BILKENT UNIVERSITY

EEE 485: STATISTICAL LEARNING AND DATA ANALYTICS

Term Project: Final Report

Prediction of Song Popularity

Berk Demirkan 21802124

Tuğcan Ünalan 21802512

Berk Demirkan
Tuğcan Ünalan

Introduction

In the scope of this project, our objective is to implement a regression analysis utilizing the comprehensive "Spotify Dataset 1921-2020, 600k+ Tracks" dataset. This dataset encompasses approximately 600,000 samples, incorporating features associated with music, such as energy, danceability, and tempo. Our primary goal is to develop a machine learning model that leverages this dataset to predict songs' popularity accurately.

We have employed various regression techniques, including linear, decision tree, random forest, and neural network regression. In the final report, we will refine the decision tree and neural network regression methods and identify a more effective cost function that accurately captures the true error. This will ultimately enhance the model's performance and improve the reliability of our predictions.

Dataset Description:

The dataset comprises 603,549 samples in total. To ensure data consistency and reliability, we have excluded samples with unknown or invalid values, resulting in a final dataset of 586.672 samples.

The key attributes in this dataset include popularity, duration, release date, danceability, energy, key, speechiness, acousticness, instrumentalness, liveness, valence, tempo, and time signature. Since the “release date” feature is in a string format, we have calculated a “days since release” value for each sample; the date used for this calculation is the dataset’s last update date which is 30th of April, 2021.

Below is a detailed explanation of some less obvious features:

- Popularity: This metric is a normalized value ranging from 0 to 100, 100 being the most popular and 0 the least popular. While there is only one song with a 100 rating and 44690 songs with a 0 rating. Meaning the dataset is quite unbalanced. The mean is 27.57, and the standard deviation for this metric is 18.37.
- Danceability: This metric assesses a track's suitability for dancing, considering factors such as tempo, rhythm stability, beat strength, and overall regularity. Values range from 0.0 (least danceable) to 1.0 (most danceable).
- Energy: Ranging from 0.0 to 1.0, this measure reflects a track's intensity and activity level, with energetic tracks typically characterized by faster tempo, louder volume, and higher noise levels. Contributing perceptual features include dynamic range, perceived loudness, timbre, onset rate, and general entropy.
- Key: This feature represents the estimated overall key of a track, utilizing standard Pitch Class notation to map integers to pitches. For instance, 0 corresponds to C, 1 to C♯/D♭, 2 to D, etc. If no key is detected, the value defaults to -1.
- Loudness: This measure denotes the average loudness of a track in decibels (dB), facilitating comparisons of relative loudness between tracks. Typically, values range between -60 and 0 dB.
- Mode: This feature indicates a track's modality or the type of scale derived from its melodic content. One represents a major modality, while zero represents a minor modality.
- Speechiness: This metric detects the presence of spoken words within a track, with values closer to 1.0 indicating a higher proportion of speech-like content. Values above 0.66 suggest that the track likely consists entirely of spoken words, while values between 0.33 and 0.66 imply a mixture of music and speech. Values below 0.33 generally indicate music and other non-speech content.
- Acousticness: This confidence measure, ranging from 0.0 to 1.0, estimates the likelihood that a track is acoustic. A value of 1.0 signifies high confidence in the track's acoustic nature.
- Instrumentalness: This feature predicts the absence of vocals within a track, treating "ooh" and "aah" sounds as instrumental. Rap or spoken word tracks are considered "vocal." As the

instrumentalness value approaches 1.0, the likelihood of the track lacking vocal content increases. Values above 0.5 typically represent instrumental tracks.

- Liveness: This metric detects the presence of a live audience in a recording, with higher values indicating a greater probability of a live performance. A value above 0.8 suggests a strong likelihood that the track was performed live.
- Valence: Ranging from 0.0 to 1.0, this measure describes the musical positivity conveyed by a track. High-valence tracks evoke positive emotions (e.g., happiness, cheerfulness, euphoria), while low-valence tracks convey more negative feelings (e.g., sadness, depression, anger).
- Tempo: This feature estimates a track's overall tempo in beats per minute (BPM), which, in musical terminology, refers to the speed or pace of a given piece and is derived directly from the average beat duration.

1. Linear Regression

Linear regression analysis is a widely-used statistical method for predicting a dependent variable based on one or more independent variables [1]. The equation can summarise the underlying mathematical concept of linear regression:

$$y = \boldsymbol{\beta}_0 + \boldsymbol{\beta}_1 x_1 + \ldots + \boldsymbol{\beta}_p x_p \text{[2]}$$

In this equation, x represents the input data, and **β** denotes the learning parameters.

We used the pre-processed dataset and the gradient descent optimization technique to implement linear regression. Gradient descent is an iterative algorithm that aims to find the local minimum/maximum. It successively calculates the next point by evaluating the gradient at the current position, scaling it with a learning rate, and subtracting the obtained value from the current location. This process can be expressed by the following equation:

$$p_{n+1} = p_n - \eta \nabla f(p_n)$$

The parameter η is crucial because it scales the gradient and controls the step size. In summary, gradient descent involves the following steps [3]:

1. Choose a starting point
2. Calculate the gradient at this point
3. Take a scaled step in the opposite direction

Steps 2 and 3 are repeated until the maximum number of iterations is reached. We implemented this algorithm in our code, as shown below:

```python
def gradient_descent(
    X_train, Y_train, beta, iterations, learning_rate, patience=10, min_delta=1e-3
):
    n = len(X_train)
    best_cost = np.inf
    patience_counter = 0

    costs = []
    for i in range(iterations):
        Y_pred = np.dot(X_train, beta.T)

        cost = mse(Y_train, Y_pred)
        costs.append(cost)

        gradient = (1 / n) * np.dot(X_train.T, (Y_pred - Y_train))

        beta -= learning_rate * gradient.T

        if (i + 1) % 100 == 0:
            print(f"Iteration: {i+1}, Training Cost: {round(cost, 5)}")

        if cost + min_delta < best_cost:
            patience_counter = 0
            best_cost = cost
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print(f"Early stop at iteration {i}. Training Cost: {round(cost, 5)}")
                break

    return beta, costs
```

Then, for the error calculation part, we used the following mean square error function:

```python
def mse(Y_true, Y_pred):
    mse_ = np.mean((Y_true-Y_pred) ** 2)
    return mse_
```

2. Decision Tree Regression

Decision trees are versatile tools for constructing regression or classification models in a tree structure. The algorithm recursively splits the dataset into smaller subsets, gradually developing the decision tree. Here is an illustration of a typical decision tree algorithm:

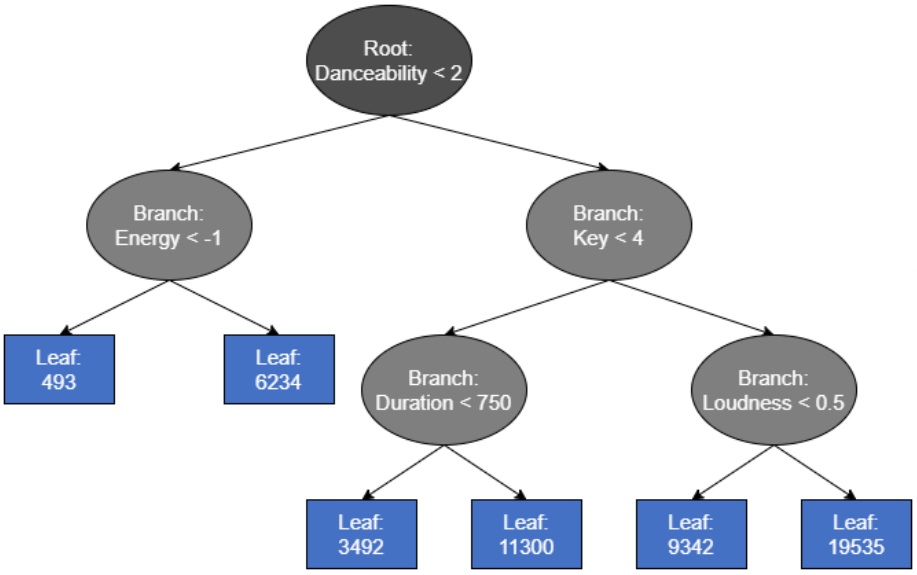


Figure 2: Decision Tree Algorithm [4]

As shown in the figure, decision trees consist of three types of nodes: root nodes, interior nodes, and leaf nodes. The root node encompasses the entire sample, branch nodes signify dataset features, and branches represent decision rules. The leaf nodes denote the outcome of the decision tree. Starting from the root node and following the conditions at each node, a leaf node is eventually reached.

For our implementation, we used an algorithm to search for the optimal split on the given dataset exhaustively. We are splitting the sample set at each sample with each feature and taking the weighted costs of these splits. After the split criterion of the root node has been established, the same algorithm is used to create the splits for these two branches. This process continues until the maximum depth is reached, in which case the mean of the sample set is taken as the leaf, or the given sample set has only one sample, in which case that sample is chosen as the sample.

Finding the best split for a sample set is done using the following function:

```python
@jit(nopython=True)
def find_best_split(X: np.ndarray, Y: np.ndarray):
    best_feature, best_threshold, best_error = None, None, np.inf
    for feature_idx in prange(X.shape[1]):
        feature_values = X[:, feature_idx]
        feature_values.sort()
        for threshold in feature_values:
            Y_left = Y[X[:, feature_idx] <= threshold]
            Y_right = Y[X[:, feature_idx] > threshold]

            if len(Y_left) == 0 or len(Y_right) == 0:
                continue

            error_left = mse(Y_left, np.mean(Y_left))
            error_right = mse(Y_right, np.mean(Y_right))
            weighted_error = len(Y_left) * error_left + len(Y_right) * error_right

            if weighted_error < best_error:
                best_error = weighted_error
                best_feature = feature_idx
                best_threshold = threshold
                break  # Break out of the loop once the best error is found

    return best_feature, best_threshold
```

Since the function is very inefficient, the @jit decorator is used to pre-compile the Python code and increase performance.

3. Random Forest

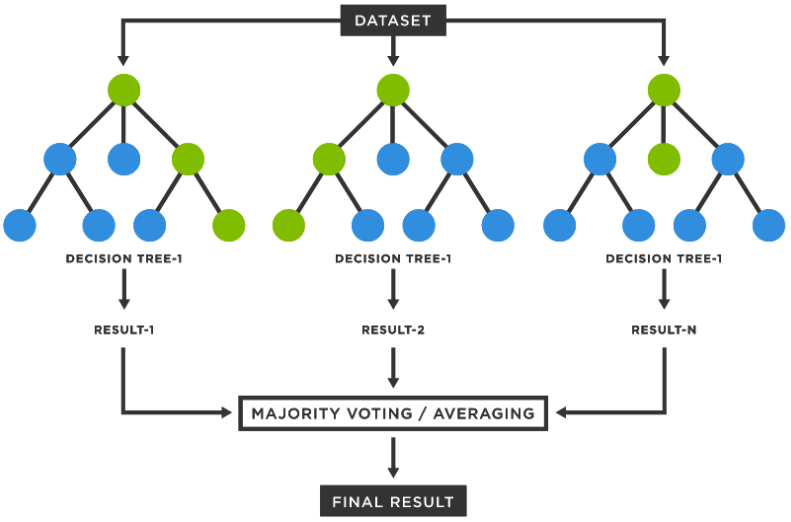


Figure 3: Random Forest Algorithm [5]

The random forest algorithm uses the decision tree as its base. A random forest is constructed by bagging different dataset sections and training many trees using these different sample sets. On a classification task, the prediction is determined using a majority vote; on a regression task such as ours, the prediction is determined by taking the mean of the predictions. Since the training time for each tree is proportional to the square of the sample size, A random forest is usually much faster to train. Due to the random nature of the algorithm, we expect some improvement in contrast to a single decision tree.

In our implementation, each tree is trained on a sample size of $(N_{dataset}/N_{trees})$. The actual samples are selected randomly with the chance for replacement which gives the model higher accuracy. This is done using the following class:

```
class RandomForest:
    def __init__(self, n_trees, max_depth):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self._trees = []

    def fit(self, X: np.ndarray, Y: np.ndarray, verbose=False):
        self.trees = []
        for i in range(self.n_trees):
            if (verbose):
                print(f"\nTree: {i+1}/{self.n_trees}")

            sample_idxs = np.random.choice(np.arange(X.shape[0]), int(X.shape[0]/self.n_trees), replace=True)

            tree = DecisionTree(max_depth=self.max_depth)
            X_sample = X[sample_idxs]
            Y_sample = Y[sample_idxs]
            tree.fit(X_sample, Y_sample, verbose=verbose)
            self._trees.append(tree)


    def predict(self, X):
        tree_preds = np.array([tree.predict(X) for tree in self._trees])
        return tree_preds.mean(axis=0)
```

4. Neural Network

In developing our fourth machine learning algorithm, we employed neural networks, which can be characterised as a supervised learning technique. Neural networks are composed of interconnected layers, each containing neurons that connect to other neurons within these layers. Every neuron within a neural network is equipped with an activation function, and the connections between neurons involve the multiplication of the signal by a weight matrix, followed by the addition of a bias vector [6]. A typical neural network comprises one input layer, an output layer, and a hidden layer. An illustration of a neural network is provided below [7]:

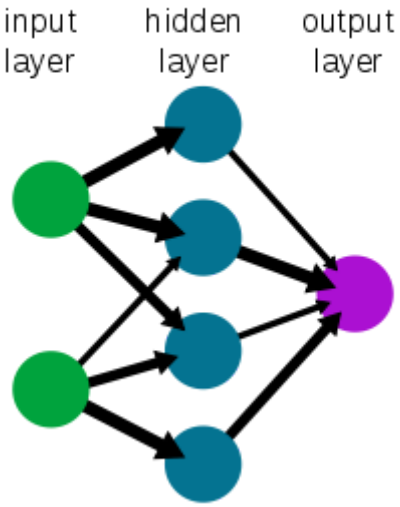


Figure 4: A Simple Neural Network

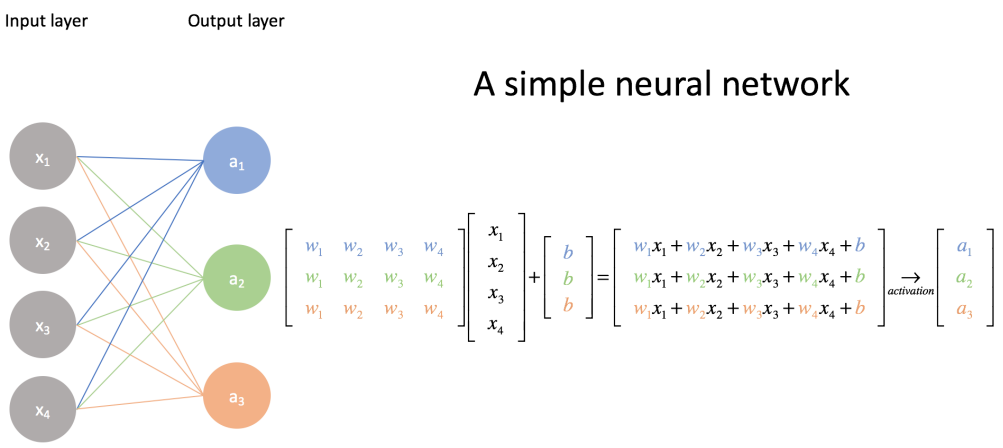
The mathematical relationships between the input, hidden, and output layers can be represented by the following equation:

$$y = w^T x + b$$

In this equation, y denotes the output, x symbolises the input, b represents the bias, and w signifies the weight matrix. The matrix notation for the neural network can also be expressed as follows [8]:

Figure 5: Matrix Representation of Neural Network

For our implementation, we used two hidden layers of size 200. Such a large hidden layer size was selected to ensure that any complex relationships between the features and the popularity were captured. We executed neural network regression through the following steps. Initially, we partitioned the dataset into a training set and a test set:



```
#Train/Test Split
#X = X.sample(n=1000).reset_index(drop=True)

idxs = np.random.choice(np.arange(X.shape[0]), int(0.8*X.shape[0]), replace=False)

X_train = X[idxs]
Y_train = Y[idxs]

X_test = X[~idxs]
Y_test = Y[~idxs]
```

Then, for the initialization part, we used the Xavier uniform distribution, which can be expressed as the following equation:

$$w_0 = \sqrt{\left(\frac{6}{L_{pre} + L_{post}}\right)}$$

Where the interval is given as $[-w_0, w_{0..}]$. The code representation is given as:

```
def Xavier_init(n_pre, n_post):
    w0 = np.sqrt(6 / (n_pre + n_post))
    W = np.random.uniform(-w0, w0, (n_post, n_pre))
    b = np.random.uniform(-w0, w0, (n_post, 1))
    return W, b
```

Next; we defined the activation function utilising the Rectified Linear Unit and linear activation functions as follows. Note that using only the linear activation function, the performance would equal the linear regression implementation using only the linear activation function.

```
def relu(X: np.ndarray):
    A = X * (X > 0)
    dA = 1 * (X > 0)
    return A, dA
```

```
def linear(X: np.ndarray):
    A = X
    dA = 1
    return A, dA
```

We then established the forward propagation process through which each sample advances in the neural network:

```python
def forward_propagate(We, X):
    W1 = We["W1"]
    W2 = We["W2"]
    W3 = We["W3"]
    b1 = We["b1"]
    b2 = We["b2"]
    b3 = We["b3"]

    Z1 = np.dot(W1, X.T) + b1
    A1, dA1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2, dA2 = relu(Z2)
    Z3 = np.dot(W3, A2) + b3
    A3, dA3 = linear(Z3)

    cache = {"Z1": Z1, "A1": A1, "dA1": dA1, "Z2": Z2, "A2": A2, "dA2": dA2, "Z3": Z3, "A3": A3, "dA3": dA3}
    return cache
```

Lastly, after writing the functions for gradient calculation and updating the weights, we trained our neural network and found the error with the following function:

```python
def mse(Y_true: np.ndarray, Y_pred: np.ndarray):
    return ((Y_pred - Y_true) ** 2).mean()
```

For faster convergence, the “ADAM” optimizer was implemented. The ADAptive Momentum estimation optimizer is an algorithm that optimizes the learning speed of a gradient descent implementation. By using both the gradients themselves and their second powers to keep track of two momentum-like values, the “ADAM” optimizer prevents oscillation and achieves fast convergence even with very low learning rates.

4. Gantt Chart

Work Package/ Week	17/05 Week	24/05 Week	01/05 Week	08/05 Week	15/05 Week
Linear Regression - Implementation	Tuğcan				
Decision Tree - Reserch / Implementation	Berk	Berk			
Random Forest - Implementation		Berk / Tuğcan	Berk		
Random Forest - Improvements			Tuğcan	Tuğcan	
Neural Network - Research / Implementation		Berk	Berk		
Neural Network - Improvements			Berk / Tuğcan	Tuğcan	
Result Analysis and Method Comparison				Berk / Tuğcan	Berk / Tuğcan
Final Report / Demo Preperation					Berk / Tuğcan

Figure 1: The Gantt Chart for Term Project

5. Results and Discussion

5.1. Linear Regression

The “cost vs. iteration” plot for the linear regression implementation is given below.

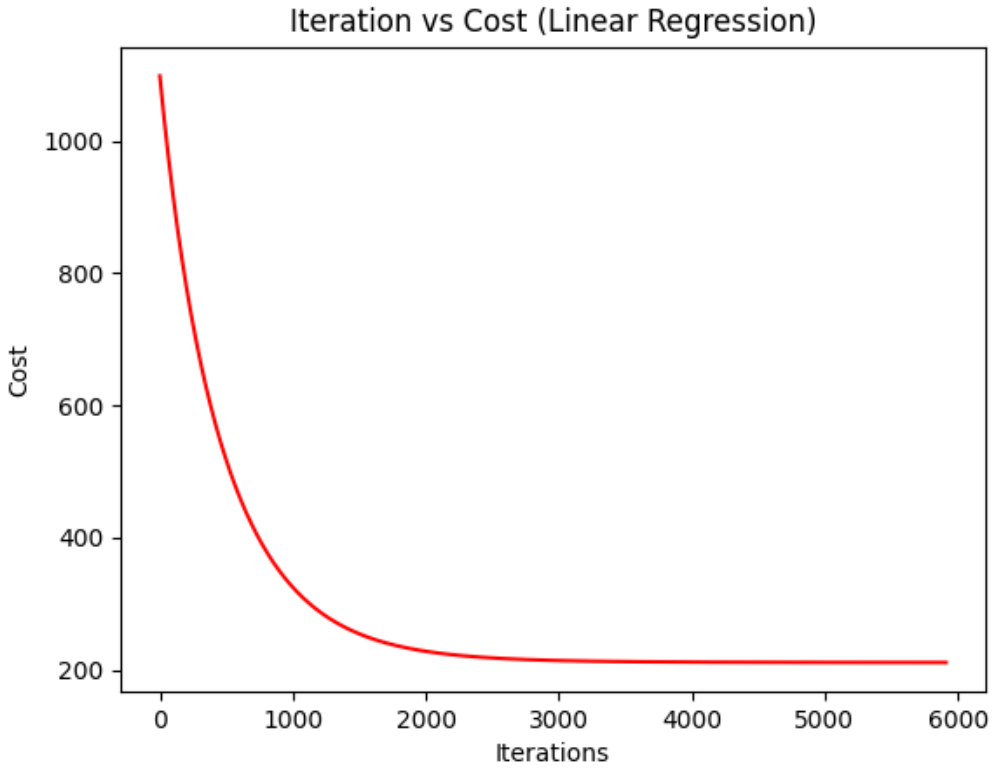


Figure 2: Iterations vs. Cost for Linear Regression

The plot reveals that as the number of iterations increases, the error approaches approximately 200. Training performance is satisfactory, averaging around ten iterations per second. An early stopping feature was implemented to stop the training when progress staggered. After 3000 iterations, the error remains virtually unchanged and is nearly 200. Specifically, the final training error is 211.06, and the final testing error is 209.92. The "prediction vs. actual" plots for both the test and training datasets are given below.

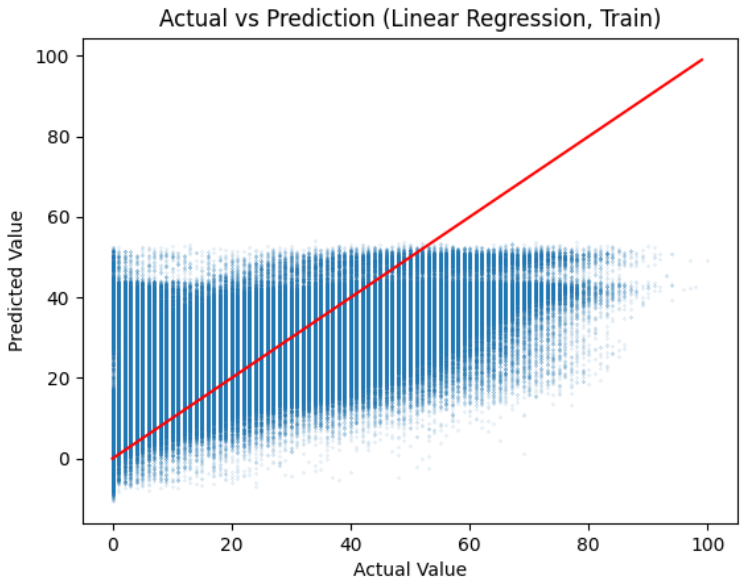


Figure 3: Predicted vs. Actual for the Train in Linear Regression

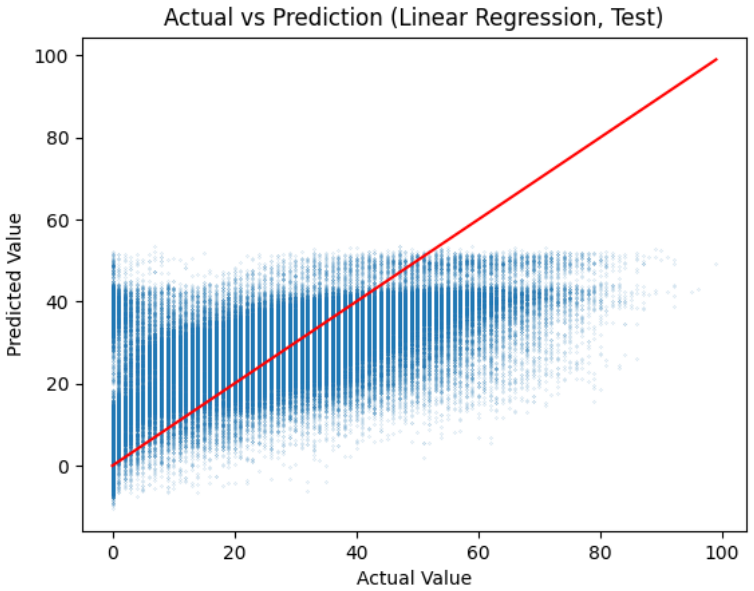


Figure 4: Predicted vs. Actual for the Test in Linear Regression

As can be seen from Figures 3 and 4, the linear regression model shows a clear indication of linearity between the actual value and prediction. However, we can see that the model fails at predicting values higher than 50 due to the low amount of samples of higher popularity.

5.2. Decision Tree Regression.

For the second method, decision tree regression, owing to the inefficient nature of finding the best split when training a decision tree, we first sampled our dataset to obtain a smaller subset of size 20,000 for training. Since the dataset is unbalanced, little to no overfitting was seen, even with high maximum depth values. The tree reached a depth of 39 on this subset with a training cost of 345.43 and a testing cost of 359.43. Below are the "actual vs. prediction" plots for both the training and testing datasets.

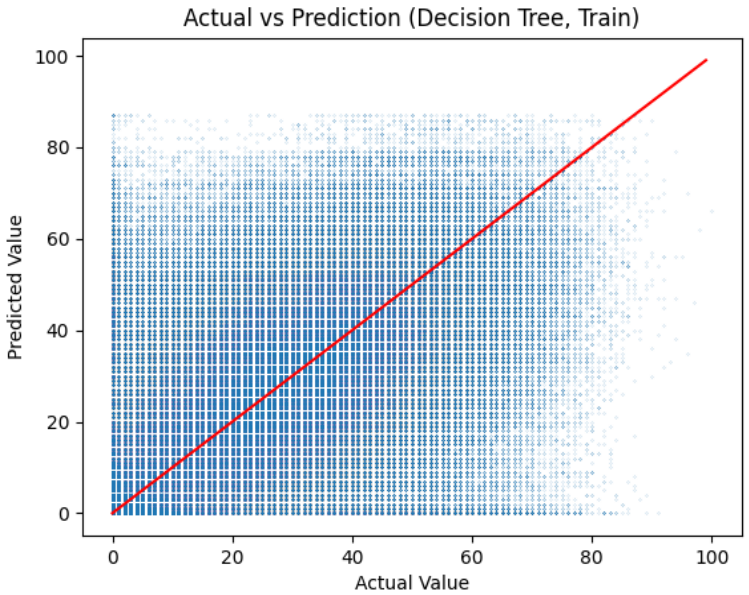


Figure 5: Actual vs. Prediction Plot for Decision Tree on the Training Dataset

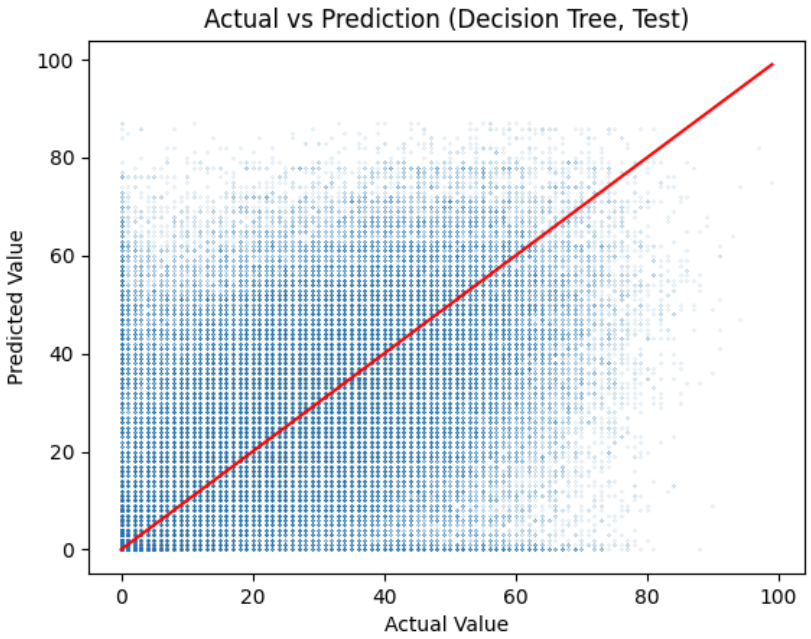


Figure 6: Actual vs. Prediction Plot for Decision Tree on the Testing Dataset

Figure 6 reveals that the decision tree algorithm has distributed the model's error around the ideal linear line. The decision tree model can predict higher values but with lower accuracy than the linear regression algorithm.

5.3. Random Forest

Since the individual trees in the random forest are trained on smaller datasets, we trained the random forest on the whole dataset. The model has little to no overfitting at high maximum depth values, so the model was trained with the maximum depth parameter set to infinite. The training cost is 174.63, and the testing cost is 176.32. The training performance of the model is satisfactory, with the model taking 10 seconds per tree. However, the prediction performance is worse, taking almost three minutes for the test dataset. The following “actual vs. prediction” plots were obtained from the model.

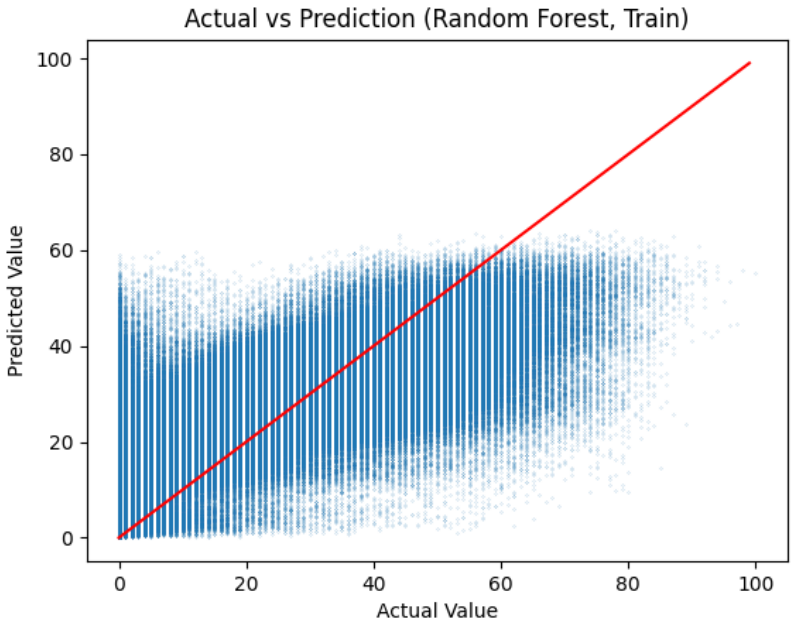


Figure 7: Actual vs. Prediction Plot for Random Forest on the Training Dataset

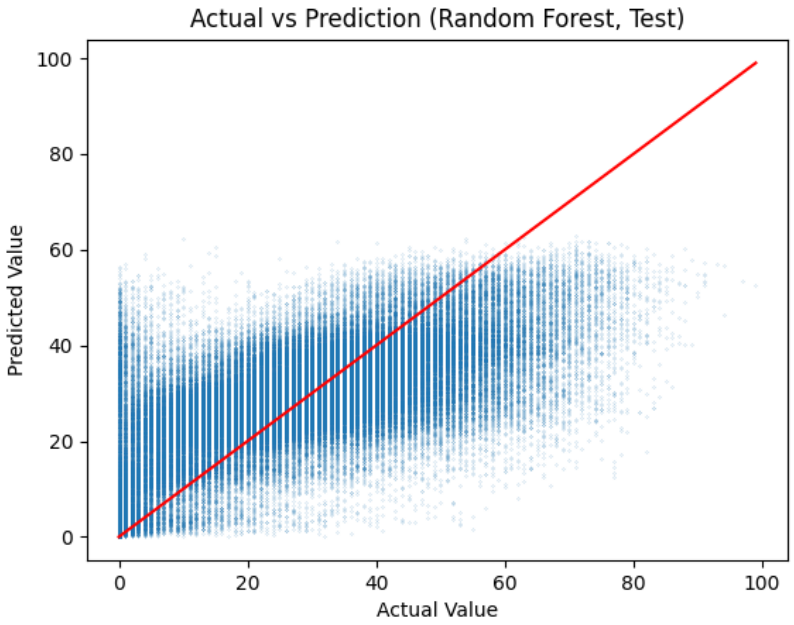


Figure 8: Actual vs. Prediction plot for Random Forest on the Testing Dataset

Figures 7 and 8 demonstrate the performance of the random forest implementation, with much lower costs than the linear regression and single decision tree implementations. The model can predict values up to 60, albeit with some error. The near training and testing costs indicate that no overfitting has occurred.

5.4 Neural Network

The following cost vs. iteration plot was obtained for the neural network method.

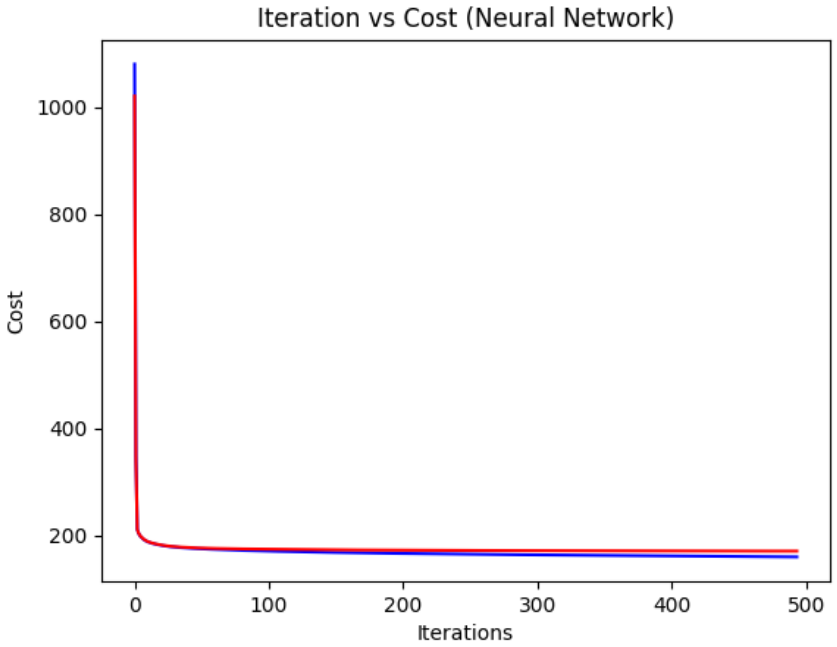


Figure 9: Iteration vs. Cost for Neural Network

As it can be seen from the figure, as the iterations increase, the cost comes closer to 200, and to be exact, after some point, it reaches 169.01, which we found the training error as and does not change after 200 iterations. With a lower batch Specifically, the training error was 169.01, and the testing error 170.59. The training performance is slow as it needs many mini-iterations over the dataset to converge until no further improvement is seen. The prediction performance is a lot better, taking mere seconds. The "prediction vs. actual" plots for both the test and train datasets are given below.

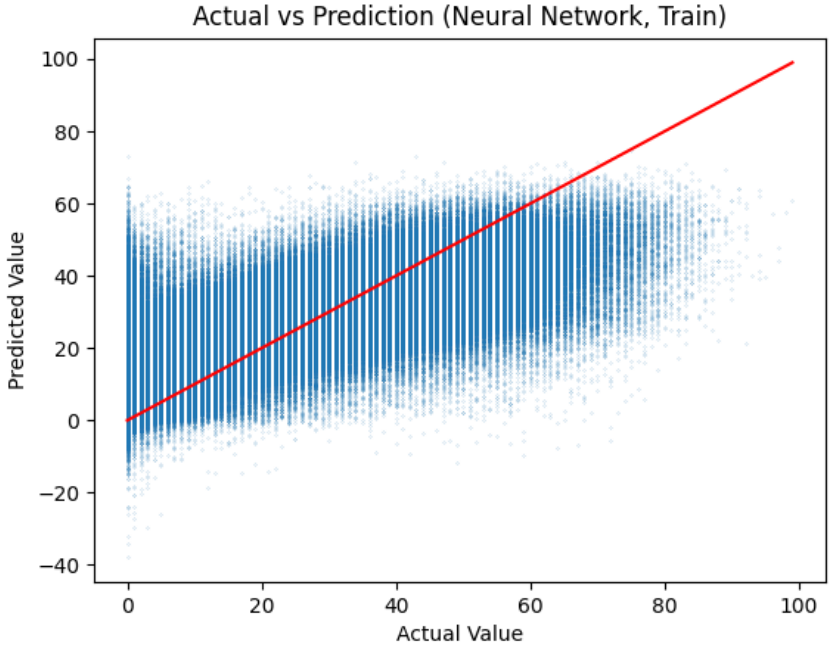


Figure 10: Predicted vs. Actual for the Training Set in Neural Network

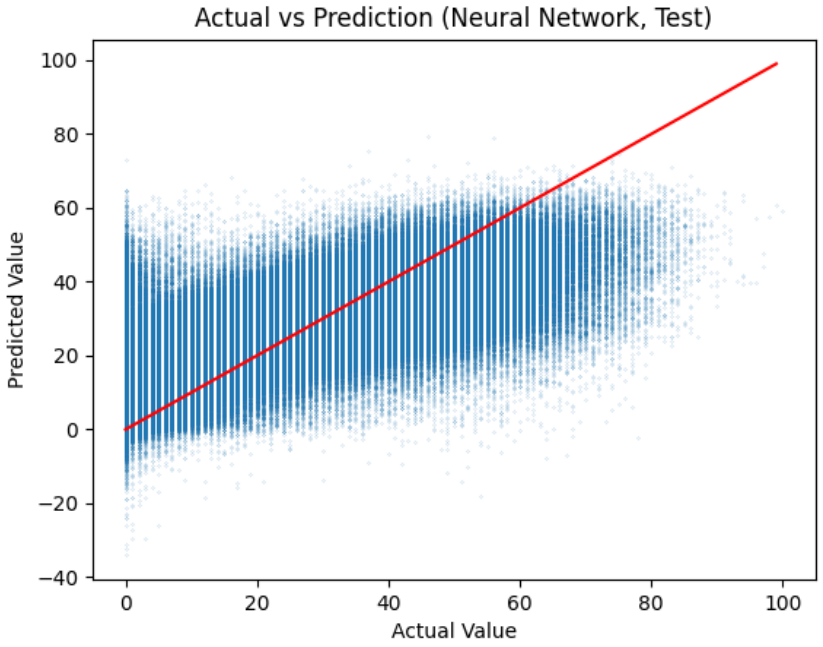


Figure 11: Predicted vs. Actual for the Testing Set in Neural Network

Just like the Random forest implementation, the "actual vs. prediction" graphs indicate a clear trend toward correct predictions. The training and testing costs are even lower than the random forest implementation.

5.4 Conclusion and Comparisons

In conclusion, the conducted analysis allows us to determine the relative strengths and weaknesses of the implemented models. The linear regression model, despite being the simplest, delivers impressive training and prediction times due to the simplicity of its calculations. However, its precision notably declines for values over 55, limiting its overall effectiveness.

On the other hand, the decision tree model's lengthy training time and prediction phase, attributable to its O(n2) training complexity and extensive traversal requirement, are significant drawbacks. Coupled with its severe error rates and negligible accuracy, it appears to be the least preferable among the models assessed.

Meanwhile, the random forest model exhibited commendable training performance, handling ~460,000 samples on 100 trees within ten minutes. However, its prediction phase is slower, as it necessitates navigating through each tree and averaging the results. Despite this, it boasts a low prediction error rate, securing its position as the second most effective model.

Lastly, the neural network model necessitates an extended training period, predominantly due to a high count of hidden layer neurons. This duration can reach up to 30 minutes for a network with two hidden layers of 200 neurons each. Nevertheless, it offers a swift prediction phase requiring just three dot products, and it stands out with the lowest prediction errors among the four models. Though increasing the neuron sizes in the hidden layers can enhance the network's performance, it should be noted that this will further elongate the training time.

Thus, while each model has its unique merits and demerits, the trade-off between training time and accuracy must be considered when selecting the most suitable model for any specific application.

6. References

[1] "About linear regression," *IBM*. [Online]. Available: https://www.ibm.com/topics/linear-regression#:~:text=Resources-,What%20is%20linear%20regression%3F,is%20called%20the%20independent%20variable. [Accessed: 26-Apr-2023].

[2] C. Tekin, Class Lecture, Topic: "Linear Regression." EEE485, Bilkent University, Ankara.

[3] R. Kwiatkowski, "Gradient descent algorithm - a deep dive," *Medium*, 13-Jul-2022. [Online]. Available: https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21#:~:text=Gradient%20descent%20(GD)%20is%20an,e.g.%20in%20a%20linear%20regression). [Accessed: 26-Apr-2023].

[4] G. M. K, "Machine learning basics: Decision tree regression," *Medium*, 18-Jul-2020. [Online]. Available: https://towardsdatascience.com/machine-learning-basics-decision-tree-regression-1d73ea003fda. [Accessed: 26-Apr-2023].

[5] "What is a Random Forest," *TIBCO*. [Online] Available: https://www.tibco.com/reference-center/what-is-a-random-forest. [Accessed: 29-Apr-2023].

[6] "What are neural networks?," *IBM*. [Online]. Available: https://www.ibm.com/topics/neural-networks#:~:text=Neural%20networks%2C%20also%20known%20as,neurons%20signal%20to%20one%20another. [Accessed: 26-Apr-2023].

[7] "Neural network," *Wikipedia*, 24-Apr-2023. [Online]. Available: https://en.wikipedia.org/wiki/Neural_network. [Accessed: 26-Apr-2023].

[8] Jeremy Jordan, "Neural networks: Representation.," *Jeremy Jordan*, 26-Jan-2018. [Online]. Available: https://www.jeremyjordan.me/intro-to-neural-networks/. [Accessed: 26-Apr-2023].

7. Appendix A: Code for “preprocess.py”

```
# [1]

# Import Libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from scipy import stats

from time import strptime, mktime, ctime

import re

from time import strptime, mktime

from dateutil.parser import isoparse

import datetime



# [2]

dataset_datetime = isoparse("2021-04-30")

print(dataset_datetime)

# Read Data

data_raw = pd.read_csv("tracks.csv", low_memory=False)

data_raw.describe()



# [3]

# Filter Relevant Columns

wanted_columns = [

    "popularity",

    "duration_ms",

    "explicit",
```

```
    "release_date",
    "danceability",
    "energy",
    "key",
    "loudness",
    "mode",
    "speechiness",
    "acousticness",
    "instrumentalness",
    "liveness",
    "valence",
    "tempo",
    "time_signature",
]
processed_data = data_raw.loc[:, wanted_columns]



# [4]
# Convert "release_date" timestamps to "days_since_release"

newest = datetime.datetime(1, 1, 1)
oldest = datetime.datetime(2023, 1, 1)

datetimes = []
times_since_release = []
for row_idx, row in processed_data.iterrows():
    datetime_str = str(row["release_date"])
```

```
    if re.search("^[0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9]$",
datetime_str):
        pass
    elif re.search("^[0-9][0-9][0-9][0-9]$", datetime_str):
        datetime_str += "-06-15"
    elif re.search("^[0-9][0-9][0-9][0-9]-[0-9][0-9]$", datetime_str):
        datetime_str += "-15"
    else:
        raise ValueError

    release_datetime = isoparse(datetime_str)
    time_since_release = (dataset_datetime -
release_datetime).total_seconds()

    # print(datetime_str)
    newest = max(newest, release_datetime)
    oldest = min(oldest, release_datetime)

    if time_since_release < 0:
        print(f"{row['name']}, {release_datetime}, {datetime_str}")

    times_since_release.append(time_since_release / (60 * 60 * 24))
processed_data["days_since_release"] = times_since_release
print(f"Newest: {newest}\nOldest: {oldest}")
processed_data.describe()



# [5]
processed_data.drop("release_date", axis=1, inplace=True)
```

```
# [6]
# Remove rows with empty values
processed_data.replace(["", None], np.nan, inplace=True)
processed_data.dropna(inplace=True)

# Display the distribution of popularity
sns.displot(processed_data["popularity"], kde=True)
processed_data.reset_index(drop=True, inplace=True)
print(processed_data.shape)
processed_data.describe()



# [7]
"""#Remove Outliers
processed_data = processed_data[processed_data["popularity"] < 50000]

sns.displot(processed_data["popularity"], kde=True)
print(processed_data.shape)
processed_data.describe()"""



# [8]
# Normalize the features
processed_data = processed_data.astype("float64")
processed_data.iloc[:, 1:] = (
    processed_data.iloc[:, 1:] - processed_data.iloc[:, 1:].mean()
) / processed_data.iloc[
```

```
    :, 1:
].std()  # z-score standardization w/ mean:0 std:1
# processed_data.iloc[:, 1:] = (processed_data-processed_data.min()) /
(processed_data.max()-processed_data.min()) # min-max normalization w/
min:0 max:1
processed_data.replace(["", None, np.nan], 0, inplace=True)



processed_data.reset_index(drop=True, inplace=True)
print(processed_data.shape)
processed_data.describe()



# [9]
# Save as another csv file
processed_data.to_csv("processed_database_2.csv", index=False)
```

8. Appendix B: Code for "linear_regression.py"

```
# [1]

# Import Libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt



# [2]

np.set_printoptions(precision=3, suppress=True)



# [3]

# Read Data

data = pd.read_csv("processed_database_2.csv", low_memory=False)


# Features List

features = data.iloc[:, 1:].columns.tolist()


data_train = data.sample(frac=0.8)

data_test = data.drop(index=data_train.index)


X_train = data_train.iloc[:, 1:].reset_index(drop=True)

X_test = data_test.iloc[:, 1:].reset_index(drop=True)


X_train.insert(0, "ones", np.ones((X_train.shape[0], 1)))

X_test.insert(0, "ones", np.ones((X_test.shape[0], 1)))


Y_train = data_train.iloc[:, :1].values
```

```
Y_test = data_test.iloc[:, :1].values


beta = np.zeros([1, len(features) + 1])


print(X_train.shape, X_test.shape, Y_train.shape, Y_test.shape,
beta.shape)



# [4]
def mse(Y_true, Y_pred):
    mse_ = np.mean((Y_true - Y_pred) ** 2)
    return mse_



# [5]
def gradient_descent(
    X_train, Y_train, beta, iterations, learning_rate, patience=10,
min_delta=1e-3
):
    n = len(X_train)
    best_cost = np.inf
    patience_counter = 0

    costs = []
    for i in range(iterations):
        Y_pred = np.dot(X_train, beta.T)

        cost = mse(Y_train, Y_pred)
        costs.append(cost)
```

```
        gradient = (1 / n) * np.dot(X_train.T, (Y_pred - Y_train))

        beta -= learning_rate * gradient.T

        if (i + 1) % 100 == 0:
            print(f"Iteration: {i+1}, Training Cost: {round(cost, 5)}")

        if cost + min_delta < best_cost:
            patience_counter = 0
            best_cost = cost
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print(f"Early stop at iteration {i}. Training Cost:
{round(cost, 5)}")
                break

    return beta, costs



# [6]
final_beta, costs = gradient_descent(
    X_train, Y_train, beta, iterations=10000, learning_rate=0.001
)
print(f"Final beta: \n{final_beta}")



# [7]
```

```
fig, ax = plt.subplots()

ax.plot(np.arange(len(costs)), costs, "r")

ax.set_xlabel("Iterations")

ax.set_ylabel("Cost")

ax.set_title("Iteration vs Cost (Linear Regression)")


fig.savefig("linear_cost")



# [8]

Y_pred_train = np.dot(X_train, final_beta.T)

residuals_train = Y_train - Y_pred_train

train_cost = mse(Y_train, Y_pred_train)

print(f"Training cost: {np.squeeze(train_cost).round(2)}")


fig, ax = plt.subplots()

ax.scatter(Y_train, Y_pred_train, (0.01))

ax.set_xlabel("Actual Value")

ax.set_ylabel("Predicted Value")

ax.set_title("Actual vs Prediction (Linear Regression, Train)")


ax.plot(np.arange(100), np.arange(100), "r")

fig.savefig("linear_train_pred")



fig, ax = plt.subplots()

ax.set_autoscale_on(False)

ax.scatter(Y_train, residuals_train, (0.01))

ax.set_xbound(-50, 100)
```

```
ax.set_ybound(-100, 100)

ax.set_xlabel("Predicted Value")

ax.set_ylabel("Residual")

ax.set_title("Predicted Value vs Residual (Linear Regression, Train)")


ax.plot(np.arange(150) - 50, np.zeros((150,)), "black")


fig.savefig("linear_train_resd")



# [9]

Y_pred_test = np.dot(X_test, final_beta.T)

residuals_test = Y_test - Y_pred_test

test_cost = mse(Y_test, Y_pred_test)

print(f"Training cost: {np.squeeze(test_cost).round(2)}")


fig, ax = plt.subplots()

ax.scatter(Y_test, Y_pred_test, (0.01))

ax.set_xlabel("Actual Value")

ax.set_ylabel("Predicted Value")

ax.set_title("Actual vs Prediction (Linear Regression, Test)")


ax.plot(np.arange(100), np.arange(100), "r")

fig.savefig("linear_test_pred")



fig, ax = plt.subplots()

ax.set_autoscale_on(False)

ax.scatter(Y_pred_test, residuals_test, (0.01))
```

```
ax.set_xbound(-50, 100)

ax.set_ybound(-100, 100)

ax.set_xlabel("Predicted Value")

ax.set_ylabel("Residual")

ax.set_title("Predicted Value vs Residual (Linear Regression, Test)")


ax.plot(np.arange(150) - 50, np.zeros((150,)), "black")


fig.savefig("linear_test_resd")
```

9. Appendix C: Code for “decision_tree.py”

```
# [1]

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from numba import jit, prange



# [2]

data = pd.read_csv("processed_database_2.csv", low_memory=False)

data.reset_index(drop=True, inplace=True)


# data = data.sample(n=100000)


# Train/Test Split

data_train = data.sample(frac=0.8)

data_test = data.drop(data_train.index)


X_train = data_train.iloc[:, 1:].to_numpy()

X_test = data_test.iloc[:, 1:].to_numpy()


Y_train = data_train.iloc[:, :1].values

Y_test = data_test.iloc[:, :1].values


print(X_train.shape, Y_train.shape, X_test.shape, Y_test.shape)



# [3]

@jit(nopython=True)
```

```
def mse(Y_true: np.ndarray, Y_pred: np.ndarray):
    return np.square(Y_true - Y_pred).mean()



# [4]
@jit(nopython=True)
def find_best_split(X: np.ndarray, Y: np.ndarray):
    best_feature, best_threshold, best_error = None, None, np.inf
    for feature_idx in prange(X.shape[1]):
        feature_values = X[:, feature_idx]
        feature_values.sort()
        for threshold in feature_values:
            Y_left = Y[X[:, feature_idx] <= threshold]
            Y_right = Y[X[:, feature_idx] > threshold]

            if len(Y_left) == 0 or len(Y_right) == 0:
                continue

            error_left = mse(Y_left, np.mean(Y_left))
            error_right = mse(Y_right, np.mean(Y_right))
            weighted_error = len(Y_left) * error_left + len(Y_right) *
error_right

            if weighted_error < best_error:
                best_error = weighted_error
                best_feature = feature_idx
                best_threshold = threshold
                break  # Break out of the loop once the best error is
found
```

```
    return best_feature, best_threshold



# [5]
class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self._depth = 0

    def _build_three(self, X: np.ndarray, Y: np.ndarray, depth=0,
verbose=False):
        self._depth = max(self._depth, depth)

        if verbose:
            print(f"Current Depth: {depth}, Max Depth: {self._depth}
", end="\r")

        if depth >= self.max_depth or len(np.unique(Y)) == 1:
            return np.mean(Y)

        feature, threshold = find_best_split(X, Y)
        if feature is None:
            return np.mean(Y)

        left_mask = X[:, feature] <= threshold
        right_mask = ~left_mask
        left_child = self._build_three(
            X[left_mask], Y[left_mask], depth + 1, verbose=verbose
```

```
        )
        right_child = self._build_three(
            X[right_mask], Y[right_mask], depth + 1, verbose=verbose
        )

        return {
            "feature": feature,
            "threshold": threshold,
            "left": left_child,
            "right": right_child,
        }

    def fit(self, X, Y, verbose=False):
        self._depth = 0
        self.tree_ = self._build_three(X, Y, verbose=verbose)

    def _predict_single(self, sample, node):
        if not isinstance(node, dict):
            return node

        feature, threshold = node["feature"], node["threshold"]

        if sample[feature] <= threshold:
            return self._predict_single(sample, node["left"])
        else:
            return self._predict_single(sample, node["right"])

    def predict(self, X):
```

```
        pred = np.array([self._predict_single(sample, self.tree_) for
sample in X])
        return pred.reshape(pred.shape[0], 1)



# [6]
regressor = DecisionTree(max_depth=np.inf)

rand_idxs = np.random.choice(X_train.shape, (1000))
regressor.fit(X_train, Y_train, verbose=True)



# [7]
Y_pred_train = regressor.predict(X_train)
residuals_train = Y_train.T - Y_pred_train
train_cost = mse(Y_train.T, Y_pred_train)
print(f"Training cost: {np.squeeze(train_cost).round(2)}")

fig, ax = plt.subplots()
ax.scatter(Y_train, Y_pred_train, (0.01))
ax.set_xlabel("Actual Value")
ax.set_ylabel("Predicted Value")
ax.set_title("Actual vs Prediction (Decision Tree, Train)")

ax.plot(np.arange(100), np.arange(100), "r")
fig.savefig("tree_train_pred")



fig, ax = plt.subplots()
```

```
ax.set_autoscale_on(False)

ax.scatter(Y_train, residuals_train, (0.01))

ax.set_xbound(-50, 100)

ax.set_ybound(-100, 100)

ax.set_xlabel("Predicted Value")

ax.set_ylabel("Residual")

ax.set_title("Predicted Value vs Residual (Decision Tree, Train)")


ax.plot(np.arange(150) - 50, np.zeros((150,)), "black")


fig.savefig("tree_train_resd")




# [8]

Y_pred_test = regressor.predict(X_train)

residuals_test = Y_test.T - Y_pred_test

test_cost = mse(Y_test.T, Y_pred_test)

print(f"Training cost: {np.squeeze(test_cost).round(2)}")


fig, ax = plt.subplots()

ax.scatter(Y_test, Y_pred_test, (0.01))

ax.set_xlabel("Actual Value")

ax.set_ylabel("Predicted Value")

ax.set_title("Actual vs Prediction (Decision Tree, Test)")


ax.plot(np.arange(100), np.arange(100), "r")

fig.savefig("tree_test_pred")
```

```
fig, ax = plt.subplots()

ax.set_autoscale_on(False)

ax.scatter(Y_pred_test, residuals_test, (0.01))

ax.set_xbound(-50, 100)

ax.set_ybound(-100, 100)

ax.set_xlabel("Predicted Value")

ax.set_ylabel("Residual")

ax.set_title("Predicted Value vs Residual (Decision Tree, Test)")


ax.plot(np.arange(150) - 50, np.zeros((150,)), "black")


fig.savefig("tree_test_resd")



# [9]
class RandomForest:
    def __init__(self, n_trees, max_depth):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self._trees = []


    def fit(self, X: np.ndarray, Y: np.ndarray, verbose=False):
        self.trees = []
        for i in range(self.n_trees):
            if verbose:
                print(f"\nTree: {i+1}/{self.n_trees}")


            sample_idxs = np.random.choice(
```

```
                np.arange(X.shape[0]), int(X.shape[0] / self.n_trees),
replace=True
            )

            tree = DecisionTree(max_depth=self.max_depth)
            X_sample = X[sample_idxs]
            Y_sample = Y[sample_idxs]
            tree.fit(X_sample, Y_sample, verbose=verbose)
            self._trees.append(tree)

    def predict(self, X):
        tree_preds = np.array([tree.predict(X) for tree in
self._trees])
        return tree_preds.mean(axis=0)



# [10]
forest = RandomForest(n_trees=100, max_depth=np.inf)
forest.fit(X_train, Y_train, verbose=True)
```

10. Appendix D: Code for “neural_network_regression.py”

```
# [1]

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from typing import Literal

from tqdm import tqdm, trange



# [2]

data = pd.read_csv("processed_database_2.csv", low_memory=False)

data.reset_index(drop=True, inplace=True)


data.drop([], axis=1)


X = data.iloc[:, 1:].to_numpy()

Y = data.iloc[:, :1].to_numpy()


print(X.shape, Y.shape)



# [3]

# Train/Test Split

# X = X.sample(n=1000).reset_index(drop=True)


idxs = np.random.choice(np.arange(X.shape[0]), int(0.8 * X.shape[0]),
replace=False)


X_train = X[idxs]
```

```
Y_train = Y[idxs]


X_test = X[~idxs]

Y_test = Y[~idxs]




# [4]

def Xavier_init(n_pre, n_post):

    w0 = np.sqrt(6 / (n_pre + n_post))

    W = np.random.uniform(-w0, w0, (n_post, n_pre))

    b = np.random.uniform(-w0, w0, (n_post, 1))

    return W, b




# [5]

def initialize_weights(n_x: int, n_h: tuple, n_y: int):

    W1, b1 = Xavier_init(n_x, n_h[0])

    W2, b2 = Xavier_init(n_h[0], n_h[1])

    W3, b3 = Xavier_init(n_h[1], n_y)


    We = {"W1": W1, "W2": W2, "W3": W3, "b1": b1, "b2": b2, "b3": b3}

    return We




# [6]

def mse(Y_true: np.ndarray, Y_pred: np.ndarray):

    return ((Y_pred - Y_true) ** 2).mean()
```

```python
# [7]
def relu(X: np.ndarray):
    A = X * (X > 0)
    dA = 1 * (X > 0)
    return A, dA



# [8]
def linear(X: np.ndarray):
    A = X
    dA = 1
    return A, dA



# [9]
def forward_propagate(We, X):
    W1 = We["W1"]
    W2 = We["W2"]
    W3 = We["W3"]
    b1 = We["b1"]
    b2 = We["b2"]
    b3 = We["b3"]

    Z1 = np.dot(W1, X.T) + b1
    A1, dA1 = relu(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2, dA2 = relu(Z2)
    Z3 = np.dot(W3, A2) + b3
    A3, dA3 = linear(Z3)
```

```
    cache = {
        "Z1": Z1,
        "A1": A1,
        "dA1": dA1,
        "Z2": Z2,
        "A2": A2,
        "dA2": dA2,
        "Z3": Z3,
        "A3": A3,
        "dA3": dA3,
    }
    return cache



# [10]
def calculate_gradients(X: np.ndarray, Y: np.ndarray, We: dict):
    N = X.shape[0]
    W1 = We["W1"]
    W2 = We["W2"]
    W3 = We["W3"]
    b1 = We["b1"]
    b2 = We["b2"]
    b3 = We["b3"]

    # Forward pass
    cache = forward_propagate(We, X)

    # Calculate loss
```

```
    J = mse(Y.T, cache["A3"])

    # Backward pass
    dZ3 = 2 * (cache["A3"] - Y.T) * cache["dA3"]
    dW3 = 1 / N * np.dot(dZ3, cache["A2"].T)
    db3 = 1 / N * np.sum(dZ3, axis=1, keepdims=True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = dA2 * cache["dA2"]  # Derivative of ReLU
    dW2 = 1 / N * np.dot(dZ2, cache["A1"].T)
    db2 = 1 / N * np.sum(dZ2, axis=1, keepdims=True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = dA1 * cache["dA1"]  # Derivative of ReLU
    dW1 = 1 / N * np.dot(dZ1, X)
    db1 = 1 / N * np.sum(dZ1, axis=1, keepdims=True)

    # Return gradients and loss
    dWe = {"dW1": dW1, "dW2": dW2, "dW3": dW3, "db1": db1, "db2": db2,
"db3": db3}
    return J, dWe


# [11]
def update_weights(We, dWe, learning_rate):
    W1 = We["W1"]
    W2 = We["W2"]
    W3 = We["W3"]
    b1 = We["b1"]
```

```
    b2 = We["b2"]
    b3 = We["b3"]

    dW1 = dWe["dW1"] * learning_rate
    dW2 = dWe["dW2"] * learning_rate
    dW3 = dWe["dW3"] * learning_rate
    db1 = dWe["db1"] * learning_rate
    db2 = dWe["db2"] * learning_rate
    db3 = dWe["db3"] * learning_rate

    W1 -= dW1
    W2 -= dW2
    W3 -= dW3
    b1 -= db1
    b2 -= db2
    b3 -= db3

    We = {"W1": W1, "W2": W2, "W3": W3, "b1": b1, "b2": b2, "b3": b3}
    return We


# [12]
def update_weights_momentum(We, dWe, mWe, learning_rate,
momentum_rate):
    W1 = We["W1"]
    W2 = We["W2"]
    W3 = We["W3"]
    b1 = We["b1"]
    b2 = We["b2"]
```

```
    b3 = We["b3"]


    dW1 = dWe["dW1"] * learning_rate + mWe["mW1"] * momentum_rate

    dW2 = dWe["dW2"] * learning_rate + mWe["mW2"] * momentum_rate

    dW3 = dWe["dW3"] * learning_rate + mWe["mW3"] * momentum_rate

    db1 = dWe["db1"] * learning_rate + mWe["mb1"] * momentum_rate

    db2 = dWe["db2"] * learning_rate + mWe["mb2"] * momentum_rate

    db3 = dWe["db3"] * learning_rate + mWe["mb3"] * momentum_rate


    W1 -= dW1

    W2 -= dW2

    W3 -= dW3

    b1 -= db1

    b2 -= db2

    b3 -= db3


    We = {"W1": W1, "W2": W2, "W3": W3, "b1": b1, "b2": b2, "b3": b3}

    mWe = {"mW1": dW1, "mW2": dW2, "mW3": dW3, "mb1": db1, "mb2": db2,
"mb3": db3}

    return We, mWe



# [13]

def update_weights_adagrad(We, dWe, mWe, learning_rate, epsilon):

    W1 = We["W1"]

    W2 = We["W2"]

    W3 = We["W3"]

    b1 = We["b1"]

    b2 = We["b2"]
```

```
    b3 = We["b3"]


    dW1 = mWe["mW1"] + np.square(dWe["dW1"])
    dW2 = mWe["mW2"] + np.square(dWe["dW2"])
    dW3 = mWe["mW3"] + np.square(dWe["dW3"])
    db1 = mWe["mb1"] + np.square(dWe["db1"])
    db2 = mWe["mb2"] + np.square(dWe["db2"])
    db3 = mWe["mb3"] + np.square(dWe["db3"])


    W1 -= (learning_rate / (np.sqrt(dW1) + epsilon)) * dWe["dW1"]
    W2 -= (learning_rate / (np.sqrt(dW2) + epsilon)) * dWe["dW2"]
    W3 -= (learning_rate / (np.sqrt(dW3) + epsilon)) * dWe["dW3"]
    b1 -= (learning_rate / (np.sqrt(db1) + epsilon)) * dWe["db1"]
    b2 -= (learning_rate / (np.sqrt(db2) + epsilon)) * dWe["db2"]
    b3 -= (learning_rate / (np.sqrt(db3) + epsilon)) * dWe["db3"]


    We = {"W1": W1, "W2": W2, "W3": W3, "b1": b1, "b2": b2, "b3": b3}
    mWe = {"mW1": dW1, "mW2": dW2, "mW3": dW3, "mb1": db1, "mb2": db2,
"mb3": db3}
    return We, mWe



# [14]
def update_weights_rmsprop(We, dWe, mWe, learning_rate, epsilon, rho):
    W1 = We["W1"]
    W2 = We["W2"]
    W3 = We["W3"]
    b1 = We["b1"]
    b2 = We["b2"]
```

```
    b3 = We["b3"]


    dW1 = (rho * mWe["mW1"]) + ((1 - rho) * np.square(dWe["dW1"]))
    dW2 = (rho * mWe["mW2"]) + ((1 - rho) * np.square(dWe["dW2"]))
    dW3 = (rho * mWe["mW3"]) + ((1 - rho) * np.square(dWe["dW3"]))
    db1 = (rho * mWe["mb1"]) + ((1 - rho) * np.square(dWe["db1"]))
    db2 = (rho * mWe["mb2"]) + ((1 - rho) * np.square(dWe["db2"]))
    db3 = (rho * mWe["mb3"]) + ((1 - rho) * np.square(dWe["db3"]))


    W1 -= (learning_rate / (np.sqrt(dW1) + epsilon)) * dWe["dW1"]
    W2 -= (learning_rate / (np.sqrt(dW2) + epsilon)) * dWe["dW2"]
    W3 -= (learning_rate / (np.sqrt(dW3) + epsilon)) * dWe["dW3"]
    b1 -= (learning_rate / (np.sqrt(db1) + epsilon)) * dWe["db1"]
    b2 -= (learning_rate / (np.sqrt(db2) + epsilon)) * dWe["db2"]
    b3 -= (learning_rate / (np.sqrt(db3) + epsilon)) * dWe["db3"]


    We = {"W1": W1, "W2": W2, "W3": W3, "b1": b1, "b2": b2, "b3": b3}
    mWe = {"mW1": dW1, "mW2": dW2, "mW3": dW3, "mb1": db1, "mb2": db2,
"mb3": db3}
    return We, mWe



# [15]
def update_weights_adam(We, dWe, mWe, vWe, learning_rate, epsilon,
rho1, rho2):
    W1 = We["W1"]
    W2 = We["W2"]
    W3 = We["W3"]
    b1 = We["b1"]
```

```
    b2 = We["b2"]

    b3 = We["b3"]


    mW1 = (rho1 * mWe["mW1"]) + ((1 - rho1) * dWe["dW1"])

    mW2 = (rho1 * mWe["mW2"]) + ((1 - rho1) * dWe["dW2"])

    mW3 = (rho1 * mWe["mW3"]) + ((1 - rho1) * dWe["dW3"])

    mb1 = (rho1 * mWe["mb1"]) + ((1 - rho1) * dWe["db1"])

    mb2 = (rho1 * mWe["mb2"]) + ((1 - rho1) * dWe["db2"])

    mb3 = (rho1 * mWe["mb3"]) + ((1 - rho1) * dWe["db3"])


    vW1 = (rho2 * vWe["vW1"]) + ((1 - rho2) * np.square(dWe["dW1"]))

    vW2 = (rho2 * vWe["vW2"]) + ((1 - rho2) * np.square(dWe["dW2"]))

    vW3 = (rho2 * vWe["vW3"]) + ((1 - rho2) * np.square(dWe["dW3"]))

    vb1 = (rho2 * vWe["vb1"]) + ((1 - rho2) * np.square(dWe["db1"]))

    vb2 = (rho2 * vWe["vb2"]) + ((1 - rho2) * np.square(dWe["db2"]))

    vb3 = (rho2 * vWe["vb3"]) + ((1 - rho2) * np.square(dWe["db3"]))


    mW1_hat = mW1 / (1 - rho1)

    mW2_hat = mW2 / (1 - rho1)

    mW3_hat = mW3 / (1 - rho1)

    mb1_hat = mb1 / (1 - rho1)

    mb2_hat = mb2 / (1 - rho1)

    mb3_hat = mb3 / (1 - rho1)


    vW1_hat = vW1 / (1 - rho2)

    vW2_hat = vW2 / (1 - rho2)

    vW3_hat = vW3 / (1 - rho2)

    vb1_hat = vb1 / (1 - rho2)

    vb2_hat = vb2 / (1 - rho2)
```

```
    vb3_hat = vb3 / (1 - rho2)


    W1 -= learning_rate * (mW1_hat / (np.sqrt(vW1_hat) + epsilon))
    W2 -= learning_rate * (mW2_hat / (np.sqrt(vW2_hat) + epsilon))
    W3 -= learning_rate * (mW3_hat / (np.sqrt(vW3_hat) + epsilon))
    b1 -= learning_rate * (mb1_hat / (np.sqrt(vb1_hat) + epsilon))
    b2 -= learning_rate * (mb2_hat / (np.sqrt(vb2_hat) + epsilon))
    b3 -= learning_rate * (mb3_hat / (np.sqrt(vb3_hat) + epsilon))


    We = {"W1": W1, "W2": W2, "W3": W3, "b1": b1, "b2": b2, "b3": b3}
    mWe = {"mW1": mW1, "mW2": mW2, "mW3": mW3, "mb1": mb1, "mb2": mb2,
"mb3": mb3}
    vWe = {"vW1": vW1, "vW2": vW2, "vW3": vW3, "vb1": vb1, "vb2": vb2,
"vb3": vb3}
    return We, mWe, vWe



# [16]
# Define training loop
def train(
    X: np.ndarray,
    Y: np.ndarray,
    We: dict,
    num_epochs: int,
    learning_rate: float,
    batch_size: int = 200,
    validation_split: float = 0.1,
    optimizer: Literal["", "none", "momentum", "adagrad", "rmsprop",
"adam"] = "none",
    momentum_rate: float = 0.9,
```

```
    epsilon: float = 1e-8,
    rho1: float = 0.9,
    rho2: float = 0.99,
    patience: int = 10,
    min_delta: float = 1e-3,
):
    patience_counter = 0
    best_valid_cost = np.inf
    mWe = {
        "mW1": 0,
        "mW2": 0,
        "mW3": 0,
        "mb1": 0,
        "mb2": 0,
        "mb3": 0,
    }
    vWe = {
        "vW1": 0,
        "vW2": 0,
        "vW3": 0,
        "vb1": 0,
        "vb2": 0,
        "vb3": 0,
    }
    if validation_split is None:
        X_train = X
        Y_train = Y
    else:
        random_idxs = np.random.choice(X.shape[0], (X.shape[0]))
```

```
        X = X[random_idxs]
        Y = Y[random_idxs]
        valid_start = int(X.shape[0] * validation_split)
        X_train = X[valid_start:]
        Y_train = Y[valid_start:]
        X_valid = X[:valid_start]
        Y_valid = Y[:valid_start]
    N = X_train.shape[0]
    if batch_size is None:
        batch_size = X_train.shape[0]
        mini_batch_count = 1
    elif isinstance(batch_size, float) and batch_size < 1:
        batch_size = int(N * batch_size)
        mini_batch_count = N // batch_size
    else:
        mini_batch_count = N // batch_size

    cache_train = forward_propagate(We, X_train[:batch_size])
    cost_train = mse(Y_train[:batch_size].T, cache_train["A3"])
    costs_train = [cost_train]

    cache_valid = forward_propagate(We, X_valid[: (batch_size // 10)])
    cost_valid = mse(Y_valid[: (batch_size // 10)].T,
cache_valid["A3"])
    costs_valid = [cost_valid]

    for epoch in range(num_epochs):
        cost_train_total = 0
        cost_valid_total = 0
```

```
        mini_batch_start = 0
        mini_batch_end = batch_size

        pbar = trange(
            mini_batch_count,
            desc=f"Epoch {epoch+1:4d}/{num_epochs}",
            ncols=130,
            leave=True,
        )
        for _ in pbar:
            mini_batch_x = X_train[mini_batch_start:mini_batch_end]
            mini_batch_y = Y_train[mini_batch_start:mini_batch_end]

            mini_batch_x_valid = X_valid[mini_batch_start // 10 :
mini_batch_end // 10]
            mini_batch_y_valid = Y_valid[mini_batch_start // 10 :
mini_batch_end // 10]

            cost_train, dWe = calculate_gradients(mini_batch_x,
mini_batch_y, We)
            cost_train_total += cost_train
            match optimizer:
                case ["" | "none"]:
                    We = update_weights(We, dWe, learning_rate)
                case "momentum":
                    We, mWe = update_weights_momentum(
                        We, dWe, mWe, learning_rate, momentum_rate
                    )
                case "adagrad":
```

```
                    We, mWe = update_weights_adagrad(
                        We, dWe, mWe, learning_rate, epsilon
                    )
                case "rmsprop":
                    We, mWe = update_weights_rmsprop(
                        We, dWe, mWe, learning_rate, epsilon, rho1
                    )
                case "adam":
                    We, mWe, vWe = update_weights_adam(
                        We, dWe, mWe, vWe, learning_rate, epsilon,
rho1, rho2
                    )

            cache_valid = forward_propagate(We, mini_batch_x_valid)
            cost_valid = mse(mini_batch_y_valid.T, cache_valid["A3"])
            cost_valid_total += cost_valid

            pbar.set_postfix_str(
                f" Training Error: {cost_train:.5f}, Validation Error:
{cost_valid:.5f}",
                refresh=False,
            )

            mini_batch_start = mini_batch_end
            mini_batch_end = min(mini_batch_end + batch_size, N)

        costs_train.append(cost_train_total / mini_batch_count)
        costs_valid.append(cost_valid_total / mini_batch_count)
```

```
        if costs_valid[-1] + min_delta < best_valid_cost:
            patience_counter = 0
            best_valid_cost = costs_valid[-1]
        else:
            patience_counter += 1
            if patience_counter >= patience:
                break

        # Print loss
        # if (epoch + 1) % 100 == 0:
        # if validation_split is None:
        #     print(f"Epoch: {epoch+1}/{num_epochs} - Training Error:
{round(J, 5)}")
        # else:
        #     print(
        #         f"Epoch: {epoch+1}/{num_epochs} - Training Error:
{round(J, 5)} - Validation Error: {round(cost_valid, 5)}            ",
end="\r"
        #     )

    return We, costs_train, costs_valid



# [17]
# Train the model
We = initialize_weights(15, (200, 200), 1)
We_final, costs_train, costs_valid = train(
    X_train,
    Y_train,
    We,
```

```
    num_epochs=1000,
    batch_size=0.01,
    learning_rate=0.001,
    optimizer="adam",
)



# [18]
fig, ax = plt.subplots()
ax.plot(np.arange(len(costs_train)), costs_train, "b",
label="Training")
ax.plot(np.arange(len(costs_valid)), costs_valid, "r",
label="Validation")
ax.set_xlabel("Iterations")
ax.set_ylabel("Cost")
ax.set_title("Iteration vs Cost (Neural Network)")


fig.savefig("neural_cost")



# [19]
cache_train = forward_propagate(We_final, X_train)
Y_pred_train = cache_train["A3"]
residuals_train = Y_train.T - Y_pred_train
train_cost = mse(Y_train.T, Y_pred_train)
print(f"Training cost: {np.squeeze(train_cost).round(2)}")


fig, ax = plt.subplots()
ax.scatter(Y_train, Y_pred_train, (0.01))
```

```
ax.set_xlabel("Actual Value")

ax.set_ylabel("Predicted Value")

ax.set_title("Actual vs Prediction (Neural Network, Train)")


ax.plot(np.arange(100), np.arange(100), "r")

fig.savefig("neural_train_pred")



fig, ax = plt.subplots()

ax.set_autoscale_on(False)

ax.scatter(Y_train, residuals_train, (0.01))

ax.set_xbound(-50, 100)

ax.set_ybound(-100, 100)

ax.set_xlabel("Predicted Value")

ax.set_ylabel("Residual")

ax.set_title("Predicted Value vs Residual (Neural Network, Train)")


ax.plot(np.arange(150) - 50, np.zeros((150,)), "black")


fig.savefig("neural_train_resd")



# [20]

cache_test = forward_propagate(We_final, X_test)

Y_pred_test = cache_test["A3"]

residuals_test = Y_test.T - Y_pred_test

test_cost = mse(Y_test.T, Y_pred_test)

print(f"Training cost: {np.squeeze(test_cost).round(2)}")
```

```
fig, ax = plt.subplots()

ax.scatter(Y_test, Y_pred_test, (0.01))

ax.set_xlabel("Actual Value")

ax.set_ylabel("Predicted Value")

ax.set_title("Actual vs Prediction (Neural Network, Test)")


ax.plot(np.arange(100), np.arange(100), "r")

fig.savefig("neural_test_pred")



fig, ax = plt.subplots()

ax.set_autoscale_on(False)

ax.scatter(Y_pred_test, residuals_test, (0.01))

ax.set_xbound(-50, 100)

ax.set_ybound(-100, 100)

ax.set_xlabel("Predicted Value")

ax.set_ylabel("Residual")

ax.set_title("Predicted Value vs Residual (Neural Network, Test)")


ax.plot(np.arange(150) - 50, np.zeros((150,)), "black")


fig.savefig("neural_test_resd")
```