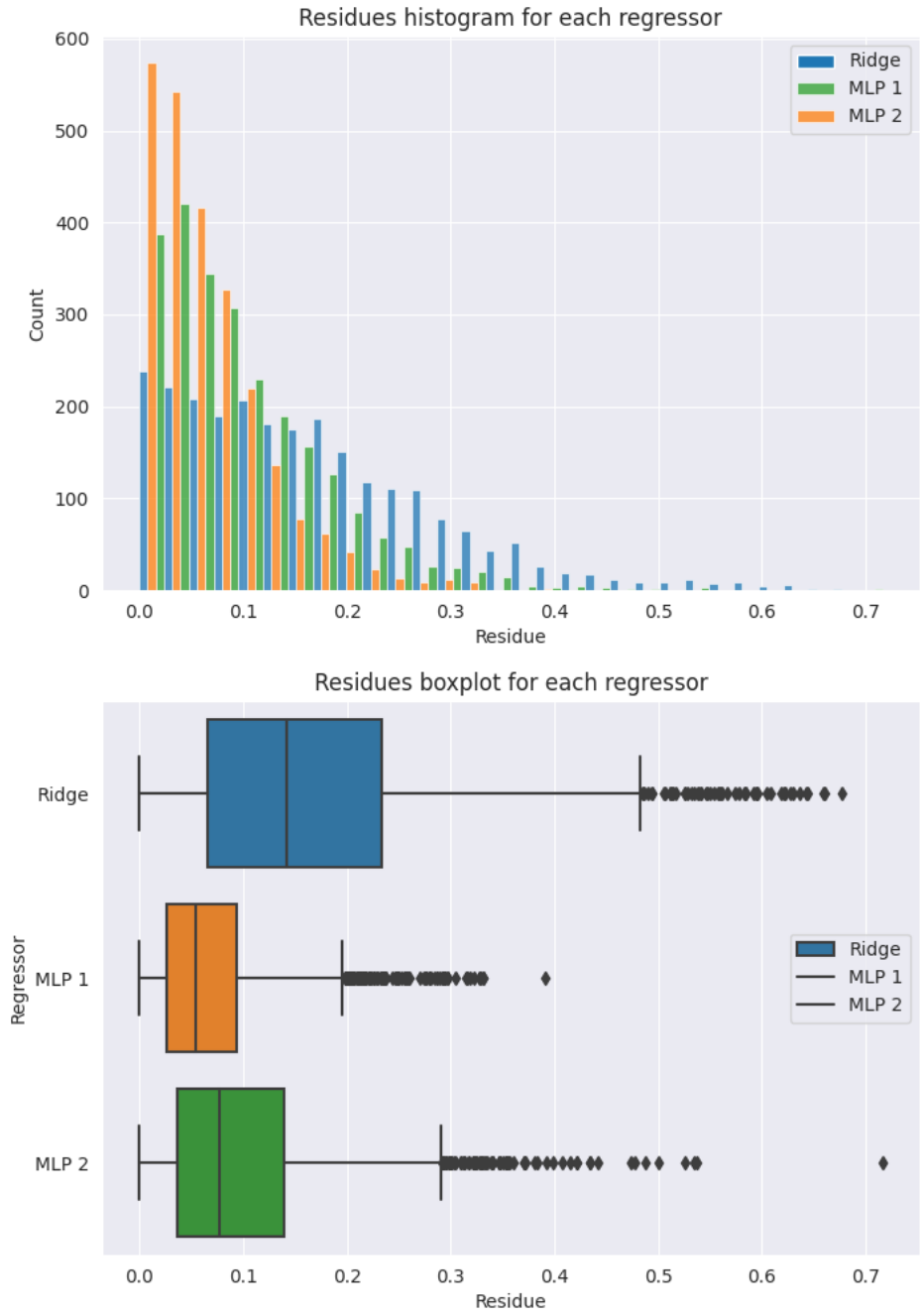

Figure 2: Ridge regression's residue plotting