

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 İynem
Student ID: 22002717

May 12, 2024

Contents

1	Introduction to the Problem	1
2	Dataset Analysis	1
2.1	Preprocessing	2
3	Training Methodology	3
3.1	Linear Regression	3
3.1.1	Likelihood Function	3
3.1.2	Maximum Likelihood Estimation	3
3.1.3	Methods for Finding Weights	4
3.2	Ridge Regression	4
3.2.1	Maximum a Posteriori Estimation	4
3.2.2	Methods for Finding Weights	5
3.3	Lasso Regression	5
3.3.1	Maximum a Posteriori Estimation	5
3.3.2	Gradient Descent	6
3.4	Neural Network (MLP) Regression	6
3.4.1	Forward Propagation	6
3.4.2	Backpropagation	6
3.4.3	Gradient Descent	7
3.4.4	Optimization Techniques for Gradients	7
3.5	K-Nearest Neighbors (KNN) Regression	8
3.5.1	Prediction	8
4	Expected & Encountered Challenges	9
5	Performance Analysis	9
6	Simulation Setup & Results	10
7	Conclusion	12
8	APPENDIXES	13
8.1	Appendix A - Dataset	13
8.2	Appendix B - Preliminary Results	18
8.3	Appendix C - Codes	32
9	REFERENCES	72

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 to 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 of the valence (response) feature. As shown above most of the features are within [0, 1] range, yet there are some features that represent categorical variables and different ranges which need 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 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).

$$\frac{x_j - \mu_j}{\sigma_j} \quad (1)$$

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).

$$\frac{x_j - x_{\min_j}}{x_{\max_j} - x_{\min_j}} \quad (2)$$

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):

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

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:

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

We choose the smallest k such that $\text{Cumulative Explained Variance}_k$ 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 (except PCA as it will be applied as a special case) dataset obtains its 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):

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p + \epsilon = \boldsymbol{\beta}^T \mathbf{x} + \epsilon \quad (5)$$

3.1.1 Likelihood Function

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

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

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

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

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:

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

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

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 $\boldsymbol{\beta}$:

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

$$\operatorname{argmin}_{\boldsymbol{\beta}} RSS(\boldsymbol{\beta}) = \operatorname{argmax}_{\boldsymbol{\beta}} (\ell(\boldsymbol{\beta})) \quad (12)$$

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

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

$$\hat{\beta} = \operatorname{argmin}_{\beta} \text{RSS}(\beta) \quad (14)$$

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} :

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

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

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

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

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

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

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

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 posteriori (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 $p(D|\beta) = \mathcal{L}(\beta)$ is same with ordinary linear regression, the prior $p(\beta)$ 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:

$$p(\beta|D) \propto p(D|\beta)p(\beta) \quad (21)$$

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

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

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

$$\hat{\beta} = \operatorname{argmax}_{\beta} p(\beta|D) = \operatorname{argmin}_{\beta} L_{ridge}(\beta, \lambda) \quad (25)$$

3.2.2 Methods for Finding Weights

Normal Equations λ is the regularization parameter and \mathbf{I} is the identity matrix.

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

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:

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

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

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

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

$$p(\beta|D) \propto p(D|\beta)p(\beta) \quad (30)$$

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

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

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

$$\hat{\beta} = \operatorname{argmax}_{\beta} p(\beta|D) = \operatorname{argmin}_{\beta} L_{lasso}(\beta, \lambda) \quad (34)$$

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:

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

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

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

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 $l = 1, 2, \dots, L - 1$

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

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

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

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

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.

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

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

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

3.4.3 Gradient Descent

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

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

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

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 θ instead of β 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.

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

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

- **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.

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

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

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

- **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.

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

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

- **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.

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

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

- **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.

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

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

- **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.

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

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

- **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.

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

- **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.

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

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}_{new} , the predicted target value \hat{y}_{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}_{new} .

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

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 $\forall i = 1, 2, \dots, \text{instanceAmount}$

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

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. Addi-

tionally, 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 validation was not strongly required and train-validation-test split was enough for assessing the performance.

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

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

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

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 structure 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

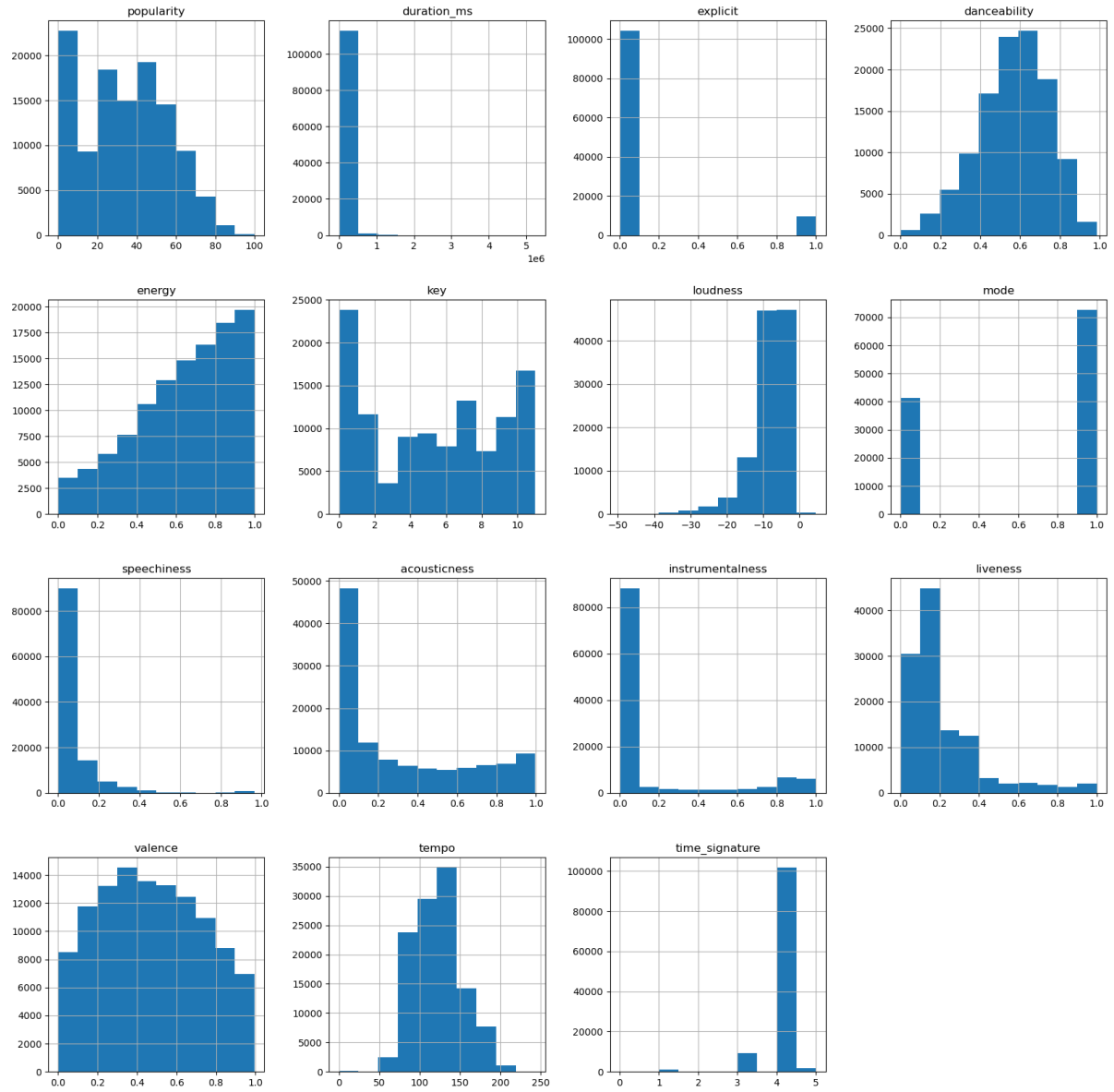


Figure 1: Histograms of features.

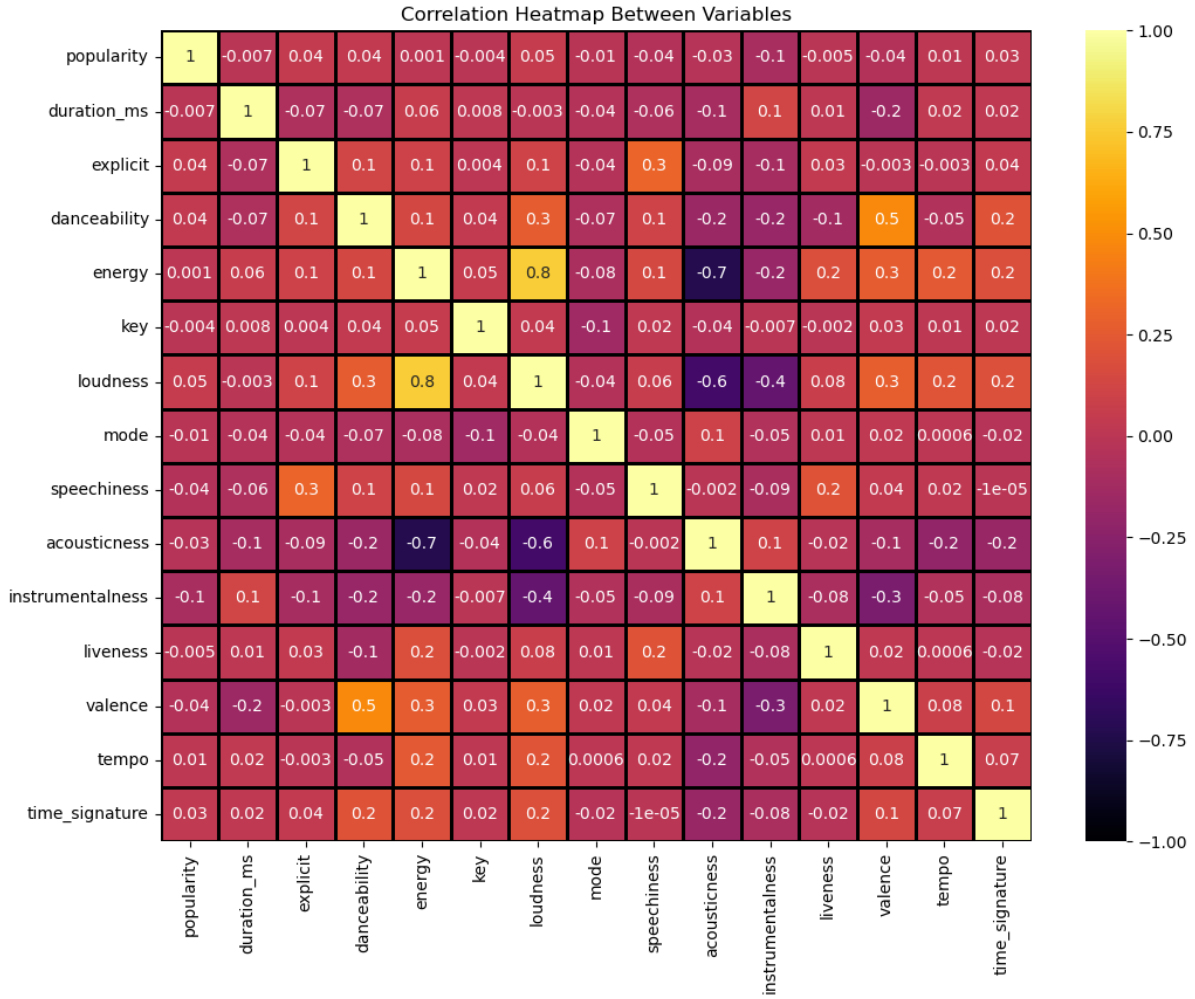


Figure 2: Correlation matrix as heatmap.

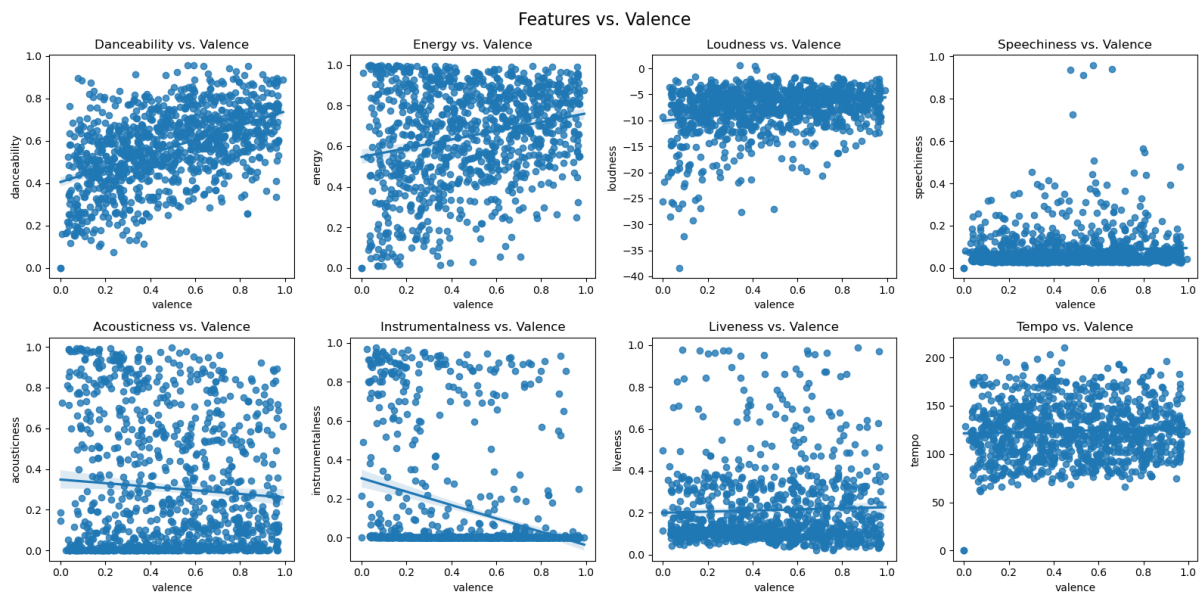


Figure 3: Correlation of key features (1000 instances sampled).

	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.

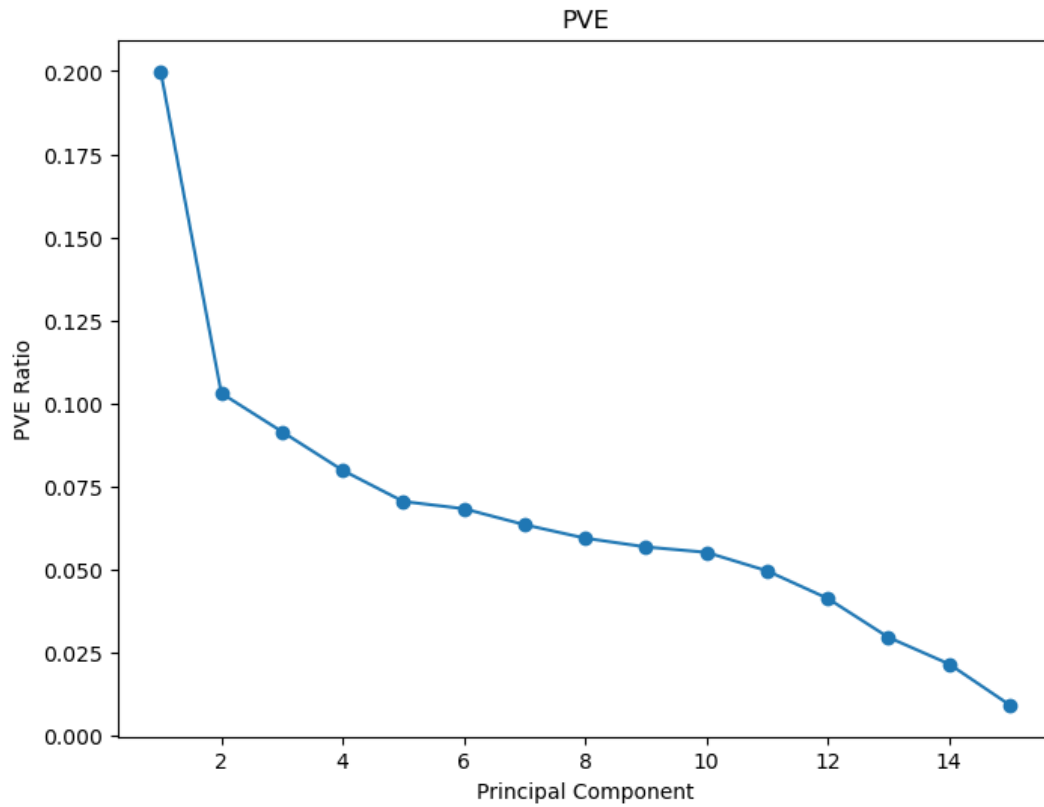


Figure 5: PVE vs PC Plot.

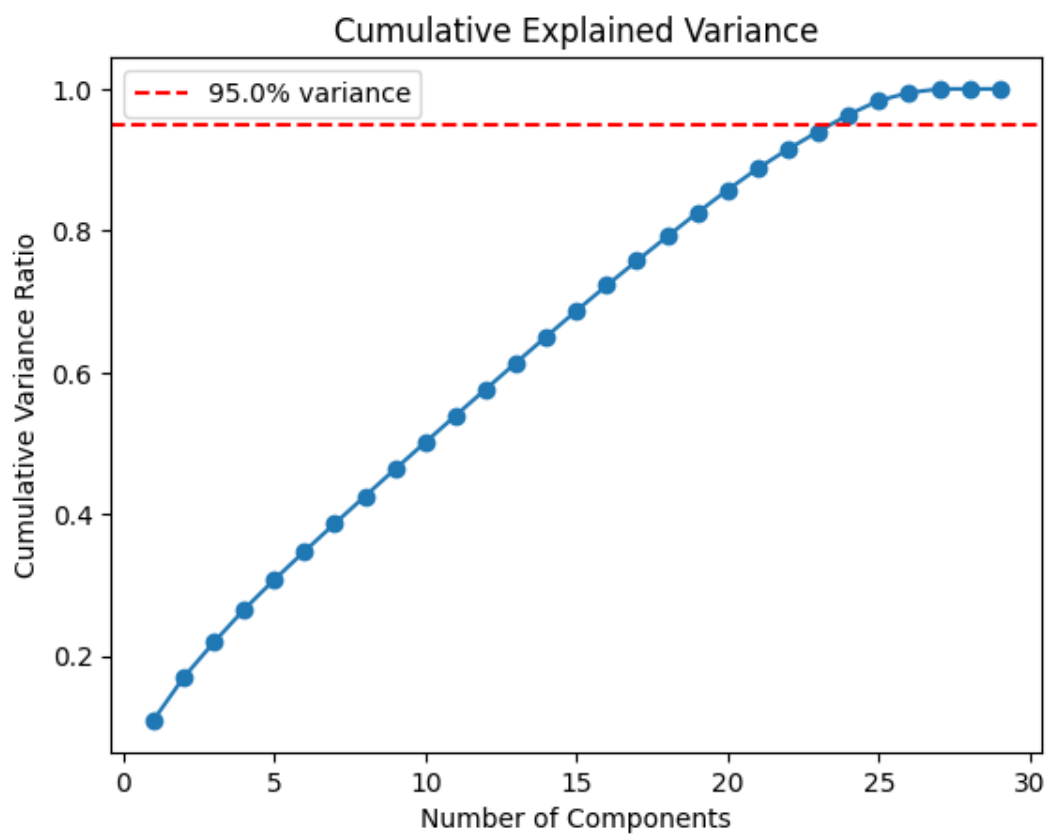


Figure 6: Cumulative PVE

	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	$K = 10$

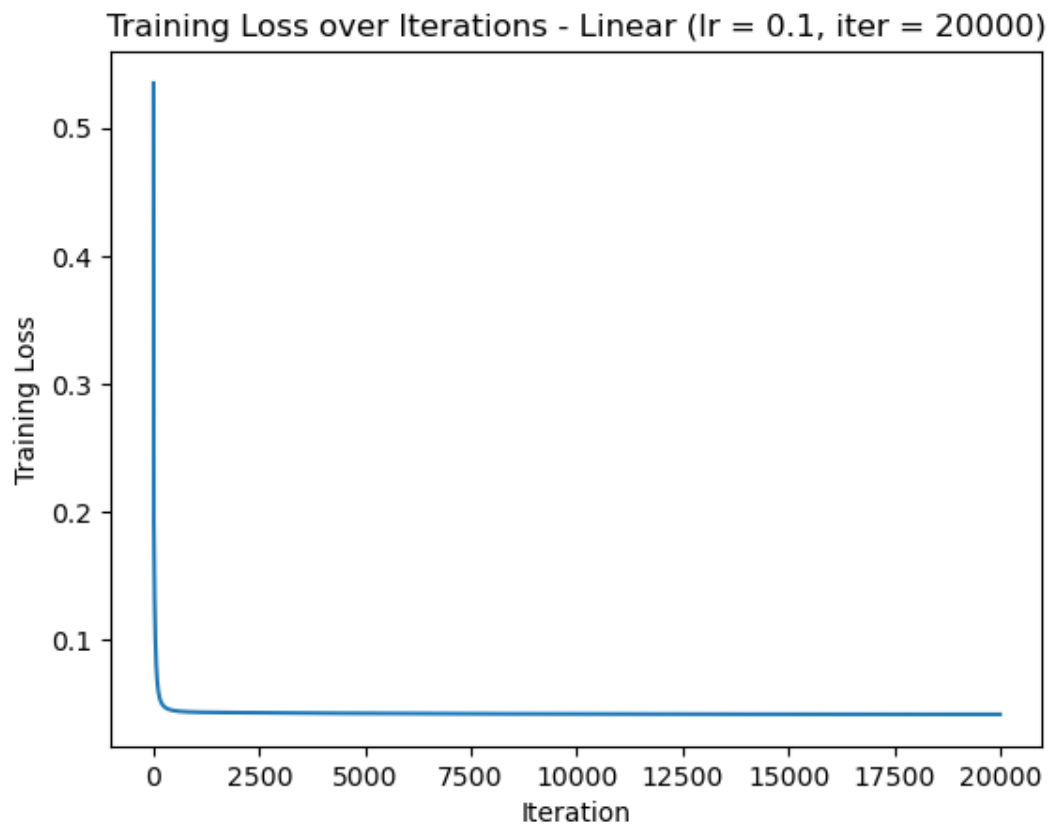


Figure 8: Train Loss vs Iterations (20.000) for Linear Regression with $\alpha = 10^{-1}$.

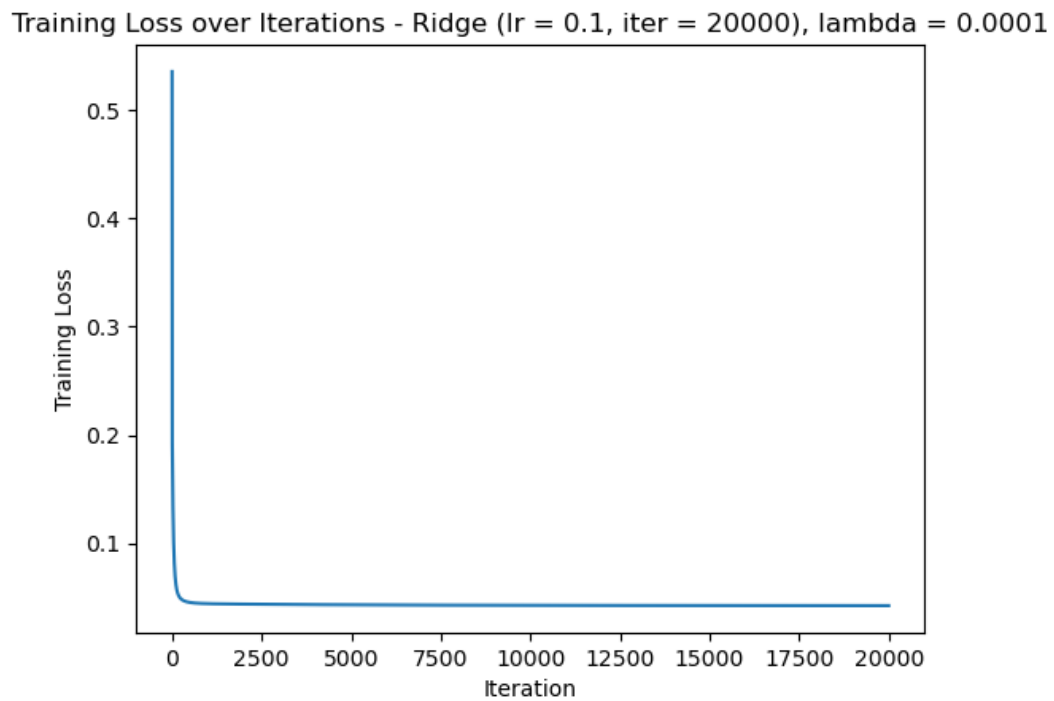


Figure 9: Train Loss vs Iterations (20.000) for Ridge Regression with $\alpha = 10^{-1}$, $\lambda = 10^{-3}$.

Training Loss over Iterations - Lasso (lr = 0.1, iter = 20000, lambda = 0.0001)

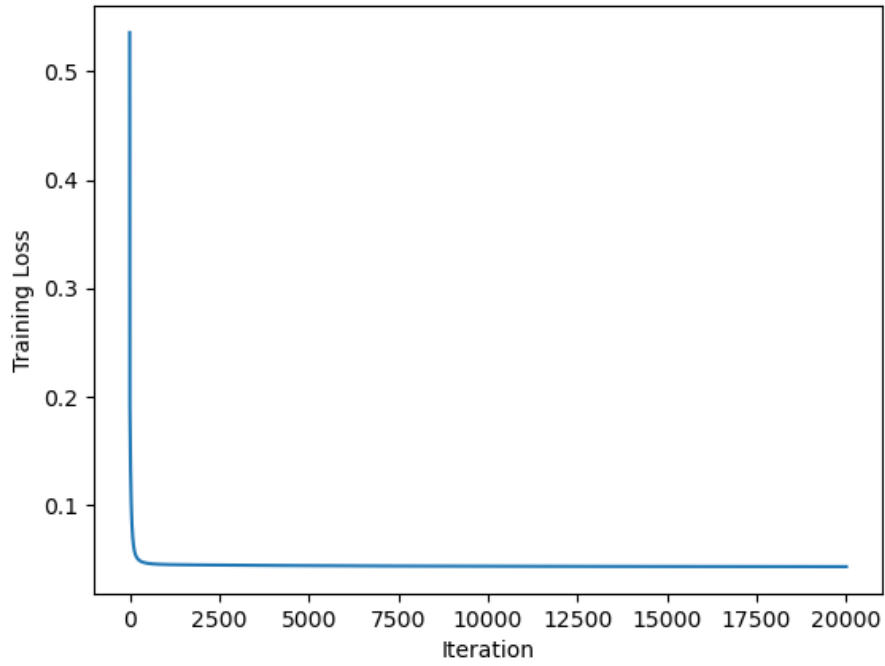


Figure 10: Train Loss vs Iterations (20.000) for Lasso Regression with $\alpha = 10^{-1}$, $\lambda = 10^{-4}$.

Training Loss over Epochs NN (lr = 10000.0, epoch = 5000)

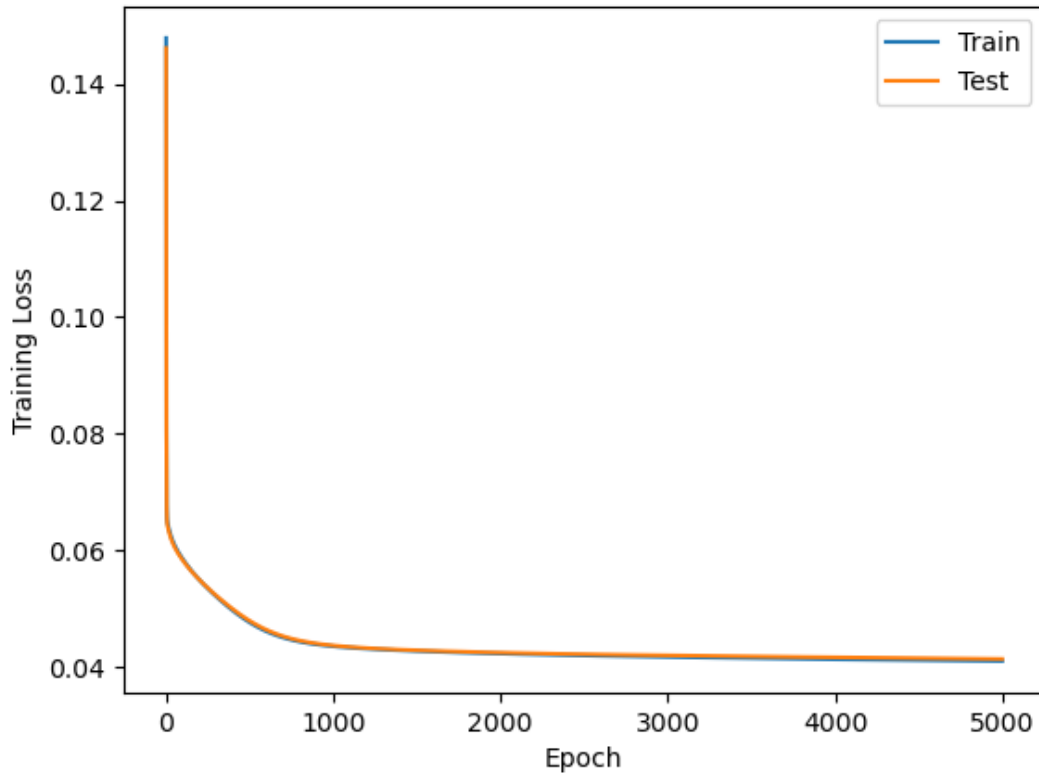


Figure 11: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network, $\alpha = 10^4$

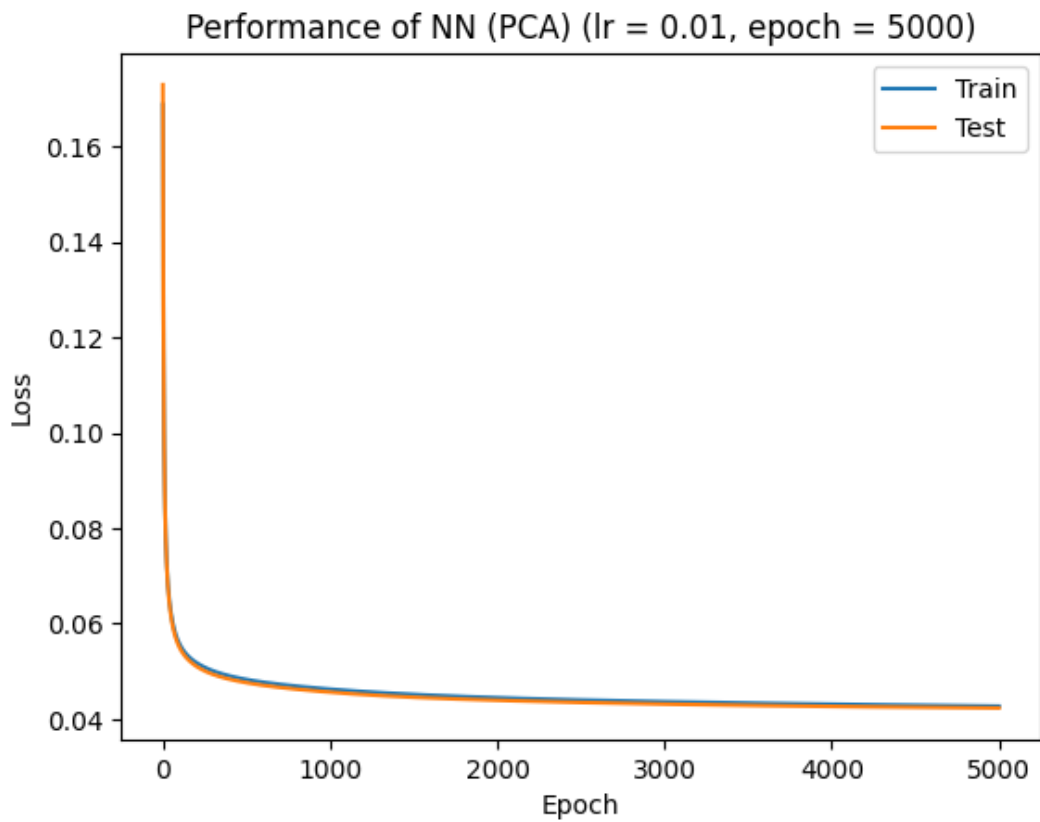


Figure 12: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (PCA), $\alpha = 10^4$

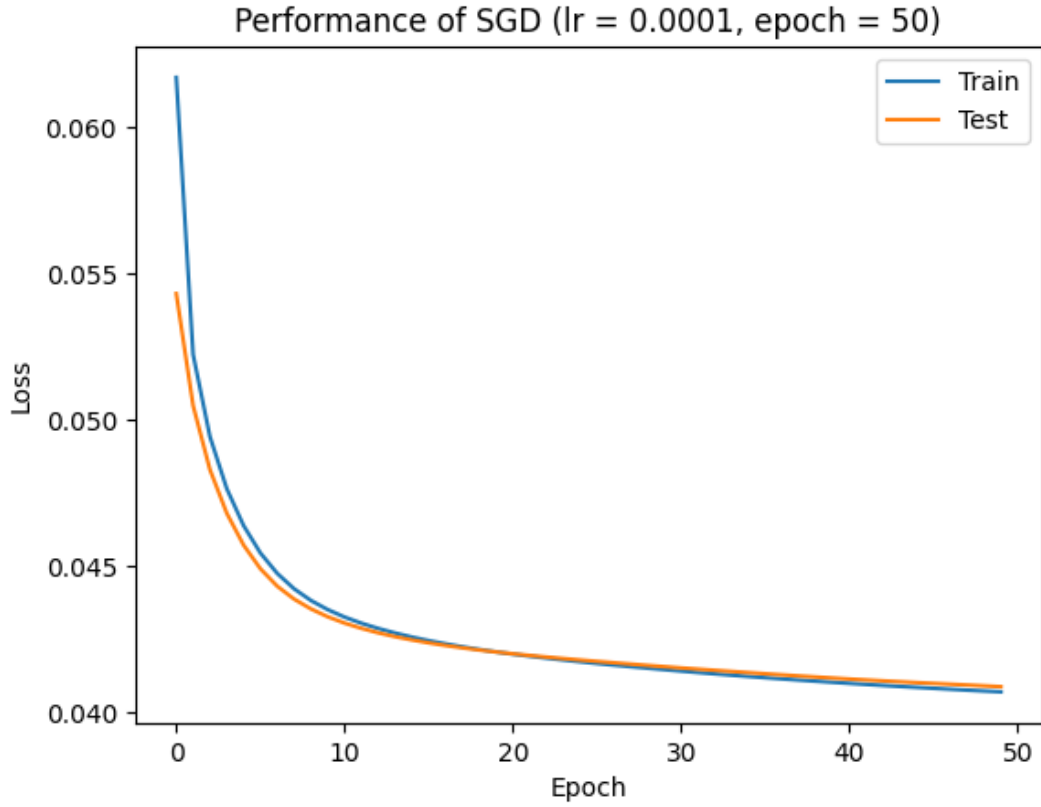


Figure 13: Train and Test Loss vs Epochs (50) for SGD 3 Layer (30, 10, 1) Neural Network, $\alpha = 10^{-4}$

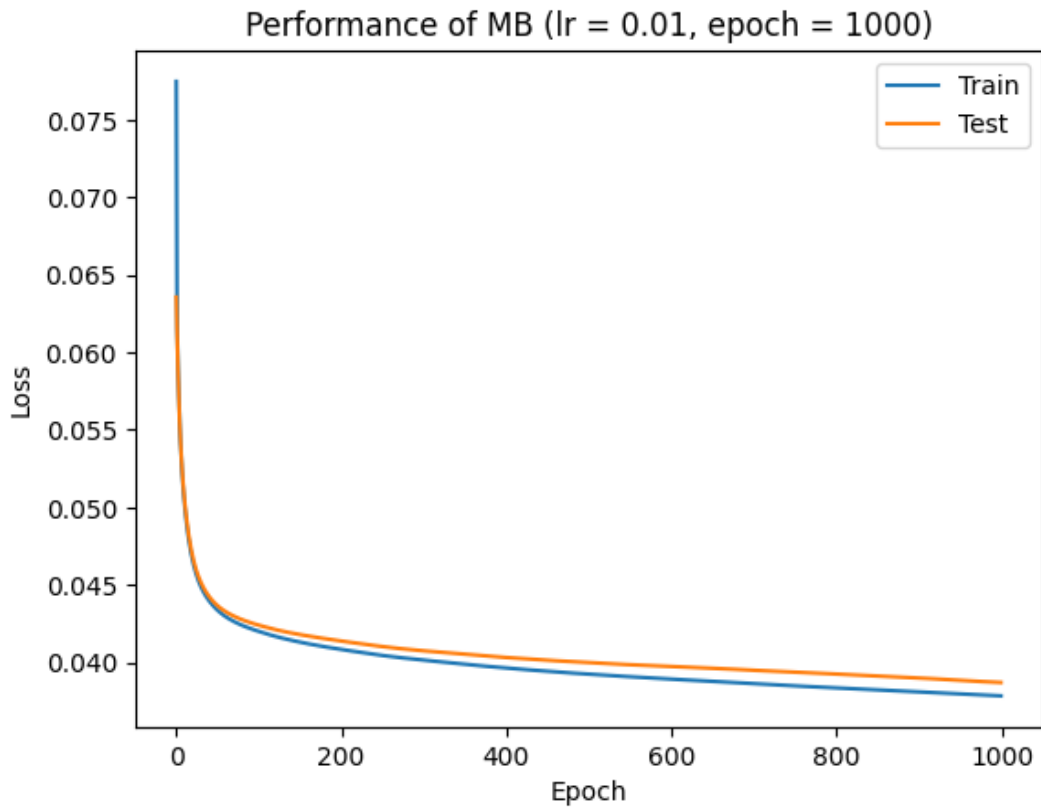


Figure 14: Train and Test Loss vs Epochs (1,000) for Mini-Batch 3 Layer (30, 10, 1) Neural Network, $\alpha = 10^{-2}$

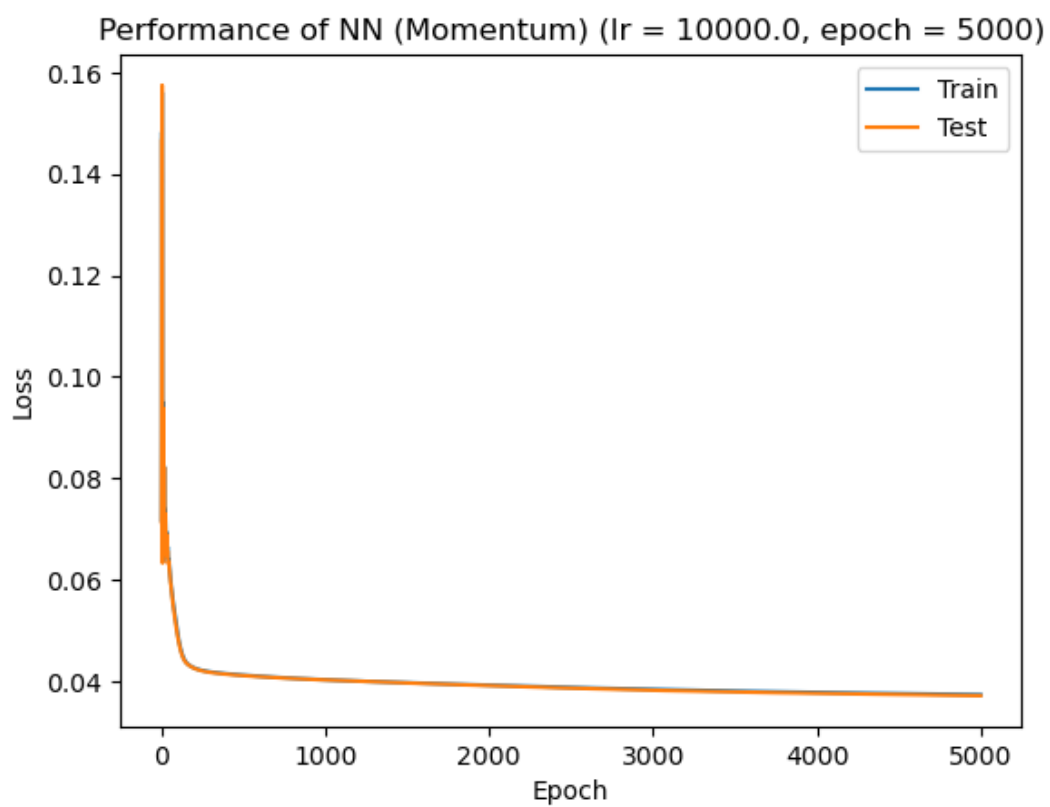


Figure 15: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (Momentum), $\alpha = 10^4$, $momentum = 0.9$

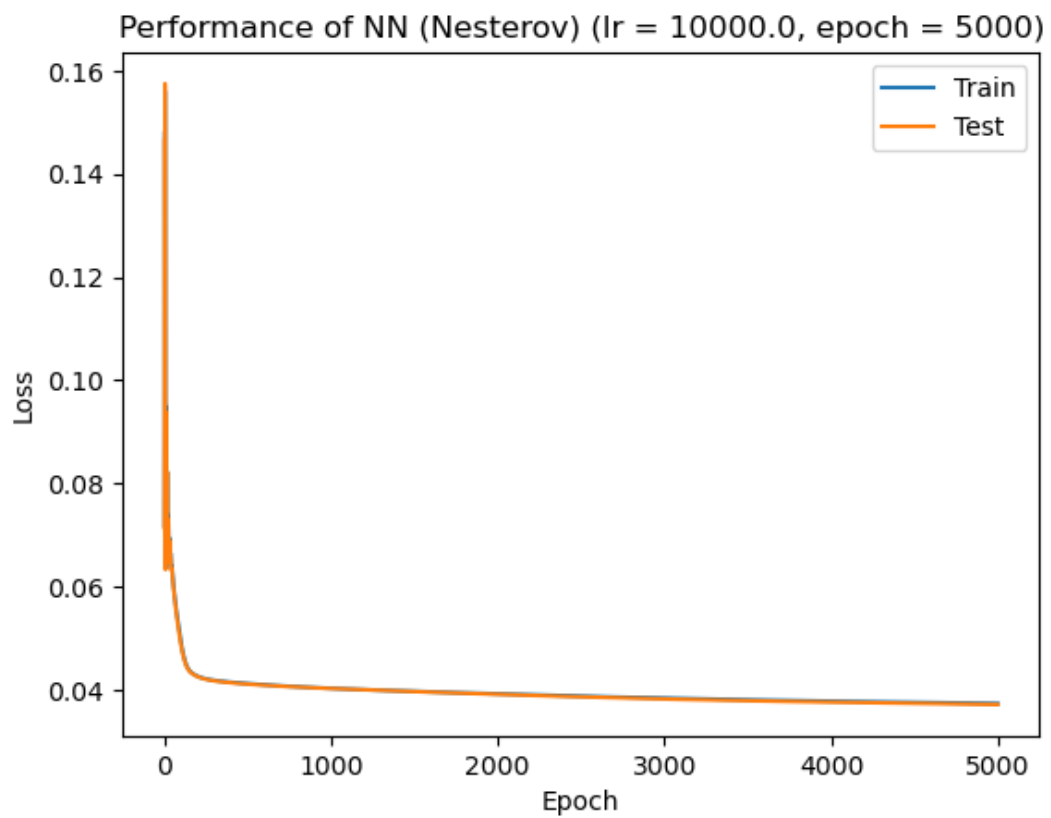


Figure 16: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (Nesterov), $\alpha = 10^4$, $momentum = 0.9$

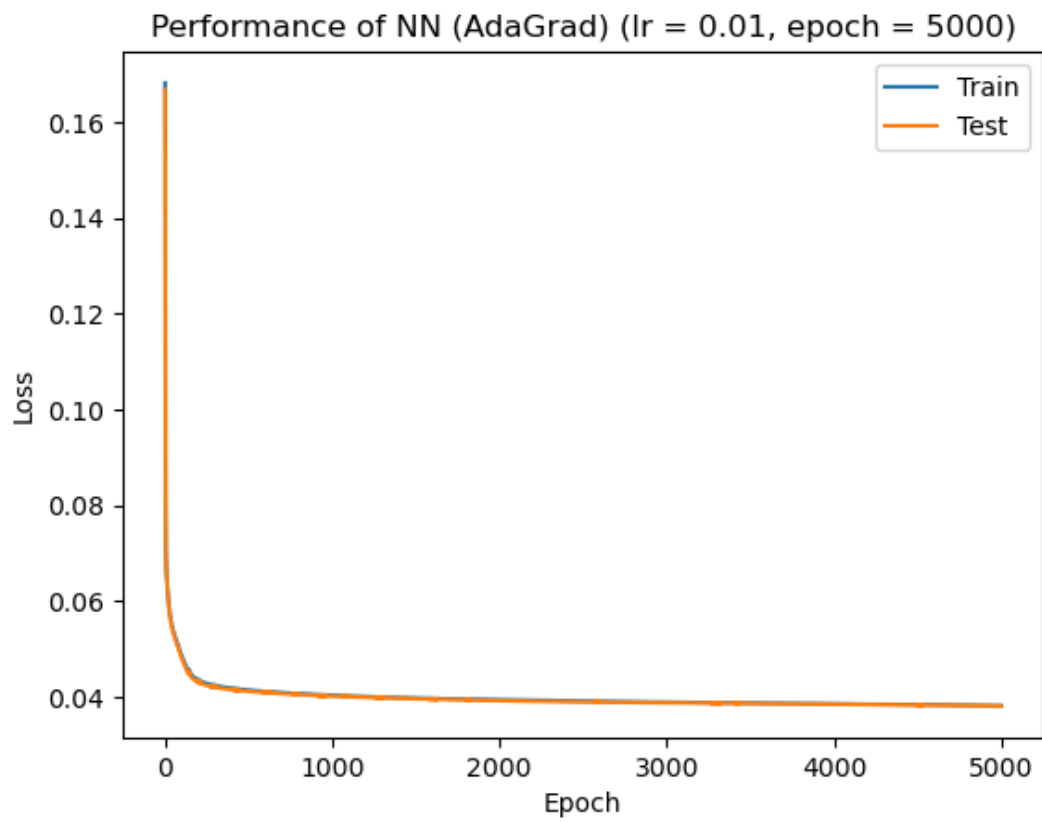


Figure 17: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (AdaGrad), $\alpha = 10^{-2}$, $\epsilon = 10^{-8}$

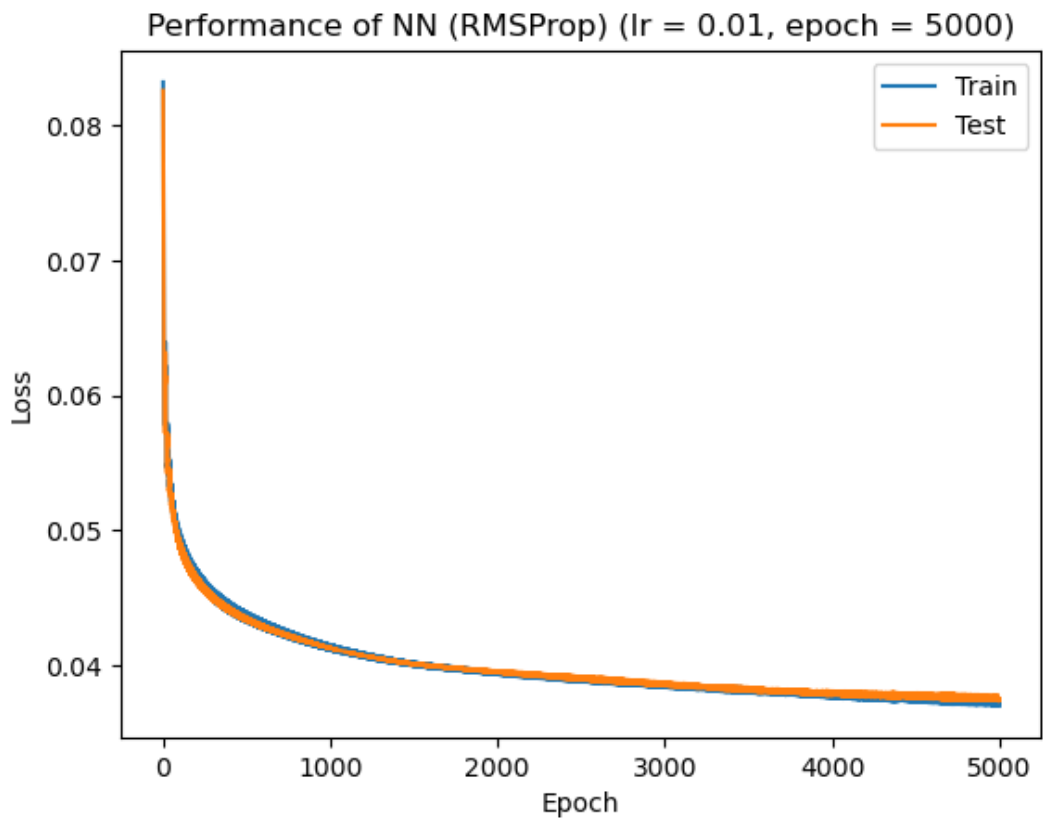


Figure 18: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (RMSProp), $\alpha = 10^{-2}$, $\epsilon = 10^{-8}$, $decayrate = 0.9$

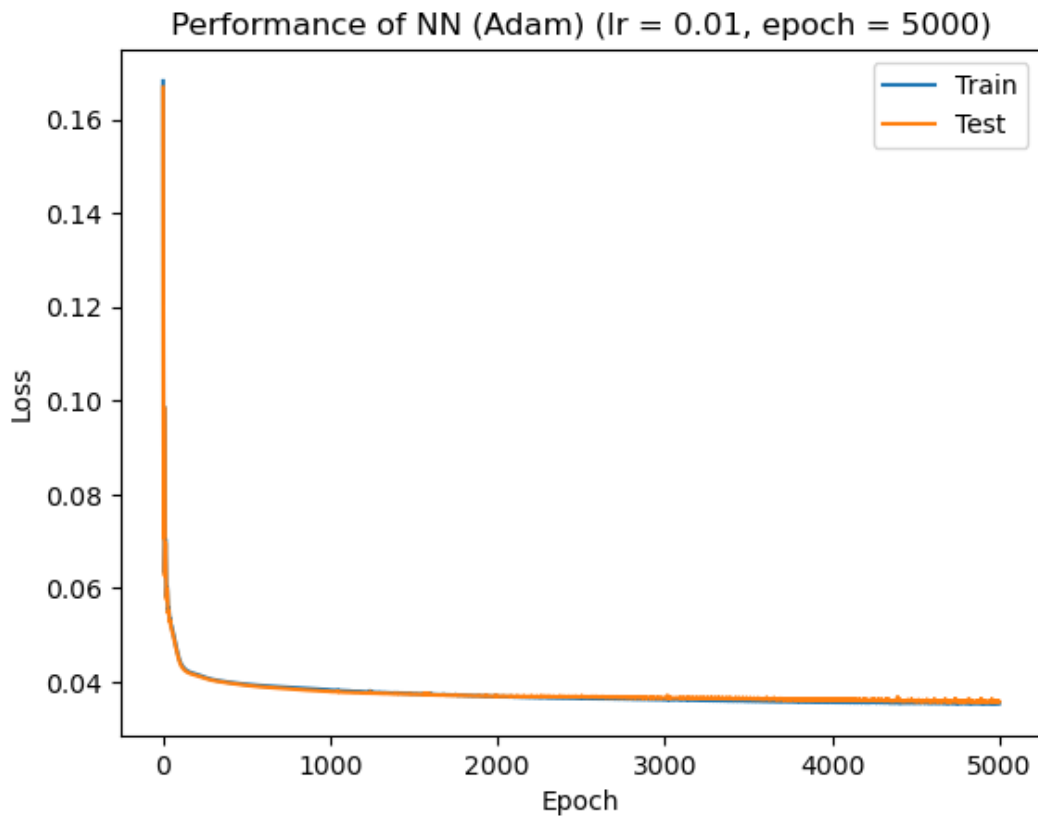


Figure 19: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (Adam), $\alpha = 10^{-2}$, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$

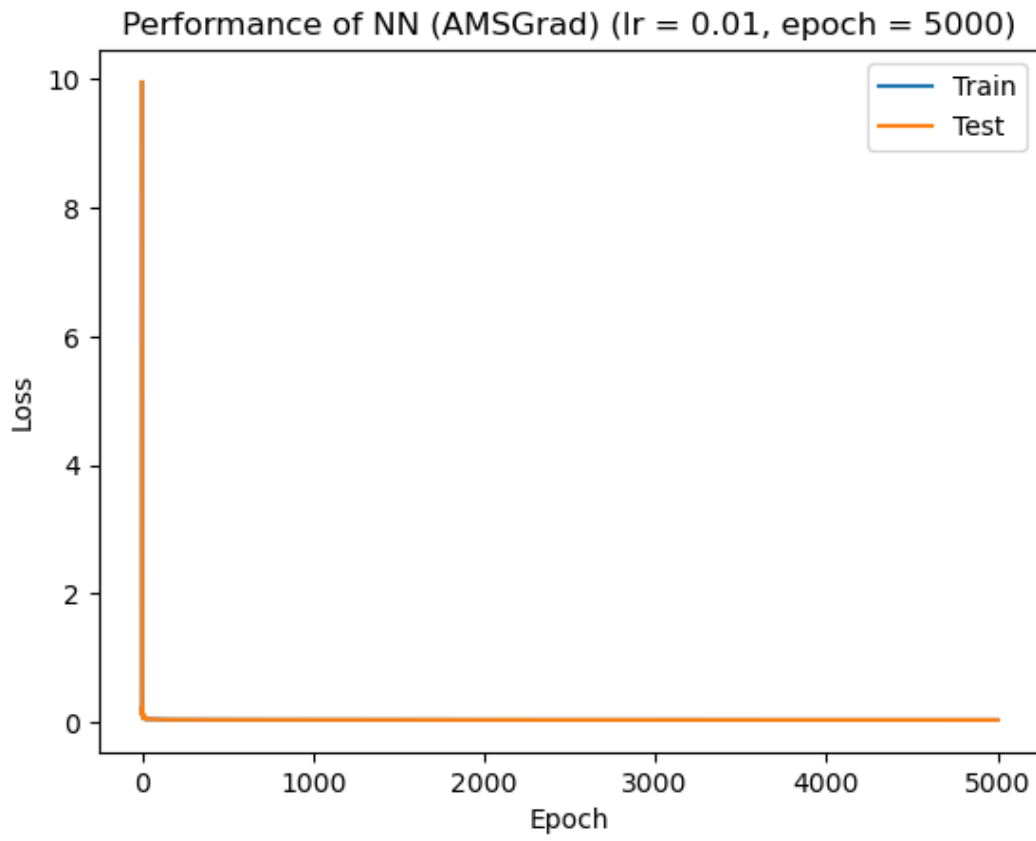


Figure 20: Train and Test Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (AMSGrad), $\alpha = 10^{-2}$, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$

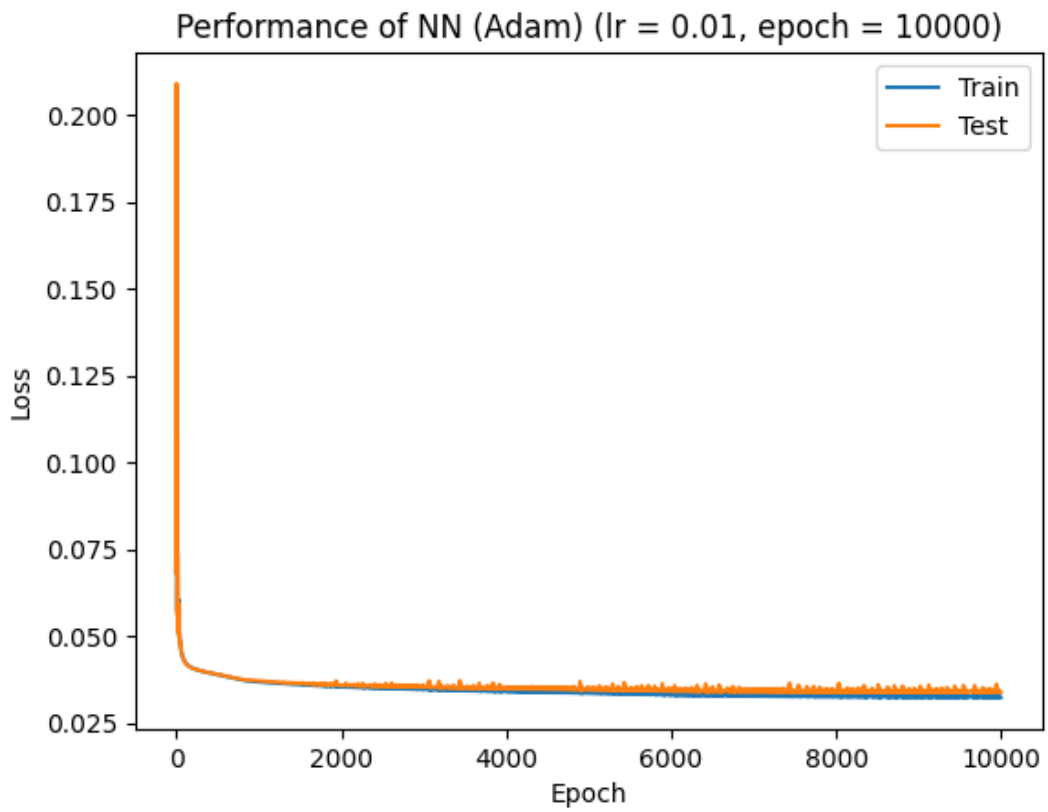


Figure 21: Train and Test Loss vs Epochs (10.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network (Adam), $\alpha = 10^{-2}$, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$

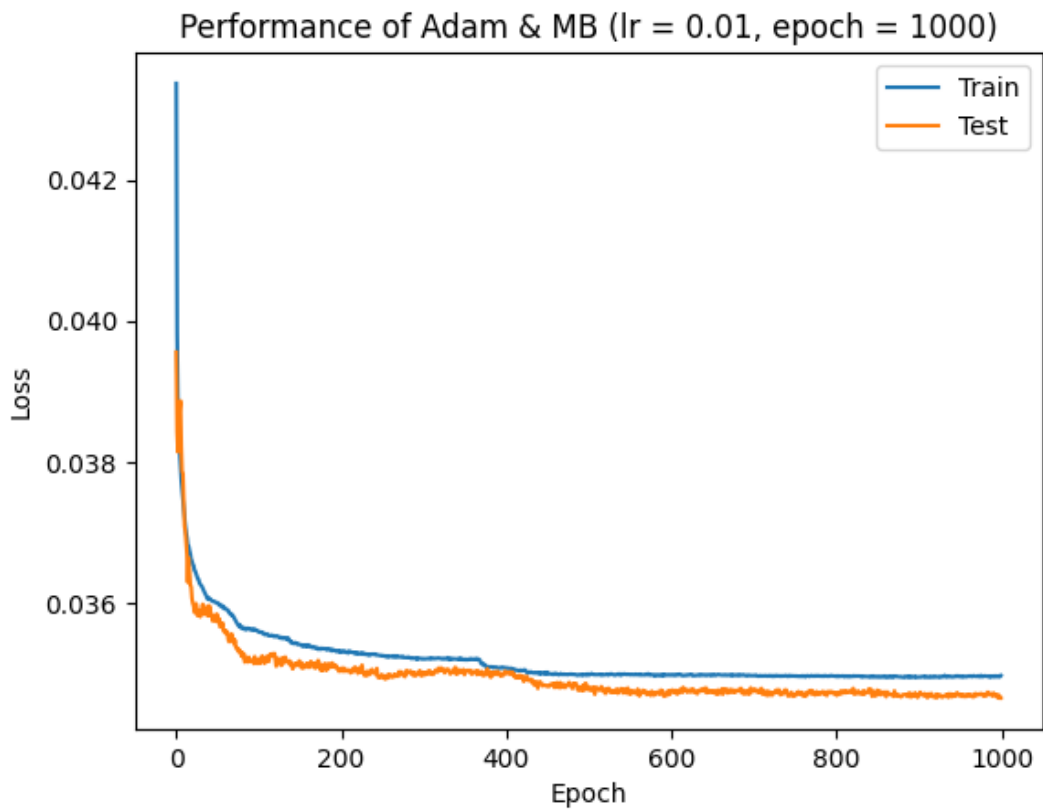


Figure 22: Train and Test Loss vs Epochs (1.000) for Mini-Batch 3 Layer (30, 10, 1) Neural Network (Adam), $\alpha = 10^{-2}$, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$

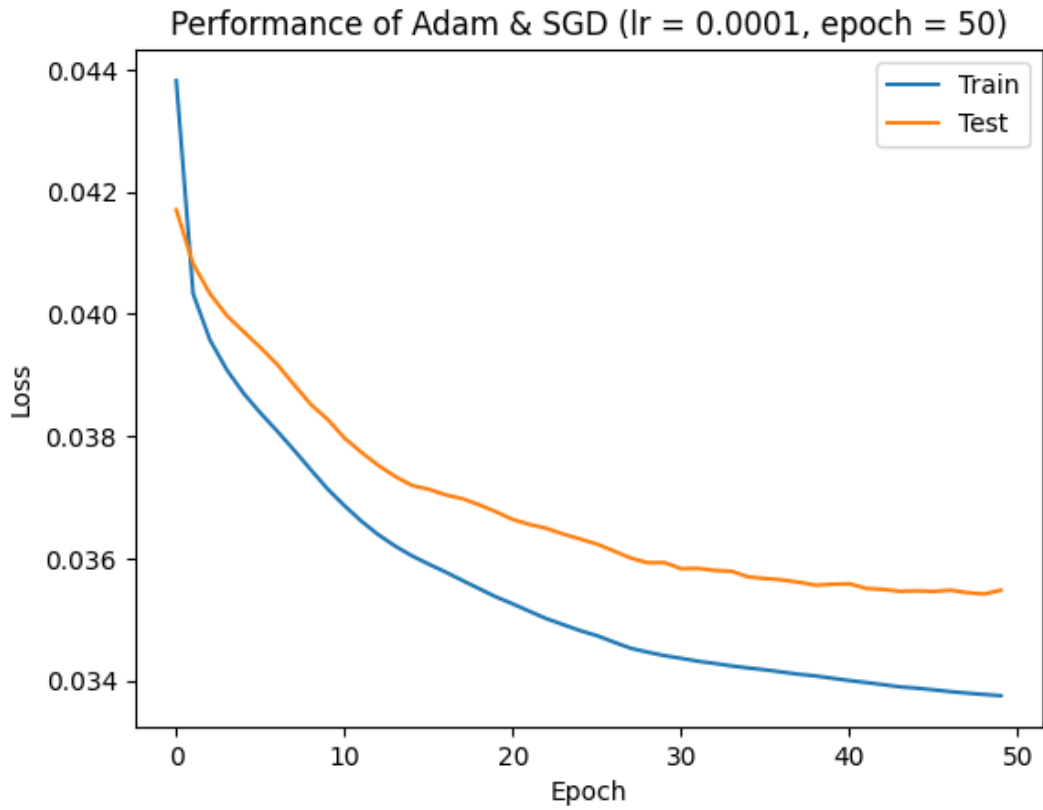


Figure 23: Train and Test Loss vs Epochs (50) for SGD 3 Layer (30, 10, 1) Neural Network (Adam), $\alpha = 10^{-4}$, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$

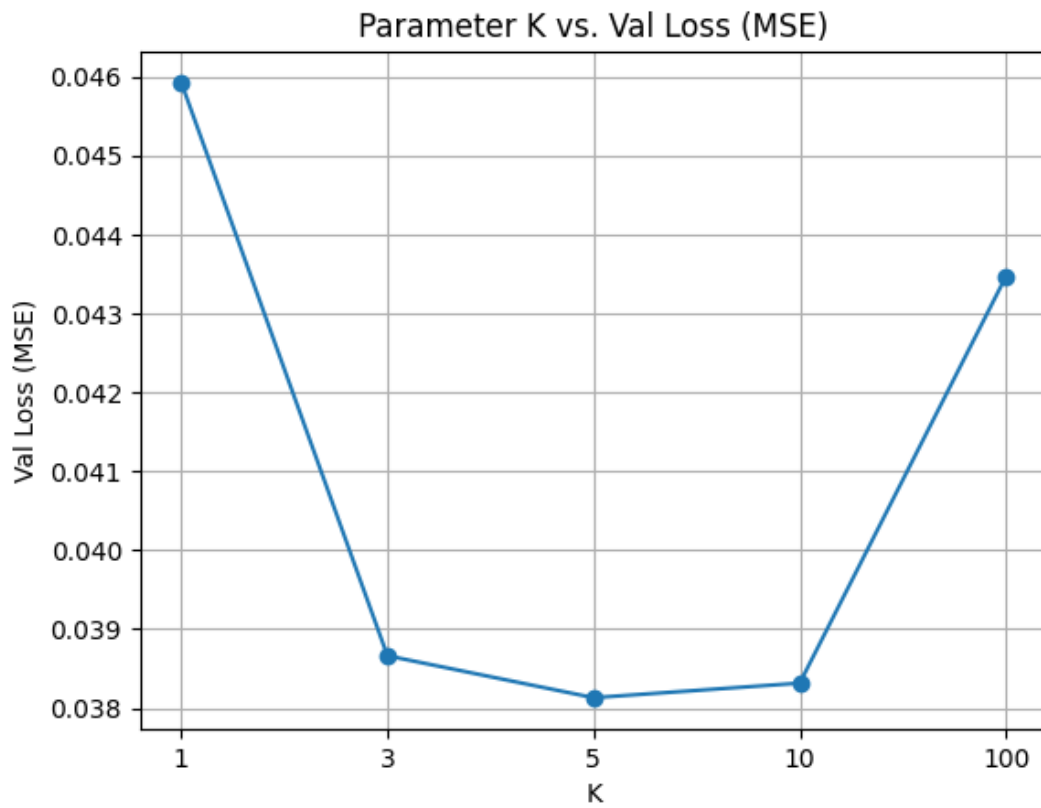


Figure 24: Hyperparameter K vs. Test Loss (MSE) for KNN Regression ($K \in \{1, 3, 5, 10, 100\}$)

8.3 Appendix C - Codes

```
1 import numpy as np
2 import pandas as pd
3 from matplotlib import pyplot as plt
4
5 dataPath = "./data/dataset.csv"
6 df = pd.read_csv(dataPath, index_col=0)
7
8 columns = list(df.columns)
9 columnsToKeep = columns[4: -1]
10
11 df = df[columnsToKeep]
12
13 # Bool to numerical data for explicit row
14 df.loc[:, 'explicit'] = df['explicit'].astype(int)
15
16 # One hot encoding for nominal categories
17 df = pd.get_dummies(df, columns=['key', 'time_signature'], dtype=int)
18
19 def min_max_scaling(df):
20     min_vals = df.min()
21     max_vals = df.max()
22
23     feature_range = max_vals - min_vals
24
25     # Check if any feature has zero range
26     zero_range_features = feature_range[feature_range == 0].index
27
28     # Remove features with zero range from normalization
29     valid_features = feature_range[feature_range != 0].index
30     df_normalized = (df[valid_features] - min_vals[valid_features]) /
31         ↪ feature_range[valid_features]
32
33     # Concatenate back the zero range features
34     if not zero_range_features.empty:
35         df_normalized = pd.concat([df_normalized,
36             ↪ df[zero_range_features]], axis=1)
37
38     return df_normalized
39
40 def standard_scaling(df):
41     mean = df.mean()
42     std = df.std()
43     return (df - mean) / std
44
45 responseFrame = df.pop('valence')
46 predictorFrame = df
47
48 # Min-Max scaling for predictor variables
49 df_normalized = min_max_scaling(predictorFrame)
50
51 # Standard scaling for predictor variables
52 df_standardized = standard_scaling(predictorFrame)
53
54 predictorFrame_scaled = df_normalized
```

Listing 1: Preprocessing Code for all Traditional Methods

```

1  responseData = responseFrame.to_numpy()
2  predictorData = predictorFrame_scaled.to_numpy()
3
4  trainSplit = 0.8
5  valSplit = 0.1
6  testSplit = 0.1
7
8  np.random.seed(42)
9  indices = np.arange(len(predictorData))
10 np.random.shuffle(indices)
11 trainIndices = indices[:int(trainSplit * len(indices))]
12 valIndices = indices[int(trainSplit * len(indices)):int((trainSplit +
    ↳ valSplit) * len(indices))]
13 testIndices = indices[int((trainSplit + valSplit) * len(indices)):]
14
15 trainPredictor, testPredictor, valPredictor =
    ↳ predictorData[trainIndices], predictorData[testIndices],
    ↳ predictorData[valIndices]
16 trainResponse, testResponse, valResponse = responseData[trainIndices],
    ↳ responseData[testIndices], responseData[valIndices]

```

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

```

1 class LinearRegression:
2     def __init__(self, lr = 0.01, n_iters: int = 1000):
3         self.lr = lr
4         self.n_iters = n_iters
5         self.weightVector = None
6         self.loss_history = []
7
8     def fit(self, X, y):
9         num_samples, num_features = X.shape
10        biasColumn = np.ones((num_samples, 1))
11        designMatrix = np.hstack((biasColumn, X))
12        self.weightVector = np.random.rand(num_features + 1)
13
14        # self.normalEquationMethod(designMatrix, y)
15        # print(f'Train Loss for Normal Equation Method:',
    ↳ self.lossMSE(designMatrix, y))
16
17        self.gradientDescent(designMatrix, y)
18        print('Final Train Loss:', self.lossMSE(designMatrix, y))
19
20    def mse_gradient(self, designMatrix, y):
21        predictions = self.predict(designMatrix)
22        return -(2/y.size) * np.dot(designMatrix.T, (y - predictions))
23
24    def gradientDescent(self, designMatrix, y):
25        for i in range(self.n_iters):
26            # Calculate predictions
27            gradientVector = self.mse_gradient(designMatrix, y)
28
29            # Update weights and bias
30            self.weightVector = self.weightVector - self.lr *
    ↳ gradientVector
31
32            # Print gradients for debugging
33            loss = self.lossMSE(designMatrix, y)

```

```

34         print(f'Train Loss at iteration {i}:', loss)
35         self.loss_history.append((i, loss))
36
37     return self
38
39     def normalEquationMethod(self, designMatrix, y):
40         # self.weightVector = np.linalg.inv(np.matmul(designMatrix.T,
41             ↪ designMatrix)).dot(designMatrix.T).dot(y)
42         # return self
43
44         # Compute the SVD of the design matrix
45         U, S, Vt = np.linalg.svd(designMatrix, full_matrices=False)
46
47         # Compute the pseudo-inverse
48         S_inv = np.diag(1 / S)
49         pseudo_inverse = np.dot(np.dot(Vt.T, S_inv), U.T)
50
51         # Calculate the weight vector
52         self.weightVector = np.dot(pseudo_inverse, y)
53
54     return self
55
56     def predict(self, designMatrix):
57         return np.dot(designMatrix, self.weightVector)
58
59     def inference(self, testData):
60         num_samples = testData.shape[0]
61         biasColumn = np.ones((num_samples, 1))
62         designMatrix = np.hstack((biasColumn, testData))
63         return np.dot(designMatrix, self.weightVector)
64
65     def lossMSE(self, designMatrix, y):
66         predictions = self.predict(designMatrix)
67         error = y - predictions
68         squaredError = np.dot(error.T, error)
69         meanSquaredError = 1/(y.size) * squaredError
70         return meanSquaredError
71
72     def plot_loss_history(self):
73         iterations, losses = zip(*self.loss_history)
74         plt.plot(iterations, losses)
75         plt.xlabel('Iteration')
76         plt.ylabel('Training Loss')
77         plt.title(f'Training Loss over Iterations - Linear (lr =
78             ↪ {self.lr}, iter = {self.n_iters})')
79         plt.show()
80
81     def save_model(self, filename):
82         np.savez(filename, weights=self.weightVector)
83
84     def load_model(self, filename):
85         data = np.load(filename)
86         self.weightVector = data['weights']

```

Listing 3: Linear Regression Code

```

1 # Training
2 import time
3
4 regressor = LinearRegression(lr = 1e-1, n_iters= 20000)
5 start = time.time()
6 # regressor.fit(trainPredictor, trainResponse)
7 regressor.load_model('linear_reg_model.npz')
8 end = time.time()
9 print(f'Time elapsed: {end - start:.4f}')
10 # regressor.save_model('linear_reg_model.npz')
11
12 regressor.plot_loss_history()
13
14 predictions = regressor.inference(testPredictor)
15
16 # Validation Performance
17 def mse(testResponse, predictions):
18     error = testResponse - predictions
19     squaredError = np.dot(error.T, error)
20     meanSquaredError = 1/(testResponse.size) * squaredError
21     return meanSquaredError
22
23 def r2_score(testResponse, predictions):
24     mean_observed = np.mean(testResponse)
25     total_sum_squares = np.sum((testResponse - mean_observed) ** 2)
26     residual_sum_squares = np.sum((testResponse - predictions) ** 2)
27     r2 = 1 - (residual_sum_squares / total_sum_squares)
28
29     return r2
30
31 predictions = regressor.inference(valPredictor)
32 MSE = mse(valResponse, predictions)
33 R2 = r2_score(valResponse, predictions)
34 print(f'Validation Performance (MSE): {MSE}')
35 print(f'Validation Performance (R2): {R2}')
36
37 # Test Performance
38 predictions = regressor.inference(testPredictor)
39 MSE = mse(testResponse, predictions)
40 R2 = r2_score(testResponse, predictions)
41 print(f'Test Performance (MSE): {MSE}')
42 print(f'Test Performance (R2): {R2}')
43
44 # Demo prediction
45 demoInstanceLoc = 11401
46 demoPredictor = predictorData[demoInstanceLoc]
47 demoResponse = responseData[demoInstanceLoc]
48 demoPredictor = np.expand_dims(demoPredictor, axis=0)
49 demoPrediction = regressor.inference(demoPredictor)
50 print('Prediction:', demoPrediction, 'Response:', demoResponse)

```

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

```

1 class RidgeRegression:
2     def __init__(self, lr = 0.01, lambdaConstant = 0.1, n_iters: int =
      ↪ 1000):
3         self.lr = lr
4         self.n_iters = n_iters
5         self.lambdaConstant = lambdaConstant
6         self.weightVector = None
7         self.loss_history = []
8
9     def fit(self, X, y):
10        num_samples, num_features = X.shape
11        biasColumn = np.ones((num_samples, 1))
12        designMatrix = np.hstack((biasColumn, X))
13        self.weightVector = np.random.rand(num_features + 1)
14
15        # # Result by matrix formula
16        # self.normalEquationMethod(designMatrix, y)
17        # print(f'Train Loss for Normal Equation Method:',
      ↪ self.lossRidgeMSE(designMatrix, y))
18
19        # Result by gradient descent
20        self.gradientDescent(designMatrix, y)
21        print('Final Train Loss:', self.lossRidgeMSE(designMatrix, y))
22
23    def ridgeMSE_gradient(self, designMatrix, y):
24        return self.mse_gradient(designMatrix, y) + 2 *
      ↪ self.lambdaConstant * self.weightVector
25
26    def mse_gradient(self, designMatrix, y):
27        predictions = self.predict(designMatrix)
28        return -(2/y.size) * np.dot(designMatrix.T, (y - predictions))
29
30    def gradientDescent(self, designMatrix, y):
31        for i in range(self.n_iters):
32            # Calculate predictions
33            gradientVector = self.ridgeMSE_gradient(designMatrix, y)
34            self.weightVector = self.weightVector - self.lr *
      ↪ gradientVector
35            loss = self.lossRidgeMSE(designMatrix, y)
36            print(f'Train Loss at iteration {i}:', loss)
37            self.loss_history.append((i, loss))
38
39        return self
40
41    def normalEquationMethod(self, designMatrix, y):
42        identityMatrix = np.identity(designMatrix.shape[1])
43        # To avoid regularizing bias when standardization not applied
44        identityMatrix[0][0] = 0
45        self.weightVector = np.linalg.inv(designMatrix.T.dot(designMatrix)
      ↪ + self.lambdaConstant *
      ↪ identityMatrix).dot(designMatrix.T).dot(y)
46        return self
47
48    def predict(self, designMatrix):
49        return np.dot(designMatrix, self.weightVector)
50
51    def inference(self, testData):

```

```

52     num_samples = testData.shape[0]
53     biasColumn = np.ones((num_samples, 1))
54     designMatrix = np.hstack((biasColumn, testData))
55     return np.dot(designMatrix, self.weightVector)
56
57 def lossMSE(self, designMatrix, y):
58     predictions = self.predict(designMatrix)
59     error = y - predictions
60     squaredError = np.dot(error.T, error)
61     meanSquaredError = 1/(y.size) * squaredError
62     return meanSquaredError
63
64 def lossRidgeMSE(self, designMatrix, y):
65     mse = self.lossMSE(designMatrix, y)
66     ridge_mse = mse + self.lambdaConstant * np.dot(self.weightVector,
67     ↪ self.weightVector)
68     return ridge_mse
69
70 def plot_loss_history(self):
71     iterations, losses = zip(*self.loss_history)
72     plt.plot(iterations, losses)
73     plt.xlabel('Iteration')
74     plt.ylabel('Training Loss')
75     plt.title(f'Training Loss over Iterations - Ridge (lr = {self.lr},
76     ↪ iter = {self.n_iters}), lambda = {self.lambdaConstant}')
77     plt.show()
78
79 def save_model(self, filename):
80     np.savez(filename, weights=self.weightVector)
81
82 def load_model(self, filename):
83     data = np.load(filename)
84     self.weightVector = data['weights']

```

Listing 5: Ridge Regression Code

```

1 # Training
2 import time
3
4 regressor = RidgeRegression(lr = 1e-1, lambdaConstant=1e-4, n_iters= 20000)
5 start = time.time()
6 # regressor.fit(trainPredictor, trainResponse)
7 regressor.load_model('ridge_reg_model.npz')
8 end = time.time()
9 print(f'Time elapsed: {end - start:.4f}')
10 # regressor.save_model('ridge_reg_model.npz')
11
12 # regressor.plot_loss_history()
13
14 # Validation Performance
15
16 def meanSquaredError(testResponse, predictions):
17     error = testResponse - predictions
18     squaredError = np.dot(error.T, error)
19     meanSquaredError = 1/(testResponse.size) * squaredError
20     return meanSquaredError
21
22 def r2_score(testResponse, predictions):
23     mean_observed = np.mean(testResponse)
24     total_sum_squares = np.sum((testResponse - mean_observed) ** 2)
25     residual_sum_squares = np.sum((testResponse - predictions) ** 2)
26     r2 = 1 - (residual_sum_squares / total_sum_squares)
27
28     return r2
29
30 predictions = regressor.inference(valPredictor)
31 MSE = meanSquaredError(valResponse, predictions)
32 R2 = r2_score(valResponse, predictions)
33 print(f'Validation Performance (MSE): {MSE}')
34 print(f'Validation Performance (R2): {R2}')
35
36 # Measuring Performance
37
38 predictions = regressor.inference(testPredictor)
39 MSE = meanSquaredError(testResponse, predictions)
40 R2 = r2_score(testResponse, predictions)
41 print(f'Test Performance (MSE): {MSE}')
42 print(f'Test Performance (R2): {R2}')
43
44 demoInstanceLoc = 11401
45 demoPredictor = predictorData[demoInstanceLoc]
46 demoResponse = responseData[demoInstanceLoc]
47 demoPredictor = np.expand_dims(demoPredictor, axis=0)
48 demoPrediction = regressor.inference(demoPredictor)
49 print('Prediction:', demoPrediction, 'Response:', demoResponse)

```

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


```

1 class LassoRegression:
2     def __init__(self, lr=0.01, lambdaConstant=0.1, n_iters=1000):
3         self.lr = lr
4         self.n_iters = n_iters
5         self.lambdaConstant = lambdaConstant
6         self.weightVector = None
7         self.loss_history = []
8
9     def fit(self, X, y):
10        num_samples, num_features = X.shape
11        biasColumn = np.ones((num_samples, 1))
12        designMatrix = np.hstack((biasColumn, X))
13        self.weightVector = np.random.rand(num_features + 1)
14
15        self.subgradientDescent(designMatrix, y)
16        print('Final Train Loss:', self.lossLassoMSE(designMatrix, y))
17
18    def mse_gradient(self, designMatrix, y):
19        predictions = self.predict(designMatrix)
20        return -(2/y.size) * np.dot(designMatrix.T, (y - predictions))
21
22    def subgradientDescent(self, designMatrix, y):
23        for i in range(self.n_iters):
24            gradientVector = self.lassoMSE_subgradient(designMatrix, y)
25
26            # Update weights using subgradient descent
27            self.weightVector = self.weightVector - self.lr *
28                ↪ gradientVector
29            loss = self.lossLassoMSE(designMatrix, y)
30            print(f'Train Loss at iteration {i}:', loss)
31            self.loss_history.append((i, loss))
32
33        return self
34
35    def lassoMSE_subgradient(self, designMatrix, y):
36        mse_gradient = self.mse_gradient(designMatrix, y)
37        lasso_gradient = np.sign(self.weightVector)
38        return mse_gradient + self.lambdaConstant * lasso_gradient
39
40    def predict(self, designMatrix):
41        return np.dot(designMatrix, self.weightVector)
42
43    def inference(self, testData):
44        num_samples = testData.shape[0]
45        biasColumn = np.ones((num_samples, 1))
46        designMatrix = np.hstack((biasColumn, testData))
47        return np.dot(designMatrix, self.weightVector)
48
49    def lossMSE(self, designMatrix, y):
50        predictions = self.predict(designMatrix)
51        error = y - predictions
52        squaredError = np.dot(error.T, error)
53        meanSquaredError = 1/(y.size) * squaredError
54        return meanSquaredError
55
56    def lossLassoMSE(self, designMatrix, y):
57        mse = self.lossMSE(designMatrix, y)

```

```

57         lasso_mse = mse + self.lambdaConstant *
           ↪ np.sum(np.abs(self.weightVector))
58     return lasso_mse
59
60     def plot_loss_history(self):
61         iterations, losses = zip(*self.loss_history)
62         plt.plot(iterations, losses)
63         plt.xlabel('Iteration')
64         plt.ylabel('Training Loss')
65         plt.title(f'Training Loss over Iterations - Lasso (lr = {self.lr},
           ↪ iter = {self.n_iters}, lambda = {self.lambdaConstant})')
66         plt.show()
67
68     def save_model(self, filename):
69         np.savez(filename, weights=self.weightVector)
70
71     def load_model(self, filename):
72         data = np.load(filename)
73         self.weightVector = data['weights']

```

Listing 7: Lasso Regression code

```

1 # Train
2 import time
3
4 regressor = LassoRegression(lr = 1e-1, lambdaConstant=1e-4, n_iters= 20000)
5
6 start = time.time()
7 # regressor.fit(trainPredictor, trainResponse)
8 regressor.load_model('lasso_reg_model.npz')
9 end = time.time()
10
11 # Save model
12 # regressor.save_model('lasso_reg_model.npz')
13
14 print(f'Time elapsed: {end - start:.4}')
15
16 # regressor.plot_loss_history()
17
18 # Validation Performance
19 def meanSquaredError(testResponse, predictions):
20     error = testResponse - predictions
21     squaredError = np.dot(error.T, error)
22     meanSquaredError = 1/(testResponse.size) * squaredError
23     return meanSquaredError
24
25 def r2_score(testResponse, predictions):
26     mean_observed = np.mean(testResponse)
27     total_sum_squares = np.sum((testResponse - mean_observed) ** 2)
28     residual_sum_squares = np.sum((testResponse - predictions) ** 2)
29     r2 = 1 - (residual_sum_squares / total_sum_squares)
30
31     return r2
32
33 predictions = regressor.inference(valPredictor)
34 MSE = meanSquaredError(valResponse, predictions)
35 R2 = r2_score(valResponse, predictions)
36 print(f'Validation Performance (MSE): {MSE}')
37 print(f'Validation Performance (R2): {R2}')
38
39 # Measuring Performance
40 predictions = regressor.inference(testPredictor)
41 MSE = meanSquaredError(testResponse, predictions)
42 R2 = r2_score(testResponse, predictions)
43 print(f'Test Performance (MSE): {MSE}')
44 print(f'Test Performance (R2): {R2}')
45
46 demoInstanceLoc = 11021
47 demoPredictor = predictorData[demoInstanceLoc]
48 demoResponse = responseData[demoInstanceLoc]
49 demoPredictor = np.expand_dims(demoPredictor, axis=0)
50 print(demoPredictor.T.shape)
51 demoPrediction = regressor.inference(demoPredictor)
52 print('Prediction:', demoPrediction, 'Response:', demoResponse)

```

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

```

1 def relu(z):
2     return np.maximum(0, z)
3
4 def tanh(z):
5     return np.tanh(z)
6
7 def linear(z):
8     return z
9
10 def reluDer(z):
11     return np.where(z > 0, 1, 0)
12
13 def tanhDer(z):
14     return 1 - z**2
15
16 def linearDer(z):
17     return 1
18
19 activationDict = {'relu': relu, 'tanh': tanh, 'linear': linear}
20 activationDerivativeDict = {'relu': reluDer, 'tanh': tanhDer, 'linear':
    ↪ linearDer}

```

Listing 9: Neural Network Activation Functions code

```

1 class Layer:
2     def __init__(self, inputNumNeuron, numNeurons, activationName,
    ↪ batchSize):
3         self.batchSize = batchSize
4         self.inputNumNeuron = inputNumNeuron
5         self.numNeurons = numNeurons
6         self.activationName = activationName
7         self.activation = activationDict[self.activationName]
8         self.activationDerivative =
    ↪ activationDerivativeDict[self.activationName]
9         self.dZ_state = np.empty((numNeurons, batchSize))
10        self.Z_state = np.empty((numNeurons, batchSize))
11        self.A_state = np.empty((numNeurons, batchSize))
12        self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
13        self.db_state = np.zeros((self.numNeurons, 1))
14        self.initWeights()
15
16    def initWeights(self):
17        # Random initialization unfortunately failed.
18        # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
19        # self.b = np.random.randn(self.numNeurons, 1)
20
21        # Xavier initialization for weights
22        self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
    ↪ np.sqrt(1 / self.inputNumNeuron)
23        # Initializing biases with zeros
24        self.b = np.zeros((self.numNeurons, 1))
25
26    def updateForwardState(self, inputToLayer):
27        # print('\nupdateForwardState():\n', 'inputToLayer:',
    ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
    ↪ self.b.shape)
28        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
29        output = self.activation(inducedLocal)

```

```

30     self.Z_state = inducedLocal
31     self.A_state = output
32     return output
33
34 def predict(self, inputToLayer, printVals=False):
35     inducedLocal = np.matmul(self.W, inputToLayer) + self.b
36     output = self.activation(inducedLocal)
37     if printVals:
38         print('inp:', self.b, self.W, 'out:', output)
39     return output
40
41 def updateDeltaState(self, dA):
42     # Derivative of loss over the weights of this layer
43     # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
44     #       ↪ 'self.Z_state', self.Z_state.shape)
45     derActivation = self.activationDerivative(self.Z_state)
46     dZ = np.multiply(dA, derActivation)
47     self.dZ_state = dZ
48
49 def calculateChange(self, A_input):
50     self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
51     #       ↪ A_input.T)
52     self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
53     #       ↪ axis=1, keepdims=True)
54     # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
55     #       ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
56     #       ↪ 'self.Z_state', self.Z_state)
57     # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
58     #       ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
59     #       ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)
60
61 def updateWeightsAndBias(self, lr):
62     self.W = self.W - lr * self.dW_state
63     self.b = self.b - lr * self.db_state

```

Listing 10: Neural Network Layer Class code

```

1 class NeuralNetwork:
2     def __init__(self):
3         self.layers = []
4         self.loss_history = []
5         self.test_loss_history = []
6
7     def addLayer(self, layer):
8         self.layers.append(layer)
9
10    def loss(self, predictions, y):
11        # MSE
12        batchSize = y.size
13        error = y - predictions
14        squaredError = np.dot(error.T, error)
15        mse = (1 / batchSize) * squaredError
16        return mse
17
18    def lossDer(self, predictions, y):
19        # MSE Derivative
20        batchSize = y.size
21        error = y - predictions
22        mseDer = (-2 / batchSize) * np.sum(error, axis=0, keepdims=True)
23        return mseDer
24
25    def predict(self, testPredictor):
26        output = testPredictor
27        for layer in self.layers:
28            printVals = True if False else False
29            output = layer.predict(output, printVals)
30        return output
31
32    def forward(self, trainPredictor):
33        output = trainPredictor
34        for layer in self.layers:
35            output = layer.updateForwardState(output)
36        return output
37
38    def backprop(self, predictions, y, x, lr):
39        # Update Delta State
40        for layerNumber in reversed(range(len(self.layers))):
41            layer = self.layers[layerNumber]
42            inputToLayer = self.layers[layerNumber - 1].A_state if
43                ↳ layerNumber > 0 else x
44
45            # Output Layer
46            if(layer == self.layers[-1]):
47                y_resaped = np.reshape(y, (1, y.size))
48                lossDerivative = self.lossDer(predictions, y_resaped)
49                # print('\nbackpropFirst():\n', 'predictions:',
50                    ↳ predictions.shape, 'y_resaped:', y_resaped.shape,
51                    ↳ 'lossDerivative:', lossDerivative.shape,
52                    ↳ 'inputToLayer', inputToLayer.shape)
53                layer.updateDeltaState(lossDerivative)
54                layer.calculateChange(inputToLayer)
55
56            # Hidden Layers
57            else:
58                dZ_next = nextLayer.dZ_state

```

```

54         W_next = nextLayer.W
55         dA = np.dot(W_next.T, dZ_next)
56         # print('\nbackpropAlt():\n', 'dZ_next:', dZ_next.shape,
57             ↳ 'W_next', W_next.shape, 'dA:', dA.shape)
58         layer.updateDeltaState(dA)
59         layer.calculateChange(inputToLayer)
60
61     nextLayer = layer
62
63     # Update Weights and Bias
64     for layerNumber in range(len(self.layers)):
65         layer = self.layers[layerNumber]
66         layer.updateWeightsAndBias(lr)
67
68     def fit(self, mini_batches_x, mini_batches_y, mini_test_x,
69         ↳ mini_test_y, lr=1e-2, epochAmount=10):
70         for epoch in range(epochAmount):
71             print('-----EPOCH-----> ', epoch + 1)
72             # Train using mini-batches
73             for mini_batch_X, mini_batch_Y in zip(mini_batches_x,
74                 ↳ mini_batches_y):
75                 predictions = self.forward(mini_batch_X)
76                 self.backprop(predictions, mini_batch_Y, mini_batch_X, lr)
77
78                 predictions = self.predict(mini_batch_X)
79                 trainLoss = np.squeeze(self.loss(predictions.T,
80                     ↳ mini_batch_Y.T))
81                 print(f'Train Loss:', trainLoss)
82                 self.loss_history.append((epoch, trainLoss))
83
84                 testError = np.squeeze(self.testLoss(mini_test_x.T,
85                     ↳ mini_test_y))
86                 self.test_loss_history.append((epoch, testError))
87
88             print('Final Train Loss:', trainLoss)
89
90     def testLoss(self, test_x, test_y):
91         predictions = np.squeeze(self.predict(test_x))
92         lossMSE = self.loss(predictions, test_y)
93         return lossMSE
94
95     def testLossR2(self, test_x, test_y):
96         predictions = np.squeeze(self.predict(test_x))
97         mean_observed = np.mean(test_y)
98         total_sum_squares = np.sum((test_y - mean_observed) ** 2)
99         residual_sum_squares = np.sum((test_y - predictions) ** 2)
100         r2 = 1 - (residual_sum_squares / total_sum_squares)
101         return r2
102
103     def plot_loss_history(self):
104         iterations, losses = zip(*self.loss_history)
105         _, testLosses = zip(*self.test_loss_history)
106         plt.plot(iterations, losses)
107         plt.plot(iterations, testLosses)
108         plt.xlabel('Epoch')
109         plt.ylabel('Training Loss')
110         plt.title(f'Training Loss over Epochs NN (lr = {1e4}, epoch =
111             ↳ {5000})')

```

```

106     plt.legend(['Train', 'Test'])
107     plt.show()
108
109     def save_weights(self, filename):
110         # Create a dictionary to hold weights and biases of all layers
111         weights_dict = {}
112         for i, layer in enumerate(self.layers):
113             weights_dict[f"Layer_{i}_W"] = layer.W
114             weights_dict[f"Layer_{i}_b"] = layer.b
115
116         # Save the weights dictionary to a file
117         np.savez(filename, **weights_dict)
118
119     def load_weights(self, filename):
120         # Load the weights dictionary from the file
121         data = np.load(filename)
122
123         # Iterate through layers and load weights and biases
124         for i, layer in enumerate(self.layers):
125             layer.W = data[f"Layer_{i}_W"]
126             layer.b = data[f"Layer_{i}_b"]

```

Listing 11: Neural Network code

```

1 responseData = responseFrame.to_numpy()
2 predictorData = predictorFrame_scaled.to_numpy()
3
4 # Full batch gradient descent
5 batchSize = predictorData.shape[0]
6
7 # Mini batch gradient descent
8 # batchSize = predictorData.shape[0] // 100
9
10 # Stochastic gradient descent
11 # batchSize = 1
12
13 trainSplit = 0.8
14 valSplit = 0.1
15 testSplit = 0.1
16
17 # np.random.seed(42)
18 indices = np.arange(len(predictorData))
19 np.random.shuffle(indices)
20 trainIndices = indices[:int(trainSplit * len(indices))]
21 valIndices = indices[int(trainSplit * len(indices)):int((trainSplit +
    ↳ valSplit) * len(indices))]
22 testIndices = indices[int((trainSplit + valSplit) * len(indices)):]
23
24 trainPredictor, testPredictor, valPredictor = predictorData[trainIndices],
    ↳ predictorData[testIndices], predictorData[valIndices]
25 trainResponse, testResponse, valResponse = responseData[trainIndices],
    ↳ responseData[testIndices], responseData[valIndices]
26
27 trainResponse = np.expand_dims(trainResponse, axis=1)
28
29 # Function to create mini-batches
30 def create_mini_batches(data, batch_size):
31     mini_batches = []

```



```

32 data_size = len(data)
33 num_batches = data_size // batch_size
34
35 for i in range(num_batches):
36     start_idx = i * batch_size
37     end_idx = start_idx + batch_size
38     mini_batch = data[start_idx:end_idx]
39     mini_batches.append(mini_batch.T)
40
41 if data_size % batch_size != 0:
42     mini_batch = data[num_batches * batch_size:]
43     mini_batches.append(mini_batch.T)
44
45 return np.array(mini_batches)
46
47 # Create mini-batches
48 mini_batches_X = create_mini_batches(trainPredictor, batch_size= batchSize)
49 mini_batches_Y = create_mini_batches(trainResponse, batch_size= batchSize)
50 miniTestX = testPredictor
51 miniTestY = testResponse
52 miniValX = valPredictor
53 miniValY = valResponse

```

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

```

1 import time
2
3 # Initialize NeuralNetwork
4 nn = NeuralNetwork()
5 nn.addLayer(Layer(mini_batches_X[0].shape[0], 30, 'relu', batchSize))
6 nn.addLayer(Layer(30, 10, 'relu', batchSize))
7 nn.addLayer(Layer(10, 1, 'linear', batchSize))
8
9 start = time.time()
10 # nn.fit(mini_batches_X, mini_batches_Y, miniValX, miniValY, lr=1e4,
11     ↪ epochAmount=5000)
12
13 # Load weights
14 nn.load_weights('nn_weights.npz')
15 end = time.time()
16
17 # nn.save_weights('nn_weights.npz')
18
19 print(f'Time elapsed: {end - start:.4f}')
20
21 # nn.plot_loss_history()
22
23 # Validation Performance
24 MSE = np.squeeze(nn.testLoss(miniValX.T, miniValY))
25 R2 = np.squeeze(nn.testLossR2(miniValX.T, miniValY))
26 print(f'Validation Performance (MSE): {MSE}')
27 print(f'Validation Performance (R2): {R2}')
28
29 # Test Performance
30 MSE = np.squeeze(nn.testLoss(miniTestX.T, miniTestY))
31 R2 = np.squeeze(nn.testLossR2(miniTestX.T, miniTestY))
32 print(f'Test Performance (MSE): {MSE}')
33 print(f'Test Performance (R2): {R2}')
34
35 demoInstanceLoc = 3
36 demoPredictor = predictorData[demoInstanceLoc]
37 demoResponse = responseData[demoInstanceLoc]
38 demoPredictor = np.expand_dims(demoPredictor, axis=0)
39 demoPrediction = np.squeeze(nn.predict(demoPredictor.T))
40 print('Prediction:', demoPrediction, 'Response:', demoResponse)

```

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

```

1 # Calculate the covariance matrix
2 cov_matrix = np.cov(df_standardized, rowvar=False)
3
4 # Calculate eigenvalues and eigenvectors
5 eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
6
7 # Sort eigenvalues and corresponding eigenvectors
8 sorted_indices = eigenvalues.argsort()[::-1]
9 eigenvalues = eigenvalues[sorted_indices]
10 eigenvectors = eigenvectors[:, sorted_indices]
11
12 # Proportion of Variance Explained
13 explained_variance_ratio = eigenvalues / np.sum(eigenvalues)
14 print("Proportion of Variance Explained:", explained_variance_ratio)
15 plt.figure(figsize=(8, 6))
16 plt.plot(range(1, len(explained_variance_ratio) + 1),
17         ↪ explained_variance_ratio, marker='o', linestyle='--')
18 plt.title('PVE')
19 plt.xlabel('Principal Component')
20 plt.ylabel('PVE Ratio')
21 plt.show()
22
23 # Cumulative explained variance
24 cumulative_variance = np.cumsum(explained_variance_ratio)
25 print("Cumulative Explained Variance:", cumulative_variance)
26 target_variance = 0.95
27 n_components = np.where(cumulative_variance >= target_variance)[0][0] + 1
28 print("Number of components to retain {}%
29     ↪ variance:".format(target_variance * 100), n_components)
30 plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance,
31         ↪ marker='o', linestyle='--')
32 plt.axhline(y=target_variance, color='r', linestyle='--', label='{}%
33     ↪ variance'.format(target_variance * 100))
34 plt.title('Cumulative Explained Variance')
35 plt.xlabel('Number of Components')
36 plt.ylabel('Cumulative Variance Ratio')
37 plt.legend()
38
39 # Transforming to the new space
40 transformed_data = np.dot(df_standardized, eigenvectors[:, :n_components])
41 principal_df = pd.DataFrame(data=transformed_data, columns=[f"PC{i}" for i
42     ↪ in range(1, n_components + 1)])
43
44 predictorFrame_scaled = principal_df

```

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

```

1 class NeuralNetwork:
2     def __init__(self):
3         self.layers = []
4         self.loss_history = []
5         self.test_loss_history = []
6
7     def addLayer(self, layer):
8         self.layers.append(layer)
9
10    def loss(self, predictions, y):
11        # MSE
12        batchSize = y.size
13        error = y - predictions
14        squaredError = np.dot(error.T, error)
15        mse = (1 / batchSize) * squaredError
16        return mse
17
18    def lossDer(self, predictions, y):
19        # MSE Derivative
20        batchSize = y.size
21        error = y - predictions
22        mseDer = (-2 / batchSize) * np.sum(error, axis=0, keepdims=True)
23        return mseDer
24
25    def predict(self, testPredictor):
26        output = testPredictor
27        for layer in self.layers:
28            printVals = True if False else False
29            output = layer.predict(output, printVals)
30        return output
31
32    def forward(self, trainPredictor):
33        output = trainPredictor
34        for layer in self.layers:
35            output = layer.updateForwardState(output)
36        return output
37
38    def backprop(self, predictions, y, x, lr):
39        # Update Delta State
40        for layerNumber in reversed(range(len(self.layers))):
41            layer = self.layers[layerNumber]
42            inputToLayer = self.layers[layerNumber - 1].A_state if
43                ↳ layerNumber > 0 else x
44
45            # Output Layer
46            if(layer == self.layers[-1]):
47                y_resaped = np.reshape(y, (1, y.size))
48                lossDerivative = self.lossDer(predictions, y_resaped)
49                # print('\nbackpropFirst():\n', 'predictions:',
50                    ↳ predictions.shape, 'y_resaped:', y_resaped.shape,
51                    ↳ 'lossDerivative:', lossDerivative.shape,
52                    ↳ 'inputToLayer', inputToLayer.shape)
53                layer.updateDeltaState(lossDerivative)
54                layer.calculateChange(inputToLayer)
55
56            # Hidden Layers
57            else:
58                dZ_next = nextLayer.dZ_state

```

```

54         W_next = nextLayer.W
55         dA = np.dot(W_next.T, dZ_next)
56         # print('\nbackpropAlt():\n', 'dZ_next:', dZ_next.shape,
57             ↪ 'W_next', W_next.shape, 'dA:', dA.shape)
58         layer.updateDeltaState(dA)
59         layer.calculateChange(inputToLayer)
60
61     nextLayer = layer
62
63     # Update Weights and Bias
64     for layerNumber in range(len(self.layers)):
65         layer = self.layers[layerNumber]
66         layer.updateWeightsAndBias(lr)
67
68     def fit(self, mini_batches_x, mini_batches_y, mini_test_x,
69         ↪ mini_test_y, lr=1e-2, epochAmount=10):
70         for epoch in range(epochAmount):
71             total_loss = 0
72             num_batches = len(mini_batches_x)
73             print('-----EPOCH-----      -----> ', epoch + 1)
74             # Train using mini-batches
75             for mini_batch_X, mini_batch_Y in zip(mini_batches_x,
76                 ↪ mini_batches_y):
77                 predictions = self.forward(mini_batch_X)
78                 self.backprop(predictions, mini_batch_Y, mini_batch_X, lr)
79
80                 # Compute loss for this mini-batch and accumulate
81                 loss = np.squeeze(self.loss(predictions.T, mini_batch_Y.T))
82                 total_loss += loss
83
84             # Average loss over all mini-batches
85             average_loss = total_loss / num_batches
86             print(f'Train Loss: {average_loss}')
87             self.loss_history.append((epoch, average_loss))
88
89             testError = np.squeeze(self.testLoss(mini_test_x.T,
90                 ↪ mini_test_y))
91             self.test_loss_history.append((epoch, testError))
92
93         print('Final Train Loss:', average_loss)
94
95     def testLoss(self, test_x, test_y):
96         predictions = np.squeeze(self.predict(test_x))
97         lossMSE = self.loss(predictions, test_y)
98         return lossMSE
99
100     def testLossR2(self, test_x, test_y):
101         predictions = np.squeeze(self.predict(test_x))
102         mean_observed = np.mean(test_y)
103         total_sum_squares = np.sum((test_y - mean_observed) ** 2)
104         residual_sum_squares = np.sum((test_y - predictions) ** 2)
105         r2 = 1 - (residual_sum_squares / total_sum_squares)
106         return r2
107
108     def plot_loss_history(self):
109         iterations, losses = zip(*self.loss_history)
110         _, testLosses = zip(*self.test_loss_history)
111         plt.plot(iterations, losses)

```

```

108     plt.plot(iterations, testLosses)
109     plt.xlabel('Epoch')
110     plt.ylabel('Training Loss')
111     plt.title(f'Training Loss over Epochs NN (lr = {1e4}, epoch =
    ↪ {5000})')
112     plt.legend(['Train', 'Test'])
113     plt.show()
114
115
116     def save_weights(self, filename):
117         # Create a dictionary to hold weights and biases of all layers
118         weights_dict = {}
119         for i, layer in enumerate(self.layers):
120             weights_dict[f"Layer_{i}_W"] = layer.W
121             weights_dict[f"Layer_{i}_b"] = layer.b
122
123         # Save the weights dictionary to a file
124         np.savez(filename, **weights_dict)
125
126     def load_weights(self, filename):
127         # Load the weights dictionary from the file
128         data = np.load(filename)
129
130         # Iterate through layers and load weights and biases
131         for i, layer in enumerate(self.layers):
132             layer.W = data[f"Layer_{i}_W"]
133             layer.b = data[f"Layer_{i}_b"]

```

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

```

1 class Layer:
2     def __init__(self, inputNumNeuron, numNeurons, activationName,
    ↪ batchSize, momentum=0.9):
3         self.batchSize = batchSize
4         self.inputNumNeuron = inputNumNeuron
5         self.numNeurons = numNeurons
6         self.activationName = activationName
7         self.activation = activationDict[self.activationName]
8         self.activationDerivative =
    ↪ activationDerivativeDict[self.activationName]
9         self.dZ_state = np.empty((numNeurons, batchSize))
10        self.Z_state = np.empty((numNeurons, batchSize))
11        self.A_state = np.empty((numNeurons, batchSize))
12        self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
13        self.db_state = np.zeros((self.numNeurons, 1))
14        self.initWeights()
15        # Momentum variables
16        self.momentum = momentum
17        self.dW_state_velocity = np.zeros((self.numNeurons,
    ↪ self.inputNumNeuron))
18        self.db_state_velocity = np.zeros((self.numNeurons, 1))
19
20    def initWeights(self):
21        # Random initialization unfortunately failed.
22        # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
23        # self.b = np.random.randn(self.numNeurons, 1)
24
25        # Xavier initialization for weights

```

```

26     self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
        ↳ np.sqrt(1 / self.inputNumNeuron)
27     # Initializing biases with zeros
28     self.b = np.zeros((self.numNeurons, 1))
29
30     def updateForwardState(self, inputToLayer):
31         # print('\nupdateForwardState():\n', 'inputToLayer:',
        ↳ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
        ↳ self.b.shape)
32         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
33         output = self.activation(inducedLocal)
34         self.Z_state = inducedLocal
35         self.A_state = output
36         return output
37
38     def predict(self, inputToLayer, printVals=False):
39         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
40         output = self.activation(inducedLocal)
41         if printVals:
42             print('inp:', self.b, self.W, 'out:', output)
43         return output
44
45     def updateDeltaState(self, dA):
46         # Derivative of loss over the weights of this layer
47         # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
        ↳ 'self.Z_state', self.Z_state.shape)
48         derActivation = self.activationDerivative(self.Z_state)
49         dZ = np.multiply(dA, derActivation)
50         self.dZ_state = dZ
51
52     def calculateChange(self, A_input):
53         self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
        ↳ A_input.T)
54         self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
        ↳ axis=1, keepdims=True)
55         # 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)
56         # 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)
57
58     def updateWeightsAndBias(self, lr):
59         # self.W = self.W - lr * self.dW_state
60         # self.b = self.b - lr * self.db_state
61
62         # Update weights and biases with momentum
63         self.dW_state_velocity = self.momentum * self.dW_state_velocity +
        ↳ lr * self.dW_state
64         self.db_state_velocity = self.momentum * self.db_state_velocity +
        ↳ lr * self.db_state
65         self.W = self.W - self.dW_state_velocity
66         self.b = self.b - self.db_state_velocity

```

Listing 16: Layer Class for Momentum Code

```

1 class Layer:
2     def __init__(self, inputNumNeuron, numNeurons, activationName,
3         ↪ batchSize, momentum=0.9):
4         self.batchSize = batchSize
5         self.inputNumNeuron = inputNumNeuron
6         self.numNeurons = numNeurons
7         self.activationName = activationName
8         self.activation = activationDict[self.activationName]
9         self.activationDerivative =
10            ↪ activationDerivativeDict[self.activationName]
11         self.dZ_state = np.empty((numNeurons, batchSize))
12         self.Z_state = np.empty((numNeurons, batchSize))
13         self.A_state = np.empty((numNeurons, batchSize))
14         self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
15         self.db_state = np.zeros((self.numNeurons, 1))
16         self.initWeights()
17         # Nesterov Accelerated Gradient variables
18         self.momentum = momentum
19         self.velocity_W = np.zeros((self.numNeurons, self.inputNumNeuron))
20         self.velocity_b = np.zeros((self.numNeurons, 1))
21
22     def initWeights(self):
23         # Random initialization unfortunately failed.
24         # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
25         # self.b = np.random.randn(self.numNeurons, 1)
26
27         # Xavier initialization for weights
28         self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
29            ↪ np.sqrt(1 / self.inputNumNeuron)
30         # Initializing biases with zeros
31         self.b = np.zeros((self.numNeurons, 1))
32
33     def updateForwardState(self, inputToLayer):
34         # print('\nupdateForwardState():\n', 'inputToLayer:',
35            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
36            ↪ self.b.shape)
37         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
38         output = self.activation(inducedLocal)
39         self.Z_state = inducedLocal
40         self.A_state = output
41         return output
42
43     def predict(self, inputToLayer, printVals=False):
44         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
45         output = self.activation(inducedLocal)
46         if printVals:
47             print('inp:', self.b, self.W, 'out:', output)
48         return output
49
50     def updateDeltaState(self, dA):
51         # Derivative of loss over the weights of this layer
52         # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
53            ↪ 'self.Z_state', self.Z_state.shape)
54         derActivation = self.activationDerivative(self.Z_state)
55         dZ = np.multiply(dA, derActivation)
56         self.dZ_state = dZ

```



```

52 def calculateChange(self, A_input):
53     self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
54         ↪ A_input.T)
55     self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
56         ↪ axis=1, keepdims=True)
57     # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
58         ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
59         ↪ 'self.Z_state', self.Z_state)
60     # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
61         ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
62         ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)
63
64 def updateWeightsAndBias(self, lr):
65     # self.W = self.W - lr * self.dW_state
66     # self.b = self.b - lr * self.db_state
67
68     # Update weights and biases with Nesterov Accelerated Gradient
69     self.velocity_W = self.momentum * self.velocity_W - lr *
70         ↪ self.dW_state
71     self.velocity_b = self.momentum * self.velocity_b - lr *
72         ↪ self.db_state
73     self.W += self.velocity_W
74     self.b += self.velocity_b

```

Listing 17: Layer Class for Nesterov Code

```

1 class Layer:
2     def __init__(self, inputNumNeuron, numNeurons, activationName,
3         ↪ batchSize, epsilon=1e-8):
4         self.batchSize = batchSize
5         self.inputNumNeuron = inputNumNeuron
6         self.numNeurons = numNeurons
7         self.activationName = activationName
8         self.activation = activationDict[self.activationName]
9         self.activationDerivative =
10             ↪ activationDerivativeDict[self.activationName]
11         self.dZ_state = np.empty((numNeurons, batchSize))
12         self.Z_state = np.empty((numNeurons, batchSize))
13         self.A_state = np.empty((numNeurons, batchSize))
14         self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
15         self.db_state = np.zeros((self.numNeurons, 1))
16         self.initWeights()
17         # AdaGrad variables
18         self.epsilon = epsilon
19         self.squared_gradient_W = np.zeros((self.numNeurons,
20             ↪ self.inputNumNeuron))
21         self.squared_gradient_b = np.zeros((self.numNeurons, 1))
22
23     def initWeights(self):
24         # Random initialization unfortunately failed.
25         # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
26         # self.b = np.random.randn(self.numNeurons, 1)
27
28         # Xavier initialization for weights
29         self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
30             ↪ np.sqrt(1 / self.inputNumNeuron)
31         # Initializing biases with zeros
32         self.b = np.zeros((self.numNeurons, 1))

```

```

29
30 def updateForwardState(self, inputToLayer):
31     # print('\nupdateForwardState():\n', 'inputToLayer:',
        ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
        ↪ self.b.shape)
32     inducedLocal = np.matmul(self.W, inputToLayer) + self.b
33     output = self.activation(inducedLocal)
34     self.Z_state = inducedLocal
35     self.A_state = output
36     return output
37
38 def predict(self, inputToLayer, printVals=False):
39     inducedLocal = np.matmul(self.W, inputToLayer) + self.b
40     output = self.activation(inducedLocal)
41     if printVals:
42         print('inp:', self.b, self.W, 'out:', output)
43     return output
44
45 def updateDeltaState(self, dA):
46     # Derivative of loss over the weights of this layer
47     # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
        ↪ 'self.Z_state', self.Z_state.shape)
48     derActivation = self.activationDerivative(self.Z_state)
49     dZ = np.multiply(dA, derActivation)
50     self.dZ_state = dZ
51
52 def calculateChange(self, A_input):
53     self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
        ↪ A_input.T)
54     self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
        ↪ axis=1, keepdims=True)
55     # 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)
56     # 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)
57
58 def updateWeightsAndBias(self, lr):
59     # self.W = self.W - lr * self.dW_state
60     # self.b = self.b - lr * self.db_state
61
62     # Update weights and biases with AdaGrad
63     self.squared_gradient_W += np.square(self.dW_state)
64     self.squared_gradient_b += np.square(self.db_state)
65
66     self.W -= lr * (self.dW_state / (np.sqrt(self.squared_gradient_W)
        ↪ + self.epsilon))
67     self.b -= lr * (self.db_state / (np.sqrt(self.squared_gradient_b)
        ↪ + self.epsilon))

```

Listing 18: Layer Class for AdaGrad Code

```

1 class Layer:
2     def __init__(self, inputNumNeuron, numNeurons, activationName,
3         ↪ batchSize, epsilon=1e-8, decay_rate=0.9):
4         self.batchSize = batchSize
5         self.inputNumNeuron = inputNumNeuron
6         self.numNeurons = numNeurons
7         self.activationName = activationName
8         self.activation = activationDict[self.activationName]
9         self.activationDerivative =
10            ↪ activationDerivativeDict[self.activationName]
11         self.dZ_state = np.empty((numNeurons, batchSize))
12         self.Z_state = np.empty((numNeurons, batchSize))
13         self.A_state = np.empty((numNeurons, batchSize))
14         self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
15         self.db_state = np.zeros((self.numNeurons, 1))
16         self.initWeights()
17         # RMSProp variables
18         self.epsilon = epsilon
19         self.decay_rate = decay_rate
20         self.squared_gradient_W = np.zeros((self.numNeurons,
21            ↪ self.inputNumNeuron))
22         self.squared_gradient_b = np.zeros((self.numNeurons, 1))
23
24     def initWeights(self):
25         # Random initialization unfortunately failed.
26         # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
27         # self.b = np.random.randn(self.numNeurons, 1)
28
29         # Xavier initialization for weights
30         self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
31            ↪ np.sqrt(1 / self.inputNumNeuron)
32         # Initializing biases with zeros
33         self.b = np.zeros((self.numNeurons, 1))
34
35     def updateForwardState(self, inputToLayer):
36         # print('\nupdateForwardState():\n', 'inputToLayer:',
37            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
38            ↪ self.b.shape)
39         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
40         output = self.activation(inducedLocal)
41         self.Z_state = inducedLocal
42         self.A_state = output
43         return output
44
45     def predict(self, inputToLayer, printVals=False):
46         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
47         output = self.activation(inducedLocal)
48         if printVals:
49             print('inp:', self.b, self.W, 'out:', output)
50         return output
51
52     def updateDeltaState(self, dA):
53         # Derivative of loss over the weights of this layer
54         # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
55            ↪ 'self.Z_state', self.Z_state.shape)
56         derActivation = self.activationDerivative(self.Z_state)
57         dZ = np.multiply(dA, derActivation)

```

```

51     self.dZ_state = dZ
52
53     def calculateChange(self, A_input):
54         self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
55             ↪ A_input.T)
56         self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
57             ↪ axis=1, keepdims=True)
58         # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
59             ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
60             ↪ 'self.Z_state', self.Z_state)
61         # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
62             ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
63             ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)
64
65     def updateWeightsAndBias(self, lr):
66         # self.W = self.W - lr * self.dW_state
67         # self.b = self.b - lr * self.db_state
68
69         # Update weights and biases with RMSProp
70         self.squared_gradient_W = self.decay_rate *
71             ↪ self.squared_gradient_W + (1 - self.decay_rate) *
72             ↪ np.square(self.dW_state)
73         self.squared_gradient_b = self.decay_rate *
74             ↪ self.squared_gradient_b + (1 - self.decay_rate) *
75             ↪ np.square(self.db_state)
76
77         self.W -= lr * (self.dW_state / (np.sqrt(self.squared_gradient_W)
78             ↪ + self.epsilon))
79         self.b -= lr * (self.db_state / (np.sqrt(self.squared_gradient_b)
80             ↪ + self.epsilon))

```

Listing 19: Layer Class for RMSProp Code

```

1 class Layer:
2     def __init__(self, inputNumNeuron, numNeurons, activationName,
3         ↪ batchSize, beta1=0.9, beta2=0.999, epsilon=1e-8):
4         self.batchSize = batchSize
5         self.inputNumNeuron = inputNumNeuron
6         self.numNeurons = numNeurons
7         self.activationName = activationName
8         self.activation = activationDict[self.activationName]
9         self.activationDerivative =
10            ↪ activationDerivativeDict[self.activationName]
11         self.dZ_state = np.empty((numNeurons, batchSize))
12         self.Z_state = np.empty((numNeurons, batchSize))
13         self.A_state = np.empty((numNeurons, batchSize))
14         self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
15         self.db_state = np.zeros((self.numNeurons, 1))
16         self.initWeights()
17         # Adam variables
18         self.beta1 = beta1
19         self.beta2 = beta2
20         self.epsilon = epsilon
21         self.moment_W = np.zeros((self.numNeurons, self.inputNumNeuron))
22         self.moment_b = np.zeros((self.numNeurons, 1))
23         self.velocity_W = np.zeros((self.numNeurons, self.inputNumNeuron))
24         self.velocity_b = np.zeros((self.numNeurons, 1))
25         self.iteration = 0
26
27     def initWeights(self):
28         # Random initialization unfortunately failed.
29         # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
30         # self.b = np.random.randn(self.numNeurons, 1)
31
32         # Xavier initialization for weights
33         self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
34            ↪ np.sqrt(1 / self.inputNumNeuron)
35         # Initializing biases with zeros
36         self.b = np.zeros((self.numNeurons, 1))
37
38     def updateForwardState(self, inputToLayer):
39         # print('\nupdateForwardState():\n', 'inputToLayer:',
40            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
41            ↪ self.b.shape)
42         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
43         output = self.activation(inducedLocal)
44         self.Z_state = inducedLocal
45         self.A_state = output
46         return output
47
48     def predict(self, inputToLayer, printVals=False):
49         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
50         output = self.activation(inducedLocal)
51         if printVals:
52             print('inp:', self.b, self.W, 'out:', output)
53         return output
54
55     def updateDeltaState(self, dA):
56         # Derivative of loss over the weights of this layer
57         # print('\nupdateDeltaState():\n', 'dA:', dA.shape,

```

```

53         ↪ 'self.Z_state', self.Z_state.shape)
54     derActivation = self.activationDerivative(self.Z_state)
55     dZ = np.multiply(dA, derActivation)
56     self.dZ_state = dZ
57
58     def calculateChange(self, A_input):
59         self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
60             ↪ A_input.T)
61         self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
62             ↪ axis=1, keepdims=True)
63         # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
64             ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
65             ↪ 'self.Z_state', self.Z_state)
66         # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
67             ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
68             ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)
69
70     def updateWeightsAndBias(self, lr):
71         # self.W = self.W - lr * self.dW_state
72         # self.b = self.b - lr * self.db_state
73
74         # Update weights and biases with Adam
75         self.iteration += 1
76         self.moment_W = self.betal * self.moment_W + (1 - self.betal) *
77             ↪ self.dW_state
78         self.moment_b = self.betal * self.moment_b + (1 - self.betal) *
79             ↪ self.db_state
80         self.velocity_W = self.beta2 * self.velocity_W + (1 - self.beta2)
81             ↪ * np.square(self.dW_state)
82         self.velocity_b = self.beta2 * self.velocity_b + (1 - self.beta2)
83             ↪ * np.square(self.db_state)
84
85         moment_W_hat = self.moment_W / (1 - self.betal ** self.iteration)
86         moment_b_hat = self.moment_b / (1 - self.betal ** self.iteration)
87         velocity_W_hat = self.velocity_W / (1 - self.beta2 **
88             ↪ self.iteration)
89         velocity_b_hat = self.velocity_b / (1 - self.beta2 **
90             ↪ self.iteration)
91
92         self.W -= lr * moment_W_hat / (np.sqrt(velocity_W_hat) +
93             ↪ self.epsilon)
94         self.b -= lr * moment_b_hat / (np.sqrt(velocity_b_hat) +
95             ↪ self.epsilon)

```

Listing 20: Layer Class for Adam Code

```

1 class Layer:
2     def __init__(self, inputNumNeuron, numNeurons, activationName,
3         ↪ batchSize, beta1=0.9, beta2=0.999, epsilon=1e-8):
4         self.batchSize = batchSize
5         self.inputNumNeuron = inputNumNeuron
6         self.numNeurons = numNeurons
7         self.activationName = activationName
8         self.activation = activationDict[self.activationName]
9         self.activationDerivative =
10            ↪ activationDerivativeDict[self.activationName]
11         self.dZ_state = np.empty((numNeurons, batchSize))
12         self.Z_state = np.empty((numNeurons, batchSize))
13         self.A_state = np.empty((numNeurons, batchSize))
14         self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
15         self.db_state = np.zeros((self.numNeurons, 1))
16         self.initWeights()
17         # AMSGrad variables
18         self.beta1 = beta1
19         self.beta2 = beta2
20         self.epsilon = epsilon
21         self.moment_W = np.zeros((self.numNeurons, self.inputNumNeuron))
22         self.moment_b = np.zeros((self.numNeurons, 1))
23         self.velocity_W = np.zeros((self.numNeurons, self.inputNumNeuron))
24         self.velocity_b = np.zeros((self.numNeurons, 1))
25         self.velocity_W_max = np.zeros((self.numNeurons,
26            ↪ self.inputNumNeuron))
27         self.velocity_b_max = np.zeros((self.numNeurons, 1))
28         self.iteration = 0
29
30     def initWeights(self):
31         # Random initialization unfortunately failed.
32         # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
33         # self.b = np.random.randn(self.numNeurons, 1)
34
35         # Xavier initialization for weights
36         self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
37            ↪ np.sqrt(1 / self.inputNumNeuron)
38         # Initializing biases with zeros
39         self.b = np.zeros((self.numNeurons, 1))
40
41     def updateForwardState(self, inputToLayer):
42         # print('\nupdateForwardState():\n', 'inputToLayer:',
43            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
44            ↪ self.b.shape)
45         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
46         output = self.activation(inducedLocal)
47         self.Z_state = inducedLocal
48         self.A_state = output
49         return output
50
51     def predict(self, inputToLayer, printVals=False):
52         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
53         output = self.activation(inducedLocal)
54         if printVals:
55             print('inp:', self.b, self.W, 'out:', output)
56         return output

```

```

52 def updateDeltaState(self, dA):
53     # Derivative of loss over the weights of this layer
54     # print('\nupdateDeltaState():\n', 'dA:', dA.shape,
55         ↪ 'self.Z_state', self.Z_state.shape)
56     derActivation = self.activationDerivative(self.Z_state)
57     dZ = np.multiply(dA, derActivation)
58     self.dZ_state = dZ
59
60 def calculateChange(self, A_input):
61     self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
62         ↪ A_input.T)
63     self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
64         ↪ axis=1, keepdims=True)
65     # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
66         ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
67         ↪ 'self.Z_state', self.Z_state)
68     # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
69         ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
70         ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)
71
72 def updateWeightsAndBias(self, lr):
73     # self.W = self.W - lr * self.dW_state
74     # self.b = self.b - lr * self.db_state
75
76     # Update weights and biases with AMSGrad
77     self.iteration += 1
78     self.moment_W = self.betal * self.moment_W + (1 - self.betal) *
79         ↪ self.dW_state
80     self.moment_b = self.betal * self.moment_b + (1 - self.betal) *
81         ↪ self.db_state
82     self.velocity_W = self.beta2 * self.velocity_W + (1 - self.beta2)
83         ↪ * np.square(self.dW_state)
84     self.velocity_b = self.beta2 * self.velocity_b + (1 - self.beta2)
85         ↪ * np.square(self.db_state)
86
87     self.velocity_W_max = np.maximum(self.velocity_W_max,
88         ↪ self.velocity_W)
89     self.velocity_b_max = np.maximum(self.velocity_b_max,
90         ↪ self.velocity_b)
91
92     moment_W_hat = self.moment_W / (1 - self.betal ** self.iteration)
93     moment_b_hat = self.moment_b / (1 - self.betal ** self.iteration)
94
95     self.W -= lr * moment_W_hat / (np.sqrt(self.velocity_W_max) +
96         ↪ self.epsilon)
97     self.b -= lr * moment_b_hat / (np.sqrt(self.velocity_b_max) +
98         ↪ self.epsilon)

```

Listing 21: Layer Class for AMSGrad Code


```

1 class Layer:
2     def __init__(self, inputNumNeuron, numNeurons, activationName,
3         ↪ batchSize, beta1=0.9, beta2=0.999, epsilon=1e-8):
4         self.batchSize = batchSize
5         self.inputNumNeuron = inputNumNeuron
6         self.numNeurons = numNeurons
7         self.activationName = activationName
8         self.activation = activationDict[self.activationName]
9         self.activationDerivative =
10            ↪ activationDerivativeDict[self.activationName]
11         self.dZ_state = np.empty((numNeurons, batchSize))
12         self.Z_state = np.empty((numNeurons, batchSize))
13         self.A_state = np.empty((numNeurons, batchSize))
14         self.dW_state = np.zeros((self.numNeurons, self.inputNumNeuron))
15         self.db_state = np.zeros((self.numNeurons, 1))
16         self.initWeights()
17         # Adam variables
18         self.beta1 = beta1
19         self.beta2 = beta2
20         self.epsilon = epsilon
21         self.moment_W = np.zeros((self.numNeurons, self.inputNumNeuron))
22         self.moment_b = np.zeros((self.numNeurons, 1))
23         self.velocity_W = np.zeros((self.numNeurons, self.inputNumNeuron))
24         self.velocity_b = np.zeros((self.numNeurons, 1))
25         self.iteration = 0
26
27     def initWeights(self):
28         # Random initialization unfortunately failed.
29         # self.W = np.random.randn(self.numNeurons, self.inputNumNeuron)
30         # self.b = np.random.randn(self.numNeurons, 1)
31
32         # Xavier initialization for weights
33         self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) *
34            ↪ np.sqrt(1 / self.inputNumNeuron)
35         # Initializing biases with zeros
36         self.b = np.zeros((self.numNeurons, 1))
37
38     def updateForwardState(self, inputToLayer):
39         # print('\nupdateForwardState():\n', 'inputToLayer:',
40            ↪ inputToLayer.shape, 'self.W', self.W.shape, 'self.b',
41            ↪ self.b.shape)
42         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
43         output = self.activation(inducedLocal)
44         self.Z_state = inducedLocal
45         self.A_state = output
46         return output
47
48     def predict(self, inputToLayer, printVals=False):
49         inducedLocal = np.matmul(self.W, inputToLayer) + self.b
50         output = self.activation(inducedLocal)
51         if printVals:
52             print('inp:', self.b, self.W, 'out:', output)
53         return output
54
55     def updateDeltaState(self, dA):
56         # Derivative of loss over the weights of this layer
57         # print('\nupdateDeltaState():\n', 'dA:', dA.shape,

```

```

53         ↪ 'self.Z_state', self.Z_state.shape)
54     derActivation = self.activationDerivative(self.Z_state)
55     dZ = np.multiply(dA, derActivation)
56     self.dZ_state = dZ
57
58     def calculateChange(self, A_input):
59         self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
60             ↪ A_input.T)
61         self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state,
62             ↪ axis=1, keepdims=True)
63         # print('\ncalculateChange():\n', 'self.dW_state:', self.dW_state,
64             ↪ 'self.db_state', self.db_state, 'A_input:', A_input.T,
65             ↪ 'self.Z_state', self.Z_state)
66         # print('\ncalculateChange():\n', 'A_input:', A_input.T.shape,
67             ↪ 'self.Z_state', self.Z_state.shape, 'self.dW_state',
68             ↪ self.dW_state.shape, 'self.db_state', self.db_state.shape)
69
70     def updateWeightsAndBias(self, lr):
71         # self.W = self.W - lr * self.dW_state
72         # self.b = self.b - lr * self.db_state
73
74         # Update weights and biases with Adam
75         self.iteration += 1
76         self.moment_W = self.betal * self.moment_W + (1 - self.betal) *
77             ↪ self.dW_state
78         self.moment_b = self.betal * self.moment_b + (1 - self.betal) *
79             ↪ self.db_state
80         self.velocity_W = self.beta2 * self.velocity_W + (1 - self.beta2)
81             ↪ * np.square(self.dW_state)
82         self.velocity_b = self.beta2 * self.velocity_b + (1 - self.beta2)
83             ↪ * np.square(self.db_state)
84
85         moment_W_hat = self.moment_W / (1 - self.betal ** self.iteration)
86         moment_b_hat = self.moment_b / (1 - self.betal ** self.iteration)
87         velocity_W_hat = self.velocity_W / (1 - self.beta2 **
88             ↪ self.iteration)
89         velocity_b_hat = self.velocity_b / (1 - self.beta2 **
90             ↪ self.iteration)
91
92         self.W -= lr * moment_W_hat / (np.sqrt(velocity_W_hat) +
93             ↪ self.epsilon)
94         self.b -= lr * moment_b_hat / (np.sqrt(velocity_b_hat) +
95             ↪ self.epsilon)
96
97 class NeuralNetwork:
98     def __init__(self):
99         self.layers = []
100         self.loss_history = []
101         self.test_loss_history = []
102
103     def addLayer(self, layer):
104         self.layers.append(layer)
105
106     def loss(self, predictions, y):
107         # MSE
108         batchSize = y.size
109         error = y - predictions
110         squaredError = np.dot(error.T, error)

```

```

96         mse = (1 / batchSize) * squaredError
97         return mse
98
99     def lossDer(self, predictions, y):
100         # MSE Derivative
101         batchSize = y.size
102         error = y - predictions
103         mseDer = (-2 / batchSize) * np.sum(error, axis=0, keepdims=True)
104         return mseDer
105
106     def predict(self, testPredictor):
107         output = testPredictor
108         for layer in self.layers:
109             printVals = True if False else False
110             output = layer.predict(output, printVals)
111         return output
112
113     def forward(self, trainPredictor):
114         output = trainPredictor
115         for layer in self.layers:
116             output = layer.updateForwardState(output)
117         return output
118
119     def backprop(self, predictions, y, x, lr):
120         # Update Delta State
121         for layerNumber in reversed(range(len(self.layers))):
122             layer = self.layers[layerNumber]
123             inputToLayer = self.layers[layerNumber - 1].A_state if
124                 ↪ layerNumber > 0 else x
125
126             # Output Layer
127             if(layer == self.layers[-1]):
128                 y_resaped = np.reshape(y, (1, y.size))
129                 lossDerivative = self.lossDer(predictions, y_resaped)
130                 # print('\nbackpropFirst():\n', 'predictions:',
131                     ↪ predictions.shape, 'y_resaped:', y_resaped.shape,
132                     ↪ 'lossDerivative:', lossDerivative.shape,
133                     ↪ 'inputToLayer', inputToLayer.shape)
134                 layer.updateDeltaState(lossDerivative)
135                 layer.calculateChange(inputToLayer)
136
137             # Hidden Layers
138             else:
139                 dZ_next = nextLayer.dZ_state
140                 W_next = nextLayer.W
141                 dA = np.dot(W_next.T, dZ_next)
142                 # print('\nbackpropAlt():\n', 'dZ_next:', dZ_next.shape,
143                     ↪ 'W_next', W_next.shape, 'dA:', dA.shape)
144                 layer.updateDeltaState(dA)
145                 layer.calculateChange(inputToLayer)
146
147             nextLayer = layer
148
149         # Update Weights and Bias
150         for layerNumber in range(len(self.layers)):
151             layer = self.layers[layerNumber]
152             layer.updateWeightsAndBias(lr)
153
154     def fit(self, mini_batches_x, mini_batches_y, mini_test_x,

```

```

149     ↪ mini_test_y, lr=1e-2, epochAmount=10):
150     # self.load_weights('nn_weights_adam_large_343.npz')
151     for epoch in range(epochAmount):
152         total_loss = 0
153         num_batches = len(mini_batches_x)
154         print('-----EPOCH-----> ', epoch + 1)
155         # Train using mini-batches
156         for mini_batch_X, mini_batch_Y in zip(mini_batches_x,
157             ↪ mini_batches_y):
158             predictions = self.forward(mini_batch_X)
159             self.backprop(predictions, mini_batch_Y, mini_batch_X, lr)
160
161             # Compute loss for this mini-batch and accumulate
162             loss = np.squeeze(self.loss(predictions.T, mini_batch_Y.T))
163             total_loss += loss
164
165             # Average loss over all mini-batches
166             average_loss = total_loss / num_batches
167             print(f'Train Loss: {average_loss}')
168             self.loss_history.append((epoch, average_loss))
169
170             testError = np.squeeze(self.testLoss(mini_test_x.T,
171             ↪ mini_test_y))
172             self.test_loss_history.append((epoch, testError))
173
174         print('Final Train Loss:', average_loss)
175
176     def testLoss(self, test_x, test_y):
177         predictions = np.squeeze(self.predict(test_x))
178         lossMSE = self.loss(predictions, test_y)
179         return lossMSE
180
181     def testLossR2(self, test_x, test_y):
182         predictions = np.squeeze(self.predict(test_x))
183         mean_observed = np.mean(test_y)
184         total_sum_squares = np.sum((test_y - mean_observed) ** 2)
185         residual_sum_squares = np.sum((test_y - predictions) ** 2)
186         r2 = 1 - (residual_sum_squares / total_sum_squares)
187         return r2
188
189     def plot_loss_history(self):
190         iterations, losses = zip(*self.loss_history)
191         _, testLosses = zip(*self.test_loss_history)
192         plt.plot(iterations, losses)
193         plt.plot(iterations, testLosses)
194         plt.xlabel('Epoch')
195         plt.ylabel('Training Loss')
196         plt.title(f'Training Loss over Epochs NN (lr = {1e4}, epoch =
197             ↪ {5000})')
198         plt.legend(['Train', 'Test'])
199         plt.show()
200
201     def save_weights(self, filename):
202         # Create a dictionary to hold weights and biases of all layers
203         weights_dict = {}
204         for i, layer in enumerate(self.layers):
205             weights_dict[f"Layer_{i}_W"] = layer.W

```

```

203         weights_dict[f"Layer_{i}_b"] = layer.b
204
205     # Save the weights dictionary to a file
206     np.savez(filename, **weights_dict)
207
208     def load_weights(self, filename):
209         # Load the weights dictionary from the file
210         data = np.load(filename)
211
212         # Iterate through layers and load weights and biases
213         for i, layer in enumerate(self.layers):
214             layer.W = data[f"Layer_{i}_W"]
215             layer.b = data[f"Layer_{i}_b"]

```

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

```

1 class KNNRegression:
2     def __init__(self, k):
3         self.k = k
4
5     def fit(self, X_train, y_train):
6         self.X_train = X_train
7         self.y_train = y_train
8
9     def predict(self, X_test, batch_size=200):
10        predictions = []
11        n_test_samples = X_test.shape[0]
12        print('Total batches:', n_test_samples // batch_size)
13
14        cnt = 0
15        for i in range(0, n_test_samples, batch_size):
16            cnt += 1
17            print('Batch processsed:', cnt)
18            X_batch = X_test[i:i+batch_size]
19
20            # Euclidean distances between each point
21            dists = np.sqrt(np.sum((X_batch[:, np.newaxis] -
22                                   → self.X_train)**2, axis=2))
23
24            # Get indices of the k-nearest neighbors
25            nearest_neighbors_indices = np.argsort(dists, axis=1)[:,
26                                   → :self.k]
27
28            # Get the corresponding target values of the nearest neighbors
29            nearest_neighbors_targets =
30                                   → self.y_train[nearest_neighbors_indices]
31
32            batch_predictions = np.mean(nearest_neighbors_targets, axis=1)
33
34            predictions.extend(batch_predictions)
35
36        return np.array(predictions)

```

Listing 23: KNN Regression code

```

1 def mse(testResponse, predictions):
2     error = testResponse - predictions
3     squaredError = np.dot(error.T, error)
4     meanSquaredError = 1/(testResponse.size) * squaredError
5     return meanSquaredError
6
7 # Validation Performance
8 MSEs = []
9 K = [1, 3, 5, 10, 100]
10 for k in K:
11     knn = KNNRegression(k=k)
12     knn.fit(trainPredictor, trainResponse)
13     predictions = knn.predict(valPredictor, batch_size=456)
14     MSE = mse(valResponse, predictions)
15     print("Mean Squared Error (MSE):", MSE)
16     MSEs.append(MSE)
17
18 x_values = np.arange(len(K))
19
20 # Plot K versus MSE
21 plt.plot(x_values, MSEs, linestyle='-', marker='o')
22 plt.title('Parameter K vs. Val Loss (MSE)')
23 plt.xlabel('K')
24 plt.ylabel('Val Loss (MSE)')
25 # Set the x-ticks to be the original K values
26 plt.xticks(x_values, K)
27 plt.grid(True)
28 plt.show()
29
30 # Test Performance
31 def r2_score(testResponse, predictions):
32     mean_observed = np.mean(testResponse)
33     total_sum_squares = np.sum((testResponse - mean_observed) ** 2)
34     residual_sum_squares = np.sum((testResponse - predictions) ** 2)
35     r2 = 1 - (residual_sum_squares / total_sum_squares)
36
37     return r2
38
39 knn = KNNRegression(k=5)
40 knn.fit(trainPredictor, trainResponse)
41 predictions = knn.predict(testPredictor, batch_size=456)
42 MSE = mse(testResponse, predictions)
43 R2 = r2_score(testResponse, predictions)
44 print(f'Test Performance (MSE): {MSE}')
45 print(f'Test Performance (R2): {R2}')
46
47 demoInstanceLoc = 11401
48 demoPredictor = predictorData[demoInstanceLoc]
49 demoResponse = responseData[demoInstanceLoc]
50 demoPredictor = np.expand_dims(demoPredictor, axis=0)
51 demoPrediction = knn.predict(demoPredictor, batch_size=1)
52 print('Prediction:', demoPrediction, 'Response:', demoResponse)

```

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

```

1 import numpy as np
2 import pandas as pd
3 from matplotlib import pyplot as plt
4 import seaborn as sns
5
6 # Preprocess
7 dataPath = "./data/dataset.csv"
8 df = pd.read_csv(dataPath, index_col=0)
9 columns = list(df.columns)
10 columnsToKeep = columns[4: -1]
11 columnsToKeep
12 df = df[columnsToKeep]
13
14 # Describe Dataset
15 df.head()
16 pd.isnull(df).sum()
17
18 # Preprocess
19 df.loc[:, 'explicit'] = df['explicit'].astype(int)
20 # One hot encoding for nominal categories
21 df = pd.get_dummies(df, columns=['key', 'time_signature'], dtype=int)
22
23 def min_max_scaling(df):
24     min_vals = df.min()
25     max_vals = df.max()
26
27     feature_range = max_vals - min_vals
28
29     # Check if any feature has zero range
30     zero_range_features = feature_range[feature_range == 0].index
31
32     # Remove features with zero range from normalization
33     valid_features = feature_range[feature_range != 0].index
34     df_normalized = (df[valid_features] - min_vals[valid_features]) /
35         ↪ feature_range[valid_features]
36
37     # Concatenate back the zero range features
38     if not zero_range_features.empty:
39         df_normalized = pd.concat([df_normalized,
40             ↪ df[zero_range_features]], axis=1)
41
42     return df_normalized
43
44 def standard_scaling(df):
45     mean = df.mean()
46     std = df.std()
47     return (df - mean) / std
48
49 responseFrame = df.pop('valence')
50 predictorFrame = df
51
52 # Min-Max scaling for predictor variables
53 df_normalized = min_max_scaling(predictorFrame)
54
55 # Standard scaling for predictor variables
56 df_standardized = standard_scaling(predictorFrame)

```

```

56 # predictorFrame_scaled = df_normalized
57 # df.info()
58
59
60 # PCA
61 # predictorFrame = df
62 # # Standard scaling for predictor variables
63 # df_standardized = standard_scaling(predictorFrame)
64
65 # # Calculate the covariance matrix
66 # cov_matrix = np.cov(df_standardized, rowvar=False)
67
68 # # Calculate eigenvalues and eigenvectors
69 # eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
70
71 # # Sort eigenvalues and corresponding eigenvectors
72 # sorted_indices = eigenvalues.argsort()[::-1]
73 # eigenvalues = eigenvalues[sorted_indices]
74 # eigenvectors = eigenvectors[:, sorted_indices]
75
76 # # Proportion of Variance Explained
77 # explained_variance_ratio = eigenvalues / np.sum(eigenvalues)
78 # print("Proportion of Variance Explained:", explained_variance_ratio)
79 # plt.figure(figsize=(8, 6))
80 # plt.plot(range(1, len(explained_variance_ratio) + 1),
81 #         ↪ explained_variance_ratio, marker='o', linestyle='--')
82 # plt.title('PVE')
83 # plt.xlabel('Principal Component')
84 # plt.ylabel('PVE Ratio')
85 # plt.show()
86
87 # # Cumulative explained variance
88 # cumulative_variance = np.cumsum(explained_variance_ratio)
89 # print("Cumulative Explained Variance:", cumulative_variance)
90 # target_variance = 0.95
91 # n_components = np.where(cumulative_variance >= target_variance)[0][0] + 1
92 # print("Number of components to retain {}%
93 #       ↪ variance:".format(target_variance * 100), n_components)
94
95 # # Transforming to the new space
96 # transformed_data = np.dot(df_standardized, eigenvectors[:,
97 #         ↪ :n_components])
98 # principal_df = pd.DataFrame(data=transformed_data, columns=[f"PC{i}" for
99 #         ↪ i in range(1, n_components + 1)])
100
101 # # Concatenate the principal components DataFrame with the original
102 #         ↪ DataFrame
103 # final_df = pd.concat([principal_df, df.reset_index()], axis=1)
104
105 df.sort_values('valence', ascending = False).head(10)
106 df.describe().transpose()
107
108 # Correlation heatmap
109 corr_df = df.corr(method="pearson")
110
111 plt.figure(figsize=(12, 9))
112 heatmap = sns.heatmap(corr_df, annot=True, fmt=".1g",

```



```

109         vmin=-1, vmax=1, center=0, cmap="inferno",
110         linewidths=1, linecolor="black")
111 heatmap.set_title("Correlation Heatmap Between Variables")
112 heatmap.set_xticklabels(heatmap.get_xticklabels(), rotation=90)
113 plt.show()
114
115 # Correlation of Features (Sampled)
116 # Sample a smaller subset of data points
117 sampled_df = df.sample(n=1000, random_state=42)
118
119 # List of features to plot
120 features = ["danceability", "energy", "loudness", "speechiness",
121            ↪ "acousticness", "instrumentalness", "liveness", "tempo"]
122
123 # Set up the figure with subplots
124 fig, axes = plt.subplots(nrows=2, ncols=4, figsize=(16, 8))
125 fig.suptitle("Features vs. Valence", fontsize=16)
126
127 # Flatten the axes array for easy iteration
128 axes = axes.flatten()
129
130 # Loop through features and plot each one against valence
131 for i, feature in enumerate(features):
132     sns.regplot(data=sampled_df, y=feature, x="valence", ax=axes[i])
133     axes[i].set_title(f"{feature.capitalize()} vs. Valence")
134
135 # Adjust layout
136 plt.tight_layout()
137 plt.show()
138
139 # Histograms
140 df.hist(figsize=(20, 20))
141 plt.show()
142
143 # Joint Correlation Plot
144 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].