

EEE485 – Term Project Phase 3 Report

Group No: 01

Submitted to: Cem Tekin <cemtekin[at]ee.bilkent.edu.tr>

Members: Ayhan Okuyan <ayhan.okuyan[at]ug.bilkent.edu.tr>

Emre Dönmez <emre.donmez[at]ug.bilkent.edu.tr>

Project Name: Music Genre Classification using Spotify API Metrics

Table of Contents

Introduction.....	2
Dataset Description and Preprocessing.....	2
Methods and Algorithms.....	3
Multilayered Perceptron (MLP).....	3
PCA.....	9
Random Forest.....	11
k-Nearest Neighbors (kNN).....	12
References.....	14
Appendix 1 - MLP.....	15
Appendix 1-A – SGD Optimization.....	15
Appendix 1-B – SGDM.....	21
Appendix 1-C – Adam.....	26
Appendix 1-D – AMSGrad.....	31
Appendix 1-E – SGD with PCA.....	37
Appendix 1-F – SGDM with PCA.....	42
Appendix 1-G – Adam with PCA.....	48
Appendix 1-H – AMSGrad with PCA.....	54
Appendix 2 – Random Forest.....	61
Appendix 3 – kNN.....	66
Appendix 4 – Data Analysis.....	71

Introduction

This term project aims to build a “Music Genre Classifier” using various machine learning and deep learning techniques and structures. We have worked on the “Spotify Tracks DB” database which includes some metrics created by Spotify API for their own recommender systems. In the data, some of the features were discrete, while some of them were continuous and that possessed a challenge in implementation. As described in the Phase 2 Report, we have implemented a Multilayered Perceptron neural network structure, a proposed hybrid k-Nearest Neighbors algorithm and a Random Forest Classifier. Furthermore, we have applied PCA as a feature extraction technique in MLP in order to observe if the extracted feature set would enhance the performance of the algorithm.

Dataset Description and Preprocessing

In order to make this project viable, we needed to find a dataset that has enough samples filled with different genre songs which have many distinct features so that it is possible to apply various machine learning algorithms on them in order to decide their genre. After searching and eliminating many possible datasets for our project, we have decided on the Spotify Tracks DB database [1] which we have stated in our proposal. The dataset has around 232 thousand songs which contain 26 different genres and 16 distinct features to learn with.

The reasons for choosing this dataset are the following: It has enough samples for training, validation, and testing; it has many features to choose from and train; it is suitable for the algorithms that we are going to use; but most importantly since the features presented in the dataset such as danceability or instrumentality are very abstract, using these kind of features in order to decide on the genre of a song would be a more unbiased decision and as a result of this, we think that it would certainly be more a more unbiased and non-human classification.

In the dataset, we see that there are both discrete and continuous features and their scales differ heavily. The most important part of this data are the features, “danceability”, “instrumentalness”, “liveness”, “energy”, “valence” and “acousticness” since these are features that are extracted by Spotify for their own recommender systems.

For preprocessing the data, we first looked through every genre and its corresponding number of samples. The histogram for this distribution was as given below.

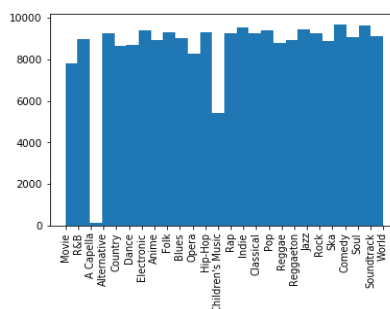


Figure 1 – Histogram for the Distribution of Classes in the Dataset

We have realized that we have two instances of Children’s Music which only differs with the apostrophe of Children’s. At first, we combined these two genres but then realized that one of the instances of Children’s Music does not have any instance of Children’s Music songs but instead, have instances of rock, metal and alternative rock songs. We decided to take this out of the dataset entirely since with only the correct Children’s music songs, the dataset is very small compared to other genres and training with such low number of songs would cause a bias towards not classifying Children’s Music. This is considered as a labelling mistake that the data creators have done. We also did not merge the wrong Children’s Music instances with the Alternative genre since doing this would make the Alternative genre class too large compared to other genres which approximately have 8500-10000 samples. The only exception to this range were the genres of Movie and Acapella. These genres also have lower samples than 8000 so we also took them out of the dataset. After these alterations, we are left with 23 genres to do classification with which is adequate for this project.

After doing these modifications, we have down sampled our data so that every genre has an equal of 8280 samples which is equal to the smallest sample sized genre among all the dataset which is Opera. We have done this by randomly selecting indices from other genres which have more samples than 8280 and removing them from the dataset. We have created dictionaries to transforms the labels to integer values. We have given them numbers from 0 to 22 so that every number corresponds to a single genre. From this point on, we need to preprocess data according to the algorithm to be used. Some of the features will be discarded for different algorithms.

MLP: All discrete features are discarded since we would need to create embeddings to classify categorical data [2] and we were not able to recover a corpus that is suited for the task. The features are, TrackID, Artist Name, Track Name, Time Signature, Key, and Mode will be discarded. In MLP, discrete data has no use to us, that is why we are only working with continuous features. After extracting unnecessary features, we have normalized the remaining features according to their nominal ranges so that the algorithm does not become biased to a single feature and the neurons in the architecture don't saturate.

kNN: TrackID, Track name, and Time Signature are discarded. Furthermore, we have used the same normalized features that we have used in MLP since kNN has no bias correction, the algorithm is open to be biased towards a set of features and discard the others.

Random Forest: Track Name and Artist Name are discarded. We have one hot encoded the categorical features. Other than that, no preprocessing was needed since the trees split according to a single feature. There is no need for normalization. We also did not one hot encode the labels in Random Forest because splitting according to information gain looks at the total number of a class in a node hence do not need encoding of labels.

Methods and Algorithms

Multilayered Perceptron (MLP)

The first algorithm that we have implemented is the Multilayered Perceptron (MLP) architecture, also known as the Feed-Forward Neural Network (FFNN). The architecture consists of layers, which have varying number of neurons in each one of them. Neuron is the basic element of a neural network. The network as a structure is based on the neural structure inside the brain and the neurons are modeled to represent a neuron.

Each neuron has a weight vector, a bias and a defined activation function. The weight vector represents the weights of the input connections to that neuron, and together with the bias, are the parts that are trained in the network. The neuron structure modelled by McCullough and Pitts is as given below.

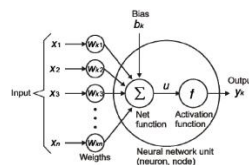


Figure 2 – McCullough-Pitts Neuron [3]

Hence, the output of the neuron j is modelled as the activation of the weighted sum of inputs and weights.

$$\hat{y} = \varphi(w_j^T x + \beta_j)$$

The upper layer structure of MLP, after the neuron is a layer, which is a stacked neuron structure. With the help of matrices, we can define the output as follows.

$$\hat{Y} = \varphi(WX)$$

Here, we concatenate the input matrix vertically with a row of ones and the weights with the bias vector horizontally to obtain the matrices X and W respectively. Hence, the forward connections of the fully connected networks constitute the architecture of the MLP as illustrated below.

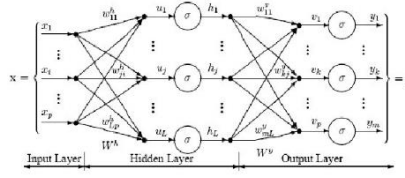


Figure 3 – An exemplary MLP Structure [4]

Mathematically, this is a directed graph structure. The output of the graph becomes the nested activations of the input matrix. Usually, all the neurons in a layer are given the same activation function for convenience. In our implementation, we have adopted four different activation functions: ReLU, Softmax, sigmoid and tanh. These are embedded in the Layer class that we have created as can be seen through Appendix A1-A4. The mathematical formulae for the activations are given below.

$$\varphi_{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad \varphi_{softmax,i}(x) = \frac{e^{x_i}}{\sum_k e^{x_k}}$$

$$\varphi_{sigmoid}(x) = \sigma(x) = \frac{e^x}{1 + e^x} \quad \varphi_{tanh}(x) = \tanh(x) = 2\sigma(2x) - 1$$

After constructing the Layer structure, we have moved on to the MLP structure that uses a list to keep track of the constructed layers. In an MLP architecture, the weights and biases are updated through backpropagation process. In the backpropagation, each layer's error and delta derivative is calculated by using the chain rule and we backpropagate according to a loss function which is a function that describes the distance of the prediction of the algorithm to the true labels. In Appendix A1-4, it is observable that we have implemented two loss functions. One is the Mean Squared Error and the other one is the Cross-Entropy Loss (to be used together with a Softmax output layer), which are as follows.

$$L_{MSE}(y_i, \hat{y}_i) = \sum_{c=1}^N (y_{i,c} - \hat{y}_{i,c})^2 \quad L_{CE}(y_i, \hat{y}_i) = \sum_{c=1}^N I(y_{i,c} = 1) \log(\hat{y}_{i,c})$$

The MSE measures the overall Euclidean distance of the prediction to the real values, which is used in both classification and algorithm tasks. However, the loss that is used more often is the Cross-Entropy loss which describes the difference between two probability distribution. It is adopted from the Information Theory and since the output of the Softmax layer compose a discrete probability distribution, we use Cross-Entropy to measure the dissimilarity between the two distributions. Softmax with Cross-Entropy was the tuple that we have worked with in the implementation of the output layer since the outputs of the Softmax activation represent the probability of the sample being of that class. For these two losses, the output layer's delta, and error are as given as follows.

$$Loss_{MSE} = \frac{1}{2} (Y - \hat{Y})^2 \Rightarrow e_{output} = \frac{\partial E}{\partial \hat{Y}} = Y - \hat{Y}$$

$$\delta_{output} = \frac{\partial \varphi(V_{output}(X))}{\partial X} \odot \frac{\partial E}{\partial Y} = \frac{\partial \varphi(V_{output}(X))}{\partial X} \odot e_{output}$$

Where V denotes the induced local fields of the neurons, before the activation. In the Cross-Entropy loss, when Softmax activation is used, that comes with a computational easiness since,

$$\delta_{output} = \hat{Y} - Y$$

Then, the errors and deltas for the hidden layers are as follows.

$$e_{hidden,t} = W_{t+1} \delta_{t+1}$$

$$\delta_{hidden,t} = \varphi'(v_{hidden,t}) \odot e_{hidden,t}$$

where, t represents the index of the layer in a sequential manner in the MLP. Hence, after all the error and deltas are found, the updates are calculated as follows in a Gradient Descent setting. Here, we see that we are ought to compute the derivatives of the activations of induced local fields with respect to the local fields. Below, we can observe that the derivatives of the activations that we have chosen yield

to the derivative forms that can be written in terms of their own functions. This fact is an important factor though memory consumption since we were than only required to keep the activations and not the induced local fields in the memory.

$$\phi'_{ReLU}(v) = \begin{cases} 0 & \text{if } v < 0 \\ 1 & \text{if } v \geq 0 \end{cases} \Rightarrow \begin{cases} 0 & \text{if } y < 0 \\ 1 & \text{if } y \geq 0 \end{cases}$$

$$\phi'_{sigmoid}(v) = \sigma(v)(1 - \sigma(v)) \Rightarrow y(1 - y)$$

$$\phi'_{tanh}(v) = 1 - \tanh^2(v) \Rightarrow 1 - y^2$$

We have used the stochastic mini batch updating as stated in the Phase 1 report. In the report, we have mentioned that we will be using the SGDM (Stochastic Gradient Descent with Momentum) as optimizer and we would implement adaptive learning algorithms if time allowed. For this phase, we have been able to implement three other optimizers: SGD (Stochastic Gradient Descent), Adam (Adaptive Momentum) and AMSGrad (An Extension of ADAM). The reason we have implemented these optimization techniques is to observe and comment on their performances, and comment on the overall performance of the algorithm itself under various optimizers. For all optimizers, we have used the same three-way split.

Initially, we have worked with SGD to find a valid architecture that would be able to capture the complex patterns in the data in an incremental way, meaning we have observed the learning curves, and added/subtracted neurons and layers accordingly. Furthermore, apart from the Phase 2, we have standardized the data to improve the performance of the training. Over the course of optimization, we have worked on ten different structures. The structures that we have worked on and their validation accuracies are given below. The output layer of all the algorithms consists of 23 neurons, equal to the number of classes and the output activation is Softmax.

<i>Iteration</i>	Hidden Layers	# Neurons in Hidden	Activation	Initialization	Loss Function	Validation Accuracy
1	1	[300]	Sigmoid	Xavier	CE	36.40%
2	2	[300,100]	Sigmoid	Xavier	CE	33.50%
3	2	[300,100]	ReLU	Xavier	CE	42.36%
4	2	[300,100]	ReLU	Kaiming	CE	43.75%
5	2	[400,200]	ReLU	Kaiming	CE	41.18%
6	2	[400,200]	ReLU	Kaiming	MSE	40.80%
7	2	[600,200]	ReLU	Kaiming	CE	43.03%
8	3	[600,400,200]	ReLU	Kaiming	CE	43.84%
9	4	[750,600,400,200]	ReLU	Kaiming	CE	43.94%
b	4	[500,300,150,65]	ReLU	Kaiming	CE	43.71%
10	6	[750,650,450,300,150,65]	ReLU	Kaiming	CE	43.09%

Table 1 – MLP Iterations and Their Validation Accuracies

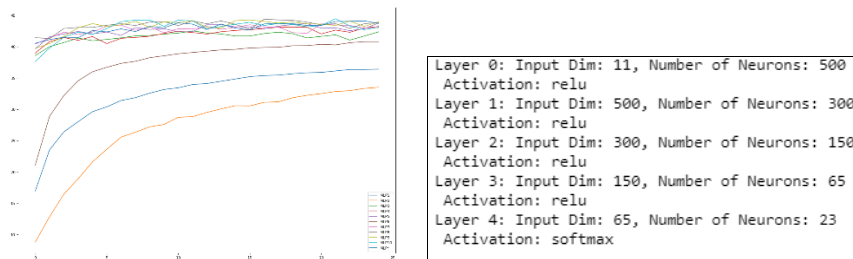


Figure 4-5 – Validation Accuracies of Architectures with respect to Epoch and the Selected MLP Architecture

The best architecture is chosen on two criteria, one is the validation accuracy and the other one is the number of neurons used. Here we observe that the accuracy doesn't go above 44% after four hidden layers, hence we have tried to optimize the network as much as possible around four hidden layers.

For weight initializations, we have used two different initializations. For the networks with ReLU activations, we have used the Kaiming Initialization [5] by He et.al. which is given as follows.

$$w_{l,i} \in \mathcal{W}_\ell, \quad w_{l,i} \sim \mathcal{N}(0, \sqrt{2/fan_{out}}), \quad \forall i = 1, \dots, n_l$$

Where fan_out is the number of units in that layer, and for the networks with softmax activations, we have used the Xavier initialization [6], given below.

$$w_{l,i} \in \mathcal{W}_\ell, \quad w_{l,i} \sim \mathcal{N}\left(0, \sqrt{6/(fan_{in} + fan_{out})}\right), \quad \forall i = 1, \dots, n_l$$

Where fan_in is the number of input connections to the layer. Then we have moved towards using the mentioned optimizers whose update rules are given as follows.

$$\text{SGD: } W_{E,t+1} = W_{E,t} + \eta \delta_t \tilde{X}_{layer,t}$$

$$\text{SGDM: } \Delta W_{E,t} = \eta \delta_t \tilde{X}_{layer,t} + \alpha \Delta W_{E,t-1}$$

$$W_{E,t+1} = W_{E,t} + \Delta W_{E,t}$$

Here, we see that a momentum parameter is added to the standard SGD algorithm. In SGD, the updates move perpendicular to each other, however, the use of momentum forwards the direction of the algorithm more towards the minimum. The comparison between the two is as illustrated below.

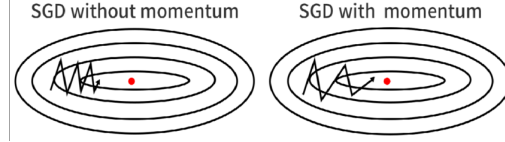


Figure 6 – SGD vs SGDM [5]

In this case, we expect a faster and more reliable convergence from SGDM. The other optimizer is Adam. Proposed by Kingma and Ba [7], Adam uses adaptive estimations of lower order moments of the gradients and it is computationally efficient. The reason we have used this optimizer was to observe the convergence of the model since Adam usually converges better than SGD variant optimizers. The algorithm is as follows.

$$m_t = (1 - \beta_1)(\delta_t \tilde{X}_{layer,t}) + \beta_1 m_{t-1}$$

$$v_t = (1 - \beta_2)(\delta_t \tilde{X}_{layer,t})^2 + \beta_2 v_{t-1}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\Delta W_{E,t+1} = W_{E,t} - \frac{\eta \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Here, the Beta decays, epsilon and learning rate are parameters. Epsilon is a small constant to keep the algorithm from division by zero. While we fine-tuned the learning rate, we have adopted the betas and epsilon from the article.

$$(\beta_1, \beta_2, \epsilon) = (0.9, 0.999, 10^{-8})$$

The algorithm AMSGrad, proposed by Reddi et.al in ICLR 2018 [8], is a modification to the original Adam algorithm that sometimes cannot converge to the optimal solution, the algorithm is as explained below.

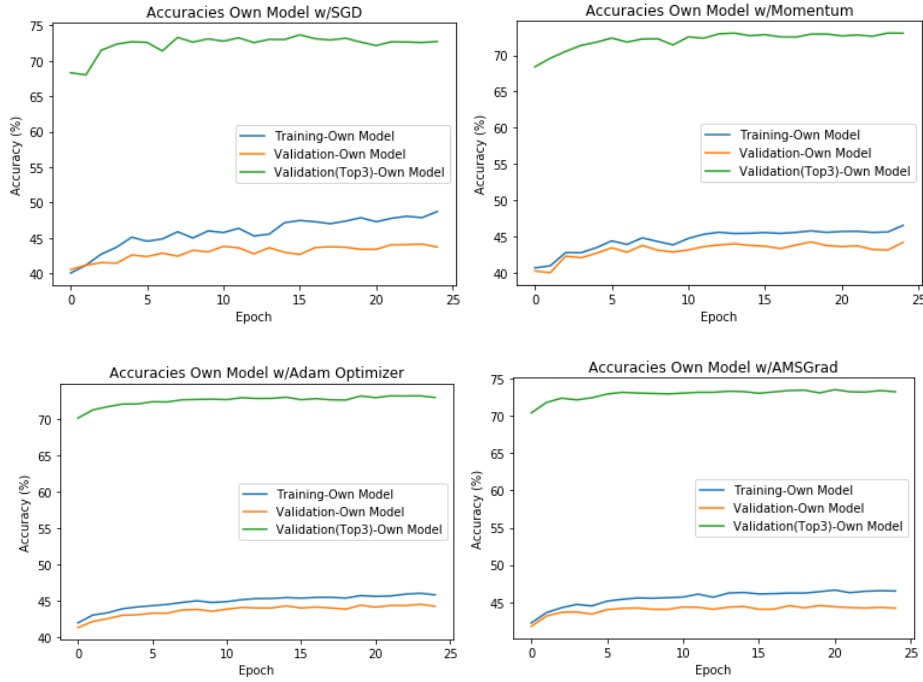
$$m_t = (1 - \beta_1)(\delta_t \tilde{X}_{layer,t}) + \beta_1 m_{t-1}$$

$$v_t = (1 - \beta_2)(\delta_t \tilde{X}_{layer,t})^2 + \beta_2 v_{t-1}$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

$$AW_{E,t+1} = W_{E,t} - \frac{\eta \cdot m_t}{\sqrt{\hat{v}_t} + \epsilon}$$

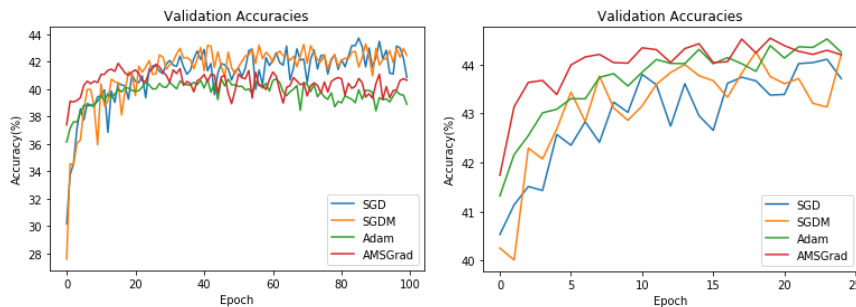
The parameters are the same as in Adam. After implementing all these algorithms that are explained above, we have run the best network on each of the with batch size of 529 and 100 epochs. In total, we have implemented four neural network training processes and each one in 100 epochs nearly consumed the same time around 1000 seconds (~17 mins), 1 epoch lasting around 10 seconds. Then, we observed that the algorithm started to overfit around 25 epochs, hence we have retrained the networks accordingly. Here are the learning curves with respect to each of the different optimizations, with their training, validation and Top-3 validation accuracies.



Figures 7-10 – Learning Curves for the Algorithm with SGD, SGDM, Adam and AMSGrad from left to right and up to down.

Here, we observe that the algorithm that we have built reaches overall 44% accuracy in the validation dataset, also top three accuracy moves well with the learning. In the SGD, we see that the learning is fast in the beginning, since we have chosen a learning rate of 0.1 for the SGD and SGDM, however, moves irregularly since SGD oscillates when reaching a minimum. We also see a similar curve with the SGDM, however, there is less oscillations (Momentum Coefficient is chosen as 0.9).

Here, we present one other reason that we have implemented the standardization. Below, is the validation accuracies that was observed before and after standardizing the data. We see lots of oscillations while learning and in the Adam, learning stopped due to gradients exploding, and in terms of validation accuracy. However, in the standardized curves, we see a more stable convergence pattern overall and the Adam gradients not exploding.



Figures 11-12 – Validation Accuracies for the Different Optimizers with/without standardization

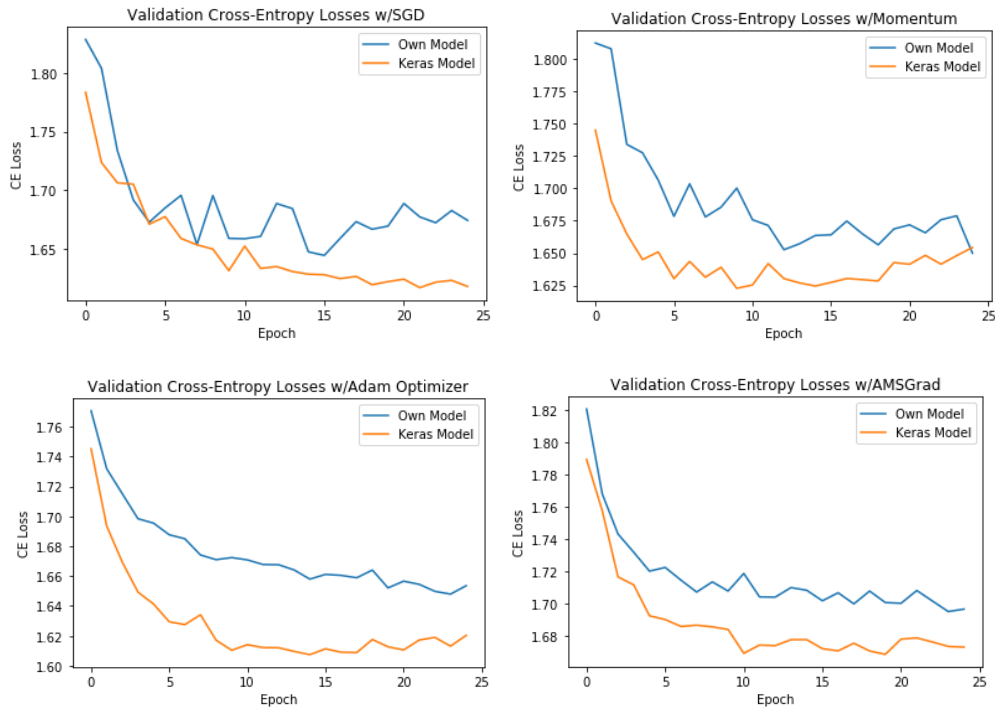
Overall, we observe what we thought we would observe. We expected that AMSGrad to be better than Adam and Adam to be better than SGDM and SGDM to be than SGD. Here, we see similar convergence points, however with different convergence rates, as we in the order we have hypothesized. These results have confirmed that we have implemented the algorithms correctly. Learning rates for AMSGrad and Adam were set as in the original paper [6], as 0.001.

Here, we observe that the initial accuracies of SGD and SGDM are far smaller with respect to Adam and Adam is smaller compared to AMSGrad. After retraining the architectures for their optimal epochs, we have moved to the test data to observe the overall test accuracy. The following table shows the test accuracies.

<i>Optimization Algorithm</i>	<i>Learning Rate</i>	<i>Other Parameters</i>	<i>Epochs</i>	<i>Training Period (min)</i>	<i>Test Accuracy</i>
<i>SGD</i>	0.1	-	25	5	43.80%
<i>SGDM</i>	0.1	$\alpha = 0.9$	25	6	44.21%
<i>Adam</i>	0.001	$\beta_1 = 0.9$ $\beta_2 = 0.999$ $\epsilon = 10^{-8}$	25	7	43.88%
<i>AMSGrad</i>	0.001	$\beta_1 = 0.9$ $\beta_2 = 0.999$ $\epsilon = 10^{-8}$	25	6	44.44%

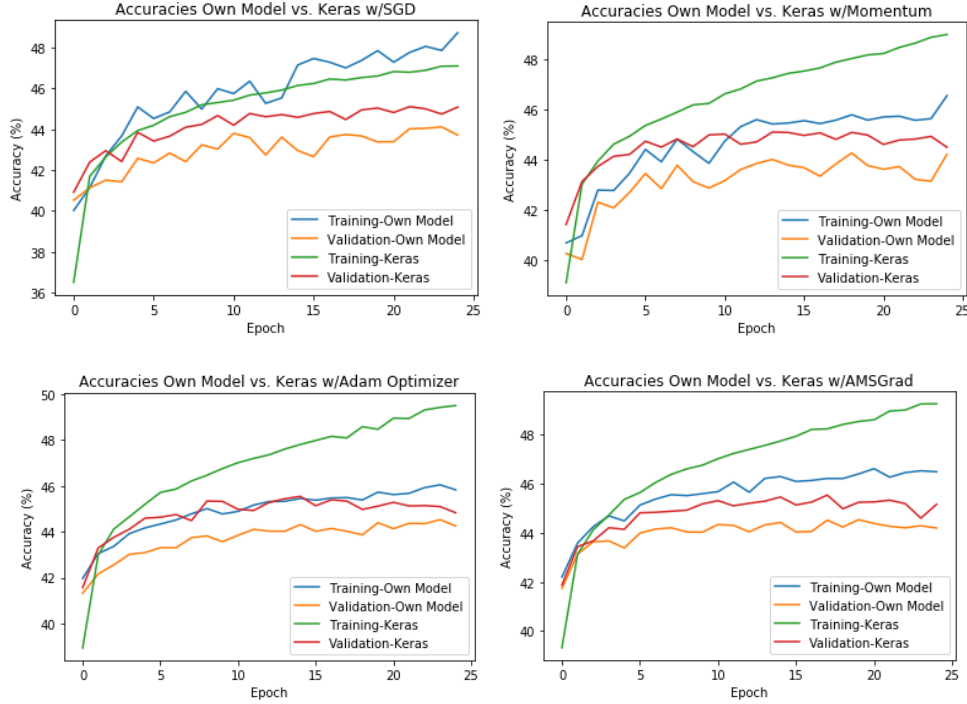
Table 2 – Test Accuracies for Best MLP Architecture

As expected, we see that the AMSGrad algorithm converges faster than the other optimizers and performs the best. Hence, the next part is to compare with the state-of-the-art. In order to do that, we have implemented the same algorithms on Keras, a highly integrated deep learning framework, and compared the results. The following graphs present the Cross-Entropy Losses on the validation set with respect to epochs for our model and the Keras Model.



Figures 13-16 – Validation Cross-Entropy Losses with SGD, SGDM, Adam and AMSGrad from left to right and up to down.

Here, we observe that the Keras implementation gives better result than ours although the overall trends are the same. This is due to the gradient checking ability of Keras that is implemented in the backend. Then, we also observe the training and validation accuracies below.



Figures 17-20 – The Training and Validation Accuracies of Our Mode land the Keras Model with SGD, SGDM, Adam and AMSGrad from left to right and up to down.

We observe that in all instances, the model built with Keras learns around 2-3% better than the model that we have constructed. Also, we observe that in all the instances, Keras model, begins to overfit to the training data after 20 epochs, even 10 in Adam and AMSGrad. Overall, we can conclude that the MLP structure has a test accuracy of 44% and a Top-3 accuracy of 72%. Overall this can be considered a successful outcome since the random guess of the data would be 4.34% and our algorithm approximately performs 10 times better than the random guess. The next phase of this algorithm was to apply PCA to extract features and then retrain the network again to see if the outcome is better.

PCA

Principle Component Analysis (PCA) is a commonly used feature extraction technique in data science to extract features or compress data. In order to apply PCA, we first standardize the data around its features, meaning the mean values of features are subtracted from the data and divided to their standard deviations. The number of features is represented with “K”.

$$\tilde{X}_{N,K} = \frac{X - \mu_X}{\sigma_X}$$

$$\mu_{X,j} = \frac{1}{N} \sum_{i=1}^N X_{i,j}, \quad \mu_X = [\mu_{X,1}, \dots, \mu_{X,K}]^T \text{ and } \sigma_{X,j} = \sqrt{\frac{1}{N} \sum_{i=1}^N X_{i,j}^2}, \quad \sigma_X = [\sigma_{X,1}, \dots, \sigma_{X,K}]^T$$

Then, we have identified the eigenvectors and eigenvalues of the covariance matrix or the autocorrelation matrix of the data, according to the orientation of the data.

$$\Sigma_X = \tilde{X}^T \tilde{X} \rightarrow (\lambda_i, e_i), \quad i = 1, \dots, K$$

The eigenvectors extracted are called as principle components. Then, we chose “k” principle components with the highest eigenvalues, which was a choice decision. We have examined explained

variance in order to choose k , for which our threshold was 90%, meaning we decided that the extracted features should explain at least 90% of the original data's variance. The graph for the percent explained variances with respect to k is given below.

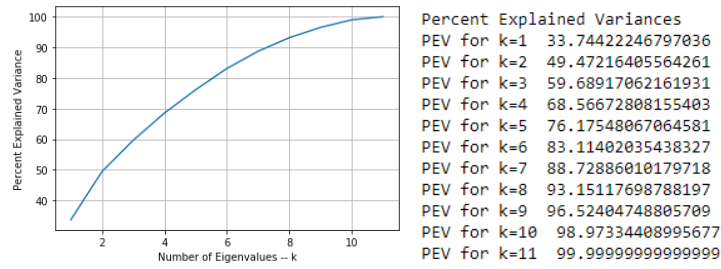


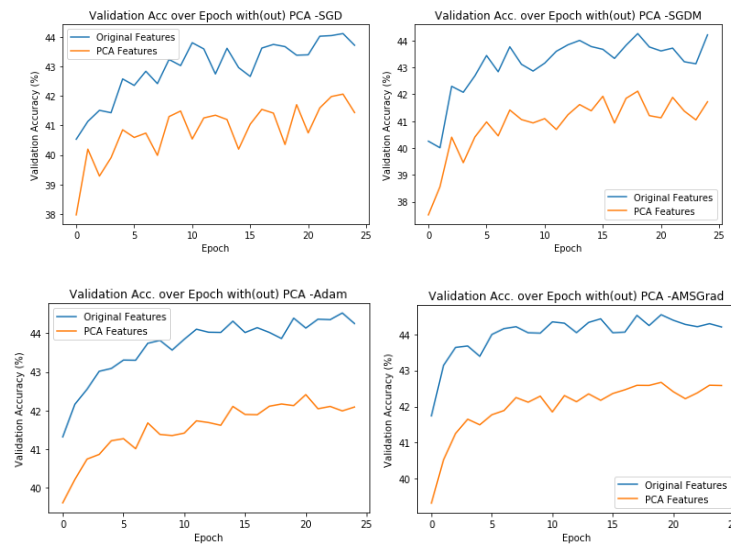
Figure 21 – Percent Explained Variance vs. Number of Eigenvalues

Hence, we have chose k value to be 8. Then, we have reconstructed the data according to the projection of the component to the feature space as follows.

$$E = [e_1, \dots, e_k]$$

$$\hat{X} = \tilde{X}EE^T + \mu_X$$

After the reconstruction, we use the reconstructed data and trained on our optimal algorithm in order to observe if the network learns better with the extracted features. The validation accuracies of the extracted features together with the original feature set for each of the optimizers are as follows.



Figures 22-25 – Validation Accuracies over Epoch with/without PCA

Here, we observe that in each of the cases, the network trained with the original features has performed better when compared with their PCA extracted versions, which was not as expected. In this project, we thought that the data would be too complex and noisy for the algorithm to understand and that is the reason we have implemented PCA. However, we now see that the algorithm we constructed could learn all the information and the reduction in variance resulted with less learning. The results of the experiment are given as a table below with the test accuracies compared.

Optimization Algorithm	Test Accuracy (PCA)	Test Accuracy (non-PCA)
SGD	41.50%	43.80%
SGDM	41.65%	44.21%
Adam	41.98%	43.88%

<i>AMSGrad</i>	42.65%	44.44%
----------------	--------	--------

Table 3 – Experiment Results with/without PCA

Random Forest

Decision Tree is an algorithm that has the advantage of classifying discrete and categorical features as well as continuous features without the need of complicated preprocessing. However, it can be very biased to the training data since it creates splits according to the training data. In order to overcome this, we have used the Random Forest algorithm which consist of many low biased weak decision Trees. Since we are doing a classification task, the split criteria of our trees' in the random forest algorithm is dependent on calculating the Information Gain by the Gini Index and of all the predictions each tree in our Random Forest does, we should get the most occurring class as our final prediction. If we were to use a Random Forest for a regression task, we could use MSE as our splitting criteria and take the mean of all the predictions that came from every single tree as our final prediction. We have used a greedy approach when splitting the trees and creating new nodes. The algorithm is as follows:

1. From the dataset, randomly select samples according to a sample size. The more the sample size, more time it takes to train the algorithm.
2. Randomly select a feature to split. We haven't been able to find the "best attribute" to split since there were many different splits possible for a single feature, the problem would become intractable given that we are building a forest and not a single tree. Hence, we have chosen the features randomly, which was the next best idea.
3. Search through the possible splits by calculating the information gain using Gini Index.
4. The maximum information gain achieving split is then determined as the splitting point.
5. Continue picking random features until the decision tree has only leaf nodes (every class has been put to a leaf node) or the maximum depth of the tree is reached.
6. Repeat steps 1-5 for the number of trees that you desire for your random forest to have.

The features that are mentioned in the Dataset Description and Preprocessing are the only necessary features in the feature set for Random Forest building. For the categorical features, we have just used one hot encoding. The reason that we have not taken the artist name in account is the fact that when it is turned to one hot encoding, the feature set became too big and at this point it is just better for the performance to put it out. To calculate the information gain, the formula that we have used is the following:

$$IG = I^{Gini}(D_{parent}) - \frac{N_{left}}{N_{parent}} I^{Gini}(D_{left}) - \frac{N_{right}}{N_{parent}} I^{Gini}(D_{right})$$

Where N_{left} denotes the number of data in the left node, N_{right} denotes the number of data in the right node, N_{parent} denotes the number of data at the parent node, D denotes the data in general, and $I^{Gini}(D)$ denotes the Gini Index value of data D . The definition of Gini Index is as given below.

$$I^{Gini}(D) = 1 - \sum_{i=1}^{N_{classes}} p_i^2$$

Where $N_{classes}$ denotes the number of classes in the dataset, and p_i denotes the probability that a random selection from data (Node) D belongs to the class "i". These are implemented as described and the one possible split with the highest information gain is selected as the best split. This algorithm is used in order to decrease the entropy of classes in every node of our trees.

We have followed the algorithm that we have specified above and created the classes for a Decision Tree and a Random Forest. Random Forest class is used for creating Decision Trees within itself according to the metrics of dataset, class labels, sample size, number of trees, number of features to be put into each tree, the maximum depth of a single tree, and the minimum number of samples at a leaf node to cause a split. In the validation split, in the Phase 1 Report, we mentioned we would use a three-way split, so we have split our data as 70%-20%-10% training-validation-test splits. We have modified our metrics and tested on the validation and test data for 14 different Random Forests. The following are the results for each Random Forest trial:

<i>Forest Number</i>	<i>Sample Size</i>	<i>Number of Trees</i>	<i>Number of Features</i>	<i>Maximum Depth</i>	<i>Minimum Leaf Size</i>	<i>Validation Accuracy</i>
1	80	40	5	10	5	18.7%
2	80	80	5	10	5	19.1%
3	120	40	5	10	5	19.2%
4	120	80	5	10	5	20.96%
5	40	40	5	10	5	18.6%
6	40	80	5	10	5	18.3%
7	120	80	3	10	5	20.3%
8	120	80	8	10	5	20.3%
9	120	80	5	20	2	19.3%
10	120	80	5	5	2	13.6%
11	120	80	5	20	10	20.94%
12	120	80	5	5	10	19.5%
13	120	80	5	10	2	15.4%
14	120	80	5	10	10	20.94%

Table 4 – Optimization Models for Random Forest Algorithm

At the end of these trials, we have chosen the best performing Random Forest as our final model which is the 4th model. For testing, we have initialized these metrics as follows according to the final model:

- Sample Size: 120
- Number of Trees in the Random Forest: 80
- Number of Features to be put into each Tree: 5 (Which is the closes integer value to the square root of total feature number).
- The maximum depth: 10
- Minimum leaf size: 5

The algorithm takes about 100 seconds to run including the prediction part. One of the disadvantages of the Random Forest algorithm is the time-consuming prediction process. Even though creating the Random Forest takes less time compared to training an MLP or CNN, the prediction part takes much longer than those algorithm since in order to predict a sample, the sample is put through all the trees in the Random Forest which takes some time.

Unfortunately, we have managed to get a training accuracy of 20.7%, validation accuracy of 20.9%, and test accuracy of 19.2%. We have retrained the best model by combining the training and validation datas. These results are the worst we have obtained compared to the other algorithms that we have used. However, it is still approximately five times the accuracy of random guessing which is about 4.3 %. Considering the low training time, this result could be expected. It is observable that the training and testing accuracies are very close. The reason for this is the uncorrelation between the trees of the Random Forest. Since while creating the tree, samples and features to be split upon are chosen at random every iteration. This leads to the Random Forest being unbiased to the training dataset.

k-Nearest Neighbors (kNN)

The other algorithm that we have implemented is the kNN (k-Nearest Neighbors) algorithm, a very easy to implement algorithm with no training phase. It emphasizes on the spatial distances and is usually a non-parametric method, containing only the value k, which determines how many neighbors will be considered for each test sample. As mentioned in the Phase 1 Report, we have used a hybrid approach to the distances, where we used a unity distance for the categorical features and Euclidean distance for the continuous features as described below.

$$d_1(x_{i,n}, x_{j,n}) = \begin{cases} 0 & \text{if } x_{i,n} = x_{j,n} \\ \alpha^2 & \text{else} \end{cases} \quad d_2(x_{i,n}, x_{j,n}) = x_{i,n} - x_{j,n}$$

$$d(x_i, x_j) = \sum_{l=0}^n ||d_t(x_{i,l}, x_{j,l})||^2, \text{ where } t = \begin{cases} 1 & \text{if } l\text{th feature is discrete} \\ 2 & \text{if } l\text{th feature is continuous} \end{cases}$$

This approach allowed us to use the information that we would gain from the categorical features, however added three new hyper-parameters to the optimization process. We have mentioned that we would run a full grid-search approach to determine the best α_i $i = 1, 2, 3$, and k . However, the training period of a kNN algorithm, and since we had around 119000 training samples, was not optimal. Hence different from Phase 1, we have applied a semi grid search. We have first found the best k value by keeping the other parameters constant, and then applied a grid search for the alpha values in a small subset of $\{0.2, 0.4, 0.6, 0.8\}$. This way, we were able find an approximately best relationship with the features and labels. We have used a 80%-1.3%-18.7% three-way split to find the best k value and hold the other parameters constant at 0.1. The validation accuracy is as follows.

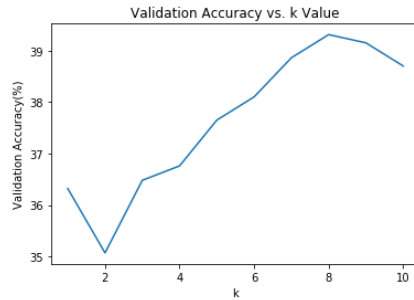


Figure 20 – Validation Accuracies with respect to k Value

Here, we observe that the best k value comes out as 8, hence we choose that for the k value that we will use. Then, we run the grid search that is mentioned before, with $k=0.8$ and the best validation accuracy comes with the parameters,

$$(\alpha_1, \alpha_2, \alpha_3, k) = (0.8, 0.2, 0.2, 0.8)$$

There is no training present in the algorithm, however the time algorithm took to classify the validation data took nearly 6 minutes for one set of parameters. In total, we have run this algorithm 74 times adding up to an amount of 7.4 hours to optimize the parameters completely.

The validation accuracy was 68.24%. This outcome was expected since the α_1 parameter represents the punishment coefficient that is dealt to the artist name. From general knowledge, we can see that the artist name is a highly correlated feature with the genre since nearly all the artists produce music under one or two genres in their careers. Then we use the same algorithm to predict the test accuracy, and the test accuracy of the algorithm with the optimized parameters become 67.18%. This was by far the most successful algorithm that we have implemented due to the use of “Artist Name” feature which was highly correlated with the labels.

Overall, implementations of all the algorithms were successful. We have faced nearly all the problems that we thought we would face in the Phase 1 report and built our algorithms through those challenges. We believe we were successful at handling categorical and continuous data in the same dataset.

References

- [1] Spotify DB Tracks, Kaggle, <https://www.kaggle.com/zaheenhamidani/ultimate-spotify-tracks-db/version/3>, [Accessed: 7-Nov-2019]
- [2] A. Aderhold, "How to code categorical inputs for a neural network?," ResearchGate. [Online]. Available: https://www.researchgate.net/post/How_to_code_categorical_inputs_for_a_neural_network. [Accessed: 27-Dec-2019].
- [3] Frey, Patrick & Ehrismann, Björn & Drück, Harald. (2011). Development of Artificial Neural Network Models for Sorption Chillers. 30th ISES Biennial Solar World Congress 2011, SWC 2011. 4. 10.18086/swc.2011.20.11.
- [4] Jayshril S. Sonawane, Dharmaraj Rajaram Patil. (2014) 'Prediction of Heart Disease Using Multilayer Perceptron Neural Network'. *International Conference on Information Communication and Embedded Systems*.
- [5] K. He, X. Zhang, S. Ren and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," 2015 IEEE International Conference on Computer Vision (ICCV), Santiago, 2015, pp. 1026-1034.
- [6] X. Glorot, Y. Bengio ; "Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics", PMLR 9:249-256, 2010.
- [7] Du, Juan. (2019). The Frontier of SGD and Its Variants in Machine Learning. *Journal of Physics: Conference Series*. 1229. 012046. 10.1088/1742-6596/1229/1/012046.
- [8] Kingma, D. P., Jimmy, and Ba, "Adam: A Method for Stochastic Optimization," arXiv.org, 30-Jan-2017. [Online]. Available: <https://arxiv.org/abs/1412.6980>. [Accessed: 06-Dec-2019].
- [9] Reddi, Satyen, Kumar, Sanjiv, and Sashank J., "On the Convergence of Adam and Beyond," arXiv.org, 19-Apr-2019. [Online]. Available: <https://arxiv.org/abs/1904.09237>. [Accessed: 07-Dec-2019].

Appendix 1 - MLP

Appendix 1-A – SGD Optimization

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3.
4. class Layer:
5.     def __init__(self, inputDim, numNeurons, activation, std, mean=0):
6.         self.inputDim = inputDim
7.         self.numNeurons = numNeurons
8.         self.activation = activation
9.
10.        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inp
utDim)
11.        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
12.        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)
13.
14.        self.delta = None
15.        self.error = None
16.        self.lastActiv = None
17.
18.        def actFcn(self,x):
19.            if(self.activation == 'sigmoid'):
20.                expx = np.exp(x)
21.                return expx/(1+expx)
22.            elif(self.activation == 'softmax'):
23.                expx = np.exp(x - np.max(x))
24.                return expx/np.sum(expx, axis=0)
25.            elif(self.activation == 'tanh'):
26.                return np.tanh(x)
27.            elif(self.activation == 'relu'):
28.                out = np.maximum(0,x)
29.                return out
30.
31.        def activate(self, x):
32.            if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation ==
'relu':
33.                if(x.ndim == 1):
34.                    x = x.reshape(x.shape[0],1)
35.                    numSamples = x.shape[1]
36.                    tempInp = np.r_[x, [np.ones(numSamples)*1]]
37.                    self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
38.                return self.lastActiv
39.
40.        def derActiv(self, x):
41.            if(self.activation == 'sigmoid'):
42.                return x*(1-x)
43.            elif(self.activation == 'softmax'):
44.                return x*(1-x)
45.            elif(self.activation == 'tanh'):
46.                return 1 - x**2;
47.            elif(self.activation == 'relu'):
48.                der = x
49.                der[x > 0] = 1
50.                der[x <= 0] = 0
51.                return der
52.
53.        def __repr__(self):
54.            return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " + str(self.numNeur
ons) + "\n Activation: " + self.activation
55.
56.        class MLP:
57.            def __init__(self):
58.                self.layers = []
```

```

59.
60.     def addLayer(self, layer):
61.         self.layers.append(layer)
62.
63.     def forward(self, inp):
64.         out = inp
65.         for lyr in self.layers:
66.             out = lyr.activate(out)
67.         return out
68.
69.     def prediction(self, inp):
70.         out = self.forward(inp)
71.         if(out.ndim == 1):
72.             return np.argmax(out)
73.         return np.argmax(out, axis=0)
74.
75.     def topKaccuracy(self, inp, realOut, k):
76.         out = self.forward(inp)
77.         cnt = 0
78.         for i in range(out.shape[1]):
79.             topKind = (out[:,i]).argsort()[::-1][:k]
80.             if realOut[i] in topKind:
81.                 cnt += 1
82.         return cnt/out.shape[1]
83.
84.     def backProp(self, inp, out, lrnRate, batchSize, loss):
85.         net_out = self.forward(inp)
86.         for i in reversed(range(len(self.layers))):
87.             lyr = self.layers[i]
88.             #outputLayer
89.             if(lyr == self.layers[-1]):
90.                 if(loss == 'mse'):
91.                     lyr.error = net_out - out
92.                     derMatrix = lyr.derActiv(lyr.lastActiv)
93.                     lyr.delta = derMatrix * lyr.error
94.                 elif(loss == 'ce' and lyr.activation == 'softmax'):
95.                     lyr.delta = net_out - out
96.                 else:
97.                     assert('Cant do that')
98.             #hiddenLayer
99.             else:
100.                 nextLyr = self.layers[i+1]
101.                 nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
102.                 lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
103.                 derMatrix = lyr.derActiv(lyr.lastActiv)
104.                 lyr.delta = derMatrix * lyr.error
105.
106.             #update weights
107.             for i in range(len(self.layers)):
108.                 lyr = self.layers[i]
109.                 if(i == 0):
110.                     if(inp.ndim == 1):
111.                         inp = inp.reshape(inp.shape[0],1)
112.                         numSamples = inp.shape[1]
113.                         inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
114.                     else:
115.                         numSamples = self.layers[i - 1].lastActiv.shape[1]
116.                         inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]
117.
118.                 update = (lrnRate * np.matmul(lyr.delta, inputToUse.T))/batchSize
119.                 lyr.weightsE -= update
120.
121.     def train(self, inp, out, inpTest, outTest, lrnRate, epochNum, batchSize, loss):
122.         errList = []
123.         trAccs = []
124.         valAccs = []

```



```

125.         valKAccs = []
126.
127.         for ep in range(epochNum):
128.             print('-----
\nEpoch', ep+1)
129.
130.             randomIndexes = np.random.permutation(len(inp))
131.             inp = inp[randomIndexes]
132.             out = out[randomIndexes]
133.             numBatches = int(np.floor(len(inp)/batchSize))
134.
135.             for j in range(numBatches):
136.                 batch_inp = inp[batchSize*j:batchSize*j+batchSize]
137.                 batch_out = out[batchSize*j:batchSize*j+batchSize]
138.
139.                 batch_out_1H = mat1H2(batch_out, np.max(out)+1).T
140.
141.                 self.backProp(batch_inp.T, batch_out_1H, lrnRate, batchSize, loss)
142.
143.                 valOutput = self.forward(inpTest.T)
144.                 if(loss == 'ce'):
145.                     err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
146.                     .shape[1]
147.                 elif(loss == 'mse'):
148.                     err = np.sum((outTest.T - self.forward(inpTest.T))**2)/valOutput.shape[1]
149.                 print(loss.upper() + ' Validation Error ', err)
150.                 errList.append(err)
151.
152.                 trPred = self.prediction(inp.T)
153.                 trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
154.                 print('Training Accuracy: ', trAcc)
155.                 trAccs.append(trAcc)
156.
157.                 valPred = self.prediction(inpTest.T)
158.                 valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]
159.                 *100
160.                 print('Validation Accuracy: ', valAcc)
161.                 valAccs.append(valAcc)
162.
163.                 #k is chosen as three
164.                 K = 3
165.                 topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
166.                 print('Top ' + str(K) + ' Accuracy: ', topKacc)
167.                 valKAccs.append(topKacc)
168.
169.             return errList, trAccs, valAccs, valKAccs
170.
171.         def __repr__(self):
172.             retStr = ""
173.             for i, lyr in enumerate(self.layers):
174.                 retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
175.             return retStr
176.
177.         features = np.load('dataMLP.npy', allow_pickle=True)
178.         labels = np.load('labelsMLP.npy', allow_pickle=True)
179.         labels = labels.reshape((labels.shape[0],1))
180.         features -= np.mean(features, axis=0)
181.         features /= np.std(features, axis=0)
182.
183.         #three-way split
184.         def splitData(X, y, tr, val):
185.             np.random.seed(456)
186.             shuffle = np.random.permutation(len(y))
187.             X = X[shuffle]

```

```

188. y = y[shuffle]
189.
190. tr_ind = int(np.floor(len(y)*tr))
191. val_ind = int(np.floor(len(y)*(tr+val)))
192.
193. X_tr = X[0:tr_ind]
194. y_tr = y[0:tr_ind]
195. X_val = X[tr_ind:val_ind]
196. y_val = y[tr_ind:val_ind]
197. X_test = X[val_ind:]
198. y_test = y[val_ind:]
199.
200. return X_tr, y_tr, X_val, y_val, X_test, y_test
201.
202. def vector1H(x, maxInd):
203.     out = np.zeros(maxInd)
204.     out[x] = 1
205.     return out
206. def mat1H2(y, maxInd):
207.     out = np.zeros((y.shape[0], maxInd))
208.     for i in range(y.shape[0]):
209.         out[i,:] = vector1H(y[i], maxInd)
210.     return out
211.
212. x_train, y_train, x_val, y_val, x_test, y_test = splitData(features, labels, 0.7, 0.2)
213.
214. from tensorflow.keras import layers, models, initializers, optimizers
215. model = models.Sequential()
216. model.add(layers.Dense(500, input_shape=(11,), activation='relu', \
217.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
218.                                distribution = 'normal'
219.                                , \
220.                                seed=None, \
221.                                scale=2.0), \
222.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
223.                                distribution = 'normal'
224.                                , \
225.                                seed=None, \
226.                                scale=2.0)))
225. model.add(layers.Dense(300, activation='relu', \
226.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
227.                                distribution = 'normal'
228.                                , \
229.                                seed=None, \
230.                                scale=2.0), \
231.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
232.                                distribution = 'normal'
233.                                , \
234.                                seed=None, \
235.                                scale=2.0)))
234. model.add(layers.Dense(150, activation='relu', \
235.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
236.                                distribution = 'normal'
237.                                , \
238.                                seed=None, \
239.                                scale=2.0), \
240.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
241.                                distribution = 'normal'
242.                                , \
243.                                seed=None, \
244.                                scale=2.0)))
243. model.add(layers.Dense(65, activation='relu', \
244.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
245.                                distribution = 'normal'
246.                                , \
247.                                seed=None, \

```

```

247.                                     scale=2.0),\
248.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
249.                                     distribution = 'normal'
250.             , \
251.                                     seed=None, \
252.                                     scale=2.0)))
252.model.add(layers.Dense(np.max(labels)+1, activation='softmax',\
253.             kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
254.                                     distribution = 'normal'
255.             , \
256.                                     seed=None,\
257.                                     scale=2.0),\
258.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
259.                                     distribution = 'normal'
260.             , \
261.                                     seed=None, \
262.                                     scale=2.0)))
261.momOp = optimizers.SGD(lr=0.1, momentum=0, nesterov=False, decay=0)
262.model.compile(loss='sparse_categorical_crossentropy', optimizer=momOp, metrics=['accuracy'])
263.hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)
264.
265.#1st Iteration
266.mlp1 = MLP()
267.mlp1.addLayer(Layer(x_train.shape[1], 300, 'sigmoid', np.sqrt(6/(x_train.shape[1]+300))))
268.mlp1.addLayer(Layer(300, np.max(labels)+1, 'softmax', np.sqrt(6/(300+np.max(labels)+1))))
269.ceErrs, trAccs, valAccs1, valKAccs = mlp1.train(x_train, y_train, x_val, y_val, 0.01, 25, 529,
270.             loss='ce')
271.#2nd Iteration
272.mlp2 = MLP()
273.mlp2.addLayer(Layer(x_train.shape[1], 300, 'sigmoid', np.sqrt(6/(x_train.shape[1]+300))))
274.mlp2.addLayer(Layer(300, 100, 'sigmoid', np.sqrt(6/(300+100))))
275.mlp2.addLayer(Layer(100, np.max(labels)+1, 'softmax', np.sqrt(6/(100+np.max(labels)+1))))
276.ceErrs, trAccs, valAccs2, valKAccs = mlp2.train(x_train, y_train, x_val, y_val, 0.01, 25, 529,
277.             loss='ce')
278.#3rd Iteration
279.mlp3 = MLP()
280.mlp3.addLayer(Layer(x_train.shape[1], 300, 'relu', np.sqrt(6/(x_train.shape[1]+300))))
281.mlp3.addLayer(Layer(300, 100, 'relu', np.sqrt(6/(300+100))))
282.mlp3.addLayer(Layer(100, np.max(labels)+1, 'softmax', np.sqrt(6/(100+np.max(labels)+1))))
283.ceErrs, trAccs, valAccs3, valKAccs = mlp3.train(x_train, y_train, x_val, y_val, 0.1, 25, 529,
284.             loss='ce')
285.#4th Iteration
286.mlp4 = MLP()
287.mlp4.addLayer(Layer(x_train.shape[1], 300, 'relu', np.sqrt(2/300)))
288.mlp4.addLayer(Layer(300, 100, 'relu', np.sqrt(2/100)))
289.mlp4.addLayer(Layer(100, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
290.ceErrs, trAccs, valAccs4, valKAccs = mlp4.train(x_train, y_train, x_val, y_val, 0.1, 25, 529,
291.             loss='ce')
292.#5th Iteration
293.mlp5 = MLP()
294.mlp5.addLayer(Layer(x_train.shape[1], 400, 'relu', np.sqrt(2/400)))
295.mlp5.addLayer(Layer(400, 200, 'relu', np.sqrt(2/200)))
296.mlp5.addLayer(Layer(200, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
297.ceErrs, trAccs, valAccs5, valKAccs = mlp5.train(x_train, y_train, x_val, y_val, 0.1, 25, 529,
298.             loss='ce')
299.#6th Iteration
300.mlp6 = MLP()
301.mlp6.addLayer(Layer(x_train.shape[1], 400, 'relu', np.sqrt(2/400)))
302.mlp6.addLayer(Layer(400, 200, 'relu', np.sqrt(2/200)))
303.mlp6.addLayer(Layer(200, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))

```

```

304.ceErrs, trAccs, valAccs6, valKAccs = mlp6.train(x_train, y_train, x_val, y_val, 0.01, 25, 529,
    loss='mse')
305.
306.#7th Iteration
307.mlp7 = MLP()
308.mlp7.addLayer(Layer(x_train.shape[1], 600, 'relu', np.sqrt(2/600)))
309.mlp7.addLayer(Layer(600, 200, 'relu', np.sqrt(2/200)))
310.mlp7.addLayer(Layer(200, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
311.ceErrs, trAccs, valAccs7, valKAccs = mlp7.train(x_train, y_train, x_val, y_val, 0.1, 25, 529,
    loss='ce')
312.
313.#8th Iteration
314.mlp8 = MLP()
315.mlp8.addLayer(Layer(x_train.shape[1], 600, 'relu', np.sqrt(2/600)))
316.mlp8.addLayer(Layer(600, 400, 'relu', np.sqrt(2/400)))
317.mlp8.addLayer(Layer(400, 200, 'relu', np.sqrt(2/200)))
318.mlp8.addLayer(Layer(200, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
319.ceErrs, trAccs, valAccs8, valKAccs = mlp8.train(x_train, y_train, x_val, y_val, 0.1, 25, 529,
    loss='ce')
320.
321.#9th Iteration
322.mlp9 = MLP()
323.mlp9.addLayer(Layer(x_train.shape[1], 750, 'relu', np.sqrt(2/750)))
324.mlp9.addLayer(Layer(750, 600, 'relu', np.sqrt(2/600)))
325.mlp9.addLayer(Layer(600, 400, 'relu', np.sqrt(2/400)))
326.mlp9.addLayer(Layer(400, 200, 'relu', np.sqrt(2/200)))
327.mlp9.addLayer(Layer(200, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
328.ceErrs, trAccs, valAccs9, valKAccs = mlp9.train(x_train, y_train, x_val, y_val, 0.1, 25, 529,
    loss='ce')
329.
330.#10th Iteration
331.mlp10 = MLP()
332.mlp10.addLayer(Layer(x_train.shape[1], 750, 'relu', np.sqrt(2/750)))
333.mlp10.addLayer(Layer(750, 650, 'relu', np.sqrt(2/650)))
334.mlp10.addLayer(Layer(650, 450, 'relu', np.sqrt(2/450)))
335.mlp10.addLayer(Layer(450, 300, 'relu', np.sqrt(2/300)))
336.mlp10.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
337.mlp10.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
338.mlp10.addLayer(Layer(65, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
339.ceErrs, trAccs, valAccs10, valKAccs = mlp10.train(x_train, y_train, x_val, y_val, 0.05, 25, 52
    9, loss='ce')
340.
341.#Final Version
342.mlp = MLP()
343.mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
344.mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
345.mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
346.mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
347.mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax', \
348.         np.sqrt(2/(np.max(labels)+1))))
349.
350.ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 0.1, 25, 529, lo
    ss='ce')
351.
352.print(hist.history.keys())
353.plt.plot(ceErrs)
354.plt.plot(hist.history['val_loss'])
355.plt.title('Validation Cross-Entropy Losses w/SGD')
356.plt.xlabel('Epoch')
357.plt.ylabel('CE Loss')
358.plt.legend(['Own Model', 'Keras Model'])
359.plt.show()
360.
361.plt.plot(trAccs)
362.plt.plot(valAccs)
363.plt.plot(valKAccs)

```

```

364.plt.title('Accuracies Own Model w/SGD')
365.plt.xlabel('Epoch')
366.plt.ylabel('Accuracy (%)')
367.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
368.plt.show()
369.
370.plt.plot(trAccs)
371.plt.plot(valAccs)
372.plt.plot(np.asarray(hist.history['accuracy'])*100)
373.plt.plot(np.asarray(hist.history['val_accuracy'])*100)
374.plt.title('Accuracies Own Model vs. Keras w/SGD')
375.plt.xlabel('Epoch')
376.plt.ylabel('Accuracy (%)')
377.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
    Keras'])
378.plt.show()
379.np.save('valAcc_SGD.npy', valAccs)
380.
381.testPred = mlp.prediction(x_test.T)
382.testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
383.print(testAcc)
384.
385.fig=plt.figure(figsize=(18, 12), dpi= 80, facecolor='w', edgecolor='k')
386.plt.plot(valAccs1)
387.plt.plot(valAccs2)
388.plt.plot(valAccs3)
389.plt.plot(valAccs4)
390.plt.plot(valAccs5)
391.plt.plot(valAccs6)
392.plt.plot(valAccs7)
393.plt.plot(valAccs8)
394.plt.plot(valAccs9)
395.plt.plot(valAccs10)
396.plt.plot(valAccs)
397.plt.legend(['MLP1', 'MLP2', 'MLP3', 'MLP4', 'MLP5', 'MLP6', 'MLP7', 'MLP8', 'MLP9', 'MLP10', 'MLP*'])
398.plt.show()
399.
400.print('MLP1\n',mlp1)
401.print('MLP2\n',mlp2)
402.print('MLP3\n',mlp3)
403.print('MLP4\n',mlp4)
404.print('MLP5\n',mlp5)
405.print('MLP6\n',mlp6)
406.print('MLP7\n',mlp7)
407.print('MLP8\n',mlp8)
408.print('MLP9\n',mlp9)
409.print('MLP10\n',mlp10)
410.print('MLP*\n',mlp)

```

Appendix 1-B – SGDM

```

1. import matplotlib.pyplot as plt
2. import numpy as np
3.
4. class Layer:
5.     def __init__(self, inputDim, numNeurons, activation, std, mean=0):
6.         self.inputDim = inputDim
7.         self.numNeurons = numNeurons
8.         self.activation = activation
9.
10.        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inp
    utDim)
11.        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
12.        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)
13.

```

```

14.         self.delta = None
15.         self.error = None
16.         self.lastActiv = None
17.
18.         self.prevUpdate = 0
19.
20.
21.     def actFcn(self,x):
22.         if(self.activation == 'sigmoid'):
23.             expx = np.exp(x)
24.             return expx/(1+expx)
25.         elif(self.activation == 'softmax'):
26.             expx = np.exp(x - np.max(x))
27.             return expx/np.sum(expx, axis=0)
28.         elif(self.activation == 'tanh'):
29.             return np.tanh(x)
30.         elif(self.activation == 'relu'):
31.             out = np.maximum(0,x)
32.             return out
33.
34.     def activate(self, x):
35.         if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation ==
'relu':
36.             if(x.ndim == 1):
37.                 x = x.reshape(x.shape[0],1)
38.                 numSamples = x.shape[1]
39.                 tempInp = np.r_[x, [np.ones(numSamples)*1]]
40.                 self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
41.                 return self.lastActiv
42.
43.     def derActiv(self, x):
44.         if(self.activation == 'sigmoid'):
45.             return x*(1-x)
46.         elif(self.activation == 'softmax'):
47.             return x*(1-x)
48.         elif(self.activation == 'tanh'):
49.             return 1 - x**2;
50.         elif(self.activation == 'relu'):
51.             der = x
52.             der[x > 0] = 1
53.             der[x <= 0] = 0
54.             return der
55.
56.     def __repr__(self):
57.         return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " + str(self.numNeur
ons) + "\n Activation: " + self.activation
58.
59. class MLP:
60.     def __init__(self):
61.         self.layers = []
62.
63.     def addLayer(self, layer):
64.         self.layers.append(layer)
65.
66.     def forward(self, inp):
67.         out = inp
68.         for lyr in self.layers:
69.             out = lyr.activate(out)
70.         return out
71.
72.     def prediction(self, inp):
73.         out = self.forward(inp)
74.         if(out.ndim == 1):
75.             return np.argmax(out)
76.         return np.argmax(out, axis=0)
77.

```

```

78.     def topKaccuracy(self, inp, realOut, k):
79.         out = self.forward(inp)
80.         cnt = 0
81.         for i in range(out.shape[1]):
82.             topKind = (out[:,i]).argsort()[::-1][:k]
83.             if realOut[i] in topKind:
84.                 cnt += 1
85.         return cnt/out.shape[1]
86.
87.     def backProp(self, inp, out, lrnRate, momCoeff, batchSize, loss):
88.         net_out = self.forward(inp)
89.         for i in reversed(range(len(self.layers))):
90.             lyr = self.layers[i]
91.             #outputLayer
92.             if(lyr == self.layers[-1]):
93.                 if(loss == 'mse'):
94.                     lyr.error = out - net_out
95.                     derMatrix = lyr.derActiv(lyr.lastActiv)
96.                     lyr.delta = derMatrix * lyr.error
97.                 elif(loss == 'ce' and lyr.activation == 'softmax'):
98.                     lyr.delta = net_out - out
99.                 else:
100.                    assert('Cant do that')
101.            #hiddenLayer
102.            else:
103.                nextLyr = self.layers[i+1]
104.                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
105.                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
106.                derMatrix = lyr.derActiv(lyr.lastActiv)
107.                lyr.delta = derMatrix * lyr.error
108.
109.            #update weights
110.            for i in range(len(self.layers)):
111.                lyr = self.layers[i]
112.                if(i == 0):
113.                    if(inp.ndim == 1):
114.                        inp = inp.reshape(inp.shape[0],1)
115.                        numSamples = inp.shape[1]
116.                        inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
117.                    else:
118.                        numSamples = self.layers[i - 1].lastActiv.shape[1]
119.                        inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]
120.
121.                update = (lrnRate * np.matmul(lyr.delta, inputToUse.T))/batchSize
122.                lyr.weightsE -= (update + momCoeff * lyr.prevUpdate)
123.                lyr.prevUpdate = update
124.
125.     def train(self, inp, out, inpTest, outTest, lrnRate, momCoeff, epochNum, batchSize, loss):
126.         errList = []
127.         trAccs = []
128.         valAccs = []
129.         valKAccs = []
130.
131.         for ep in range(epochNum):
132.             print('-----\nEpoch', ep+1)
133.
134.             randomIndexes = np.random.permutation(len(inp))
135.             inp = inp[randomIndexes]
136.             out = out[randomIndexes]
137.             numBatches = int(np.floor(len(inp)/batchSize))
138.
139.             for j in range(numBatches):
140.                 batch_inp = inp[batchSize*j:batchSize*j+batchSize]
141.                 batch_out = out[batchSize*j:batchSize*j+batchSize]

```

```

142.
143.         batch_out_1H = mat1H2(batch_out, np.max(out)+1).T
144.
145.         self.backProp(batch_inp.T, batch_out_1H, lrnRate, momCoeff, batchSize, loss)
146.
147.         valOutput = self.forward(inpTest.T)
148.         if(loss == 'ce'):
149.             err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
150.             .shape[1]
151.         elif(loss == 'mse'):
152.             err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
153.             print(loss.upper() + ' Validation Error ', err)
154.             errList.append(err)
155.
156.             trPred = self.prediction(inp.T)
157.             trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
158.             print('Training Accuracy: ', trAcc)
159.             trAccs.append(trAcc)
160.
161.             valPred = self.prediction(inpTest.T)
162.             valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]
163.             *100
164.             print('Validation Accuracy: ', valAcc)
165.             valAccs.append(valAcc)
166.
167.             #k is chosen as three
168.             K = 3
169.             topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
170.             print('Top ' + str(K) + ' Accuracy: ', topKacc)
171.             valKAccs.append(topKacc)
172.
173.         return errList, trAccs, valAccs, valKAccs
174.
175.     def __repr__(self):
176.         retStr = ""
177.         for i, lyr in enumerate(self.layers):
178.             retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
179.         return retStr
180.
181. features = np.load('dataMLP.npy', allow_pickle=True)
182. labels = np.load('labelsMLP.npy', allow_pickle=True)
183. labels = labels.reshape((labels.shape[0],1))
184. features -= np.mean(features, axis=0)
185. features /= np.std(features,axis=0)
186.
187. #three-way split
188. def splitData(X, y, tr, val):
189.     np.random.seed(456)
190.     shuffle = np.random.permutation(len(y))
191.     X = X[shuffle]
192.     y = y[shuffle]
193.
194.     tr_ind = int(np.floor(len(y)*tr))
195.     val_ind = int(np.floor(len(y)*(tr+val)))
196.
197.     X_tr = X[0:tr_ind]
198.     y_tr = y[0:tr_ind]
199.     X_val = X[tr_ind:val_ind]
200.     y_val = y[tr_ind:val_ind]
201.     X_test = X[val_ind:]
202.     y_test = y[val_ind:]
203.
204.     return X_tr, y_tr, X_val, y_val, X_test, y_test
205.

```



```

206. def vector1H(x, maxInd):
207.     out = np.zeros(maxInd)
208.     out[x] = 1
209.     return out
210. def mat1H2(y, maxInd):
211.     out = np.zeros((y.shape[0], maxInd))
212.     for i in range(y.shape[0]):
213.         out[i,:] = vector1H(y[i], maxInd)
214.     return out
215.
216. x_train, y_train, x_val, y_val, x_test, y_test = splitData(features, labels, 0.7, 0.2)
217.
218. from tensorflow.keras import layers, models, initializers, optimizers
219. model = models.Sequential()
220. model.add(layers.Dense(500, input_shape=(11,), \
221.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
222.                                                                           distribution = 'normal'
223.                                                                           , \
224.                                                                           seed=None, \
225.                                                                           scale=2.0), \
226.                        activation='relu'))
227. model.add(layers.Dense(300, \
228.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
229.                                                                           distribution = 'normal'
230.                                                                           , \
231.                                                                           seed=None, \
232.                                                                           scale=2.0), \
233.                        activation='relu'))
234. model.add(layers.Dense(150, \
235.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
236.                                                                           distribution = 'normal'
237.                                                                           , \
238.                                                                           seed=None, \
239.                                                                           scale=2.0), \
240.                        activation = 'relu'))
241. model.add(layers.Dense(65, \
242.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
243.                                                                           distribution = 'normal'
244.                                                                           , \
245.                                                                           seed=None, \
246.                                                                           scale=2.0), \
247.                        activation='softmax'))
248. momOp = optimizers.SGD(lr=0.1, momentum=0.9, nesterov=False)
249. model.compile(loss='sparse_categorical_crossentropy', optimizer=momOp, metrics=['accuracy'])
250. hist = model.fit(x_train, y_train, epochs=25, validation_data=(x_val, y_val), batch_size=529)
251.
252. mlp = MLP()
253. mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
254. mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
255. mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
256. mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
257. mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax', \
258.                    np.sqrt(2/(np.max(labels)+1))))
259.
260. ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, \
261.                                                0.1, 0.9, 25, 529, loss='ce')
262.
263. print(hist.history.keys())
264. plt.plot(ceErrs)

```

```

267.plt.plot(hist.history['val_loss'])
268.plt.title('Validation Cross-Entropy Losses w/Momentum')
269.plt.xlabel('Epoch')
270.plt.ylabel('CE Loss')
271.plt.legend(['Own Model', 'Keras Model'])
272.plt.show()
273.
274.plt.plot(trAccs)
275.plt.plot(valAccs)
276.plt.plot(valKAccs)
277.plt.title('Accuracies Own Model w/Momentum')
278.plt.xlabel('Epoch')
279.plt.ylabel('Accuracy (%)')
280.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
281.plt.show()
282.
283.plt.plot(trAccs)
284.plt.plot(valAccs)
285.plt.plot(np.asarray(hist.history['accuracy'])*100)
286.plt.plot(np.asarray(hist.history['val_accuracy'])*100)
287.plt.title('Accuracies Own Model vs. Keras w/Momentum')
288.plt.xlabel('Epoch')
289.plt.ylabel('Accuracy (%)')
290.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
Keras'])
291.plt.show()
292.
293.np.save('valAcc_Mom.npy', valAccs)
294.
295.testPred = mlp.prediction(x_test.T)
296.testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
297.     print(testAcc)

```

Appendix 1-C – Adam

```

1. import matplotlib.pyplot as plt
2. import numpy as np
3.
4. class Layer:
5.     def __init__(self, inputDim, numNeurons, activation, std, mean=0):
6.         self.inputDim = inputDim
7.         self.numNeurons = numNeurons
8.         self.activation = activation
9.
10.        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inp
utDim)
11.        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
12.        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)
13.
14.        self.delta = None
15.        self.error = None
16.        self.lastActiv = None
17.
18.        self.prevM = 0.0
19.        self.prevV = 0.0
20.
21.
22.    def actFcn(self,x):
23.        if(self.activation == 'sigmoid'):
24.            expx = np.exp(x)
25.            return expx/(1+expx)
26.        elif(self.activation == 'softmax'):
27.            expx = np.exp(x - np.max(x))
28.            return expx/np.sum(expx, axis=0)
29.        elif(self.activation == 'tanh'):

```

```

30.         return np.tanh(x)
31.     elif(self.activation == 'relu'):
32.         out = np.maximum(0,x)
33.         return out
34.
35.     def activate(self, x):
36.         if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation ==
'relu':
37.             if(x.ndim == 1):
38.                 x = x.reshape(x.shape[0],1)
39.                 numSamples = x.shape[1]
40.                 tempInp = np.r_[x, [np.ones(numSamples)*1]]
41.                 self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
42.                 return self.lastActiv
43.
44.     def derActiv(self, x):
45.         if(self.activation == 'sigmoid'):
46.             return x*(1-x)
47.         elif(self.activation == 'softmax'):
48.             return x*(1-x)
49.         elif(self.activation == 'tanh'):
50.             return 1 - x**2;
51.         elif(self.activation == 'relu'):
52.             der = x
53.             der[x > 0] = 1
54.             der[x <= 0] = 0
55.             return der
56.
57.     def __repr__(self):
58.         return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " + str(self.numNeur
ons) + "\n Activation: " + self.activation
59.
60. class MLP:
61.     def __init__(self):
62.         self.layers = []
63.
64.     def addLayer(self, layer):
65.         self.layers.append(layer)
66.
67.     def forward(self, inp):
68.         out = inp
69.         for lyr in self.layers:
70.             out = lyr.activate(out)
71.         return out
72.
73.     def prediction(self, inp):
74.         out = self.forward(inp)
75.         if(out.ndim == 1):
76.             return np.argmax(out)
77.         return np.argmax(out, axis=0)
78.
79.     def topKaccuracy(self, inp, realOut, k):
80.         out = self.forward(inp)
81.         cnt = 0
82.         for i in range(out.shape[1]):
83.             topKind = (out[:,i]).argsort()[::-1][:k]
84.             if realOut[i] in topKind:
85.                 cnt += 1
86.         return cnt/out.shape[1]
87.
88.     def backProp(self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
89.         net_out = self.forward(inp)
90.         for i in reversed(range(len(self.layers))):
91.             lyr = self.layers[i]
92.             #outputLayer
93.             if(lyr == self.layers[-1]):

```

```

94.         if(loss == 'mse'):
95.             lyr.error = out - net_out
96.             derMatrix = lyr.derActiv(lyr.lastActiv)
97.             lyr.delta = derMatrix * lyr.error
98.         elif(loss == 'ce' and lyr.activation == 'softmax'):
99.             lyr.delta = net_out - out
100.        else:
101.            assert('Cant do that')
102.        #hiddenLayer
103.        else:
104.            nextLyr = self.layers[i+1]
105.            nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
106.            lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
107.            derMatrix = lyr.derActiv(lyr.lastActiv)
108.            lyr.delta = derMatrix * lyr.error
109.
110.        #update weights
111.        for i in range(len(self.layers)):
112.            lyr = self.layers[i]
113.            if(i == 0):
114.                if(inp.ndim == 1):
115.                    inp = inp.reshape(inp.shape[0],1)
116.                    numSamples = inp.shape[1]
117.                    inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
118.                else:
119.                    numSamples = self.layers[i - 1].lastActiv.shape[1]
120.                    inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]
121.
122.            grad = np.matmul(lyr.delta, inputToUse.T)/batchSize
123.
124.            M = (1-beta1)*grad + beta1*lyr.prevM
125.            V = (1-beta2)*np.power(grad,2)+ beta2*lyr.prevV
126.
127.            alpha_t = lrnRate * np.sqrt(1-np.power(beta2,curEpoch))/(1-
np.power(beta1,curEpoch))
128.            lyr.weightsE -= alpha_t * M / (np.sqrt(V) + 1e-7)
129.            ....
130.            Algorithm 1 from Kingma and Ba
131.            M_hat = M/(1- np.power(beta1,curEpoch))
132.            V_hat = V/(1- np.power(beta2,curEpoch))
133.
134.            lyr.weightsE -= (lrnRate*M_hat)/(np.sqrt(V_hat) + 1e-7)'''
135.
136.            lyr.prevM = M
137.            lyr.prevV = V
138.
139.        def train(self, inp, out, inpTest, outTest, epochNum, batchSize, loss, lrnRate=0.001, beta
1=0.9, beta2=0.999):
140.            errList = []
141.            trAccs = []
142.            valAccs = []
143.            valKAccs = []
144.
145.            for ep in range(epochNum):
146.                print('-----
\nEpoch', ep+1)
147.
148.                randomIndexes = np.random.permutation(len(inp))
149.                inp = inp[randomIndexes]
150.                out = out[randomIndexes]
151.                numBatches = int(np.floor(len(inp)/batchSize))
152.                for j in range(numBatches):
153.                    batch_inp = inp[batchSize*j:batchSize*j+batchSize]
154.                    batch_out = out[batchSize*j:batchSize*j+batchSize]
155.
156.                    batch_out_1H = mat1H2(batch_out, np.max(out)+1).T

```

```

157.
158.         #self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
159.         self.backProp(batch_inp.T, batch_out_1H, batchSize, loss, ep+1, lrnRate, beta1
, beta2)
160.
161.         valOutput = self.forward(inpTest.T)
162.         if(loss == 'ce'):
163.             err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
.shape[1]
164.         elif(loss == 'mse'):
165.             err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
166.             print(loss.upper() + ' Validation Error ', err)
167.             errList.append(err)
168.
169.             trPred = self.prediction(inp.T)
170.             trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
171.             print('Training Accuracy: ', trAcc)
172.             trAccs.append(trAcc)
173.
174.             valPred = self.prediction(inpTest.T)
175.             valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]
*100
176.             print('Validation Accuracy: ', valAcc)
177.             valAccs.append(valAcc)
178.
179.             #k is chosen as three
180.             K = 3
181.             topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
182.             print('Top ' + str(K) + ' Accuracy: ', topKacc)
183.             valKAccs.append(topKacc)
184.
185.
186.         return errList, trAccs, valAccs, valKAccs
187.
188.     def __repr__(self):
189.         retStr = ""
190.         for i, lyr in enumerate(self.layers):
191.             retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
192.         return retStr
193.
194. features = np.load('dataMLP.npy', allow_pickle=True)
195. labels = np.load('labelsMLP.npy', allow_pickle=True)
196. labels = labels.reshape((labels.shape[0],1))
197. features -= np.mean(features,axis=0)
198. features /= np.std(features, axis=0)
199.
200. #three-way split
201. def splitData(X, y, tr, val):
202.     np.random.seed(456)
203.     shuffle = np.random.permutation(len(y))
204.     X = X[shuffle]
205.     y = y[shuffle]
206.
207.     tr_ind = int(np.floor(len(y)*tr))
208.     val_ind = int(np.floor(len(y)*(tr+val)))
209.
210.     X_tr = X[0:tr_ind]
211.     y_tr = y[0:tr_ind]
212.     X_val = X[tr_ind:val_ind]
213.     y_val = y[tr_ind:val_ind]
214.     X_test = X[val_ind:]
215.     y_test = y[val_ind:]
216.
217.     return X_tr, y_tr, X_val, y_val, X_test, y_test
218.
219. def vector1H(x, maxInd):

```

```

220.     out = np.zeros(maxInd)
221.     out[x] = 1
222.     return out
223. def mat1H2(y, maxInd):
224.     out = np.zeros((y.shape[0], maxInd))
225.     for i in range(y.shape[0]):
226.         out[i,:] = vector1H(y[i], maxInd)
227.     return out
228.
229. x_train, y_train, x_val, y_val, x_test, y_test = splitData(features, labels, 0.7, 0.2)
230.
231. from tensorflow.keras import layers, models, initializers, optimizers
232. model = models.Sequential()
233. model.add(layers.Dense(500, input_shape=(11,), activation='relu',\
234.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
235.                                distribution = 'normal'
236.                                , \
237.                                seed=None, \
238.                                scale=2.0),\
239.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
240.                                distribution = 'normal'
241.                                , \
242.                                seed=None, \
243.                                scale=2.0)))
244. model.add(layers.Dense(300,activation='relu',\
245.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
246.                                distribution = 'normal'
247.                                , \
248.                                seed=None, \
249.                                scale=2.0),\
250.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
251.                                distribution = 'normal'
252.                                , \
253.                                seed=None, \
254.                                scale=2.0)))
255. model.add(layers.Dense(150,activation='relu',\
256.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
257.                                distribution = 'normal'
258.                                , \
259.                                seed=None, \
260.                                scale=2.0)))
261. model.add(layers.Dense(65,activation='relu',\
262.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
263.                                distribution = 'normal'
264.                                , \
265.                                seed=None, \
266.                                scale=2.0),\
267.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
268.                                distribution = 'normal'
269.                                , \
270.                                seed=None, \
271.                                scale=2.0)))
272. model.add(layers.Dense(np.max(labels)+1, activation='softmax',\
273.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
274.                                distribution = 'normal'
275.                                , \
276.                                seed=None,\
277.                                scale=2.0),\
278.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
279.                                distribution = 'normal'
280.                                , \
281.                                seed=None,\
282.                                scale=2.0)))

```

```

276. seed=None, \
277. scale=2.0)))
278.
279. adamOP = optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=False)
280. model.compile(loss='sparse_categorical_crossentropy', optimizer=adamOP, metrics=['accuracy'])

281. hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)
282.
283. mlp = MLP()
284. mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
285. mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
286. mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
287. mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
288. mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax', \
289. np.sqrt(2/(np.max(labels)+1))))
290.
291. ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 25, 529, \
292. loss='ce', lrnRate=0.001)
293.
294. print(hist.history.keys())
295. plt.plot(ceErrs)
296. plt.plot(hist.history['val_loss'])
297. plt.title('Validation Cross-Entropy Losses w/Adam Optimizer')
298. plt.xlabel('Epoch')
299. plt.ylabel('CE Loss')
300. plt.legend(['Own Model', 'Keras Model'])
301. plt.show()
302.
303. plt.plot(trAccs)
304. plt.plot(valAccs)
305. plt.plot(valKAccs)
306. plt.title('Accuracies Own Model w/Adam Optimizer')
307. plt.xlabel('Epoch')
308. plt.ylabel('Accuracy (%)')
309. plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
310. plt.show()
311.
312. plt.plot(trAccs)
313. plt.plot(valAccs)
314. plt.plot(np.asarray(hist.history['accuracy'])*100)
315. plt.plot(np.asarray(hist.history['val_accuracy'])*100)
316. plt.title('Accuracies Own Model vs. Keras w/Adam Optimizer')
317. plt.xlabel('Epoch')
318. plt.ylabel('Accuracy (%)')
319. plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
Keras'])
320. plt.show()
321.
322. print(mlp)
323. np.save('valAcc_Adam.npy', valAccs)
324.
325. testPred = mlp.prediction(x_test.T)
326. testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
327. print(testAcc)

```

Appendix 1-D – AMSGrad

```

1. import matplotlib.pyplot as plt
2. import numpy as np
3.
4. class Layer:
5.     def __init__(self, inputDim, numNeurons, activation, std, mean=0):
6.         self.inputDim = inputDim
7.         self.numNeurons = numNeurons
8.         self.activation = activation

```

```

9.
10.         self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inp
utDim)
11.         self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
12.         self.weightsE = np.concatenate((self.weights, self.biases), axis=1)
13.
14.         self.delta = None
15.         self.error = None
16.         self.lastActiv = None
17.
18.         self.prevM = 0.0
19.         self.prevV_hat = 0.0
20.         self.prevV = 0.0
21.
22.
23.     def actFcn(self,x):
24.         if(self.activation == 'sigmoid'):
25.             expx = np.exp(x)
26.             return expx/(1+expx)
27.         elif(self.activation == 'softmax'):
28.             expx = np.exp(x - np.max(x))
29.             return expx/np.sum(expx, axis=0)
30.         elif(self.activation == 'tanh'):
31.             return np.tanh(x)
32.         elif(self.activation == 'relu'):
33.             out = np.maximum(0,x)
34.             return out
35.
36.     def activate(self, x):
37.         if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation ==
'relu':
38.             if(x.ndim == 1):
39.                 x = x.reshape(x.shape[0],1)
40.                 numSamples = x.shape[1]
41.                 tempInp = np.r_[x, [np.ones(numSamples)*1]]
42.                 self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
43.                 return self.lastActiv
44.
45.     def derActiv(self, x):
46.         if(self.activation == 'sigmoid'):
47.             return x*(1-x)
48.         elif(self.activation == 'softmax'):
49.             return x*(1-x)
50.         elif(self.activation == 'tanh'):
51.             return 1 - x**2;
52.         elif(self.activation == 'relu'):
53.             der = x
54.             der[x > 0] = 1
55.             der[x <= 0] = 0
56.             return der
57.
58.     def __repr__(self):
59.         return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " + str(self.numNeur
ons) + "\n Activation: " + self.activation
60.
61. class MLP:
62.     def __init__(self):
63.         self.layers = []
64.
65.     def addLayer(self, layer):
66.         self.layers.append(layer)
67.
68.     def forward(self, inp):
69.         out = inp
70.         for lyr in self.layers:
71.             out = lyr.activate(out)

```



```

72.         return out
73.
74.     def prediction(self, inp):
75.         out = self.forward(inp)
76.         if(out.ndim == 1):
77.             return np.argmax(out)
78.         return np.argmax(out, axis=0)
79.
80.     def topKaccuracy(self, inp, realOut, k):
81.         out = self.forward(inp)
82.         cnt = 0
83.         for i in range(out.shape[1]):
84.             topKind = (out[:,i]).argsort()[::-1][:k]
85.             if realOut[i] in topKind:
86.                 cnt += 1
87.         return cnt/out.shape[1]
88.
89.     def backProp(self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
90.         net_out = self.forward(inp)
91.         for i in reversed(range(len(self.layers))):
92.             lyr = self.layers[i]
93.             #outputLayer
94.             if(lyr == self.layers[-1]):
95.                 if(loss == 'mse'):
96.                     lyr.error = out - net_out
97.                     derMatrix = lyr.derActiv(lyr.lastActiv)
98.                     lyr.delta = derMatrix * lyr.error
99.                 elif(loss == 'ce' and lyr.activation == 'softmax'):
100.                    lyr.delta = net_out - out
101.                else:
102.                    assert('Cant do that')
103.            #hiddenLayer
104.            else:
105.                nextLyr = self.layers[i+1]
106.                nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
107.                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
108.                derMatrix = lyr.derActiv(lyr.lastActiv)
109.                lyr.delta = derMatrix * lyr.error
110.
111.            #update weights
112.            for i in range(len(self.layers)):
113.                lyr = self.layers[i]
114.                if(i == 0):
115.                    if(inp.ndim == 1):
116.                        inp = inp.reshape(inp.shape[0],1)
117.                        numSamples = inp.shape[1]
118.                        inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
119.                    else:
120.                        numSamples = self.layers[i - 1].lastActiv.shape[1]
121.                        inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]
122.
123.                grad = np.matmul(lyr.delta, inputToUse.T)/batchSize
124.
125.                M = (1-beta1)*grad + beta1*lyr.prevM
126.                V = (1-beta2)*np.power(grad,2)+ beta2*lyr.prevV
127.
128.                V_hat = np.maximum(V, lyr.prevV_hat)
129.
130.                lyr.weightsE -= (lrnRate*M)/(np.sqrt(V_hat) + 1e-7)
131.
132.                lyr.prevV_hat = V_hat
133.                lyr.prevM = M
134.                lyr.prevV = V
135.
136.     def train(self, inp, out, inpTest, outTest, epochNum, batchSize, loss, lrnRate=0.001, beta
1=0.9, beta2=0.999):

```

```

137.         errList = []
138.         trAccs = []
139.         valAccs = []
140.         valKAccs = []
141.
142.         for ep in range(epochNum):
143.             print('-----
\nEpoch', ep+1)
144.
145.             randomIndexes = np.random.permutation(len(inp))
146.             inp = inp[randomIndexes]
147.             out = out[randomIndexes]
148.             numBatches = int(np.floor(len(inp)/batchSize))
149.             for j in range(numBatches):
150.                 batch_inp = inp[batchSize*j:batchSize*j+batchSize]
151.                 batch_out = out[batchSize*j:batchSize*j+batchSize]
152.
153.                 batch_out_1H = mat1H2(batch_out, np.max(out)+1).T
154.
155.                 #self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
156.                 self.backProp(batch_inp.T, batch_out_1H, batchSize, loss, ep+1, lrnRate, beta1
, beta2)
157.
158.                 valOutput = self.forward(inpTest.T)
159.                 if(loss == 'ce'):
160.                     err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
.shape[1]
161.                 elif(loss == 'mse'):
162.                     err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
163.                 print(loss.upper() + ' Validation Error ', err)
164.                 errList.append(err)
165.
166.                 trPred = self.prediction(inp.T)
167.                 trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
168.                 print('Training Accuracy: ', trAcc)
169.                 trAccs.append(trAcc)
170.
171.                 valPred = self.prediction(inpTest.T)
172.                 valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]
*100
173.                 print('Validation Accuracy: ', valAcc)
174.                 valAccs.append(valAcc)
175.
176.                 #k is chosen as three
177.                 K = 3
178.                 topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
179.                 print('Top ' + str(K) + ' Accuracy: ', topKacc)
180.                 valKAccs.append(topKacc)
181.
182.
183.         return errList, trAccs, valAccs, valKAccs
184.
185.     def __repr__(self):
186.         retStr = ""
187.         for i, lyr in enumerate(self.layers):
188.             retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
189.         return retStr
190.
191. features = np.load('dataMLP.npy', allow_pickle=True)
192. labels = np.load('labelsMLP.npy', allow_pickle=True)
193. labels = labels.reshape((labels.shape[0],1))
194.
195. features -= np.mean(features, axis=0)
196. features /= np.std(features, axis=0)
197.
198. #three-way split

```

```

199. def splitData(X, y, tr, val):
200.     np.random.seed(456)
201.     shuffle = np.random.permutation(len(y))
202.     X = X[shuffle]
203.     y = y[shuffle]
204.
205.     tr_ind = int(np.floor(len(y)*tr))
206.     val_ind = int(np.floor(len(y)*(tr+val)))
207.
208.     X_tr = X[0:tr_ind]
209.     y_tr = y[0:tr_ind]
210.     X_val = X[tr_ind:val_ind]
211.     y_val = y[tr_ind:val_ind]
212.     X_test = X[val_ind:]
213.     y_test = y[val_ind:]
214.
215.     return X_tr, y_tr, X_val, y_val, X_test, y_test
216.
217. def vector1H(x, maxInd):
218.     out = np.zeros(maxInd)
219.     out[x] = 1
220.     return out
221. def mat1H2(y, maxInd):
222.     out = np.zeros((y.shape[0], maxInd))
223.     for i in range(y.shape[0]):
224.         out[i,:] = vector1H(y[i], maxInd)
225.     return out
226.
227. x_train, y_train, x_val, y_val, x_test, y_test = splitData(features, labels, 0.7, 0.2)
228.
229. from tensorflow.keras import layers, models, initializers, optimizers
230. model = models.Sequential()
231. model.add(layers.Dense(500, input_shape=(11,), activation='relu', \
232.                             kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
233.                                     distribution = 'normal'
234.                                     , \
235.                                     seed=None, \
236.                                     scale=2.0), \
237.                             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
238.                                     distribution = 'normal'
239.                                     , \
240.                                     seed=None, \
241.                                     scale=2.0)))
242. model.add(layers.Dense(300, activation='relu', \
243.                             kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
244.                                     distribution = 'normal'
245.                                     , \
246.                                     seed=None, \
247.                                     scale=2.0), \
248.                             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
249.                                     distribution = 'normal'
250.                                     , \
251.                                     seed=None, \
252.                                     scale=2.0)))
253. model.add(layers.Dense(150, activation='relu', \
254.                             kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
255.                                     distribution = 'normal'
256.                                     , \
257.                                     seed=None, \
258.                                     scale=2.0)))
259. model.add(layers.Dense(65, activation='relu', \

```

```

259.             kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
260.                                                         distribution = 'normal'
261.             , \
262.                                                         seed=None, \
263.                                                         scale=2.0),\
264.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
265.                                                         distribution = 'normal'
266.             , \
267.                                                         seed=None, \
268.                                                         scale=2.0)))
269. model.add(layers.Dense(np.max(labels)+1, activation='softmax',\
270.                         kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
271.                                                         distribution = 'normal'
272.                         , \
273.                                                         seed=None, \
274.                                                         scale=2.0),\
275.                         bias_initializer = initializers.VarianceScaling(mode='fan_out', \
276.                                                         distribution = 'normal'
277.                         , \
278.                                                         seed=None, \
279.                                                         scale=2.0)))
280.
281. adamOP = optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=True)
282. model.compile(loss='sparse_categorical_crossentropy', optimizer=adamOP, metrics=['accuracy'])
283.
284. hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)
285.
286. mlp = MLP()
287. mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
288. mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
289. mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
290. mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
291. mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
292.                 np.sqrt(2/(np.max(labels)+1))))
293.
294. ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 25, 529,\
295.                                             loss='ce', lrnRate=0.001)
296.
297. print(hist.history.keys())
298. plt.plot(ceErrs)
299. plt.plot(hist.history['val_loss'])
300. plt.title('Validation Cross-Entropy Losses w/AMSGrad')
301. plt.xlabel('Epoch')
302. plt.ylabel('CE Loss')
303. plt.legend(['Own Model', 'Keras Model'])
304. plt.show()
305.
306. plt.plot(trAccs)
307. plt.plot(valAccs)
308. plt.plot(valKAccs)
309. plt.title('Accuracies Own Model w/AMSGrad')
310. plt.xlabel('Epoch')
311. plt.ylabel('Accuracy (%)')
312. plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
313. plt.show()
314.
315. plt.plot(trAccs)
316. plt.plot(valAccs)
317. plt.plot(np.asarray(hist.history['accuracy'])*100)
318. plt.plot(np.asarray(hist.history['val_accuracy'])*100)
319. plt.title('Accuracies Own Model vs. Keras w/AMSGrad')
320. plt.xlabel('Epoch')
321. plt.ylabel('Accuracy (%)')
322. plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
323.             Keras'])
324. plt.show()

```

```

319.
320.np.save('valAcc_AMSGrad.npy', valAccs)
321.
322.testPred = mlp.prediction(x_test.T)
323.testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
324.print(testAcc)

```

Appendix 1-E – SGD with PCA

```

1. import matplotlib.pyplot as plt
2. import numpy as np
3.
4. class Layer:
5.     def __init__(self, inputDim, numNeurons, activation, std, mean=0):
6.         self.inputDim = inputDim
7.         self.numNeurons = numNeurons
8.         self.activation = activation
9.
10.        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inp
utDim)
11.        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
12.        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)
13.
14.        self.delta = None
15.        self.error = None
16.        self.lastActiv = None
17.
18.    def actFcn(self,x):
19.        if(self.activation == 'sigmoid'):
20.            expx = np.exp(x)
21.            return expx/(1+expx)
22.        elif(self.activation == 'softmax'):
23.            expx = np.exp(x - np.max(x))
24.            return expx/np.sum(expx, axis=0)
25.        elif(self.activation == 'tanh'):
26.            return np.tanh(x)
27.        elif(self.activation == 'relu'):
28.            out = np.maximum(0,x)
29.            return out
30.
31.    def activate(self, x):
32.        if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation ==
'relu':
33.            if(x.ndim == 1):
34.                x = x.reshape(x.shape[0],1)
35.                numSamples = x.shape[1]
36.                tempInp = np.r_[x, [np.ones(numSamples)*1]]
37.                self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
38.            return self.lastActiv
39.
40.    def derActiv(self, x):
41.        if(self.activation == 'sigmoid'):
42.            return x*(1-x)
43.        elif(self.activation == 'softmax'):
44.            return x*(1-x)
45.        elif(self.activation == 'tanh'):
46.            return 1 - x**2;
47.        elif(self.activation == 'relu'):
48.            der = x
49.            der[x > 0] = 1
50.            der[x <= 0] = 0
51.            return der
52.
53.    def __repr__(self):

```

```

54.         return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " + str(self.numNeurons) + "\n Activation: " + self.activation
55.
56. class MLP:
57.     def __init__(self):
58.         self.layers = []
59.
60.     def addLayer(self, layer):
61.         self.layers.append(layer)
62.
63.     def forward(self, inp):
64.         out = inp
65.         for lyr in self.layers:
66.             out = lyr.activate(out)
67.         return out
68.
69.     def prediction(self, inp):
70.         out = self.forward(inp)
71.         if(out.ndim == 1):
72.             return np.argmax(out)
73.         return np.argmax(out, axis=0)
74.
75.     def topKaccuracy(self, inp, realOut, k):
76.         out = self.forward(inp)
77.         cnt = 0
78.         for i in range(out.shape[1]):
79.             topKind = (out[:,i]).argsort()[::-1][:k]
80.             if realOut[i] in topKind:
81.                 cnt += 1
82.         return cnt/out.shape[1]
83.
84.     def backProp(self, inp, out, lrnRate, batchSize, loss):
85.         net_out = self.forward(inp)
86.         for i in reversed(range(len(self.layers))):
87.             lyr = self.layers[i]
88.             #outputLayer
89.             if(lyr == self.layers[-1]):
90.                 if(loss == 'mse'):
91.                     lyr.error = net_out - out
92.                     derMatrix = lyr.derActiv(lyr.lastActiv)
93.                     lyr.delta = derMatrix * lyr.error
94.                 elif(loss == 'ce' and lyr.activation == 'softmax'):
95.                     lyr.delta = net_out - out
96.                 else:
97.                     assert('Cant do that')
98.             #hiddenLayer
99.             else:
100.                 nextLyr = self.layers[i+1]
101.                 nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
102.                 lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
103.                 derMatrix = lyr.derActiv(lyr.lastActiv)
104.                 lyr.delta = derMatrix * lyr.error
105.
106.             #update weights
107.             for i in range(len(self.layers)):
108.                 lyr = self.layers[i]
109.                 if(i == 0):
110.                     if(inp.ndim == 1):
111.                         inp = inp.reshape(inp.shape[0],1)
112.                         numSamples = inp.shape[1]
113.                         inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
114.                 else:
115.                     numSamples = self.layers[i - 1].lastActiv.shape[1]
116.                     inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]
117.
118.                 update = (lrnRate * np.matmul(lyr.delta, inputToUse.T))/batchSize

```

```

119.         lyr.weightsE -= update
120.
121.     def train(self, inp, out, inpTest, outTest, lrnRate, epochNum, batchSize, loss):
122.         errList = []
123.         trAccs = []
124.         valAccs = []
125.         valKAccs = []
126.
127.         for ep in range(epochNum):
128.             print('-----
\nEpoch', ep+1)
129.
130.             randomIndexes = np.random.permutation(len(inp))
131.             inp = inp[randomIndexes]
132.             out = out[randomIndexes]
133.             numBatches = int(np.floor(len(inp)/batchSize))
134.
135.             for j in range(numBatches):
136.                 batch_inp = inp[batchSize*j:batchSize*j+batchSize]
137.                 batch_out = out[batchSize*j:batchSize*j+batchSize]
138.
139.                 batch_out_1H = mat1H2(batch_out, np.max(out)+1).T
140.
141.                 self.backProp(batch_inp.T, batch_out_1H, lrnRate, batchSize, loss)
142.
143.                 valOutput = self.forward(inpTest.T)
144.                 if(loss == 'ce'):
145.                     err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
.shape[1]
146.                 elif(loss == 'mse'):
147.                     err = np.sum((outTest.T - self.forward(inpTest.T))**2)/valOutput.shape[1]
148.                 print(loss.upper() + ' Validation Error ', err)
149.                 errList.append(err)
150.
151.                 trPred = self.prediction(inp.T)
152.                 trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
153.                 print('Training Accuracy: ', trAcc)
154.                 trAccs.append(trAcc)
155.
156.                 valPred = self.prediction(inpTest.T)
157.                 valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]
*100
158.                 print('Validation Accuracy: ', valAcc)
159.                 valAccs.append(valAcc)
160.
161.                 #k is chosen as three
162.                 K = 3
163.                 topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
164.                 print('Top ' + str(K) + ' Accuracy: ', topKacc)
165.                 valKAccs.append(topKacc)
166.
167.
168.             return errList, trAccs, valAccs, valKAccs
169.
170.     def __repr__(self):
171.         retStr = ""
172.         for i, lyr in enumerate(self.layers):
173.             retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
174.         return retStr
175.
176. features = np.load('dataMLP.npy', allow_pickle=True)
177. labels = np.load('labelsMLP.npy', allow_pickle=True)
178. labels = labels.reshape((labels.shape[0],1))
179.
180. def PCA(eigenvalues, eigenvectors, k):
181.     #argsort the eigenvalues in descending order

```

```

182. eigvals_ind = np.argsort(eigenvalues)[::-1]
183. #sort the eigenvalues and eigenvectors accordingly
184. eigenvalues = eigenvalues[eigvals_ind]
185. eigenvectors = eigenvectors[:,eigvals_ind]
186. #select k biggest eigenvectors
187. eigfcs = eigenvectors[:,k]
188. #get the transpose of the eigenmatrix
189. eigfcs = eigfcs.T
190. return eigfcs
191.
192. def reconstruct(data_mn, eigfcs, mean):
193.     #data==D, eigenfaces==E
194.     #reconstructed_data = (D*E')*E + mean
195.     proj = np.matmul(data_mn,eigfcs.T)
196.     proj = np.matmul(proj,eigfcs) + mean
197.     return proj
198.
199. mean = np.mean(features, axis=0)
200. data_mn = features - mean
201. data_mn /= np.std(data_mn, axis=0)
202.
203. cov_mat = np.matmul(data_mn.T,data_mn)
204. eigenvalues, eigenvectors = np.linalg.eigh(cov_mat)
205.
206. #explained variances
207. #argsort the eigenvalues in descending order
208. eigvals_ind = np.argsort(eigenvalues)[::-1]
209. #sort the eigenvalues and eigenvectors accordingly
210. eigenvalues = eigenvalues[eigvals_ind]
211. eigenvectors = eigenvectors[:,eigvals_ind]
212. k_vals = np.array([1,2,3,4,5,6,7,8,9,10,11])
213. tot_var = np.sum(eigenvalues)
214. vas = np.zeros(k_vals.shape[0])
215. for ind in range(k_vals.shape[0]):
216.     vas[ind] = np.sum(eigenvalues[:k_vals[ind]]/tot_var*100.0)
217. print("Percent Explained Variances")
218. for i in range(vas.shape[0]):
219.     print("PEV for k=" + str(k_vals[i]) + " ", vas[i])
220. plt.plot(k_vals,vas)
221. plt.xlabel("Number of Eigenvalues -- k")
222. plt.ylabel("Percent Explained Variance")
223. plt.grid()
224. plt.show()
225.
226. k = 8
227. pcs = PCA(eigenvalues,eigenvectors,k)
228. ext_features = reconstruct(data_mn,pcs,mean)
229. print(ext_features.shape)
230.
231. #three-way split
232. def splitData(X, y, tr, val):
233.     np.random.seed(456)
234.     shuffle = np.random.permutation(len(y))
235.     X = X[shuffle]
236.     y = y[shuffle]
237.
238.     tr_ind = int(np.floor(len(y)*tr))
239.     val_ind = int(np.floor(len(y)*(tr+val)))
240.
241.     X_tr = X[0:tr_ind]
242.     y_tr = y[0:tr_ind]
243.     X_val = X[tr_ind:val_ind]
244.     y_val = y[tr_ind:val_ind]
245.     X_test = X[val_ind:]
246.     y_test = y[val_ind:]
247.

```



```

248.     return X_tr, y_tr, X_val, y_val, X_test, y_test
249.
250. def vector1H(x, maxInd):
251.     out = np.zeros(maxInd)
252.     out[x] = 1
253.     return out
254. def mat1H2(y, maxInd):
255.     out = np.zeros((y.shape[0], maxInd))
256.     for i in range(y.shape[0]):
257.         out[i,:] = vector1H(y[i], maxInd)
258.     return out
259.
260. x_train, y_train, x_val, y_val, x_test, y_test = splitData(ext_features, labels, 0.7, 0.2)
261.
262. from tensorflow.keras import layers, models, initializers, optimizers
263. model = models.Sequential()
264. model.add(layers.Dense(500, input_shape=(11,), activation='relu',\
265.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
266.                                distribution = 'normal'
267.                                , \
268.                                seed=None, \
269.                                scale=2.0),\
270.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
271.                                distribution = 'normal'
272.                                , \
273.                                seed=None, \
274.                                scale=2.0)))
275. model.add(layers.Dense(300, activation='relu',\
276.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
277.                                distribution = 'normal'
278.                                , \
279.                                seed=None, \
280.                                scale=2.0),\
281.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
282.                                distribution = 'normal'
283.                                , \
284.                                seed=None, \
285.                                scale=2.0)))
286. model.add(layers.Dense(150, activation='relu',\
287.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
288.                                distribution = 'normal'
289.                                , \
290.                                seed=None, \
291.                                scale=2.0),\
292.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
293.                                distribution = 'normal'
294.                                , \
295.                                seed=None, \
296.                                scale=2.0),\
297.                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
298.                                distribution = 'normal'
299.                                , \
300.                                seed=None, \
301.                                scale=2.0)))
302. model.add(layers.Dense(np.max(labels)+1, activation='softmax',\
303.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
304.                                distribution = 'normal'

```

```

305.                 bias_initializer = initializers.VarianceScaling(mode='fan_out', \
306.                                                                distribution = 'normal'
307.                                                                , \
308.                                                                seed=None, \
309.                                                                scale=2.0)))
309.momOp = optimizers.SGD(lr=0.1, momentum=0, nesterov=False, decay=0)
310.model.compile(loss='sparse_categorical_crossentropy', optimizer=momOp, metrics=['accuracy'])
311.hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)
312.
313.#Final Version
314.mlp = MLP()
315.mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
316.mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
317.mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
318.mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
319.mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax', \
320.                  np.sqrt(2/(np.max(labels)+1))))
321.
322.ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 0.1, 25, 529, lo
    ss='ce')
323.
324.print(hist.history.keys())
325.plt.plot(ceErrs)
326.plt.plot(hist.history['val_loss'])
327.plt.title('Validation Cross-Entropy Losses w/SGD')
328.plt.xlabel('Epoch')
329.plt.ylabel('CE Loss')
330.plt.legend(['Own Model', 'Keras Model'])
331.plt.show()
332.
333.plt.plot(trAccs)
334.plt.plot(valAccs)
335.plt.plot(valKAccs)
336.plt.title('Accuracies Own Model w/SGD')
337.plt.xlabel('Epoch')
338.plt.ylabel('Accuracy (%)')
339.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
340.plt.show()
341.
342.plt.plot(trAccs)
343.plt.plot(valAccs)
344.plt.plot(np.asarray(hist.history['accuracy'])*100)
345.plt.plot(np.asarray(hist.history['val_accuracy'])*100)
346.plt.title('Accuracies Own Model vs. Keras w/SGD')
347.plt.xlabel('Epoch')
348.plt.ylabel('Accuracy (%)')
349.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
    Keras'])
350.plt.show()
351.
352.np.save('valAcc_SGD_pca.npy', valAccs)
353.testPred = mlp.prediction(x_test.T)
354.testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
355.print(testAcc)

```

Appendix 1-F – SGDM with PCA

```

1. import matplotlib.pyplot as plt
2. import numpy as np
3.
4. class Layer:
5.     def __init__(self, inputDim, numNeurons, activation, std, mean=0):
6.         self.inputDim = inputDim
7.         self.numNeurons = numNeurons
8.         self.activation = activation

```

```

9.
10.     self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inp
    utDim)
11.     self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
12.     self.weightsE = np.concatenate((self.weights, self.biases), axis=1)
13.
14.     self.delta = None
15.     self.error = None
16.     self.lastActiv = None
17.
18.     self.prevUpdate = 0
19.
20.
21.     def actFcn(self,x):
22.         if(self.activation == 'sigmoid'):
23.             expx = np.exp(x)
24.             return expx/(1+expx)
25.         elif(self.activation == 'softmax'):
26.             expx = np.exp(x - np.max(x))
27.             return expx/np.sum(expx, axis=0)
28.         elif(self.activation == 'tanh'):
29.             return np.tanh(x)
30.         elif(self.activation == 'relu'):
31.             out = np.maximum(0,x)
32.             return out
33.
34.     def activate(self, x):
35.         if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation ==
    'relu':
36.             if(x.ndim == 1):
37.                 x = x.reshape(x.shape[0],1)
38.                 numSamples = x.shape[1]
39.                 tempInp = np.r_[x, [np.ones(numSamples)*1]]
40.                 self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
41.                 return self.lastActiv
42.
43.     def derActiv(self, x):
44.         if(self.activation == 'sigmoid'):
45.             return x*(1-x)
46.         elif(self.activation == 'softmax'):
47.             return x*(1-x)
48.         elif(self.activation == 'tanh'):
49.             return 1 - x**2;
50.         elif(self.activation == 'relu'):
51.             der = x
52.             der[x > 0] = 1
53.             der[x <= 0] = 0
54.             return der
55.
56.     def __repr__(self):
57.         return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " + str(self.numNeur
    ons) + "\n Activation: " + self.activation
58.
59. class MLP:
60.     def __init__(self):
61.         self.layers = []
62.
63.     def addLayer(self, layer):
64.         self.layers.append(layer)
65.
66.     def forward(self, inp):
67.         out = inp
68.         for lyr in self.layers:
69.             out = lyr.activate(out)
70.         return out
71.

```

```

72.     def prediction(self, inp):
73.         out = self.forward(inp)
74.         if(out.ndim == 1):
75.             return np.argmax(out)
76.         return np.argmax(out, axis=0)
77.
78.     def topKaccuracy(self, inp, realOut, k):
79.         out = self.forward(inp)
80.         cnt = 0
81.         for i in range(out.shape[1]):
82.             topKind = (out[:,i]).argsort()[::-1][:k]
83.             if realOut[i] in topKind:
84.                 cnt += 1
85.         return cnt/out.shape[1]
86.
87.     def backProp(self, inp, out, lrnRate, momCoeff, batchSize, loss):
88.         net_out = self.forward(inp)
89.         for i in reversed(range(len(self.layers))):
90.             lyr = self.layers[i]
91.             #outputLayer
92.             if(lyr == self.layers[-1]):
93.                 if(loss == 'mse'):
94.                     lyr.error = out - net_out
95.                     derMatrix = lyr.derActiv(lyr.lastActiv)
96.                     lyr.delta = derMatrix * lyr.error
97.                 elif(loss == 'ce' and lyr.activation == 'softmax'):
98.                     lyr.delta = net_out - out
99.                 else:
100.                    assert('Cant do that')
101.            #hiddenLayer
102.            else:
103.                nextLyr = self.layers[i+1]
104.                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
105.                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
106.                derMatrix = lyr.derActiv(lyr.lastActiv)
107.                lyr.delta = derMatrix * lyr.error
108.
109.            #update weights
110.            for i in range(len(self.layers)):
111.                lyr = self.layers[i]
112.                if(i == 0):
113.                    if(inp.ndim == 1):
114.                        inp = inp.reshape(inp.shape[0],1)
115.                        numSamples = inp.shape[1]
116.                        inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
117.                    else:
118.                        numSamples = self.layers[i - 1].lastActiv.shape[1]
119.                        inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]
120.
121.                    update = (lrnRate * np.matmul(lyr.delta, inputToUse.T))/batchSize
122.                    lyr.weightsE -= (update + momCoeff * lyr.prevUpdate)
123.                    lyr.prevUpdate = update
124.
125.     def train(self, inp, out, inpTest, outTest, lrnRate, momCoeff, epochNum, batchSize, loss):
126.         errList = []
127.         trAccs = []
128.         valAccs = []
129.         valKAccs = []
130.
131.         for ep in range(epochNum):
132.             print('-----
\nEpoch', ep+1)
133.
134.             randomIndexes = np.random.permutation(len(inp))
135.             inp = inp[randomIndexes]

```

```

136.         out = out[randomIndexes]
137.         numBatches = int(np.floor(len(inp)/batchSize))
138.
139.         for j in range(numBatches):
140.             batch_inp = inp[batchSize*j:batchSize*j+batchSize]
141.             batch_out = out[batchSize*j:batchSize*j+batchSize]
142.
143.             batch_out_1H = mat1H2(batch_out, np.max(out)+1).T
144.
145.             self.backProp(batch_inp.T, batch_out_1H, lrnRate, momCoeff, batchSize, loss)
146.
147.             valOutput = self.forward(inpTest.T)
148.             if(loss == 'ce'):
149.                 err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
150.                 .shape[1]
151.             elif(loss == 'mse'):
152.                 err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
153.                 print(loss.upper() + ' Validation Error ', err)
154.                 errList.append(err)
155.
156.             trPred = self.prediction(inp.T)
157.             trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
158.             print('Training Accuracy: ', trAcc)
159.             trAccs.append(trAcc)
160.
161.             valPred = self.prediction(inpTest.T)
162.             valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]
163.             *100
164.             print('Validation Accuracy: ', valAcc)
165.             valAccs.append(valAcc)
166.
167.             #k is chosen as three
168.             K = 3
169.             topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
170.             print('Top ' + str(K) + ' Accuracy: ', topKacc)
171.             valKAccs.append(topKacc)
172.
173.         return errList, trAccs, valAccs, valKAccs
174.
175.     def __repr__(self):
176.         retStr = ""
177.         for i, lyr in enumerate(self.layers):
178.             retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
179.         return retStr
180.
181. features = np.load('dataMLP.npy', allow_pickle=True)
182. labels = np.load('labelsMLP.npy', allow_pickle=True)
183. labels = labels.reshape((labels.shape[0],1))
184.
185. def PCA(eigenvalues, eigenvectors, k):
186.     #argsort the eigenvalues in descending order
187.     eigvals_ind = np.argsort(eigenvalues)[::-1]
188.     #sort the eigenvalues and eigenvectors accordingly
189.     eigenvalues = eigenvalues[eigvals_ind]
190.     eigenvectors = eigenvectors[:,eigvals_ind]
191.     #select k biggest eigenvectors
192.     eigfcs = eigenvectors[:,k]
193.     #get the transpose of the eigenmatrix
194.     eigfcs = eigfcs.T
195.     return eigfcs
196.
197. def reconstruct(data_mn, eigfcs, mean):
198.     #data==D, eigenfaces==E
199.     #reconstructed_data = (D*E')*E + mean
200.     proj = np.matmul(data_mn,eigfcs.T)

```

```

200.     proj = np.matmul(proj,eigfcs) + mean
201.     return proj
202.
203.mean = np.mean(features, axis=0)
204.data_mn = features - mean
205.data_mn /= np.std(data_mn, axis=0)
206.
207.cov_mat = np.matmul(data_mn.T,data_mn)
208.eigenvalues, eigenvectors = np.linalg.eigh(cov_mat)
209.
210.#explained variances
211.#argsort the eigenvalues in descending order
212.eigvals_ind = np.argsort(eigenvalues)[::-1]
213.#sort the eigenvalues and eigenvectors accordingly
214.eigenvalues = eigenvalues[eigvals_ind]
215.eigenvectors = eigenvectors[:,eigvals_ind]
216.k_vals = np.array([1,2,3,4,5,6,7,8,9,10,11])
217.tot_var = np.sum(eigenvalues)
218.vas = np.zeros(k_vals.shape[0])
219.for ind in range(k_vals.shape[0]):
220.     vas[ind] = np.sum(eigenvalues[:,k_vals[ind]]/tot_var*100.0)
221.print("Percent Explained Variances")
222.for i in range(vas.shape[0]):
223.     print("PEV for k=" + str(k_vals[i]) + " ", vas[i])
224.plt.plot(k_vals,vas)
225.plt.xlabel("Number of Eigenvalues -- k")
226.plt.ylabel("Percent Explained Variance")
227.plt.grid()
228.plt.show()
229.
230.k = 8
231.pcs = PCA(eigenvalues,eigenvectors,k)
232.ext_features = reconstruct(data_mn,pcs,mean)
233.print(ext_features.shape)
234.
235.#three-way split
236.def splitData(X, y, tr, val):
237.     np.random.seed(456)
238.     shuffle = np.random.permutation(len(y))
239.     X = X[shuffle]
240.     y = y[shuffle]
241.
242.     tr_ind = int(np.floor(len(y)*tr))
243.     val_ind = int(np.floor(len(y)*(tr+val)))
244.
245.     X_tr = X[0:tr_ind]
246.     y_tr = y[0:tr_ind]
247.     X_val = X[tr_ind:val_ind]
248.     y_val = y[tr_ind:val_ind]
249.     X_test = X[val_ind:]
250.     y_test = y[val_ind:]
251.
252.     return X_tr, y_tr, X_val, y_val, X_test, y_test
253.
254.def vector1H(x, maxInd):
255.     out = np.zeros(maxInd)
256.     out[x] = 1
257.     return out
258.def mat1H2(y, maxInd):
259.     out = np.zeros((y.shape[0], maxInd))
260.     for i in range(y.shape[0]):
261.         out[i,:] = vector1H(y[i], maxInd)
262.     return out
263.
264.x_train, y_train, x_val, y_val, x_test, y_test = splitData(ext_features, labels, 0.7, 0.2)
265.

```

```

266. from tensorflow.keras import layers, models, initializers, optimizers
267. model = models.Sequential()
268. model.add(layers.Dense(500, input_shape=(11,),\
269.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
270.                                distribution = 'normal'
271.                                ,\
272.                                seed=None,\
273.                                scale=2.0),\
274.                        activation='relu'))
275. model.add(layers.Dense(300,\
276.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
277.                                distribution = 'normal'
278.                                ,\
279.                                seed=None,\
280.                                scale=2.0),\
281.                        activation='relu'))
282. model.add(layers.Dense(150,\
283.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
284.                                distribution = 'normal'
285.                                ,\
286.                                seed=None,\
287.                                scale=2.0),\
288.                        activation = 'relu'))
289. model.add(layers.Dense(65,\
290.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
291.                                distribution = 'normal'
292.                                ,\
293.                                seed=None,\
294.                                scale=2.0),\
295.                        activation = 'relu'))
296. model.add(layers.Dense(np.max(labels)+1,\
297.                        kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
298.                                distribution = 'normal'
299.                                ,\
300.                                seed=None,\
301.                                scale=2.0),\
302.                        activation='softmax'))
303. momOp = optimizers.SGD(lr=0.1, momentum=0.9, nesterov=False)
304. model.compile(loss='sparse_categorical_crossentropy', optimizer=momOp, metrics=['accuracy'])
305. hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)
306.
307. mlp = MLP()
308. mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
309. mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
310. mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
311. mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
312. mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
313.                  np.sqrt(2/(np.max(labels)+1))))
314.
315. ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, \
316.        0.1, 0.9 , 25, 529, loss='ce')
317.
318. print(hist.history.keys())
319. plt.plot(ceErrs)
320. plt.plot(hist.history['val_loss'])
321. plt.title('Validation Cross-Entropy Losses w/Momentum')
322. plt.xlabel('Epoch')
323. plt.ylabel('CE Loss')
324. plt.legend(['Own Model', 'Keras Model'])
325. plt.show()
326.
327. plt.plot(trAccs)
328. plt.plot(valAccs)
329. plt.plot(valKAccs)
330. plt.title('Accuracies Own Model w/Momentum')
331. plt.xlabel('Epoch')

```

```

327.plt.ylabel('Accuracy (%)')
328.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
329.plt.show()
330.
331.plt.plot(trAccs)
332.plt.plot(valAccs)
333.plt.plot(np.asarray(hist.history['accuracy'])*100)
334.plt.plot(np.asarray(hist.history['val_accuracy'])*100)
335.plt.title('Accuracies Own Model vs. Keras w/Momentum')
336.plt.xlabel('Epoch')
337.plt.ylabel('Accuracy (%)')
338.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
Keras'])
339.plt.show()
340.
341.np.save('valAcc_Mom_pca.npy', valAccs)
342.
343.testPred = mlp.prediction(x_test.T)
344.testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
345.print(testAcc)

```

Appendix 1-G – Adam with PCA

```

1. import matplotlib.pyplot as plt
2. import numpy as np
3.
4. class Layer:
5.     def __init__(self, inputDim, numNeurons, activation, std, mean=0):
6.         self.inputDim = inputDim
7.         self.numNeurons = numNeurons
8.         self.activation = activation
9.
10.        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inp
utDim)
11.        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
12.        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)
13.
14.        self.delta = None
15.        self.error = None
16.        self.lastActiv = None
17.
18.        self.prevM = 0.0
19.        self.prevV = 0.0
20.
21.
22.    def actFcn(self,x):
23.        if(self.activation == 'sigmoid'):
24.            expx = np.exp(x)
25.            return expx/(1+expx)
26.        elif(self.activation == 'softmax'):
27.            expx = np.exp(x - np.max(x))
28.            return expx/np.sum(expx, axis=0)
29.        elif(self.activation == 'tanh'):
30.            return np.tanh(x)
31.        elif(self.activation == 'relu'):
32.            out = np.maximum(0,x)
33.            return out
34.
35.    def activate(self, x):
36.        if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation ==
'relu':
37.            if(x.ndim == 1):
38.                x = x.reshape(x.shape[0],1)
39.                numSamples = x.shape[1]
40.                tempInp = np.r_[x, [np.ones(numSamples)*1]]

```



```

41.         self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
42.     return self.lastActiv
43.
44.     def derActiv(self, x):
45.         if(self.activation == 'sigmoid'):
46.             return x*(1-x)
47.         elif(self.activation == 'softmax'):
48.             return x*(1-x)
49.         elif(self.activation == 'tanh'):
50.             return 1 - x**2;
51.         elif(self.activation == 'relu'):
52.             der = x
53.             der[x > 0] = 1
54.             der[x <= 0] = 0
55.             return der
56.
57.     def __repr__(self):
58.         return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " + str(self.numNeurons) + "\n Activation: " + self.activation
59.
60. class MLP:
61.     def __init__(self):
62.         self.layers = []
63.
64.     def addLayer(self, layer):
65.         self.layers.append(layer)
66.
67.     def forward(self, inp):
68.         out = inp
69.         for lyr in self.layers:
70.             out = lyr.activate(out)
71.         return out
72.
73.     def prediction(self, inp):
74.         out = self.forward(inp)
75.         if(out.ndim == 1):
76.             return np.argmax(out)
77.         return np.argmax(out, axis=0)
78.
79.     def topKaccuracy(self, inp, realOut, k):
80.         out = self.forward(inp)
81.         cnt = 0
82.         for i in range(out.shape[1]):
83.             topKind = (out[:,i]).argsort()[::-1][:k]
84.             if realOut[i] in topKind:
85.                 cnt += 1
86.         return cnt/out.shape[1]
87.
88.     def backProp(self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
89.         net_out = self.forward(inp)
90.         for i in reversed(range(len(self.layers))):
91.             lyr = self.layers[i]
92.             #outputLayer
93.             if(lyr == self.layers[-1]):
94.                 if(loss == 'mse'):
95.                     lyr.error = out - net_out
96.                     derMatrix = lyr.derActiv(lyr.lastActiv)
97.                     lyr.delta = derMatrix * lyr.error
98.                 elif(loss == 'ce' and lyr.activation == 'softmax'):
99.                     lyr.delta = net_out - out
100.             else:
101.                 assert('Cant do that')
102.             #hiddenLayer
103.         else:
104.             nextLyr = self.layers[i+1]
105.             nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]

```

```

106.         lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
107.         derMatrix = lyr.derActiv(lyr.lastActiv)
108.         lyr.delta = derMatrix * lyr.error
109.
110.         #update weights
111.         for i in range(len(self.layers)):
112.             lyr = self.layers[i]
113.             if(i == 0):
114.                 if(inp.ndim == 1):
115.                     inp = inp.reshape(inp.shape[0],1)
116.                     numSamples = inp.shape[1]
117.                     inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
118.             else:
119.                 numSamples = self.layers[i - 1].lastActiv.shape[1]
120.                 inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]
121.
122.             grad = np.matmul(lyr.delta, inputToUse.T)/batchSize
123.
124.             M = (1-beta1)*grad + beta1*lyr.prevM
125.             V = (1-beta2)*np.power(grad,2)+ beta2*lyr.prevV
126.
127.             alpha_t = lrnRate * np.sqrt(1-np.power(beta2,curEpoch))/(1-
np.power(beta1,curEpoch))
128.             lyr.weightsE -= alpha_t * M / (np.sqrt(V) + 1e-7)
129.             '''
130.             Algorithm 1 from Kingma and Ba
131.             M_hat = M/(1- np.power(beta1,curEpoch))
132.             V_hat = V/(1- np.power(beta2,curEpoch))
133.
134.             lyr.weightsE -= (lrnRate*M_hat)/(np.sqrt(V_hat) + 1e-7)'''
135.
136.             lyr.prevM = M
137.             lyr.prevV = V
138.
139.         def train(self, inp, out, inpTest, outTest, epochNum, batchSize, loss, lrnRate=0.001, beta
1=0.9, beta2=0.999):
140.             errList = []
141.             trAccs = []
142.             valAccs = []
143.             valKAccs = []
144.
145.             for ep in range(epochNum):
146.                 print('-----
\nEpoch', ep+1)
147.
148.                 randomIndexes = np.random.permutation(len(inp))
149.                 inp = inp[randomIndexes]
150.                 out = out[randomIndexes]
151.                 numBatches = int(np.floor(len(inp)/batchSize))
152.                 for j in range(numBatches):
153.                     batch_inp = inp[batchSize*j:batchSize*j+batchSize]
154.                     batch_out = out[batchSize*j:batchSize*j+batchSize]
155.
156.                     batch_out_1H = mat1H2(batch_out, np.max(out)+1).T
157.
158.                     #self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
159.                     self.backProp(batch_inp.T, batch_out_1H, batchSize, loss, ep+1, lrnRate, beta1
, beta2)
160.
161.                     valOutput = self.forward(inpTest.T)
162.                     if(loss == 'ce'):
163.                         err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
.shape[1]
164.                     elif(loss == 'mse'):
165.                         err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
166.                     print(loss.upper() + ' Validation Error ', err)

```

```

167.         errList.append(err)
168.
169.         trPred = self.prediction(inp.T)
170.         trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
171.         print('Training Accuracy: ', trAcc)
172.         trAccs.append(trAcc)
173.
174.         valPred = self.prediction(inpTest.T)
175.         valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]
        *100
176.         print('Validation Accuracy: ', valAcc)
177.         valAccs.append(valAcc)
178.
179.         #k is chosen as three
180.         K = 3
181.         topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
182.         print('Top ' + str(K) + ' Accuracy: ', topKacc)
183.         valKAccs.append(topKacc)
184.
185.
186.         return errList, trAccs, valAccs, valKAccs
187.
188.     def __repr__(self):
189.         retStr = ""
190.         for i, lyr in enumerate(self.layers):
191.             retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
192.         return retStr
193.
194. features = np.load('dataMLP.npy', allow_pickle=True)
195. labels = np.load('labelsMLP.npy', allow_pickle=True)
196. labels = labels.reshape((labels.shape[0],1))
197.
198. def PCA(eigenvalues, eigenvectors, k):
199.     #argsort the eigenvalues in descending order
200.     eigvals_ind = np.argsort(eigenvalues)[::-1]
201.     #sort the eigenvalues and eigenvectors accordingly
202.     eigenvalues = eigenvalues[eigvals_ind]
203.     eigenvectors = eigenvectors[:,eigvals_ind]
204.     #select k biggest eigenvectors
205.     eigfcs = eigenvectors[:, :k]
206.     #get the transpose of the eigenmatrix
207.     eigfcs = eigfcs.T
208.     return eigfcs
209.
210. def reconstruct(data_mn, eigfcs, mean):
211.     #data==D, eigenfaces==E
212.     #reconstructed_data = (D*E')*E + mean
213.     proj = np.matmul(data_mn,eigfcs.T)
214.     proj = np.matmul(proj,eigfcs) + mean
215.     return proj
216.
217. mean = np.mean(features, axis=0)
218. data_mn = features - mean
219. data_mn /= np.std(data_mn, axis=0)
220.
221. cov_mat = np.matmul(data_mn.T,data_mn)
222. eigenvalues, eigenvectors = np.linalg.eigh(cov_mat)
223.
224. #explained variances
225. #argsort the eigenvalues in descending order
226. eigvals_ind = np.argsort(eigenvalues)[::-1]
227. #sort the eigenvalues and eigenvectors accordingly
228. eigenvalues = eigenvalues[eigvals_ind]
229. eigenvectors = eigenvectors[:,eigvals_ind]
230. k_vals = np.array([1,2,3,4,5,6,7,8,9,10,11])
231. tot_var = np.sum(eigenvalues)

```

```

232.vas = np.zeros(k_vals.shape[0])
233.for ind in range(k_vals.shape[0]):
234.    vas[ind] = np.sum(eigenvalues[:k_vals[ind]]/tot_var*100.0)
235.print("Percent Explained Variances")
236.for i in range(vas.shape[0]):
237.    print("PEV for k=" + str(k_vals[i]) + " ", vas[i])
238.plt.plot(k_vals,vas)
239.plt.xlabel("Number of Eigenvalues -- k")
240.plt.ylabel("Percent Explained Variance")
241.plt.grid()
242.plt.show()
243.
244.k = 8
245.pcs = PCA(eigenvalues,eigenvectors,k)
246.ext_features = reconstruct(data_mn,pcs,mean)
247.print(ext_features.shape)
248.
249.#three-way split
250.def splitData(X, y, tr, val):
251.    np.random.seed(456)
252.    shuffle = np.random.permutation(len(y))
253.    X = X[shuffle]
254.    y = y[shuffle]
255.
256.    tr_ind = int(np.floor(len(y)*tr))
257.    val_ind = int(np.floor(len(y)*(tr+val)))
258.
259.    X_tr = X[0:tr_ind]
260.    y_tr = y[0:tr_ind]
261.    X_val = X[tr_ind:val_ind]
262.    y_val = y[tr_ind:val_ind]
263.    X_test = X[val_ind:]
264.    y_test = y[val_ind:]
265.
266.    return X_tr, y_tr, X_val, y_val, X_test, y_test
267.
268.def vector1H(x, maxInd):
269.    out = np.zeros(maxInd)
270.    out[x] = 1
271.    return out
272.def mat1H2(y, maxInd):
273.    out = np.zeros((y.shape[0], maxInd))
274.    for i in range(y.shape[0]):
275.        out[i,:] = vector1H(y[i], maxInd)
276.    return out
277.
278.x_train, y_train, x_val, y_val, x_test, y_test = splitData(ext_features, labels, 0.7, 0.2)
279.
280.from tensorflow.keras import layers, models, initializers, optimizers
281.model = models.Sequential()
282.model.add(layers.Dense(500, input_shape=(11,), activation='relu',\
283.    kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
284.    distribution = 'normal'
285.    , \
286.    seed=None, \
287.    scale=2.0),\
288.    bias_initializer = initializers.VarianceScaling(mode='fan_out', \
289.    distribution = 'normal'
290.    , \
291.    seed=None, \
292.    scale=2.0)))
293.model.add(layers.Dense(300,activation='relu',\
294.    kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
295.    distribution = 'normal'
296.    , \
297.    seed=None, \
298.    scale=2.0)))

```

```

295.                                     scale=2.0),\
296.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
297.                                     distribution = 'normal'
298.             , \
299.                                     seed=None, \
300.                                     scale=2.0)))
301.model.add(layers.Dense(150,activation='relu',\
302.             kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
303.                                     distribution = 'normal'
304.             , \
305.                                     seed=None, \
306.                                     scale=2.0),\
307.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
308.                                     distribution = 'normal'
309.             , \
310.                                     seed=None, \
311.                                     scale=2.0)))
312.model.add(layers.Dense(65,activation='relu',\
313.             kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
314.                                     distribution = 'normal'
315.             , \
316.                                     seed=None, \
317.                                     scale=2.0),\
318.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
319.                                     distribution = 'normal'
320.             , \
321.                                     seed=None, \
322.                                     scale=2.0),\
323.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
324.                                     distribution = 'normal'
325.             , \
326.                                     seed=None, \
327.                                     scale=2.0)))
328.adamOP = optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=False)
329.model.compile(loss='sparse_categorical_crossentropy', optimizer=adamOP, metrics=['accuracy'])

330.hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)
331.
332.mlp = MLP()
333.mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
334.mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
335.mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
336.mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
337.mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
338.             np.sqrt(2/(np.max(labels)+1))))
339.
340.ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 25, 529,\
341.             loss='ce', lrnRate=0.001)
342.
343.print(hist.history.keys())
344.plt.plot(ceErrs)
345.plt.plot(hist.history['val_loss'])
346.plt.title('Validation Cross-Entropy Losses w/Adam Optimizer')
347.plt.xlabel('Epoch')
348.plt.ylabel('CE Loss')
349.plt.legend(['Own Model', 'Keras Model'])
350.plt.show()
351.
352.plt.plot(trAccs)

```

```

353.plt.plot(valAccs)
354.plt.plot(valKAccs)
355.plt.title('Accuracies Own Model w/Adam Optimizer')
356.plt.xlabel('Epoch')
357.plt.ylabel('Accuracy (%)')
358.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
359.plt.show()
360.
361.plt.plot(trAccs)
362.plt.plot(valAccs)
363.plt.plot(np.asarray(hist.history['accuracy'])*100)
364.plt.plot(np.asarray(hist.history['val_accuracy'])*100)
365.plt.title('Accuracies Own Model vs. Keras w/Adam Optimizer')
366.plt.xlabel('Epoch')
367.plt.ylabel('Accuracy (%)')
368.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
Keras'])
369.plt.show()
370.
371.print(mlp)
372.np.save('valAcc_Adam_pca.npy', valAccs)
373.testPred = mlp.prediction(x_test.T)
374.testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
375.print(testAcc)

```

Appendix 1-H – AMSGrad with PCA

```

1. import matplotlib.pyplot as plt
2. import numpy as np
3.
4. class Layer:
5.     def __init__(self, inputDim, numNeurons, activation, std, mean=0):
6.         self.inputDim = inputDim
7.         self.numNeurons = numNeurons
8.         self.activation = activation
9.
10.        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inp
utDim)
11.        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
12.        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)
13.
14.        self.delta = None
15.        self.error = None
16.        self.lastActiv = None
17.
18.        self.prevM = 0.0
19.        self.prevV_hat = 0.0
20.        self.prevV = 0.0
21.
22.
23.    def actFcn(self,x):
24.        if(self.activation == 'sigmoid'):
25.            expx = np.exp(x)
26.            return expx/(1+expx)
27.        elif(self.activation == 'softmax'):
28.            expx = np.exp(x - np.max(x))
29.            return expx/np.sum(expx, axis=0)
30.        elif(self.activation == 'tanh'):
31.            return np.tanh(x)
32.        elif(self.activation == 'relu'):
33.            out = np.maximum(0,x)
34.            return out
35.
36.    def activate(self, x):

```

```

37.         if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation ==
'relu':
38.             if(x.ndim == 1):
39.                 x = x.reshape(x.shape[0],1)
40.                 numSamples = x.shape[1]
41.                 tempInp = np.r_[x, [np.ones(numSamples)*1]]
42.                 self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
43.             return self.lastActiv
44.
45.     def derActiv(self, x):
46.         if(self.activation == 'sigmoid'):
47.             return x*(1-x)
48.         elif(self.activation == 'softmax'):
49.             return x*(1-x)
50.         elif(self.activation == 'tanh'):
51.             return 1 - x**2;
52.         elif(self.activation == 'relu'):
53.             der = x
54.             der[x > 0] = 1
55.             der[x <= 0] = 0
56.             return der
57.
58.     def __repr__(self):
59.         return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " + str(self.numNeur
ons) + "\n Activation: " + self.activation
60.
61. class MLP:
62.     def __init__(self):
63.         self.layers = []
64.
65.     def addLayer(self, layer):
66.         self.layers.append(layer)
67.
68.     def forward(self, inp):
69.         out = inp
70.         for lyr in self.layers:
71.             out = lyr.activate(out)
72.         return out
73.
74.     def prediction(self, inp):
75.         out = self.forward(inp)
76.         if(out.ndim == 1):
77.             return np.argmax(out)
78.         return np.argmax(out, axis=0)
79.
80.     def topKaccuracy(self, inp, realOut, k):
81.         out = self.forward(inp)
82.         cnt = 0
83.         for i in range(out.shape[1]):
84.             topKind = (out[:,i]).argsort()[::-1][:k]
85.             if realOut[i] in topKind:
86.                 cnt += 1
87.         return cnt/out.shape[1]
88.
89.     def backProp(self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
90.         net_out = self.forward(inp)
91.         for i in reversed(range(len(self.layers))):
92.             lyr = self.layers[i]
93.             #outputLayer
94.             if(lyr == self.layers[-1]):
95.                 if(loss == 'mse'):
96.                     lyr.error = out - net_out
97.                     derMatrix = lyr.derActiv(lyr.lastActiv)
98.                     lyr.delta = derMatrix * lyr.error
99.                 elif(loss == 'ce' and lyr.activation == 'softmax'):
100.                     lyr.delta = net_out - out

```

```

101.         else:
102.             assert('Cant do that')
103.         #hiddenLayer
104.         else:
105.             nextLyr = self.layers[i+1]
106.             nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
107.             lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
108.             derMatrix = lyr.derActiv(lyr.lastActiv)
109.             lyr.delta = derMatrix * lyr.error
110.
111.         #update weights
112.         for i in range(len(self.layers)):
113.             lyr = self.layers[i]
114.             if(i == 0):
115.                 if(inp.ndim == 1):
116.                     inp = inp.reshape(inp.shape[0],1)
117.                     numSamples = inp.shape[1]
118.                     inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
119.                 else:
120.                     numSamples = self.layers[i - 1].lastActiv.shape[1]
121.                     inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]
122.
123.             grad = np.matmul(lyr.delta, inputToUse.T)/batchSize
124.
125.             M = (1-beta1)*grad + beta1*lyr.prevM
126.             V = (1-beta2)*np.power(grad,2)+ beta2*lyr.prevV
127.
128.             V_hat = np.maximum(V, lyr.prevV_hat)
129.
130.             lyr.weightsE -= (lrnRate*M)/(np.sqrt(V_hat) + 1e-7)
131.
132.             lyr.prevV_hat = V_hat
133.             lyr.prevM = M
134.             lyr.prevV = V
135.
136.         def train(self, inp, out, inpTest, outTest, epochNum, batchSize, loss, lrnRate=0.001, beta
137.             1=0.9, beta2=0.999):
138.             errList = []
139.             trAccs = []
140.             valAccs = []
141.             valKAccs = []
142.
143.             for ep in range(epochNum):
144.                 print('-----
145.                 \nEpoch', ep+1)
146.
147.                 randomIndexes = np.random.permutation(len(inp))
148.                 inp = inp[randomIndexes]
149.                 out = out[randomIndexes]
150.                 numBatches = int(np.floor(len(inp)/batchSize))
151.                 for j in range(numBatches):
152.                     batch_inp = inp[batchSize*j:batchSize*j+batchSize]
153.                     batch_out = out[batchSize*j:batchSize*j+batchSize]
154.
155.                     batch_out_1H = mat1H2(batch_out, np.max(out)+1).T
156.
157.                     #self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
158.                     self.backProp(batch_inp.T, batch_out_1H, batchSize, loss, ep+1, lrnRate, beta1
159.                         , beta2)
160.
161.                     valOutput = self.forward(inpTest.T)
162.                     if(loss == 'ce'):
163.                         err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
164.                         .shape[1]
165.                     elif(loss == 'mse'):
166.                         err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val

```



```

163.         print(loss.upper() + ' Validation Error ', err)
164.         errList.append(err)
165.
166.         trPred = self.prediction(inp.T)
167.         trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
168.         print('Training Accuracy: ', trAcc)
169.         trAccs.append(trAcc)
170.
171.         valPred = self.prediction(inpTest.T)
172.         valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]
173.         *100
174.         print('Validation Accuracy: ', valAcc)
175.         valAccs.append(valAcc)
176.
177.         #k is chosen as three
178.         K = 3
179.         topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
180.         print('Top ' + str(K) + ' Accuracy: ', topKacc)
181.         valKAccs.append(topKacc)
182.
183.         return errList, trAccs, valAccs, valKAccs
184.
185.     def __repr__(self):
186.         retStr = ""
187.         for i, lyr in enumerate(self.layers):
188.             retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
189.         return retStr
190.
191. features = np.load('dataMLP.npy', allow_pickle=True)
192. labels = np.load('labelsMLP.npy', allow_pickle=True)
193. labels = labels.reshape((labels.shape[0],1))
194.
195. def PCA(eigenvalues, eigenvectors, k):
196.     #argsort the eigenvalues in descending order
197.     eigvals_ind = np.argsort(eigenvalues)[::-1]
198.     #sort the eigenvalues and eigenvectors accordingly
199.     eigenvalues = eigenvalues[eigvals_ind]
200.     eigenvectors = eigenvectors[:,eigvals_ind]
201.     #select k biggest eigenvectors
202.     eigfcs = eigenvectors[:, :k]
203.     #get the transpose of the eigenmatrix
204.     eigfcs = eigfcs.T
205.     return eigfcs
206.
207. def reconstruct(data_mn, eigfcs, mean):
208.     #data==D, eigenfaces==E
209.     #reconstructed_data = (D*E')*E + mean
210.     proj = np.matmul(data_mn,eigfcs.T)
211.     proj = np.matmul(proj,eigfcs) + mean
212.     return proj
213.
214. mean = np.mean(features, axis=0)
215. data_mn = features - mean
216. data_mn /= np.std(data_mn, axis=0)
217.
218. cov_mat = np.matmul(data_mn.T,data_mn)
219. eigenvalues, eigenvectors = np.linalg.eigh(cov_mat)
220.
221. #explained variances
222. #argsort the eigenvalues in descending order
223. eigvals_ind = np.argsort(eigenvalues)[::-1]
224. #sort the eigenvalues and eigenvectors accordingly
225. eigenvalues = eigenvalues[eigvals_ind]
226. eigenvectors = eigenvectors[:,eigvals_ind]
227. k_vals = np.array([1,2,3,4,5,6,7,8,9,10,11])

```

```

228.tot_var = np.sum(eigenvalues)
229.vas = np.zeros(k_vals.shape[0])
230.for ind in range(k_vals.shape[0]):
231.    vas[ind] = np.sum(eigenvalues[:k_vals[ind]]/tot_var*100.0)
232.print("Percent Explained Variances")
233.for i in range(vas.shape[0]):
234.    print("PEV for k=" + str(k_vals[i]) + " ", vas[i])
235.plt.plot(k_vals,vas)
236.plt.xlabel("Number of Eigenvalues -- k")
237.plt.ylabel("Percent Explained Variance")
238.plt.grid()
239.plt.show()
240.
241.k = 8
242.pcs = PCA(eigenvalues,eigenvectors,k)
243.ext_features = reconstruct(data_mn,pcs,mean)
244.print(ext_features.shape)
245.
246.#three-way split
247.def splitData(X, y, tr, val):
248.    np.random.seed(456)
249.    shuffle = np.random.permutation(len(y))
250.    X = X[shuffle]
251.    y = y[shuffle]
252.
253.    tr_ind = int(np.floor(len(y)*tr))
254.    val_ind = int(np.floor(len(y)*(tr+val)))
255.
256.    X_tr = X[0:tr_ind]
257.    y_tr = y[0:tr_ind]
258.    X_val = X[tr_ind:val_ind]
259.    y_val = y[tr_ind:val_ind]
260.    X_test = X[val_ind:]
261.    y_test = y[val_ind:]
262.
263.    return X_tr, y_tr, X_val, y_val, X_test, y_test
264.
265.def vector1H(x, maxInd):
266.    out = np.zeros(maxInd)
267.    out[x] = 1
268.    return out
269.def mat1H2(y, maxInd):
270.    out = np.zeros((y.shape[0], maxInd))
271.    for i in range(y.shape[0]):
272.        out[i,:] = vector1H(y[i], maxInd)
273.    return out
274.
275.x_train, y_train, x_val, y_val, x_test, y_test = splitData(ext_features, labels, 0.7, 0.2)
276.
277.from tensorflow.keras import layers, models, initializers, optimizers
278.model = models.Sequential()
279.model.add(layers.Dense(500, input_shape=(11,), activation='relu',\
280.                                kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
281.                                    distribution = 'normal'
282.                                , \
283.                                    seed=None, \
284.                                    scale=2.0),\
285.                                bias_initializer = initializers.VarianceScaling(mode='fan_out', \
286.                                    distribution = 'normal'
287.                                , \
288.                                    seed=None, \
289.                                    scale=2.0)))
288.model.add(layers.Dense(300,activation='relu',\
289.                                kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
290.                                    distribution = 'normal'
291.                                , \

```

```

291.                                     seed=None, \
292.                                     scale=2.0),\
293.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
294.                                                         distribution = 'normal'
295.             , \
296.             seed=None, \
297.             scale=2.0)))
298. model.add(layers.Dense(150,activation='relu',\
299.             kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
300.                                                         distribution = 'normal'
301.             , \
302.             seed=None, \
303.             scale=2.0),\
304.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
305.                                                         distribution = 'normal'
306.             , \
307.             seed=None, \
308.             scale=2.0)))
309. model.add(layers.Dense(65,activation='relu',\
310.             kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
311.                                                         distribution = 'normal'
312.             , \
313.             seed=None, \
314.             scale=2.0),\
315.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
316.                                                         distribution = 'normal'
317.             , \
318.             seed=None, \
319.             scale=2.0),\
320.             bias_initializer = initializers.VarianceScaling(mode='fan_out', \
321.                                                         distribution = 'normal'
322.             , \
323.             seed=None, \
324.             scale=2.0)))
325. adamOP = optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=True)
326. model.compile(loss='sparse_categorical_crossentropy', optimizer=adamOP, metrics=['accuracy'])

327. hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)
328.
329. mlp = MLP()
330. mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
331. mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
332. mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
333. mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
334. mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
335.                 np.sqrt(2/(np.max(labels)+1))))
336.
337. ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 25, 529,\
338.                                             loss='ce', lrnRate=0.001)
339.
340. print(hist.history.keys())
341. plt.plot(ceErrs)
342. plt.plot(hist.history['val_loss'])
343. plt.title('Validation Cross-Entropy Losses w/AMSGrad')
344. plt.xlabel('Epoch')
345. plt.ylabel('CE Loss')
346. plt.legend(['Own Model', 'Keras Model'])
347. plt.show()
348.

```

```

349.plt.plot(trAccs)
350.plt.plot(valAccs)
351.plt.plot(valKAccs)
352.plt.title('Accuracies Own Model w/AMSGrad')
353.plt.xlabel('Epoch')
354.plt.ylabel('Accuracy (%)')
355.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
356.plt.show()
357.
358.plt.plot(trAccs)
359.plt.plot(valAccs)
360.plt.plot(np.asarray(hist.history['accuracy'])*100)
361.plt.plot(np.asarray(hist.history['val_accuracy'])*100)
362.plt.title('Accuracies Own Model vs. Keras w/AMSGrad')
363.plt.xlabel('Epoch')
364.plt.ylabel('Accuracy (%)')
365.plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
Keras'])
366.plt.show()
367.
368.np.save('valAcc_AMSGrad_pca.npy', valAccs)
369.testPred = mlp.prediction(x_test.T)
370.testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
371.print(testAcc)

```

Appendix 2 – Random Forest

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3. import h5py
4. import pandas as pd
5. import math
6.
7. features_cont = pd.read_pickle('dataRF_cont.pkl')
8. features_disc = pd.read_pickle('dataRF_disc.pkl')
9. features = pd.read_pickle('dataRF.pkl')
10. labels = np.load('labelsRF.npy', allow_pickle=True)
11. print(features.shape)
12. features.head()
13. print(labels)
14. np.random.seed()
15.
16. class Random_Forest():
17.     def __init__(self, x, y, n_trees, n_features, sample_size, max_depth=10, min_leaf=5):
18.         self.x = x
19.         self.y = y
20.         self.n_features = n_features
21.         self.sample_size = sample_size
22.         self.max_depth = max_depth
23.         self.min_leaf = min_leaf
24.         self.trees = [self.plant_tree(i) for i in range(n_trees)]
25.
26.     def plant_tree(self, tree_num):
27.         #print('Tree Number ' + str(tree_num + 1) + ' is being created.')
28.         indices = np.random.permutation(len(self.y))[:self.sample_size]
29.         f_indices = np.random.permutation(self.x.shape[1])[:self.n_features]
30.         return Decision_Tree(self.x.iloc[indices], self.y[indices], self.n_features, f_indices
31.
32.                                     , indices=np.array(range(self.sample_size)),
33.                                     max_depth = self.max_depth, min_leaf=self.min_leaf)
34.
35.     def predict(self, x):
36.         predictor = np.array([t.predict(x) for t in self.trees])
37.         return predictor
38.
39. class Decision_Tree():
40.     def __init__(self, x, y, n_features, f_indices, indices, max_depth=10, min_leaf=5):
41.         self.x = x
42.         self.y = y
43.         self.n_features = n_features
44.         self.f_indices = f_indices
45.         self.indices = indices
46.         self.max_depth = max_depth
47.         self.min_leaf = min_leaf
48.
49.         self.n_tot, self.f_tot = len(indices), x.shape[1]
50.
51.         counts = np.bincount(y[indices])
52.         self.value = np.argmax(counts)
53.         self.split_index = None
54.         self.score = 0
55.         self.do_split()
56.
57.     def do_split(self):
58.         for i in self.f_indices:
59.             self.find_best_split(i)
60.         if self.is_leaf:
61.             return
62.         x = self.split_point
```

```

63.
64.     left = np.nonzero(x<=self.split)[0]
65.     right = np.nonzero(x>self.split)[0]
66.     lf_indices = np.random.permutation(self.x.shape[1]):self.n_features]
67.     rf_indices = np.random.permutation(self.x.shape[1]):self.n_features]
68.     self.lhs = Decision_Tree(self.x, self.y, self.n_features, lf_indices,
69.                             self.indices[left], max_depth=self.max_depth-
1, min_leaf=self.min_leaf)
70.     self.rhs = Decision_Tree(self.x, self.y, self.n_features, rf_indices,
71.                             self.indices[right], max_depth=self.max_depth-
1, min_leaf=self.min_leaf)
72.
73.     def find_best_split(self, split_index):
74.
75.         x = self.x.values[self.indices,split_index]
76.         y = self.y[self.indices]
77.         sort_index = np.argsort(x)
78.         sort_x = x[sort_index]
79.         sort_y = y[sort_index]
80.
81.         right_node = y
82.         left_node = []
83.         left_node = np.asarray(left_node)
84.
85.         for i in range(0,self.n_tot-self.min_leaf-1):
86.             xi = sort_x[i]
87.             yi = sort_y[i]
88.             #xi, yi = x[i], y[i]
89.             np.append(left_node, right_node[0:1])
90.             right_node = right_node[1:]
91.             if i < self.min_leaf or xi == sort_x[i+1]:
92.                 continue
93.
94.             IG = self.information_gain(y, right_node, left_node)
95.             if IG > self.score:
96.                 self.split_index = split_index
97.                 self.score = IG
98.                 self.split = xi
99.
100.
101.     def information_gain(self, y, r_node, l_node):
102.         groups = [l_node, r_node]
103.         Number_all = len(y)
104.         IG = self.gini(y)
105.         for group in groups:
106.             IG -= self.gini(group)*len(group)/Number_all
107.         return IG
108.
109.     def gini(self, y):
110.         class_ids = [i for i in range(23)]
111.         Number_group = len(y)
112.         if Number_group == 0:
113.             return 0
114.
115.         sum_of_classes = 0.
116.         for class_id in class_ids:
117.             p = list(y).count(class_id)/Number_group
118.             sum_of_classes += p**2
119.         return 1. - sum_of_classes
120.
121.     @property
122.     def split_point(self):
123.         point = self.x.values[self.indices,self.split_index]
124.         return point
125.
126.     @property

```

```

127. def is_leaf(self):
128.     if self.score == 0 or self.max_depth <= 0:
129.         return True
130.     else:
131.         return False
132.
133. def predict(self, x):
134.     predictor = np.array([self.predict_single(xi) for xi in x])
135.     return predictor
136.
137. def predict_single(self, xi):
138.     if self.is_leaf:
139.         return self.value
140.     if xi[self.split_index] <= self.split:
141.         t = self.lhs
142.     else:
143.         t = self.rhs
144.     return t.predict_single(xi)
145.
146. def accuracy(d, y):
147.     t = 0
148.     f = 0
149.     for i in range(len(y)):
150.         counts = np.bincount(d[:,i])
151.         if np.argmax(counts) == y[i]:
152.             t = t + 1
153.         else:
154.             f = f + 1
155.     return (t / (t + f)) * 100
156.
157. def splitter(data,y,perctrain, percv, perctest):
158.     a = (len(data)*perctrain)/100
159.     data_train = data[:int(a)]
160.     label_train = y[:int(a)]
161.
162.     b = (len(data)*percv)/100
163.     data_val = data[int(a):int(b)+int(a)]
164.     label_val = y[int(a):int(b)+int(a)]
165.
166.     c = (len(data)*perctest)/100
167.     data_test = data[int(b)+int(a):int(c)+int(b)+int(a)]
168.     label_test = y[int(b)+int(a):int(c)+int(b)+int(a)]
169.     return data_train, label_train, data_val, label_val, data_test, label_test
170.
171. data_np = np.asarray(features)
172. label_np = np.asarray(labels)
173. shuffle=np.random.permutation(data_np.shape[0])
174. data_np = data_np[shuffle]
175. label_np = label_np[shuffle]
176.
177. train_features, train_labels, val_features, val_labels, test_features, test_labels= splitter(d
    ata_np, label_np, 70, 20, 10)
178. print(train_features.shape)
179. print(train_labels.shape)
180. print(val_features.shape)
181. print(val_labels.shape)
182. print(test_features.shape)
183. print(test_labels.shape)
184.
185. np.random.seed(1)
186. MyForest = Random_Forest(pd.DataFrame(train_features), train_labels, 40, 5, 80)
187. prediction_train = MyForest.predict(np.asarray(train_features))
188. print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
189. prediction_val = MyForest.predict(np.asarray(val_features))
190. print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
191.

```

```

192. np.random.seed(1)
193. MyForest2 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 80)
194. prediction_train = MyForest2.predict(np.asarray(train_features))
195. print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
196. prediction_val = MyForest2.predict(np.asarray(val_features))
197. print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
198.
199. np.random.seed(1)
200. MyForest3 = Random_Forest(pd.DataFrame(train_features), train_labels, 40, 5, 120)
201. prediction_train = MyForest3.predict(np.asarray(train_features))
202. print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
203. prediction_val = MyForest3.predict(np.asarray(val_features))
204. print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
205.
206. np.random.seed(1)
207. MyForest4 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120)
208. prediction_train = MyForest4.predict(np.asarray(train_features))
209. print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
210. prediction_val = MyForest4.predict(np.asarray(val_features))
211. print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
212.
213. np.random.seed(1)
214. MyForest5 = Random_Forest(pd.DataFrame(train_features), train_labels, 40, 5, 40)
215. prediction_train = MyForest5.predict(np.asarray(train_features))
216. print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
217. prediction_val = MyForest5.predict(np.asarray(val_features))
218. print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
219.
220. np.random.seed(1)
221. MyForest6 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 40)
222. prediction_train = MyForest6.predict(np.asarray(train_features))
223. print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
224. prediction_val = MyForest6.predict(np.asarray(val_features))
225. print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
226.
227. np.random.seed(1)
228. MyForest7 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 3, 120)
229. prediction_train = MyForest7.predict(np.asarray(train_features))
230. print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
231. prediction_val = MyForest7.predict(np.asarray(val_features))
232. print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
233.
234. np.random.seed(1)
235. MyForest8 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 8, 120)
236. prediction_train = MyForest8.predict(np.asarray(train_features))
237. print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
238. prediction_val = MyForest8.predict(np.asarray(val_features))
239. print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
240.
241. np.random.seed(1)
242. MyForest9 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=20
    , min_leaf=2)
243. prediction_train = MyForest9.predict(np.asarray(train_features))
244. print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
245. prediction_val = MyForest9.predict(np.asarray(val_features))
246. print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
247.
248. np.random.seed(1)
249. MyForest10 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=5
    , min_leaf=2)
250. prediction_train = MyForest10.predict(np.asarray(train_features))
251. print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
252. prediction_val = MyForest10.predict(np.asarray(val_features))
253. print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
254.
255. np.random.seed(1)

```



```

256.MyForest11 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=2
    0, min_leaf=10)
257.prediction_train = MyForest11.predict(np.asarray(train_features))
258.print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
259.prediction_val = MyForest11.predict(np.asarray(val_features))
260.print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
261.
262.np.random.seed(1)
263.MyForest12 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=5
    , min_leaf=10)
264.prediction_train = MyForest12.predict(np.asarray(train_features))
265.print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
266.prediction_val = MyForest12.predict(np.asarray(val_features))
267.print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
268.
269.np.random.seed(1)
270.MyForest13 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=1
    0, min_leaf=2)
271.prediction_train = MyForest13.predict(np.asarray(train_features))
272.print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
273.prediction_val = MyForest13.predict(np.asarray(val_features))
274.print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
275.
276.np.random.seed(1)
277.MyForest14 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=1
    0, min_leaf=10)
278.prediction_train = MyForest14.predict(np.asarray(train_features))
279.print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
280.prediction_val = MyForest14.predict(np.asarray(val_features))
281.print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')
282.
283.final_features = np.concatenate((train_features, val_features))
284.final_labels = np.concatenate((train_labels, val_labels))
285.print(final_features.shape)
286.print(final_labels.shape)
287.
288.import time
289.start = time.time()
290.np.random.seed(1)
291.MyForest_last = Random_Forest(pd.DataFrame(final_features), final_labels, 80, 5, 120)
292.prediction_test = MyForest_last.predict(np.asarray(test_features))
293.print('Testing Accuracy: ' + str(accuracy(prediction_test, test_labels)) + '%')
294.elapsed_time = time.time() - start
295.    print('Execution time: ' + str(elapsed_time))

```

Appendix 3 – kNN

```
1. import matplotlib.pyplot as plt
2. import numpy as np
3. import pandas as pd
4.
5. #load data
6. featuresCont = np.load('dataKNNcont.npy', allow_pickle=True)
7. featuresDisc = np.load('dataKNNdiscrete.npy', allow_pickle=True)
8. labels = np.load('labelsKNN.npy', allow_pickle=True)
9. print(featuresCont, featuresCont.shape)
10. print(featuresDisc, featuresDisc.shape)
11. print(labels, labels.shape)
12.
13. #three way split
14. def splitData(X_cont, X_disc, y, tr, val):
15.     shuffle = np.random.permutation(len(y))
16.     X_cont = X_cont[shuffle]
17.     X_disc = X_disc[shuffle]
18.     y = y[shuffle]
19.
20.     tr_ind = int(np.floor(len(y)*tr))
21.     val_ind = int(np.floor(len(y)*(tr+val)))
22.
23.     X_tr_cont = X_cont[0:tr_ind]
24.     X_tr_disc = X_disc[0:tr_ind]
25.     y_tr = y[0:tr_ind]
26.
27.     X_val_cont = X_cont[tr_ind:val_ind]
28.     X_val_disc = X_disc[tr_ind:val_ind]
29.     y_val = y[tr_ind:val_ind]
30.
31.     X_test_cont = X_cont[val_ind:]
32.     X_test_disc = X_disc[val_ind:]
33.     y_test = y[val_ind:]
34.
35.     return X_tr_cont, X_tr_disc, y_tr, X_val_cont, X_val_disc, y_val, X_test_cont, X_test_disc
    , y_test
36.
37. import sys
38. def progressBar(value, endvalue, bar_length=20):
39.     percent = float(value) / endvalue
40.     arrow = '-' * int(round(percent * bar_length)-1) + '>'
41.     spaces = ' ' * (bar_length - len(arrow))
42.
43.     sys.stdout.write("\rPercent: [{0}] {1}%".format(arrow + spaces, int(round(percent * 10
0))))
44.     sys.stdout.flush()
45.
46. def KNN(trainC, trainD, trainL, valC, valD, valL , params):
47.     k, alpha1, alpha2, alpha3 = params
48.     trCount = 0
49.
50.     for i in range(valC.shape[0]):
51.         #print(i)
52.         #continuous distances
53.         instanceMat = np.broadcast_to(valC[i], trainC.shape)
54.         distVecC = np.sum( (trainC-instanceMat)**2, axis=1)
55.
56.         #discrete distances
57.         instanceMatD = np.broadcast_to(valD[i], trainD.shape)
58.         distMatD = np.zeros(instanceMatD.shape)
59.         distMatD[trainD != instanceMatD] = 1
60.         distMatD[:,0] *= alpha1
61.         distMatD[:,1] *= alpha2
```

```

62.     distMatD[:,2] *= alpha3
63.     distVecD = np.sum(distMatD, axis=1)
64.
65.     totalDist = distVecD + distVecC
66.     #print(totalDist.shape)
67.     #retriev the labels with the k smallest distance
68.     idx = np.argsort(totalDist)[:k]
69.     #print(idx, idx.shape)
70.     kLabels = trainL[idx]
71.     #print(kLabels, kLabels.shape)
72.
73.     #run a majority vote for the labels
74.     pred = np.bincount(kLabels).argmax()
75.     #print(np.bincount(kLabels))
76.     #print(pred)
77.
78.     if(pred == valL[i]):
79.         trCount += 1
80.
81.     progressBar(i, valC.shape[0])
82.
83.     valAcc = 100.0*trCount/valC.shape[0]
84.     return valAcc
85.
86. def KNN(trainC, trainD, trainL, valC, valD, valL , params):
87.     k, alpha1, alpha2, alpha3 = params
88.     trCount = 0
89.
90.     for i in range(valC.shape[0]):
91.         #print(i)
92.         #continuous distances
93.         instanceMat = np.broadcast_to(valC[i], trainC.shape)
94.         distVecC = np.sum( (trainC-instanceMat)**2, axis=1)
95.
96.         #discrete distances
97.         instanceMatD = np.broadcast_to(valD[i], trainD.shape)
98.         distMatD = np.zeros(instanceMatD.shape)
99.         distMatD[trainD != instanceMatD] = 1
100.        distMatD[:,0] *= alpha1
101.        distMatD[:,1] *= alpha2
102.        distMatD[:,2] *= alpha3
103.        distVecD = np.sum(distMatD, axis=1)
104.
105.        totalDist = distVecD + distVecC
106.        #print(totalDist.shape)
107.        #retriev the labels with the k smallest distance
108.        idx = np.argsort(totalDist)[:k]
109.        #print(idx, idx.shape)
110.        kLabels = trainL[idx]
111.        #print(kLabels, kLabels.shape)
112.
113.        #run a majority vote for the labels
114.        pred = np.bincount(kLabels).argmax()
115.        #print(np.bincount(kLabels))
116.        #print(pred)
117.
118.        if(pred == valL[i]):
119.            trCount += 1
120.
121.        progressBar(i, valC.shape[0])
122.
123.        valAcc = 100.0*trCount/valC.shape[0]
124.        return valAcc
125.
126. X_tr_cont, X_tr_disc, y_tr, X_val_cont, X_val_disc, y_val, X_test_cont, X_test_disc, y_test =
    \

```

```

127.splitData(featuresCont, featuresDisc, labels, 0.8, 0.013)
128.
129.kList = [1,2,3,4,5,6,7,8,9,10]
130.
131.def gridSearchV2(trainC, trainD, trainL, valC, valD, valL , kList, alpha=0.1):
132.     bestValAcc = 0.0
133.     bestParams = (-1,-1,-1,-1)
134.     for k in kList:
135.         params = (k,alpha, alpha, alpha)
136.         print('kNN with params', params)
137.         acc = KNN(trainC, trainD, trainL, valC, valD, valL, params)
138.         print('\nValidation Accuracy:', acc)
139.         if(acc > bestValAcc):
140.             bestValAcc = acc
141.             bestK = k
142.     return bestValAcc, bestK
143.
144.kList = [1,2,3,4,5,6,7,8,9,10]
145.valAcc, bestK = gridSearchV2(X_tr_cont, X_tr_disc, y_tr, X_val_cont, X_val_disc, y_val, kList)

146.
147.def gridSearchV3(trainC, trainD, trainL, valC, valD, valL , alphaList, K=bestK):
148.     bestValAcc = 0.0
149.     bestParams = (-1,-1,-1)
150.     for a1 in alphaList:
151.         for a2 in alphaList:
152.             for a3 in alphaList:
153.                 params = (K, a1, a2, a3)
154.                 print('kNN with params', params)
155.                 acc = KNN(trainC, trainD, trainL, valC, valD, valL, params)
156.                 print('\nValidation Accuracy', acc)
157.                 if(acc > bestValAcc):
158.                     bestValAcc = acc
159.                     bestParams = (K,a1,a2,a3)
160.     return bestValAcc, bestParams
161.
162.alphaList = [0.2,0.4,0.6,0.8]
163.valAcc, bestK = gridSearchV3(X_tr_cont, X_tr_disc, y_tr, X_val_cont, X_val_disc, y_val, alphaList)
164.
165.bestParams = bestK
166.print('The best parameters that maximize accuracy in the given grid structure', bestParams)
167.
168.def KNNv2(trainC, trainD, trainL, valC, valD, valL , params):
169.     k, alpha1, alpha2, alpha3 = params
170.     trCount = 0
171.     preds = []
172.     for i in range(valC.shape[0]):
173.         #print(i)
174.         #continuous distances
175.         instanceMat = np.broadcast_to(valC[i], trainC.shape)
176.         distVecC = np.sum( (trainC-instanceMat)**2, axis=1)
177.
178.         #discrete distances
179.         instanceMatD = np.broadcast_to(valD[i], trainD.shape)
180.         distMatD = np.zeros(instanceMatD.shape)
181.         distMatD[trainD != instanceMatD] = 1
182.         distMatD[:,0] *= alpha1
183.         distMatD[:,1] *= alpha2
184.         distMatD[:,2] *= alpha3
185.         distVecD = np.sum(distMatD, axis=1)
186.
187.         totalDist = distVecD + distVecC
188.         #retrieve the labels with the k smallest distance
189.         idx = np.argsort(totalDist)[:k]
190.         kLabels = trainL[idx]

```

```

191.
192.     #run a majority vote for the labels
193.     pred = np.bincount(kLabels).argmax()
194.     preds.append(pred)
195.
196.     if(pred == valL[i]):
197.         trCount += 1
198.
199.     progressBar(i, valC.shape[0])
200.
201.     valAcc = 100.0*trCount/valC.shape[0]
202.     print('\Test Accuracy', valAcc)
203.     return preds
204.
205. predVector = KNNv2(X_tr_cont, X_tr_disc, y_tr, X_test_cont, X_test_disc, y_test, bestParams)
206.
207. # https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
208. from sklearn.metrics import confusion_matrix
209. from sklearn.utils.multiclass import unique_labels
210.
211. def plot_confusion_matrix(y_true, y_pred, classes,
212.                           normalize=False,
213.                           title=None,
214.                           cmap=plt.cm.Blues):
215.     """
216.     This function prints and plots the confusion matrix.
217.     Normalization can be applied by setting `normalize=True`.
218.     """
219.     if not title:
220.         if normalize:
221.             title = 'Normalized confusion matrix'
222.         else:
223.             title = 'Confusion matrix, without normalization'
224.
225.     # Compute confusion matrix
226.     cm = confusion_matrix(y_true, y_pred)
227.     # Only use the labels that appear in the data
228.     classes = classes[unique_labels(y_true, y_pred)]
229.     if normalize:
230.         cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
231.         print("Normalized confusion matrix")
232.     else:
233.         print('Confusion matrix, without normalization')
234.
235.     fig, ax = plt.subplots(figsize=(20,20))
236.     im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
237.     ax.figure.colorbar(im, ax=ax)
238.     # We want to show all ticks...
239.     ax.set(xticks=np.arange(cm.shape[1]),
240.           yticks=np.arange(cm.shape[0]),
241.           # ... and label them with the respective list entries
242.           xticklabels=classes, yticklabels=classes,
243.           title=title,
244.           ylabel='True label',
245.           xlabel='Predicted label')
246.
247.     # Rotate the tick labels and set their alignment.
248.     plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
249.             rotation_mode="anchor")
250.
251.     # Loop over data dimensions and create text annotations.
252.     fmt = '.2f' if normalize else 'd'
253.     thresh = cm.max() / 2.
254.     for i in range(cm.shape[0]):
255.         for j in range(cm.shape[1]):
256.             ax.text(j, i, format(cm[i, j], fmt),

```

```

257.             ha="center", va="center",
258.             color="white" if cm[i, j] > thresh else "black")
259.     fig.tight_layout()
260.     return ax
261.
262. # Plot normalized confusion matrix
263. class_names = ['Dance', 'Comedy', 'Anime', 'Rap', 'Pop', 'Ska', 'Alternative', 'Country', 'Ele
    ctronic', 'Reggaeton', \
264.               'Soul', 'Hip-
    Hop', 'Opera', 'Soundtrack', 'R&B', 'Indie', 'Folk', 'Reggae', 'Classical', 'Blues', 'World', \
265.               'Rock', 'Jazz']
266. plot_confusion_matrix(np.asarray(y_test), np.asarray(predVector), classes=np.asarray(class_nam
    es), normalize=True,
267.                       title='Confusion Matrix for kNN on Test Data')
268. plt.show()

```

Appendix 4 – Data Analysis

```
1. import pandas as pd
2. import numpy as np
3. import matplotlib.pyplot as plt
4.
5. data = pd.read_csv('SpotifyFeatures.csv')
6. data.head()
7.
8. classes = list(set(data['genre']))
9. print(classes)
10.
11. data = pd.read_csv('spotifyData_v2.csv')
12. classes = list(set(data['genre']))
13. print(classes)
14.
15. ax = plt.figure()
16. labels = data['genre'].values
17. ax = plt.hist(labels, bins=len(classes))
18. plt.xticks(rotation='vertical')
19. plt.show()
20.
21. data = pd.read_csv('spotifyData_v3.csv')
22.
23. classes = list(set(data['genre']))
24. print(classes)
25.
26. ax = plt.figure()
27. labels = data['genre'].values
28. ax = plt.hist(labels, bins=len(classes))
29. plt.xticks(rotation='vertical')
30. plt.show()
31.
32. from collections import Counter
33. labels = data['genre'].values
34. print(len(labels))
35. Counter(labels)
36.
37. downsampleSize = 8280
38. '''dataDS = pd.DataFrame(columns=['genre','artist_name','track_name','track_id','popularity',
39. 'acousticness','danceability',\
40. 'duration_ms','energy','instrumentalness','key','liveness','loudness','mode','speechiness',
41. 'tempo','time_signature','valence'])'''
41. for i in range(len(classes)):
42.     part = data.loc[data['genre'] == classes[i]]
43.     #part = part.reset_index(drop=True)
44.     indexToRemove = np.random.permutation(len(part))[0:len(part)-downsampleSize]
45.     partDS = part.iloc[indexToRemove]
46.     data.drop(partDS.index, inplace=True)
47. data.reset_index(drop=True)
48.
49. ax = plt.figure()
50. labels = data['genre'].values
51. ax = plt.hist(labels, bins=len(classes))
52. plt.xticks(rotation='vertical')
53. plt.show()
54. data.head()
55.
56. dataKNN = data.copy()
57. dataKNN.drop(['track_id','track_name','time_signature'], axis=1, inplace=True)
58. dataKNN.head()
59.
60. dataMLP = data.copy()
```

```

61. dataMLP.drop(['track_id', 'track_name', 'artist_name', 'time_signature', 'key', 'mode'], axis=1, in
    place=True)
62. dataMLP.head()
63.
64. dataRF = data.copy()
65. dataRF.drop(['track_id', 'track_name'], axis=1, inplace=True)
66. dataRF.head()
67.
68. col_num = 3
69. row_num = 4
70. for i in range(numFeatures):
71.     cnt = 0
72.     print(dataMLP.columns[i+1])
73.     for j in range(numFeatures-1, -1, -1):
74.         cnt += 1
75.         plt.subplot(row_num, col_num, cnt)
76.         plt.scatter(dataCont[:, i], dataCont[:, j], s=1, c=labels, cmap='jet')
77.         plt.ylabel(dataMLP.columns[j+1])
78.         plt.gca().axes.get_xaxis().set_visible(False)
79.     plt.show()
80.
81. dataCont = np.asarray(dataMLP)
82.
83. #create a dictionary for the integer labels
84. dictionary = {}
85. dictrev = {}
86. for i in range(len(classes)):
87.     dictionary[classes[i]] = i
88.     dictrev[i] = classes[i]
89. print(dictionary)
90. print(dictrev)
91.
92. labels = [dictionary[smp] for smp in dataCont[:, 0]]
93. dataCont = dataCont[:, 1:]
94.
95. numFeatures = dataCont.shape[1]
96.
97. dataMLP = np.asarray(dataMLP[:, 1:]).astype(float)
98. dataMLP[:, 0] = dataMLP[:, 0]/100.0
99. dataMLP[:, 3] = (dataMLP[:, 3]-15e3)/(5.6e6-15e3)
100. dataMLP[:, 7] = (dataMLP[:, 7]+60.0)/60.0
101. dataMLP[:, 9] = (dataMLP[:, 9]-30.0)/250.0
102.
103. np.save('dataMLP.npy', dataMLP)
104. np.save('labelsMLP.npy', np.asarray(labels))
105.
106. np.load('dataMLP.npy', allow_pickle=True)
107. np.load('labelsMLP.npy', allow_pickle=True)
108.
109. np.load('labelsMLP.npy', allow_pickle=True)
110.
111. dataKNN.head()
112.
113. dataKNNdisc = dataKNN[['artist_name', 'key', 'mode']]
114. dataKNNdisc.head()
115. np.save('dataKNNdiscrete.npy', np.asarray(dataKNNdisc).astype(str))
116. np.save('labelsKNN.npy', labels)
117. np.save('dataKNNcont.npy', dataMLP)
118.
119. def vector1H(x, maxInd):
120.     out = np.zeros(maxInd)
121.     out[x] = 1
122.     return out
123.
124. def mat1H2(y, maxInd):
125.     out = np.zeros((len(y), maxInd))

```



```

126.     for i in range(len(y)):
127.         out[i,:] = vector1H(y[i], maxInd)
128.     return out
129.
130. def dictionary_maker(data, feature):
131.     classes = list(set(data[feature]))
132.     dictionary = {}
133.     for i in range(len(classes)):
134.         dictionary[classes[i]] = i
135.     return(dictionary)
136.
137. #dataCont[:,0] = [dictionary[smp] for smp in dataCont[:,0]]
138.
139. dataCont = np.asarray(data)
140.
141. dictionary_key = dictionary_maker(data, 'key')
142. dictionary_ts = dictionary_maker(data, 'time_signature')
143. dictionary_mode = dictionary_maker(data, 'mode')
144.
145. index_key = data.columns.get_loc('key')
146. index_ts = data.columns.get_loc('time_signature')
147. index_mode = data.columns.get_loc('mode')
148.
149. key_dict = [dictionary_key[smp] for smp in dataCont[:,index_key]]
150. ts_dict = [dictionary_ts[smp] for smp in dataCont[:,index_ts]]
151. mode_dict = [dictionary_mode[smp] for smp in dataCont[:,index_mode]]
152.
153. one_hot_key = mat1H2(key_dict, len(dictionary_key))
154. one_hot_ts = mat1H2(ts_dict, len(dictionary_ts))
155. one_hot_mode = mat1H2(mode_dict, len(dictionary_mode))
156.
157. dataRF = data.copy()
158. dataRF.drop(['track_id', 'track_name', 'genre', 'key', 'time_signature', 'mode', 'artist_name']
159.             , axis=1, inplace=True)
159. #dataRF.head()
160. dataRF = dataRF.to_numpy()
161. dataRF = pd.DataFrame(dataRF)
162. one_hot_key = pd.DataFrame(one_hot_key)
163. one_hot_ts = pd.DataFrame(one_hot_ts)
164. one_hot_mode = pd.DataFrame(one_hot_mode)
165. dataRF = pd.concat([dataRF, one_hot_key, one_hot_ts, one_hot_mode], axis=1, ignore_index=True)
166. dataRF.head()
167.
168. dataCont = np.asarray(dataMLP)
169.
170. #create a dictionary for the integer labels
171. dictionary = {}
172. for i in range(len(classes)):
173.     dictionary[classes[i]] = i
174. print(dictionary)
175.
176. dataRF.to_pickle('dataRF.pkl')
177. np.save('labelsRF.npy', labels)

```