BILKENT UNIVERSITY

EEE 485: STATISTICAL LEARNING AND DATA ANALYTICS

Term Project: First 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 utilising the comprehensive "Spotify HUGE database" dataset. This dataset encompasses approximately 170,000 instances, incorporating features associated with music, such as energy, danceability, tempo, and sentimentality. Our primary goal is to develop a machine learning model that leverages the Spotify dataset to accurately predict the popularity of songs.

To achieve this, we will employ a variety of regression techniques, including linear regression, decision tree regression, and neural network regression. In the final report, we will refine the decision tree regression and neural network regression methods, as well as 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 daily records of songs exceeding 8 minutes in duration that have ranked in the top 200. It contains a total of 170,633 samples, of which approximately 41.1% include sentiment analysis. To ensure data consistency and reliability, we have excluded samples lacking sentiment analysis, resulting in a final dataset of 70,489 samples.

The key attributes in this dataset include popularity, country, danceability, energy, key, loudness, mode, speechiness, acousticness, instrumentalness, liveness, valence, tempo, duration, time signature, days since release, explicitness, single/album classification, genre, and the presence of words associated with specific emotions. Both genre and country are class-based features, and the dataset conveniently incorporates one-hot encoded variables for these classes.

Popularity was determined through a combination of reverse scoring and uneven multiplication. Each sample received an intermediate score ranging from 1 to 200, with the most popular song assigned a score of 200 and the least popular assigned a score of 1. These scores were then multiplied according to the following table:

Popularity Ranking	Multiplier
#1	3
#2	2.2
#3	1.7
#4 - #10	1.3
#11 - #50	1
#51 - #100	0.85
#101 - #200	0.8

Subsequently, the scores for each song within the same country were added together. Below is a detailed explanation of some of the less obvious features:

- Danceability: This metric assesses a track's suitability for dancing, taking into account 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 characterised by faster tempo, louder volume, and higher levels of noise. 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, utilising standard Pitch Class notation to map integers to pitches. For instance, 0 corresponds to C, 1 to C♯/D ♭ , 2 to D, and so forth. 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 from which its melodic content is derived. A value of 1 represents a major modality, while 0 denotes 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 underlying mathematical concept of linear regression can be summarised by the equation:

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

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

To implement linear regression, we utilised a pre-processed dataset in conjunction with the gradient descent optimization technique. Gradient descent is an iterative algorithm that aims to find the local minimum/maximum of a given function. 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:

```
def gradient_descent(X, Y, beta, iterations, learning_rate):
    costs = np.zeros(iterations)
    for i in range(iterations):
        H = np.dot(X, beta.T)
        beta = beta - (learning_rate/len(X)) * np.sum(X * (H - Y), axis=0)
        costs[i] = compute_cost(X, Y, beta)

        if ((i+1) % 100 == 0):
            print(f"Iteration: {i+1}, Cost: {costs[i]}")
    return beta, costs
```

After implementing gradient descent, we generated the following cost vs iteration plot:

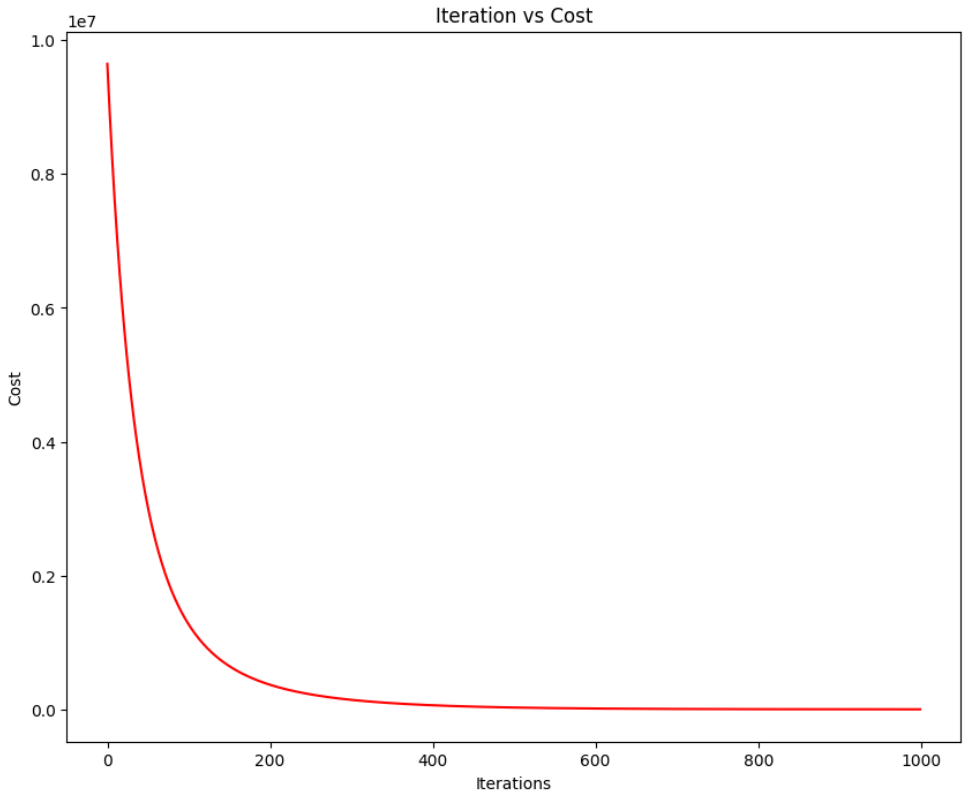

Figure 1: Cost vs Iteration Plot

The plot reveals that as the number of iterations increases, the error approaches zero. After 400 iterations, the error remains virtually unchanged and is very close to zero. Training performance is satisfactory, averaging around 10 iterations per second.

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:

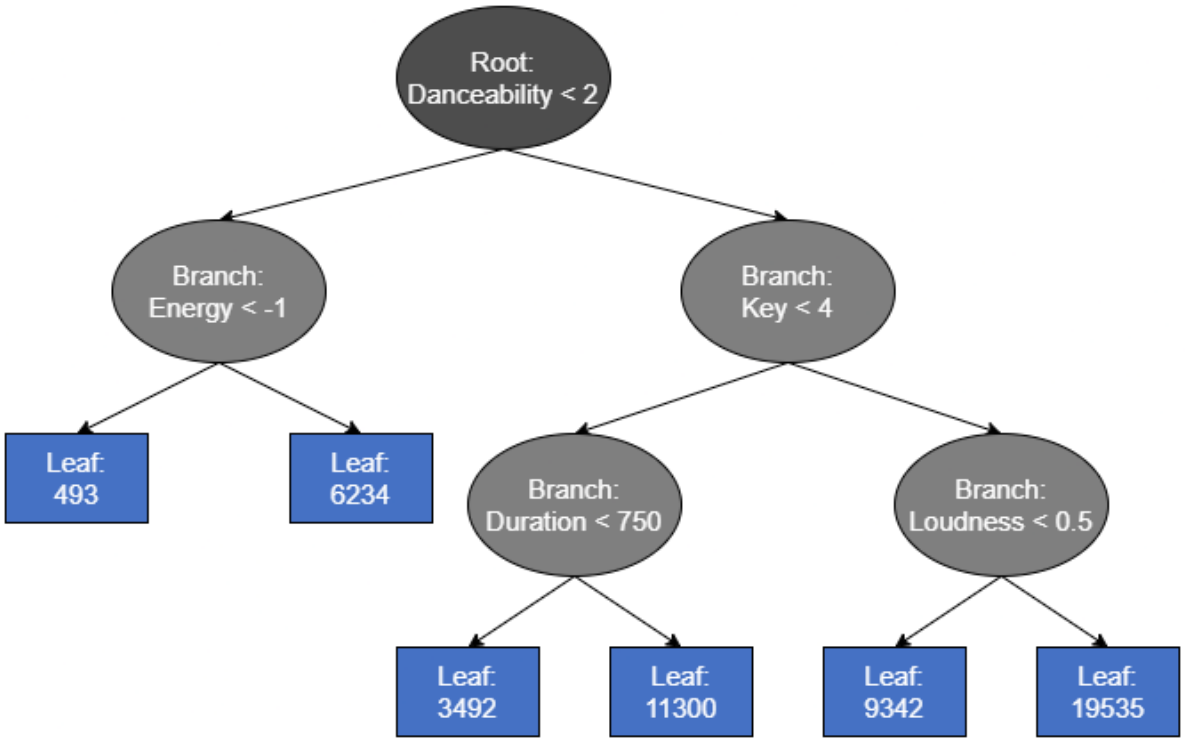

Figure 2: Decision Tree Algorithm [4]

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.

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 for training and testing, dividing it into 80% training data and 20% test data. With a subset of 5,000 samples and a maximum tree depth of 20, the mean squared error was found to be 269,505. This value is quite high for our dataset, particularly considering its highly linear nature. Examining the Y_test vs Y_prediction reveals the reason:

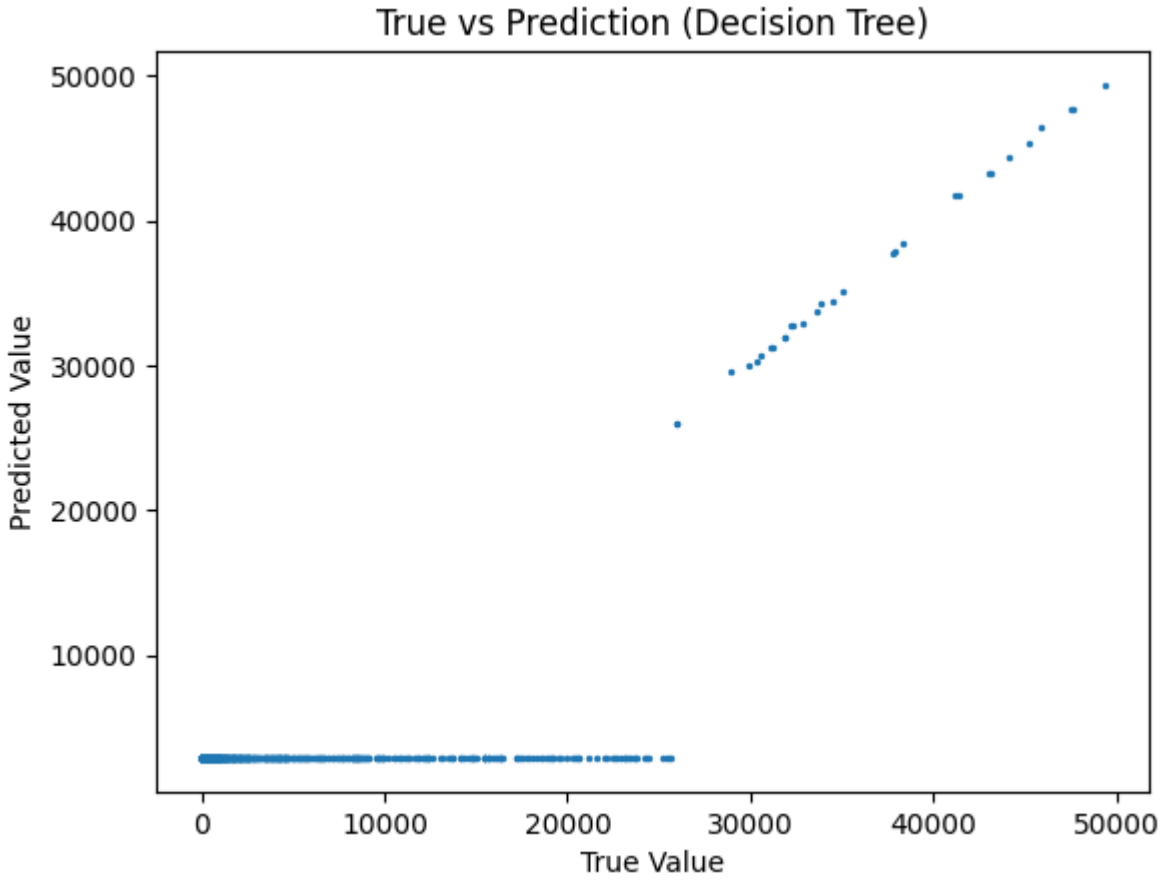

Figure 3: True Value vs Prediction (Decision Tree)

Figure 3 demonstrates that higher popularity scores have been mostly accurately identified, while samples with lower popularity scores have been clustered at a single lower prediction. This result also occurred when the lower side was more accurate. This highlights the primary limitations of decision tree regression, including the method's discrete nature. If the data is imbalanced or the tree depth is insufficient, the decision tree may completely disregard a portion of the dataset. Increasing the maximum tree depth can improve the results, but the computational cost may render it inefficient.

In the next phase of the project, we will implement a random forest regression algorithm that utilises multiple decision trees, each mapping different sections of the dataset, to obtain a more comprehensive regressor. Additionally, we are exploring ways to enhance the training performance of the decision tree, as training a decision tree with 5,000 samples and a maximum tree depth of 20 takes 15 minutes on the most powerful computer available to us.

The random forest algorithm is expected to overcome some of the limitations of individual decision trees by aggregating their outputs, resulting in a more accurate and robust prediction. This ensemble technique should help mitigate the issues caused by imbalanced data and insufficient tree depth.

3. Neural Network

In the development of our third machine learning algorithm, we employed the use of neural networks, which can be characterised as a supervised learning technique. Neural networks are composed of interconnected layers, with each layer containing neurons that connect to other neurons within these layers. Every neuron within a neural network is equipped with an activation function, while the connections between neurons involve the multiplication of the signal by a weight, followed by the addition of a bias [5]. A typical neural network comprises one input layer, one output layer, and one hidden layer. An illustration of a neural network is provided below [6]:

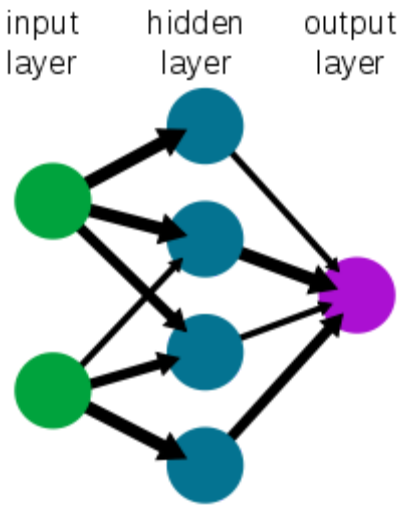

Figure 4: A Simple Neural Network

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 [7]:

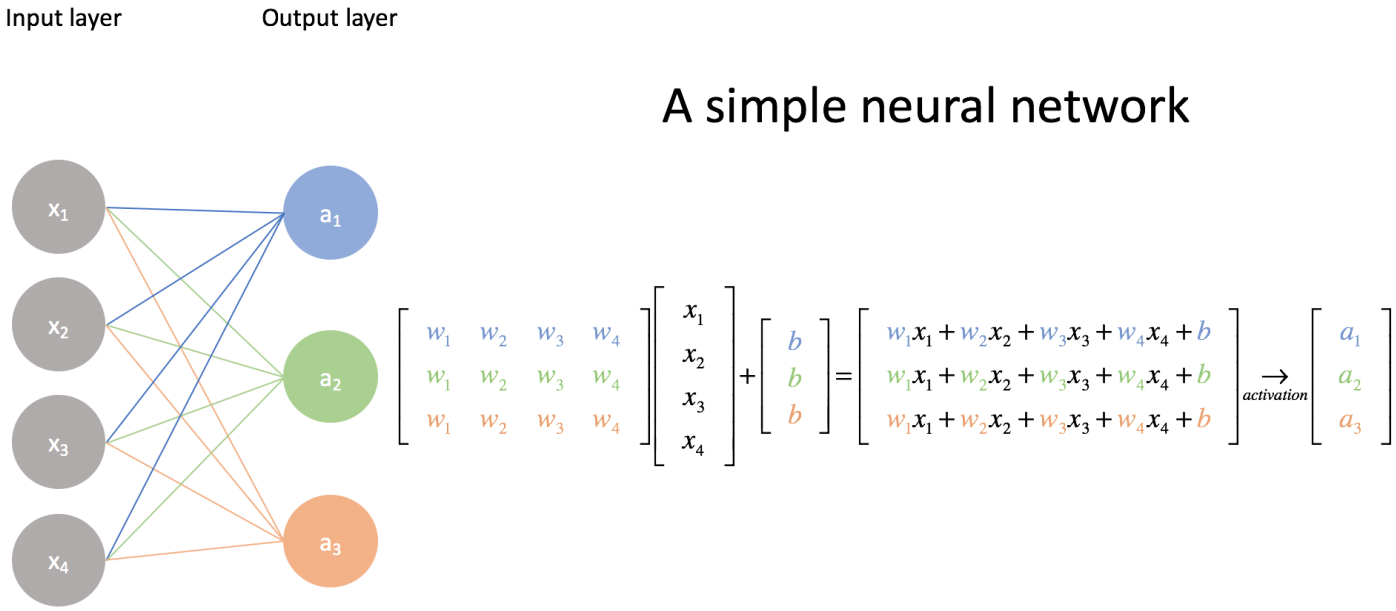

Figure 5: Matrix Representation of Neural Network

For our specific dataset, we executed neural network regression through a series of 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)

X_train = X.sample(frac=0.8).reset_index(drop=True)
X_test = X.drop(X_train.index, axis=0).reset_index(drop=True)

Y_train = Y.iloc[X_train.index, :].values
Y_test = Y.drop(X_train.index, axis=0).values
```

Subsequently, we determined the number of neurons and hidden layers before initialising the weights and biases:

```
# Configure the number of neurons and hidden layers
num_neurons = 96
num_hidden_layers = 2

# Initialize weights and biases
weights = [np.random.randn(num_neurons, num_neurons) for _ in range(num_hidden_layers)]
biases = [np.random.randn(num_neurons) for _ in range(num_hidden_layers)]
```

Next, we defined the activation function utilising the sigmoid function as:

```
def sigmoid(x):
    val = 1 / (1 + np.exp(-x))
    return val
```

We then established the forward propagation process through which each sample advances in the neural network. By defining the backward propagation function, we trained the neural network and integrated a feedback mechanism that updates the parameters in the forward propagation accordingly:

```python
def backpropagation(X, Y):
    # Forward pass
    y_pred = forward_propagation(X)

    # Calculate loss
    loss = loss_function(y_pred, Y)

    # Backward pass
    gradients = []
    for i in reversed(range(num_hidden_layers)):
        # Calculate gradient for weights
        grad_w = np.dot(X.T, (y_pred - Y) * sigmoid(np.dot(X, weights[i]) + biases[i]))

        # Calculate gradient for biases
        grad_b = np.sum((y_pred - Y) * sigmoid(np.dot(X, weights[i]) + biases[i]), axis=0)

        # Append to list of gradients
        gradients.append([grad_w, grad_b])

        # Update input for next iteration
        X = sigmoid(np.dot(X, weights[i]) + biases[i])

    # Return gradients and loss
    return gradients, loss
```

Lastly, we outlined the training loop and identified the corresponding epochs and losses as follows:

```python
# Define training loop
def train(X, Y, learning_rate=0.01, num_epochs=1000):
    for epoch in range(num_epochs):
        # Calculate gradients and loss
        gradients, loss = backpropagation(X, Y)

        # Update weights and biases
        for i in range(num_hidden_layers):
            weights[i] -= learning_rate * gradients[i][0]
            biases[i] -= learning_rate * gradients[i][1]

        # Print loss
        #if (epoch+1) % 100 == 0:
        print('Epoch {}/{} - Loss: {:.4f}'.format(epoch+1, num_epochs, loss))
```

At the current stage of the project, the neural network component is not yet operational; however, we are committed to completing it in accordance with the established timeline.

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

5. 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 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].

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

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

6. Appendix A: Code for preprocess.py

```
# Import Libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from scipy import stats

# Read Data

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

data_raw.describe()

# Filter Relevant Columns

wanted_columns = ["Popularity", "danceability", "energy", "key",
"loudness", "mode", "speechiness", "acoustics", "instrumentalness",
"liveliness", "valence", "tempo", "duration_ms", "time_signature",
"Days_since_release", "Explicit_true", "album", "compilation",
"single",

                  "bolero", "boy band", "country", "dance/electronic",
"else", "funk", "hip hop", "house", "indie", "jazz", "k-pop", "latin",
"metal", "opm", "pop", "r&b/soul", "rap", "reggae", "reggaeton",
"rock", "trap", "anger", "anticipation", "disgust", "fear", "joy",

                  "sadness", "surprise", "trust", "negative",
"positive", "n_words", "Celebrate", "Desire", "Explore", "Fun", "Hope",
"Love", "Nostalgia", "Thug", "Argentina", "Australia", "Austria",
"Belgium", "Brazil", "Canada", "Chile", "Colombia", "Costa Rica",
"Denmark",

                  "Ecuador", "Finland", "France", "Germany", "Global",
"Indonesia", "Ireland", "Italy", "Malaysia", "Mexico", "Netherlands",
"New Zealand", "Norway", "Peru", "Philippines", "Poland", "Portugal",
"Singapore", "Spain", "Sweden", "Switzerland", "Taiwan", "Turkey",

                  "UK", "USA", "Popu_max", "Top10_dummy",
"Top50_dummy"]

processed_data = data_raw.loc[:, wanted_columns]


#Remove rows with empty values
```

```
processed_data.replace(["", None], np.nan, inplace=True)

processed_data.dropna(inplace=True)


"""#replace invalid values with 0

processed_data.replace(["", None, np.nan], 0, 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()

# Normalise the features

processed_data = processed_data.astype("float64")

#processed_data.iloc[:, 1:] = (processed_data -
processed_data.mean())/processed_data.std()  # z-score standardization

processed_data.iloc[:, 1:] = (processed_data-processed_data.min()) /
(processed_data.max()-processed_data.min()) # min-max normalization

processed_data.replace(["", None, np.nan], 0, inplace=True)



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

print(processed_data.shape)

processed_data.describe()

# Save as another csv file

try:

    processed_data.to_csv("processed_database.csv", index=False)

except:

    pass
```

7. Appendix B: Code for: linear_regression.py

```
#Import Libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

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

#Read Data

data = pd.read_csv("processed_database.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)

def mse(Y_true, Y_pred):
```

```
    return np.mean((Y_true-Y_pred) ** 2, axis=0)

def gradient_descent(X, Y, beta, iterations, learning_rate):

    costs = np.zeros(iterations)

    n = len(X)

    for i in range(iterations):

        H = np.matmul(X, beta.T)

        gradient = (1/n) * np.dot(X.T, (H-Y))

        beta = beta - learning_rate * gradient.T

        costs[i] = mse(Y, H)


        if ((i+1) % 100 == 0):

            print(f"Iteration: {i+1}, Training Cost: {round(costs[i],
2)}")


        """if (costs[i] < 0.01):

            print(f"Early complete. Cost: {costs[:i+1]}")

            return beta, costs[:i+1]"""


    return beta, costs

final_beta, costs = gradient_descent(X_train, Y_train, beta,
iterations=100000, learning_rate=0.1)

print(f"Final beta: \n{final_beta}")

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

test_cost = mse(Y_test, Y_pred)

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

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")
```

```
fig_2, ax = plt.subplots()

ax.scatter(Y_test[:1000], Y_pred[:1000], (2))

ax.set_xlabel("True Value")

ax.set_ylabel("Predicted Value")

ax.set_title("True vs Prediction")
```

8. Appendix C: Code for decision_tree_regression.py

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

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

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


data = data.sample(n=1000)


#Train/Test Split

data_train = data.sample(frac=0.8)

data_test = data.drop(data_train.index)


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

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


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

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

def mse(Y, Y_pred):

    return np.mean((Y-Y_pred) ** 2)

def find_best_split(X, Y):

    best_feature, best_threshold, best_error = None, None, float("inf")

    for feature_label in X.columns.tolist():

        X_sorted = X.sort_values(feature_label)

        feature_values = X_sorted[feature_label]

        for threshold in feature_values:

            Y_left = Y[X_sorted[feature_label] <= threshold]

            Y_right = Y[X_sorted[feature_label] > 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_label
                best_threshold = threshold

    return best_feature, best_threshold
class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self._depth = 0

    def _build_three(self, X, Y, depth=0, verbose=False):
        if (verbose):
            print(depth, end=" ")

        if (self._depth < depth):
            self._depth = depth

        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.iterrows()])

        return pred.reshape(pred.shape[0], 1)

regressor = DecisionTree(max_depth=10)

regressor.fit(X_train, Y_train, verbose=True)

Y_pred = regressor.predict(X_test)


print(f"Mean Squared Error: {mse(Y_test, Y_pred)}")

fig, ax = plt.subplots()

ax.scatter(Y_test[:1000], Y_pred[:1000], (2))

ax.set_xlabel("True Value")

ax.set_ylabel("Predicted Value")

ax.set_title("True vs Prediction (Decision Tree)")


class RandomForest:

    def __init__(self, n_trees=10, max_depth=10):

        self.n_trees = n_trees

        self.max_depth = max_depth

        self._trees = []


    def fit(self, X, Y, verbose=False):

        self.trees = []

        for i in range(self.n_trees):

            if (verbose):

                print(f"\nTree: {i+1}/{self.n_trees}\n")


            tree = DecisionTree(max_depth=self.max_depth)

            idxs = np.random.choice(X.shape[0], X.shape[0],
replace=True)

            X_sample = X.iloc[idxs, :]
```

```
            Y_sample = Y[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()


forest = RandomForest(n_trees=10, max_depth=10)

forest.fit(X_train, Y_train, verbose=True)

Y_pred = forest.predict(X_test)

print(f"Mean Squared Error: {mse(Y_test, Y_pred)}")

fig_2, ax = plt.subplots()

ax.scatter(Y_test[:1000], Y_pred[:1000], (2))

ax.set_xlabel("True Value")

ax.set_ylabel("Predicted Value")

ax.set_title("True vs Prediction (Random Forest)")
```

9. Appendix D: Code for neural_network_regression.py

```
import pandas as pd

import numpy as np

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

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


data.drop([], axis=1)


X = data.iloc[:, 1:]

Y = data.iloc[:, 0:1]


X.shape, Y.shape

#Train/Test Split

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


X_train = X.sample(frac=0.8).reset_index(drop=True)

X_test = X.drop(X_train.index, axis=0).reset_index(drop=True)


Y_train = Y.iloc[X_train.index, :].values

Y_test = Y.drop(X_train.index, axis=0).values

# Configure the number of neurons and hidden layers

num_neurons = 96

num_hidden_layers = 2


# Initialize weights and biases

weights = [np.random.randn(num_neurons, num_neurons) for _ in
range(num_hidden_layers)]

biases = [np.random.randn(num_neurons) for _ in
range(num_hidden_layers)]

# Define activation function
```

```
def sigmoid(x):
    val = 1 / (1 + np.exp(-x))
    return val
# Define forward propagation
def forward_propagation(X):
    # Input layer
    layer_input = X

    # Hidden layers
    for i in range(num_hidden_layers):
        layer_output = sigmoid(np.dot(layer_input, weights[i]) +
biases[i])
        layer_input = layer_output

    # Output layer
    output = np.dot(layer_input, weights[-1]) + biases[-1]

    return output
# Define loss function
def loss_function(Y_pred, Y_true):
    return np.mean((Y_pred - Y_true)**2)
# Define backpropagation
def backpropagation(X, Y):
    # Forward pass
    y_pred = forward_propagation(X)

    # Calculate loss
    loss = loss_function(y_pred, Y)
```

```
    # Backward pass
    gradients = []
    for i in reversed(range(num_hidden_layers)):
        # Calculate gradient for weights
        grad_w = np.dot(X.T, (y_pred - Y) * sigmoid(np.dot(X,
weights[i]) + biases[i]))

        # Calculate gradient for biases
        grad_b = np.sum((y_pred - Y) * sigmoid(np.dot(X, weights[i]) +
biases[i]), axis=0)

        # Append to list of gradients
        gradients.append([grad_w, grad_b])

        # Update input for next iteration
        X = sigmoid(np.dot(X, weights[i]) + biases[i])

    # Return gradients and loss
    return gradients, loss
# Define training loop
def train(X, Y, learning_rate=0.01, num_epochs=1000):
    for epoch in range(num_epochs):
        # Calculate gradients and loss
        gradients, loss = backpropagation(X, Y)

        # Update weights and biases
        for i in range(num_hidden_layers):
            weights[i] -= learning_rate * gradients[i][0]
            biases[i] -= learning_rate * gradients[i][1]
```

```
        # Print loss
        #if (epoch+1) % 100 == 0:
        print('Epoch {}/{} - Loss: {:.4f}'.format(epoch+1, num_epochs,
loss))
# Train the model
train(X_train, Y_train)
result = forward_propagation(X_test, Y_test)
result
```