EEE 485 Statistical Learning and Data Analytics

Spring 2023-2024 Term Project Final Report:
Emotional Analysis of a Musical Piece with Regression

Full Name: Arda Iynem
Student ID: 22002717

May 12, 2024

Contents

1 Introduction to the Problem

The objective of this project is to predict the emotional valence of musical pieces on a scale ranging from 0 to 1, where 0 represents negative emotions such as melancholy and anger, and 1 represents positive emotions like happiness and joy. To accomplish this, various machine learning regression methods will be employed, including Linear Regression, Ridge Regression, Lasso Regression, Neural Network, and K-Nearest Neighbors Regression. Utilizing the Spotify API Dataset, which contains numerical data on characteristic elements and valence values for thousands of musical pieces, allows for a comprehensive analysis. By employing different algorithms, each with its unique capabilities in capturing underlying patterns, this regression task aims to provide insights into the emotional content of musical compositions.

2 Dataset Analysis

Spotify API provides the metrics related to musical characteristics of every piece on Spotify in JSON format. I will use the dataset on Kaggle that contains metrics for nearly 100.000 pieces from 125 different genres and in CSV format [1]. The musical characteristic metrics that Spotify provides can be listed as below [2]. Spotify obtained these accurate valence values by utilizing advanced machine learning algorithms and consulting the expertise of music experts [3], [4].

- **duration_ms**: The track length in milliseconds. (milliseconds)
- **explicit**: Whether or not the track has explicit lyrics. (0 or 1)
- **danceability**: How suitable a track is for dancing. (Range: 0 - 1)
- **energy**: Energy represents a perceptual measure of intensity and activity. (Range: 0 - 1)
- **key**: The key the track is in, if no key was detected, the value is -1. (Integers map to pitches using standard Pitch Class notation)
- **loudness**: The overall loudness of a track in decibels. (dB)
- **mode**: Mode indicates the modality (major or minor) of a track (0 or 1)
- **speechiness**: Speechiness detects the presence of spoken words. (Range: 0 - 1)
- **acousticness**: A confidence measure whether the track is acoustic. (Range: 0 - 1)
- **instrumentalness**: Predicts whether a track contains no vocals. (Range: 0 - 1)
- **liveness**: Detects the presence of an audience in the recording. (Range: 0 - 1)
- **valence**: Musical positiveness (happiness) conveyed by a track. (Range: 0 - 1)
- **tempo**: The overall estimated tempo of a track in beats per minute. (Average BPM)
- **time_signature**: An estimated time signature. (3 to 7 indicating 3/4 to 7/4)

The dataset initially (before preprocessing operations) has the shape (114000 x 20) representing instance amount and feature amount respectively. Since only the features listed above are related the our problem we eliminate irrelevant features (such as track id, artist etc.). Thus, after keeping the relevant (musical characteristic features) columns, the dataset has the shape (114000 x 15). The histogram of the features (Appendix A, Fig. 1) in the dataset depicts the rather balanced distribution the valence (response) feature. As shown above most of the features are within [0, 1] range, yet there are some features that represents categorical variables and different ranges which needs to be preprocessed. The correlation matrix (Appendix A, Fig. 2) suggests valence feature has relatively high correlation with danceability, energy, and loudness features which is intuitive; and valence is has relatively strong negative correlation with instrumentalness and acousticness which is also intuitive considering my own subjective experience. Some of the key correlations are visualized (Appendix A, Fig. 3), and some statistical information about each feature is tabularized (Appendix A, Fig. 4) to describe the dataset. The dataset will be split into train, validation and test dataset.

2.1 Preprocessing

One-hot Encoding: Converted categorical variables (key and time_signature features) into binary vectors, crucial for machine learning algorithms, as they require numerical input. It's used over labeling to avoid imposing an ordinal relationship on categorical data.

Shuffling: Randomizes data instances' order, essential for preventing sequence-based patterns from influencing learning.

Standardization (Standard Scaling): Scales features to have a mean of 0 and a standard deviation of 1, useful for models sensitive to feature scales (j represents j^{th} feature).

$$\frac{x_j - \mu_j}{\sigma_j} \tag{1}$$

Normalization (Min-Max Scaling): Scales features to a range between 0 and 1, aiding models where magnitude matters, but distribution shape doesn't (j represents j^{th} feature).

$$\frac{x_j - x_{\text{min}_j}}{x_{\text{max}_j} - x_{\text{min}_j}} \tag{2}$$

PCA (Principal Component Analysis): PCA is a dimensionality reduction technique used to transform high-dimensional data into a lower-dimensional space while preserving most of its variance. This is achieved by finding the principal components, which are orthogonal vectors that represent the directions of maximum variance in the data.

To find the principal components, we first compute the covariance matrix $\mathbf{C}$ of the standardized data. Then, we calculate the eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ of $\mathbf{C}$.

The proportion of variance explained (PVE) by each principal component can be calculated as the ratio of its eigenvalue to the sum of all eigenvalues (Appendix A, Fig. 5):

$$\text{PVE}_i = \frac{\lambda_i}{\sum_{i=1}^{p} \lambda_i} \tag{3}$$

where p is the total number of principal components.

We then sort the eigenvalues in descending order and select the top k eigenvectors corresponding to the largest eigenvalues to form the principal components. To choose the number of principal components, we often look at the cumulative explained variance. We select the smallest number of principal components that explains a high percentage of the total variance in the data. This can be visualized using a scree plot or by calculating the cumulative explained variance:

$$\text{Cumulative Explained Variance}_k = \sum_{i=1}^{k} \text{PVE}_i \tag{4}$$

We choose the smallest k such that Cumulative Explained Variance$_k$ exceeds a predefined threshold (Appendix A, Fig. 6), typically 95%. After PCA, the dataset's dimensions are reduced to a specified number of principal components, providing a more compact representation of the data while retaining most of its variance.

After applying the mentioned preprocessing methods (expect PCA as it will be applied as a special case) dataset obtains it's final state (Appendix A, Fig. 7) with shape (114000 rows x 29 columns), which is ready for the training step.

3 Training Methodology

3.1 Linear Regression

Linear regression is selected for its simplicity and interpretability, making it well-suited for providing insights into the linear relationships. By analyzing the coefficients of the regression model, we can identify which features have the most significant impact on valence prediction. Its straightforward nature also allows for easy comparison with more complex methods, serving as a baseline model for evaluating predictive performance.

In linear regression, we model the relationship between the dependent variable y and the independent variables $\mathbf{x}$ using the following equation, where β_i are the weights and ϵ is the zero mean noise (error):

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \epsilon = \boldsymbol{\beta}^T \mathbf{x} + \epsilon \tag{5}$$

3.1.1 Likelihood Function

The likelihood function $\mathcal{L}(\boldsymbol{\beta})$ for linear regression is calculated as below, where $f(y_i|\mathbf{x}_i, \beta)$ is the probability density function assuming the errors ϵ_i are normally distributed with mean 0 and variance σ^2:

$$f(y_i|\mathbf{x}_i, \boldsymbol{\beta}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2}{2\sigma^2}\right) \tag{6}$$

$$\mathcal{L}(\boldsymbol{\beta}) = \prod_{i=1}^{n} f(y_i|\mathbf{x}_i, \boldsymbol{\beta}) \tag{7}$$

$$\mathcal{L}(\boldsymbol{\beta}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2}{2\sigma^2}\right) \tag{8}$$

3.1.2 Maximum Likelihood Estimation

The maximum likelihood estimation (MLE) seeks to find the values of the parameters $\boldsymbol{\beta}$ that maximize the likelihood function $\mathcal{L}(\boldsymbol{\beta})$. It is more convenient to maximize the log-likelihood function:

$$\ell(\boldsymbol{\beta}) = \log(\mathcal{L}(\boldsymbol{\beta})) = \sum_{i=1}^{n} \log(f(y_i|\mathbf{x}_i, \boldsymbol{\beta})) \tag{9}$$

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(-\frac{1}{2}\log(2\pi\sigma^2) - \frac{(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2}{2\sigma^2}\right) \tag{10}$$

The Residual Sum of Squares (RSS) is defined as in (Eq. 11) and one can see that minimizing $RSS(\boldsymbol{\beta})$ and maximizing $\ell(\boldsymbol{\beta})$ results in same estimations for parameters $\boldsymbol{\beta}$:

$$RSS(\boldsymbol{\beta}) = \sum_{i=1}^{n} (y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \tag{11}$$

$$\text{argmin}_{\boldsymbol{\beta}} RSS(\boldsymbol{\beta}) = \text{argmax}_{\boldsymbol{\beta}}(\ell(\boldsymbol{\beta})) \tag{12}$$

Therefore, our loss function to minimize in order to find estimated weights is:

$$L_{OLS}(\boldsymbol{\beta}) = RSS(\boldsymbol{\beta}) \tag{13}$$

$$\hat{\boldsymbol{\beta}} = argmin_{\boldsymbol{\beta}} RSS(\boldsymbol{\beta}) \tag{14}$$

3.1.3 Methods for Finding Weights

Normal Equations The normal equations provide a closed-form solution for finding the optimal weights β. We start by defining the design matrix $\mathbf{X}$:

$$\mathbf{X} = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix} \tag{15}$$

$$\nabla_{\boldsymbol{\beta}} RSS(\boldsymbol{\beta}) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = 0 \tag{16}$$

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{17}$$

Gradient Descent Gradient descent is an iterative optimization algorithm that finds the optimal weights $\hat{\boldsymbol{\beta}}$ by iteratively updating them in the direction of the negative gradient of cost function $J(\boldsymbol{\beta})$ until convergence, where α is a hyper-parameter known as learning rate:

$$J(\hat{\boldsymbol{\beta}}) = \frac{1}{n} L_{OLS}(\hat{\boldsymbol{\beta}}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\hat{\boldsymbol{\beta}})^2 \tag{18}$$

$$\nabla J(\hat{\boldsymbol{\beta}}) = -\frac{2}{n}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) \tag{19}$$

$$\hat{\boldsymbol{\beta}}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \alpha \nabla J(\hat{\boldsymbol{\beta}}^{(t)}) \tag{20}$$

3.2 Ridge Regression

Ridge regression is a regularization technique that adds a penalty term to the linear regression cost function (Eq. 24), which helps mitigate overfitting by shrinking the coefficients towards zero. From a probabilistic perspective, ridge performs a maximum a posterior (MAP) estimation (see Section 3.2.1) instead of MLE; resulting in a regularized loss function (Eq. 24). Ridge regression is chosen due to its effectiveness in handling large feature spaces. Its regularization term helps mitigate overfitting. regression offers robustness in identifying the most influential features for predicting emotional valence.

3.2.1 Maximum a Posteriori Estimation

While the likelihood $p(D|\boldsymbol{\beta}) = \mathcal{L}(\boldsymbol{\beta})$ is same with ordinary linear regression, the prior $p(\boldsymbol{\beta})$ where $\beta_j \sim \mathcal{N}\left(0, \frac{\sigma^2}{\lambda}\right)$ causes modification on the posterior where σ^2 is the variance of the error term ϵ, and λ is the regularization parameter:

$$p(\boldsymbol{\beta}|D) \propto p(D|\boldsymbol{\beta})p(\boldsymbol{\beta}) \tag{21}$$

$$p(\boldsymbol{\beta}|D) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2}{2\sigma^2}\right) \cdot \frac{1}{\sqrt{2\pi\lambda^2}} \exp\left(-\frac{\boldsymbol{\beta}^T\boldsymbol{\beta}}{2\lambda^2}\right) \tag{22}$$

$$-\log(p(\boldsymbol{\beta}|D)) \propto \frac{1}{2\sigma^2}\left(\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 + \lambda\sum_{j=1}^{p}\beta_j^2\right) - nlog(\frac{1}{\sqrt{2\pi}\sigma}) - plog\left(\frac{\sqrt{\lambda}}{\sqrt{2\pi}\sigma}\right) \tag{23}$$

$$L_{ridge}(\boldsymbol{\beta}, \lambda) = \sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 + \lambda\sum_{j=1}^{p}\beta_j^2 = RSS(\boldsymbol{\beta}) + \lambda\|\boldsymbol{\beta}\|_2^2 \tag{24}$$

$$\hat{\boldsymbol{\beta}} = argmax_{\boldsymbol{\beta}}p(\boldsymbol{\beta}|D) = argmin_{\boldsymbol{\beta}}L_{ridge}(\boldsymbol{\beta}, \lambda) \tag{25}$$

3.2.2 Methods for Finding Weights

Normal Equations λ is the regularization parameter and **I** is the identity matrix.

$$\hat{\boldsymbol{\beta}}_{\text{ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{26}$$

Gradient Descent In the context of gradient descent, the cost function J for ridge regression is defined as the sum of the squared errors plus the regularization term. It is given by:

$$J(\hat{\boldsymbol{\beta}}) = \frac{1}{n}L_{ridge}(\hat{\boldsymbol{\beta}}, \lambda) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\hat{\boldsymbol{\beta}})^2 + \lambda\|\hat{\boldsymbol{\beta}}\|_2^2 \tag{27}$$

$$\nabla J(\hat{\boldsymbol{\beta}}) = -\frac{2}{n}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) + 2\lambda\hat{\boldsymbol{\beta}} \tag{28}$$

$$\hat{\boldsymbol{\beta}}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \alpha\nabla J(\hat{\boldsymbol{\beta}}^{(t)}) \tag{29}$$

3.3 Lasso Regression

Lasso regression is another regularization technique similar to Ridge regression but uses a different penalty term (Eq. 33) that encourages sparsity by forcing some coefficients to be exactly zero resulting in an automatic feature selection. Similar to ridge, lasso performs MAP estimation with a different prior (See Section 3.3.1) resulting in a different regularized loss function (Eq. 33). Lasso regression is employed for its capability in feature selection and avoiding overfitting due to it's regularization term, particularly in datasets with a large number of features. This feature selection property not only reduces model complexity but also enhances interpretability by focusing on the most relevant features.

3.3.1 Maximum a Posteriori Estimation

While the likelihood $p(D|\boldsymbol{\beta}) = \mathcal{L}(\boldsymbol{\beta})$ is the same as ordinary linear regression, the prior $p(\boldsymbol{\beta})$ where $\beta_j \sim Lap\left(0, \frac{2\sigma^2}{\lambda}\right)$ causes modification on the posterior where σ^2 is the variance of the error term ϵ, and λ is the regularization parameter:

$$p(\boldsymbol{\beta}|D) \propto p(D|\boldsymbol{\beta})p(\boldsymbol{\beta}) \tag{30}$$

$$p(\boldsymbol{\beta}|D) = \prod_{i=1}^{n}\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2}{2\sigma^2}\right) \cdot \prod_{j=1}^{p}\frac{\lambda}{4\sigma^2}\exp\left(-\frac{\lambda}{2\sigma^2}|\beta_j|\right) \tag{31}$$

$$-\log(p(\boldsymbol{\beta}|D)) \propto \frac{1}{2\sigma^2}\left(\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 + \lambda\sum_{j=1}^{p}|\beta_j|\right) - nlog(\frac{1}{\sqrt{2\pi}\sigma}) - plog\left(\frac{\lambda}{4\sigma^2}\right) \tag{32}$$

$$L_{lasso}(\boldsymbol{\beta}, \lambda) = \sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 + \lambda\sum_{j=1}^{p}|\beta_j| = RSS(\boldsymbol{\beta}) + \lambda\sum_{j=1}^{p}|\beta_j| \tag{33}$$

$$\hat{\boldsymbol{\beta}} = argmax_{\boldsymbol{\beta}}p(\boldsymbol{\beta}|D) = argmin_{\boldsymbol{\beta}}L_{lasso}(\boldsymbol{\beta}, \lambda) \tag{34}$$

3.3.2 Gradient Descent

In contrast to other methods, Lasso regression has no closed-form solution. Thus, the MAP estimation can be only found by gradient descent:

$$J(\hat{\boldsymbol{\beta}}) = \frac{1}{n} L_{lasso}(\hat{\boldsymbol{\beta}}, \lambda) = \frac{1}{n} \left(\sum_{i=1}^{n} (y_i - \mathbf{x}_i^T \hat{\boldsymbol{\beta}})^2 + \lambda \sum_{j=1}^{p} |\hat{\beta}_j| \right) \tag{35}$$

$$\nabla J(\hat{\boldsymbol{\beta}}) = -\frac{2}{n} \mathbf{X}^T (\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) + \lambda \cdot \text{sign}(\hat{\boldsymbol{\beta}}) \tag{36}$$

$$\hat{\boldsymbol{\beta}}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \alpha \nabla J(\hat{\boldsymbol{\beta}}^{(t)}) \tag{37}$$

3.4 Neural Network (MLP) Regression

Neural networks are chosen for their ability to capture complex non-linear relationships and ability of handling large datasets efficiently. Their flexible architecture allows for learning complex patterns in the data, leading to higher predictive performance compared to traditional linear models. Neural networks for regression tasks involve training a network to predict continuous target variables given a set of input features.

3.4.1 Forward Propagation

Forward propagation is the process of computing the output of the neural network given a set of input features. Given an input feature matrix $\mathbf{X}$, where each row represents a sample, the output of a neural network with L layers can be computed as follows, where $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the weights and biases of layer l, and g is the activation function for $l = 1, 2, \ldots, L-1$

$$\mathbf{A}^{(0)} = \mathbf{X} \tag{38}$$

$$\mathbf{Z}^{(l)} = \mathbf{W}^{(l)} \mathbf{A}^{(l-1)} + \mathbf{b}^{(l)} \tag{39}$$

$$\mathbf{A}^{(l)} = g(\mathbf{Z}^{(l)}) \tag{40}$$

$$\hat{\mathbf{y}} = \mathbf{A}^{(L)} \tag{41}$$

3.4.2 Backpropagation

Backpropagation is the process of computing the gradients of the cost function with respect to the weights and biases of the neural network. It involves propagating the error backwards through the network, layer by layer, and applying the chain rule to compute the gradients. m is the number of samples, $\mathbf{dZ}^{(l)}$ is the delta term for layer l, $\mathbf{A}^{(l-1)}$ is the activation of the previous layer, and $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the weights and biases of layer l and $\odot$ denotes element-wise multiplication.

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \frac{1}{m} \mathbf{dZ}^{(l)} (\mathbf{A}^{(l-1)})^T \tag{42}$$

$$\frac{\partial J}{\partial \mathbf{b}^{(l)}} = \frac{1}{m} \sum_{i=1}^{m} \mathbf{dZ}^{(l)} \tag{43}$$

$$\mathbf{dZ}^{(l)} = (\mathbf{W}^{(l+1)})^T \mathbf{dZ}^{(l+1)} \odot g'(\mathbf{Z}^{(l)}) \tag{44}$$

3.4.3 Gradient Descent

Gradient descent is repeated for multiple iterations or until convergence, where α is the learning rate.

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \alpha \frac{\partial J}{\partial \mathbf{W}^{(l)}} \tag{45}$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \alpha \frac{\partial J}{\partial \mathbf{b}^{(l)}} \tag{46}$$

3.4.4 Optimization Techniques for Gradients

Gradient descent is a fundamental optimization algorithm for training neural networks. However, various enhancements and modifications have been proposed to improve its efficiency and convergence speed. Here, we discuss several popular gradient techniques and their equations (Note that weight vector is shown with $\boldsymbol{\theta}$ instead of $\boldsymbol{\beta}$ as β generally represents some terms related to these techniques):

- **AdaGrad (Adaptive Gradient Algorithm)**: AdaGrad adapts the learning rate of each parameter based on the historical gradients. It performs larger updates for infrequent parameters and smaller updates for frequent parameters. The term $\mathbf{G}^{(t)}$ is an accumulated sum of squares of past gradients, and ϵ is a small constant to prevent division by zero.

$$\mathbf{G}^{(t)} = \mathbf{G}^{(t-1)} + (\nabla_\theta J(\theta^{(t)}))^2 \tag{47}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\mathbf{G}^{(t)} + \epsilon}} \cdot \nabla_\theta J(\theta^{(t)}) \tag{48}$$

- **Adam (Adaptive Moment Estimation)**: Adam combines the advantages of both AdaGrad and RMSProp by using adaptive learning rates and momentum. It computes adaptive learning rates for each parameter and includes bias correction. $\mathbf{m}^{(t)}$ and $\mathbf{v}^{(t)}$ are exponentially decaying moving averages of gradients and squared gradients, respectively, and β_1 and β_2 are decay rates for the moment estimates.

$$\mathbf{m}^{(t)} = \beta_1 \mathbf{m}^{(t-1)} + (1 - \beta_1) \nabla_\theta J(\theta^{(t)}) \tag{49}$$

$$\mathbf{v}^{(t)} = \beta_2 \mathbf{v}^{(t-1)} + (1 - \beta_2) (\nabla_\theta J(\theta^{(t)}))^2 \tag{50}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\mathbf{v}^{(t)} + \epsilon}} \cdot \mathbf{m}^{(t)} \tag{51}$$

- **AMSGrad (Adaptive Moment Estimation with AMSGrad)**: AMSGrad is a modification of Adam that prevents the learning rate from decreasing drastically. It ensures that the past gradients do not dominate the current update direction. It uses a modified update rule for the squared gradients $\mathbf{v}^{(t)}$, preventing its decay.

$$\mathbf{v}^{(t)} = \max(\mathbf{v}^{(t-1)}, (\nabla_\theta J(\theta^{(t)}))^2) \tag{52}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\mathbf{v}^{(t)} + \epsilon}} \cdot \mathbf{m}^{(t)} \tag{53}$$

- **Momentum**: Momentum accelerates SGD in the relevant direction and dampens oscillations. It accumulates a momentum term $\mathbf{v}^{(t)}$ to update the parameters. The term β is the momentum coefficient, controlling the contribution of the previous gradient direction to the current update.

$$\mathbf{v}^{(t)} = \beta \mathbf{v}^{(t-1)} + (1 - \beta) \nabla_\theta J(\theta^{(t)}) \tag{54}$$

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \mathbf{v}^{(t)} \tag{55}$$

- **Nesterov Accelerated Gradient (NAG)**: Nesterov Accelerated Gradient is an improvement over traditional momentum by considering the momentum term ahead of the current parameter update. It computes the gradient at the "look-ahead" point, adjusting the momentum term accordingly.

$$\mathbf{v}^{(t)} = \beta \mathbf{v}^{(t-1)} + (1-\beta)\nabla_\theta J(\theta^{(t)} - \beta \mathbf{v}^{(t-1)}) \tag{56}$$

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \mathbf{v}^{(t)} \tag{57}$$

- **RMSProp (Root Mean Square Propagation)**: RMSProp adapts the learning rates based on the average of recent magnitudes of gradients. It prevents the learning rates from decreasing too rapidly for frequently occurring features. $\mathbf{v}^{(t)}$ is an exponentially decaying moving average of squared gradients.

$$\mathbf{v}^{(t)} = \beta \mathbf{v}^{(t-1)} + (1-\beta)(\nabla_\theta J(\theta^{(t)}))^2 \tag{58}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{\eta}{\sqrt{\mathbf{v}^{(t)} + \epsilon}} \cdot \nabla_\theta J(\theta^{(t)}) \tag{59}$$

- **Stochastic Gradient Descent (SGD)**: SGD updates the parameters using the gradient of the loss function with respect to a randomly chosen sample from training samples. It computes the gradient using only a sample of data at each iteration, making it faster than full-batch gradient descent.

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla_\theta J(\theta^{(t)}) \tag{60}$$

- **Mini-Batch Gradient Descent**: Mini-batch gradient descent combines the advantages of SGD and full-batch gradient descent by updating the parameters using a small random subset of the training data. It strikes a balance between the efficiency of SGD and the stability of full-batch gradient descent.

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta J(\theta^{(t)}) \tag{61}$$

3.5 K-Nearest Neighbors (KNN) Regression

K-Nearest Neighbors regression is selected for its non-parametric nature, making minimal assumptions about the underlying data distribution. This method is well-suited for exploring complex relationships and local patterns where traditional linear models may not suffice. KNN regression is a simple yet effective algorithm used for regression tasks. It predicts the target value for a new data point by averaging the target values of the K nearest neighbors in the feature space.

3.5.1 Prediction

Given a new data point $\mathbf{x}_{\text{new}}$, the predicted target value $\hat{y}_{\text{new}}$ using KNN regression is computed as follows, where y_{nearest_i} represents the target value of the i-th nearest neighbor to $\mathbf{x}_{\text{new}}$.

$$\hat{y}_{\text{new}} = \frac{1}{K} \sum_{i=1}^{K} y_{\text{nearest}_i} \tag{62}$$

The choice of distance metric plays a crucial role in KNN regression. Euclidean distance is the most commonly used and is calculated as below, where $\mathbf{x}_i$ and $\mathbf{x}_{new}$ are two data points, and p is the number of features. Distance is calculated for ∀ i = 1,2 ..., instanceAmount

$$d(\mathbf{x}_i, \mathbf{x}_{new}) = \|\mathbf{x}_i - \mathbf{x}_{new}\|_2 = \sqrt{\sum_{j=1}^{p} (x_{i,j} - x_{new,j})^2} \tag{63}$$

The choice of the parameter K in KNN regression is critical. A small value of K can lead to high variance and overfitting, while a large value of K can lead to high bias and underfitting. The optimal value of K is often determined using techniques such as cross-validation.

4 Expected & Encountered Challenges

Linear regression and the variations (Lasso and Ridge) has struggled with capturing non-linear relationships and generalizing to new music. Ridge and Lasso regression caused difficulties in selecting regularization parameters (finding and selecting the correct hyper-parameters) and interpreting sparse solutions, impacting their generalization. Neural networks were expected to offer flexibility in capturing complex patterns but they could have overfit on small datasets and lack interpretability; as expected, it offered the greatest flexibility in terms of capturing non-linear relationships yet it did not overfit even a bit underfitted due to high complexity of data. Therefore learning the very complex patterns in the data was hard even with neural networks. Additionally, neural network method was converging very slowly as the loss decreased, which required me to implement aforementioned gradient techniques. K-Nearest Neighbors regression encountered issues with high-dimensional feature spaces as batching the test data became necessary which also produced a new hyperparameter (batch size) impacting the result.

Beyond method-specific challenges, the large scale of the dataset, comprising around 100,000 instances and 29 features, introduces computational complexities. Expensive computations arise from the need to process and analyze vast amounts of data, especially for algorithms like neural networks and distance-based methods such as K-Nearest Neighbors. Moreover, managing high dimensionality poses a universal challenge, requiring careful feature selection and dimensionality reduction techniques (such as the aforementioned PCA method) to mitigate the curse of dimensionality. For instance, training neural networks with full batch gradient descent was computationally expensive since the dataset I used is a large one (relative to the projects I work with). This required me to also implement mini-batch gradient descent and SGD gradient descent methods. Similarly, KNN regression method required me to process the data in batches instead of a whole full batch since the memory was not enough for this operation due to large dataset. Additionally, addressing biases and imbalances in the data, such as class distributions in happiness ratings and potential biases in feature representation, remains crucial for building reliable and generalizable models.

Furthermore, the uncertainty that models may not make good generalized predictions arises from the weak correlation between features and valence (happiness rating). Learning from this dataset doesn't guarantee capturing hidden patterns, if any exist, necessitating cautious interpretation of the model's predictions. Achieving accurate predictions demands not only method-specific optimizations but also robust preprocessing, feature engineering, and model evaluation strategies tailored to the dataset's characteristics and computational constraints.

5 Performance Analysis

Validating the performance of the methods employed for the given dataset is very important to ensure the reliability and effectiveness of the predictive models. To achieve this, a comprehensive evaluation strategy will be adopted, comprising multiple steps. Firstly, the dataset will be split into training, validation, and test sets using appropriate proportions, such as an 80-10-10 split. This allows for training the models on a subset of the data, tuning hyperparameters using the validation set, and assessing final performance on the test set to estimate real-world generalization.

For each machine learning method utilized, suitable performance metrics will be employed to quantify predictive accuracy, such as mean squared error (MSE) or mean absolute error (MAE). MSE measures the average squared difference between predicted and actual values, providing insight into the model's overall predictive accuracy. MAE, on the other hand, measures the average absolute difference between predicted and actual values, offering a more interpretable metric that is less sensitive to outliers. Additionally, the coefficient of determination (R^2) will be calculated to assess the goodness of fit of regression models and their predictive performance. R^2 represents the proportion of the variance for a dependent variable explained by the independent variable(s) in the regression model. It ranges (generally) from 0 to 1, with 1 indicating a perfect fit and 0 indicating no linear relationship between the variables (negative values indicate extremely poor fit).

Additionally, techniques like cross-validation could be employed to assess model robustness and generalization across different subsets of the data. Cross-validation involves repeatedly splitting the data into training and validation sets, training the model on each split, and evaluating its performance, thereby providing a more reliable estimate of the model's performance on unseen data. By rigorously validating the performance of the methods through these systematic approaches, confidence in the predictive capabilities of the models can be established, facilitating informed decision-making in real-world applications. However, for this specific problem and owing to the large amount of data, cross validition was not strongly required and train-validation-test split was enough for assessing the performance.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \qquad (64) \qquad\qquad \text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \qquad (65)$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{66}$$

6 Simulation Setup & Results

For each method, I have trained models with the preprocessed data (Normalization, one-hot encoding and shuffling) then dataset is split into train, test and validation sets (with proportions 0.8, 0.1, 0.1). The train loss vs. iteration (epoch) plots, validation and test performances (MSE and R^2 metrics are used) and elapsed training times are reported. Due to large dataset, hyperparameters are not exhaustively searched by algorithms (except for KNN), yet best hyperparameters are chosen empirically by testing on the validation set, and so-far best performing ones are used for these results. Results and model details are reported in a table (Appendix B, Table 1).

Linear Regression: The normal equation method failed to produce a successful result as design matrix was not invertible (Extremely bad R^2 score obtained). Therefore, Pseudo-inverse [6] method for least squares is used which is a least squares solution not affected by non-invertibility due to SVD. The gradient descent (20.000 epochs and 10^{-1} learning rate) also produced (Appendix B, Fig. 8) a very similar loss to pseudo-inverse method. Therefore linear regression served as a baseline model and captured the linear relationship as much as it can.

Ridge Regression: Normal equation (with lambda 10^{-4}) of ridge does not suffer from invertibility as the closed form is guaranteed to be always invertible. The gradient descent method (20.000 epochs, 10^{-1} learning rate and lambda constant 10^{-4}), successfully, produced a very similar result to the baseline normal equation solution. As expected ridge solution, has a slightly worse training loss than linear regression but it generalized as good as linear regression (Appendix B, Fig. 9).

Lasso Regression: As expected and very similar to the ridge regression, Lasso produced a worse training loss after gradient descent method (20.000 epochs, 10^{-1} learning rate and lambda constant 10^{-4}) than linear regression, yet it generalized almost with same performance with linear and ridge regression. (Appendix B, Fig. 10). Since there is no closed-form solution form of lasso, it is rather harder to compare with a baseline performance.

Neural Network: For all neural network variation below parameters initialized according to Xavier initialization in order to avoid exploding/vanishing gradients and local minimas (Random initialization resulted in being stuck at a local optima).

Firstly, a neural network with two hidden layers (30 neurons, 10 neurons at hidden layers) is used for experimenting the batch size with gradient descent. **Full batch (FB) gradient descent** (5.000 epochs, 10^4 learning rate) (Appendix B, Fig. 11) immediately resulted in a better performing model than the linear models, however the learning process became slow as the loss decreased which required other optimizer and batching techniques. The same FB gradient descent setup (5.000 epochs, 10^4 learning rate) was combined with **Principal Component Analysis (PCA)** (Appendix B, Fig. 12) method for experimental reasons (since all methods can handle high dimensional data as well), yet the performance results were worse than plain FB method due to the information loss in dimensionality reduction process.

At the other extreme **Stochastic Gradient Descent (SGD)** (Appendix B, Fig. 13) uses 1 randomly sampled instance for a weight update. Knowing that there are 91200 train instances; There were 91200 weight updates happening at one epoch whereas only 1 updates were happening with FB gradient descent at one epoch. Thus, SGD converged (50 epochs, 10^{-4} learning rate) to a better train loss value within 50 epochs where FB gradient descent took 5000 epochs to get (almost) there. Note that SGD required a much lower learning rate since loss function of a single instance has much steeper points, that caused exploding gradients with prior learning rates, yet these random loss functions of each instance helped with avoiding local minimas.

The **Mini-batch (MB)** (Appendix B, Fig. 14) preserved the benefits of two extremes (FB and SGD), neither the computations were inefficient as SGD nor the weight update rate was as low as FB gradient descent. Actually, mini-batch (1.000 epochs, 10^{-2} learning rate) performed best among all batching options (FB and SGD) where the whole training set is distributed to the batches of size 16 (5700 batches in total). This outstanding performance implies that the combination of MB with an optimizer could be impressive.

The **Momentum** (with Momentum constant 0.9) (Appendix B, Fig. 15) as an optimizer dampened the oscillations in the learning process with FB gradient descent (5.000 epochs, 10^4 learning rate), which led to a faster and more stable convergence which is also depicted in both the training and test performances. The performance is much better compared to plain FB gradient descent. Similarly, **Nesterov** (with Momentum constant 0.9) (Appendix B, Fig. 16) method is used with FB gradient descent (5.000 epochs, 10^4 learning rate) and produced a very similar performance to Momentum method. Even though, both optimizers performed very well, Nesterov was not a huge upgrade on Momentum.

Another optimizer used in experiments was **AdaGrad** (with $\epsilon = 10^{-8}$) (Appendix B, Fig. 17) which aims to adapt the learning rate based on gradients; having been trained with FB gradient descent (5.000 epochs, 10^{-2} learning rate), AdaGrad performed almost as good as momentum based optimizers. On the other hand **RMSProp** (with $\epsilon = 10^{-8}$ and decay rate 0.9) (Appendix B, Fig. 18), which is an upgrade on AdaGrad by solving decaying learning rate issue, performed better than both momentum based methods and AdaGrad with the same setup of FB gradient descent (5.000 epochs, 10^{-2} learning rate). Hence RMSProp has proven to be a real upgrade on AdaGrad.

Most promising optimizer **Adam** (with $\epsilon = 10^{-8}$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$) (Appendix B, Fig. 19) combines the Momentum and RMSProp method which are the best performing momentum method and adaptive method respectively. As expected with FB gradient descent setup (5.000 epochs, 10^{-2} learning rate), Adam performed the best with a significant difference. **AMSGrad** (with $\epsilon = 10^{-8}$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$) (Appendix B, Fig. 20) is an update on Adam by solving some convergence issues, yet with the same setup AMSGrad failed to be better than Adam in terms of performance, still it performed better than rest of the optimizers.

Finally, the best performing optimizer **Adam** is combined with different batching techniques (FB, MB and SGD) with a different shallow neural network setup (100 neurons at hidden layer) but more neurons. The FB Adam of shallow NN (Appendix B, Fig. 21) performed slightly better FB Adam deep NN (30, 10). Moreover, MB shallow NN (1.000 epochs, 10^{-2} learning rate) (Appendix B, Fig. 22) was better than FB method in terms of train loss but worse in generalization. Most importantly, the shallow NN with optimizer Adam and SGD batching (50 epochs, 10^{-4} learning rate) (Appendix B, Fig. 23) performed best in this project in terms of both train loss, test (MSE) and R^2 score.

KNN Regression: Calculation-heavy nature of this method made it very computationally expensive both time and space-wise, therefore I had to sample my dataset (test or validation) for the predictions. More specifically, test split has 11400 instance which causes memory error due to insufficient memory, hence I batched the test data with sample size 456 so that whole test split was used for prediction with 25 batches. The performance was very surprisingly outstanding considering that the implementation complexity is very low compared to other methods. The ability of KNN method to capture local complex patterns was much suitable for my problem compared to models assuming a linear relationship as the performances depicts.After trying many different values for hyperparameter K, K = 5 gave the best result on the validation data (Appendix B, Fig. 24). Therefore test performance with fine tuned hyperparameters was much better than linear methods and even better than many of the neural network with additional techniques.

7 Conclusion

The purpose of this project was to predict the emotional valence of musical pieces across various genres. Utilizing machine learning regression methods including Linear Regression, Ridge Regression, Lasso Regression, Neural Network, and K-Nearest Neighbors Regression, the aim was to uncover the hidden relationships between musical characteristic elements and valence by using the Spotify API Dataset.

Dataset analysis revealed a diverse range of musical characteristics, from duration and danceability to acousticness and valence. Preprocessing steps including one-hot encoding, shuffling, standardization, normalization, and Principal Component Analysis (PCA) were crucial in preparing the dataset for training. These steps helped in managing high dimensionality and ensuring compatibility with machine learning algorithms.

Encountered challenges encompassed method-specific limitations, computational complexities, and the nuanced nature of music-emotion relationships. Linear regression models provided baseline insights, while Ridge and Lasso regression addressed complexities related to feature selection and regularization, offering means to mitigate overfitting and improve generalization performance. Neural networks exhibited potential for capturing complex patterns, albeit with computational challenges and slow convergence rates. K-Nearest Neighbors Regression stood out for its simplicity and ability to capture local patterns, offering promising results despite computational constraints.

Performance analysis underscored the importance of rigorous evaluation strategies, including train-validation-test splits, performance metrics like mean squared error (MSE) and R^2 score. Experimental results showcased the effectiveness of different optimization techniques and batching strategies, with Adam optimizer coupled with stochastic gradient descent (SGD) batching emerging as the best-performing combination for neural networks with a network sturcture containing only 1 hidden layer but many neurons within rather than a deep network with less neurons for this specific problem and dataset.

In conclusion, while each method presented its own set of advantages and challenges, the project demonstrated the feasibility of predicting emotional valence in music using machine learning techniques. Future endeavors could explore ensemble methods, feature engineering, and larger datasets to further enhance predictive accuracy and deepen understanding of music-emotion dynamics. Ultimately, this research contributes to the interdisciplinary domain of music psychology and computational musicology, paving the way for applications in music recommendation systems, affective computing, and digital music therapy.

8 APPENDIXES

8.1 Appendix A - Dataset

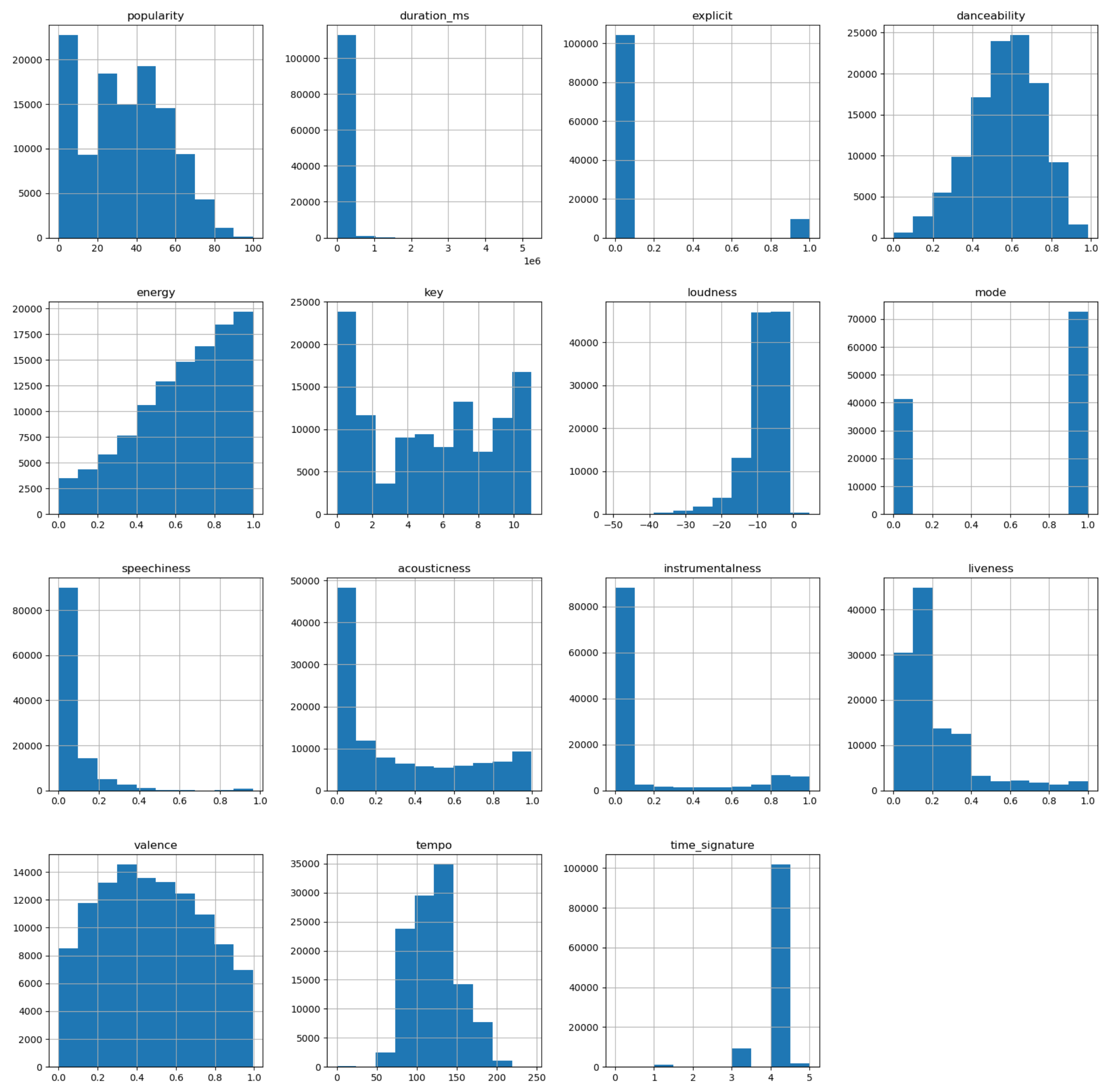


Figure 1: Histograms of features.

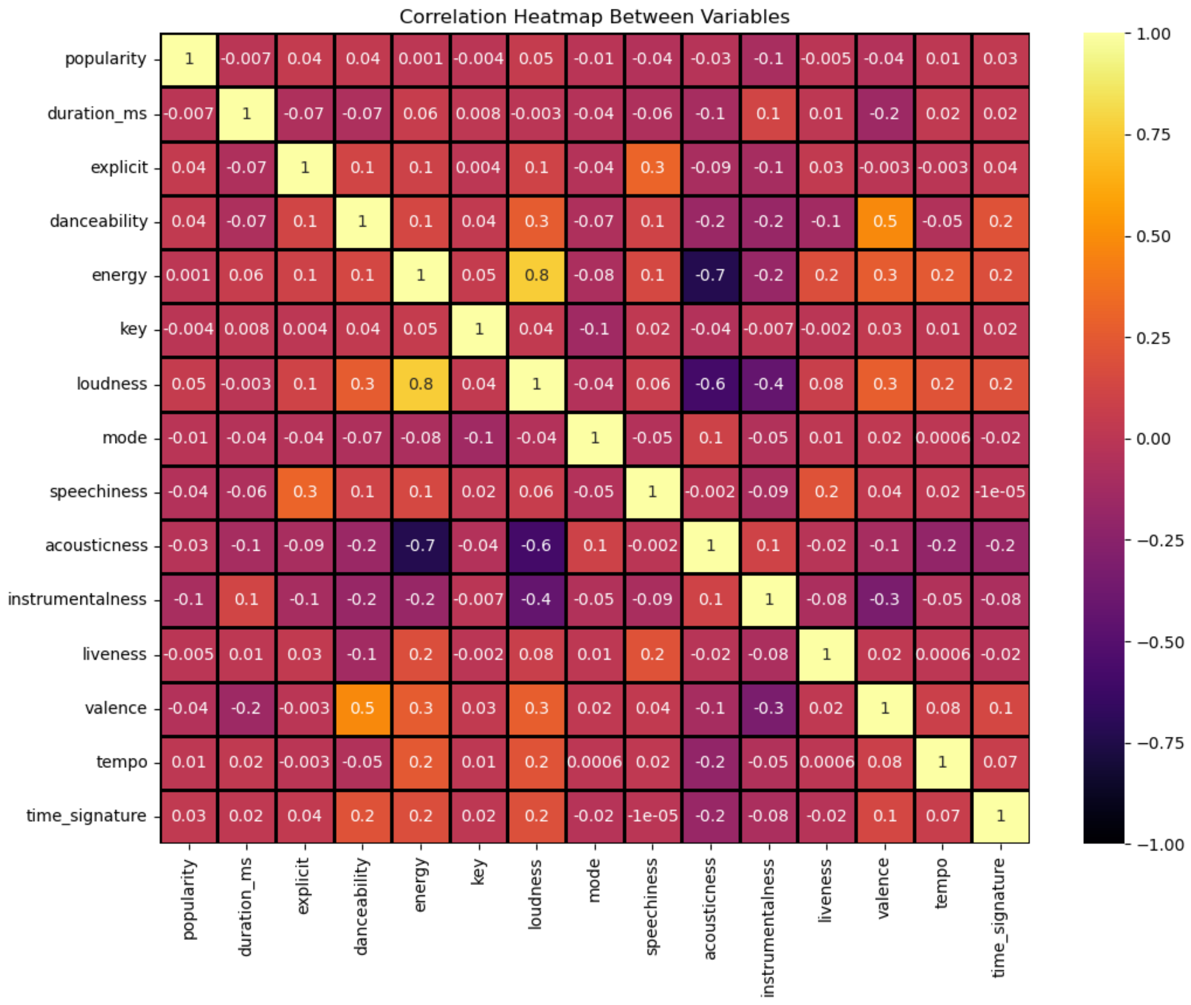


Figure 2: Correlation matrix as heatmap.

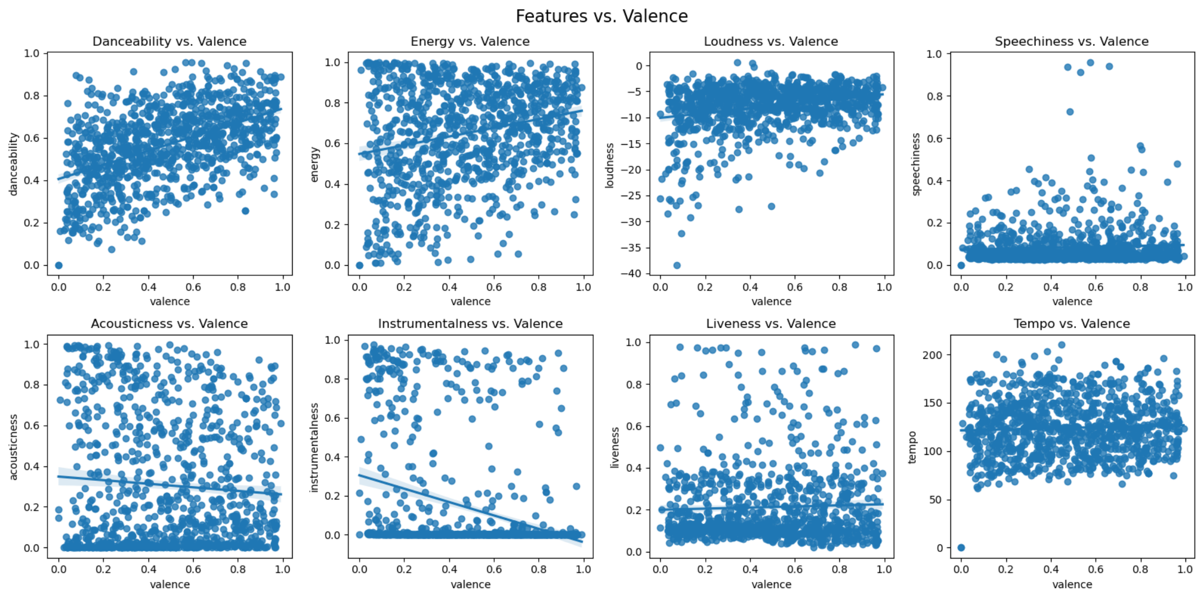
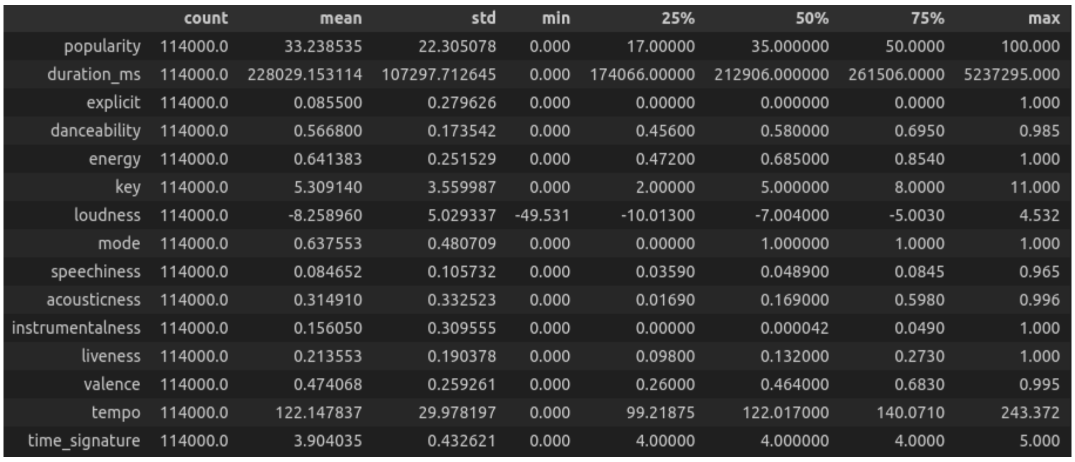


Figure 3: Correlation of key features (1000 instances sampled).



	count	mean	std	min	25%	50%	75%	max
popularity	114000.0	33.238535	22.305078	0.000	17.00000	35.000000	50.0000	100.000
duration_ms	114000.0	228029.153114	107297.712645	0.000	174066.00000	212906.000000	261506.0000	5237295.000
explicit	114000.0	0.085500	0.279626	0.000	0.00000	0.000000	0.0000	1.000
danceability	114000.0	0.566800	0.173542	0.000	0.45600	0.580000	0.6950	0.985
energy	114000.0	0.641383	0.251529	0.000	0.47200	0.685000	0.8540	1.000
key	114000.0	5.309140	3.559987	0.000	2.00000	5.000000	8.0000	11.000
loudness	114000.0	-8.258960	5.029337	-49.531	-10.01300	-7.004000	-5.0030	4.532
mode	114000.0	0.637553	0.480709	0.000	0.00000	1.000000	1.0000	1.000
speechiness	114000.0	0.084652	0.105732	0.000	0.03590	0.048900	0.0845	0.965
acousticness	114000.0	0.314910	0.332523	0.000	0.01690	0.169000	0.5980	0.996
instrumentalness	114000.0	0.156050	0.309555	0.000	0.00000	0.000042	0.0490	1.000
liveness	114000.0	0.213553	0.190378	0.000	0.09800	0.132000	0.2730	1.000
valence	114000.0	0.474068	0.259261	0.000	0.26000	0.464000	0.6830	0.995
tempo	114000.0	122.147837	29.978197	0.000	99.21875	122.017000	140.0710	243.372
time_signature	114000.0	3.904035	0.432621	0.000	4.00000	4.000000	4.0000	5.000

Figure 4: Statistic of non-processed dataset.

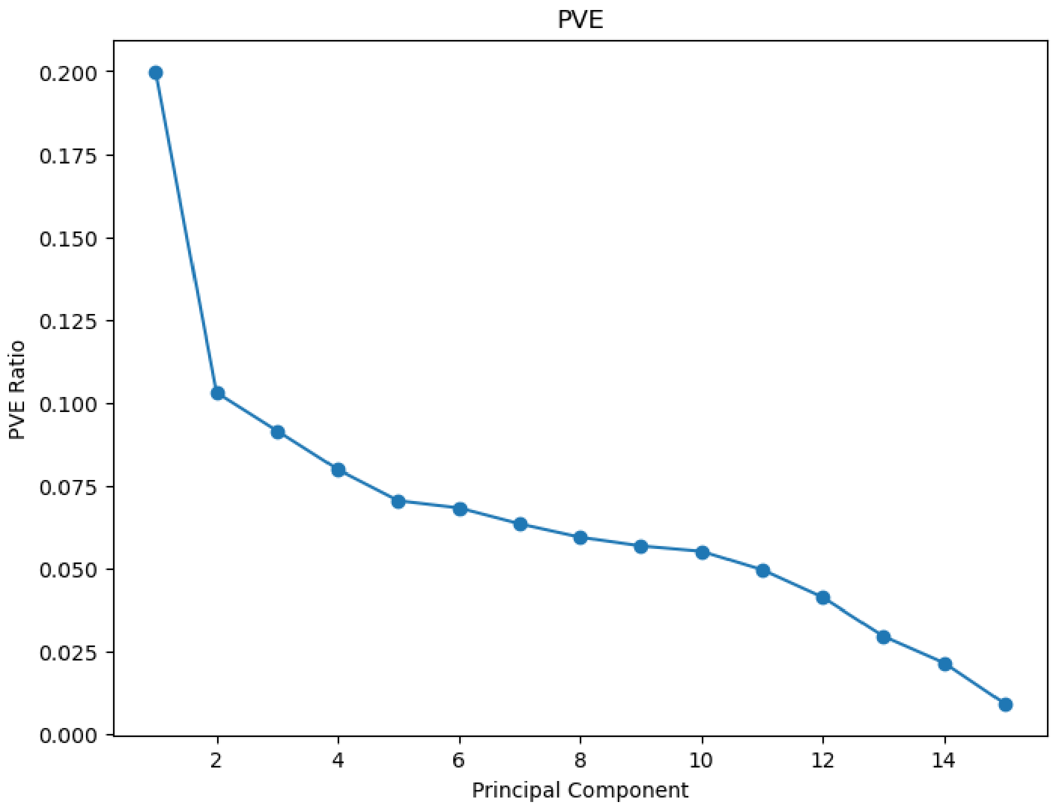


Figure 5: PVE vs PC Plot.

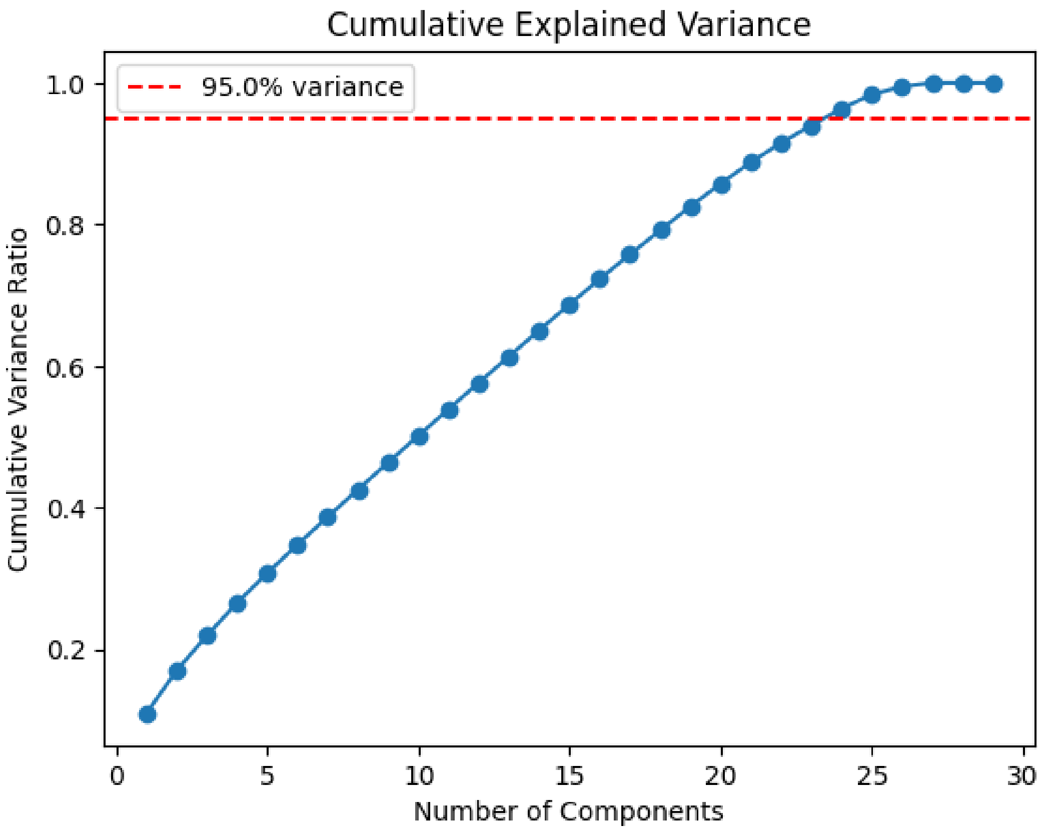
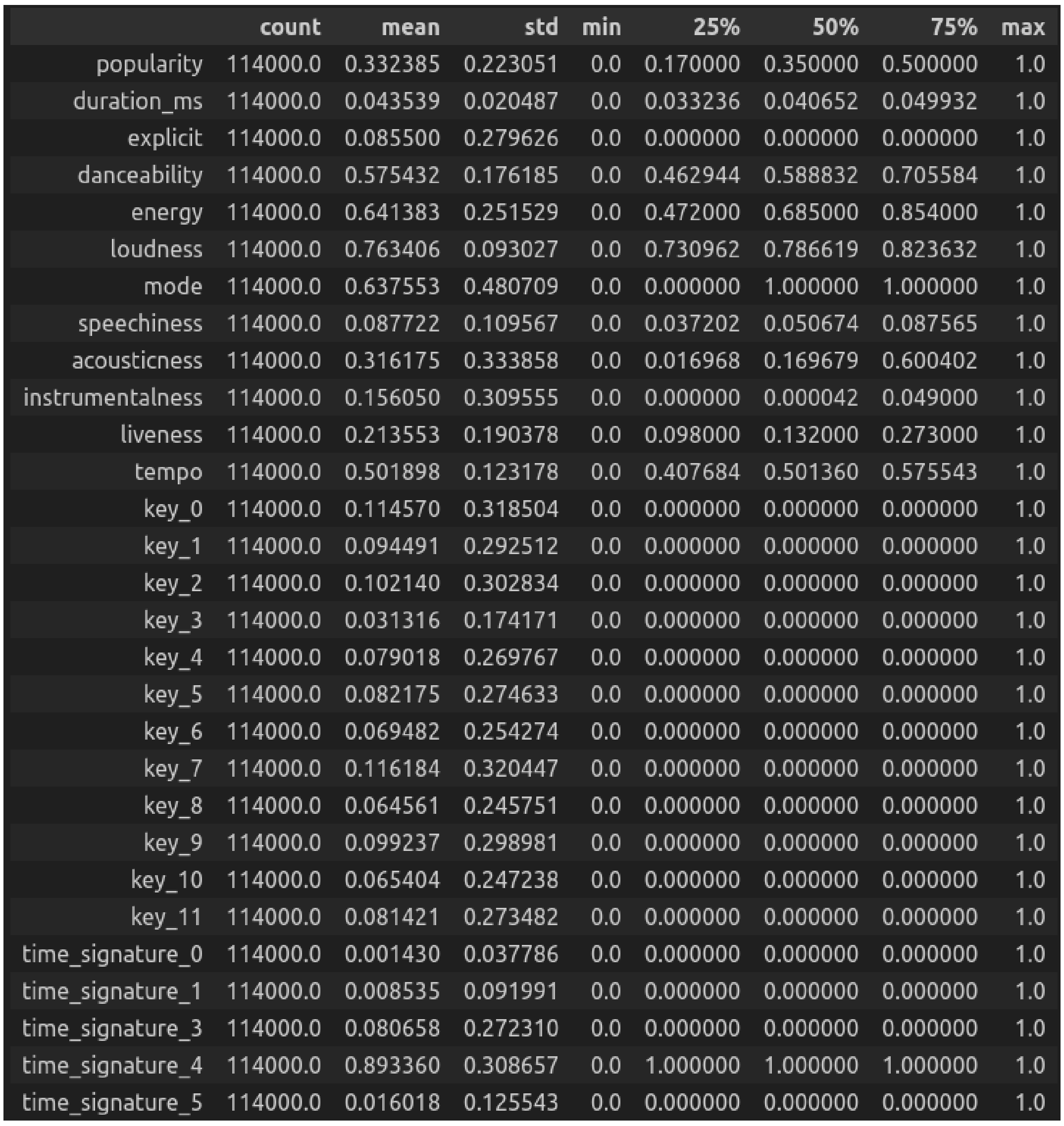


Figure 6: Cumulative PVE



	count	mean	std	min	25%	50%	75%	max
popularity	114000.0	0.332385	0.223051	0.0	0.170000	0.350000	0.500000	1.0
duration_ms	114000.0	0.043539	0.020487	0.0	0.033236	0.040652	0.049932	1.0
explicit	114000.0	0.085500	0.279626	0.0	0.000000	0.000000	0.000000	1.0
danceability	114000.0	0.575432	0.176185	0.0	0.462944	0.588832	0.705584	1.0
energy	114000.0	0.641383	0.251529	0.0	0.472000	0.685000	0.854000	1.0
loudness	114000.0	0.763406	0.093027	0.0	0.730962	0.786619	0.823632	1.0
mode	114000.0	0.637553	0.480709	0.0	0.000000	1.000000	1.000000	1.0
speechiness	114000.0	0.087722	0.109567	0.0	0.037202	0.050674	0.087565	1.0
acousticness	114000.0	0.316175	0.333858	0.0	0.016968	0.169679	0.600402	1.0
instrumentalness	114000.0	0.156050	0.309555	0.0	0.000000	0.000042	0.049000	1.0
liveness	114000.0	0.213553	0.190378	0.0	0.098000	0.132000	0.273000	1.0
tempo	114000.0	0.501898	0.123178	0.0	0.407684	0.501360	0.575543	1.0
key_0	114000.0	0.114570	0.318504	0.0	0.000000	0.000000	0.000000	1.0
key_1	114000.0	0.094491	0.292512	0.0	0.000000	0.000000	0.000000	1.0
key_2	114000.0	0.102140	0.302834	0.0	0.000000	0.000000	0.000000	1.0
key_3	114000.0	0.031316	0.174171	0.0	0.000000	0.000000	0.000000	1.0
key_4	114000.0	0.079018	0.269767	0.0	0.000000	0.000000	0.000000	1.0
key_5	114000.0	0.082175	0.274633	0.0	0.000000	0.000000	0.000000	1.0
key_6	114000.0	0.069482	0.254274	0.0	0.000000	0.000000	0.000000	1.0
key_7	114000.0	0.116184	0.320447	0.0	0.000000	0.000000	0.000000	1.0
key_8	114000.0	0.064561	0.245751	0.0	0.000000	0.000000	0.000000	1.0
key_9	114000.0	0.099237	0.298981	0.0	0.000000	0.000000	0.000000	1.0
key_10	114000.0	0.065404	0.247238	0.0	0.000000	0.000000	0.000000	1.0
key_11	114000.0	0.081421	0.273482	0.0	0.000000	0.000000	0.000000	1.0
time_signature_0	114000.0	0.001430	0.037786	0.0	0.000000	0.000000	0.000000	1.0
time_signature_1	114000.0	0.008535	0.091991	0.0	0.000000	0.000000	0.000000	1.0
time_signature_3	114000.0	0.080658	0.272310	0.0	0.000000	0.000000	0.000000	1.0
time_signature_4	114000.0	0.893360	0.308657	0.0	1.000000	1.000000	1.000000	1.0
time_signature_5	114000.0	0.016018	0.125543	0.0	0.000000	0.000000	0.000000	1.0

Figure 7: Statistic of pre-processed dataset.png

8.2 Appendix B - Preliminary Results

Table 1: Performance Metrics for Different Machine Learning Methods

Method	**Train MSE**	**Test MSE**	**Test R^2**	**Epoch**	**Time**	**Parameter**
Linear (Normal Equation)	2.1948	2.1942	-31.63	-	0.08	-
Linear (Pseudo-inverse)	0.0419	0.0419	0.3758	-	0.23	-
Linear (Gradient Descent)	0.0420	0.0421	0.3743	20 K	36.88	$\alpha = 10^{-1}$
Ridge (Normal Equation)	0.0422	0.0419	0.3766	-	0.074	$\lambda = 10^{-4}$
Ridge (Gradient Descent)	0.0423	0.0421	0.3738	20 K	34.45	$\alpha = 10^{-1}$, $\lambda = 10^{-4}$
Lasso	0.0427	0.0421	0.3737	20 K	60.5	$\alpha = 10^{-1}$, $\lambda = 10^{-4}$
NN (Full Batch (30, 10, 1))	0.0409	0.0413	0.3858	5 K	500.6	$\alpha = 10^{4}$
NN (Mini Batch (30, 10, 1))	0.0378	0.0380	0.4344	1 K	443.8	$\alpha = 10^{-2}$
NN (SGD (30, 10, 1))	0.0407	0.0401	0.3879	50	324.6	$\alpha = 10^{-4}$
NN (FB - Momentum(30, 10, 1))	0.0374	0.0385	0.4272	5 K	348.7	$\alpha = 10^{4}$
NN (FB - RMSProp(30, 10, 1))	0.0370	0.0385	0.4275	5 K	341.6	$\alpha = 10^{-2}$
NN (FB - AdaGrad(30, 10, 1))	0.0381	0.0390	0.4205	5 K	319.0	$\alpha = 10^{-2}$
NN (FB - Nesterov(30, 10, 1))	0.0374	0.0385	0.4271	5 K	332.4	$\alpha = 10^{4}$
NN (FB - Adam(30, 10, 1))	0.0354	0.0364	0.4582	5 K	351.1	$\alpha = 10^{-2}$
NN (FB - Adam(100, 1))	0.0354	0.0350	0.4780	10 K	672.4	$\alpha = 10^{-2}$
NN (FB - AMSGrad(30, 10, 1))	0.0369	0.0376	0.4405	5 K	313.6	$\alpha = 10^{-2}$
NN (FB - PCA(30, 10, 1))	0.0426	0.0433	0.3554	5 K	372.7	$\alpha = 10^{4}$
NN (MB - Adam(100, 1))	0.0349	0.0361	0.4688	1 K	246.2	$\alpha = 10^{-2}$
NN (SGD - Adam(100, 1))	0.0337	0.0346	0.4852	50	614.3	$\alpha = 10^{-4}$
KNN	-	0.0386	0.4253	-	158.9	$K = 10$

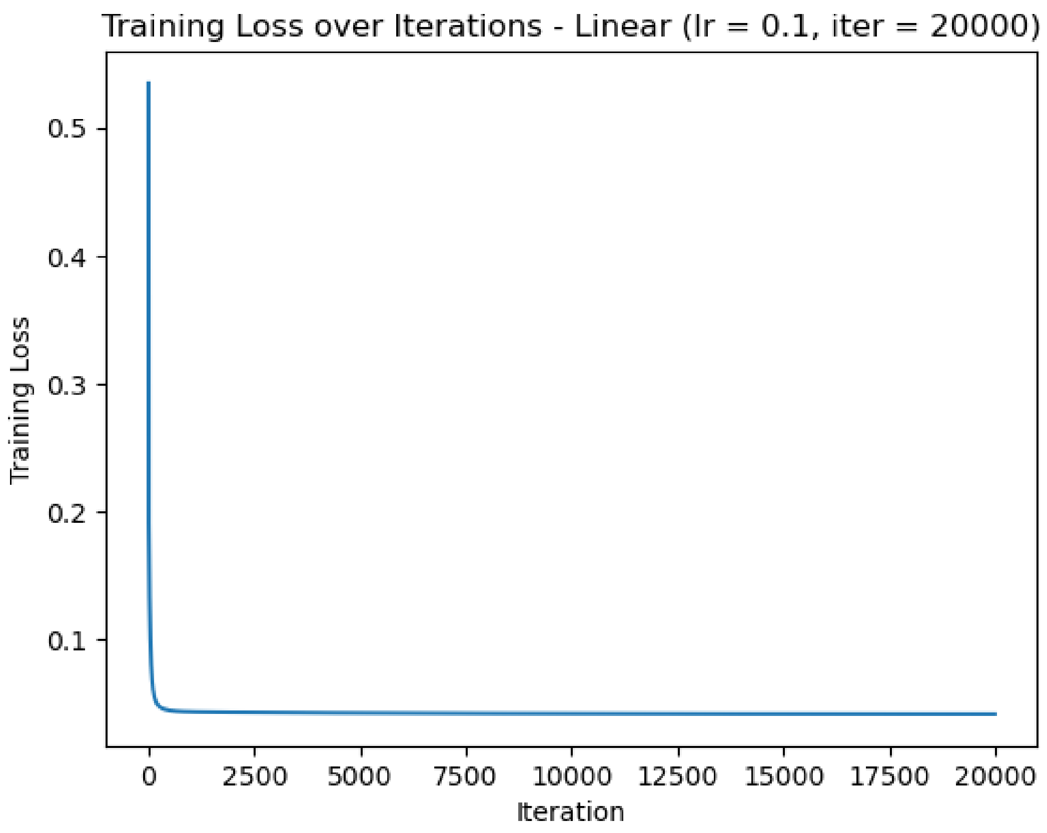


Figure 8: Train Loss vs Iterations (20.000) for Linear Regression with $\alpha = 10^{-1}$.

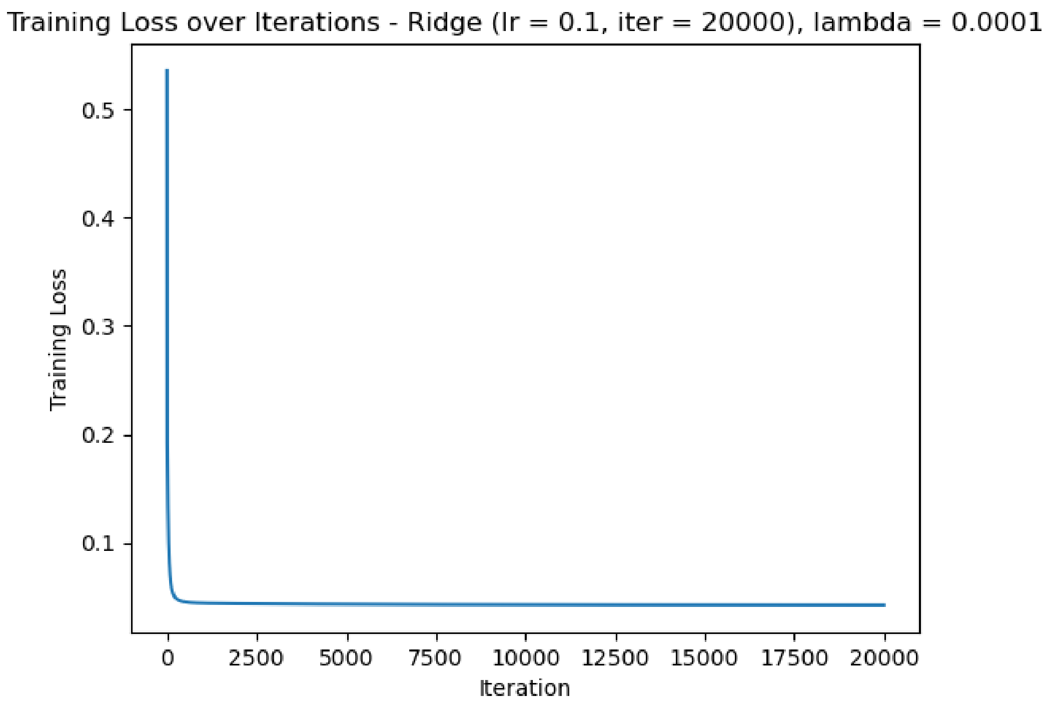


Figure 9: Train Loss vs Iterations (20.000) for Ridge Regression with $\alpha = 10^{-1}, \lambda = 10^{-3}$.

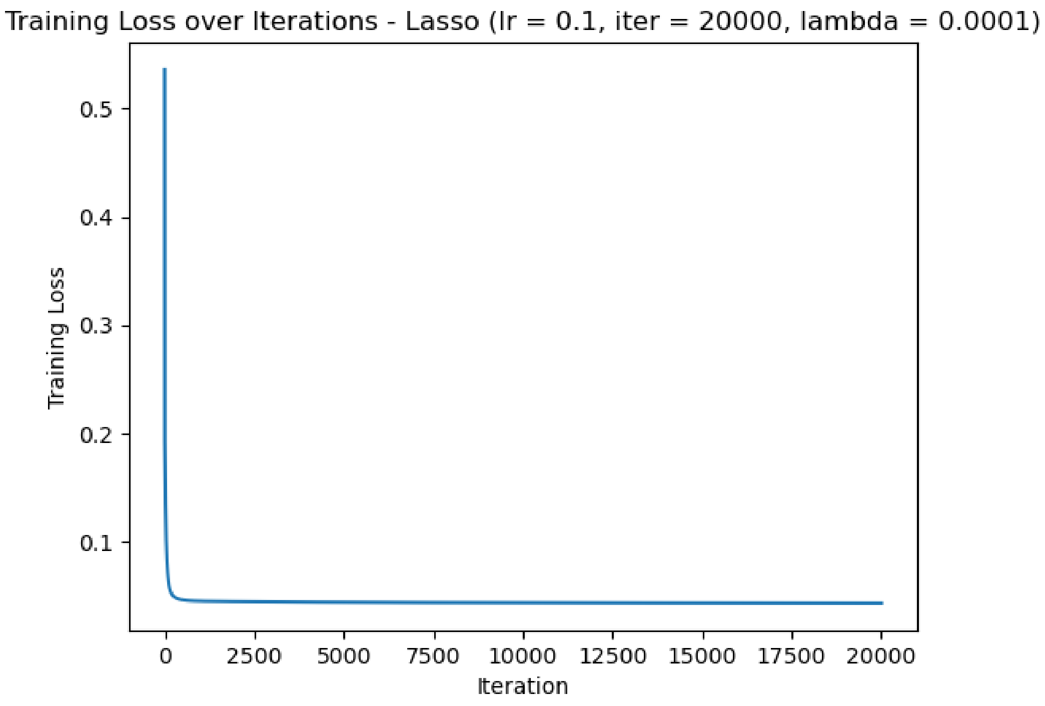


Figure 10: Train Loss vs Iterations (20.000) for Lasso Regression with $\alpha = 10^{-1}, \lambda = 10^{-4}$.

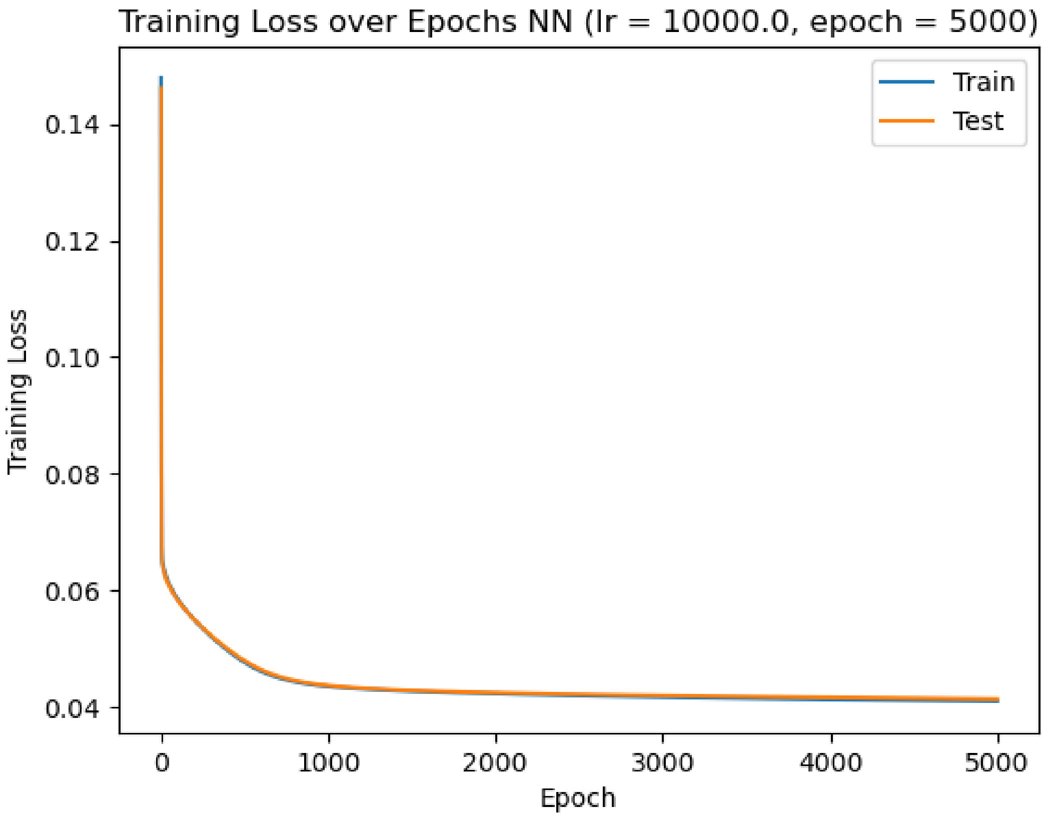


Figure 11: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network, $\alpha = 10^4$

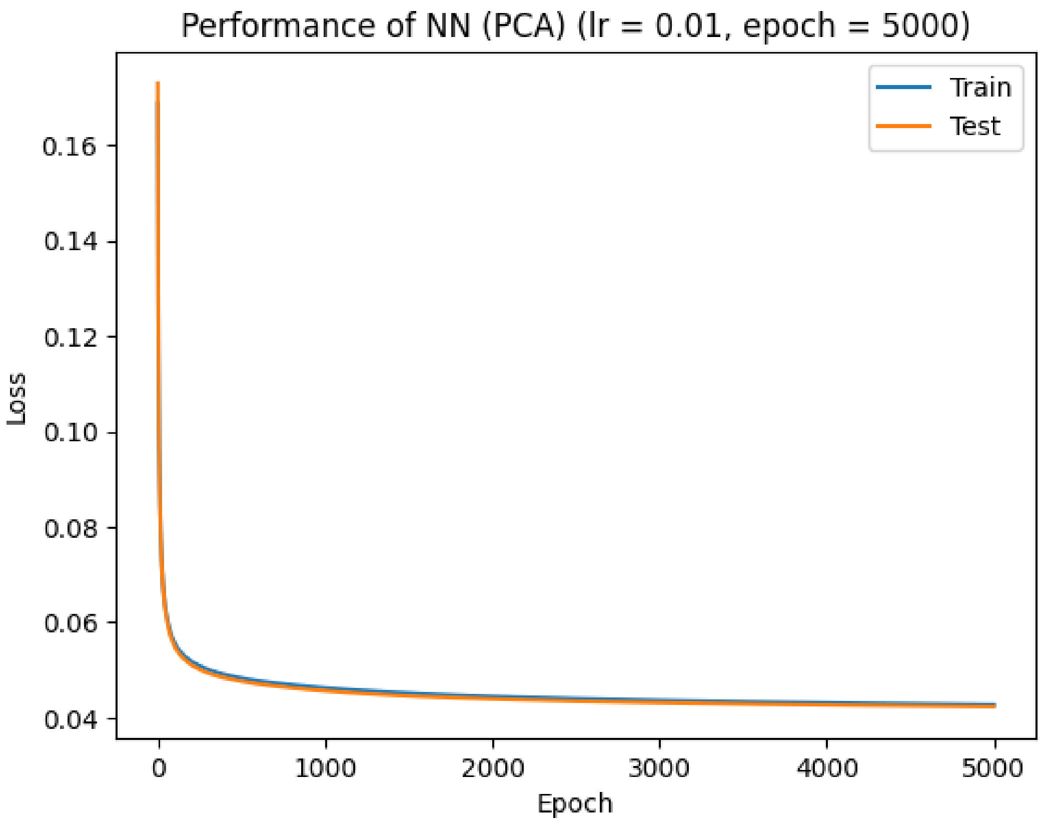


Figure 12: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (PCA), $\alpha = 10^4$

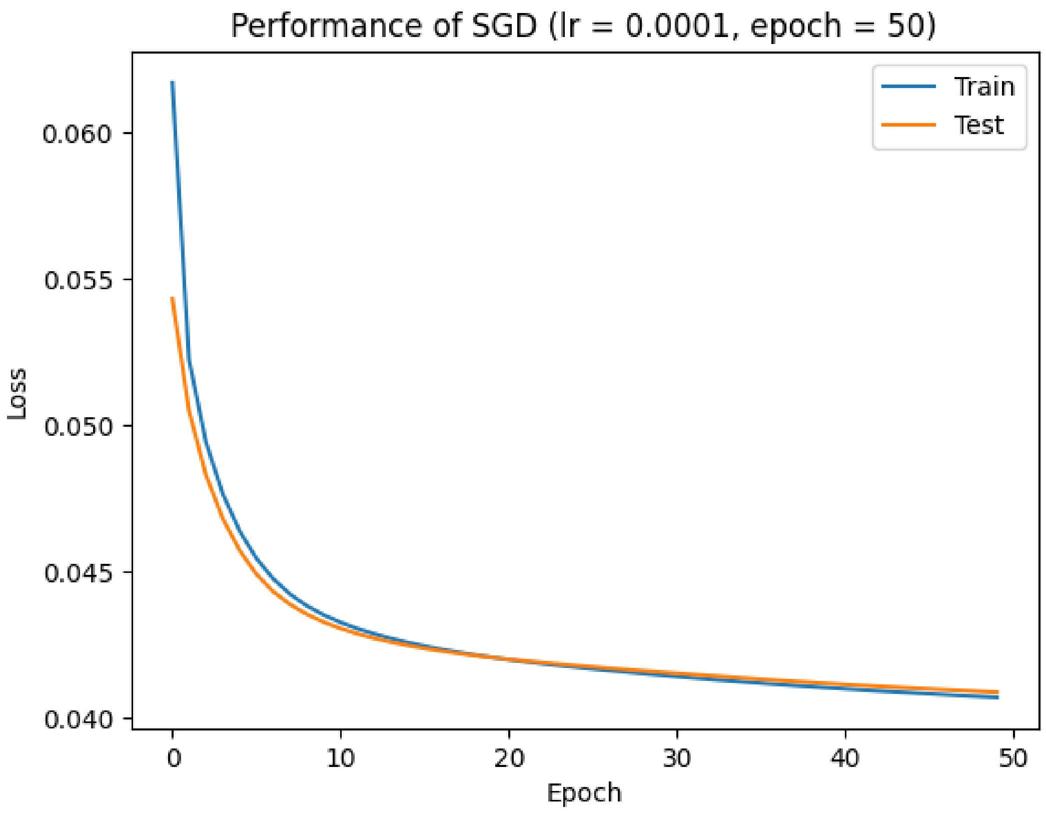


Figure 13: Train and Test Loss vs Epochs (50) for SGD 3 Layer (30, 10, 1) Neural Network, $\alpha = 10^{-4}$

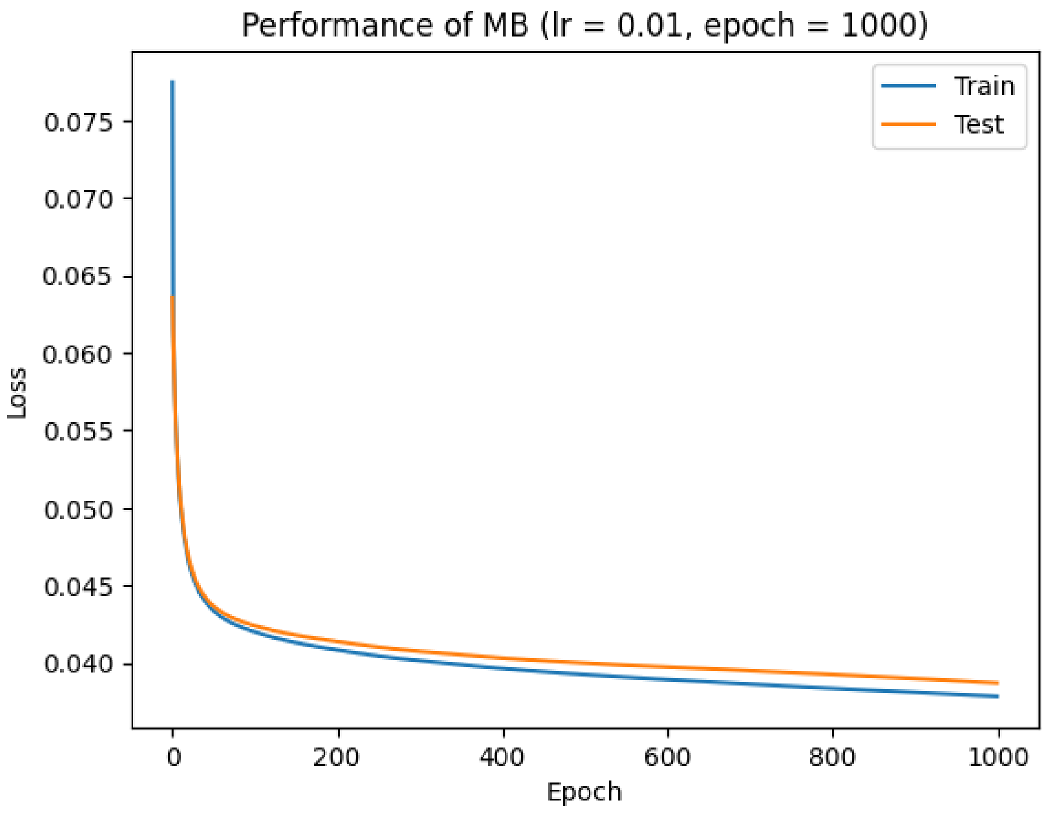


Figure 14: Train and Test Loss vs Epochs (1.000) for Mini-Batch 3 Layer (30, 10, 1) Neural Network, $\alpha = 10^{-2}$

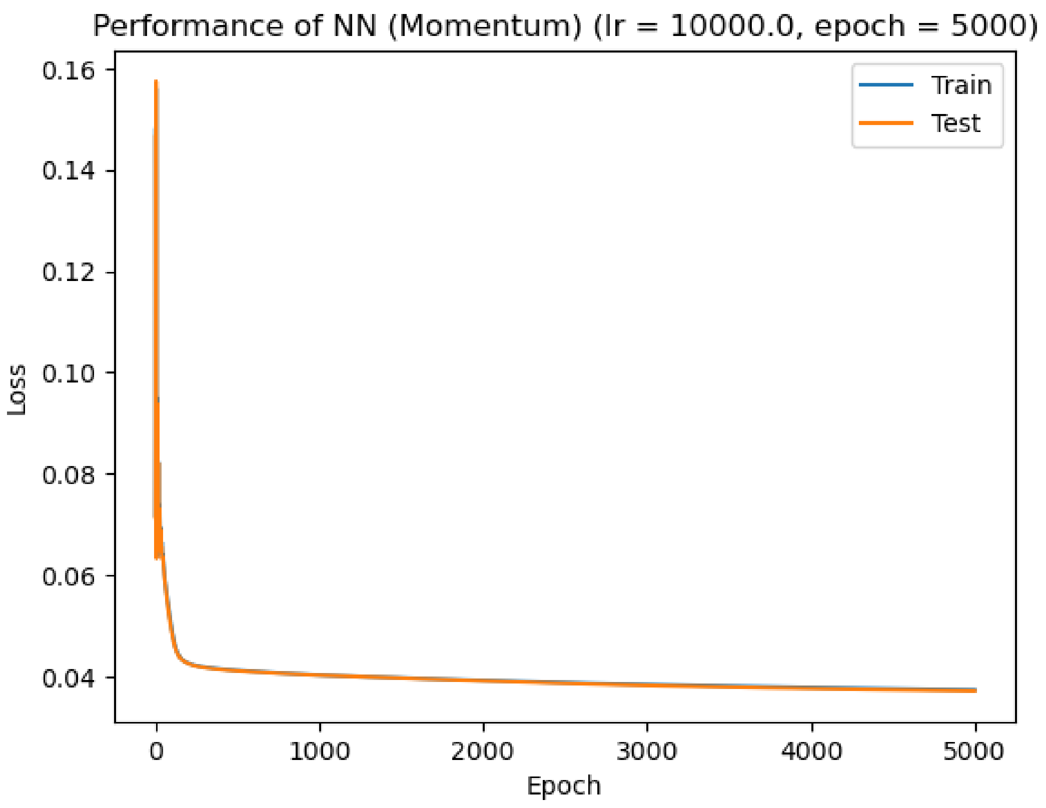


Figure 15: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (Momentum), $\alpha = 10^4, momentum = 0.9$

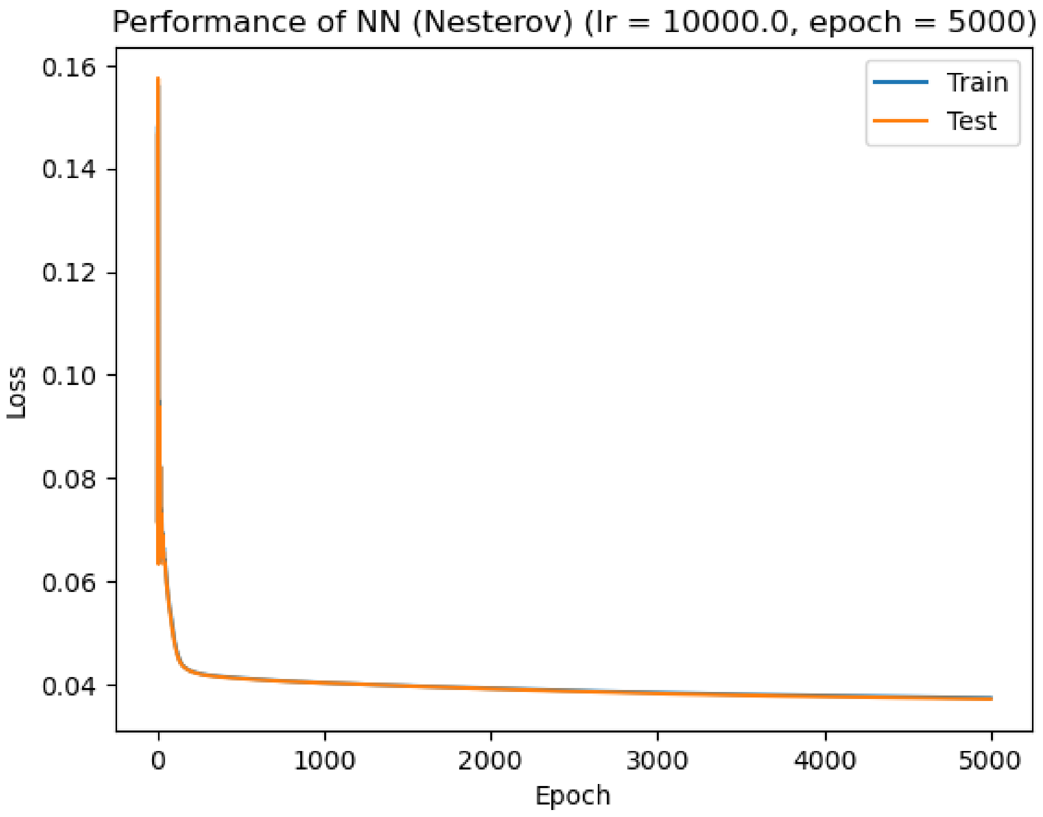


Figure 16: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (Nesterov), $\alpha = 10^4, momentum = 0.9$

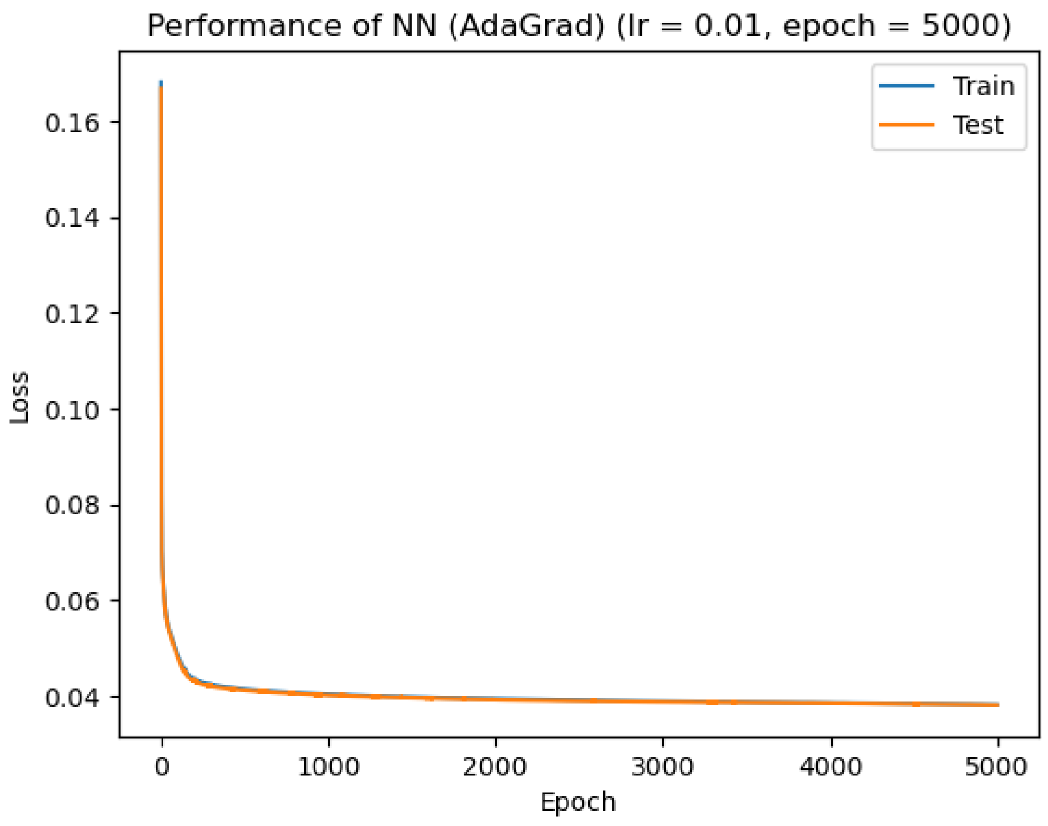


Figure 17: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (AdaGrad), $\alpha = 10^{-2}, \epsilon = 10^{-8}$

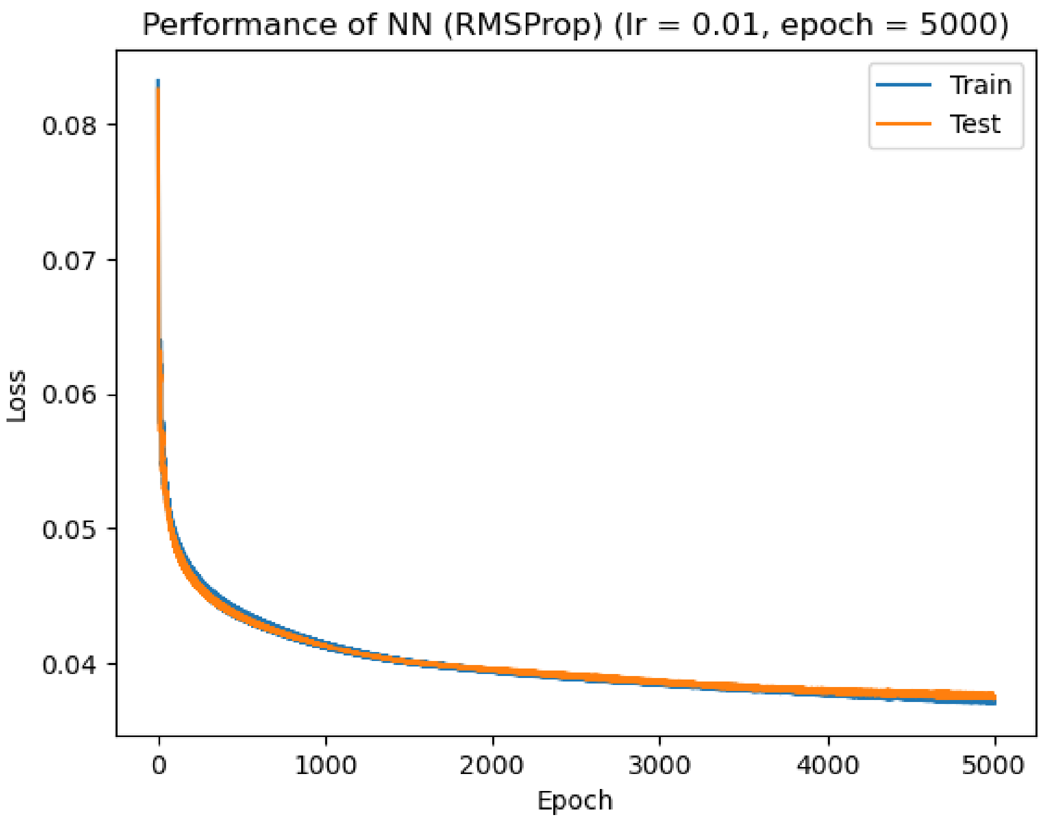


Figure 18: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (RMSProp), $\alpha = 10^{-2}, \epsilon = 10^{-8}, decayrate = 0.9$

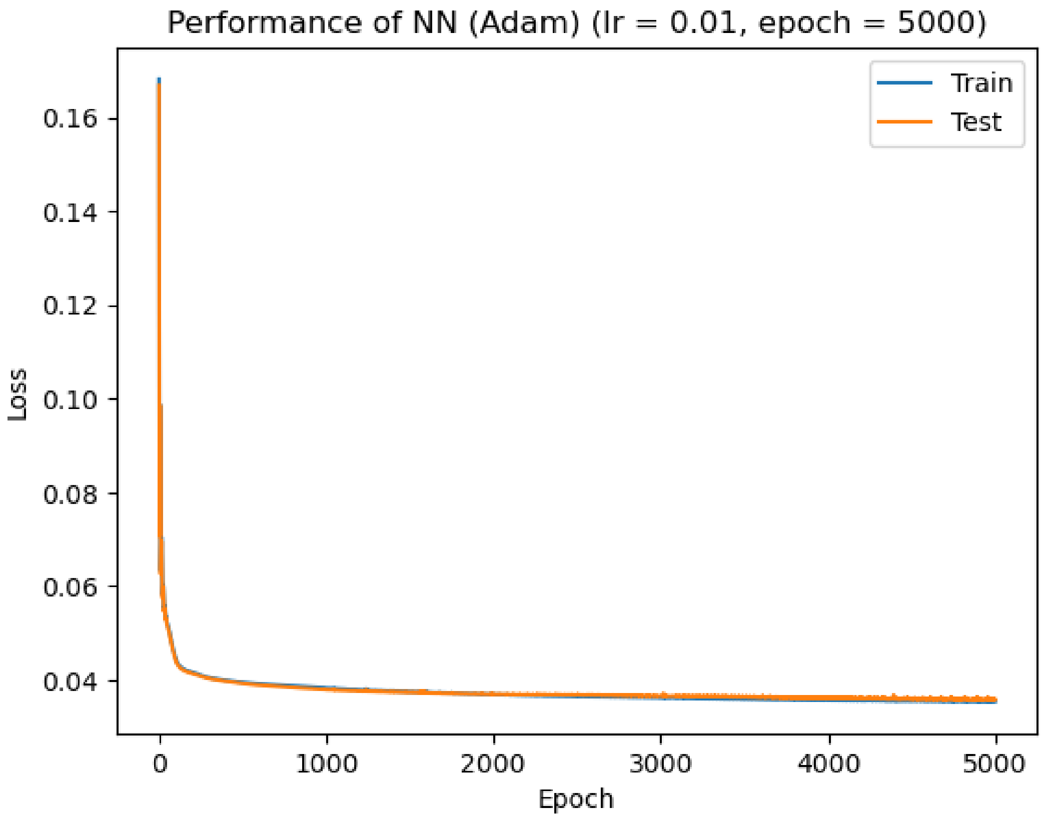


Figure 19: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (Adam), $\alpha = 10^{-2}, \epsilon = 10^{-8}, \beta_1 = 0.9, \beta_2 = 0.999$

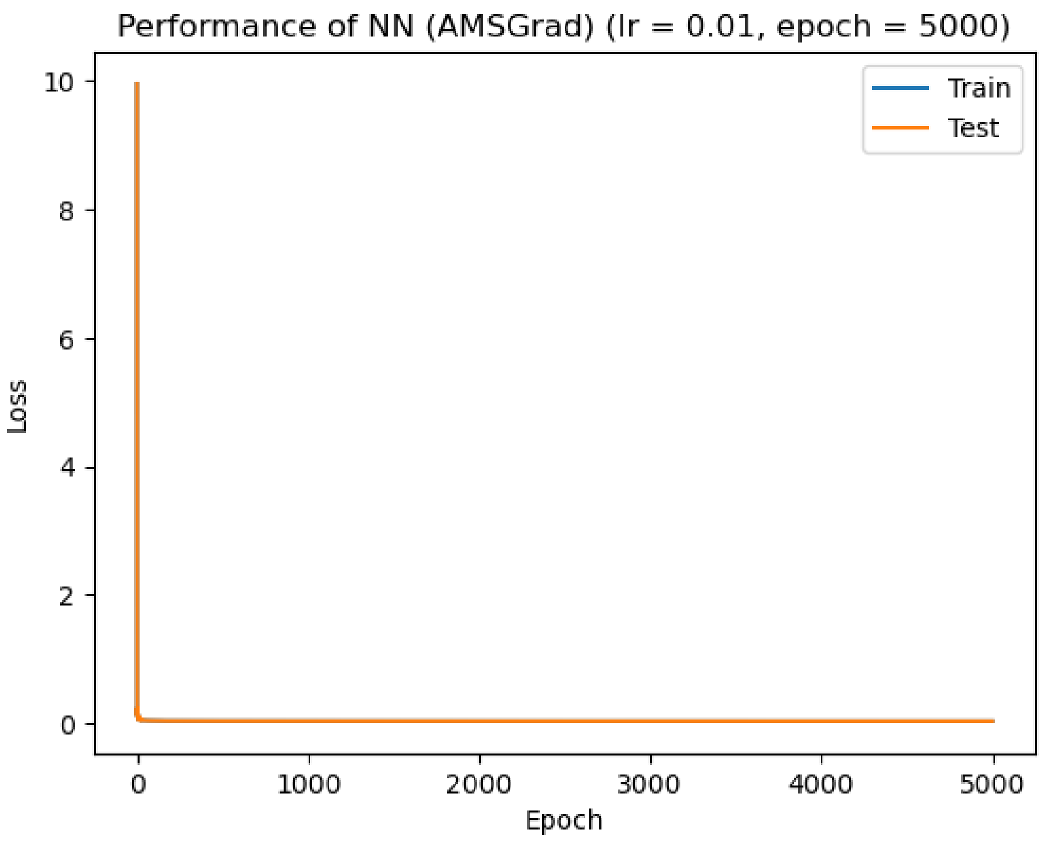


Figure 20: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (AMSGrad), $\alpha = 10^{-2}, \epsilon = 10^{-8}, \beta_1 = 0.9, \beta_2 = 0.999$

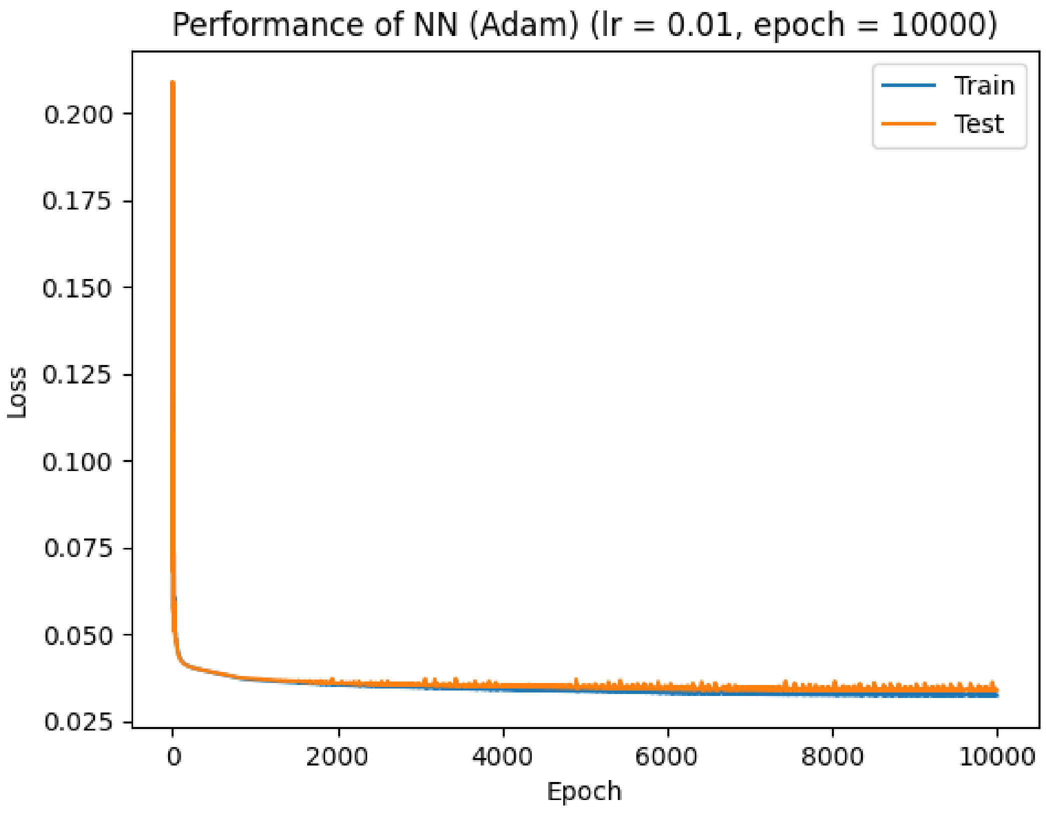


Figure 21: Train and Test Loss vs Epochs (10.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (Adam), $\alpha = 10^{-2}, \epsilon = 10^{-8}, \beta_1 = 0.9, \beta_2 = 0.999$

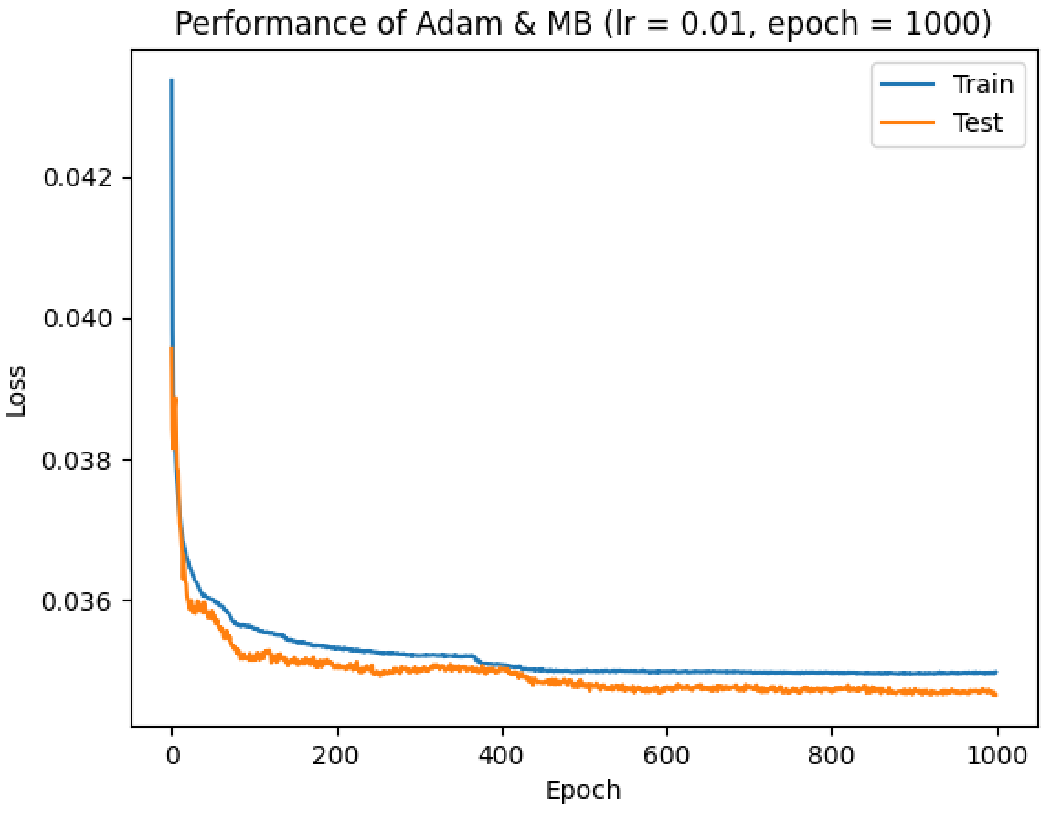


Figure 22: Train and Test Loss vs Epochs (1.000) for Mini-Batch 3 Layer (30, 10, 1) Neural Network (Adam), $\alpha = 10^{-2}, \epsilon = 10^{-8}, \beta_1 = 0.9, \beta_2 = 0.999$

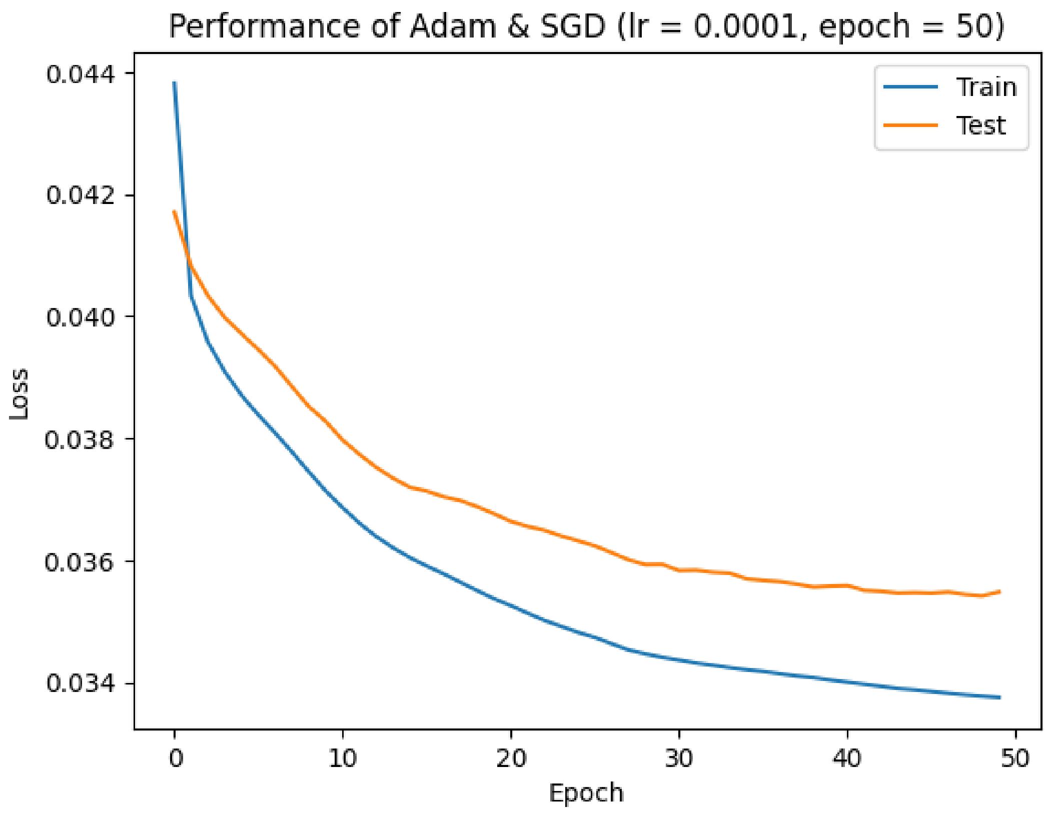


Figure 23: Train and Test Loss vs Epochs (50) for SGD 3 Layer (30, 10, 1) Neural Network (Adam), $\alpha = 10^{-4}, \epsilon = 10^{-8}, \beta_1 = 0.9, \beta_2 = 0.999$

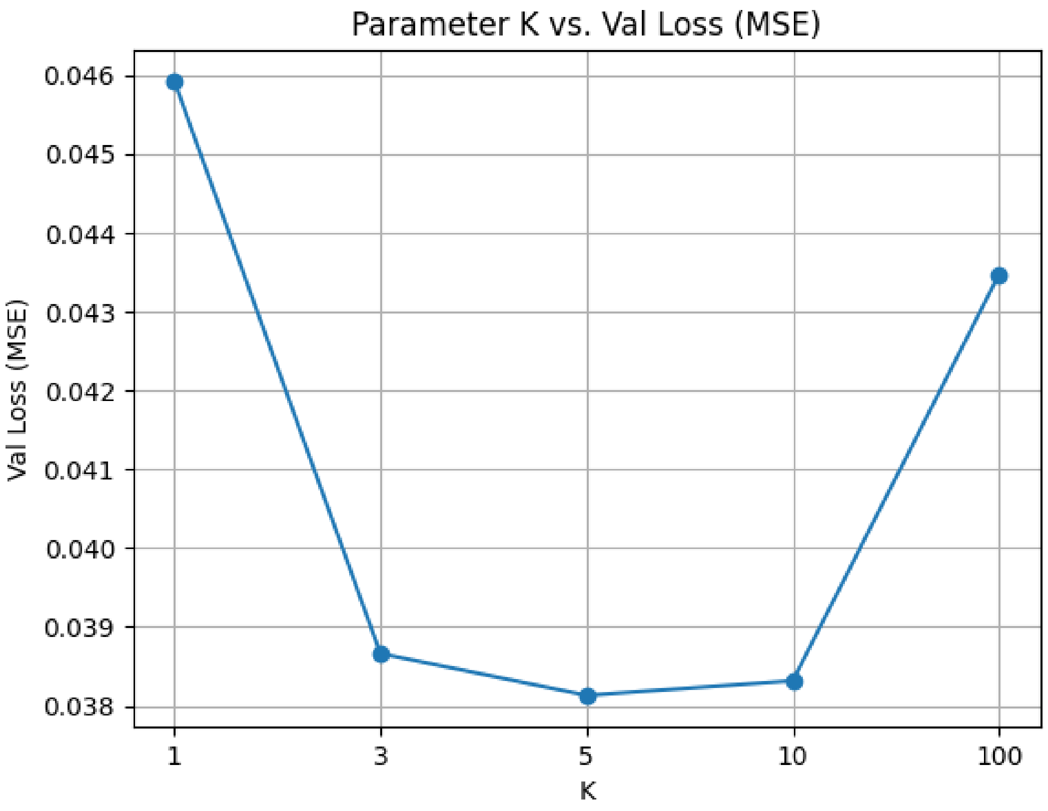


Figure 24: Hyperparameter K vs. Test Loss (MSE) for KNN Regression ($K \in \{1, 3, 5, 10, 100\}$)

8.3 Appendix C - Codes

```python
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

dataPath = "./data/dataset.csv"
df = pd.read_csv(dataPath, index_col=0)

columns = list(df.columns)
columnsToKeep = columns[4: -1]

df = df[columnsToKeep]

# Bool to numerical data for explicit row
df.loc[:, 'explicit'] = df['explicit'].astype(int)

# One hot encoding for nominal categroies
df = pd.get_dummies(df, columns=['key', 'time_signature'], dtype=int)

def min_max_scaling(df):
    min_vals = df.min()
    max_vals = df.max()

    feature_range = max_vals - min_vals

    # Check if any feature has zero range
    zero_range_features = feature_range[feature_range == 0].index

    # Remove features with zero range from normalization
    valid_features = feature_range[feature_range != 0].index
    df_normalized = (df[valid_features] - min_vals[valid_features]) /
       ↪ feature_range[valid_features]

    # Concatenate back the zero range features
    if not zero_range_features.empty:
        df_normalized = pd.concat([df_normalized,
           ↪ df[zero_range_features]], axis=1)

    return df_normalized

def standard_scaling(df):
    mean = df.mean()
    std = df.std()
    return (df - mean) / std

responseFrame = df.pop('valence')
predictorFrame = df

# Min-Max scaling for predictor variables
df_normalized = min_max_scaling(predictorFrame)

# Standard scaling for predictor variables
df_standardized = standard_scaling(predictorFrame)

predictorFrame_scaled = df_normalized
```

Listing 1: Preprocessing Code for all Traditional Methods

```
    responseData = responseFrame.to_numpy()
    predictorData = predictorFrame_scaled.to_numpy()

    trainSplit = 0.8
    valSplit = 0.1
    testSplit = 0.1

    np.random.seed(42)
    indices = np.arange(len(predictorData))
    np.random.shuffle(indices)
    trainIndices = indices[:int(trainSplit * len(indices))]
    valIndices = indices[int(trainSplit* len(indices)):int((trainSplit +
        ↪ valSplit) * len(indices))]
    testIndices = indices[int((trainSplit + valSplit) * len(indices)):]

    trainPredictor, testPredictor, valPredictor =
        ↪ predictorData[trainIndices], predictorData[testIndices],
        ↪ predictorData[valIndices]
    trainResponse, testResponse, valResponse = responseData[trainIndices],
        ↪ responseData[testIndices], responseData[valIndices]
```

Listing 2: Splitting preprocessed data to train, test, validation split code

```
class LinearRegression:
    def __init__(self, lr = 0.01, n_iters: int = 1000):
        self.lr = lr
        self.n_iters = n_iters
        self.weightVector = None
        self.loss_history = []

    def fit(self, X, y):
        num_samples, num_features = X.shape
        biasColumn = np.ones((num_samples, 1))
        designMatrix = np.hstack((biasColumn, X))
        self.weightVector = np.random.rand(num_features + 1)

        # self.normalEquationMethod(designMatrix, y)
        # print(f'Train Loss for Normal Equation Method:',
            ↪ self.lossMSE(designMatrix, y))

        self.gradientDescent(designMatrix, y)
        print('Final Train Loss:', self.lossMSE(designMatrix, y))

    def mse_gradient(self, designMatrix, y):
        predictions = self.predict(designMatrix)
        return  -(2/y.size) * np.dot(designMatrix.T, (y - predictions))

    def gradientDescent(self, designMatrix, y):
        for i in range(self.n_iters):
            # Calculate predictions
            gradientVector = self.mse_gradient(designMatrix, y)

            # Update weights and bias
            self.weightVector = self.weightVector - self.lr *
                ↪ gradientVector

            # Print gradients for debugging
            loss = self.lossMSE(designMatrix, y)
```

```python
            print(f'Train Loss at iteration {i}:', loss)
            self.loss_history.append((i, loss))

        return self

    def normalEquationMethod(self, designMatrix, y):
        # self.weightVector = np.linalg.inv(np.matmul(designMatrix.T, designMatrix)).dot(designMatrix.T).dot(y)
        # return self

         # Compute the SVD of the design matrix
        U, S, Vt = np.linalg.svd(designMatrix, full_matrices=False)

        # Compute the pseudo-inverse
        S_inv = np.diag(1 / S)
        pseudo_inverse = np.dot(np.dot(Vt.T, S_inv), U.T)

        # Calculate the weight vector
        self.weightVector = np.dot(pseudo_inverse, y)

        return self

    def predict(self, designMatrix):
        return np.dot(designMatrix, self.weightVector)

    def inference(self, testData):
        num_samples = testData.shape[0]
        biasColumn = np.ones((num_samples, 1))
        designMatrix = np.hstack((biasColumn, testData))
        return np.dot(designMatrix, self.weightVector)

    def lossMSE(self, designMatrix, y):
        predictions = self.predict(designMatrix)
        error = y - predictions
        squaredError = np.dot(error.T, error)
        meanSquaredError = 1/(y.size) * squaredError
        return meanSquaredError

    def plot_loss_history(self):
        iterations, losses = zip(*self.loss_history)
        plt.plot(iterations, losses)
        plt.xlabel('Iteration')
        plt.ylabel('Training Loss')
        plt.title(f'Training Loss over Iterations - Linear (lr = {self.lr}, iter = {self.n_iters})')
        plt.show()

    def save_model(self, filename):
        np.savez(filename, weights=self.weightVector)

    def load_model(self, filename):
        data = np.load(filename)
        self.weightVector = data['weights']
```

Listing 3: Linear Regression Code

```python
# Training
import time

regressor = LinearRegression(lr = 1e-1, n_iters= 20000)
start = time.time()
# regressor.fit(trainPredictor, trainResponse)
regressor.load_model('linear_reg_model.npz')
end = time.time()
print(f'Time elapsed: {end - start:.4}')
# regressor.save_model('linear_reg_model.npz')

regressor.plot_loss_history()

predictions = regressor.inference(testPredictor)

# Validation Performance
def mse(testResponse, predictions):
    error = testResponse - predictions
    squaredError = np.dot(error.T, error)
    meanSquaredError = 1/(testResponse.size) * squaredError
    return meanSquaredError

def r2_score(testResponse, predictions):
    mean_observed = np.mean(testResponse)
    total_sum_squares = np.sum((testResponse - mean_observed) ** 2)
    residual_sum_squares = np.sum((testResponse - predictions) ** 2)
    r2 = 1 - (residual_sum_squares / total_sum_squares)

    return r2

predictions = regressor.inference(valPredictor)
MSE = mse(valResponse, predictions)
R2 = r2_score(valResponse, predictions)
print(f'Validation Performance (MSE): {MSE}')
print(f'Validation Performance (R2): {R2}')

# Test Performance
predictions = regressor.inference(testPredictor)
MSE = mse(testResponse, predictions)
R2 = r2_score(testResponse, predictions)
print(f'Test Performance (MSE): {MSE}')
print(f'Test Performance (R2): {R2}')

# Demo prediction
demoInstanceLoc = 11401
demoPredictor = predictorData[demoInstanceLoc]
demoResponse = responseData[demoInstanceLoc]
demoPredictor = np.expand_dims(demoPredictor, axis=0)
demoPrediction = regressor.inference(demoPredictor)
print('Prediction:', demoPrediction, 'Response:', demoResponse)
```

Listing 4: Training, Validation and Test Performance of Linear Regression code

```python
class RidgeRegression:
    def __init__(self, lr = 0.01, lambdaConstant = 0.1, n_iters: int = 1000):
        self.lr = lr
        self.n_iters = n_iters
        self.lambdaConstant = lambdaConstant
        self.weightVector = None
        self.loss_history = []

    def fit(self, X, y):
        num_samples, num_features = X.shape
        biasColumn = np.ones((num_samples, 1))
        designMatrix = np.hstack((biasColumn, X))
        self.weightVector = np.random.rand(num_features + 1)

        # # Result by matrix formula
        # self.normalEquationMethod(designMatrix, y)
        # print(f'Train Loss for Normal Equation Method:', self.lossRidgeMSE(designMatrix, y))

        # Result by gradient descent
        self.gradientDescent(designMatrix, y)
        print('Final Train Loss:', self.lossRidgeMSE(designMatrix, y))

    def ridgeMSE_gradient(self, designMatrix, y):
        return self.mse_gradient(designMatrix, y) + 2 * self.lambdaConstant * self.weightVector

    def mse_gradient(self, designMatrix, y):
        predictions = self.predict(designMatrix)
        return  -(2/y.size) * np.dot(designMatrix.T, (y - predictions))

    def gradientDescent(self, designMatrix, y):
        for i in range(self.n_iters):
            # Calculate predictions
            gradientVector = self.ridgeMSE_gradient(designMatrix, y)
            self.weightVector = self.weightVector - self.lr * gradientVector
            loss = self.lossRidgeMSE(designMatrix, y)
            print(f'Train Loss at iteration {i}:', loss)
            self.loss_history.append((i, loss))

        return self

    def normalEquationMethod(self, designMatrix, y):
        identityMatrix = np.identity(designMatrix.shape[1])
        # To avoid regularizing bias when standardization not applied
        identityMatrix[0][0] = 0
        self.weightVector = np.linalg.inv(designMatrix.T.dot(designMatrix) + self.lambdaConstant * identityMatrix).dot(designMatrix.T).dot(y)
        return self

    def predict(self, designMatrix):
        return np.dot(designMatrix, self.weightVector)

    def inference(self, testData):
        num_samples = testData.shape[0]
        biasColumn = np.ones((num_samples, 1))
        designMatrix = np.hstack((biasColumn, testData))
        return np.dot(designMatrix, self.weightVector)

    def lossMSE(self, designMatrix, y):
        predictions = self.predict(designMatrix)
        error = y - predictions
        squaredError = np.dot(error.T, error)
        meanSquaredError = 1/(y.size) * squaredError
        return meanSquaredError

    def lossRidgeMSE(self, designMatrix, y):
        mse = self.lossMSE(designMatrix, y)
        ridge_mse = mse + self.lambdaConstant * np.dot(self.weightVector,
           ↪ self.weightVector)
        return ridge_mse

    def plot_loss_history(self):
        iterations, losses = zip(*self.loss_history)
        plt.plot(iterations, losses)
        plt.xlabel('Iteration')
        plt.ylabel('Training Loss')
        plt.title(f'Training Loss over Iterations - Ridge (lr = {self.lr},
           ↪ iter = {self.n_iters}), lambda = {self.lambdaConstant}')
        plt.show()

    def save_model(self, filename):
        np.savez(filename, weights=self.weightVector)

    def load_model(self, filename):
        data = np.load(filename)
        self.weightVector = data['weights']
```

Listing 5: Ridge Regression Code

```
# Training
import time

regressor = RidgeRegression(lr = 1e-1, lambdaConstant=1e-4, n_iters= 20000)
start = time.time()
# regressor.fit(trainPredictor, trainResponse)
regressor.load_model('ridge_reg_model.npz')
end = time.time()
print(f'Time elapsed: {end - start:.4}')
# regressor.save_model('ridge_reg_model.npz')

# regressor.plot_loss_history()

# Validation Performance

def meanSquaredError(testResponse, predictions):
    error = testResponse - predictions
    squaredError = np.dot(error.T, error)
    meanSquaredError =  1/(testResponse.size) * squaredError
    return meanSquaredError

def r2_score(testResponse, predictions):
    mean_observed = np.mean(testResponse)
    total_sum_squares = np.sum((testResponse - mean_observed) ** 2)
    residual_sum_squares = np.sum((testResponse - predictions) ** 2)
    r2 = 1 - (residual_sum_squares / total_sum_squares)

    return r2

predictions = regressor.inference(valPredictor)
MSE = meanSquaredError(valResponse, predictions)
R2 = r2_score(valResponse, predictions)
print(f'Validation Performance (MSE): {MSE}')
print(f'Validation Performance (R2): {R2}')

# Measuring Performance

predictions = regressor.inference(testPredictor)
MSE = meanSquaredError(testResponse, predictions)
R2 = r2_score(testResponse, predictions)
print(f'Test Performance (MSE): {MSE}')
print(f'Test Performance (R2): {R2}')

demoInstanceLoc = 11401
demoPredictor = predictorData[demoInstanceLoc]
demoResponse = responseData[demoInstanceLoc]
demoPredictor = np.expand_dims(demoPredictor, axis=0)
demoPrediction = regressor.inference(demoPredictor)
print('Prediction:', demoPrediction, 'Response:', demoResponse)
```

Listing 6: Training, Validation and Test Performance of Ridge Regression code

```
class LassoRegression:
    def __init__(self, lr=0.01, lambdaConstant=0.1, n_iters=1000):
        self.lr = lr
        self.n_iters = n_iters
        self.lambdaConstant = lambdaConstant
        self.weightVector = None
        self.loss_history = []

    def fit(self, X, y):
        num_samples, num_features = X.shape
        biasColumn = np.ones((num_samples, 1))
        designMatrix = np.hstack((biasColumn, X))
        self.weightVector = np.random.rand(num_features + 1)

        self.subgradientDescent(designMatrix, y)
        print('Final Train Loss:', self.lossLassoMSE(designMatrix, y))

    def mse_gradient(self, designMatrix, y):
        predictions = self.predict(designMatrix)
        return  -(2/y.size) * np.dot(designMatrix.T, (y - predictions))

    def subgradientDescent(self, designMatrix, y):
        for i in range(self.n_iters):
            gradientVector = self.lassoMSE_subgradient(designMatrix, y)

            # Update weights using subgradient descent
            self.weightVector = self.weightVector - self.lr *
                ↪ gradientVector
            loss = self.lossLassoMSE(designMatrix, y)
            print(f'Train Loss at iteration {i}:', loss)
            self.loss_history.append((i, loss))

        return self

    def lassoMSE_subgradient(self, designMatrix, y):
        mse_gradient = self.mse_gradient(designMatrix, y)
        lasso_gradient = np.sign(self.weightVector)
        return mse_gradient + self.lambdaConstant * lasso_gradient

    def predict(self, designMatrix):
        return np.dot(designMatrix, self.weightVector)

    def inference(self, testData):
        num_samples = testData.shape[0]
        biasColumn = np.ones((num_samples, 1))
        designMatrix = np.hstack((biasColumn, testData))
        return np.dot(designMatrix, self.weightVector)

    def lossMSE(self, designMatrix, y):
        predictions = self.predict(designMatrix)
        error = y - predictions
        squaredError = np.dot(error.T, error)
        meanSquaredError = 1/(y.size) * squaredError
        return meanSquaredError

    def lossLassoMSE(self, designMatrix, y):
        mse = self.lossMSE(designMatrix, y)
        lasso_mse = mse + self.lambdaConstant *
           ↪ np.sum(np.abs(self.weightVector))
        return lasso_mse

    def plot_loss_history(self):
        iterations, losses = zip(*self.loss_history)
        plt.plot(iterations, losses)
        plt.xlabel('Iteration')
        plt.ylabel('Training Loss')
        plt.title(f'Training Loss over Iterations - Lasso (lr = {self.lr},
           ↪ iter = {self.n_iters}, lambda = {self.lambdaConstant})')
        plt.show()

    def save_model(self, filename):
        np.savez(filename, weights=self.weightVector)

    def load_model(self, filename):
        data = np.load(filename)
        self.weightVector = data['weights']
```
Listing 7: Lasso Regression code

```python
# Train
import time

regressor = LassoRegression(lr = 1e-1, lambdaConstant=1e-4, n_iters= 20000)

start = time.time()
# regressor.fit(trainPredictor, trainResponse)
regressor.load_model('lasso_reg_model.npz')
end = time.time()

# Save model
# regressor.save_model('lasso_reg_model.npz')

print(f'Time elapsed: {end - start:.4}')

# regressor.plot_loss_history()

# Validation Performance
def meanSquaredError(testResponse, predictions):
    error = testResponse - predictions
    squaredError = np.dot(error.T, error)
    meanSquaredError =  1/(testResponse.size) * squaredError
    return meanSquaredError

def r2_score(testResponse, predictions):
    mean_observed = np.mean(testResponse)
    total_sum_squares = np.sum((testResponse - mean_observed) ** 2)
    residual_sum_squares = np.sum((testResponse - predictions) ** 2)
    r2 = 1 - (residual_sum_squares / total_sum_squares)

    return r2

predictions = regressor.inference(valPredictor)
MSE = meanSquaredError(valResponse, predictions)
R2 = r2_score(valResponse, predictions)
print(f'Validation Performance (MSE): {MSE}')
print(f'Validation Performance (R2): {R2}')

# Measuring Performance
predictions = regressor.inference(testPredictor)
MSE = meanSquaredError(testResponse, predictions)
R2 = r2_score(testResponse, predictions)
print(f'Test Performance (MSE): {MSE}')
print(f'Test Performance (R2): {R2}')

demoInstanceLoc = 11021
demoPredictor = predictorData[demoInstanceLoc]
demoResponse = responseData[demoInstanceLoc]
demoPredictor = np.expand_dims(demoPredictor, axis=0)
print(demoPredictor.T.shape)
demoPrediction = regressor.inference(demoPredictor)
print('Prediction:', demoPrediction, 'Response:', demoResponse)
```

Listing 8: Training, Validation and Test Performance of Lasso Regression code

```
def relu(z):
    return np.maximum(0, z)

def tanh(z):
    return np.tanh(z)

def linear(z):
    return z

def reluDer(z):
    return np.where(z > 0, 1, 0)

def tanhDer(z):
    return 1 - z**2

def linearDer(z):
    return 1

activationDict = {'relu': relu, 'tanh': tanh, 'linear': linear}
activationDerivativeDict = {'relu': reluDer, 'tanh': tanhDer, 'linear':
    ↪ linearDer}
```

Listing 9: Neural Network Activation Functions code

```
class Layer:
    def __init__(self, inputNumNeuron, numNeurons, activationName,
        ↪ batchSize):
        self.batchSize = batchSize
        self.inputNumNeuron = inputNumNeuron
        self.numNeurons = numNeurons
        self.activationName = activationName
        self.activation = activationDict[self.activationName]
        self.activationDerivative =
            ↪ activationDerivativeDict[self.activationName]
        self.dZ_state = np.empty((numNeurons, batchSize))
        self.Z_state = np.empty((numNeurons, batchSize))
        self.A_state = np.empty((numNeurons, batchSize))
        self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.db_state = np.zeros((self.numNeurons, 1))
        self.initWeights()

    def initWeights(self):
        # Random initialization unfortunately failed.
        # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
        # self.b = np.random.randn(self.numNeurons, 1)

        # Xavier initialization for weights
        self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
            ↪ np.sqrt(1 / self.inputNumNeuron)
        # Initializing biases with zeros
        self.b = np.zeros((self.numNeurons, 1))

    def updateForwardState(self, inputToLayer):
        # print('\nupdateForwardState():\n', 'inputToLayer:',
            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
            ↪ self.b.shape)
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
```

```python
        self.Z_state = inducedLocal
        self.A_state = output
        return output

    def predict(self, inputToLayer, printVals=False):
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        if printVals:
            print('inp:', self.b, self.W, 'out:', output)
        return output

    def updateDeltaState(self, dA):
        # Derivative of loss over the weihts of this layer
        # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
           ↪ 'self.Z_state', self.Z_state.shape)
        derActivation = self.activationDerivative(self.Z_state)
        dZ = np.multiply(dA, derActivation)
        self.dZ_state = dZ

    def calculateChange(self, A_input):
        self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
           ↪ A_input.T)
        self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
           ↪ axis=1, keepdims=True)
        # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
           ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
           ↪ 'self.Z_state', self.Z_state)
        # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
           ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
           ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)

    def updateWeightsAndBias(self, lr):
        self.W = self.W - lr * self.dW_state
        self.b = self.b - lr * self.db_state
```
Listing 10: Neural Network Layer Class code

```python
class NeuralNetwork:
    def __init__(self):
        self.layers = []
        self.loss_history = []
        self.test_loss_history = []

    def addLayer(self, layer):
        self.layers.append(layer)

    def loss(self, predictions, y):
        # MSE
        batchSize = y.size
        error = y - predictions
        squaredError = np.dot(error.T, error)
        mse = (1 / batchSize) * squaredError
        return mse

    def lossDer(self, predictions, y):
        # MSE Derivative
        batchSize = y.size
        error = y - predictions
        mseDer = (-2 / batchSize) * np.sum(error, axis=0, keepdims=True)
        return mseDer

    def predict(self, testPredictor):
        output = testPredictor
        for layer in self.layers:
            printVals = True if False else False
            output = layer.predict(output, printVals)
        return output

    def forward(self, trainPredictor):
        output = trainPredictor
        for layer in self.layers:
            output = layer.updateForwardState(output)
        return output

    def backprop(self, predictions, y, x, lr):
        # Update Delta State
        for layerNumber in reversed(range(len(self.layers))):
            layer = self.layers[layerNumber]
            inputToLayer = self.layers[layerNumber - 1].A_state if
               ↪ layerNumber > 0 else x

            # Output Layer
            if(layer == self.layers[-1]):
                y_reshaped = np.reshape(y, (1, y.size))
                lossDerivative = self.lossDer(predictions, y_reshaped)
                # print('\nbackpropFirst():\n', 'predictions:',
                   ↪ predictions.shape, 'y_reshaped:', y_reshaped.shape,
                   ↪ 'lossDerivative:', lossDerivative.shape,
                   ↪ 'inputToLayer', inputToLayer.shape)
                layer.updateDeltaState(lossDerivative)
                layer.calculateChange(inputToLayer)
            # Hidden Layers
            else:
                dZ_next = nextLayer.dZ_state
                W_next = nextLayer.W
                dA = np.dot(W_next.T, dZ_next)
                # print('\nbackpropAlt():\n', 'dZ_next:', dZ_next.shape,
                   ↪ 'W_next', W_next.shape, 'dA:', dA.shape)
                layer.updateDeltaState(dA)
                layer.calculateChange(inputToLayer)

            nextLayer = layer

        # Update Weights and Bias
        for layerNumber in range(len(self.layers)):
            layer = self.layers[layerNumber]
            layer.updateWeightsAndBias(lr)

    def fit(self, mini_batches_x, mini_batches_y, mini_test_x,
        ↪ mini_test_y, lr=1e-2, epochAmount=10):
        for epoch in range(epochAmount):
            print('-----------EPOCH-----------    -----> ', epoch + 1)
            # Train using mini-batches
            for mini_batch_X, mini_batch_Y in zip(mini_batches_x,
                ↪ mini_batches_y):
                predictions = self.forward(mini_batch_X)
                self.backprop(predictions, mini_batch_Y, mini_batch_X, lr)

            predictions = self.predict(mini_batch_X)
            trainLoss = np.squeeze(self.loss(predictions.T,
                ↪ mini_batch_Y.T))
            print(f'Train Loss:', trainLoss)
            self.loss_history.append((epoch, trainLoss))

            testError = np.squeeze(self.testLoss(mini_test_x.T,
                ↪ mini_test_y))
            self.test_loss_history.append((epoch, testError))

        print('Final Train Loss:', trainLoss)

    def testLoss(self, test_x, test_y):
        predictions = np.squeeze(self.predict(test_x))
        lossMSE = self.loss(predictions, test_y)
        return lossMSE

    def testLossR2(self, test_x, test_y):
        predictions = np.squeeze(self.predict(test_x))
        mean_observed = np.mean(test_y)
        total_sum_squares = np.sum((test_y - mean_observed) ** 2)
        residual_sum_squares = np.sum((test_y - predictions) ** 2)
        r2 = 1 - (residual_sum_squares / total_sum_squares)
        return r2

    def plot_loss_history(self):
        iterations, losses = zip(*self.loss_history)
        _, testLosses = zip(*self.test_loss_history)
        plt.plot(iterations, losses)
        plt.plot(iterations, testLosses)
        plt.xlabel('Epoch')
        plt.ylabel('Training Loss')
        plt.title(f'Training Loss over Epochs NN (lr = {1e4}, epoch =
            ↪ {5000})')
        plt.legend(['Train', 'Test'])
        plt.show()

    def save_weights(self, filename):
        # Create a dictionary to hold weights and biases of all layers
        weights_dict = {}
        for i, layer in enumerate(self.layers):
            weights_dict[f"Layer_{i}_W"] = layer.W
            weights_dict[f"Layer_{i}_b"] = layer.b

        # Save the weights dictionary to a file
        np.savez(filename, **weights_dict)

    def load_weights(self, filename):
        # Load the weights dictionary from the file
        data = np.load(filename)

        # Iterate through layers and load weights and biases
        for i, layer in enumerate(self.layers):
            layer.W = data[f"Layer_{i}_W"]
            layer.b = data[f"Layer_{i}_b"]
```

Listing 11: Neural Network code

```
responseData = responseFrame.to_numpy()
predictorData = predictorFrame_scaled.to_numpy()

# Full batch gradient descent
batchSize = predictorData.shape[0]

# Mini batch gradient descent
# batchSize = predictorData.shape[0] // 100

# Stochastic gradient descent
# batchSize = 1

trainSplit = 0.8
valSplit = 0.1
testSplit = 0.1

# np.random.seed(42)
indices = np.arange(len(predictorData))
np.random.shuffle(indices)
trainIndices = indices[:int(trainSplit * len(indices))]
valIndices = indices[int(trainSplit* len(indices)):int((trainSplit +
    ↪ valSplit) * len(indices))]
testIndices = indices[int((trainSplit + valSplit) * len(indices)):]

trainPredictor, testPredictor, valPredictor = predictorData[trainIndices],
    ↪ predictorData[testIndices], predictorData[valIndices]
trainResponse, testResponse, valResponse = responseData[trainIndices],
    ↪ responseData[testIndices], responseData[valIndices]

trainResponse = np.expand_dims(trainResponse, axis=1)

# Function to create mini-batches
def create_mini_batches(data, batch_size):
    mini_batches = []
    data_size = len(data)
    num_batches = data_size // batch_size

    for i in range(num_batches):
        start_idx = i * batch_size
        end_idx = start_idx + batch_size
        mini_batch = data[start_idx:end_idx]
        mini_batches.append(mini_batch.T)

    if data_size % batch_size != 0:
        mini_batch = data[num_batches * batch_size:]
        mini_batches.append(mini_batch.T)

    return np.array(mini_batches)

# Create mini-batches
mini_batches_X = create_mini_batches(trainPredictor, batch_size= batchSize)
mini_batches_Y = create_mini_batches(trainResponse, batch_size= batchSize)
miniTestX = testPredictor
miniTestY = testResponse
miniValX = valPredictor
miniValY = valResponse
```

Listing 12: Neural Network Batch and Data Splitting Code for all NN based methods

```python
import time

# Initialize NeuralNetwork
nn = NeuralNetwork()
nn.addLayer(Layer(mini_batches_X[0].shape[0], 30, 'relu', batchSize))
nn.addLayer(Layer(30, 10, 'relu', batchSize))
nn.addLayer(Layer(10, 1, 'linear', batchSize))

start = time.time()
# nn.fit(mini_batches_X, mini_batches_Y, miniValX, miniValY, lr=1e4,
    ↪ epochAmount=5000)

# Load weights
nn.load_weights('nn_weights.npz')
end = time.time()

# nn.save_weights('nn_weights.npz')

print(f'Time elapsed: {end - start:.4}')

# nn.plot_loss_history()

# Validation Performance
MSE = np.squeeze(nn.testLoss(miniValX.T, miniValY))
R2 = np.squeeze(nn.testLossR2(miniValX.T, miniValY))
print(f'Validation Performance (MSE): {MSE}')
print(f'Validation Performance (R2): {R2}')

# Test Performance
MSE = np.squeeze(nn.testLoss(miniTestX.T, miniTestY))
R2 = np.squeeze(nn.testLossR2(miniTestX.T, miniTestY))
print(f'Test Performance (MSE): {MSE}')
print(f'Test Performance (R2): {R2}')

demoInstanceLoc = 3
demoPredictor = predictorData[demoInstanceLoc]
demoResponse = responseData[demoInstanceLoc]
demoPredictor = np.expand_dims(demoPredictor, axis=0)
demoPrediction = np.squeeze(nn.predict(demoPredictor.T))
print('Prediction:', demoPrediction, 'Response:', demoResponse)
```

Listing 13: Training, Validation and Test Performance of Neural Network code

```python
# Calculate the covariance matrix
cov_matrix = np.cov(df_standardized, rowvar=False)

# Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# Sort eigenvalues and corresponding eigenvectors
sorted_indices = eigenvalues.argsort()[::-1]
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:, sorted_indices]

# Proportion of Variance Explained
explained_variance_ratio = eigenvalues / np.sum(eigenvalues)
print("Proportion of Variance Explained:", explained_variance_ratio)
plt.figure(figsize=(8, 6))
plt.plot(range(1, len(explained_variance_ratio) + 1),
    ↪ explained_variance_ratio, marker='o', linestyle='-')
plt.title('PVE')
plt.xlabel('Principal Component')
plt.ylabel('PVE Ratio')
plt.show()

# Cumulative explained variance
cumulative_variance = np.cumsum(explained_variance_ratio)
print("Cumulative Explained Variance:", cumulative_variance)
target_variance = 0.95
n_components = np.where(cumulative_variance >= target_variance)[0][0] + 1
print("Number of components to retain {}%
    ↪ variance:".format(target_variance * 100), n_components)
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance,
    ↪ marker='o', linestyle='-')
plt.axhline(y=target_variance, color='r', linestyle='--', label='{}%
    ↪ variance'.format(target_variance * 100))
plt.title('Cumulative Explained Variance')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Variance Ratio')
plt.legend()

# Transforming to the new space
transformed_data = np.dot(df_standardized, eigenvectors[:, :n_components])
principal_df = pd.DataFrame(data=transformed_data, columns=[f"PC{i}" for i
    ↪ in range(1, n_components + 1)])

predictorFrame_scaled = principal_df
```

Listing 14: Dimensionality Reduction (PCA) Data Preprocessing for Neural Network code

```python
class NeuralNetwork:
    def __init__(self):
        self.layers = []
        self.loss_history = []
        self.test_loss_history = []

    def addLayer(self, layer):
        self.layers.append(layer)

    def loss(self, predictions, y):
        # MSE
        batchSize = y.size
        error = y - predictions
        squaredError = np.dot(error.T, error)
        mse = (1 / batchSize) * squaredError
        return mse

    def lossDer(self, predictions, y):
        # MSE Derivative
        batchSize = y.size
        error = y - predictions
        mseDer = (-2 / batchSize) * np.sum(error, axis=0, keepdims=True)
        return mseDer

    def predict(self, testPredictor):
        output = testPredictor
        for layer in self.layers:
            printVals = True if False else False
            output = layer.predict(output, printVals)
        return output

    def forward(self, trainPredictor):
        output = trainPredictor
        for layer in self.layers:
            output = layer.updateForwardState(output)
        return output

    def backprop(self, predictions, y, x, lr):
        # Update Delta State
        for layerNumber in reversed(range(len(self.layers))):
            layer = self.layers[layerNumber]
            inputToLayer = self.layers[layerNumber - 1].A_state if
                ↪ layerNumber > 0 else x

            # Output Layer
            if(layer == self.layers[-1]):
                y_reshaped = np.reshape(y, (1, y.size))
                lossDerivative = self.lossDer(predictions, y_reshaped)
                # print('\nbackpropFirst():\n', 'predictions:',
                    ↪ predictions.shape, 'y_reshaped:', y_reshaped.shape,
                    ↪ 'lossDerivative:', lossDerivative.shape,
                    ↪ 'inputToLayer', inputToLayer.shape)
                layer.updateDeltaState(lossDerivative)
                layer.calculateChange(inputToLayer)
            # Hidden Layers
            else:
                dZ_next = nextLayer.dZ_state
                W_next = nextLayer.W
                dA = np.dot(W_next.T, dZ_next)
                # print('\nbackpropAlt():\n', 'dZ_next:', dZ_next.shape,
                   ↪ 'W_next', W_next.shape, 'dA:', dA.shape)
                layer.updateDeltaState(dA)
                layer.calculateChange(inputToLayer)

            nextLayer = layer

        # Update Weights and Bias
        for layerNumber in range(len(self.layers)):
            layer = self.layers[layerNumber]
            layer.updateWeightsAndBias(lr)

    def fit(self, mini_batches_x, mini_batches_y, mini_test_x,
       ↪ mini_test_y, lr=1e-2, epochAmount=10):
        for epoch in range(epochAmount):
            total_loss = 0
            num_batches = len(mini_batches_x)
            print('-----------EPOCH-----------    -----> ', epoch + 1)
            # Train using mini-batches
            for mini_batch_X, mini_batch_Y in zip(mini_batches_x,
               ↪ mini_batches_y):
                predictions = self.forward(mini_batch_X)
                self.backprop(predictions, mini_batch_Y, mini_batch_X, lr)

                # Compute loss for this mini-batch and accumulate
                loss = np.squeeze(self.loss(predictions.T, mini_batch_Y.T))
                total_loss += loss

            # Average loss over all mini-batches
            average_loss = total_loss / num_batches
            print(f'Train Loss: {average_loss}')
            self.loss_history.append((epoch, average_loss))

            testError = np.squeeze(self.testLoss(mini_test_x.T,
               ↪ mini_test_y))
            self.test_loss_history.append((epoch, testError))

        print('Final Train Loss:', average_loss)

    def testLoss(self, test_x, test_y):
        predictions = np.squeeze(self.predict(test_x))
        lossMSE = self.loss(predictions, test_y)
        return lossMSE

    def testLossR2(self, test_x, test_y):
        predictions = np.squeeze(self.predict(test_x))
        mean_observed = np.mean(test_y)
        total_sum_squares = np.sum((test_y - mean_observed) ** 2)
        residual_sum_squares = np.sum((test_y - predictions) ** 2)
        r2 = 1 - (residual_sum_squares / total_sum_squares)
        return r2

    def plot_loss_history(self):
        iterations, losses = zip(*self.loss_history)
        _, testLosses = zip(*self.test_loss_history)
        plt.plot(iterations, losses)
        plt.plot(iterations, testLosses)
        plt.xlabel('Epoch')
        plt.ylabel('Training Loss')
        plt.title(f'Training Loss over Epochs NN (lr = {1e4}, epoch =
            ↪ {5000})')
        plt.legend(['Train', 'Test'])
        plt.show()


    def save_weights(self, filename):
        # Create a dictionary to hold weights and biases of all layers
        weights_dict = {}
        for i, layer in enumerate(self.layers):
            weights_dict[f"Layer_{i}_W"] = layer.W
            weights_dict[f"Layer_{i}_b"] = layer.b

        # Save the weights dictionary to a file
        np.savez(filename, **weights_dict)

    def load_weights(self, filename):
        # Load the weights dictionary from the file
        data = np.load(filename)

        # Iterate through layers and load weights and biases
        for i, layer in enumerate(self.layers):
            layer.W = data[f"Layer_{i}_W"]
            layer.b = data[f"Layer_{i}_b"]
```

Listing 15: For SGD and Mini-Batch, Average Loss Calculating Neural Network code

```
class Layer:
    def __init__(self, inputNumNeuron, numNeurons, activationName,
        ↪ batchSize, momentum=0.9):
        self.batchSize = batchSize
        self.inputNumNeuron = inputNumNeuron
        self.numNeurons = numNeurons
        self.activationName = activationName
        self.activation = activationDict[self.activationName]
        self.activationDerivative =
            ↪ activationDerivativeDict[self.activationName]
        self.dZ_state = np.empty((numNeurons, batchSize))
        self.Z_state = np.empty((numNeurons, batchSize))
        self.A_state = np.empty((numNeurons, batchSize))
        self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.db_state = np.zeros((self.numNeurons, 1))
        self.initWeights()
        # Momentum variables
        self.momentum = momentum
        self.dW_state_velocity = np.zeros((self.numNeurons,
            ↪ self.inputNumNeuron))
        self.db_state_velocity = np.zeros((self.numNeurons, 1))

    def initWeights(self):
        # Random initialization unfortunately failed.
        # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
        # self.b = np.random.randn(self.numNeurons, 1)

        # Xavier initialization for weights
        self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
            ↪ np.sqrt(1 / self.inputNumNeuron)
        # Initializing biases with zeros
        self.b = np.zeros((self.numNeurons, 1))

    def updateForwardState(self, inputToLayer):
        # print('\nupdateForwardState():\n', 'inputToLayer:',
            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
            ↪ self.b.shape)
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        self.Z_state = inducedLocal
        self.A_state = output
        return output

    def predict(self, inputToLayer, printVals=False):
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        if printVals:
            print('inp:', self.b, self.W, 'out:', output)
        return output

    def updateDeltaState(self, dA):
        # Derivative of loss over the weihts of this layer
        # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
            ↪ 'self.Z_state', self.Z_state.shape)
        derActivation = self.activationDerivative(self.Z_state)
        dZ = np.multiply(dA, derActivation)
        self.dZ_state = dZ

    def calculateChange(self, A_input):
        self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
            ↪ A_input.T)
        self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
            ↪ axis=1, keepdims=True)
        # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
            ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
            ↪ 'self.Z_state', self.Z_state)
        # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
            ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
            ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)

    def updateWeightsAndBias(self, lr):
        # self.W = self.W - lr * self.dW_state
        # self.b = self.b - lr * self.db_state

        # Update weights and biases with momentum
        self.dW_state_velocity = self.momentum * self.dW_state_velocity +
            ↪ lr * self.dW_state
        self.db_state_velocity = self.momentum * self.db_state_velocity +
            ↪ lr * self.db_state
        self.W = self.W - self.dW_state_velocity
        self.b = self.b - self.db_state_velocity
```

Listing 16: Layer Class for Momentum Code

```python
class Layer:
    def __init__(self, inputNumNeuron, numNeurons, activationName, batchSize, momentum=0.9):
        self.batchSize = batchSize
        self.inputNumNeuron = inputNumNeuron
        self.numNeurons = numNeurons
        self.activationName = activationName
        self.activation = activationDict[self.activationName]
        self.activationDerivative = activationDerivativeDict[self.activationName]
        self.dZ_state = np.empty((numNeurons, batchSize))
        self.Z_state = np.empty((numNeurons, batchSize))
        self.A_state = np.empty((numNeurons, batchSize))
        self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.db_state = np.zeros((self.numNeurons, 1))
        self.initWeights()
        # Nesterov Accelerated Gradient variables
        self.momentum = momentum
        self.velocity_W = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.velocity_b = np.zeros((self.numNeurons, 1))

    def initWeights(self):
        # Random initialization unfortunately failed.
        # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
        # self.b = np.random.randn(self.numNeurons, 1)

        # Xavier initialization for weights
        self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) * np.sqrt(1 / self.inputNumNeuron)
        # Initializing biases with zeros
        self.b = np.zeros((self.numNeurons, 1))

    def updateForwardState(self, inputToLayer):
        # print('\nupdateForwardState():\n', 'inputToLayer:', inputToLayer.shape, 'self.W', self.W.shape, 'self.b', self.b.shape)
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        self.Z_state = inducedLocal
        self.A_state = output
        return output

    def predict(self, inputToLayer, printVals=False):
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        if printVals:
            print('inp:', self.b, self.W, 'out:', output)
        return output

    def updateDeltaState(self, dA):
        # Derivative of loss over the weihts of this layer
        # print('\nupdateDeltaState():\n', 'dA:', dA.shape, 'self.Z_state', self.Z_state.shape)
        derActivation = self.activationDerivative(self.Z_state)
        dZ = np.multiply(dA, derActivation)
        self.dZ_state = dZ

    def calculateChange(self, A_input):
        self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
            ↪ A_input.T)
        self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
            ↪ axis=1, keepdims=True)
        # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
            ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
            ↪ 'self.Z_state', self.Z_state)
        # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
            ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
            ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)

    def updateWeightsAndBias(self, lr):
        # self.W = self.W - lr * self.dW_state
        # self.b = self.b - lr * self.db_state

        # Update weights and biases with Nesterov Accelerated Gradient
        self.velocity_W = self.momentum * self.velocity_W - lr *
            ↪ self.dW_state
        self.velocity_b = self.momentum * self.velocity_b - lr *
            ↪ self.db_state
        self.W += self.velocity_W
        self.b += self.velocity_b
```

Listing 17: Layer Class for Nesterov Code

```
class Layer:
    def __init__(self, inputNumNeuron, numNeurons, activationName,
        ↪ batchSize, epsilon=1e-8):
        self.batchSize = batchSize
        self.inputNumNeuron = inputNumNeuron
        self.numNeurons = numNeurons
        self.activationName = activationName
        self.activation = activationDict[self.activationName]
        self.activationDerivative =
            ↪ activationDerivativeDict[self.activationName]
        self.dZ_state = np.empty((numNeurons, batchSize))
        self.Z_state = np.empty((numNeurons, batchSize))
        self.A_state = np.empty((numNeurons, batchSize))
        self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.db_state = np.zeros((self.numNeurons, 1))
        self.initWeights()
        # AdaGrad variables
        self.epsilon = epsilon
        self.squared_gradient_W = np.zeros((self.numNeurons,
            ↪ self.inputNumNeuron))
        self.squared_gradient_b = np.zeros((self.numNeurons, 1))

    def initWeights(self):
        # Random initialization unfortunately failed.
        # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
        # self.b = np.random.randn(self.numNeurons, 1)

        # Xavier initialization for weights
        self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
            ↪ np.sqrt(1 / self.inputNumNeuron)
        # Initializing biases with zeros
        self.b = np.zeros((self.numNeurons, 1))

    def updateForwardState(self, inputToLayer):
        # print('\nupdateForwardState():\n', 'inputToLayer:',
            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
            ↪ self.b.shape)
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        self.Z_state = inducedLocal
        self.A_state = output
        return output

    def predict(self, inputToLayer, printVals=False):
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        if printVals:
            print('inp:', self.b, self.W, 'out:', output)
        return output

    def updateDeltaState(self, dA):
        # Derivative of loss over the weihts of this layer
        # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
            ↪ 'self.Z_state', self.Z_state.shape)
        derActivation = self.activationDerivative(self.Z_state)
        dZ = np.multiply(dA, derActivation)
        self.dZ_state = dZ

    def calculateChange(self, A_input):
        self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
            ↪ A_input.T)
        self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
            ↪ axis=1, keepdims=True)
        # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
            ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
            ↪ 'self.Z_state', self.Z_state)
        # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
            ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
            ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)

    def updateWeightsAndBias(self, lr):
        # self.W = self.W - lr * self.dW_state
        # self.b = self.b - lr * self.db_state

        # Update weights and biases with AdaGrad
        self.squared_gradient_W += np.square(self.dW_state)
        self.squared_gradient_b += np.square(self.db_state)

        self.W -= lr * (self.dW_state / (np.sqrt(self.squared_gradient_W)
            ↪ + self.epsilon))
        self.b -= lr * (self.db_state / (np.sqrt(self.squared_gradient_b)
            ↪ + self.epsilon))
```

Listing 18: Layer Class for AdaGrad Code

```python
class Layer:
    def __init__(self, inputNumNeuron, numNeurons, activationName,
      ↪ batchSize, epsilon=1e-8, decay_rate=0.9):
        self.batchSize = batchSize
        self.inputNumNeuron = inputNumNeuron
        self.numNeurons = numNeurons
        self.activationName = activationName
        self.activation = activationDict[self.activationName]
        self.activationDerivative =
            ↪ activationDerivativeDict[self.activationName]
        self.dZ_state = np.empty((numNeurons, batchSize))
        self.Z_state = np.empty((numNeurons, batchSize))
        self.A_state = np.empty((numNeurons, batchSize))
        self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.db_state = np.zeros((self.numNeurons, 1))
        self.initWeights()
        # RMSProp variables
        self.epsilon = epsilon
        self.decay_rate = decay_rate
        self.squared_gradient_W = np.zeros((self.numNeurons,
            ↪ self.inputNumNeuron))
        self.squared_gradient_b = np.zeros((self.numNeurons, 1))

    def initWeights(self):
        # Random initialization unfortunately failed.
        # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
        # self.b = np.random.randn(self.numNeurons, 1)

        # Xavier initialization for weights
        self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
            ↪ np.sqrt(1 / self.inputNumNeuron)
        # Initializing biases with zeros
        self.b = np.zeros((self.numNeurons, 1))

    def updateForwardState(self, inputToLayer):
        # print('\nupdateForwardState():\n', 'inputToLayer:',
            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
            ↪ self.b.shape)
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        self.Z_state = inducedLocal
        self.A_state = output
        return output

    def predict(self, inputToLayer, printVals=False):
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        if printVals:
            print('inp:', self.b, self.W, 'out:', output)
        return output

    def updateDeltaState(self, dA):
        # Derivative of loss over the weihts of this layer
        # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
            ↪ 'self.Z_state', self.Z_state.shape)
        derActivation = self.activationDerivative(self.Z_state)
        dZ = np.multiply(dA, derActivation)
        self.dZ_state = dZ

    def calculateChange(self, A_input):
        self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state, A_input.T)
        self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state, axis=1, keepdims=True)
        # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state, 'self.db_state', self.db_state, 'A_input:', A_input.T, 'self.Z_state', self.Z_state)
        # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape, 'self.Z_state', self.Z_state.shape, 'self.dW_state', self.dW_state.shape, 'self.db_state', self.db_state.shape)

    def updateWeightsAndBias(self, lr):
        # self.W = self.W - lr * self.dW_state
        # self.b = self.b - lr * self.db_state

        # Update weights and biases with RMSProp
        self.squared_gradient_W = self.decay_rate * self.squared_gradient_W + (1 - self.decay_rate) * np.square(self.dW_state)
        self.squared_gradient_b = self.decay_rate * self.squared_gradient_b + (1 - self.decay_rate) * np.square(self.db_state)

        self.W -= lr * (self.dW_state / (np.sqrt(self.squared_gradient_W) + self.epsilon))
        self.b -= lr * (self.db_state / (np.sqrt(self.squared_gradient_b) + self.epsilon))
```
Listing 19: Layer Class for RMSProp Code

```python
class Layer:
    def __init__(self, inputNumNeuron, numNeurons, activationName,
        ↪ batchSize, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.batchSize = batchSize
        self.inputNumNeuron = inputNumNeuron
        self.numNeurons = numNeurons
        self.activationName = activationName
        self.activation = activationDict[self.activationName]
        self.activationDerivative =
            ↪ activationDerivativeDict[self.activationName]
        self.dZ_state = np.empty((numNeurons, batchSize))
        self.Z_state = np.empty((numNeurons, batchSize))
        self.A_state = np.empty((numNeurons, batchSize))
        self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.db_state = np.zeros((self.numNeurons, 1))
        self.initWeights()
        # Adam variables
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.moment_W = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.moment_b = np.zeros((self.numNeurons, 1))
        self.velocity_W = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.velocity_b = np.zeros((self.numNeurons, 1))
        self.iteration = 0

    def initWeights(self):
        # Random initialization unfortunately failed.
        # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
        # self.b = np.random.randn(self.numNeurons, 1)

        # Xavier initialization for weights
        self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
            ↪ np.sqrt(1 / self.inputNumNeuron)
        # Initializing biases with zeros
        self.b = np.zeros((self.numNeurons, 1))

    def updateForwardState(self, inputToLayer):
        # print('\nupdateForwardState():\n', 'inputToLayer:',
            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
            ↪ self.b.shape)
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        self.Z_state = inducedLocal
        self.A_state = output
        return output

    def predict(self, inputToLayer, printVals=False):
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        if printVals:
            print('inp:', self.b, self.W, 'out:', output)
        return output

    def updateDeltaState(self, dA):
        # Derivative of loss over the weihts of this layer
        # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
            'self.Z_state', self.Z_state.shape)
        derActivation = self.activationDerivative(self.Z_state)
        dZ = np.multiply(dA, derActivation)
        self.dZ_state = dZ

    def calculateChange(self, A_input):
        self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state, A_input.T)
        self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state, axis=1, keepdims=True)
        # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state, 'self.db_state', self.db_state, 'A_input:', A_input.T, 'self.Z_state', self.Z_state)
        # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape, 'self.Z_state', self.Z_state.shape, 'self.dW_state', self.dW_state.shape, 'self.db_state', self.db_state.shape)

    def updateWeightsAndBias(self, lr):
        # self.W = self.W - lr * self.dW_state
        # self.b = self.b - lr * self.db_state

        # Update weights and biases with Adam
        self.iteration += 1
        self.moment_W = self.beta1 * self.moment_W + (1 - self.beta1) * self.dW_state
        self.moment_b = self.beta1 * self.moment_b + (1 - self.beta1) * self.db_state
        self.velocity_W = self.beta2 * self.velocity_W + (1 - self.beta2) * np.square(self.dW_state)
        self.velocity_b = self.beta2 * self.velocity_b + (1 - self.beta2) * np.square(self.db_state)

        moment_W_hat = self.moment_W / (1 - self.beta1 ** self.iteration)
        moment_b_hat = self.moment_b / (1 - self.beta1 ** self.iteration)
        velocity_W_hat = self.velocity_W / (1 - self.beta2 ** self.iteration)
        velocity_b_hat = self.velocity_b / (1 - self.beta2 ** self.iteration)

        self.W -= lr * moment_W_hat / (np.sqrt(velocity_W_hat) + self.epsilon)
        self.b -= lr * moment_b_hat / (np.sqrt(velocity_b_hat) + self.epsilon)
```

Listing 20: Layer Class for Adam Code

```python
class Layer:
    def __init__(self, inputNumNeuron, numNeurons, activationName, batchSize, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.batchSize = batchSize
        self.inputNumNeuron = inputNumNeuron
        self.numNeurons = numNeurons
        self.activationName = activationName
        self.activation = activationDict[self.activationName]
        self.activationDerivative = activationDerivativeDict[self.activationName]
        self.dZ_state = np.empty((numNeurons, batchSize))
        self.Z_state = np.empty((numNeurons, batchSize))
        self.A_state = np.empty((numNeurons, batchSize))
        self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.db_state = np.zeros((self.numNeurons, 1))
        self.initWeights()
        # AMSGrad variables
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.moment_W = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.moment_b = np.zeros((self.numNeurons, 1))
        self.velocity_W = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.velocity_b = np.zeros((self.numNeurons, 1))
        self.velocity_W_max = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.velocity_b_max = np.zeros((self.numNeurons, 1))
        self.iteration = 0

    def initWeights(self):
        # Random initialization unfortunately failed.
        # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
        # self.b = np.random.randn(self.numNeurons, 1)

        # Xavier initialization for weights
        self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) * np.sqrt(1 / self.inputNumNeuron)
        # Initializing biases with zeros
        self.b = np.zeros((self.numNeurons, 1))

    def updateForwardState(self, inputToLayer):
        # print('\nupdateForwardState():\n', 'inputToLayer:', inputToLayer.shape, 'self.W', self.W.shape, 'self.b', self.b.shape)
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        self.Z_state = inducedLocal
        self.A_state = output
        return output

    def predict(self, inputToLayer, printVals=False):
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        if printVals:
            print('inp:', self.b, self.W, 'out:', output)
        return output

    def updateDeltaState(self, dA):
        # Derivative of loss over the weihts of this layer
        # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
            ↪ 'self.Z_state', self.Z_state.shape)
        derActivation = self.activationDerivative(self.Z_state)
        dZ = np.multiply(dA, derActivation)
        self.dZ_state = dZ

    def calculateChange(self, A_input):
        self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
            ↪ A_input.T)
        self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
            ↪ axis=1, keepdims=True)
        # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
            ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
            ↪ 'self.Z_state', self.Z_state)
        # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
            ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
            ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)

    def updateWeightsAndBias(self, lr):
        # self.W = self.W - lr * self.dW_state
        # self.b = self.b - lr * self.db_state

        # Update weights and biases with AMSGrad
        self.iteration += 1
        self.moment_W = self.beta1 * self.moment_W + (1 - self.beta1) *
            ↪ self.dW_state
        self.moment_b = self.beta1 * self.moment_b + (1 - self.beta1) *
            ↪ self.db_state
        self.velocity_W = self.beta2 * self.velocity_W + (1 - self.beta2)
            ↪ * np.square(self.dW_state)
        self.velocity_b = self.beta2 * self.velocity_b + (1 - self.beta2)
            ↪ * np.square(self.db_state)

        self.velocity_W_max = np.maximum(self.velocity_W_max,
            ↪ self.velocity_W)
        self.velocity_b_max = np.maximum(self.velocity_b_max,
            ↪ self.velocity_b)

        moment_W_hat = self.moment_W / (1 - self.beta1 ** self.iteration)
        moment_b_hat = self.moment_b / (1 - self.beta1 ** self.iteration)

        self.W -= lr * moment_W_hat / (np.sqrt(self.velocity_W_max) +
            ↪ self.epsilon)
        self.b -= lr * moment_b_hat / (np.sqrt(self.velocity_b_max) +
            ↪ self.epsilon)
```

Listing 21: Layer Class for AMSGrad Code

```python
class Layer:
    def __init__(self, inputNumNeuron, numNeurons, activationName,
    ↪ batchSize, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.batchSize = batchSize
        self.inputNumNeuron = inputNumNeuron
        self.numNeurons = numNeurons
        self.activationName = activationName
        self.activation = activationDict[self.activationName]
        self.activationDerivative =
            ↪ activationDerivativeDict[self.activationName]
        self.dZ_state = np.empty((numNeurons, batchSize))
        self.Z_state = np.empty((numNeurons, batchSize))
        self.A_state = np.empty((numNeurons, batchSize))
        self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.db_state = np.zeros((self.numNeurons, 1))
        self.initWeights()
        # Adam variables
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.moment_W = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.moment_b = np.zeros((self.numNeurons, 1))
        self.velocity_W = np.zeros((self.numNeurons, self.inputNumNeuron))
        self.velocity_b = np.zeros((self.numNeurons, 1))
        self.iteration = 0

    def initWeights(self):
        # Random initialization unfortunately failed.
        # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
        # self.b = np.random.randn(self.numNeurons, 1)

        # Xavier initialization for weights
        self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
            ↪ np.sqrt(1 / self.inputNumNeuron)
        # Initializing biases with zeros
        self.b = np.zeros((self.numNeurons, 1))

    def updateForwardState(self, inputToLayer):
        # print('\nupdateForwardState():\n', 'inputToLayer:',
            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
            ↪ self.b.shape)
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        self.Z_state = inducedLocal
        self.A_state = output
        return output

    def predict(self, inputToLayer, printVals=False):
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        if printVals:
            print('inp:', self.b, self.W, 'out:', output)
        return output

    def updateDeltaState(self, dA):
        # Derivative of loss over the weihts of this layer
        # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
            'self.Z_state', self.Z_state.shape)
        derActivation = self.activationDerivative(self.Z_state)
        dZ = np.multiply(dA, derActivation)
        self.dZ_state = dZ

    def calculateChange(self, A_input):
        self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state, A_input.T)
        self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state, axis=1, keepdims=True)
        # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state, 'self.db_state', self.db_state, 'A_input:', A_input.T, 'self.Z_state', self.Z_state)
        # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape, 'self.Z_state', self.Z_state.shape, 'self.dW_state', self.dW_state.shape, 'self.db_state', self.db_state.shape)

    def updateWeightsAndBias(self, lr):
        # self.W = self.W - lr * self.dW_state
        # self.b = self.b - lr * self.db_state

        # Update weights and biases with Adam
        self.iteration += 1
        self.moment_W = self.beta1 * self.moment_W + (1 - self.beta1) * self.dW_state
        self.moment_b = self.beta1 * self.moment_b + (1 - self.beta1) * self.db_state
        self.velocity_W = self.beta2 * self.velocity_W + (1 - self.beta2) * np.square(self.dW_state)
        self.velocity_b = self.beta2 * self.velocity_b + (1 - self.beta2) * np.square(self.db_state)

        moment_W_hat = self.moment_W / (1 - self.beta1 ** self.iteration)
        moment_b_hat = self.moment_b / (1 - self.beta1 ** self.iteration)
        velocity_W_hat = self.velocity_W / (1 - self.beta2 ** self.iteration)
        velocity_b_hat = self.velocity_b / (1 - self.beta2 ** self.iteration)

        self.W -= lr * moment_W_hat / (np.sqrt(velocity_W_hat) + self.epsilon)
        self.b -= lr * moment_b_hat / (np.sqrt(velocity_b_hat) + self.epsilon)

class NeuralNetwork:
    def __init__(self):
        self.layers = []
        self.loss_history = []
        self.test_loss_history = []

    def addLayer(self, layer):
        self.layers.append(layer)

    def loss(self, predictions, y):
        # MSE
        batchSize = y.size
        error = y - predictions
        squaredError = np.dot(error.T, error)
        mse = (1 / batchSize) * squaredError
        return mse

    def lossDer(self, predictions, y):
        # MSE Derivative
        batchSize = y.size
        error = y - predictions
        mseDer = (-2 / batchSize) * np.sum(error, axis=0, keepdims=True)
        return mseDer

    def predict(self, testPredictor):
        output = testPredictor
        for layer in self.layers:
            printVals = True if False else False
            output = layer.predict(output, printVals)
        return output

    def forward(self, trainPredictor):
        output = trainPredictor
        for layer in self.layers:
            output = layer.updateForwardState(output)
        return output

    def backprop(self, predictions, y, x, lr):
        # Update Delta State
        for layerNumber in reversed(range(len(self.layers))):
            layer = self.layers[layerNumber]
            inputToLayer = self.layers[layerNumber - 1].A_state if layerNumber > 0 else x

            # Output Layer
            if(layer == self.layers[-1]):
                y_reshaped = np.reshape(y, (1, y.size))
                lossDerivative = self.lossDer(predictions, y_reshaped)
                # print('\nbackpropFirst():\n', 'predictions:', predictions.shape, 'y_reshaped:', y_reshaped.shape, 'lossDerivative:', lossDerivative.shape, 'inputToLayer', inputToLayer.shape)
                layer.updateDeltaState(lossDerivative)
                layer.calculateChange(inputToLayer)
            # Hidden Layers
            else:
                dZ_next = nextLayer.dZ_state
                W_next = nextLayer.W
                dA = np.dot(W_next.T, dZ_next)
                # print('\nbackpropAlt():\n', 'dZ_next:', dZ_next.shape, 'W_next', W_next.shape, 'dA:', dA.shape)
                layer.updateDeltaState(dA)
                layer.calculateChange(inputToLayer)

            nextLayer = layer

        # Update Weights and Bias
        for layerNumber in range(len(self.layers)):
            layer = self.layers[layerNumber]
            layer.updateWeightsAndBias(lr)

    def fit(self, mini_batches_x, mini_batches_y, mini_test_x,
        ↪ mini_test_y, lr=1e-2, epochAmount=10):
        # self.load_weights('nn_weights_adam_large_343.npz')
        for epoch in range(epochAmount):
            total_loss = 0
            num_batches = len(mini_batches_x)
            print('-----------EPOCH-----------    -----> ', epoch + 1)
            # Train using mini-batches
            for mini_batch_X, mini_batch_Y in zip(mini_batches_x,
                ↪ mini_batches_y):
                predictions = self.forward(mini_batch_X)
                self.backprop(predictions, mini_batch_Y, mini_batch_X, lr)

                # Compute loss for this mini-batch and accumulate
                loss = np.squeeze(self.loss(predictions.T, mini_batch_Y.T))
                total_loss += loss

            # Average loss over all mini-batches
            average_loss = total_loss / num_batches
            print(f'Train Loss: {average_loss}')
            self.loss_history.append((epoch, average_loss))

            testError = np.squeeze(self.testLoss(mini_test_x.T,
                ↪ mini_test_y))
            self.test_loss_history.append((epoch, testError))

        print('Final Train Loss:', average_loss)


    def testLoss(self, test_x, test_y):
        predictions = np.squeeze(self.predict(test_x))
        lossMSE = self.loss(predictions, test_y)
        return lossMSE

    def testLossR2(self, test_x, test_y):
        predictions = np.squeeze(self.predict(test_x))
        mean_observed = np.mean(test_y)
        total_sum_squares = np.sum((test_y - mean_observed) ** 2)
        residual_sum_squares = np.sum((test_y - predictions) ** 2)
        r2 = 1 - (residual_sum_squares / total_sum_squares)
        return r2

    def plot_loss_history(self):
        iterations, losses = zip(*self.loss_history)
        _, testLosses = zip(*self.test_loss_history)
        plt.plot(iterations, losses)
        plt.plot(iterations, testLosses)
        plt.xlabel('Epoch')
        plt.ylabel('Training Loss')
        plt.title(f'Training Loss over Epochs NN (lr = {1e4}, epoch =
            ↪ {5000})')
        plt.legend(['Train', 'Test'])
        plt.show()

    def save_weights(self, filename):
        # Create a dictionary to hold weights and biases of all layers
        weights_dict = {}
        for i, layer in enumerate(self.layers):
            weights_dict[f"Layer_{i}_W"] = layer.W
            weights_dict[f"Layer_{i}_b"] = layer.b

        # Save the weights dictionary to a file
        np.savez(filename, **weights_dict)

    def load_weights(self, filename):
        # Load the weights dictionary from the file
        data = np.load(filename)

        # Iterate through layers and load weights and biases
        for i, layer in enumerate(self.layers):
            layer.W = data[f"Layer_{i}_W"]
            layer.b = data[f"Layer_{i}_b"]
```

Listing 22: Neural Network and Layer Class for Adam and SGD or MB Combinations Code

```
class KNNRegression:
    def __init__(self, k):
        self.k = k

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test, batch_size=200):
        predictions = []
        n_test_samples = X_test.shape[0]
        print('Total batches:', n_test_samples // batch_size)

        cnt = 0
        for i in range(0, n_test_samples, batch_size):
            cnt += 1
            print('Batch processsed:', cnt)
            X_batch = X_test[i:i+batch_size]

            # Euclidean distances between each point
            dists = np.sqrt(np.sum((X_batch[:, np.newaxis] -
                ↪ self.X_train)**2, axis=2))

            # Get indices of the k-nearest neighbors
            nearest_neighbors_indices = np.argsort(dists, axis=1)[:,
                ↪ :self.k]

            # Get the corresponding target values of the nearest neighbors
            nearest_neighbors_targets =
                ↪ self.y_train[nearest_neighbors_indices]

            batch_predictions = np.mean(nearest_neighbors_targets, axis=1)

            predictions.extend(batch_predictions)

        return np.array(predictions)
```

Listing 23: KNN Regression code

```python
def mse(testResponse, predictions):
    error = testResponse - predictions
    squaredError = np.dot(error.T, error)
    meanSquaredError = 1/(testResponse.size) * squaredError
    return meanSquaredError

# Validation Performance
MSEs = []
K = [1, 3, 5, 10, 100]
for k in K:
    knn = KNNRegression(k=k)
    knn.fit(trainPredictor, trainResponse)
    predictions = knn.predict(valPredictor, batch_size=456)
    MSE = mse(valResponse, predictions)
    print("Mean Squared Error (MSE):", MSE)
    MSEs.append(MSE)

x_values = np.arange(len(K))

# Plot K versus MSE
plt.plot(x_values, MSEs, linestyle='-', marker='o')
plt.title('Parameter K vs. Val Loss (MSE)')
plt.xlabel('K')
plt.ylabel('Val Loss (MSE)')
# Set the x-ticks to be the original K values
plt.xticks(x_values, K)
plt.grid(True)
plt.show()

# Test Performance
def r2_score(testResponse, predictions):
    mean_observed = np.mean(testResponse)
    total_sum_squares = np.sum((testResponse - mean_observed) ** 2)
    residual_sum_squares = np.sum((testResponse - predictions) ** 2)
    r2 = 1 - (residual_sum_squares / total_sum_squares)

    return r2

knn = KNNRegression(k=5)
knn.fit(trainPredictor, trainResponse)
predictions = knn.predict(testPredictor, batch_size=456)
MSE = mse(testResponse, predictions)
R2 = r2_score(testResponse, predictions)
print(f'Test Performance (MSE): {MSE}')
print(f'Test Performance (R2): {R2}')

demoInstanceLoc = 11401
demoPredictor = predictorData[demoInstanceLoc]
demoResponse = responseData[demoInstanceLoc]
demoPredictor = np.expand_dims(demoPredictor, axis=0)
demoPrediction = knn.predict(demoPredictor, batch_size=1)
print('Prediction:', demoPrediction, 'Response:', demoResponse)
```

Listing 24: Training, Validation and Test Performance of KNN Regression code

```python
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns

# Preprocess
dataPath = "./data/dataset.csv"
df = pd.read_csv(dataPath, index_col=0)
columns = list(df.columns)
columnsToKeep = columns[4: -1]
columnsToKeep
df = df[columnsToKeep]

# Describe Dataset
df.head()
pd.isnull(df).sum()

# Preprocess
df.loc[:, 'explicit'] = df['explicit'].astype(int)
# One hot encoding for nominal categroies
df = pd.get_dummies(df, columns=['key', 'time_signature'], dtype=int)

def min_max_scaling(df):
    min_vals = df.min()
    max_vals = df.max()

    feature_range = max_vals - min_vals

    # Check if any feature has zero range
    zero_range_features = feature_range[feature_range == 0].index

    # Remove features with zero range from normalization
    valid_features = feature_range[feature_range != 0].index
    df_normalized = (df[valid_features] - min_vals[valid_features]) /
    ↪ feature_range[valid_features]

    # Concatenate back the zero range features
    if not zero_range_features.empty:
        df_normalized = pd.concat([df_normalized,
        ↪ df[zero_range_features]], axis=1)

    return df_normalized

def standard_scaling(df):
    mean = df.mean()
    std = df.std()
    return (df - mean) / std

responseFrame = df.pop('valence')
predictorFrame = df

# Min-Max scaling for predictor variables
df_normalized = min_max_scaling(predictorFrame)

# Standard scaling for predictor variables
df_standardized = standard_scaling(predictorFrame)

```

```
# predictorFrame_scaled = df_normalized
# df.info()


# PCA
# predictorFrame = df
# # Standard scaling for predictor variables
# df_standardized = standard_scaling(predictorFrame)

# # Calculate the covariance matrix
# cov_matrix = np.cov(df_standardized, rowvar=False)

# # Calculate eigenvalues and eigenvectors
# eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)

# # Sort eigenvalues and corresponding eigenvectors
# sorted_indices = eigenvalues.argsort()[::-1]
# eigenvalues = eigenvalues[sorted_indices]
# eigenvectors = eigenvectors[:, sorted_indices]

# # Proportion of Variance Explained
# explained_variance_ratio = eigenvalues / np.sum(eigenvalues)
# print("Proportion of Variance Explained:", explained_variance_ratio)
# plt.figure(figsize=(8, 6))
# plt.plot(range(1, len(explained_variance_ratio) + 1),
    ↪ explained_variance_ratio, marker='o', linestyle='-')
# plt.title('PVE')
# plt.xlabel('Principal Component')
# plt.ylabel('PVE Ratio')
# plt.show()

# # Cumulative explained variance
# cumulative_variance = np.cumsum(explained_variance_ratio)
# print("Cumulative Explained Variance:", cumulative_variance)
# target_variance = 0.95
# n_components = np.where(cumulative_variance >= target_variance)[0][0] + 1
# print("Number of components to retain {}%
    ↪ variance:".format(target_variance * 100), n_components)

# # Transforming to the new space
# transformed_data = np.dot(df_standardized, eigenvectors[:,
    ↪ :n_components])
# principal_df = pd.DataFrame(data=transformed_data, columns=[f"PC{i}" for
    ↪ i in range(1, n_components + 1)])

# # Concatenate the principal components DataFrame with the original
    ↪ DataFrame
# final_df = pd.concat([principal_df, df.reset_index()], axis=1)


df.sort_values('valence', ascending = False).head(10)
df.describe().transpose()

# Correlation heatmap
corr_df = df.corr(method="pearson")

plt.figure(figsize=(12, 9))
heatmap = sns.heatmap(corr_df, annot=True, fmt=".1g",
                              vmin=-1, vmax=1, center=0, cmap="inferno",
                              linewidths=1, linecolor="black")
heatmap.set_title("Correlation Heatmap Between Variables")
heatmap.set_xticklabels(heatmap.get_xticklabels(), rotation=90)
plt.show()

# Correlation of Features (Sampled)
# Sample a smaller subset of data points
sampled_df = df.sample(n=1000, random_state=42)

# List of features to plot
features = ["danceability", "energy", "loudness", "speechiness",
    ↪ "acousticness", "instrumentalness", "liveness", "tempo"]

# Set up the figure with subplots
fig, axes = plt.subplots(nrows=2, ncols=4, figsize=(16, 8))
fig.suptitle("Features vs. Valence", fontsize=16)

# Flatten the axes array for easy iteration
axes = axes.flatten()

# Loop through features and plot each one against valence
for i, feature in enumerate(features):
    sns.regplot(data=sampled_df, y=feature, x="valence", ax=axes[i])
    axes[i].set_title(f"{feature.capitalize()} vs. Valence")

# Adjust layout
plt.tight_layout()
plt.show()

# Histograms
df.hist(figsize=(20, 20))
plt.show()

# Joint Correlation Plot
plt.figure(figsize=(16, 8))
ax = sns.jointplot(x=df['danceability'],y=df["valence"],data=df)
```

Listing 25: Data Analysis code

9 REFERENCES

1. ”Spotify Tracks Dataset,” Kaggle. [Online]. Available: https://www.kaggle.com/datasets/maharshipandya/-spotify-tracks-dataset. [Accessed: Mar. 5, 2024].

2. ”Get Track’s Audio Features,” Spotify for Developers. [Online]. Available: https://developer.spotify.com/documentation/web-api/reference/get-audio-features. [Accessed: Mar. 5, 2024].

3. ”Plotting Music’s Emotional Valence, 1950-2013,” The Echo Nest, Nov. 5, 2013. [Online]. Available: https://web.archive.org/web/20170422195736/http://blog.echonest.com/post/66097438564/plotting-musics-emotional-valence-1950-2013. [Accessed: Mar. 5, 2024].

4. J. Constine, ”Inside The Spotify – Echo Nest Skunkworks,” Tech Crunch, Oct. 19, 2014. [Online]. Available: https://techcrunch.com/2014/10/19/the-sonic-mad-scientists/. [Accessed: Mar. 5, 2024].

5. B. Whitfield, ”A Step-by-Step Explanation of Principal Component Analysis (PCA),” Builtin, Feb. 23, 2024. [Online]. Available: https://builtin.com/data-science/step-step-explanation-principal-component-analysis. [Accessed: Apr. 21, 2024].

6. ”Pseudo-Inverse of a Matrix,” UC Berkeley, Mar. 2, 2021. [Online]. Available: https://inst.eecs.berkeley.edu/ ee127/sp21/livebook/def_pseudo_inv.html. [Accessed: Apr. 21, 2024].