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

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.

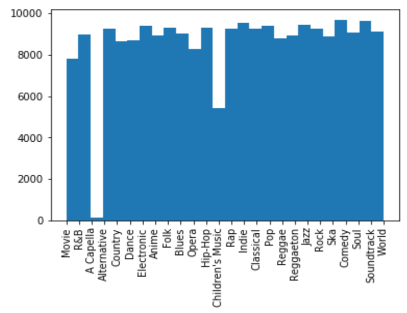


Figure 1 – Histogram for the Distribution of Classes in the Dataset

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.

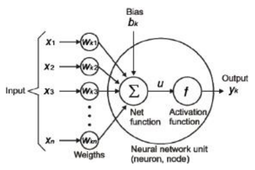


Figure 2 – McCullough-Pitts Neuron [3]

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.

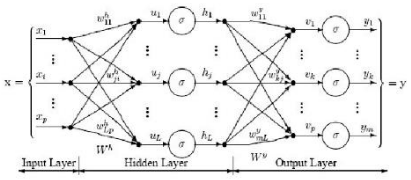


Figure 3 – An exemplary MLP Structure [4]

Mathematically, this is a directed graph structure. The output of the graph the 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.

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

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

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.

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

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.

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

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

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.

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

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

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.

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

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

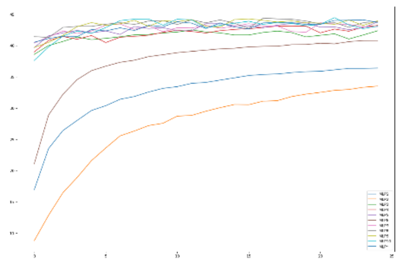
$$\phi'_{tanh}(v) = 1 - tanh^2(v) \Rightarrow 1 - y^2$$

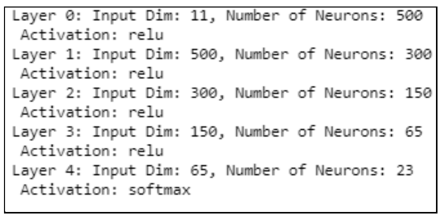
We have used the stochastic mini batch updating as stated in the Phase 1 report. In the report, we have mentioned that we will be using the SGDM (Stochastic Gradient Descent with Momentum) as optimizer and we would implement adaptive learning algorithms if time allowed. For this phase, we have been able 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.

Iteration	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





```
Layer 0: Input Dim: 11, Number of Neurons: 500
 Activation: relu
Layer 1: Input Dim: 500, Number of Neurons: 300
 Activation: relu
Layer 2: Input Dim: 300, Number of Neurons: 150
 Activation: relu
Layer 3: Input Dim: 150, Number of Neurons: 65
 Activation: relu
Layer 4: Input Dim: 65, Number of Neurons: 23
 Activation: softmax
```

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.

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

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.

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

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

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

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.

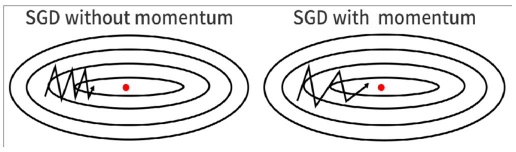


Figure 6 – SGD vs SGDM [5]

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.

$$m_t = (1 - \beta_1)\left(\delta_t \widetilde{\mathrm{X}}_{layer,t}\right) + \beta_1 m_{t-1}$$

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

$$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t} \qquad \widehat{v_t} = \frac{v_t}{1 - \beta_2^t}$$

$$AW_{E,t+1} = W_{E,t} - \frac{\eta \cdot \widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon}$$

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.

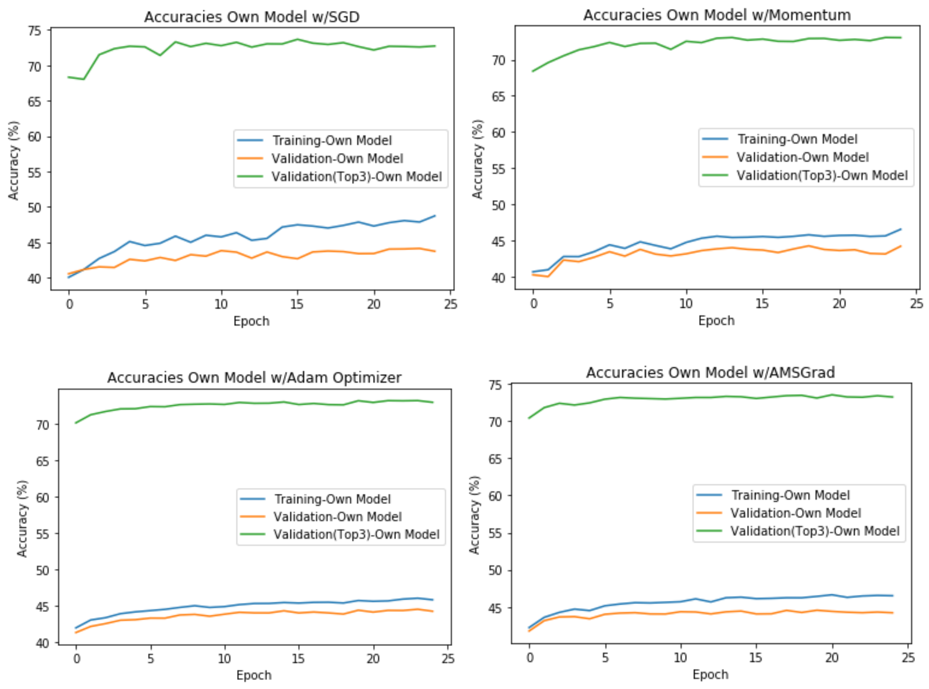
$$m_t = (1 - \beta_1)\left(\delta_t \widetilde{\mathrm{X}}_{layer,t}\right) + \beta_1 m_{t-1}$$

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

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

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

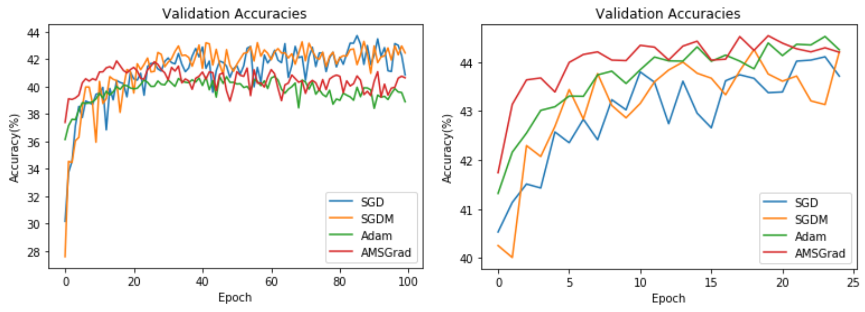
The parameters are the same as in Adam. After implementing all these algorithms that are explained above, we have run the best network on each of the with batch size of 529 and 100 epochs. In total, we have implemented four neural network training processes and each one in 100 epochs nearly consumed 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

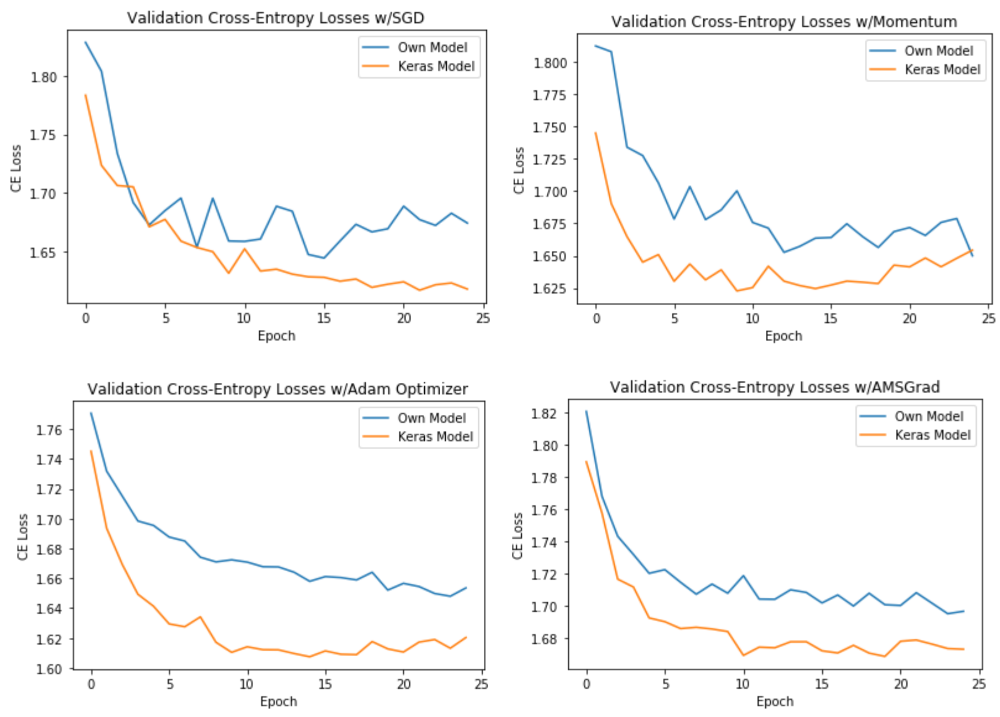
Overall, we observe what we thought we would observe. We expected that AMSGrad to be better than Adam and Adam to be better than SGDM and SGDM to be than SGD. Here, we see similar convergence points, however with different convergence rates, as we in the order we have hypothesized. These results have 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.

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

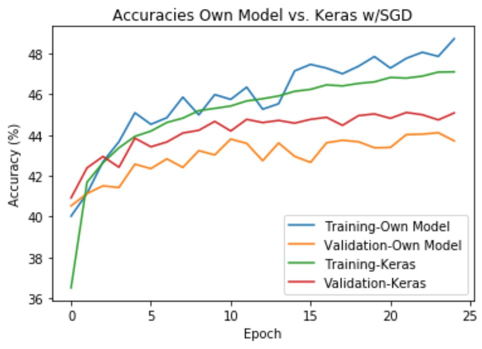
Table 2 – Test Accuracies for Best MLP Architecture

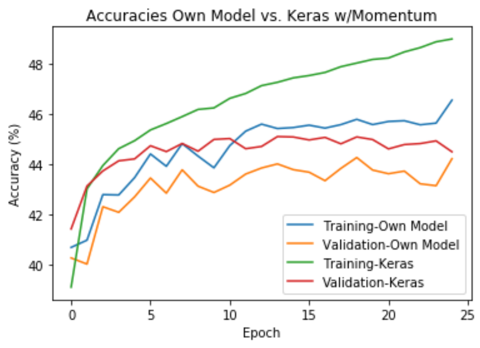
As expected, we see that the AMSGrad algorithm converges faster than the other optimizers and performs the best. Hence, the next part is to compare with the state-of-the-art. In order to do that, we have implemented the same algorithms on Keras, a highly integrated deep learning framework, and compared 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.

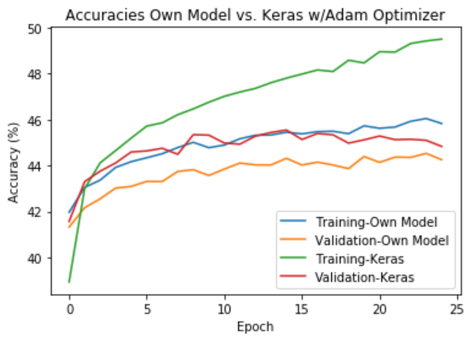


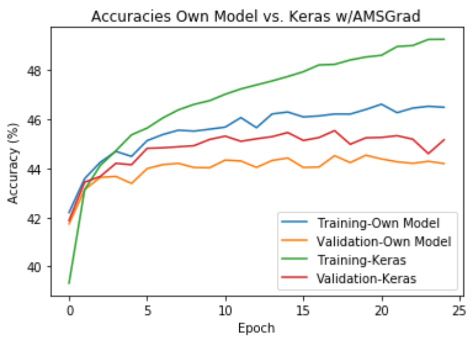
Figures 13-16 – Validation Cross-Entropy Losses with SGD, SGDM, Adam and AMSGrad from left to right and up to down.

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".

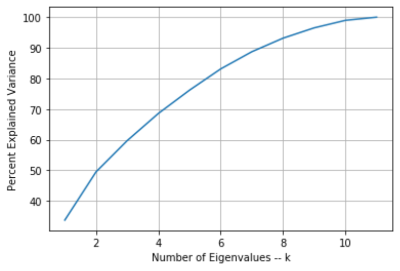
$$\tilde{X}_{N,K} = \frac{X - \mu_X}{\sigma_X}$$

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

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.

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

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.



```
Percent Explained Variances
PEV for k=1  33.74422246797036
PEV for k=2  49.47216405564261
PEV for k=3  59.68917062161931
PEV for k=4  68.56672808155403
PEV for k=5  76.17548067064581
PEV for k=6  83.11402035438327
PEV for k=7  88.72886010179718
PEV for k=8  93.15117698788197
PEV for k=9  96.52404748805709
PEV for k=10  98.97334408995677
PEV for k=11  99.99999999999999
```

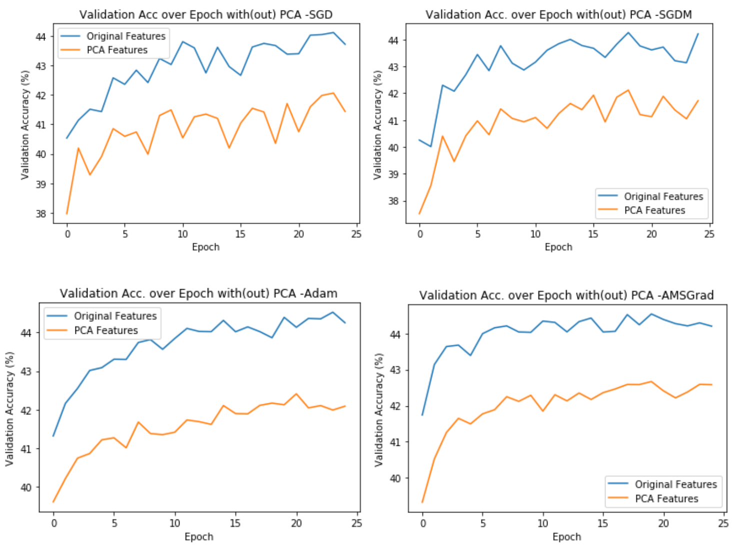
Figure 21 – Percent Explained Variance vs. Number of Eigenvalues

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.

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

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

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%

AMSGrad	**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:

$$\boldsymbol{IG} = \boldsymbol{I}^{Gini}(\boldsymbol{D}_{parent}) - \frac{\boldsymbol{N}_{left}}{\boldsymbol{N}_{parent}}\boldsymbol{I}^{Gini}(\boldsymbol{D}_{left}) - \frac{\boldsymbol{N}_{right}}{\boldsymbol{N}_{parent}}\boldsymbol{I}^{Gini}(\boldsymbol{D}_{right})$$

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

$$\boldsymbol{I}^{Gini}(\boldsymbol{D}) = 1 - \sum_{i=1}^{\boldsymbol{N}_{classes}} \boldsymbol{p}_i^{\,2}$$

Where $\boldsymbol{N}_{classes}$ denotes the number of classes in the dataset, and $\boldsymbol{p}_i$ denotes the probability that a random selection from data (Node) $\boldsymbol{D}$ 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:

Forest Number	Sample Size	Number of Trees	Number of Features	Maximum Depth	Minimum Leaf Size	Validation Accuracy
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.

$$d_1(x_{i,n}, x_{j,n}) = \begin{cases} 0 \ if \ x_{i,n} = x_{j,n} \\ \propto^2 \ else \end{cases} \qquad d_2(x_{i,n}, x_{j,n}) = x_{i,n} - x_{j,n}$$

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

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 $\propto_i \ \ i = 1,2,3,$ 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.

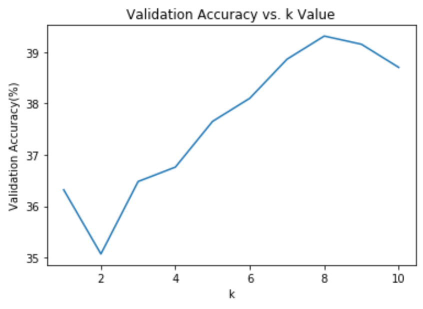


Figure 20 – Validation Accuracies with respect to k Value

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,

$$(\propto_1, \propto_2, \propto_3, k) \ = \ (0.8{,}0.2.0.2.0.8)$$

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 $\propto_1$ 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?,” ResearcGate. [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

```python
import matplotlib.pyplot as plt
import numpy as np

class Layer:
    def __init__(self, inputDim, numNeurons, activation, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation

        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

    def actFcn(self,x):
        if(self.activation == 'sigmoid'):
            expx = np.exp(x)
            return expx/(1+expx)
        elif(self.activation == 'softmax'):
            expx = np.exp(x - np.max(x))
            return expx/np.sum(expx, axis=0)
        elif(self.activation == 'tanh'):
            return np.tanh(x)
        elif(self.activation == 'relu'):
            out = np.maximum(0,x)
            return out

    def activate(self, x):
        if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation == 'relu':
            if(x.ndim == 1):
                x = x.reshape(x.shape[0],1)
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*1]]
            self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def derActiv(self, x):
        if(self.activation == 'sigmoid'):
            return x*(1-x)
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation == 'tanh'):
            return 1 - x**2;
        elif(self.activation == 'relu'):
            der = x
            der[x > 0] = 1
            der[x <= 0] = 0
            return der

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

class MLP:
    def __init__(self):
        self.layers = []

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

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activate(out)
        return out

    def prediction(self, inp):
        out = self.forward(inp)
        if(out.ndim == 1):
            return np.argmax(out)
        return np.argmax(out, axis=0)

    def topKaccuracy(self, inp, realOut, k):
        out = self.forward(inp)
        cnt = 0
        for i in range(out.shape[1]):
            topKind = (out[:,i]).argsort()[::-1][:k]
            if realOut[i] in topKind:
                cnt += 1
        return cnt/out.shape[1]

    def backProp(self, inp, out, lrnRate, batchSize, loss):
        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                if(loss == 'mse'):
                    lyr.error = net_out -out
                    derMatrix = lyr.derActiv(lyr.lastActiv)
                    lyr.delta = derMatrix * lyr.error
                elif(loss == 'ce' and lyr.activation == 'softmax'):
                    lyr.delta = net_out - out
                else:
                    assert('Cant do that')
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.derActiv(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]

            update =  (lrnRate * np.matmul(lyr.delta, inputToUse.T))/batchSize
            lyr.weightsE -= update

    def train(self, inp, out, inpTest, outTest, lrnRate, epochNum, batchSize, loss):
        errList = []
        trAccs = []
        valAccs = []
        valKAccs = []

        for ep in range(epochNum):
            print('--------------------------------------------------------------\nEpoch', ep+1)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]
            numBatches = int(np.floor(len(inp)/batchSize))

            for j in range(numBatches):
                batch_inp = inp[batchSize*j:batchSize*j+batchSize]
                batch_out = out[batchSize*j:batchSize*j+batchSize]

                batch_out_1H = mat1H2(batch_out, np.max(out)+1).T

                self.backProp(batch_inp.T, batch_out_1H, lrnRate, batchSize, loss)

            valOutput = self.forward(inpTest.T)
            if(loss == 'ce'):
                err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput.shape[1]
            elif(loss == 'mse'):
                err = np.sum((outTest.T - self.forward(inpTest.T))**2)/valOutput.shape[1]
            print(loss.upper() +' Validation Error ', err)
            errList.append(err)

            trPred = self.prediction(inp.T)
            trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
            print('Training Accuracy: ', trAcc)
            trAccs.append(trAcc)

            valPred = self.prediction(inpTest.T)
            valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]*100
            print('Validation Accuracy: ', valAcc)
            valAccs.append(valAcc)

            #k is chosen as three
            K = 3
            topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
            print('Top ' + str(K) + ' Accuracy: ', topKacc)
            valKAccs.append(topKacc)


        return errList, trAccs, valAccs, valKAccs

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr

features = np.load('dataMLP.npy', allow_pickle=True)
labels = np.load('labelsMLP.npy', allow_pickle=True)
labels = labels.reshape((labels.shape[0],1))

features -= np.mean(features, axis=0)
features /= np.std(features, axis=0)

#three-way split
def splitData(X, y, tr, val):
    np.random.seed(456)
    shuffle = np.random.permutation(len(y))
    X = X[shuffle]
    y = y[shuffle]

    tr_ind = int(np.floor(len(y)*tr))
    val_ind = int(np.floor(len(y)*(tr+val)))

    X_tr = X[0:tr_ind]
    y_tr = y[0:tr_ind]
    X_val = X[tr_ind:val_ind]
    y_val = y[tr_ind:val_ind]
    X_test = X[val_ind:]
    y_test = y[val_ind:]

    return X_tr, y_tr, X_val, y_val, X_test, y_test

def vector1H(x, maxInd):
    out = np.zeros(maxInd)
    out[x] = 1
    return out
def mat1H2(y, maxInd):
    out = np.zeros((y.shape[0], maxInd))
    for i in range(y.shape[0]):
        out[i,:] = vector1H(y[i], maxInd)
    return out

x_train, y_train, x_val, y_val, x_test, y_test = splitData(features, labels, 0.7, 0.2)

from tensorflow.keras import layers, models, initializers, optimizers
model = models.Sequential()
model.add(layers.Dense(500, input_shape=(11,), activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                      distribution = 'normal', \
                                                                      seed=None, \
                                                                      scale=2.0),\
                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                      distribution = 'normal', \
                                                                      seed=None, \
                                                                      scale=2.0)))
model.add(layers.Dense(300,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                      distribution = 'normal', \
                                                                      seed=None, \
                                                                      scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                      distribution = 'normal', \
                                                                      seed=None, \
                                                                      scale=2.0)))
model.add(layers.Dense(150,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                      distribution = 'normal', \
                                                                      seed=None, \
                                                                      scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                      distribution = 'normal', \
                                                                      seed=None, \
                                                                      scale=2.0)))
model.add(layers.Dense(65,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                      distribution = 'normal', \
                                                                      seed=None, \
                                                                          scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                          distribution = 'normal', \
                                                                          seed=None, \
                                                                          scale=2.0)))
model.add(layers.Dense(np.max(labels)+1, activation='softmax',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                          distribution = 'normal',\
                                                                          seed=None,\
                                                                          scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                          distribution = 'normal', \
                                                                          seed=None, \
                                                                          scale=2.0)))
momOp = optimizers.SGD(lr=0.1, momentum=0, nesterov=False, decay=0)
model.compile(loss='sparse_categorical_crossentropy', optimizer=momOp, metrics=['accuracy'])
hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)

#1st Iteration
mlp1 = MLP()
mlp1.addLayer(Layer(x_train.shape[1], 300, 'sigmoid', np.sqrt(6/(x_train.shape[1]+300))))
mlp1.addLayer(Layer(300, np.max(labels)+1, 'softmax', np.sqrt(6/(300+np.max(labels)+1))))
ceErrs, trAccs, valAccs1, valKAccs = mlp1.train(x_train, y_train, x_val, y_val, 0.01, 25, 529,
 loss='ce')

#2nd Iteration
mlp2 = MLP()
mlp2.addLayer(Layer(x_train.shape[1], 300, 'sigmoid', np.sqrt(6/(x_train.shape[1]+300))))
mlp2.addLayer(Layer(300, 100, 'sigmoid', np.sqrt(6/(300+100))))
mlp2.addLayer(Layer(100, np.max(labels)+1, 'softmax', np.sqrt(6/(100+np.max(labels)+1))))
ceErrs, trAccs, valAccs2, valKAccs = mlp2.train(x_train, y_train, x_val, y_val, 0.01, 25, 529,
 loss='ce')

#3rd Iteration
mlp3 = MLP()
mlp3.addLayer(Layer(x_train.shape[1], 300, 'relu', np.sqrt(6/(x_train.shape[1]+300))))
mlp3.addLayer(Layer(300, 100, 'relu', np.sqrt(6/(300+100))))
mlp3.addLayer(Layer(100, np.max(labels)+1, 'softmax', np.sqrt(6/(100+np.max(labels)+1))))
ceErrs, trAccs, valAccs3, valKAccs = mlp3.train(x_train, y_train, x_val, y_val, 0.1, 25, 529,
loss='ce')

#4th Iteration
mlp4 = MLP()
mlp4.addLayer(Layer(x_train.shape[1], 300, 'relu', np.sqrt(2/300)))
mlp4.addLayer(Layer(300, 100, 'relu', np.sqrt(2/100)))
mlp4.addLayer(Layer(100, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
ceErrs, trAccs, valAccs4, valKAccs = mlp4.train(x_train, y_train, x_val, y_val, 0.1, 25, 529,
loss='ce')

#5th Iteration
mlp5 = MLP()
mlp5.addLayer(Layer(x_train.shape[1], 400, 'relu', np.sqrt(2/400)))
mlp5.addLayer(Layer(400, 200, 'relu', np.sqrt(2/200)))
mlp5.addLayer(Layer(200, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
ceErrs, trAccs, valAccs5, valKAccs = mlp5.train(x_train, y_train, x_val, y_val, 0.1, 25, 529,
loss='ce')

#6th Iteration
mlp6 = MLP()
mlp6.addLayer(Layer(x_train.shape[1], 400, 'relu', np.sqrt(2/400)))
mlp6.addLayer(Layer(400, 200, 'relu', np.sqrt(2/200)))
mlp6.addLayer(Layer(200, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
```

```python
ceErrs, trAccs, valAccs6, valKAccs = mlp6.train(x_train, y_train, x_val, y_val, 0.01, 25, 529, loss='mse')

#7th Iteration
mlp7 = MLP()
mlp7.addLayer(Layer(x_train.shape[1], 600, 'relu', np.sqrt(2/600)))
mlp7.addLayer(Layer(600, 200, 'relu', np.sqrt(2/200)))
mlp7.addLayer(Layer(200, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
ceErrs, trAccs, valAccs7, valKAccs = mlp7.train(x_train, y_train, x_val, y_val, 0.1, 25, 529, loss='ce')

#8th Iteration
mlp8 = MLP()
mlp8.addLayer(Layer(x_train.shape[1], 600, 'relu', np.sqrt(2/600)))
mlp8.addLayer(Layer(600, 400, 'relu', np.sqrt(2/400)))
mlp8.addLayer(Layer(400, 200, 'relu', np.sqrt(2/200)))
mlp8.addLayer(Layer(200, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
ceErrs, trAccs, valAccs8, valKAccs = mlp8.train(x_train, y_train, x_val, y_val, 0.1, 25, 529, loss='ce')

#9th Iteration
mlp9 = MLP()
mlp9.addLayer(Layer(x_train.shape[1], 750, 'relu', np.sqrt(2/750)))
mlp9.addLayer(Layer(750, 600, 'relu', np.sqrt(2/600)))
mlp9.addLayer(Layer(600, 400, 'relu', np.sqrt(2/400)))
mlp9.addLayer(Layer(400, 200, 'relu', np.sqrt(2/200)))
mlp9.addLayer(Layer(200, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
ceErrs, trAccs, valAccs9, valKAccs = mlp9.train(x_train, y_train, x_val, y_val, 0.1, 25, 529, loss='ce')

#10th Iteration
mlp10 = MLP()
mlp10.addLayer(Layer(x_train.shape[1], 750, 'relu', np.sqrt(2/750)))
mlp10.addLayer(Layer(750, 650, 'relu', np.sqrt(2/650)))
mlp10.addLayer(Layer(650, 450, 'relu', np.sqrt(2/450)))
mlp10.addLayer(Layer(450, 300, 'relu', np.sqrt(2/300)))
mlp10.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
mlp10.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
mlp10.addLayer(Layer(65, np.max(labels)+1, 'softmax', np.sqrt(2/np.max(labels)+1)))
ceErrs, trAccs, valAccs10, valKAccs = mlp10.train(x_train, y_train, x_val, y_val, 0.05, 25, 529, loss='ce')

#Final Version
mlp = MLP()
mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
                   np.sqrt(2/(np.max(labels)+1))))

ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 0.1, 25, 529, loss='ce')

print(hist.history.keys())
plt.plot(ceErrs)
plt.plot(hist.history['val_loss'])
plt.title('Validation Cross-Entropy Losses w/SGD')
plt.xlabel('Epoch')
plt.ylabel('CE Loss')
plt.legend(['Own Model', 'Keras Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(valKAccs)
plt.title('Accuracies Own Model w/SGD')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(np.asarray(hist.history['accuracy'])*100)
plt.plot(np.asarray(hist.history['val_accuracy'])*100)
plt.title('Accuracies Own Model vs. Keras w/SGD')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-Keras'])
plt.show()
np.save('valAcc_SGD.npy', valAccs)

testPred = mlp.prediction(x_test.T)
testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
print(testAcc)

fig=plt.figure(figsize=(18, 12), dpi= 80, facecolor='w', edgecolor='k')
plt.plot(valAccs1)
plt.plot(valAccs2)
plt.plot(valAccs3)
plt.plot(valAccs4)
plt.plot(valAccs5)
plt.plot(valAccs6)
plt.plot(valAccs7)
plt.plot(valAccs8)
plt.plot(valAccs9)
plt.plot(valAccs10)
plt.plot(valAccs)
plt.legend(['MLP1','MLP2','MLP3','MLP4','MLP5','MLP6','MLP7','MLP8','MLP9','MLP10','MLP*'])
plt.show()

print('MLP1\n',mlp1)
print('MLP2\n',mlp2)
print('MLP3\n',mlp3)
print('MLP4\n',mlp4)
print('MLP5\n',mlp5)
print('MLP6\n',mlp6)
print('MLP7\n',mlp7)
print('MLP8\n',mlp8)
print('MLP9\n',mlp9)
print('MLP10\n',mlp10)
print('MLP*\n',mlp)
```

Appendix 1-B – SGDM

```python
import matplotlib.pyplot as plt
import numpy as np

class Layer:
    def __init__(self, inputDim, numNeurons, activation, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation

        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

        self.prevUpdate = 0


    def actFcn(self,x):
        if(self.activation == 'sigmoid'):
            expx = np.exp(x)
            return expx/(1+expx)
        elif(self.activation == 'softmax'):
            expx = np.exp(x - np.max(x))
            return expx/np.sum(expx, axis=0)
        elif(self.activation == 'tanh'):
            return np.tanh(x)
        elif(self.activation == 'relu'):
            out = np.maximum(0,x)
            return out

    def activate(self, x):
        if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation == 
'relu':
            if(x.ndim == 1):
                x = x.reshape(x.shape[0],1)
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*1]]
            self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def derActiv(self, x):
        if(self.activation == 'sigmoid'):
            return x*(1-x)
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation == 'tanh'):
            return 1 - x**2;
        elif(self.activation == 'relu'):
            der = x
            der[x > 0] = 1
            der[x <= 0] = 0
            return der

    def __repr__(self):
        return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " + str(self.numNeur
ons) + "\n Activation: " + self.activation

class MLP:
    def __init__(self):
        self.layers = []

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

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activate(out)
        return out

    def prediction(self, inp):
        out = self.forward(inp)
        if(out.ndim == 1):
            return np.argmax(out)
        return np.argmax(out, axis=0)

    def topKaccuracy(self, inp, realOut, k):
        out = self.forward(inp)
        cnt = 0
        for i in range(out.shape[1]):
            topKind = (out[:,i]).argsort()[::-1][:k]
            if realOut[i] in topKind:
                cnt += 1
        return cnt/out.shape[1]

    def backProp(self, inp, out, lrnRate, momCoeff, batchSize, loss):
        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                if(loss == 'mse'):
                    lyr.error = out - net_out
                    derMatrix = lyr.derActiv(lyr.lastActiv)
                    lyr.delta = derMatrix * lyr.error
                elif(loss == 'ce' and lyr.activation == 'softmax'):
                    lyr.delta = net_out - out
                else:
                    assert('Cant do that')
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.derActiv(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]

            update =  (lrnRate * np.matmul(lyr.delta, inputToUse.T))/batchSize
            lyr.weightsE -= (update + momCoeff * lyr.prevUpdate)
            lyr.prevUpdate = update

    def train(self, inp, out, inpTest, outTest, lrnRate, momCoeff, epochNum, batchSize, loss):

        errList = []
        trAccs = []
        valAccs = []
        valKAccs = []

        for ep in range(epochNum):
            print('--------------------------------------------------------------
\nEpoch', ep+1)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]
            numBatches = int(np.floor(len(inp)/batchSize))

            for j in range(numBatches):
                batch_inp = inp[batchSize*j:batchSize*j+batchSize]
                batch_out = out[batchSize*j:batchSize*j+batchSize]

                batch_out_1H = mat1H2(batch_out, np.max(out)+1).T

                self.backProp(batch_inp.T, batch_out_1H, lrnRate, momCoeff, batchSize, loss)

            valOutput = self.forward(inpTest.T)
            if(loss == 'ce'):
                err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
.shape[1]
            elif(loss == 'mse'):
                err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
            print(loss.upper() +' Validation Error ', err)
            errList.append(err)

            trPred = self.prediction(inp.T)
            trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
            print('Training Accuracy: ', trAcc)
            trAccs.append(trAcc)

            valPred = self.prediction(inpTest.T)
            valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]
*100
            print('Validation Accuracy: ', valAcc)
            valAccs.append(valAcc)

            #k is chosen as three
            K = 3
            topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
            print('Top ' + str(K) + ' Accuracy: ', topKacc)
            valKAccs.append(topKacc)


        return errList, trAccs, valAccs, valKAccs

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr

features = np.load('dataMLP.npy', allow_pickle=True)
labels = np.load('labelsMLP.npy', allow_pickle=True)
labels = labels.reshape((labels.shape[0],1))

features -= np.mean(features, axis=0)
features /= np.std(features,axis=0)

#three-way split
def splitData(X, y, tr, val):
    np.random.seed(456)
    shuffle = np.random.permutation(len(y))
    X = X[shuffle]
    y = y[shuffle]

    tr_ind = int(np.floor(len(y)*tr))
    val_ind = int(np.floor(len(y)*(tr+val)))

    X_tr = X[0:tr_ind]
    y_tr = y[0:tr_ind]
    X_val = X[tr_ind:val_ind]
    y_val = y[tr_ind:val_ind]
    X_test = X[val_ind:]
    y_test = y[val_ind:]

    return X_tr, y_tr, X_val, y_val, X_test, y_test

```

```
def vector1H(x, maxInd):
    out = np.zeros(maxInd)
    out[x] = 1
    return out
def mat1H2(y, maxInd):
    out = np.zeros((y.shape[0], maxInd))
    for i in range(y.shape[0]):
        out[i,:] = vector1H(y[i], maxInd)
    return out

x_train, y_train, x_val, y_val, x_test, y_test = splitData(features, labels, 0.7, 0.2)

from tensorflow.keras import layers, models, initializers, optimizers
model = models.Sequential()
model.add(layers.Dense(500, input_shape=(11,),\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       activation='relu'))
model.add(layers.Dense(300,\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       activation='relu'))
model.add(layers.Dense(150,\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       activation = 'relu'))
model.add(layers.Dense(65,\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       activation = 'relu'))
model.add(layers.Dense(np.max(labels)+1,\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       activation='softmax'))
momOp = optimizers.SGD(lr=0.1, momentum=0.9, nesterov=False)
model.compile(loss='sparse_categorical_crossentropy', optimizer=momOp, metrics=['accuracy'])
hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)

mlp = MLP()
mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
                   np.sqrt(2/(np.max(labels)+1))))

ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, \
                                              0.1, 0.9 , 25, 529, loss='ce')

print(hist.history.keys())
plt.plot(ceErrs)
```

```python
plt.plot(hist.history['val_loss'])
plt.title('Validation Cross-Entropy Losses w/Momentum')
plt.xlabel('Epoch')
plt.ylabel('CE Loss')
plt.legend(['Own Model', 'Keras Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(valKAccs)
plt.title('Accuracies Own Model w/Momentum')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(np.asarray(hist.history['accuracy'])*100)
plt.plot(np.asarray(hist.history['val_accuracy'])*100)
plt.title('Accuracies Own Model vs. Keras w/Momentum')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-Keras'])
plt.show()

np.save('valAcc_Mom.npy', valAccs)

testPred = mlp.prediction(x_test.T)
testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
        print(testAcc)
```

Appendix 1-C – Adam

```python
import matplotlib.pyplot as plt
import numpy as np

class Layer:
    def __init__(self, inputDim, numNeurons, activation, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation

        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

        self.prevM = 0.0
        self.prevV = 0.0


    def actFcn(self,x):
        if(self.activation == 'sigmoid'):
            expx = np.exp(x)
            return expx/(1+expx)
        elif(self.activation == 'softmax'):
            expx = np.exp(x - np.max(x))
            return expx/np.sum(expx, axis=0)
        elif(self.activation == 'tanh'):
            return np.tanh(x)
        elif(self.activation == 'relu'):
            out = np.maximum(0,x)
            return out

    def activate(self, x):
        if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation == 'relu':
            if(x.ndim == 1):
                x = x.reshape(x.shape[0],1)
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*1]]
            self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def derActiv(self, x):
        if(self.activation == 'sigmoid'):
            return x*(1-x)
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation == 'tanh'):
            return 1 - x**2;
        elif(self.activation == 'relu'):
            der = x
            der[x > 0] = 1
            der[x <= 0] = 0
            return der

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

class MLP:
    def __init__(self):
        self.layers = []

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

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activate(out)
        return out

    def prediction(self, inp):
        out = self.forward(inp)
        if(out.ndim == 1):
            return np.argmax(out)
        return np.argmax(out, axis=0)

    def topKaccuracy(self, inp, realOut, k):
        out = self.forward(inp)
        cnt = 0
        for i in range(out.shape[1]):
            topKind = (out[:,i]).argsort()[::-1][:k]
            if realOut[i] in topKind:
                cnt += 1
        return cnt/out.shape[1]

    def backProp(self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                if(loss == 'mse'):
                    lyr.error = out - net_out
                    derMatrix = lyr.derActiv(lyr.lastActiv)
                    lyr.delta = derMatrix * lyr.error
                elif(loss == 'ce' and lyr.activation == 'softmax'):
                    lyr.delta = net_out - out
                else:
                    assert('Cant do that')
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.derActiv(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]

            grad = np.matmul(lyr.delta, inputToUse.T)/batchSize

            M = (1-beta1)*grad + beta1*lyr.prevM
            V = (1-beta2)*np.power(grad,2)+ beta2*lyr.prevV

            alpha_t = lrnRate * np.sqrt(1-np.power(beta2,curEpoch))/(1-np.power(beta1,curEpoch))
            lyr.weightsE -= alpha_t * M / (np.sqrt(V) + 1e-7)
            '''
            Algorithm 1 from Kingma and Ba
            M_hat = M/(1- np.power(beta1,curEpoch))
            V_hat = V/(1- np.power(beta2,curEpoch))

            lyr.weightsE -= (lrnRate*M_hat)/(np.sqrt(V_hat) + 1e-7)'''

            lyr.prevM = M
            lyr.prevV = V

    def train(self, inp, out, inpTest, outTest, epochNum, batchSize, loss, lrnRate=0.001, beta1=0.9, beta2=0.999):
        errList = []
        trAccs = []
        valAccs = []
        valKAccs = []

        for ep in range(epochNum):
            print('-------------------------------------------------------------\nEpoch', ep+1)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]
            numBatches = int(np.floor(len(inp)/batchSize))
            for j in range(numBatches):
                batch_inp = inp[batchSize*j:batchSize*j+batchSize]
                batch_out = out[batchSize*j:batchSize*j+batchSize]

                batch_out_1H = mat1H2(batch_out, np.max(out)+1).T

                #self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
                self.backProp(batch_inp.T, batch_out_1H, batchSize, loss, ep+1, lrnRate, beta1, beta2)

            valOutput = self.forward(inpTest.T)
            if(loss == 'ce'):
                err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput.shape[1]
            elif(loss == 'mse'):
                err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
            print(loss.upper() +' Validation Error ', err)
            errList.append(err)

            trPred = self.prediction(inp.T)
            trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
            print('Training Accuracy: ', trAcc)
            trAccs.append(trAcc)

            valPred = self.prediction(inpTest.T)
            valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]*100
            print('Validation Accuracy: ', valAcc)
            valAccs.append(valAcc)

            #k is chosen as three
            K = 3
            topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
            print('Top ' + str(K) + ' Accuracy: ', topKacc)
            valKAccs.append(topKacc)


        return errList, trAccs, valAccs, valKAccs

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr

features = np.load('dataMLP.npy', allow_pickle=True)
labels = np.load('labelsMLP.npy', allow_pickle=True)
labels = labels.reshape((labels.shape[0],1))
features -= np.mean(features,axis=0)
features /= np.std(features, axis=0)

#three-way split
def splitData(X, y, tr, val):
    np.random.seed(456)
    shuffle = np.random.permutation(len(y))
    X = X[shuffle]
    y = y[shuffle]

    tr_ind = int(np.floor(len(y)*tr))
    val_ind = int(np.floor(len(y)*(tr+val)))

    X_tr = X[0:tr_ind]
    y_tr = y[0:tr_ind]
    X_val = X[tr_ind:val_ind]
    y_val = y[tr_ind:val_ind]
    X_test = X[val_ind:]
    y_test = y[val_ind:]

    return X_tr, y_tr, X_val, y_val, X_test, y_test

def vector1H(x, maxInd):
    out = np.zeros(maxInd)
    out[x] = 1
    return out
def mat1H2(y, maxInd):
    out = np.zeros((y.shape[0], maxInd))
    for i in range(y.shape[0]):
        out[i,:] = vector1H(y[i], maxInd)
    return out

x_train, y_train, x_val, y_val, x_test, y_test = splitData(features, labels, 0.7, 0.2)

from tensorflow.keras import layers, models, initializers, optimizers
model = models.Sequential()
model.add(layers.Dense(500, input_shape=(11,), activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(300,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(150,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(65,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(np.max(labels)+1, activation='softmax',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))

adamOP = optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=False)
model.compile(loss='sparse_categorical_crossentropy', optimizer=adamOP, metrics=['accuracy'])

hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)

mlp = MLP()
mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
                   np.sqrt(2/(np.max(labels)+1))))

ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 25, 529,\
                                               loss='ce', lrnRate=0.001)

print(hist.history.keys())
plt.plot(ceErrs)
plt.plot(hist.history['val_loss'])
plt.title('Validation Cross-Entropy Losses w/Adam Optimizer')
plt.xlabel('Epoch')
plt.ylabel('CE Loss')
plt.legend(['Own Model', 'Keras Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(valKAccs)
plt.title('Accuracies Own Model w/Adam Optimizer')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(np.asarray(hist.history['accuracy'])*100)
plt.plot(np.asarray(hist.history['val_accuracy'])*100)
plt.title('Accuracies Own Model vs. Keras w/Adam Optimizer')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
Keras'])
plt.show()

print(mlp)
np.save('valAcc_Adam.npy', valAccs)

testPred = mlp.prediction(x_test.T)
testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
print(testAcc)
```

Appendix 1-D – AMSGrad

```
import matplotlib.pyplot as plt
import numpy as np

class Layer:
    def __init__(self, inputDim, numNeurons, activation, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation
```

```python

        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

        self.prevM = 0.0
        self.prevV_hat = 0.0
        self.prevV = 0.0


    def actFcn(self,x):
        if(self.activation == 'sigmoid'):
            expx = np.exp(x)
            return expx/(1+expx)
        elif(self.activation == 'softmax'):
            expx = np.exp(x - np.max(x))
            return expx/np.sum(expx, axis=0)
        elif(self.activation == 'tanh'):
            return np.tanh(x)
        elif(self.activation == 'relu'):
            out = np.maximum(0,x)
            return out

    def activate(self, x):
        if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation == 'relu':
            if(x.ndim == 1):
                x = x.reshape(x.shape[0],1)
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*1]]
            self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def derActiv(self, x):
        if(self.activation == 'sigmoid'):
            return x*(1-x)
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation == 'tanh'):
            return 1 - x**2;
        elif(self.activation == 'relu'):
            der = x
            der[x > 0] = 1
            der[x <= 0] = 0
            return der

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

class MLP:
    def __init__(self):
        self.layers = []

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

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activate(out)
        return out

    def prediction(self, inp):
        out = self.forward(inp)
        if(out.ndim == 1):
            return np.argmax(out)
        return np.argmax(out, axis=0)

    def topKaccuracy(self, inp, realOut, k):
        out = self.forward(inp)
        cnt = 0
        for i in range(out.shape[1]):
            topKind = (out[:,i]).argsort()[::-1][:k]
            if realOut[i] in topKind:
                cnt += 1
        return cnt/out.shape[1]

    def backProp(self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                if(loss == 'mse'):
                    lyr.error = out - net_out
                    derMatrix = lyr.derActiv(lyr.lastActiv)
                    lyr.delta = derMatrix * lyr.error
                elif(loss == 'ce' and lyr.activation == 'softmax'):
                    lyr.delta = net_out - out
                else:
                    assert('Cant do that')
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.derActiv(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]

            grad = np.matmul(lyr.delta, inputToUse.T)/batchSize

            M = (1-beta1)*grad + beta1*lyr.prevM
            V = (1-beta2)*np.power(grad,2)+ beta2*lyr.prevV

            V_hat = np.maximum(V, lyr.prevV_hat)

            lyr.weightsE -= (lrnRate*M)/(np.sqrt(V_hat) + 1e-7)

            lyr.prevV_hat = V_hat
            lyr.prevM = M
            lyr.prevV = V

    def train(self, inp, out, inpTest, outTest, epochNum, batchSize, loss, lrnRate=0.001, beta
1=0.9, beta2=0.999):
```

```python
        errList = []
        trAccs = []
        valAccs = []
        valKAccs = []

        for ep in range(epochNum):
            print('-------------------------------------------------------------\nEpoch', ep+1)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]
            numBatches = int(np.floor(len(inp)/batchSize))
            for j in range(numBatches):
                batch_inp = inp[batchSize*j:batchSize*j+batchSize]
                batch_out = out[batchSize*j:batchSize*j+batchSize]

                batch_out_1H = mat1H2(batch_out, np.max(out)+1).T

                #self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
                self.backProp(batch_inp.T, batch_out_1H, batchSize, loss, ep+1, lrnRate, beta1, beta2)

            valOutput = self.forward(inpTest.T)
            if(loss == 'ce'):
                err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput.shape[1]
            elif(loss == 'mse'):
                err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
            print(loss.upper() +' Validation Error ', err)
            errList.append(err)

            trPred = self.prediction(inp.T)
            trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
            print('Training Accuracy: ', trAcc)
            trAccs.append(trAcc)

            valPred = self.prediction(inpTest.T)
            valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]*100
            print('Validation Accuracy: ', valAcc)
            valAccs.append(valAcc)

            #k is chosen as three
            K = 3
            topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
            print('Top ' + str(K) + ' Accuracy: ', topKacc)
            valKAccs.append(topKacc)


        return errList, trAccs, valAccs, valKAccs

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr

features = np.load('dataMLP.npy', allow_pickle=True)
labels = np.load('labelsMLP.npy', allow_pickle=True)
labels = labels.reshape((labels.shape[0],1))

features -= np.mean(features, axis=0)
features /= np.std(features, axis=0)

#three-way split
```

```python
def splitData(X, y, tr, val):
    np.random.seed(456)
    shuffle = np.random.permutation(len(y))
    X = X[shuffle]
    y = y[shuffle]

    tr_ind = int(np.floor(len(y)*tr))
    val_ind = int(np.floor(len(y)*(tr+val)))

    X_tr = X[0:tr_ind]
    y_tr = y[0:tr_ind]
    X_val = X[tr_ind:val_ind]
    y_val = y[tr_ind:val_ind]
    X_test = X[val_ind:]
    y_test = y[val_ind:]

    return X_tr, y_tr, X_val, y_val, X_test, y_test

def vector1H(x, maxInd):
    out = np.zeros(maxInd)
    out[x] = 1
    return out
def mat1H2(y, maxInd):
    out = np.zeros((y.shape[0], maxInd))
    for i in range(y.shape[0]):
        out[i,:] = vector1H(y[i], maxInd)
    return out

x_train, y_train, x_val, y_val, x_test, y_test = splitData(features, labels, 0.7, 0.2)

from tensorflow.keras import layers, models, initializers, optimizers
model = models.Sequential()
model.add(layers.Dense(500, input_shape=(11,), activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(300,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(150,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(65,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(np.max(labels)+1, activation='softmax',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))

adamOP = optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=True)
model.compile(loss='sparse_categorical_crossentropy', optimizer=adamOP, metrics=['accuracy'])
hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)

mlp = MLP()
mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
                   np.sqrt(2/(np.max(labels)+1))))

ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 25, 529,\
                                               loss='ce', lrnRate=0.001)

print(hist.history.keys())
plt.plot(ceErrs)
plt.plot(hist.history['val_loss'])
plt.title('Validation Cross-Entropy Losses w/AMSGrad')
plt.xlabel('Epoch')
plt.ylabel('CE Loss')
plt.legend(['Own Model', 'Keras Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(valKAccs)
plt.title('Accuracies Own Model w/AMSGrad')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(np.asarray(hist.history['accuracy'])*100)
plt.plot(np.asarray(hist.history['val_accuracy'])*100)
plt.title('Accuracies Own Model vs. Keras w/AMSGrad')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-Keras'])
plt.show()
```

```

np.save('valAcc_AMSGrad.npy', valAccs)

testPred = mlp.prediction(x_test.T)
testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
print(testAcc)
```

Appendix 1-E – SGD with PCA

```
import matplotlib.pyplot as plt
import numpy as np

class Layer:
    def __init__(self, inputDim, numNeurons, activation, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation

        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

    def actFcn(self,x):
        if(self.activation == 'sigmoid'):
            expx = np.exp(x)
            return expx/(1+expx)
        elif(self.activation == 'softmax'):
            expx = np.exp(x - np.max(x))
            return expx/np.sum(expx, axis=0)
        elif(self.activation == 'tanh'):
            return np.tanh(x)
        elif(self.activation == 'relu'):
            out = np.maximum(0,x)
            return out

    def activate(self, x):
        if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation == 'relu':
            if(x.ndim == 1):
                x = x.reshape(x.shape[0],1)
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*1]]
            self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def derActiv(self, x):
        if(self.activation == 'sigmoid'):
            return x*(1-x)
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation == 'tanh'):
            return 1 - x**2;
        elif(self.activation == 'relu'):
            der = x
            der[x > 0] = 1
            der[x <= 0] = 0
            return der

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

class MLP:
    def __init__(self):
        self.layers = []

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

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activate(out)
        return out

    def prediction(self, inp):
        out = self.forward(inp)
        if(out.ndim == 1):
            return np.argmax(out)
        return np.argmax(out, axis=0)

    def topKaccuracy(self, inp, realOut, k):
        out = self.forward(inp)
        cnt = 0
        for i in range(out.shape[1]):
            topKind = (out[:,i]).argsort()[::-1][:k]
            if realOut[i] in topKind:
                cnt += 1
        return cnt/out.shape[1]

    def backProp(self, inp, out, lrnRate, batchSize, loss):
        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                if(loss == 'mse'):
                    lyr.error = net_out -out
                    derMatrix = lyr.derActiv(lyr.lastActiv)
                    lyr.delta = derMatrix * lyr.error
                elif(loss == 'ce' and lyr.activation == 'softmax'):
                    lyr.delta = net_out - out
                else:
                    assert('Cant do that')
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.derActiv(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]

            update =  (lrnRate * np.matmul(lyr.delta, inputToUse.T))/batchSize
```

```python
            lyr.weightsE -= update

    def train(self, inp, out, inpTest, outTest, lrnRate, epochNum, batchSize, loss):
        errList = []
        trAccs = []
        valAccs = []
        valKAccs = []

        for ep in range(epochNum):
            print('--------------------------------------------------------------\nEpoch', ep+1)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]
            numBatches = int(np.floor(len(inp)/batchSize))

            for j in range(numBatches):
                batch_inp = inp[batchSize*j:batchSize*j+batchSize]
                batch_out = out[batchSize*j:batchSize*j+batchSize]

                batch_out_1H = mat1H2(batch_out, np.max(out)+1).T

                self.backProp(batch_inp.T, batch_out_1H, lrnRate, batchSize, loss)

            valOutput = self.forward(inpTest.T)
            if(loss == 'ce'):
                err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput.shape[1]
            elif(loss == 'mse'):
                err = np.sum((outTest.T - self.forward(inpTest.T))**2)/valOutput.shape[1]
            print(loss.upper() +' Validation Error ', err)
            errList.append(err)

            trPred = self.prediction(inp.T)
            trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
            print('Training Accuracy: ', trAcc)
            trAccs.append(trAcc)

            valPred = self.prediction(inpTest.T)
            valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]*100
            print('Validation Accuracy: ', valAcc)
            valAccs.append(valAcc)

            #k is chosen as three
            K = 3
            topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
            print('Top ' + str(K) + ' Accuracy: ', topKacc)
            valKAccs.append(topKacc)


        return errList, trAccs, valAccs, valKAccs

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr

features = np.load('dataMLP.npy', allow_pickle=True)
labels = np.load('labelsMLP.npy', allow_pickle=True)
labels = labels.reshape((labels.shape[0],1))

def PCA(eigenvalues, eigenvectors, k):
    #argsort the eigenvalues in descending order
    eigvals_ind = np.argsort(eigenvalues)[::-1]
    #sort the eigenvalues and eigenvectors accordingly
    eigenvalues = eigenvalues[eigvals_ind]
    eigenvectors = eigenvectors[:,eigvals_ind]
    #select k biggest eigenvectors
    eigfcs = eigenvectors[:,:k]
    #get the transpose of the eigenmatrix
    eigfcs = eigfcs.T
    return eigfcs

def reconstruct(data_mn, eigfcs, mean):
    #data==D, eigenfaces==E
    #reconstructed_data = (D*E')*E + mean
    proj = np.matmul(data_mn,eigfcs.T)
    proj = np.matmul(proj,eigfcs) + mean
    return proj

mean = np.mean(features, axis=0)
data_mn = features - mean
data_mn /= np.std(data_mn, axis=0)

cov_mat = np.matmul(data_mn.T,data_mn)
eigenvalues, eigenvectors = np.linalg.eigh(cov_mat)

#explained variances
#argsort the eigenvalues in descending order
eigvals_ind = np.argsort(eigenvalues)[::-1]
#sort the eigenvalues and eigenvectors accordingly
eigenvalues = eigenvalues[eigvals_ind]
eigenvectors = eigenvectors[:,eigvals_ind]
k_vals = np.array([1,2,3,4,5,6,7,8,9,10,11])
tot_var = np.sum(eigenvalues)
vas = np.zeros(k_vals.shape[0])
for ind in range(k_vals.shape[0]):
    vas[ind] = np.sum(eigenvalues[:k_vals[ind]]/tot_var*100.0)
print("Percent Explained Variances")
for i in range(vas.shape[0]):
    print("PEV for k=" + str(k_vals[i]) + " ", vas[i])
plt.plot(k_vals,vas)
plt.xlabel("Number of Eigenvalues -- k")
plt.ylabel("Percent Explained Variance")
plt.grid()
plt.show()

k = 8
pcs = PCA(eigenvalues,eigenvectors,k)
ext_features = reconstruct(data_mn,pcs,mean)
print(ext_features.shape)

#three-way split
def splitData(X, y, tr, val):
    np.random.seed(456)
    shuffle = np.random.permutation(len(y))
    X = X[shuffle]
    y = y[shuffle]

    tr_ind = int(np.floor(len(y)*tr))
    val_ind = int(np.floor(len(y)*(tr+val)))

    X_tr = X[0:tr_ind]
    y_tr = y[0:tr_ind]
    X_val = X[tr_ind:val_ind]
    y_val = y[tr_ind:val_ind]
    X_test = X[val_ind:]
    y_test = y[val_ind:]

    return X_tr, y_tr, X_val, y_val, X_test, y_test

def vector1H(x, maxInd):
    out = np.zeros(maxInd)
    out[x] = 1
    return out
def mat1H2(y, maxInd):
    out = np.zeros((y.shape[0], maxInd))
    for i in range(y.shape[0]):
        out[i,:] = vector1H(y[i], maxInd)
    return out

x_train, y_train, x_val, y_val, x_test, y_test = splitData(ext_features, labels, 0.7, 0.2)

from tensorflow.keras import layers, models, initializers, optimizers
model = models.Sequential()
model.add(layers.Dense(500, input_shape=(11,), activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(300,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(150,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(65,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(np.max(labels)+1, activation='softmax',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                      distribution = 'normal'
, \
                                                                      seed=None, \
                                                                      scale=2.0)))
momOp = optimizers.SGD(lr=0.1, momentum=0, nesterov=False, decay=0)
model.compile(loss='sparse_categorical_crossentropy', optimizer=momOp, metrics=['accuracy'])
hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)

#Final Version
mlp = MLP()
mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
                   np.sqrt(2/(np.max(labels)+1))))

ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 0.1, 25, 529, lo
ss='ce')

print(hist.history.keys())
plt.plot(ceErrs)
plt.plot(hist.history['val_loss'])
plt.title('Validation Cross-Entropy Losses w/SGD')
plt.xlabel('Epoch')
plt.ylabel('CE Loss')
plt.legend(['Own Model', 'Keras Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(valKAccs)
plt.title('Accuracies Own Model w/SGD')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(np.asarray(hist.history['accuracy'])*100)
plt.plot(np.asarray(hist.history['val_accuracy'])*100)
plt.title('Accuracies Own Model vs. Keras w/SGD')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
Keras'])
plt.show()

np.save('valAcc_SGD_pca.npy', valAccs)
testPred = mlp.prediction(x_test.T)
testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
print(testAcc)
```

Appendix 1-F – SGDM with PCA

```
import matplotlib.pyplot as plt
import numpy as np

class Layer:
    def __init__(self, inputDim, numNeurons, activation, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation
```

```python

        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

        self.prevUpdate = 0


    def actFcn(self,x):
        if(self.activation == 'sigmoid'):
            expx = np.exp(x)
            return expx/(1+expx)
        elif(self.activation == 'softmax'):
            expx = np.exp(x - np.max(x))
            return expx/np.sum(expx, axis=0)
        elif(self.activation == 'tanh'):
            return np.tanh(x)
        elif(self.activation == 'relu'):
            out = np.maximum(0,x)
            return out

    def activate(self, x):
        if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation == 'relu':
            if(x.ndim == 1):
                x = x.reshape(x.shape[0],1)
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*1]]
            self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def derActiv(self, x):
        if(self.activation == 'sigmoid'):
            return x*(1-x)
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation == 'tanh'):
            return 1 - x**2;
        elif(self.activation == 'relu'):
            der = x
            der[x > 0] = 1
            der[x <= 0] = 0
            return der

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

class MLP:
    def __init__(self):
        self.layers = []

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

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activate(out)
        return out

    def prediction(self, inp):
        out = self.forward(inp)
        if(out.ndim == 1):
            return np.argmax(out)
        return np.argmax(out, axis=0)

    def topKaccuracy(self, inp, realOut, k):
        out = self.forward(inp)
        cnt = 0
        for i in range(out.shape[1]):
            topKind = (out[:,i]).argsort()[::-1][:k]
            if realOut[i] in topKind:
                cnt += 1
        return cnt/out.shape[1]

    def backProp(self, inp, out, lrnRate, momCoeff, batchSize, loss):
        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                if(loss == 'mse'):
                    lyr.error = out - net_out
                    derMatrix = lyr.derActiv(lyr.lastActiv)
                    lyr.delta = derMatrix * lyr.error
                elif(loss == 'ce' and lyr.activation == 'softmax'):
                    lyr.delta = net_out - out
                else:
                    assert('Cant do that')
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.derActiv(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]

            update =  (lrnRate * np.matmul(lyr.delta, inputToUse.T))/batchSize
            lyr.weightsE -= (update + momCoeff * lyr.prevUpdate)
            lyr.prevUpdate = update

    def train(self, inp, out, inpTest, outTest, lrnRate, momCoeff, epochNum, batchSize, loss):

        errList = []
        trAccs = []
        valAccs = []
        valKAccs = []

        for ep in range(epochNum):
            print('-------------------------------------------------------------
\nEpoch', ep+1)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]
            numBatches = int(np.floor(len(inp)/batchSize))

            for j in range(numBatches):
                batch_inp = inp[batchSize*j:batchSize*j+batchSize]
                batch_out = out[batchSize*j:batchSize*j+batchSize]

                batch_out_1H = mat1H2(batch_out, np.max(out)+1).T

                self.backProp(batch_inp.T, batch_out_1H, lrnRate, momCoeff, batchSize, loss)

            valOutput = self.forward(inpTest.T)
            if(loss == 'ce'):
                err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
.shape[1]
            elif(loss == 'mse'):
                err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
            print(loss.upper() +' Validation Error ', err)
            errList.append(err)

            trPred = self.prediction(inp.T)
            trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
            print('Training Accuracy: ', trAcc)
            trAccs.append(trAcc)

            valPred = self.prediction(inpTest.T)
            valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]
*100
            print('Validation Accuracy: ', valAcc)
            valAccs.append(valAcc)

            #k is chosen as three
            K = 3
            topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
            print('Top ' + str(K) + ' Accuracy: ', topKacc)
            valKAccs.append(topKacc)


        return errList, trAccs, valAccs, valKAccs

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr

features = np.load('dataMLP.npy', allow_pickle=True)
labels = np.load('labelsMLP.npy', allow_pickle=True)
labels = labels.reshape((labels.shape[0],1))

def PCA(eigenvalues, eigenvectors, k):
    #argsort the eigenvalues in descending order
    eigvals_ind = np.argsort(eigenvalues)[::-1]
    #sort the eigenvalues and eigenvectors accordingly
    eigenvalues = eigenvalues[eigvals_ind]
    eigenvectors = eigenvectors[:,eigvals_ind]
    #select k biggest eigenvectors
    eigfcs = eigenvectors[:,:k]
    #get the transpose of the eigenmatrix
    eigfcs = eigfcs.T
    return eigfcs

def reconstruct(data_mn, eigfcs, mean):
    #data==D, eigenfaces==E
    #reconstructed_data = (D*E')*E + mean
    proj = np.matmul(data_mn,eigfcs.T)
    proj = np.matmul(proj,eigfcs) + mean
    return proj

mean = np.mean(features, axis=0)
data_mn = features - mean
data_mn /= np.std(data_mn, axis=0)

cov_mat = np.matmul(data_mn.T,data_mn)
eigenvalues, eigenvectors = np.linalg.eigh(cov_mat)

#explained variances
#argsort the eigenvalues in descending order
eigvals_ind = np.argsort(eigenvalues)[::-1]
#sort the eigenvalues and eigenvectors accordingly
eigenvalues = eigenvalues[eigvals_ind]
eigenvectors = eigenvectors[:,eigvals_ind]
k_vals = np.array([1,2,3,4,5,6,7,8,9,10,11])
tot_var = np.sum(eigenvalues)
vas = np.zeros(k_vals.shape[0])
for ind in range(k_vals.shape[0]):
    vas[ind] = np.sum(eigenvalues[:k_vals[ind]]/tot_var*100.0)
print("Percent Explained Variances")
for i in range(vas.shape[0]):
    print("PEV for k=" + str(k_vals[i]) + " ", vas[i])
plt.plot(k_vals,vas)
plt.xlabel("Number of Eigenvalues -- k")
plt.ylabel("Percent Explained Variance")
plt.grid()
plt.show()

k = 8
pcs = PCA(eigenvalues,eigenvectors,k)
ext_features = reconstruct(data_mn,pcs,mean)
print(ext_features.shape)

#three-way split
def splitData(X, y, tr, val):
    np.random.seed(456)
    shuffle = np.random.permutation(len(y))
    X = X[shuffle]
    y = y[shuffle]

    tr_ind = int(np.floor(len(y)*tr))
    val_ind = int(np.floor(len(y)*(tr+val)))

    X_tr = X[0:tr_ind]
    y_tr = y[0:tr_ind]
    X_val = X[tr_ind:val_ind]
    y_val = y[tr_ind:val_ind]
    X_test = X[val_ind:]
    y_test = y[val_ind:]

    return X_tr, y_tr, X_val, y_val, X_test, y_test

def vector1H(x, maxInd):
    out = np.zeros(maxInd)
    out[x] = 1
    return out
def mat1H2(y, maxInd):
    out = np.zeros((y.shape[0], maxInd))
    for i in range(y.shape[0]):
        out[i,:] = vector1H(y[i], maxInd)
    return out

x_train, y_train, x_val, y_val, x_test, y_test = splitData(ext_features, labels, 0.7, 0.2)

```

```python
from tensorflow.keras import layers, models, initializers, optimizers
model = models.Sequential()
model.add(layers.Dense(500, input_shape=(11,),\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       activation='relu'))
model.add(layers.Dense(300,\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       activation='relu'))
model.add(layers.Dense(150,\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       activation = 'relu'))
model.add(layers.Dense(65,\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       activation = 'relu'))
model.add(layers.Dense(np.max(labels)+1,\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       activation='softmax'))
momOp = optimizers.SGD(lr=0.1, momentum=0.9, nesterov=False)
model.compile(loss='sparse_categorical_crossentropy', optimizer=momOp, metrics=['accuracy'])
hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)

mlp = MLP()
mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
                   np.sqrt(2/(np.max(labels)+1))))

ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, \
                                              0.1, 0.9 , 25, 529, loss='ce')

print(hist.history.keys())
plt.plot(ceErrs)
plt.plot(hist.history['val_loss'])
plt.title('Validation Cross-Entropy Losses w/Momentum')
plt.xlabel('Epoch')
plt.ylabel('CE Loss')
plt.legend(['Own Model', 'Keras Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(valKAccs)
plt.title('Accuracies Own Model w/Momentum')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(np.asarray(hist.history['accuracy'])*100)
plt.plot(np.asarray(hist.history['val_accuracy'])*100)
plt.title('Accuracies Own Model vs. Keras w/Momentum')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-Keras'])
plt.show()

np.save('valAcc_Mom_pca.npy', valAccs)

testPred = mlp.prediction(x_test.T)
testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
print(testAcc)
```

Appendix 1-G – Adam with PCA

```
import matplotlib.pyplot as plt
import numpy as np

class Layer:
    def __init__(self, inputDim, numNeurons, activation, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation

        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

        self.prevM = 0.0
        self.prevV = 0.0


    def actFcn(self,x):
        if(self.activation == 'sigmoid'):
            expx = np.exp(x)
            return expx/(1+expx)
        elif(self.activation == 'softmax'):
            expx = np.exp(x - np.max(x))
            return expx/np.sum(expx, axis=0)
        elif(self.activation == 'tanh'):
            return np.tanh(x)
        elif(self.activation == 'relu'):
            out = np.maximum(0,x)
            return out

    def activate(self, x):
        if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation == 'relu':
            if(x.ndim == 1):
                x = x.reshape(x.shape[0],1)
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*1]]
```

```python
            self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def derActiv(self, x):
        if(self.activation == 'sigmoid'):
            return x*(1-x)
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation == 'tanh'):
            return 1 - x**2;
        elif(self.activation == 'relu'):
            der = x
            der[x > 0] = 1
            der[x <= 0] = 0
            return der

    def __repr__(self):
        return "Input Dim: " + str(self.inputDim) + ", Number of Neurons: " + str(self.numNeur
ons) + "\n Activation: " + self.activation

class MLP:
    def __init__(self):
        self.layers = []

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

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activate(out)
        return out

    def prediction(self, inp):
        out = self.forward(inp)
        if(out.ndim == 1):
            return np.argmax(out)
        return np.argmax(out, axis=0)

    def topKaccuracy(self, inp, realOut, k):
        out = self.forward(inp)
        cnt = 0
        for i in range(out.shape[1]):
            topKind = (out[:,i]).argsort()[::-1][:k]
            if realOut[i] in topKind:
                cnt += 1
        return cnt/out.shape[1]

    def backProp(self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                if(loss == 'mse'):
                    lyr.error = out - net_out
                    derMatrix = lyr.derActiv(lyr.lastActiv)
                    lyr.delta = derMatrix * lyr.error
                elif(loss == 'ce' and lyr.activation == 'softmax'):
                    lyr.delta = net_out - out
                else:
                    assert('Cant do that')
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.derActiv(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]

            grad = np.matmul(lyr.delta, inputToUse.T)/batchSize

            M = (1-beta1)*grad + beta1*lyr.prevM
            V = (1-beta2)*np.power(grad,2)+ beta2*lyr.prevV

            alpha_t = lrnRate * np.sqrt(1-np.power(beta2,curEpoch))/(1-
np.power(beta1,curEpoch))
            lyr.weightsE -= alpha_t * M / (np.sqrt(V) + 1e-7)
            '''
            Algorithm 1 from Kingma and Ba
            M_hat = M/(1- np.power(beta1,curEpoch))
            V_hat = V/(1- np.power(beta2,curEpoch))

            lyr.weightsE -= (lrnRate*M_hat)/(np.sqrt(V_hat) + 1e-7)'''

            lyr.prevM = M
            lyr.prevV = V

    def train(self, inp, out, inpTest, outTest, epochNum, batchSize, loss, lrnRate=0.001, beta
1=0.9, beta2=0.999):
        errList = []
        trAccs = []
        valAccs = []
        valKAccs = []

        for ep in range(epochNum):
            print('---------------------------------------------------------------
\nEpoch', ep+1)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]
            numBatches = int(np.floor(len(inp)/batchSize))
            for j in range(numBatches):
                batch_inp = inp[batchSize*j:batchSize*j+batchSize]
                batch_out = out[batchSize*j:batchSize*j+batchSize]

                batch_out_1H = mat1H2(batch_out, np.max(out)+1).T

                #self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
                self.backProp(batch_inp.T, batch_out_1H, batchSize, loss, ep+1, lrnRate, beta1
, beta2)

            valOutput = self.forward(inpTest.T)
            if(loss == 'ce'):
                err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput
.shape[1]
            elif(loss == 'mse'):
                err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
            print(loss.upper() +' Validation Error ', err)
```

```python
            errList.append(err)

            trPred = self.prediction(inp.T)
            trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
            print('Training Accuracy: ', trAcc)
            trAccs.append(trAcc)

            valPred = self.prediction(inpTest.T)
            valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]*100
            print('Validation Accuracy: ', valAcc)
            valAccs.append(valAcc)

            #k is chosen as three
            K = 3
            topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
            print('Top ' + str(K) + ' Accuracy: ', topKacc)
            valKAccs.append(topKacc)


        return errList, trAccs, valAccs, valKAccs

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr

features = np.load('dataMLP.npy', allow_pickle=True)
labels = np.load('labelsMLP.npy', allow_pickle=True)
labels = labels.reshape((labels.shape[0],1))

def PCA(eigenvalues, eigenvectors, k):
    #argsort the eigenvalues in descending order
    eigvals_ind = np.argsort(eigenvalues)[::-1]
    #sort the eigenvalues and eigenvectors accordingly
    eigenvalues = eigenvalues[eigvals_ind]
    eigenvectors = eigenvectors[:,eigvals_ind]
    #select k biggest eigenvectors
    eigfcs = eigenvectors[:,:k]
    #get the transpose of the eigenmatrix
    eigfcs = eigfcs.T
    return eigfcs

def reconstruct(data_mn, eigfcs, mean):
    #data==D, eigenfaces==E
    #reconstructed_data = (D*E')*E + mean
    proj = np.matmul(data_mn,eigfcs.T)
    proj = np.matmul(proj,eigfcs) + mean
    return proj

mean = np.mean(features, axis=0)
data_mn = features - mean
data_mn /= np.std(data_mn, axis=0)

cov_mat = np.matmul(data_mn.T,data_mn)
eigenvalues, eigenvectors = np.linalg.eigh(cov_mat)

#explained variances
#argsort the eigenvalues in descending order
eigvals_ind = np.argsort(eigenvalues)[::-1]
#sort the eigenvalues and eigenvectors accordingly
eigenvalues = eigenvalues[eigvals_ind]
eigenvectors = eigenvectors[:,eigvals_ind]
k_vals = np.array([1,2,3,4,5,6,7,8,9,10,11])
tot_var = np.sum(eigenvalues)
```

```python
vas = np.zeros(k_vals.shape[0])
for ind in range(k_vals.shape[0]):
    vas[ind] = np.sum(eigenvalues[:k_vals[ind]]/tot_var*100.0)
print("Percent Explained Variances")
for i in range(vas.shape[0]):
    print("PEV for k=" + str(k_vals[i]) + " ", vas[i])
plt.plot(k_vals,vas)
plt.xlabel("Number of Eigenvalues -- k")
plt.ylabel("Percent Explained Variance")
plt.grid()
plt.show()

k = 8
pcs = PCA(eigenvalues,eigenvectors,k)
ext_features = reconstruct(data_mn,pcs,mean)
print(ext_features.shape)

#three-way split
def splitData(X, y, tr, val):
    np.random.seed(456)
    shuffle = np.random.permutation(len(y))
    X = X[shuffle]
    y = y[shuffle]

    tr_ind = int(np.floor(len(y)*tr))
    val_ind = int(np.floor(len(y)*(tr+val)))

    X_tr = X[0:tr_ind]
    y_tr = y[0:tr_ind]
    X_val = X[tr_ind:val_ind]
    y_val = y[tr_ind:val_ind]
    X_test = X[val_ind:]
    y_test = y[val_ind:]

    return X_tr, y_tr, X_val, y_val, X_test, y_test

def vector1H(x, maxInd):
    out = np.zeros(maxInd)
    out[x] = 1
    return out
def mat1H2(y, maxInd):
    out = np.zeros((y.shape[0], maxInd))
    for i in range(y.shape[0]):
        out[i,:] = vector1H(y[i], maxInd)
    return out

x_train, y_train, x_val, y_val, x_test, y_test = splitData(ext_features, labels, 0.7, 0.2)

from tensorflow.keras import layers, models, initializers, optimizers
model = models.Sequential()
model.add(layers.Dense(500, input_shape=(11,), activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal'
, \
                                                                       seed=None, \
                                                                       scale=2.0),\
                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal'
, \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(300,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal'
, \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(150,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(65,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(np.max(labels)+1, activation='softmax',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))

adamOP = optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=False)
model.compile(loss='sparse_categorical_crossentropy', optimizer=adamOP, metrics=['accuracy'])

hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)

mlp = MLP()
mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
                   np.sqrt(2/(np.max(labels)+1))))

ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 25, 529,\
                                               loss='ce', lrnRate=0.001)

print(hist.history.keys())
plt.plot(ceErrs)
plt.plot(hist.history['val_loss'])
plt.title('Validation Cross-Entropy Losses w/Adam Optimizer')
plt.xlabel('Epoch')
plt.ylabel('CE Loss')
plt.legend(['Own Model', 'Keras Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(valKAccs)
plt.title('Accuracies Own Model w/Adam Optimizer')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(np.asarray(hist.history['accuracy'])*100)
plt.plot(np.asarray(hist.history['val_accuracy'])*100)
plt.title('Accuracies Own Model vs. Keras w/Adam Optimizer')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-Keras'])
plt.show()

print(mlp)
np.save('valAcc_Adam_pca.npy', valAccs)
testPred = mlp.prediction(x_test.T)
testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
print(testAcc)
```

Appendix 1-H – AMSGrad with PCA

```
import matplotlib.pyplot as plt
import numpy as np

class Layer:
    def __init__(self, inputDim, numNeurons, activation, std, mean=0):
        self.inputDim = inputDim
        self.numNeurons = numNeurons
        self.activation = activation

        self.weights = np.random.normal(mean,std, inputDim*numNeurons).reshape(numNeurons, inputDim)
        self.biases = np.random.normal(mean,std, numNeurons).reshape(numNeurons,1)
        self.weightsE = np.concatenate((self.weights, self.biases), axis=1)

        self.delta = None
        self.error = None
        self.lastActiv = None

        self.prevM = 0.0
        self.prevV_hat = 0.0
        self.prevV = 0.0


    def actFcn(self,x):
        if(self.activation == 'sigmoid'):
            expx = np.exp(x)
            return expx/(1+expx)
        elif(self.activation == 'softmax'):
            expx = np.exp(x - np.max(x))
            return expx/np.sum(expx, axis=0)
        elif(self.activation == 'tanh'):
            return np.tanh(x)
        elif(self.activation == 'relu'):
            out = np.maximum(0,x)
            return out

    def activate(self, x):
```

```python
        if self.activation == 'sigmoid' or self.activation == 'softmax' or self.activation == 'relu':
            if(x.ndim == 1):
                x = x.reshape(x.shape[0],1)
            numSamples = x.shape[1]
            tempInp = np.r_[x, [np.ones(numSamples)*1]]
            self.lastActiv = self.actFcn(np.matmul(self.weightsE, tempInp))
        return self.lastActiv

    def derActiv(self, x):
        if(self.activation == 'sigmoid'):
            return x*(1-x)
        elif(self.activation == 'softmax'):
            return x*(1-x)
        elif(self.activation == 'tanh'):
            return 1 - x**2;
        elif(self.activation == 'relu'):
            der = x
            der[x > 0] = 1
            der[x <= 0] = 0
            return der

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

class MLP:
    def __init__(self):
        self.layers = []

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

    def forward(self, inp):
        out = inp
        for lyr in self.layers:
            out = lyr.activate(out)
        return out

    def prediction(self, inp):
        out = self.forward(inp)
        if(out.ndim == 1):
            return np.argmax(out)
        return np.argmax(out, axis=0)

    def topKaccuracy(self, inp, realOut, k):
        out = self.forward(inp)
        cnt = 0
        for i in range(out.shape[1]):
            topKind = (out[:,i]).argsort()[::-1][:k]
            if realOut[i] in topKind:
                cnt += 1
        return cnt/out.shape[1]

    def backProp(self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
        net_out = self.forward(inp)
        for i in reversed(range(len(self.layers))):
            lyr = self.layers[i]
            #outputLayer
            if(lyr == self.layers[-1]):
                if(loss == 'mse'):
                    lyr.error = out - net_out
                    derMatrix = lyr.derActiv(lyr.lastActiv)
                    lyr.delta = derMatrix * lyr.error
                elif(loss == 'ce' and lyr.activation == 'softmax'):
                    lyr.delta = net_out - out
                else:
                    assert('Cant do that')
            #hiddenLayer
            else:
                nextLyr = self.layers[i+1]
                nextLyr.weights = nextLyr.weightsE[:,0:nextLyr.weights.shape[1]]
                lyr.error = np.matmul(nextLyr.weights.T, nextLyr.delta)
                derMatrix = lyr.derActiv(lyr.lastActiv)
                lyr.delta = derMatrix * lyr.error

        #update weights
        for i in range(len(self.layers)):
            lyr = self.layers[i]
            if(i == 0):
                if(inp.ndim == 1):
                    inp = inp.reshape(inp.shape[0],1)
                numSamples = inp.shape[1]
                inputToUse = inputToUse = np.r_[inp, [np.ones(numSamples)*1]]
            else:
                numSamples = self.layers[i - 1].lastActiv.shape[1]
                inputToUse = np.r_[self.layers[i - 1].lastActiv, [np.ones(numSamples)*1]]

            grad = np.matmul(lyr.delta, inputToUse.T)/batchSize

            M = (1-beta1)*grad + beta1*lyr.prevM
            V = (1-beta2)*np.power(grad,2)+ beta2*lyr.prevV

            V_hat = np.maximum(V, lyr.prevV_hat)

            lyr.weightsE -= (lrnRate*M)/(np.sqrt(V_hat) + 1e-7)

            lyr.prevV_hat = V_hat
            lyr.prevM = M
            lyr.prevV = V

    def train(self, inp, out, inpTest, outTest, epochNum, batchSize, loss, lrnRate=0.001, beta1=0.9, beta2=0.999):
        errList = []
        trAccs = []
        valAccs = []
        valKAccs = []

        for ep in range(epochNum):
            print('--------------------------------------------------------------\nEpoch', ep+1)

            randomIndexes = np.random.permutation(len(inp))
            inp = inp[randomIndexes]
            out = out[randomIndexes]
            numBatches = int(np.floor(len(inp)/batchSize))
            for j in range(numBatches):
                batch_inp = inp[batchSize*j:batchSize*j+batchSize]
                batch_out = out[batchSize*j:batchSize*j+batchSize]

                batch_out_1H = mat1H2(batch_out, np.max(out)+1).T

                #self, inp, out, batchSize, loss, curEpoch,lrnRate, beta1, beta2):
                self.backProp(batch_inp.T, batch_out_1H, batchSize, loss, ep+1, lrnRate, beta1, beta2)

            valOutput = self.forward(inpTest.T)
            if(loss == 'ce'):
                err = - np.sum(np.log(valOutput) * mat1H2(outTest, np.max(out)+1).T)/valOutput.shape[1]
            elif(loss == 'mse'):
                err = np.sum((outTest.T - self.forward(inpTest.T))**2)/val
```

```python
            print(loss.upper() +' Validation Error ', err)
            errList.append(err)

            trPred = self.prediction(inp.T)
            trAcc = np.sum(trPred.reshape((trPred.shape[0],1)) == out)/trPred.shape[0]*100
            print('Training Accuracy: ', trAcc)
            trAccs.append(trAcc)

            valPred = self.prediction(inpTest.T)
            valAcc = np.sum(valPred.reshape((valPred.shape[0],1)) == outTest)/valPred.shape[0]*100
            print('Validation Accuracy: ', valAcc)
            valAccs.append(valAcc)

            #k is chosen as three
            K = 3
            topKacc = self.topKaccuracy(inpTest.T, outTest, K)*100
            print('Top ' + str(K) + ' Accuracy: ', topKacc)
            valKAccs.append(topKacc)


        return errList, trAccs, valAccs, valKAccs

    def __repr__(self):
        retStr = ""
        for i, lyr in enumerate(self.layers):
            retStr += "Layer " + str(i) + ": " + lyr.__repr__() + "\n"
        return retStr

features = np.load('dataMLP.npy', allow_pickle=True)
labels = np.load('labelsMLP.npy', allow_pickle=True)
labels = labels.reshape((labels.shape[0],1))

def PCA(eigenvalues, eigenvectors, k):
    #argsort the eigenvalues in descending order
    eigvals_ind = np.argsort(eigenvalues)[::-1]
    #sort the eigenvalues and eigenvectors accordingly
    eigenvalues = eigenvalues[eigvals_ind]
    eigenvectors = eigenvectors[:,eigvals_ind]
    #select k biggest eigenvectors
    eigfcs = eigenvectors[:,:k]
    #get the transpose of the eigenmatrix
    eigfcs = eigfcs.T
    return eigfcs

def reconstruct(data_mn, eigfcs, mean):
    #data==D, eigenfaces==E
    #reconstructed_data = (D*E')*E + mean
    proj = np.matmul(data_mn,eigfcs.T)
    proj = np.matmul(proj,eigfcs) + mean
    return proj

mean = np.mean(features, axis=0)
data_mn = features - mean
data_mn /= np.std(data_mn, axis=0)

cov_mat = np.matmul(data_mn.T,data_mn)
eigenvalues, eigenvectors = np.linalg.eigh(cov_mat)

#explained variances
#argsort the eigenvalues in descending order
eigvals_ind = np.argsort(eigenvalues)[::-1]
#sort the eigenvalues and eigenvectors accordingly
eigenvalues = eigenvalues[eigvals_ind]
eigenvectors = eigenvectors[:,eigvals_ind]
k_vals = np.array([1,2,3,4,5,6,7,8,9,10,11])
```

```python
tot_var = np.sum(eigenvalues)
vas = np.zeros(k_vals.shape[0])
for ind in range(k_vals.shape[0]):
    vas[ind] = np.sum(eigenvalues[:k_vals[ind]]/tot_var*100.0)
print("Percent Explained Variances")
for i in range(vas.shape[0]):
    print("PEV for k=" + str(k_vals[i]) + " ", vas[i])
plt.plot(k_vals,vas)
plt.xlabel("Number of Eigenvalues -- k")
plt.ylabel("Percent Explained Variance")
plt.grid()
plt.show()

k = 8
pcs = PCA(eigenvalues,eigenvectors,k)
ext_features = reconstruct(data_mn,pcs,mean)
print(ext_features.shape)

#three-way split
def splitData(X, y, tr, val):
    np.random.seed(456)
    shuffle = np.random.permutation(len(y))
    X = X[shuffle]
    y = y[shuffle]

    tr_ind = int(np.floor(len(y)*tr))
    val_ind = int(np.floor(len(y)*(tr+val)))

    X_tr = X[0:tr_ind]
    y_tr = y[0:tr_ind]
    X_val = X[tr_ind:val_ind]
    y_val = y[tr_ind:val_ind]
    X_test = X[val_ind:]
    y_test = y[val_ind:]

    return X_tr, y_tr, X_val, y_val, X_test, y_test

def vector1H(x, maxInd):
    out = np.zeros(maxInd)
    out[x] = 1
    return out
def mat1H2(y, maxInd):
    out = np.zeros((y.shape[0], maxInd))
    for i in range(y.shape[0]):
        out[i,:] = vector1H(y[i], maxInd)
    return out

x_train, y_train, x_val, y_val, x_test, y_test = splitData(ext_features, labels, 0.7, 0.2)

from tensorflow.keras import layers, models, initializers, optimizers
model = models.Sequential()
model.add(layers.Dense(500, input_shape=(11,), activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                        bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(300,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(150,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(65,activation='relu',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))
model.add(layers.Dense(np.max(labels)+1, activation='softmax',\
                       kernel_initializer=initializers.VarianceScaling(mode='fan_out',\
                                                                       distribution = 'normal',\
                                                                       seed=None,\
                                                                       scale=2.0),\
                       bias_initializer = initializers.VarianceScaling(mode='fan_out', \
                                                                       distribution = 'normal', \
                                                                       seed=None, \
                                                                       scale=2.0)))

adamOP = optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, amsgrad=True)
model.compile(loss='sparse_categorical_crossentropy', optimizer=adamOP, metrics=['accuracy'])
hist = model.fit(x_train,y_train, epochs=25, validation_data=(x_val,y_val), batch_size=529)

mlp = MLP()
mlp.addLayer(Layer(x_train.shape[1], 500, 'relu', np.sqrt(2/500)))
mlp.addLayer(Layer(500, 300, 'relu', np.sqrt(2/300)))
mlp.addLayer(Layer(300, 150, 'relu', np.sqrt(2/150)))
mlp.addLayer(Layer(150, 65, 'relu', np.sqrt(2/65)))
mlp.addLayer(Layer(65, np.max(labels)+1, 'softmax',\
                   np.sqrt(2/(np.max(labels)+1))))

ceErrs, trAccs, valAccs, valKAccs = mlp.train(x_train, y_train, x_val, y_val, 25, 529,\
                                               loss='ce', lrnRate=0.001)

print(hist.history.keys())
plt.plot(ceErrs)
plt.plot(hist.history['val_loss'])
plt.title('Validation Cross-Entropy Losses w/AMSGrad')
plt.xlabel('Epoch')
plt.ylabel('CE Loss')
plt.legend(['Own Model', 'Keras Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(valKAccs)
plt.title('Accuracies Own Model w/AMSGrad')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Validation(Top3)-Own Model'])
plt.show()

plt.plot(trAccs)
plt.plot(valAccs)
plt.plot(np.asarray(hist.history['accuracy'])*100)
plt.plot(np.asarray(hist.history['val_accuracy'])*100)
plt.title('Accuracies Own Model vs. Keras w/AMSGrad')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.legend(['Training-Own Model', 'Validation-Own Model', 'Training-Keras', 'Validation-
Keras'])
plt.show()

np.save('valAcc_AMSGrad_pca.npy', valAccs)
testPred = mlp.prediction(x_test.T)
testAcc = np.sum(testPred.reshape((testPred.shape[0],1)) == y_test)/testPred.shape[0]*100
print(testAcc)
```

Appendix 2 – Random Forest

```python
import matplotlib.pyplot as plt
import numpy as np
import h5py
import pandas as pd
import math

features_cont = pd.read_pickle('dataRF_cont.pkl')
features_disc = pd.read_pickle('dataRF_disc.pkl')
features = pd.read_pickle('dataRF.pkl')
labels = np.load('labelsRF.npy', allow_pickle=True)
print(features.shape)
features.head()
print(labels)
np.random.seed()

class Random_Forest():
    def __init__(self, x, y, n_trees, n_features, sample_size, max_depth=10, min_leaf=5):
        self.x = x
        self.y = y
        self.n_features = n_features
        self.sample_size = sample_size
        self.max_depth = max_depth
        self.min_leaf = min_leaf
        self.trees = [self.plant_tree(i) for i in range(n_trees)]

    def plant_tree(self, tree_num):
        #print('Tree Number ' + str(tree_num + 1) + ' is being created.')
        indices = np.random.permutation(len(self.y))[:self.sample_size]
        f_indices = np.random.permutation(self.x.shape[1])[:self.n_features]
        return Decision_Tree(self.x.iloc[indices], self.y[indices], self.n_features, f_indices,
                            indices=np.array(range(self.sample_size)),
                            max_depth = self.max_depth, min_leaf=self.min_leaf)

    def predict(self, x):
        predictor = np.array([t.predict(x) for t in self.trees])
        return predictor


class Decision_Tree():
    def __init__(self, x, y, n_features, f_indices,indices, max_depth=10, min_leaf=5):
        self.x = x
        self.y = y
        self.n_features = n_features
        self.f_indices = f_indices
        self.indices = indices
        self.max_depth = max_depth
        self.min_leaf = min_leaf

        self.n_tot, self.f_tot = len(indices), x.shape[1]

        counts = np.bincount(y[indices])
        self.value = np.argmax(counts)
        self.split_index = None
        self.score = 0
        self.do_split()

    def do_split(self):
        for i in self.f_indices:
            self.find_best_split(i)
        if self.is_leaf:
            return
        x = self.split_point

        left = np.nonzero(x<=self.split)[0]
        right = np.nonzero(x>self.split)[0]
        lf_indices = np.random.permutation(self.x.shape[1])[:self.n_features]
        rf_indices = np.random.permutation(self.x.shape[1])[:self.n_features]
        self.lhs = Decision_Tree(self.x, self.y, self.n_features, lf_indices,
                                self.indices[left], max_depth=self.max_depth-1, min_leaf=self.min_leaf)
        self.rhs = Decision_Tree(self.x, self.y, self.n_features, rf_indices,
                                self.indices[right], max_depth=self.max_depth-1, min_leaf=self.min_leaf)

    def find_best_split(self, split_index):

        x = self.x.values[self.indices,split_index]
        y = self.y[self.indices]
        sort_index = np.argsort(x)
        sort_x = x[sort_index]
        sort_y = y[sort_index]

        right_node = y
        left_node = []
        left_node = np.asarray(left_node)

        for i in range(0,self.n_tot-self.min_leaf-1):
            xi = sort_x[i]
            yi = sort_y[i]
            #xi, yi = x[i], y[i]
            np.append(left_node, right_node[0:1])
            right_node = right_node[1:]
            if i < self.min_leaf or xi == sort_x[i+1]:
                continue

            IG = self.information_gain(y, right_node, left_node)
            if IG > self.score:
                self.split_index = split_index
                self.score = IG
                self.split = xi


    def information_gain(self, y, r_node, l_node):
        groups = [l_node, r_node]
        Number_all = len(y)
        IG = self.gini(y)
        for group in groups:
            IG -= self.gini(group)*len(group)/Number_all
        return IG

    def gini(self, y):
        class_ids = [i for i in range(23)]
        Number_group = len(y)
        if Number_group == 0:
            return 0

        sum_of_classes = 0.
        for class_id in class_ids:
            p = list(y).count(class_id)/Number_group
            sum_of_classes += p**2
        return 1. - sum_of_classes

    @property
    def split_point(self):
        point = self.x.values[self.indices,self.split_index]
        return point

    @property
```

```python
    def is_leaf(self):
        if self.score == 0 or self.max_depth <= 0:
            return True
        else:
            return False

    def predict(self, x):
        predictor = np.array([self.predict_single(xi) for xi in x])
        return predictor

    def predict_single(self, xi):
        if self.is_leaf:
            return self.value
        if xi[self.split_index] <= self.split:
            t = self.lhs
        else:
            t = self.rhs
        return t.predict_single(xi)

def accuracy(d, y):
    t = 0
    f = 0
    for i in range(len(y)):
        counts = np.bincount(d[:,i])
        if np.argmax(counts) == y[i]:
            t = t + 1
        else:
            f = f + 1
    return (t / (t + f)) * 100

def splitter(data,y,perctrain, percv, perctest):
    a = (len(data)*perctrain)/100
    data_train = data[:int(a)]
    label_train = y[:int(a)]

    b = (len(data)*percv)/100
    data_val = data[int(a):int(b)+int(a)]
    label_val = y[int(a):int(b)+int(a)]

    c = (len(data)*perctest)/100
    data_test = data[int(b)+int(a):int(c)+int(b)+int(a)]
    label_test = y[int(b)+int(a):int(c)+int(b)+int(a)]
    return data_train, label_train, data_val, label_val, data_test, label_test

data_np = np.asarray(features)
label_np = np.asarray(labels)
shuffle=np.random.permutation(data_np.shape[0])
data_np = data_np[shuffle]
label_np = label_np[shuffle]

train_features, train_labels, val_features, val_labels, test_features, test_labels= splitter(d
ata_np, label_np, 70, 20, 10)
print(train_features.shape)
print(train_labels.shape)
print(val_features.shape)
print(val_labels.shape)
print(test_features.shape)
print(test_labels.shape)

np.random.seed(1)
MyForest = Random_Forest(pd.DataFrame(train_features), train_labels, 40, 5, 80)
prediction_train = MyForest.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

```

```python
np.random.seed(1)
MyForest2 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 80)
prediction_train = MyForest2.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest2.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest3 = Random_Forest(pd.DataFrame(train_features), train_labels, 40, 5, 120)
prediction_train = MyForest3.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest3.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest4 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120)
prediction_train = MyForest4.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest4.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest5 = Random_Forest(pd.DataFrame(train_features), train_labels, 40, 5, 40)
prediction_train = MyForest5.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest5.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest6 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 40)
prediction_train = MyForest6.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest6.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest7 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 3, 120)
prediction_train = MyForest7.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest7.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest8 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 8, 120)
prediction_train = MyForest8.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest8.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest9 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=20
, min_leaf=2)
prediction_train = MyForest9.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest9.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest10 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=5
, min_leaf=2)
prediction_train = MyForest10.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest10.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
```

```
MyForest11 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=20, min_leaf=10)
prediction_train = MyForest11.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest11.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest12 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=5, min_leaf=10)
prediction_train = MyForest12.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest12.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest13 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=10, min_leaf=2)
prediction_train = MyForest13.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest13.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

np.random.seed(1)
MyForest14 = Random_Forest(pd.DataFrame(train_features), train_labels, 80, 5, 120, max_depth=10, min_leaf=10)
prediction_train = MyForest14.predict(np.asarray(train_features))
print('Training Accuracy: ' + str(accuracy(prediction_train, train_labels)) + '%')
prediction_val = MyForest14.predict(np.asarray(val_features))
print('Validation Accuracy: ' + str(accuracy(prediction_val, val_labels)) + '%')

final_features = np.concatenate((train_features, val_features))
final_labels = np.concatenate((train_labels, val_labels))
print(final_features.shape)
print(final_labels.shape)

import time
start = time.time()
np.random.seed(1)
MyForest_last = Random_Forest(pd.DataFrame(final_features), final_labels, 80, 5, 120)
prediction_test = MyForest_last.predict(np.asarray(test_features))
print('Testing Accuracy: ' + str(accuracy(prediction_test, test_labels)) + '%')
elapsed_time = time.time() - start
        print('Execution time: ' + str(elapsed_time))
```

Appendix 3 – kNN

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

#load data
featuresCont = np.load('dataKNNcont.npy', allow_pickle=True)
featuresDisc = np.load('dataKNNdiscrete.npy', allow_pickle=True)
labels = np.load('labelsKNN.npy', allow_pickle=True)
print(featuresCont, featuresCont.shape)
print(featuresDisc, featuresDisc.shape)
print(labels, labels.shape)

#three way split
def splitData(X_cont, X_disc, y, tr, val):
    shuffle = np.random.permutation(len(y))
    X_cont = X_cont[shuffle]
    X_disc = X_disc[shuffle]
    y = y[shuffle]

    tr_ind = int(np.floor(len(y)*tr))
    val_ind = int(np.floor(len(y)*(tr+val)))

    X_tr_cont = X_cont[0:tr_ind]
    X_tr_disc = X_disc[0:tr_ind]
    y_tr = y[0:tr_ind]

    X_val_cont = X_cont[tr_ind:val_ind]
    X_val_disc = X_disc[tr_ind:val_ind]
    y_val = y[tr_ind:val_ind]

    X_test_cont = X_cont[val_ind:]
    X_test_disc = X_disc[val_ind:]
    y_test = y[val_ind:]

    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

import sys
def progressBar(value, endvalue, bar_length=20):
        percent = float(value) / endvalue
        arrow = '-' * int(round(percent * bar_length)-1) + '>'
        spaces = ' ' * (bar_length - len(arrow))

        sys.stdout.write("\rPercent: [{0}] {1}%".format(arrow + spaces, int(round(percent * 10
0))))
        sys.stdout.flush()

def KNN(trainC, trainD, trainL, valC, valD, valL , params):
    k, alpha1, alpha2, alpha3 = params
    trCount = 0

    for i in range(valC.shape[0]):
        #print(i)
        #continuous distances
        instanceMat = np.broadcast_to(valC[i], trainC.shape)
        distVecC = np.sum( (trainC-instanceMat)**2, axis=1)

        #discrete distances
        instanceMatD = np.broadcast_to(valD[i], trainD.shape)
        distMatD = np.zeros(instanceMatD.shape)
        distMatD[trainD != instanceMatD] = 1
        distMatD[:,0] *= alpha1
        distMatD[:,1] *= alpha2
        distMatD[:,2] *= alpha3
        distVecD = np.sum(distMatD, axis=1)

        totalDist = distVecD + distVecC
        #print(totalDist.shape)
        #retreive the labels with the k smallest distance
        idx = np.argsort(totalDist)[:k]
        #print(idx, idx.shape)
        kLabels = trainL[idx]
        #print(kLabels, kLabels.shape)

        #run a majority vote for the labels
        pred = np.bincount(kLabels).argmax()
        #print(np.bincount(kLabels))
        #print(pred)

        if(pred == valL[i]):
            trCount += 1

        progressBar(i,valC.shape[0])

    valAcc = 100.0*trCount/valC.shape[0]
    return valAcc

def KNN(trainC, trainD, trainL, valC, valD, valL , params):
    k, alpha1, alpha2, alpha3 = params
    trCount = 0

    for i in range(valC.shape[0]):
        #print(i)
        #continuous distances
        instanceMat = np.broadcast_to(valC[i], trainC.shape)
        distVecC = np.sum( (trainC-instanceMat)**2, axis=1)

        #discrete distances
        instanceMatD = np.broadcast_to(valD[i], trainD.shape)
        distMatD = np.zeros(instanceMatD.shape)
        distMatD[trainD != instanceMatD] = 1
        distMatD[:,0] *= alpha1
        distMatD[:,1] *= alpha2
        distMatD[:,2] *= alpha3
        distVecD = np.sum(distMatD, axis=1)

        totalDist = distVecD + distVecC
        #print(totalDist.shape)
        #retreive the labels with the k smallest distance
        idx = np.argsort(totalDist)[:k]
        #print(idx, idx.shape)
        kLabels = trainL[idx]
        #print(kLabels, kLabels.shape)

        #run a majority vote for the labels
        pred = np.bincount(kLabels).argmax()
        #print(np.bincount(kLabels))
        #print(pred)

        if(pred == valL[i]):
            trCount += 1

        progressBar(i,valC.shape[0])

    valAcc = 100.0*trCount/valC.shape[0]
    return valAcc

X_tr_cont, X_tr_disc, y_tr, X_val_cont, X_val_disc, y_val, X_test_cont, X_test_disc, y_test = 
\
```

```python
splitData(featuresCont, featuresDisc, labels, 0.8, 0.013)

kList = [1,2,3,4,5,6,7,8,9,10]

def gridSearchV2(trainC, trainD, trainL, valC, valD, valL , kList, alpha=0.1):
    bestValAcc = 0.0
    bestParams = (-1,-1,-1,-1)
    for k in kList:
        params = (k,alpha, alpha, alpha)
        print('kNN with params', params)
        acc = KNN(trainC, trainD, trainL, valC, valD, valL, params)
        print('\nValidation Accuracy:', acc)
        if(acc > bestValAcc):
            bestValAcc = acc
            bestK = k
    return bestValAcc, bestK

kList = [1,2,3,4,5,6,7,8,9,10]
valAcc, bestK = gridSearchV2(X_tr_cont, X_tr_disc, y_tr, X_val_cont, X_val_disc, y_val, kList)


def gridSearchV3(trainC, trainD, trainL, valC, valD, valL , alphaList, K=bestK):
    bestValAcc = 0.0
    bestParams = (-1,-1,-1)
    for a1 in alphaList:
        for a2 in alphaList:
            for a3 in alphaList:
                params = (K, a1, a2, a3)
                print('kNN with params', params)
                acc = KNN(trainC, trainD, trainL, valC, valD, valL, params)
                print('\nValidation Accuracy', acc)
                if(acc > bestValAcc):
                    bestValAcc = acc
                    bestParams = (K,a1,a2,a3)
    return bestValAcc, bestParams

alphaList = [0.2,0.4,0.6,0.8]
valAcc, bestK = gridSearchV3(X_tr_cont, X_tr_disc, y_tr, X_val_cont, X_val_disc, y_val, alphaL
ist)

bestParams = bestK
print('The best parameters that maximize accuracy in the given grid structure', bestParams)

def KNNv2(trainC, trainD, trainL, valC, valD, valL , params):
    k, alpha1, alpha2, alpha3 = params
    trCount = 0
    preds = []
    for i in range(valC.shape[0]):
        #print(i)
        #continuous distances
        instanceMat = np.broadcast_to(valC[i], trainC.shape)
        distVecC = np.sum( (trainC-instanceMat)**2, axis=1)

        #discrete distances
        instanceMatD = np.broadcast_to(valD[i], trainD.shape)
        distMatD = np.zeros(instanceMatD.shape)
        distMatD[trainD != instanceMatD] = 1
        distMatD[:,0] *= alpha1
        distMatD[:,1] *= alpha2
        distMatD[:,2] *= alpha3
        distVecD = np.sum(distMatD, axis=1)

        totalDist = distVecD + distVecC
        #retreive the labels with the k smallest distance
        idx = np.argsort(totalDist)[:k]
        kLabels = trainL[idx]

        #run a majority vote for the labels
        pred = np.bincount(kLabels).argmax()
        preds.append(pred)

        if(pred == valL[i]):
            trCount += 1

        progressBar(i,valC.shape[0])

    valAcc = 100.0*trCount/valC.shape[0]
    print('\Test Accuracy', valAcc)
    return preds

predVector = KNNv2(X_tr_cont, X_tr_disc, y_tr, X_test_cont,X_test_disc,y_test, bestParams)

# https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
from sklearn.metrics import confusion_matrix
from sklearn.utils.multiclass import unique_labels

def plot_confusion_matrix(y_true, y_pred, classes,
                          normalize=False,
                          title=None,
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if not title:
        if normalize:
            title = 'Normalized confusion matrix'
        else:
            title = 'Confusion matrix, without normalization'

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    # Only use the labels that appear in the data
    classes = classes[unique_labels(y_true, y_pred)]
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    fig, ax = plt.subplots(figsize=(20,20))
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    # We want to show all ticks...
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           # ... and label them with the respective list entries
           xticklabels=classes, yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()
    return ax

# Plot normalized confusion matrix
class_names = ['Dance', 'Comedy', 'Anime', 'Rap', 'Pop', 'Ska', 'Alternative', 'Country', 'Electronic', 'Reggaeton',\
               'Soul', 'Hip-Hop', 'Opera', 'Soundtrack', 'R&B', 'Indie', 'Folk', 'Reggae', 'Classical', 'Blues', 'World',\
               'Rock', 'Jazz']
plot_confusion_matrix(np.asarray(y_test), np.asarray(predVector), classes=np.asarray(class_names), normalize=True,
                      title='Confusion Matrix for kNN on Test Data')
plt.show()
```

Appendix 4 – Data Analysis

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

data = pd.read_csv('SpotifyFeatures.csv')
data.head()

classes = list(set(data['genre']))
print(classes)

data = pd.read_csv('spotifyData_v2.csv')
classes = list(set(data['genre']))
print(classes)

ax = plt.figure()
labels = data['genre'].values
ax = plt.hist(labels, bins=len(classes))
plt.xticks(rotation='vertical')
plt.show()

data = pd.read_csv('spotifyData_v3.csv')

classes = list(set(data['genre']))
print(classes)

ax = plt.figure()
labels = data['genre'].values
ax = plt.hist(labels, bins=len(classes))
plt.xticks(rotation='vertical')
plt.show()

from collections import Counter
labels = data['genre'].values
print(len(labels))
Counter(labels)

downsampleSize = 8280
'''''dataDS = pd.DataFrame(columns=['genre','artist_name','track_name','track_id','popularity'
,'acousticness','danceability',\
                               'duration_ms','energy','instrumentalness','key','liveness','lou
dness','mode','speechiness'\
                               'tempo','time_signature','valence'])'''
for i in range(len(classes)):
    part = data.loc[data['genre'] == classes[i]]
    #part = part.reset_index(drop=True)
    indexToRemove = np.random.permutation(len(part))[0:len(part)-downsampleSize]
    partDS = part.iloc[indexToRemove]
    data.drop(partDS.index, inplace=True)
data.reset_index(drop=True)

ax = plt.figure()
labels = data['genre'].values
ax = plt.hist(labels, bins=len(classes))
plt.xticks(rotation='vertical')
plt.show()
data.head()

dataKNN = data.copy()
dataKNN.drop(['track_id','track_name','time_signature'], axis=1, inplace=True)
dataKNN.head()

dataMLP = data.copy()
```

```
dataMLP.drop(['track_id','track_name','artist_name','time_signature','key','mode'], axis=1, in
place=True)
dataMLP.head()

dataRF = data.copy()
dataRF.drop(['track_id','track_name'], axis=1, inplace=True)
dataRF.head()

col_num = 3
row_num = 4
for i in range(numFeatures):
    cnt = 0
    print(dataMLP.columns[i+1])
    for j in range(numFeatures-1,-1,-1):
        cnt += 1
        plt.subplot(row_num,col_num, cnt)
        plt.scatter(dataCont[:,i], dataCont[:,j], s=1, c=labels, cmap='jet')
        plt.ylabel(dataMLP.columns[j+1])
        plt.gca().axes.get_xaxis().set_visible(False)
    plt.show()

dataCont = np.asarray(dataMLP)

#create a dictionary for the integer labels
dictionary = {}
dictrev = {}
for i in range(len(classes)):
    dictionary[classes[i]] = i
    dictrev[i] = classes[i]
print(dictionary)
print(dictrev)

labels = [dictionary[smp] for smp in dataCont[:,0]]
dataCont = dataCont[:,1:]

numFeatures = dataCont.shape[1]

dataMLP = np.asarray(dataMLP)[:,1:].astype(float)
dataMLP[:,0] = dataMLP[:,0]/100.0
dataMLP[:,3] = (dataMLP[:,3]-15e3)/(5.6e6-15e3)
dataMLP[:,7] = (dataMLP[:,7]+60.0)/60.0
dataMLP[:,9] = (dataMLP[:,9]-30.0)/250.0

np.save('dataMLP.npy', dataMLP)
np.save('labelsMLP.npy', np.asarray(labels))

np.load('dataMLP.npy', allow_pickle=True)
np.load('labelsMLP.npy', allow_pickle=True)

np.load('labelsMLP.npy', allow_pickle=True)

dataKNN.head()

dataKNNdisc = dataKNN[['artist_name', 'key', 'mode']]
dataKNNdisc.head()
np.save('dataKNNdiscrete.npy', np.asarray(dataKNNdisc).astype(str))
np.save('labelsKNN.npy', labels)
np.save('dataKNNcont.npy', dataMLP)

def vector1H(x, maxInd):
    out = np.zeros(maxInd)
    out[x] = 1
    return out

def mat1H2(y, maxInd):
    out = np.zeros((len(y), maxInd))
    for i in range(len(y)):
        out[i,:] = vector1H(y[i], maxInd)
    return out

def dictionary_maker(data, feature):
    classes = list(set(data[feature]))
    dictionary = {}
    for i in range(len(classes)):
        dictionary[classes[i]] = i
    return(dictionary)

#dataCont[:,0] = [dictionary[smp] for smp in dataCont[:,0]]

dataCont = np.asarray(data)

dictionary_key = dictionary_maker(data, 'key')
dictionary_ts = dictionary_maker(data, 'time_signature')
dictionary_mode = dictionary_maker(data, 'mode')

index_key = data.columns.get_loc('key')
index_ts = data.columns.get_loc('time_signature')
index_mode = data.columns.get_loc('mode')

key_dict = [dictionary_key[smp] for smp in dataCont[:,index_key]]
ts_dict = [dictionary_ts[smp] for smp in dataCont[:,index_ts]]
mode_dict = [dictionary_mode[smp] for smp in dataCont[:,index_mode]]

one_hot_key = mat1H2(key_dict, len(dictionary_key))
one_hot_ts = mat1H2(ts_dict, len(dictionary_ts))
one_hot_mode = mat1H2(mode_dict, len(dictionary_mode))

dataRF = data.copy()
dataRF.drop(['track_id','track_name', 'genre', 'key', 'time_signature', 'mode', 'artist_name']
, axis=1, inplace=True)
#dataRF.head()
dataRF = dataRF.to_numpy()
dataRF = pd.DataFrame(dataRF)
one_hot_key = pd.DataFrame(one_hot_key)
one_hot_ts = pd.DataFrame(one_hot_ts)
one_hot_mode = pd.DataFrame(one_hot_mode)
dataRF = pd.concat([dataRF, one_hot_key, one_hot_ts, one_hot_mode], axis=1, ignore_index=True)

dataRF.head()

dataCont = np.asarray(dataMLP)

#create a dictionary for the integer labels
dictionary = {}
for i in range(len(classes)):
    dictionary[classes[i]] = i
print(dictionary)

dataRF.to_pickle('dataRF.pkl')
np.save('labelsRF.npy', labels)
```