EEE 485 Statistical Learning and Data Analytics

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

Full Name: Arda Iynem
Student ID: 22002717

April 21, 2024

Contents

1 Introduction to the Problem

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

2 Dataset Analysis

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

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

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

2.1 Preprocessing

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

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

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

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

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

$$\frac{x_j - x_{\min_j}}{x_{\max_j} - x_{\min_j}} \tag{2}$$

PCA: PCA method [5] applied for future but not utilized for this phase yet, PVEs corresponding to PCs are reported for further analysis (Appendix A, Fig. 5).

After applying the mentioned preprocessing methods dataset obtains it's final state (Appendix A, Fig. 6) with shape (114000 rows x 29 columns), which is ready for training step.

3 Training Methodology

3.1 Linear Regression

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

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

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \epsilon = \boldsymbol{\beta}^T \mathbf{x} + \epsilon \tag{3}$$

3.1.1 Likelihood Function

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

$$f(y_i|\mathbf{x}_i, \boldsymbol{\beta}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2}{2\sigma^2}\right) \tag{4}$$

$$\mathcal{L}(\boldsymbol{\beta}) = \prod_{i=1}^{n} f(y_i|\mathbf{x}_i, \boldsymbol{\beta}) \tag{5}$$

$$\mathcal{L}(\boldsymbol{\beta}) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2}{2\sigma^2}\right) \tag{6}$$

3.1.2 Maximum Likelihood Estimation

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

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

$$\ell(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(-\frac{1}{2}\log(2\pi\sigma^2) - \frac{(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2}{2\sigma^2} \right) \tag{8}$$

The Residual Sum of Squares (RSS) is defined as in (Eq. 9) and one can see that minimizing $RSS(\boldsymbol{\beta})$ and maximizing $\ell(\boldsymbol{\beta})$ results in same estimations for parameters $\boldsymbol{\beta}$:

$$RSS(\boldsymbol{\beta}) = \sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \tag{9}$$

$$\text{argmin}_{\boldsymbol{\beta}} RSS(\boldsymbol{\beta}) = \text{argmax}_{\boldsymbol{\beta}}(\ell(\boldsymbol{\beta})) \tag{10}$$

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

$$L_{OLS}(\boldsymbol{\beta}) = RSS(\boldsymbol{\beta}) \tag{11}$$

$$\hat{\boldsymbol{\beta}} = argmin_{\boldsymbol{\beta}} RSS(\boldsymbol{\beta}) \tag{12}$$

3.1.3 Methods for Finding Weights

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

$$\mathbf{X} = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix} \tag{13}$$

$$\nabla_{\boldsymbol{\beta}} RSS(\boldsymbol{\beta}) = -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = 0 \tag{14}$$

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{15}$$

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

$$J(\hat{\boldsymbol{\beta}}) = \frac{1}{n} L_{OLS}(\hat{\boldsymbol{\beta}}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\hat{\boldsymbol{\beta}})^2 \tag{16}$$

$$\nabla J(\hat{\boldsymbol{\beta}}) = -\frac{2}{n}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) \tag{17}$$

$$\hat{\boldsymbol{\beta}}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \alpha\nabla J(\hat{\boldsymbol{\beta}}^{(t)}) \tag{18}$$

3.2 Ridge Regression

Ridge regression is a regularization technique that adds a penalty term to the linear regression cost function (Eq. 22), which helps mitigate overfitting by shrinking the coefficients towards zero. From a probabilistic perspective, ridge performs a maximum a posterior (MAP) estimation (see Section 3.2.1) instead of MLE; resulting in a regularized loss function (Eq. 22). Ridge regression is chosen due to its effectiveness in handling large feature spaces. Its regularization term helps mitigate overfitting. regression offers robustness in identifying the most influential features for predicting emotional valence.

3.2.1 Maximum a Posteriori Estimation

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

$$p(\boldsymbol{\beta}|D) \propto p(D|\boldsymbol{\beta})p(\boldsymbol{\beta}) \tag{19}$$

$$p(\boldsymbol{\beta}|D) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2}{2\sigma^2}\right) \cdot \frac{1}{\sqrt{2\pi\lambda^2}} \exp\left(-\frac{\boldsymbol{\beta}^T\boldsymbol{\beta}}{2\lambda^2}\right) \tag{20}$$

$$-\log(p(\boldsymbol{\beta}|D)) \propto \frac{1}{2\sigma^2}\left(\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 + \lambda\sum_{j=1}^{p}\beta_j^2\right) - nlog(\frac{1}{\sqrt{2\pi}\sigma}) - plog\left(\frac{\sqrt{\lambda}}{\sqrt{2\pi}\sigma}\right) \tag{21}$$

$$L_{ridge}(\boldsymbol{\beta}, \lambda) = \sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 + \lambda\sum_{j=1}^{p}\beta_j^2 = RSS(\boldsymbol{\beta}) + \lambda\|\boldsymbol{\beta}\|_2^2 \tag{22}$$

$$\hat{\boldsymbol{\beta}} = argmax_{\boldsymbol{\beta}} p(\boldsymbol{\beta}|D) = argmin_{\boldsymbol{\beta}} L_{ridge}(\boldsymbol{\beta}, \lambda) \tag{23}$$

3.2.2 Methods for Finding Weights

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

$$\hat{\boldsymbol{\beta}}_{\text{ridge}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{24}$$

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

$$J(\hat{\boldsymbol{\beta}}) = \frac{1}{n}L_{ridge}(\hat{\boldsymbol{\beta}}, \lambda) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\hat{\boldsymbol{\beta}})^2 + \lambda\|\hat{\boldsymbol{\beta}}\|_2^2 \tag{25}$$

$$\nabla J(\hat{\boldsymbol{\beta}}) = -\frac{2}{n}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) + 2\lambda\hat{\boldsymbol{\beta}} \tag{26}$$

$$\hat{\boldsymbol{\beta}}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \alpha\nabla J(\hat{\boldsymbol{\beta}}^{(t)}) \tag{27}$$

3.3 Lasso Regression

Lasso regression is another regularization technique similar to Ridge regression but uses a different penalty term (Eq. 31) 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. 31). Lasso regression is employed for its capability in feature selection and avoiding overfitting due to it's regularization term, particularly in datasets with a large number of features. This feature selection property not only reduces model complexity but also enhances interpretability by focusing on the most relevant features.

3.3.1 Maximum a Posteriori Estimation

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

$$p(\boldsymbol{\beta}|D) \propto p(D|\boldsymbol{\beta})p(\boldsymbol{\beta}) \tag{28}$$

$$p(\boldsymbol{\beta}|D) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2}{2\sigma^2}\right) \cdot \prod_{j=1}^{p} \frac{\lambda}{4\sigma^2} \exp\left(-\frac{\lambda}{2\sigma^2}|\beta_j|\right) \tag{29}$$

$$-\log(p(\boldsymbol{\beta}|D)) \propto \frac{1}{2\sigma^2}\left(\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 + \lambda\sum_{j=1}^{p}|\beta_j|\right) - nlog(\frac{1}{\sqrt{2\pi}\sigma}) - plog\left(\frac{\lambda}{4\sigma^2}\right) \tag{30}$$

$$L_{lasso}(\boldsymbol{\beta}, \lambda) = \sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\boldsymbol{\beta})^2 + \lambda\sum_{j=1}^{p}|\beta_j| = RSS(\boldsymbol{\beta}) + \lambda\sum_{j=1}^{p}|\beta_j| \tag{31}$$

$$\hat{\boldsymbol{\beta}} = argmax_{\boldsymbol{\beta}} p(\boldsymbol{\beta}|D) = argmin_{\boldsymbol{\beta}} L_{lasso}(\boldsymbol{\beta}, \lambda) \tag{32}$$

3.3.2 Gradient Descent

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

$$J(\hat{\boldsymbol{\beta}}) = \frac{1}{n}L_{lasso}(\hat{\boldsymbol{\beta}}, \lambda) = \frac{1}{n}\left(\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\hat{\boldsymbol{\beta}})^2 + \lambda\sum_{j=1}^{p}|\hat{\beta}_j|\right) \tag{33}$$

$$\nabla J(\hat{\boldsymbol{\beta}}) = -\frac{2}{n}\mathbf{X}^T(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) + \lambda \cdot \text{sign}(\hat{\boldsymbol{\beta}}) \tag{34}$$

$$\hat{\boldsymbol{\beta}}^{(t+1)} = \hat{\boldsymbol{\beta}}^{(t)} - \alpha\nabla J(\hat{\boldsymbol{\beta}}^{(t)}) \tag{35}$$

3.4 Neural Network (MLP) Regression

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

3.4.1 Forward Propagation

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

$$\mathbf{A}^{(0)} = \mathbf{X} \tag{36}$$

$$\mathbf{Z}^{(l)} = \mathbf{W}^{(l)}\mathbf{A}^{(l-1)} + \mathbf{b}^{(l)} \tag{37}$$

$$\mathbf{A}^{(l)} = g(\mathbf{Z}^{(l)}) \tag{38}$$

$$\hat{\mathbf{y}} = \mathbf{A}^{(L)} \tag{39}$$

3.4.2 Backpropagation

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

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \frac{1}{m}\mathbf{dZ}^{(l)}(\mathbf{A}^{(l-1)})^T \tag{40}$$

$$\frac{\partial J}{\partial \mathbf{b}^{(l)}} = \frac{1}{m}\sum_{i=1}^{m}\mathbf{dZ}^{(l)} \tag{41}$$

$$\mathbf{dZ}^{(l)} = (\mathbf{W}^{(l+1)})^T\mathbf{dZ}^{(l+1)} \odot g'(\mathbf{Z}^{(l)}) \tag{42}$$

3.4.3 Gradient Descent

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

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \alpha\frac{\partial J}{\partial \mathbf{W}^{(l)}} \tag{43}$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \alpha\frac{\partial J}{\partial \mathbf{b}^{(l)}} \tag{44}$$

3.5 K-Nearest Neighbors (KNN) Regression

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

3.5.1 Prediction

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

$$\hat{y}_{\text{new}} = \frac{1}{K}\sum_{i=1}^{K} y_{\text{nearest}_i} \tag{45}$$

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 ..., instanceAmount

$$d(\mathbf{x}_i, \mathbf{x}_{new}) = \|\mathbf{x}_i - \mathbf{x}_{new}\|_2 = \sqrt{\sum_{j=1}^{p}(x_{i,j} - x_{new,j})^2} \tag{46}$$

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 may struggle with capturing non-linear relationships and generalizing to new music styles. Ridge and Lasso regression face difficulties in selecting regularization parameters and interpreting sparse solutions, impacting their generalization. Neural networks offer flexibility in capturing complex patterns but may overfit on small datasets and lack interpretability. K-Nearest Neighbors regression encounters issues with high-dimensional feature spaces and local generalization. Moreover, all methods must contend with the inherent subjectivity of happiness labeling and the need for robust validation across diverse musical landscapes.

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 to mitigate the curse of dimensionality. Additionally, addressing biases and imbalances in the data, such as class distributions in happiness ratings and potential biases in feature representation, remains crucial for building reliable and generalizable models.

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

5 Performance Analysis

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

For each machine learning method utilized, suitable performance metrics will be employed to quantify predictive accuracy, such as mean squared error (MSE) or mean absolute error (MAE). MSE measures the average squared difference between predicted and actual values, providing insight into the model's overall predictive accuracy. MAE, on the other hand, measures the average absolute difference between predicted and actual values, offering a more interpretable metric that is less sensitive to outliers.

Additionally, techniques like cross-validation will 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.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \qquad (47)$$

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \qquad (48)$$

6 Simulation Setup & Preliminary 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 for neural network) plots, test performances (MSE metric used) and elapsed training times are reported. Due to large dataset, hyperparameters are not exhaustively searched by algorithms but best hyperparameters are chosen empirically by testing on the validation set, and so-far best performing ones are used for these preliminary 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. 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 also produced (Appendix B, Fig. 7) 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 of ridge does not suffer from invertibility as the closed form is guaranteed to be always invertible. The gradient descent method, successfully, produced a very similar result to the baseline normal equation solution. As expected ridge solution, has a worse training loss than linear regression but it generalized as good as linear regression (Appendix B, Fig. 8). Results depict that, better hyper-parameters could be searched for better generalization performance.

Lasso Regression: As expected and very similar to the ridge regression, Lasso has a worse training loss than linear regression, it generalized almost same with linear regression. Result implies that method needs more work in terms of finding better hyper-parameters (Appendix B, Fig. 9). Since there is no closed-form solution form of lasso, it is rather harder to compare with a baseline performance.

Neural Network: Neural Network performed best among all in terms of capturing the complex patterns and learning train dataset as much as possible. For this result a Full Batch NN with 2 layers with 40 and 1 neurons (Appendix B, Fig. 10) and Full Batch NN with 3 layers with 30 and 10 neurons (Appendix B, Fig. 11) are used and parameters initialized according to Xavier initialization in order to avoid exploding/vanishing gradients and local minimums (Random initialization resulted in local optima stuck models). Both models performed very similar and better than all other models. The training was not converged and if I had more computation power the loss would get better. The steady yet slow learning process and parallel train and test loss (Appendix B, Fig. 12) depicted the need of gradient methods (Momentum, Adam etc.) and techniques like PCA for a more efficient and fast learning process for the future of this project. Due to my insufficient computation power NN method could not fully shown how much better it could be, it has shown it has the highest capability of capturing complex relationships without even fully converging.

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 for the predictions. The performance was slightly worse compared to other methods yet it is still very surprising considering that the implementation complexity is low compared to other methods. After trying many different values for K, K = 10 gave the best result (Appendix B, Fig. 13).

To summarize, the preliminary results were satisfactory for my expectations yet for the future of process I plan to achieve better performance with neural network method by utilizing aforementioned techniques. As expected, it took much longer to train NN and KNN models compared to other linear methods. Other than that, results for linear methods depict they peaked their performance by approximating normal equation results by gradient descent, yet they may benefit from advanced pre-processing techniques as well as KNN regression.

7 APPENDIXES

7.1 Appendix A - Dataset

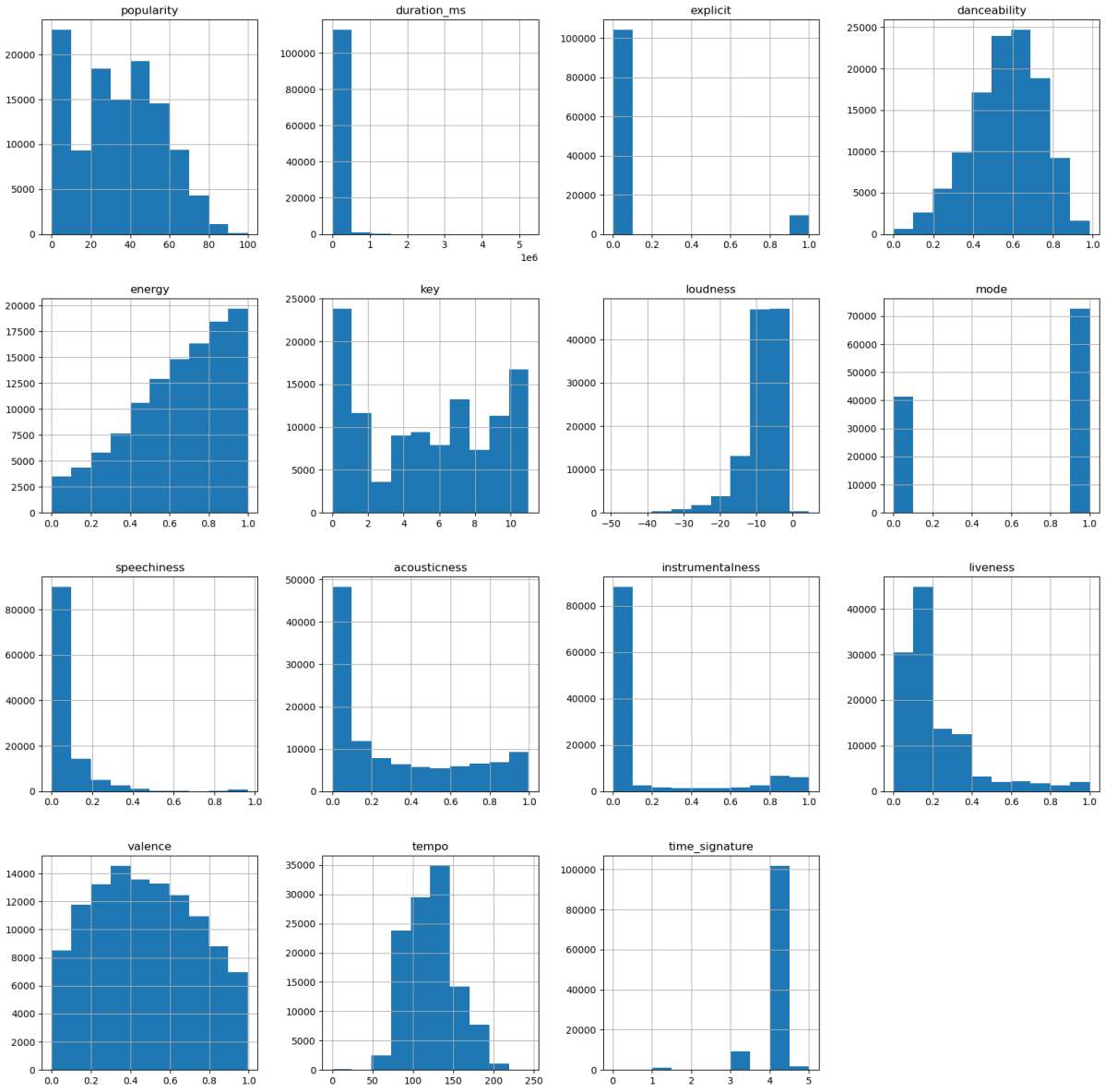

Figure 1: Histograms of features.

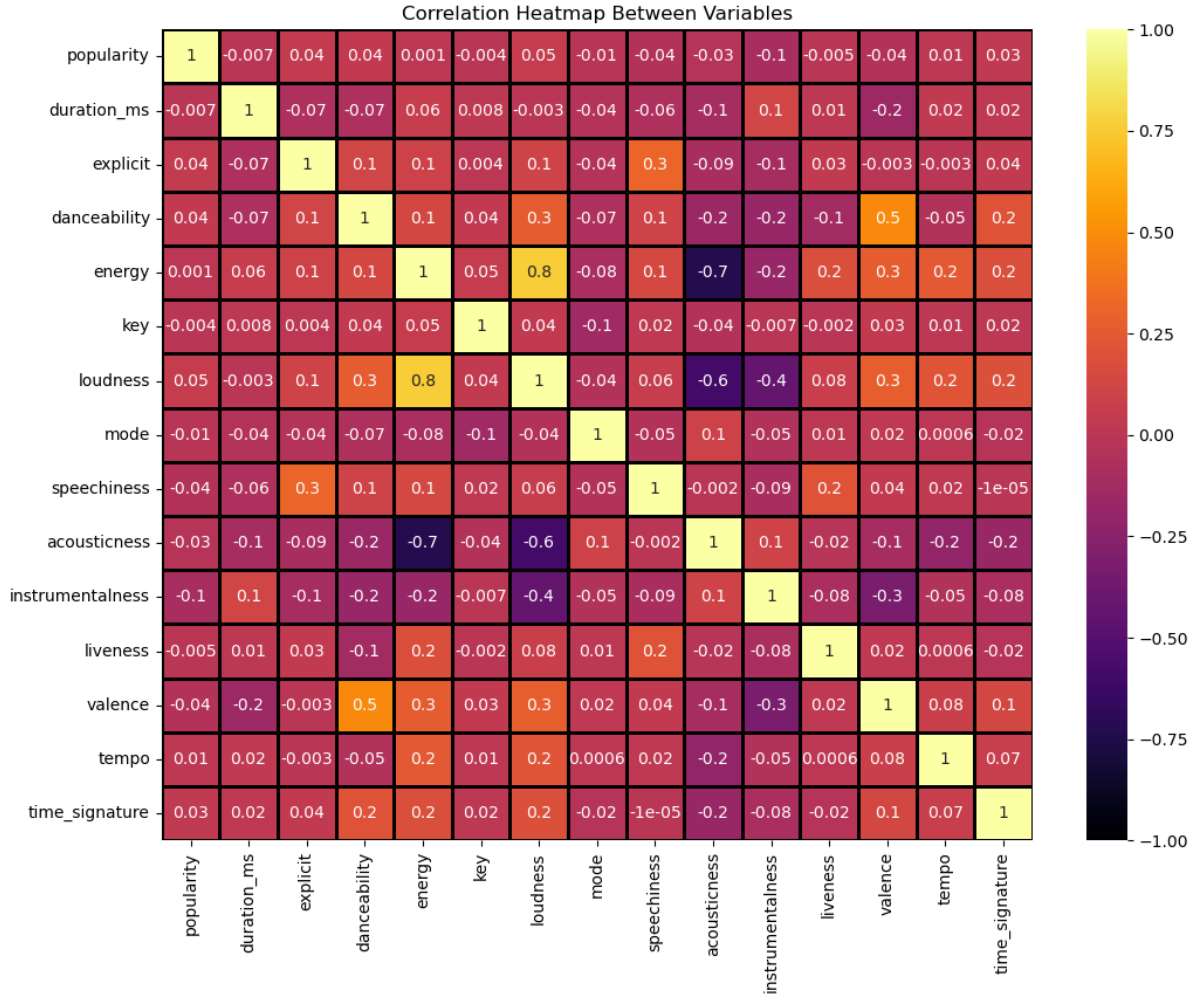

Figure 2: Correlation matrix as heatmap.

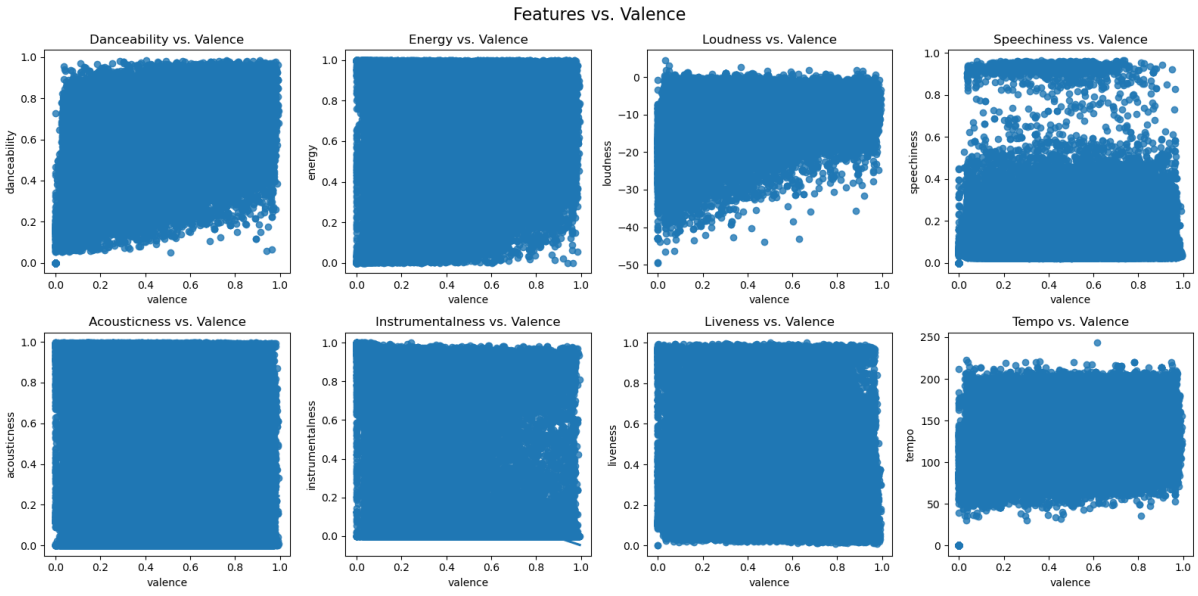

Figure 3: Correlation of key features.

	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.

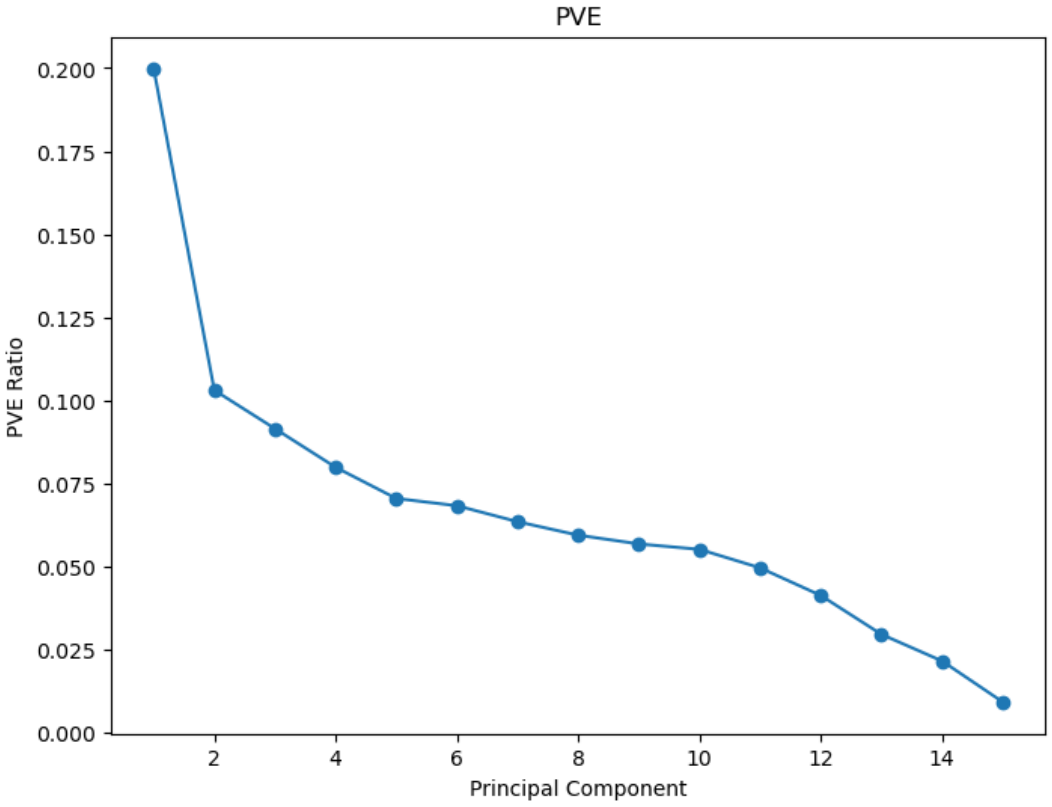

Figure 5: PVE vs PC Plot.

	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 6: Statistic of pre-processed dataset.png

7.2 Appendix B - Preliminary Results

Table 1: Performance Metrics for Different Machine Learning Methods

Method	**Train Loss**	**Test Loss**	**Iteration**	**Time(s)**	**Parameter**
Linear (Normal Equation)	2.1948	2.1942	-	0.08	-
Linear (Pseudo-inverse)	0.0419	0.0419	-	0.23	-
Linear (Gradient Descent)	0.0420	0.0421	20.000	36.88	$\alpha = 10^{-1}$
Ridge (Normal Equation)	0.0422	0.0419	-	0.074	$\lambda = 10^{-4}$
Ridge (Gradient Descent)	0.0423	0.0421	20.000	34.45	$\alpha = 10^{-1}$, $\lambda = 10^{-4}$
Lasso	0.0427	0.0421	20.000	60.5	$\alpha = 10^{-1}$, $\lambda = 10^{-4}$
NN (Full Batch (40, 1))	0.0411	0.0413	5.000 E	236.6	$\alpha = 10^{4}$
NN (Full Batch (30, 10, 1))	0.0409	0.0413	5.000 E	500.6	$\alpha = 10^{4}$
KNN	-	0.0438	-	256.9	$K = 10$

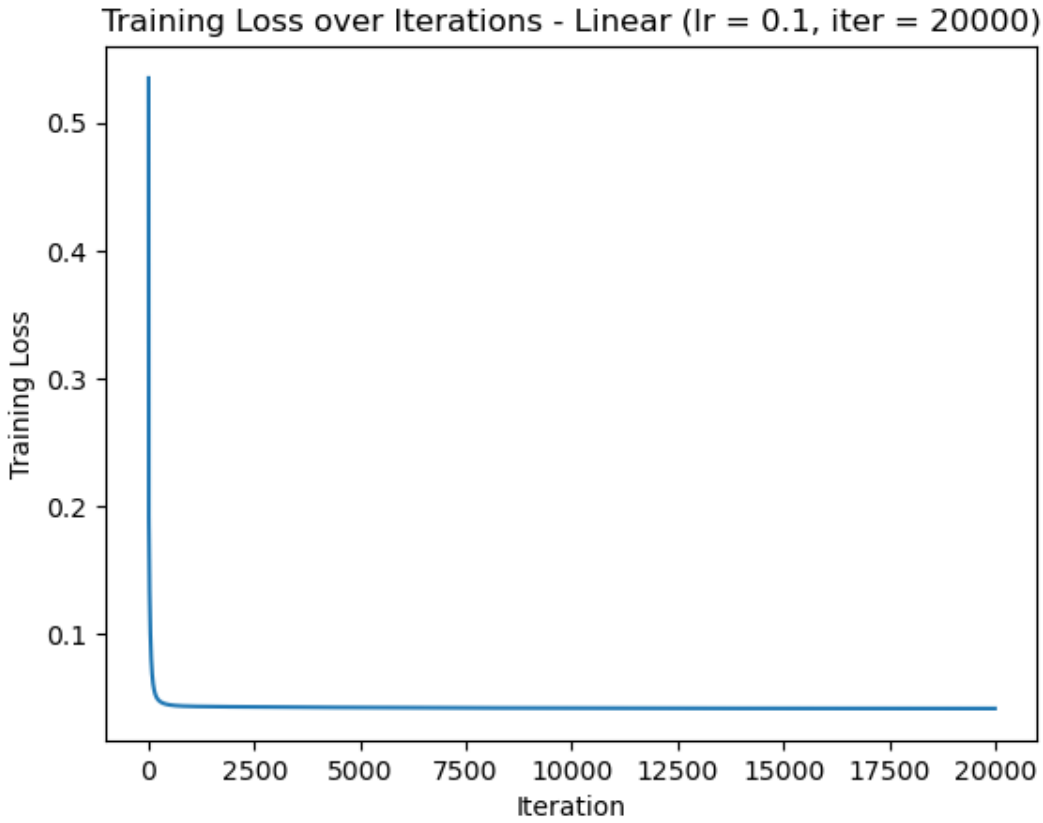

Figure 7: Train Loss vs Iterations (20.000) for Linear Regression with $\alpha = 10^{-1}$.

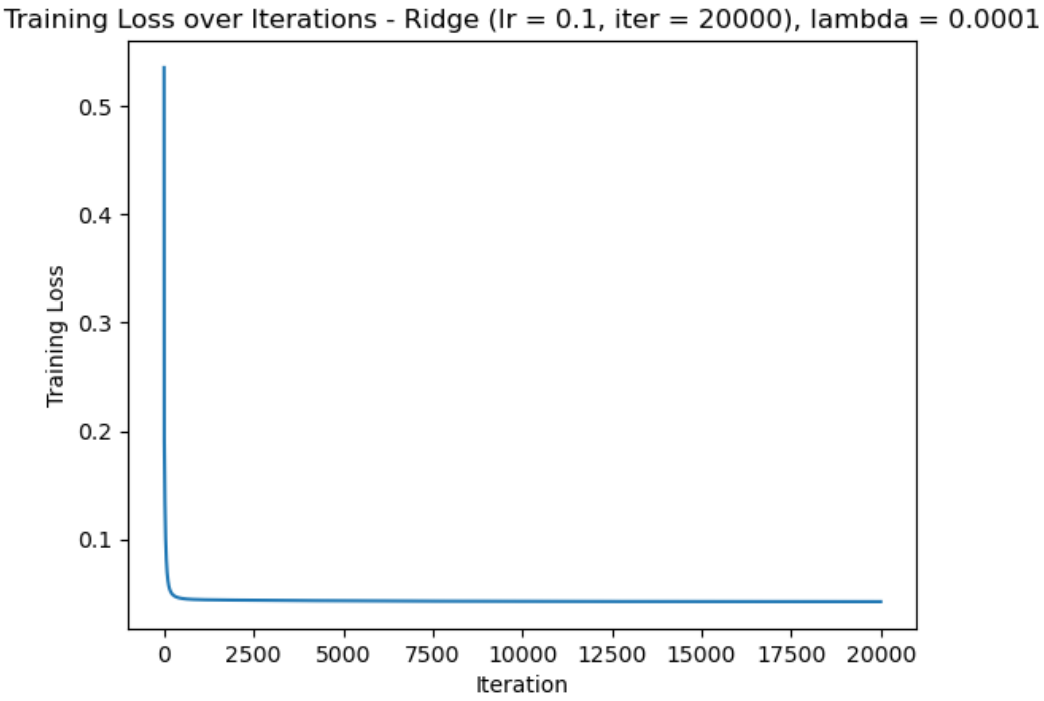

Figure 8: Train Loss vs Iterations (20.000) for Ridge Regression with $\alpha = 10^{-1}, \lambda = 10^{-3}$.

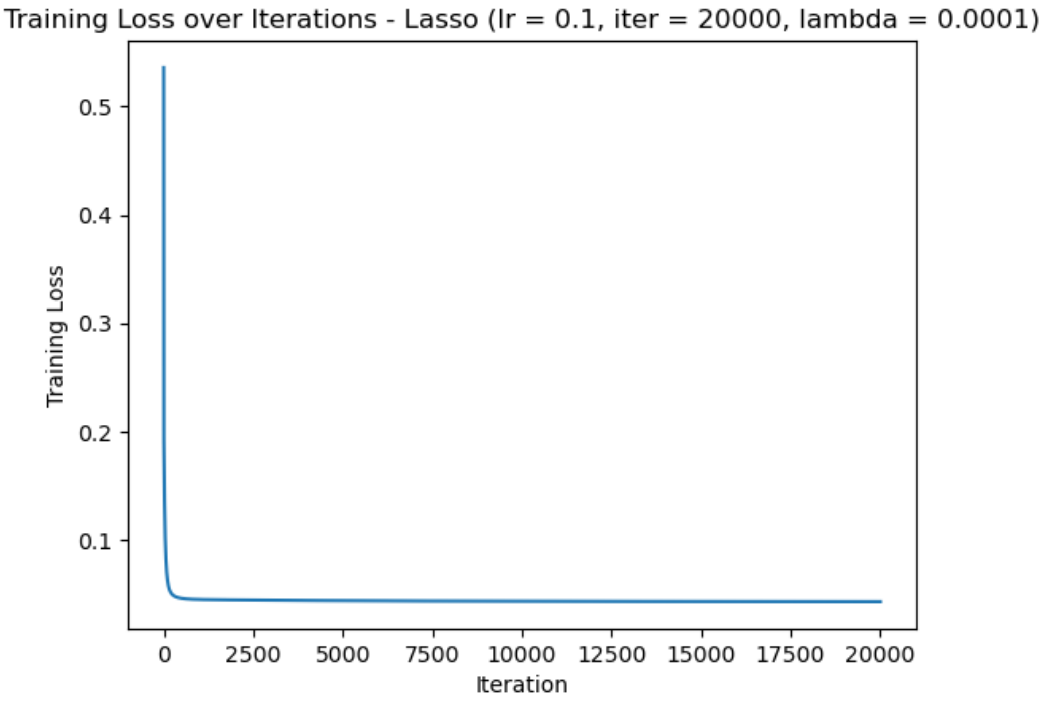

Figure 9: Train Loss vs Iterations (20.000) for Lasso Regression with $\alpha = 10^{-1}, \lambda = 10^{-4}$.

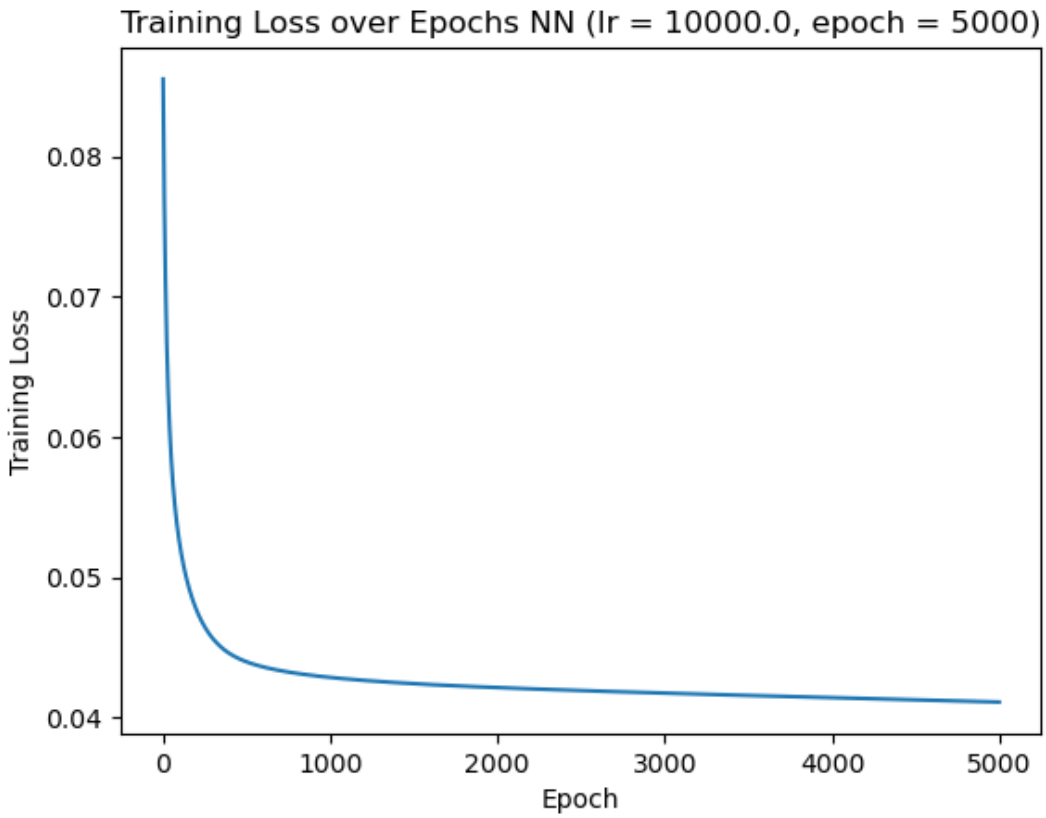

Figure 10: Train Loss vs Epochs (5.000) for Full-Batch 2 Layer (40, 1) Neural Network with $\alpha = 10^4$.

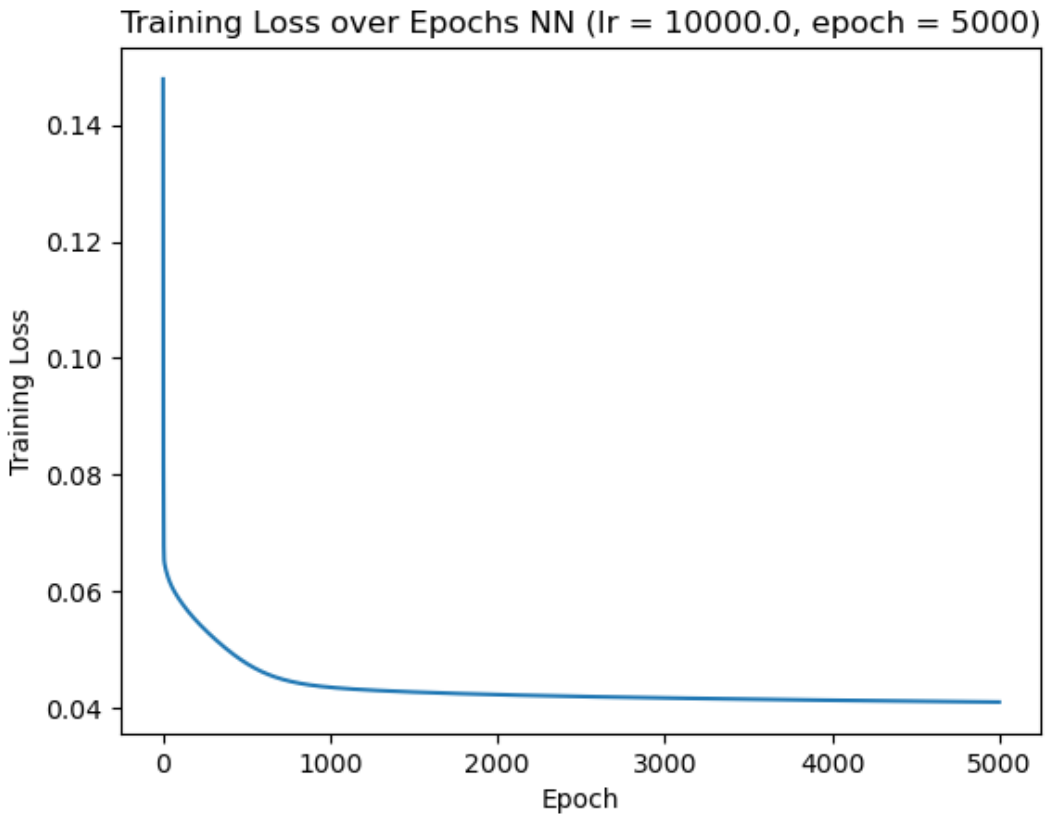

Figure 11: Train Loss vs Epochs (5.000) for Full-Batch 3 Layer (30, 10, 1) Neural Network, $\alpha = 10^4$

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

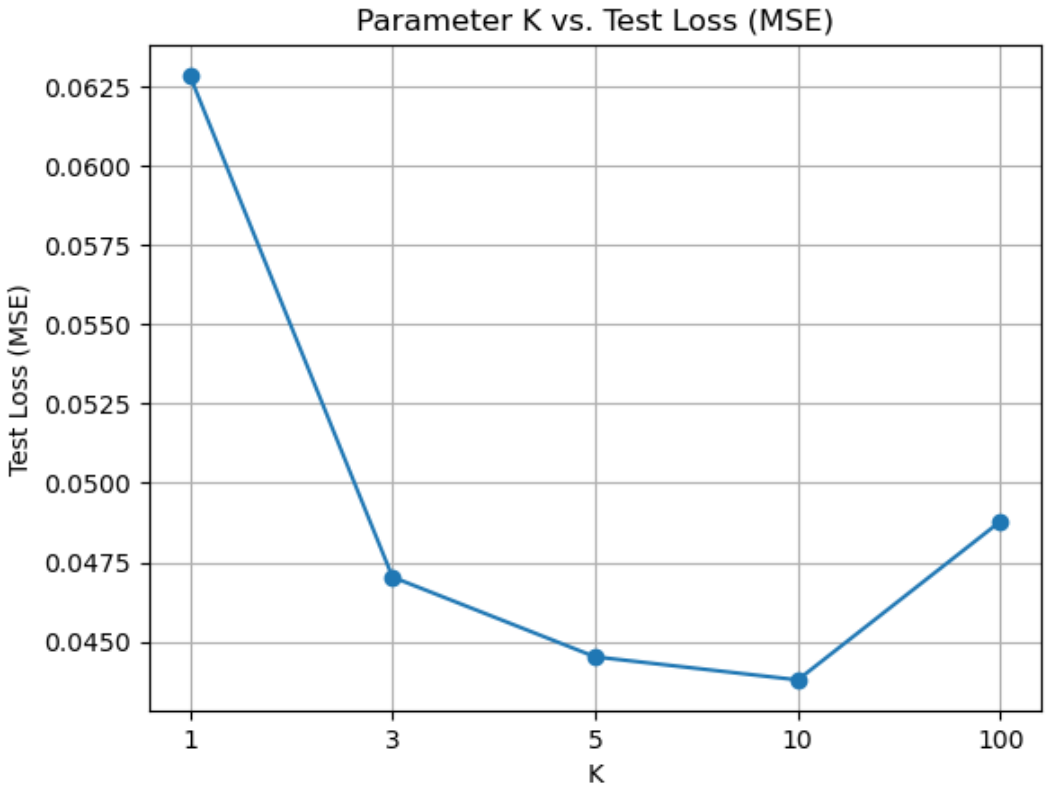

Figure 13: Hyperparameter K vs. Test Loss (MSE) for KNN Regression ($K \in \{1, 3, 5, 10, 100\}$)

7.3 Appendix C - Codes

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

dataPath = "./data/dataset.csv"
df = pd.read_csv(dataPath, index_col=0)
columns = list(df.columns)
columnsToKeep = columns[4: -1]
columnsToKeep
df = df[columnsToKeep]
# Bool to numerical data for explicit row
df.loc[:, 'explicit'] = df['explicit'].astype(int)
# One hot encoding for nominal categories
df = pd.get_dummies(df, columns=['key', 'time_signature'], dtype=int)

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

    # Remove features with zero range from normalization
    zero_range_features = feature_range[feature_range == 0].index
    valid_features = feature_range[feature_range != 0].index
    df_normalized = (df[valid_features] - min_vals[valid_features]) /
        feature_range[valid_features]
    if not zero_range_features.empty:
        df_normalized = pd.concat([df_normalized, df[zero_range_features]],
            axis=1)

    return df_normalized

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

responseFrame = df.pop('valence')
predictorFrame = df
# Min-Max scaling for predictor variables
df_normalized = min_max_scaling(predictorFrame)
# Standard scaling for predictor variables
df_standardized = standard_scaling(predictorFrame)
predictorFrame_scaled = df_normalized
```

Listing 1: Preprocessing Code for all Methods

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

trainSplit = 0.8
valSplit = 0.1
testSplit = 0.1

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

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

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

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

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

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

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

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

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

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

            # Print gradients for debugging
            loss = self.lossMSE(designMatrix, y)
            print(f'Train Loss at iteration {i}:', loss)
            self.loss_history.append((i, loss))

        return self
```

Listing 3: Linear Regression code (Part 1)

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

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

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

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

    return self

def svdMethod(self, designMatrix, y):
    self.weightVector, residuals, _, _ = np.linalg.lstsq(designMatrix, y,
        rcond=None)
    return self

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

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

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

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

Listing 4: Linear Regression code (Part 2)

```
import time

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

regressor.plot_loss_history()

predictions = regressor.inference(testPredictor)

# Validation Performance

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

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

# Measuring Performance

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

# Demo prediction

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

Listing 5: Validation and Test Performance of Linear Regression code

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

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

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

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

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

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

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

        return self

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

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

Listing 6: Ridge Regression code (Part 1)

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

Listing 7: Ridge Regression code (Part 2)

```
import time

regressor = RidgeRegression(lr = 1e-1, lambdaConstant=1e-3, n_iters= 20000)
start = time.time()
regressor.fit(trainPredictor, trainResponse)
end = time.time()
print(f'Time elapsed: {end - start:.4}')

regressor.plot_loss_history()

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

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

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

# Demo Prediction
demoInstanceLoc = 11401
demoPredictor = predictorData[demoInstanceLoc]
demoResponse = responseData[demoInstanceLoc]
demoPredictor = np.expand_dims(demoPredictor, axis=0)
print(demoPredictor.T.shape)
demoPrediction = regressor.inference(demoPredictor)
print('Prediction:', demoPrediction, 'Response:', demoResponse)
```

Listing 8: Validation and Test Performance of Ridge Regression code

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

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

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

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

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

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

        return self

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

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

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

Listing 9: Lasso Regression code (Part 1)

```
47      def lossMSE(self, designMatrix, y):
48          predictions = self.predict(designMatrix)
49          error = y - predictions
50          squaredError = np.dot(error.T, error)
51          meanSquaredError = 1/(y.size) * squaredError
52          return meanSquaredError
53
54      def lossLassoMSE(self, designMatrix, y):
55          mse = self.lossMSE(designMatrix, y)
56          lasso_mse = mse + self.lambdaConstant * np.sum(np.abs(self.
                weightVector))
57          return lasso_mse
58
59      def plot_loss_history(self):
60          iterations, losses = zip(*self.loss_history)
61          plt.plot(iterations, losses)
62          plt.xlabel('Iteration')
63          plt.ylabel('Training Loss')
64          plt.title(f'Training Loss over Iterations - Lasso (lr = {self.lr},
                iter = {self.n_iters}, lambda = {self.lambdaConstant})')
65          plt.show()
```

Listing 10: Lasso Regression code (Part 2)

```
import time

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

start = time.time()
regressor.fit(trainPredictor, trainResponse)
end = time.time()
print(f'Time elapsed: {end - start:.4}')

regressor.plot_loss_history()

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

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

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

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

Listing 11: Validation and Test Performance of Lasso Regression code

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

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

def linear(z):
    return z

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

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

def linearDer(z):
    return 1

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

Listing 12: Neural Network Activation Functions code

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

    def initWeights(self):
        # Random initialization failed. Xavier initialization for weights
        self.W = np.random.randn(self.numNeurons, self.inputNumNeuron) * np
            .sqrt(1 / self.inputNumNeuron)
        self.b = np.zeros((self.numNeurons, 1))

    def updateForwardState(self, inputToLayer):
        inducedLocal = np.matmul(self.W, inputToLayer) + self.b
        output = self.activation(inducedLocal)
        self.Z_state = inducedLocal
        self.A_state = output
        return output

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

    def updateDeltaState(self, dA):
        derActivation = self.activationDerivative(self.Z_state)
        dZ = np.multiply(dA, derActivation)
        self.dZ_state = dZ

    def calculateChange(self, A_input):
        self.dW_state = (1 / self.batchSize) * np.dot(self.dZ_state,
            A_input.T)
        self.db_state = (1 / self.batchSize) * np.sum(self.dZ_state, axis
            =1, keepdims=True)

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

    def __repr__(self):
        return "Input Dim: " + str(self.inputNumNeuron) + ", Number of
            Neurons: " + str(self.numNeurons) + "\n Activation: " + self.
            activationName
```

Listing 13: Neural Network Layer Class code

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

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

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

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

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

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

Listing 14: Neural Network code (Part 1)

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

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

            nextLayer = layer

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

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

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

        print('Final Train Loss:', trainLoss)
```

Listing 15: Neural Network code (Part 2)

```
78      def testLoss(self, test_x, test_y):
79          predictions = np.squeeze(self.predict(test_x))
80          lossMSE = self.loss(predictions, test_y)
81          return lossMSE
82
83      def plot_loss_history(self):
84          iterations, losses = zip(*self.loss_history)
85          plt.plot(iterations, losses)
86          plt.xlabel('Epoch')
87          plt.ylabel('Training Loss')
88          plt.title(f'Training Loss over Epochs NN (lr = {1e4}, epoch =
                {5000})')
89          plt.show()
```

Listing 16: Neural Network code (Part 3)

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

# Full batch gradient descent
batchSize = predictorData.shape[0]
trainSplit = 0.8
valSplit = 0.1
testSplit = 0.1

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

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

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

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

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

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

    return np.array(mini_batches)

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

Listing 17: Neural Network Batch and Data Splitting code

```
import time

# Initialize NeuralNetwork
nn = NeuralNetwork()
nn.addLayer(Layer(mini_batches_X[0].shape[0], 40, 'relu', batchSize))
nn.addLayer(Layer(40, 1, 'linear', batchSize))

start = time.time()
nn.fit(mini_batches_X, mini_batches_Y, lr=1e4, epochAmount=5000)
end = time.time()

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

nn.plot_loss_history()

# Validation Performance
valError = np.squeeze(nn.testLoss(miniValX.T, miniValY))
print('Validation Loss:', valError)

# Test Performance
testError = np.squeeze(nn.testLoss(miniTestX.T, miniTestY))
print('Test Loss:', testError)

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

Listing 18: Validation and Test Performance of Neural Network code

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

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

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

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

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

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

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

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

            predictions.extend(batch_predictions)

        return np.array(predictions)
```

Listing 19: KNN Regression code

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

# Validation Performance
MSEs = []
K = [1, 3, 5, 10, 100]
for k in K:
    knn = KNNRegression(k=k)
    knn.fit(trainPredictor, trainResponse)
    predictions = knn.predict(valPredictor, batch_size=200)
    MSE = mse(valResponse, predictions)
    print("Mean Squared Error (MSE):", MSE)
    MSEs.append(MSE)

x_values = np.arange(len(K))

# Plot K versus MSE
plt.plot(x_values, MSEs, linestyle='-', marker='o')
plt.title('Parameter K vs. Val Loss (MSE)')
plt.xlabel('K')
plt.ylabel('Val Loss (MSE)')

# Set the x-ticks to be the original K values
plt.xticks(x_values, K)

plt.grid(True)
plt.show()

# Test Performance
knn = KNNRegression(k=5)
knn.fit(trainPredictor, trainResponse)
predictions = knn.predict(testPredictor, batch_size=456)
MSE = mse(testResponse, predictions)
print("Test Performance:", MSE)

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

Listing 20: Validation and Test Performance of KNN Regression code

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

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

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

# Null Value Check
pd.isnull(df).sum()

# PCA & PVE Analysis

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

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

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

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

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

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

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

Listing 21: Data Analysis code (Part 1)

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

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

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

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

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

# Plot feature correlations
features = ["danceability", "energy", "loudness", "speechiness", "
    acousticness", "instrumentalness", "liveness", "tempo"]

fig, axes = plt.subplots(nrows=2, ncols=4, figsize=(16, 8))
fig.suptitle("Features vs. Valence", fontsize=16)

axes = axes.flatten()

for i, feature in enumerate(features):
    sns.regplot(data=df, y=feature, x="valence", ax=axes[i])
    axes[i].set_title(f"{feature.capitalize()} vs. Valence")

plt.tight_layout()
plt.show()

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

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

Listing 22: Data Analysis code (Part 2)

8 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].