CONVERSATIONAL AI: SPEECH PROCESSING AND SYNTHESIS
PROJECT REPORT

(UCS749)

HarmonyAI: Unraveling Music Genres with GNNs

Submitted by:

Ritwick Roy	**102016018**
Snehleen Cheema	**102016037**

B.E. Fourth Year - CSE

Submitted To:
Dr. Anurag Tiwari

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

Computer Science and Engineering Department
Thapar Institute of Engineering and Technology
Patiala - 147001

Abstract:

This project delved into music genre classification using the GTZAN dataset, exploring various cutting-edge Graph Neural Network (GNN) architectures. The primary aim was to discern the most effective GNN framework for accurately categorizing musical genres. Diverse GNN models, such as GCNs, GATs, and GINCNs, were assessed, considering their adaptability, performance metrics, and limitations. The investigation revealed that moderate-depth GCNs (with 3 or 4 layers) outperformed other architectures, demonstrating superior stability and accuracy compared to deeper networks like GATs and multi-layer GCNs, achieving accuracy of 99.875% with 4 genres and _ with 5. Notably, the dataset's characteristics hinted at simpler models like GCNs being more suitable at capturing essential relationships, suggesting that complex attention mechanisms might not be important for this dataset. The project illuminated the potency of Graph Convolutional Networks in music genre classification. Despite the challenges, the study made noticeable the significance of tailored architectures and feature representations for optimal GNN performance on music datasets like GTZAN.

Table of Contents:

1. Introduction 3
2. Literature Review 5
3. Methodology 8
   3.1 dataset ……………………………………………………………………… 9
   3.2 Data Preprocessing and Feature extraction ……………………………. 9
   3.3 Graph Representation ……………………………………………….... 12
   3.4 Proposed models ………………………………………………….... 14
   3.5 Model evaluation …………………………………………………….. 14
   3.6 Visualisation of results ……………………………………………….. 15
4. Implementation ……………………………………………………….. 16
   4.1 dataset preparation ………………………………………………… 16
   4.2 Construction of graph structure ………………………………….… 17
   4.3 building models …………………………………………………… 19
   4.4 Training and testing models ………………………………………….. 26
5. Results …………………………………………………………………. 30
6. Discuss ………………………………………………………………… 34
7. Conclusion …………………………………………………………….. 36
8. References …………………………………………………………….... 37

1. Introduction:

The GTZAN dataset serves as a cornerstone in music genre classification, encapsulating a rich repository of audio clips covering ten distinct genres. These include jazz, blues, rock, pop, classical, reggae, hip-hop, country, metal, and disco. Each snippet encapsulates unique acoustic characteristics representative of its genre, making GTZAN an invaluable resource for machine learning endeavors. MIR (Music Information Retrieval) techniques offer diverse feature representations for audio analysis. From MFCCs (Mel Frequency Cepstral Coefficients) to spectral features like Spectral Centroid and Rolloff, these representations encapsulate advanced aspects of musical data.

Initially, the benchmarking on GTZAN relied on classical machine learning algorithms such as SVMs, Decision Trees, and k-NN, where accuracy hovered around 60-70%. As deep learning gained traction, CNNs and RNNs raised the bar, surpassing 90% accuracy. However, these models struggled to capture nuanced relationships within music due to their limited ability to handle sequential data or hierarchies in the audio features or simply not being able to capture relationships among genres.

Amidst these advancements, Graph Neural Networks (GNNs) have emerged as a powerful method for modeling relational data that at times better represents real world objects or ideas, including music. GNNs encode information within a graph structure, capturing the inherent relationships between audio elements. This property makes GNNs an interesting choice for music classification tasks. Leveraging GNNs like Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and GraphSAGE, researchers have explored their applicability in discerning intricate patterns within music data. Graph-based models like GCNs and GATs showcased their potential, often achieving accuracies surpassing 90% while demonstrating better capture of complex relationships and structures in the data.

Recent research has showcased GNNs' potential in capturing complex relationships in music, leading to improved genre classification. Our exploration involved employing these cutting-edge models on the GTZAN dataset, evaluating their efficacy in classifying genre-specific patterns within audio samples. This involved analyzing diverse architectures, tuning hyperparameters, and assessing performance metrics like accuracy, precision, recall, and F1-score. The objective was to unravel the most effective GNN framework designed for music genre classification, focusing on their adaptability, limitations, and effectiveness in modeling music data.

While GNNs exhibited promising potential, there still are multiple challenges toface. The dataset's heterogeneity, varying audio lengths, and noise posed complexities etc. Some models exhibit tendencies toward overfitting or struggling with convergence due to their inherent complexity.

The convergence of MIR techniques, graph-based representations, and GNNs in music genre classification seems promising for future research. Enhanced feature engineering, attention mechanisms tailored for music data, and innovative model architectures stand as potential directions for further exploration and beating of the benchmarks by realising which graph based model works the best on the given dataset. This project served as an exploration into the intersection of music, graph-based learning, and machine intelligence, paving the way for deeper investigations into music analysis and understanding through cutting-edge AI techniques.

2. Literature Review:

The GTZAN dataset was developed by George Tzanetakis, Perry Cook, and Dan Ellis [1] comprises 1,000 audio tracks across 10 genres (blues, classical, country, disco, hip-hop, jazz, metal, pop, reggae, and rock), who set the benchmark on the dataset by obtaining the highest accuracy of 61.0% using Gaussian Mixture Model (GMM).

Tao Li et al. [2] used SVM and LDA for content based music genre classification on the GTZAN dataset augmented with a custom dataset constructed by the author thereby achieving the best accuracy of 78.5%.

Li, T., Chan, A., Chun, A [3] used CNN (Convolutional Neural Networks) for extracting features from audio files instead of concentrating entirely on timbral features, achieving the best accuracy of 84%. Since the success rate drops when the number of classifications is above 4, results were evaluated on only 4 out of the 10 classes.

Chathuranga et al. [4] used SVM as a base learner in Adaboost techniques, and used a late fusion method to combine individual results for genre classification, surpassing the performance of the trained fusion method. to attain an accuracy of 81% on the GTZAN Audio dataset and 78% on ISMIR2004 Genre dataset.

Parth Rohilla et al. [5] proposed a multiphase ensemble model based on machine learning algorithms obtaining a final accuracy of 82.25% and topsis score of 0.97 on 5 classes.

Prasenjeet Fulzele et al. [6] proposed a hybrid model for classification using LSTM and SVM models and obtained an accuracy of 89% and thus exceeding the results of standalone LSTM and SVM models..

Michael Haggblade et al. [7] investigated various machine learning algorithms including k-nearest neighbor (k-NN), k-means, multi-class SVM, and neural networks using the MFCCs as features, achieving the best accuracy of 96% on 4 classes using a feedforward neural network with 10 layers making it computationally expensive to implement.

Chun Pui Tang [8] used Long Short Term Memory(LSTM) and obtained the highest accuracy of 57.45%.

Table 1 - Gatherings from several research papers on graphs

Paper Title	**Graph operator**	**Novelty/Strenghts**	**Limitations**
Semi-Supervised Classification with Graph Convolutional Networks [9]	GCNConv	The paper introduced Graph Convolutional Networks (GCNs) for semi-supervised learning on graph-structured data, utilising relationships to improve classification with limited labeled samples.	Sensitivity of GCNs to irregular or noisy graphs might affect the model's performance, which makes it hard to generalize to diverse graph topologies.
Inductive Representation Learning on Large Graphs [10]	SAGEConv	Leveraged node feature information to efficiently generate node embeddings for previously unseen data thus algorithm generalizes to completely unseen graphs	Difficulty in handling dynamic or evolving graph structures efficiently, affecting the model's adaptability to changes in the graph over time.
Graph Attention Networks [11]	GATConv	Ability to selectively attend to different nodes information of a graph thus enabling adaptive learning of node representations using the importance of neighboring nodes.	Scalability issues as the graph grows, the attention mechanism's computational demands increase significantly, making it difficult to process and learn from such extensive graph structures.

Attention-based Graph Neural Network for Semi-supervised Learning [12]	AGNNConv	Significantly reduces the number of parameters, which is critical for semi-supervised learning where number of labeled examples are small.	Sensitivity to hyperparameters and the need for fine-tuning impacts its performance and generalization across diverse graph structures.
How Powerful are Graph Neural Networks? [13]	GINConv	Developed a simple architecture that is provably the most expressive among the class of GNNs and is as powerful as the Weisfeiler-Lehman graph isomorphism test.	Despite its comprehensive investigation into GNNs' expressiveness regarding graph isomorphism and automorphism, further research may be needed to extend its scope and address broader practical applications in real-world scenarios.

The graph operators are the operators each paper used for message passing or convolutions etc, in the development of their models and are implemented and abstracted in *pytorch-geometric* a python library, using which multiple models will be developed and tested on the GTZAN dataset.

3. Methodology

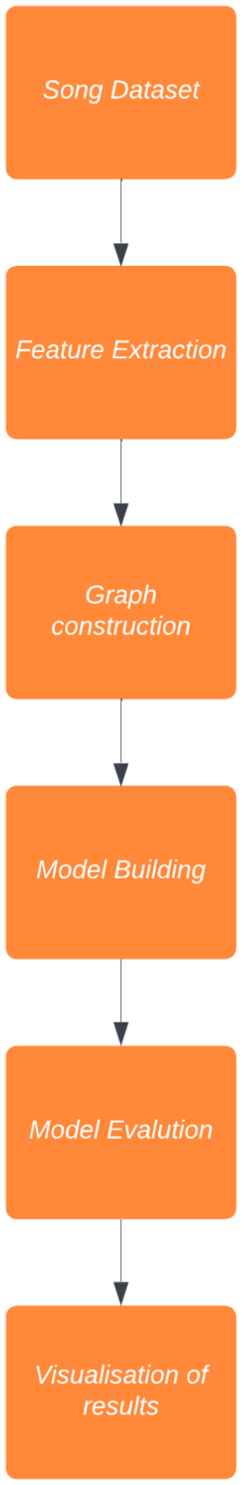


Figure 1: Flowchart of the methodology

3.1. Dataset:

The dataset used for the present work is GTZAN Audio dataset consisting of 10 musical genres namely Metal, Classical, Blues, Country, Disco, Hip-Hop, Jazz, Pop, Reggae and Rock each having 100 audio clips each. Each audio sample is approximately 30 seconds long and sampled at a 44.1 kHz sampling rate in WAV format. Due to its availability and well-defined genre labels, the GTZAN dataset has been extensively utilized in research and experimentation for developing and evaluating algorithms in music genre classification, audio signal processing, feature extraction, and machine learning.While widely used, the GTZAN dataset has also faced criticisms regarding its sample quality, labeling accuracy, and potential limitations in representing the diversity within each genre. Despite its limitations, the GTZAN dataset remains a fundamental resource for developing and benchmarking music genre classification algorithms.

3.2 Data Preprocessing and Feature extraction:

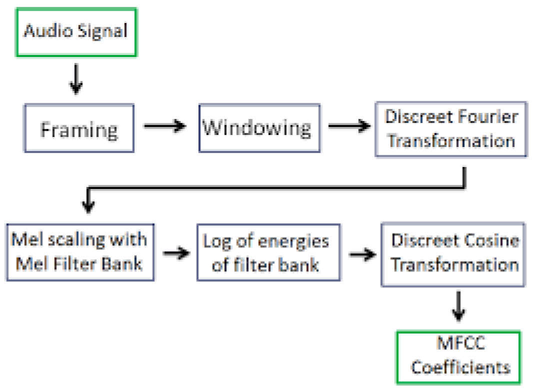
Due to the dataset being balanced and clean, no preprocessing was required. For feature extraction from the audio files, an open source python package called Librosa was used. Since the number of samples (1000) was low, 10 clips of 3 seconds were sampled from each 30 second clip which increases the number of data points tenfold but also makes it computationally expensive to store and compute which is the main reason that time is also considered for comparison of models.The following features were calculated to create the feature vector for each 3 second clip-

Table 2 - Features breakdown

Feature Group	Number of features
MFCCs	40
Chroma	12
Mel Spectrogram	128
Tonnetz	6
Spectral contrast	7

Spectral centroid	2
Spectral bandwidth	2
Spectral flatness	2
Rolloff	2
RMS	2
Tempo	2
Total	**203**

I. MFCCs: Mel-frequency cepstral coefficients (MFCCs) capture spectral and timbral qualities similar to human auditory perception and thus serve as effective descriptors in audio signal processing, especially music analysis. A time signal which correlates over time on taking the DCT, gives the spectrum of the signal. The MFCC are derived from the log-mel spectrum through mel-frequency mapping, logarithm, and finally DCT. The MFCC is an abstract domain, which is not easy to interpret visually. However, since it is designed to correspond to resemble perception in both magnitude and frequency axis, and to be roughly uncorrelated, it is efficient for computation.Following steps are followed for extracting MFCCs.


Audio Signal
Framing
Windowing
Discreet Fourier Transformation
Mel scaling with Mel Filter Bank
Log of energies of filter bank
Discreet Cosine Transformation
MFCC Coefficients

- Take the Fourier transform of (a windowed excerpt of) a signal.
- Map the powers of the spectrum obtained above onto the mel scale, using triangular overlapping windows or cosine overlapping windows.

- Take the logs of the powers at each of the mel frequencies.
- Take the discrete cosine transform of the list of mel log powers, as if it were a signal.
- The MFCCs are the amplitudes of the resulting spectrum.

librosa.feature.mfcc() method from library librosa in python is used to extract the MFCCs features.

II. Chroma: It closely relates to twelve different pitch classes. Chroma-based features, which are also referred to as "pitch class profiles", are a powerful tool for analyzing music whose pitches can be meaningfully categorized often into twelve categories, thus the 12 features, and whose tuning approximates to the equal-tempered scale. One main property of chroma features is that they capture harmonic and melodic characteristics of music, while being robust to changes in timbre and instrumentation. [14]

librosa.feature.chroma_stft() method is used for the same.

III. Mel: A Mel spectrogram visually represents signal energy distributed across specific Mel-frequency bands, highlighting frequencies aligned with human auditory perception for an enhanced viewing of the audio signal. To convert from Hertz (Hz) to Mel (M), the following formula is used.

$$M(f)=2595 \cdot log\ 10\ (1+f/700)$$

librosa.feature.melspectrogram() method is used for the same.

IV. Tonnetz: Short for Tonal Network, is a conceptual representation used in music theory and analysis. Relationships between musical pitches (based on their harmonic properties) are visualised creating a hexagonal lattice that reflects tonal harmony, thus giving the 6 features. This representation helps illustrate harmonic connections and transformations between different musical notes. *librosa.feature.tonnetz()* method is used for the same.

V. Spectral features: They refer to distinctive characteristics extracted from the frequency content of an audio signal. These features contain information about the distribution of energy, across different frequency bands and are thus commonly used in audio signal processing and analysis. The set of features used in this case encapsulate:

- **Contrast:** Measures the difference in magnitude between peaks and valleys across frequency bands, highlighting tonal variations.
- **Centroid:** The spectral centroid identifies the spectrum's center of mass, closely linked to the perceived brightness of a sound.
- **Bandwidth:** Measures the width of the spectrum, indicating how broad or narrow the range of frequencies present in the signal is.
- **Rolloff:** Indicates the frequency below which a certain percentage (e.g., 85%) of the total spectral energy is contained.
- **Flatness:** Measures how uniform or peaked the energy is across frequencies with values closer to 1 indicating a more uniform distribution (like white noise) and values nearer to 0 suggesting a more peaked spectrum (like a pure tone).

spectral_contrast(), spectral_centroid(), spectral_bandwidth(), spectral_rolloff() and *spectral_flatness()* methods from librosa.feature is used to extract corresponding features.

VI. RMS: It quantifies the signal's energy by calculating the square root of the mean square amplitude, thus gaining insight into the signal's overall loudness or power level. *librosa.feature.rms()* method is used to calculate the rms for each frame sampled.

VII. Tempo: Denotes the speed or pace at which a piece of music is played, measured in beats per minute (BPM) in order to determine rhythm categorizing it as slow, moderate, or fast-paced, thus evaluating the mood of the song. *librosa.beat.tempo()* method is used to estimate the global tempo of the song.

3.3 Graph Representation:

A graph G is represented as an ordered pair G=(V,E) where V denotes a set of vertices (nodes) and E represents a set of edges connecting these vertices. An edge between 2 vertices represents some form of similarity or connection between the two.

We propose to utilize graph networks on the GTZAN dataset for classification purposes as it holds several advantages:

I. **Intrinsic wtructural Relationships:** The GTZAN dataset, although an audio dataset primarily, can also be modelled as a graph where each node represents a song or an audio sample, and edges indicate relationships or similarities between them. Graph networks are good at handling such structured data and so it can effectively capture relationships between music samples.

II. **Diverse feature set representation:** Graph networks succeed at incorporating various types of features and relationships within a dataset. Features such as Mel-frequency cepstral coefficients (MFCCs), chroma features, and spectrogram representations etc. can be effectively embedded into a graph structure.

III. **Robustness to noise and irregularities:** Graph networks are robust to noise and missing data, which can often be encountered in audio datasets. They can learn from both, the audio content itself and its relations or similarities with other samples, making them robust in handling diverse and incomplete information.

IV. **Contextual Information Utilization:** Since genre classification often relies on the context or connections between songs within a genre or across genres, graph networks can exploit this contextual information by analyzing relationships between songs or genres, potentially leading to more accurate results.

Thus convert audio data into a graph representation where nodes represent audio samples and edges signify relationships (e.g., similarity, adjacency). The graph is by nature undirected, since one song similar to second also implies second song being similar to one.

Now, since for most of the features direct comparison of values leads to inadequate results due to the high dimensionalities of features and differences in variances, cosine similarity was selected to measure similarities between nodes or songs and draw edges in the case of high similarities. Edges were all also drawn between all songs belonging to the same genre, and all self loops were removed for easy visualisation and also because it doesn't make a difference in case of processing of graph networks because the feature set of any node is anyway used in message passing.

sklearn.metrics.pairwise.cosine_similarity() method was used for finding similarities between nodes from the library sklearn in python.

3.4. Proposed models:

- Design and implement graph convolution networks and graph attention networks using GCNConv, SAGEConv, GATConv, AGNNConv, GINConv operators using *torch_geometric* a library specializing in graph neural networks.
- Change the number of layers, iterations, attention heads wherever possible.
- Split the dataset into training, validation, and test sets.
- Train the models on the training set and validate.
- Repeat the above steps for 4 classes and 5 classes.

3.5 Model Evaluation:

- Evaluate model performance on the test dataset.
- Measure classification metrics such as accuracy, precision, recall, and F1-score.

Below are are definitions and formulas for each:

I. **Accuracy:** Measures the proportion of correct predictions out of the total predictions made.

Accuracy=Total Number of Predictions/Number of Correct Predictions

II. **Precision:** Indicates the ratio of correctly predicted positive observations to the total predicted positives.

Precision=True Positives/(False Positives+True Positives)

III. **Recall (Sensitivity):** Represents the ratio of correctly predicted positive observations to the actual positives.

Recall=True Positives /(False Negatives+True Positives)

IV. **F1 Score:** Harmonic mean of precision and recall, providing a balanced measure between the two metrics.

F1 Score=2×Precision×Recall/(Precision+Recall)

3.6 Visualisation of results

During training, losses and accuracies across epochs for both training and validation datasets are tracked. After training, a final evaluation is performed on the test set and the evaluation metrics (accuracy, precision, recall, F1-score) are stored in a results dataframe. Additionally, graphs illustrating the training and validation losses, as well as the validation and test accuracies across epochs are generated using *matplotlib* library, providing a visual representation of the model's performance during training.

4. Implementation

4.1 Dataset Preparation: Implemented *prepare_dataset()* which initialises a pandas dataframe, iterates through the genres, then corresponding to each genre load files using the *librosa.load(path_to_file, duration, offset)* method where duration and offset parameters are used to essentially split the file from 30 to 3 seconds and load the different clips by specifying the offset value.

```
for dur in range(10):
    y, sr = librosa.load(base_dir+file,duration=3,offset=3*dur)
```

Passing the (y, sr) audio time series and sampling rate of y respectively obtained from above to the librosa methods mentioned above helps extracts the features from each clip

Table 3 - Dataset's first 5 rows

S.No	feature1	feature2	feature3	...	feature202	feature203	label
0	-118.133260	1255.9818	50.151493	...	0.091761	0.001133	disco
1	-100.267510	3266.9482	44.737953	...	0.100778	0.002665	disco
2	-22.712584	1397.5577	62.249786	...	0.179238	0.002508	disco
3	-44.962524	1055.3086	60.939137	...	0.114344	0.001327	disco
4	-63.569725	1498.1869	60.786636	...	0.105870	0.001530	disco

4.2 Construction of graph structure

A single graph in PyG (pytorch-geometric) is described by an instance of *torch_geometric.data.Data* which holds the following attributes by default, all of which are optional:

- *data.x*: Node feature matrix with shape [num_nodes, num_node_features]
- *data.edge_index*: Graph connectivity in COO format with shape [2, num_edges] and type torch.long
- *data.y*: Target to train against (may have arbitrary shape), *e.g.*, node-level targets of shape [num_nodes, *] or graph-level targets of shape [1, *]
- *data.train_mask*: denotes against which nodes to train the model on.
- *data.val_mask*: denotes which nodes to use for validation.
- *data.test_mask*: denotes against which nodes to test.

[15]

data.x is simply the feature set generated in the above steps and *data.y* is the corresponding set of genres associated with each entry in *data.x*. *data.edge_index* attribute is built, following the sequence of steps:

I. Draw edges between songs belonging to same genre.

```
genre_edges = [(i, j) for i in range(len(genres)) for j in range(i + 1,
len(genres)) if genres[i] == genres[j]]
```

II. Create similarity matrix of order NxN (N=4000 in case of 4 classes and 5000 in case of 5 classes) where element at position (i,j) denotes cosine similarity between feature sets of clip i and clip j, then set the diagonals to 0 to remove self loops.

```
similarity_matrix = cosine_similarity(song_features)

np.fill_diagonal(similarity_matrix, 0)
```

III. Using the similarity matrix and similarity threshold which essentially is a hyperparameter and the value of 0.99999 was reached after trying out multiple set of values for the same, store the edges between 2 clips whose similarity exceeds the threshold (this forms the basis for using graph neural networks for classification in GTZAN dataset) in COO format. Thus add (i,j) in similarity_edges whenever

similarity_matrix[i][j]>0.99999 where *i!=j*

```
similarity_threshold = 0.99999

similarity_edge_indices = np.where((similarity_matrix >
similarity_threshold))

similarity_edges = list(zip(similarity_edge_indices[0],
similarity_edge_indices[1]))
```

IV. Using a set combine the similarity edges and genre edges to create the final set of edges for our graph structure.

```
all_edges = set(similarity_edges + genre_edges)
```

Range of values from 1 to N (=number of sampled) is broken down in 3 parts, 80% 10% 10% respectively for training, validation and testing purposes using *train_test_split* method imported from *sklearn.model_selection* for creation of masks.

```
train_indices, temp_indices = train_test_split(range(N), test_size=0.4)

val_indices, test_indices = train_test_split(temp_indices, test_size=0.5)

train_mask = torch.zeros(num, dtype=torch.bool)

val_mask = torch.zeros(num, dtype=torch.bool)

test_mask = torch.zeros(num, dtype=torch.bool)

train_mask[train_indices] = 1

val_mask[val_indices] = 1
```

```
test_mask[test_indices] = 1
```

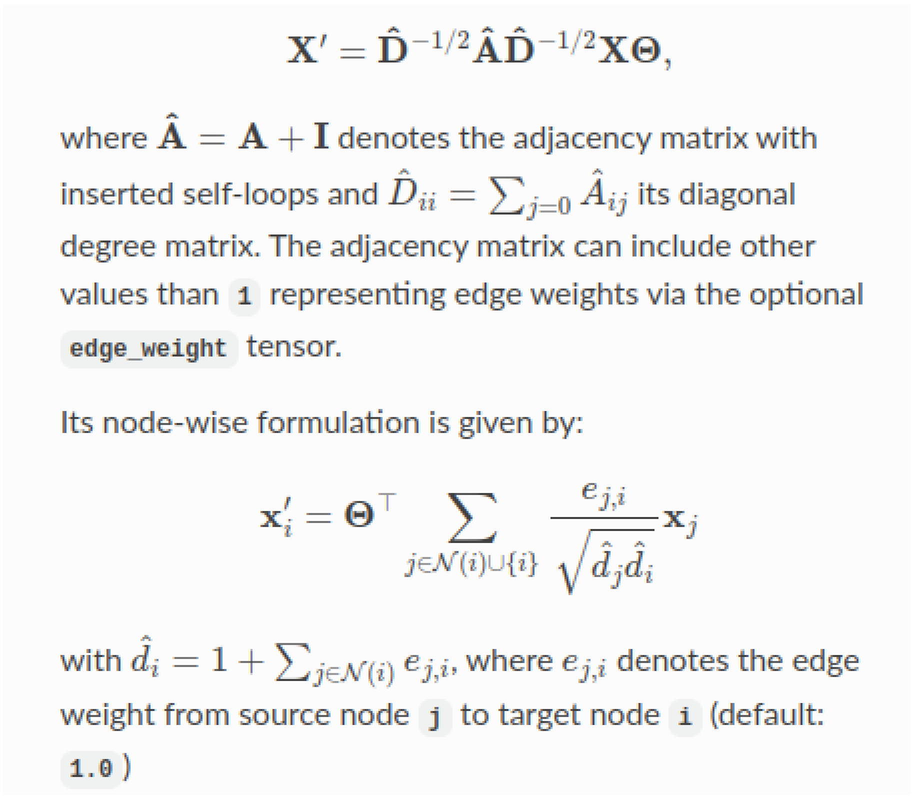
The final undirected graph is initialised using the *torch_geometric.data.Data* method.

```
edge_index = torch.tensor(list(all_edges),
dtype=torch.long).t().contiguous()
x = torch.tensor(song_features, dtype=torch.float)
data = Data(x=x,
edge_index=edge_index,y=torch.tensor(y),train_mask=train_mask,test_mask=te
st_mask,val_mask=val_mask)
```

4.3 Building models

Multiple model architectures were constructed, each experimenting with different combinations of layer counts, dropout rates, activation functions, attention heads, and operators. Throughout training, these models were subjected to different number of iterations, and notably, as model complexity increased , the training time also surged considerably. However, most portion of these complex models encountered issues in processing the large volume of data. Some models crashed due to their scalability issues, while others proved inefficient and less accurate. So after carefully considering the model complexity and removing ones that crashed for any reasons mentioned above the following set of models were retained and further processing was carried out by them. *a pytorch-geometric* library was used for building the models.

I. **Graph convolution networks GCN -** Simple GCN with 2, 3, 4 and 5 layers each with dropout = 0.5 and relu activation function. Given below is the architecture for a five layer GCN, other GCNs with fewer layers can simply be constructed by removing some layers from the given architecture. Hidden channels is also a hyperparameter which upon experimentation with several values yielded the best results when hidden channels was kept at 64. The depth allows the model to capture intricate relationships between nodes.



$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta},$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with inserted self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ its diagonal degree matrix. The adjacency matrix can include other values than `1` representing edge weights via the optional `edge_weight` tensor.

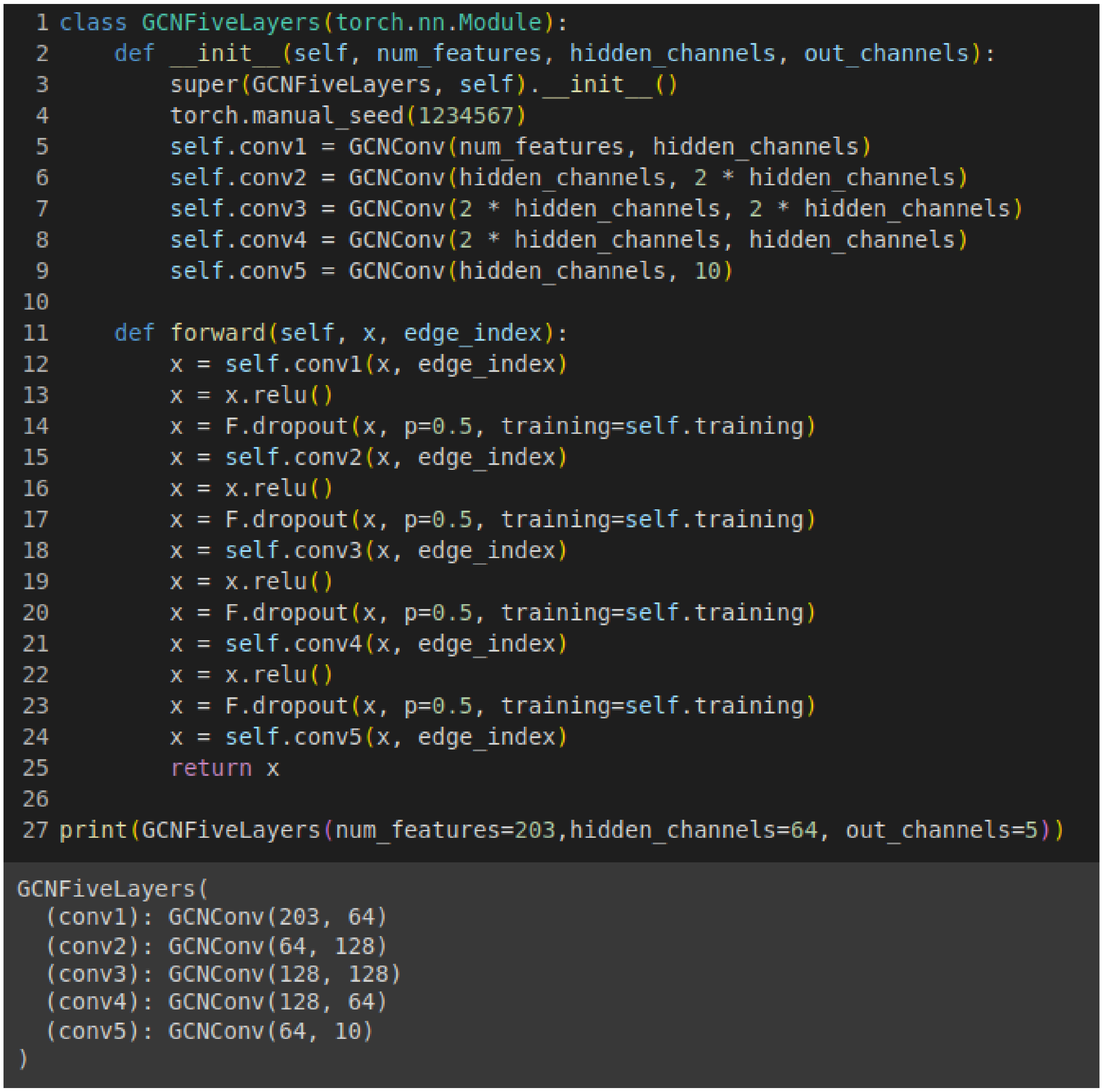
Its node-wise formulation is given by:

$$\mathbf{x}'_i = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j$$

with $\hat{d}_i = 1 + \sum_{j \in \mathcal{N}(i)} e_{j,i}$, where $e_{j,i}$ denotes the edge weight from source node `j` to target node `i` (default: `1.0`)

GCNConv operator [15]

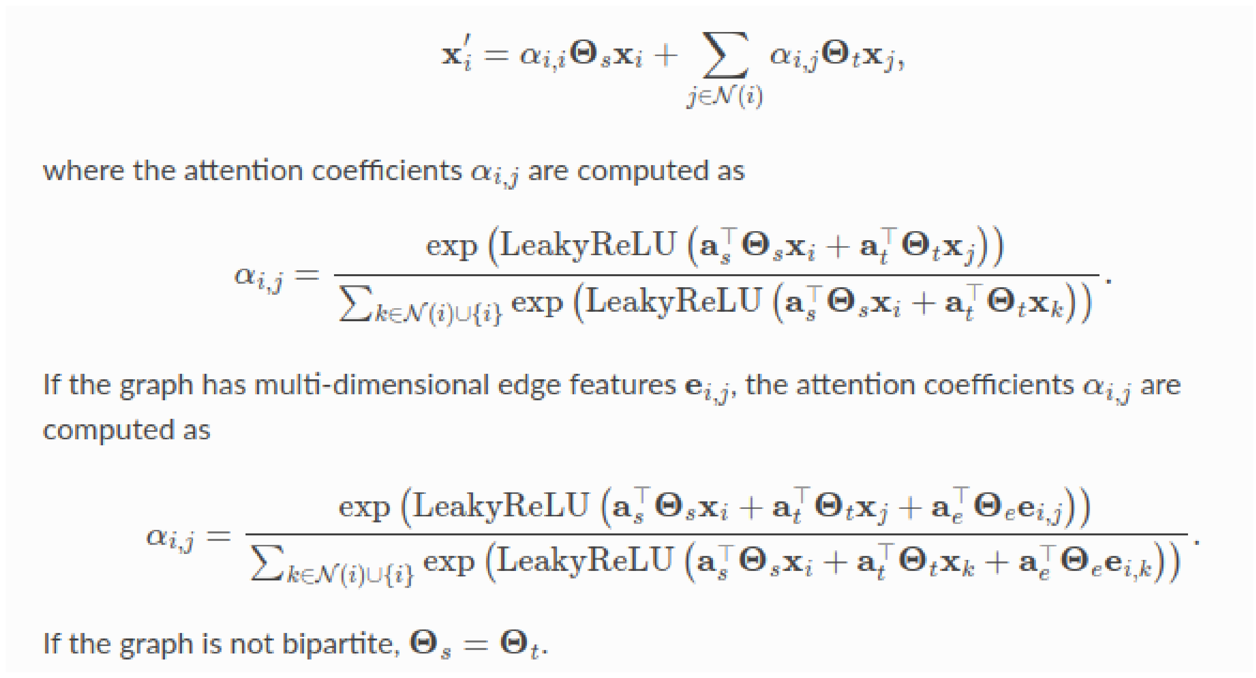
Here is the code snippet for the model architecture.



```
class GCNFiveLayers(torch.nn.Module):
    def __init__(self, num_features, hidden_channels, out_channels):
        super(GCNFiveLayers, self).__init__()
        torch.manual_seed(1234567)
        self.conv1 = GCNConv(num_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, 2 * hidden_channels)
        self.conv3 = GCNConv(2 * hidden_channels, 2 * hidden_channels)
        self.conv4 = GCNConv(2 * hidden_channels, hidden_channels)
        self.conv5 = GCNConv(hidden_channels, 10)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv2(x, edge_index)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv3(x, edge_index)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv4(x, edge_index)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv5(x, edge_index)
        return x

print(GCNFiveLayers(num_features=203,hidden_channels=64, out_channels=5))
```

```
GCNFiveLayers(
  (conv1): GCNConv(203, 64)
  (conv2): GCNConv(64, 128)
  (conv3): GCNConv(128, 128)
  (conv4): GCNConv(128, 64)
  (conv5): GCNConv(64, 10)
)
```

II. **Graph Attention network GAT -** Using the GAT class, we initialize two layers of GAT operations: self.conv1 and self.conv2, instantiated with GATConv modulIn the __init__ method where the GAT is configured with an input dimension (in_dim), hidden dimension (hidden_dim), output dimension (out_dim), and the number of attention heads (num_heads). These parameters define the architecture's structure. The forward method executes the data flow through the network. It applies the first GAT convolution (self.conv1), followed by a rectified linear unit (ReLU) activation function and dropout regularization. Subsequently, the data passes through the second GAT convolution (self.conv2). The attention mechanism allows nodes of the graph to attend to different portions of the input, which is important in capturing any interdependencies between the genres.



$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}_s\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}_t\mathbf{x}_j,$$

where the attention coefficients $\alpha_{i,j}$ are computed as

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}_s^{\top}\mathbf{\Theta}_s\mathbf{x}_i + \mathbf{a}_t^{\top}\mathbf{\Theta}_t\mathbf{x}_j\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}_s^{\top}\mathbf{\Theta}_s\mathbf{x}_i + \mathbf{a}_t^{\top}\mathbf{\Theta}_t\mathbf{x}_k\right)\right)}.$$

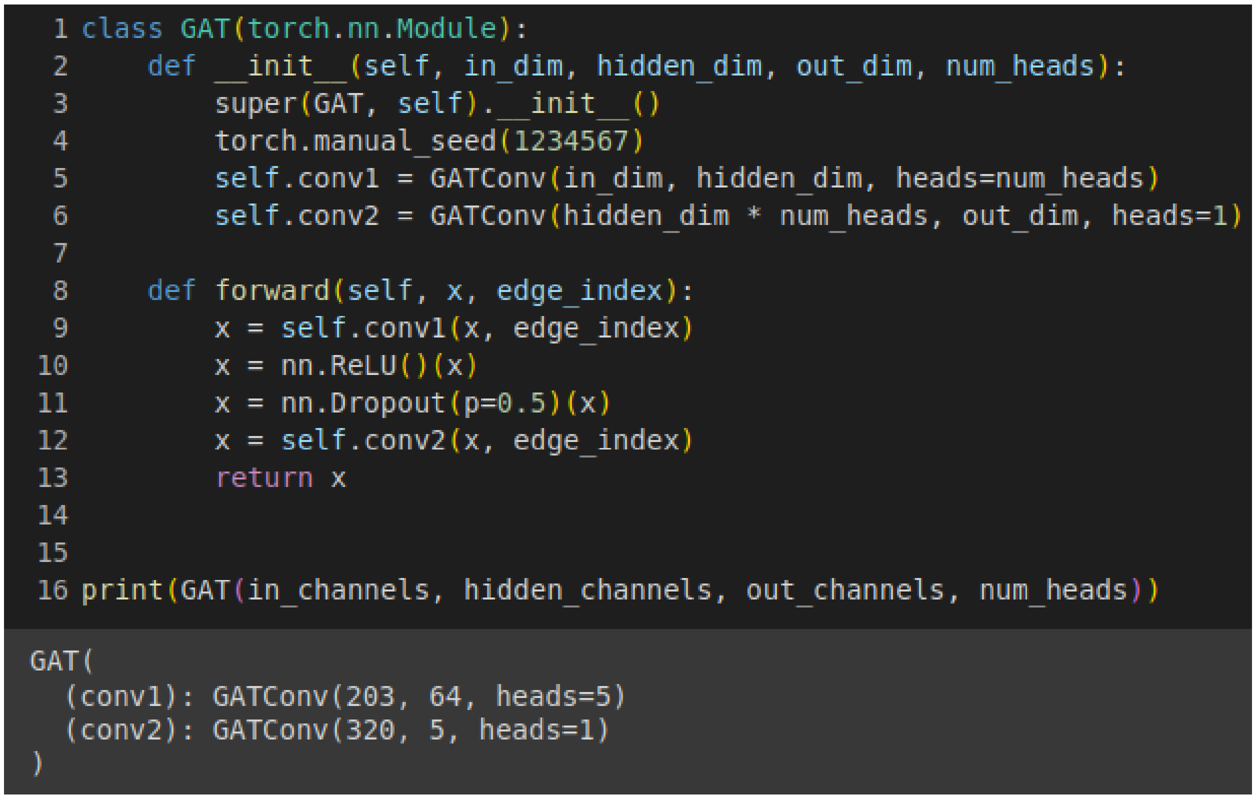
If the graph has multi-dimensional edge features $\mathbf{e}_{i,j}$, the attention coefficients $\alpha_{i,j}$ are computed as

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}_s^{\top}\mathbf{\Theta}_s\mathbf{x}_i + \mathbf{a}_t^{\top}\mathbf{\Theta}_t\mathbf{x}_j + \mathbf{a}_e^{\top}\mathbf{\Theta}_e\mathbf{e}_{i,j}\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}_s^{\top}\mathbf{\Theta}_s\mathbf{x}_i + \mathbf{a}_t^{\top}\mathbf{\Theta}_t\mathbf{x}_k + \mathbf{a}_e^{\top}\mathbf{\Theta}_e\mathbf{e}_{i,k}\right)\right)}.$$

If the graph is not bipartite, $\mathbf{\Theta}_s = \mathbf{\Theta}_t$.

GATConv operator [15]

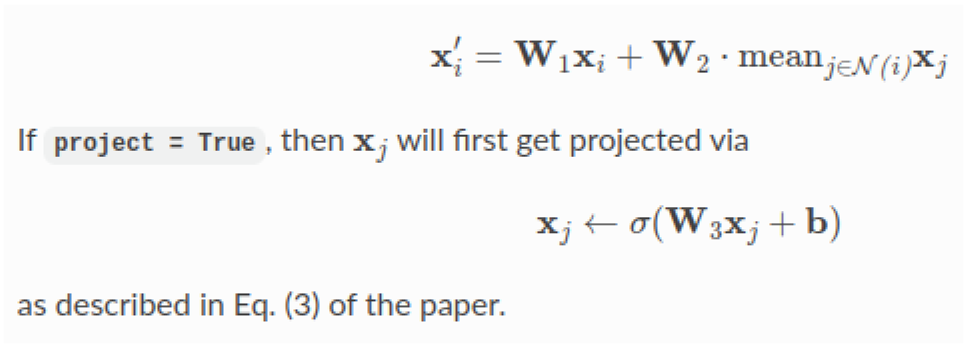
Here is the code snippet for the model architecture.



```python
class GAT(torch.nn.Module):
    def __init__(self, in_dim, hidden_dim, out_dim, num_heads):
        super(GAT, self).__init__()
        torch.manual_seed(1234567)
        self.conv1 = GATConv(in_dim, hidden_dim, heads=num_heads)
        self.conv2 = GATConv(hidden_dim * num_heads, out_dim, heads=1)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = nn.ReLU()(x)
        x = nn.Dropout(p=0.5)(x)
        x = self.conv2(x, edge_index)
        return x


print(GAT(in_channels, hidden_channels, out_channels, num_heads))
```

```
GAT(
  (conv1): GATConv(203, 64, heads=5)
  (conv2): GATConv(320, 5, heads=1)
)
```

III. **GraphSAGE network -** The GraphSAGE model, defined in GraphSAGENet, utilizes a stack of SAGEConv layers (self.conv1, self.convs, self.conv_out). The __init__ method configures the model with adjustable input, hidden, and output channel dimensions along with the number of layers. The forward method applies SAGEConv operations with ReLU activations in a layered manner. GraphSAGE networks are effective in aggregating information from node neighborhoods, which are important for capturing genre relationships in the GTZAN dataset's music graph. The model's ability to aggregate information from adjacent nodes representing different music genres helps in learning meaningful genre representations.



$$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \mathrm{mean}_{j \in \mathcal{N}(i)} \mathbf{x}_j$$

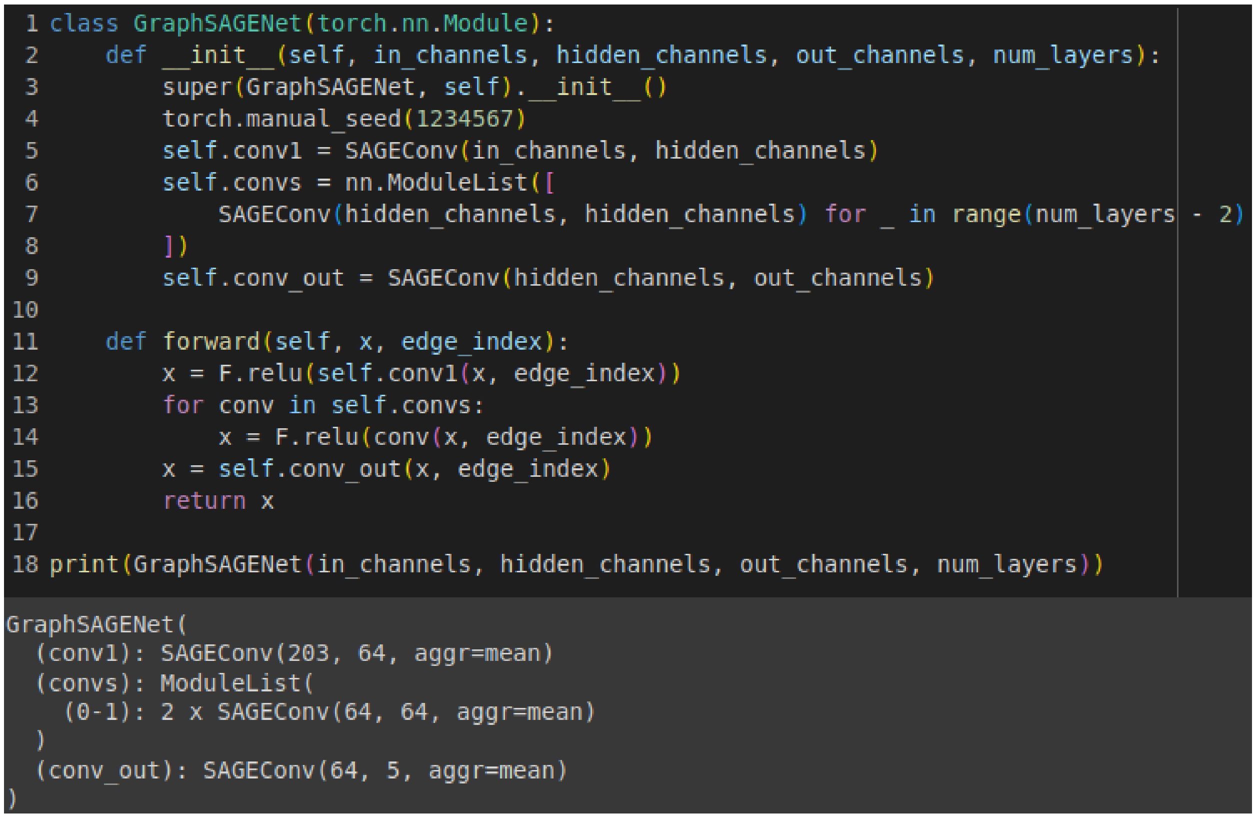
If `project = True`, then $\mathbf{x}_j$ will first get projected via

$$\mathbf{x}_j \leftarrow \sigma(\mathbf{W}_3 \mathbf{x}_j + \mathbf{b})$$

as described in Eq. (3) of the paper.

SAGEConv operator [15]

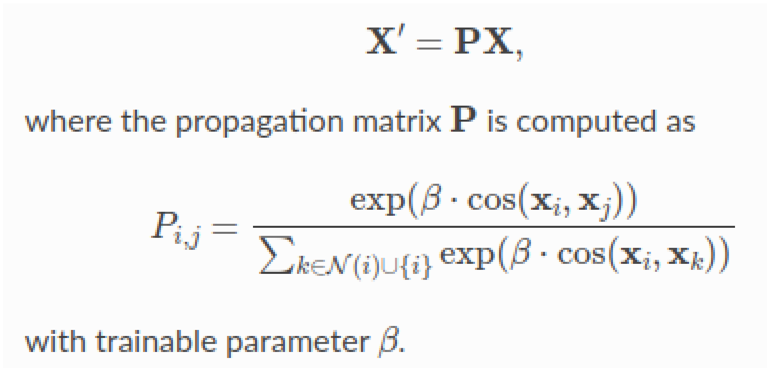
Here is the code snippet for the model architecture.



```
class GraphSAGENet(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, num_layers):
        super(GraphSAGENet, self).__init__()
        torch.manual_seed(1234567)
        self.conv1 = SAGEConv(in_channels, hidden_channels)
        self.convs = nn.ModuleList([
            SAGEConv(hidden_channels, hidden_channels) for _ in range(num_layers - 2)
        ])
        self.conv_out = SAGEConv(hidden_channels, out_channels)

    def forward(self, x, edge_index):
        x = F.relu(self.conv1(x, edge_index))
        for conv in self.convs:
            x = F.relu(conv(x, edge_index))
        x = self.conv_out(x, edge_index)
        return x

print(GraphSAGENet(in_channels, hidden_channels, out_channels, num_layers))
```

```
GraphSAGENet(
  (conv1): SAGEConv(203, 64, aggr=mean)
  (convs): ModuleList(
    (0-1): 2 x SAGEConv(64, 64, aggr=mean)
  )
  (conv_out): SAGEConv(64, 5, aggr=mean)
)
```

IV. **AGNN operator based network -** The multi-layer architecture allows for learning hierarchical music features, capturing local patterns in lower layers and more abstract features in the deeper ones. This progressive representation helps recognize specific genre-related patterns or motifs. The adaptive learning ability of AGNNs, allows the model to adapt its parameters during training. Dropout in Linear and AGNNConv layers prevents overfitting which aids in generalization, enabling the model to better understand and classify unseen music samples beyond the training data.



$$\mathbf{X}' = \mathbf{P}\mathbf{X},$$

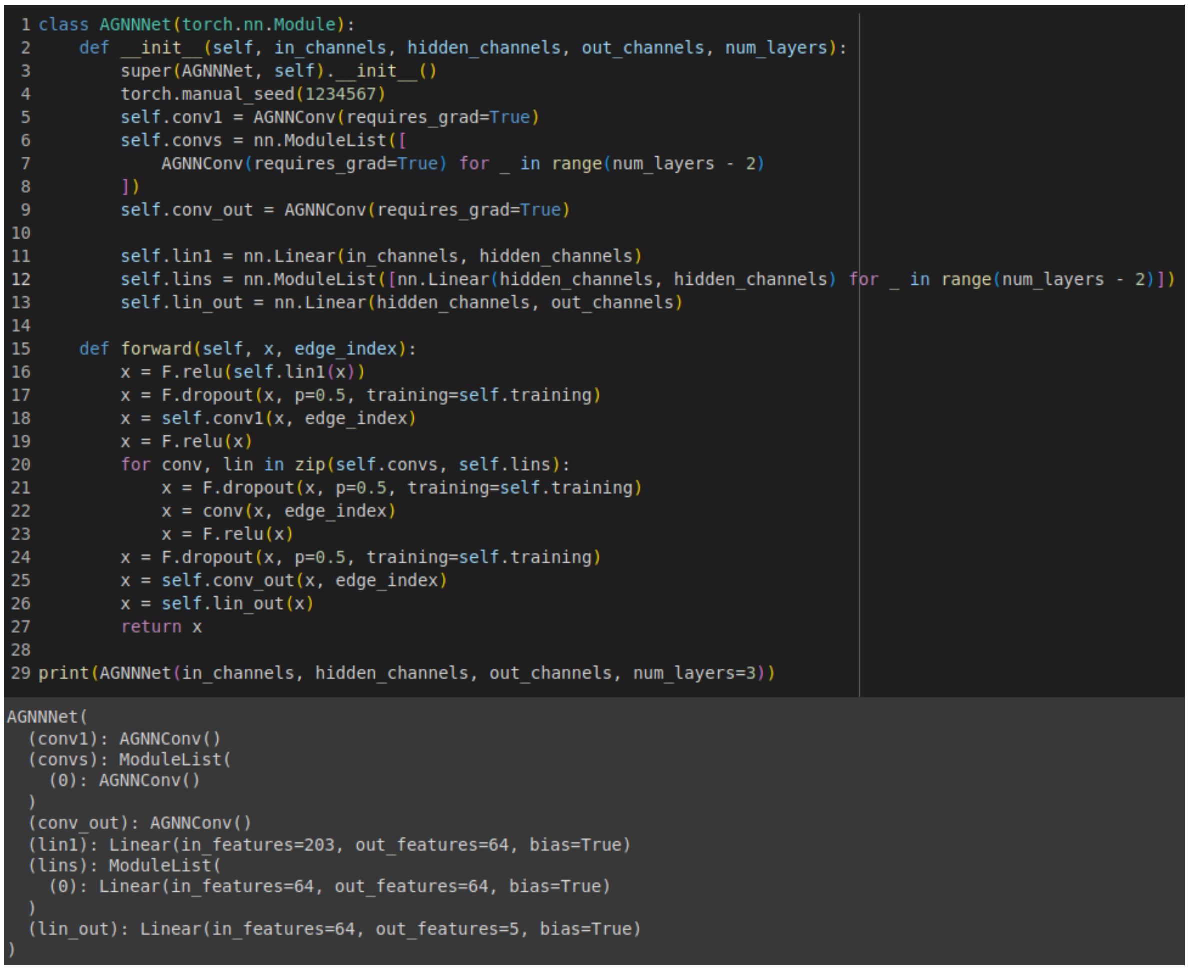
where the propagation matrix $\mathbf{P}$ is computed as

$$P_{i,j} = \frac{\exp(\beta \cdot \cos(\mathbf{x}_i, \mathbf{x}_j))}{\sum_{k \in \mathcal{N}(i) \cup \{ i \}} \exp(\beta \cdot \cos(\mathbf{x}_i, \mathbf{x}_k))}$$

with trainable parameter β.

AGNNConv operator [16]

Here is the code snippet for the model architecture.



```
class AGNNNet(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, num_layers):
        super(AGNNNet, self).__init__()
        torch.manual_seed(1234567)
        self.conv1 = AGNNConv(requires_grad=True)
        self.convs = nn.ModuleList([
            AGNNConv(requires_grad=True) for _ in range(num_layers - 2)
        ])
        self.conv_out = AGNNConv(requires_grad=True)

        self.lin1 = nn.Linear(in_channels, hidden_channels)
        self.lins = nn.ModuleList([nn.Linear(hidden_channels, hidden_channels) for _ in range(num_layers - 2)])
        self.lin_out = nn.Linear(hidden_channels, out_channels)

    def forward(self, x, edge_index):
        x = F.relu(self.lin1(x))
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        for conv, lin in zip(self.convs, self.lins):
            x = F.dropout(x, p=0.5, training=self.training)
            x = conv(x, edge_index)
            x = F.relu(x)
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv_out(x, edge_index)
        x = self.lin_out(x)
        return x

print(AGNNNet(in_channels, hidden_channels, out_channels, num_layers=3))
```

```
AGNNNet(
  (conv1): AGNNConv()
  (convs): ModuleList(
    (0): AGNNConv()
  )
  (conv_out): AGNNConv()
  (lin1): Linear(in_features=203, out_features=64, bias=True)
  (lins): ModuleList(
    (0): Linear(in_features=64, out_features=64, bias=True)
  )
  (lin_out): Linear(in_features=64, out_features=5, bias=True)
)
```

V. **GIN convolution network -** The GINCN model uses a series of GINConv layers, applying linear transformations with trainable parameters to process the graph-structured data. These layers, starting with input to hidden spaces and terminating in an output space, leverage ReLU activation for introducing non-linearity. The architecture, controlled by num_layers and num_heads, allows for learning intricate patterns. Its design enables comprehensive feature transformations across multiple layers.

$$\mathbf{x}'_i = h_{\mathbf{\Theta}}\left((1+\epsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right)$$

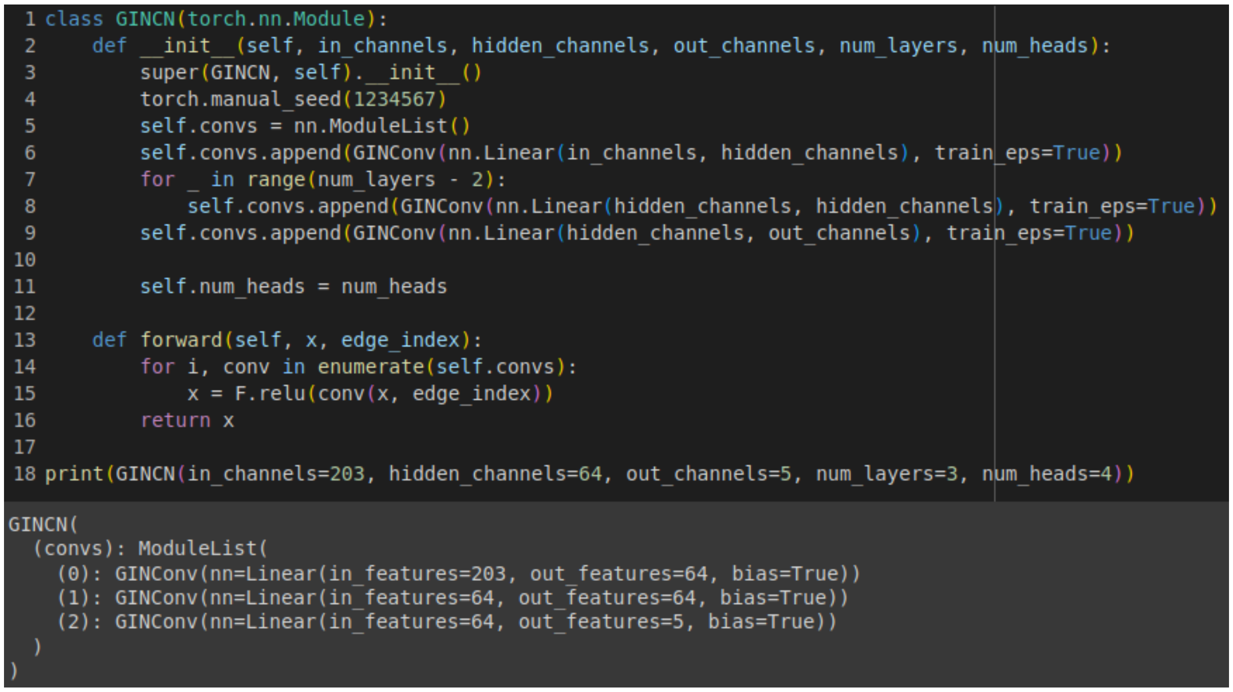
or

$$\mathbf{X}' = h_{\mathbf{\Theta}}\left(\left(\mathbf{A} + (1+\epsilon)\cdot\mathbf{I}\right)\cdot\mathbf{X}\right),$$

here $h_{\mathbf{\Theta}}$ denotes a neural network, .*i.e.* an MLP.

GINConv operator [16]

Here is the code snippet for the model architecture.



```
class GINCN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, num_layers, num_heads):
        super(GINCN, self).__init__()
        torch.manual_seed(1234567)
        self.convs = nn.ModuleList()
        self.convs.append(GINConv(nn.Linear(in_channels, hidden_channels), train_eps=True))
        for _ in range(num_layers - 2):
            self.convs.append(GINConv(nn.Linear(hidden_channels, hidden_channels), train_eps=True))
        self.convs.append(GINConv(nn.Linear(hidden_channels, out_channels), train_eps=True))

        self.num_heads = num_heads

    def forward(self, x, edge_index):
        for i, conv in enumerate(self.convs):
            x = F.relu(conv(x, edge_index))
        return x

print(GINCN(in_channels=203, hidden_channels=64, out_channels=5, num_layers=3, num_heads=4))
```

```
GINCN(
  (convs): ModuleList(
    (0): GINConv(nn=Linear(in_features=203, out_features=64, bias=True))
    (1): GINConv(nn=Linear(in_features=64, out_features=64, bias=True))
    (2): GINConv(nn=Linear(in_features=64, out_features=5, bias=True))
  )
)
```

4.4 Training and testing models

Having implemented the models, we create a *trainTestModel* function which trains on a model passed as a parameter on the provided data for a specified number of iterations. Number of classes in the dataset currently in use is also passed as a parameter for easily adding the results in a table. It uses an Adam optimizer and Cross Entropy Loss criterion. *train* and *test* are functions implemented within the function for applying them within a loop. During training, it evaluates and stores training and validation losses along with their respective accuracies. After training, it performs a final evaluation on the test data and stores the accuracy, precision, recall, and F1 score evaluated using *precision_recall_fscore_support function* in sklearn.metrics. The results are appended to a global dataframe results. Additionally, it also generates plots illustrating the training and validation losses, as well as the validation and test accuracies per epoch. This aids in visualizing the model's performance throughout the training process. Following is the code for the entire process

```
def trainTestModel(model, name, data, classes=5, iterations=100):
   global k, results

   test_acc = 0
   train_losses = []
   val_losses = []
   val_accuracies = []
   test_accuracies = []
         optimizer    =    torch.optim.Adam(model.parameters(),    lr=0.01,
weight_decay=5e-4)
   criterion = torch.nn.CrossEntropyLoss()

   def train():
       model.train()
       optimizer.zero_grad()
       out = model(data.x, data.edge_index)
       loss = criterion(out[data.train_mask], data.y[data.train_mask])
       loss.backward()
       optimizer.step()
       return loss.item()

   def evaluate(loader, mask):
       model.eval()
       out = model(data.x, data.edge_index)
```

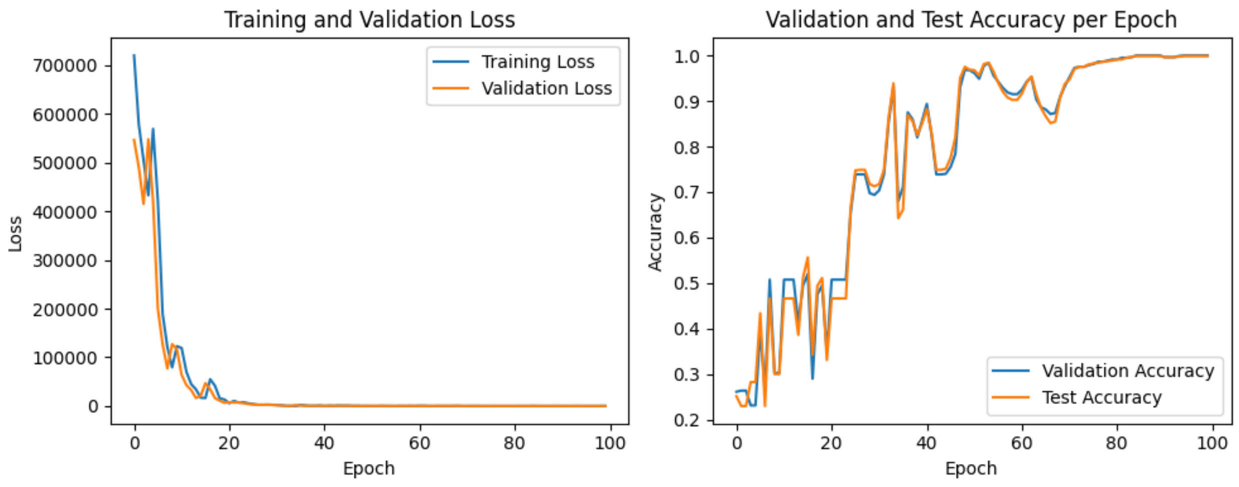
```
        loss = criterion(out[mask], data.y[mask]).item()
        _, predicted = out.max(dim=1)
        correct = predicted[mask].eq(data.y[mask]).sum().item()
        total = mask.sum().item()
        accuracy = correct / total
        return loss, accuracy

    def final_test(loader, mask):
        model.eval()
        out = model(data.x, data.edge_index)
        loss = criterion(out[mask], data.y[mask]).item()
        _, predicted = out.max(dim=1)
        predicted = predicted[mask]
        labels = data.y[mask]

        correct = predicted.eq(labels).sum().item()
        total = mask.sum().item()
        accuracy = correct / total

                                  precision,     recall,     f1,     _     =
precision_recall_fscore_support(labels.cpu(),             predicted.cpu(),
average='macro', zero_division=1)
        return loss, accuracy, precision, recall, f1


    for epoch in range(1, iterations + 1):
        loss = train()
        train_losses.append(loss)

        val_loss, val_acc = evaluate(data.val_mask, data.val_mask)
        val_losses.append(val_loss)
        val_accuracies.append(val_acc)
        _ , acc = evaluate(data.test_mask, data.test_mask)
        test_accuracies.append(acc)
                print(f'Epoch:  {epoch:03d},  Loss:  {loss:.4f},  Val  Loss:
{val_loss:.4f}, Val Accuracy: {val_acc:.4f}')

        _  ,  acc,  precision,  recall,  f1  =  final_test(data.test_mask,
data.test_mask)
    results.loc[k] = [classes, name, acc * 100, precision, recall, f1]
```

```
 k += 1

# Plotting the graphs
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(val_accuracies, label='Validation Accuracy')
plt.plot(test_accuracies, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Validation and Test Accuracy per Epoch')
plt.legend()

plt.tight_layout()
plt.show()
```

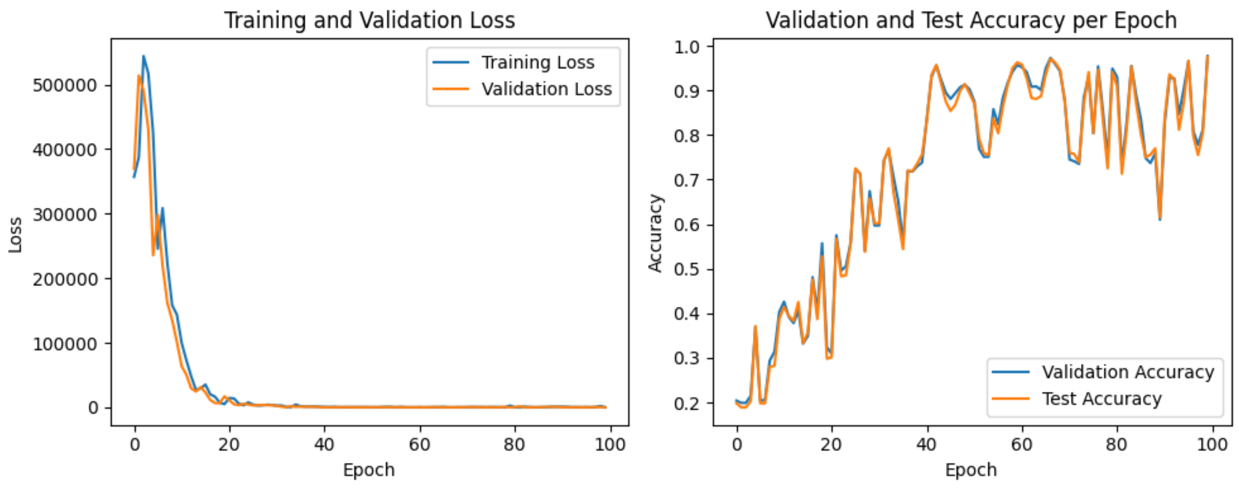
5. Results:

Table 4 - Results

Number of classes	Model	Accuracy	Precision	Recall	F1-Score
4	2_layer_gcn	89.000	0.929936	0.880435	0.880688
	3_layer_gcn	99.875	0.998762	0.998894	0.998825
	4_layer_gcn	99.750	0.997312	0.997560	0.997471
	5_layer_gcn	99.750	0.997542	0.997560	0.997593
	graph_sage_network	99.625	0.995989	0.996248	0.996094
	gat	78.000	0.807845	0.770268	0.761767
	attention_graph_sage_net	99.750	0.997411	0.997571	0.997485
	gin_cn	23.625	0.809063	0.250000	0.095551
5	2_layer_gcn	90.000	0.936508	0.897436	0.893299
	3_layer_gcn	97.500	0.979167	0.974359	0.975312
	4_layer_gcn	80.9	0.855638	0.821371	0.784417
	5_layer_gcn	62.3	0.794901	0.635692	0.562118
	graph_sage_network	70.9	0.724419	0.704490	0.641684
	gat	60.5	0.657098	0.604998	0.545802
	attention_graph_sage_net	94.6	0.959049	0.944583	0.945571
	gin_cn	18.9	0.837800	0.200000	0.063583

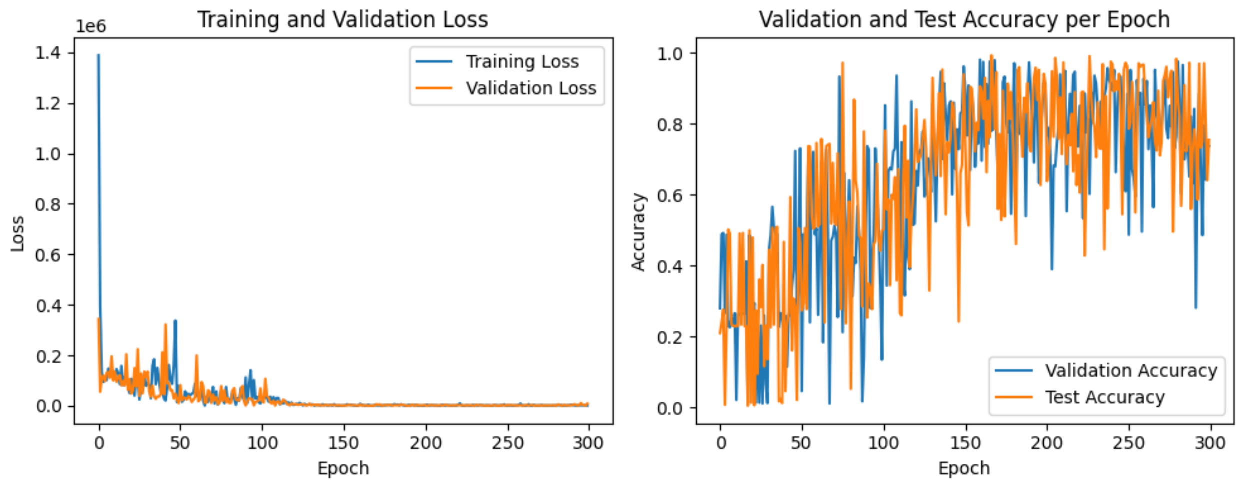


Graph 1: Training loss and accuracy of 3 layer GCN on 4 classes



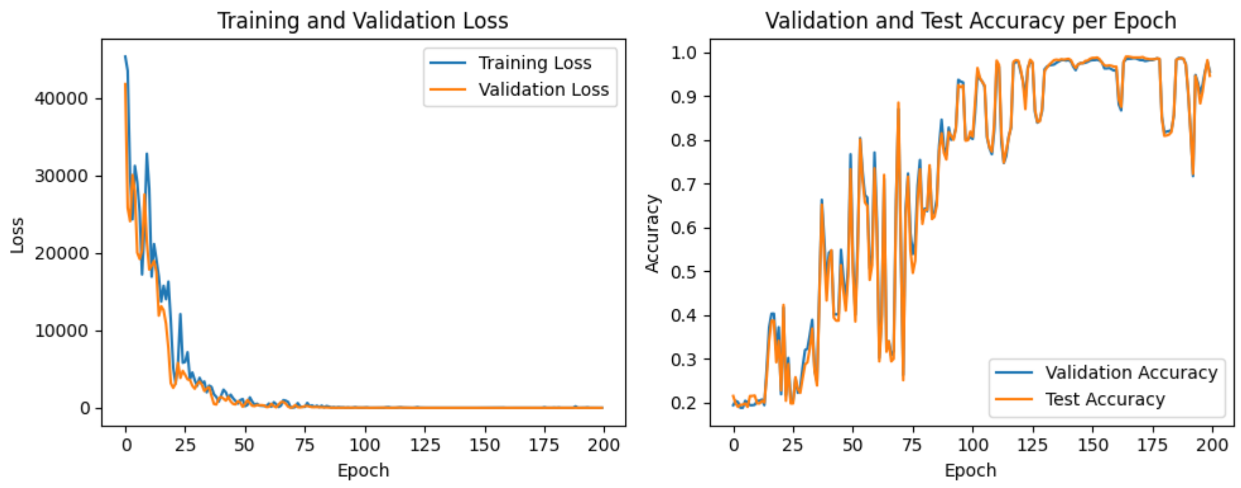
Graph 2: Training loss and accuracy of 3 layer GCN on 5 classes

The best results were achieved by a 3 layer GCN although the accuracy of other gcns and sage networks is very close but they lag behind when compared with other metrics. The model started achieving stable results on training dataset within the first 30 iterations. The comparison with other GCNs with different number of layers show that shallower GCN could adequately learn the relevant representations within the dataset for both 4 and 5 number of classes.



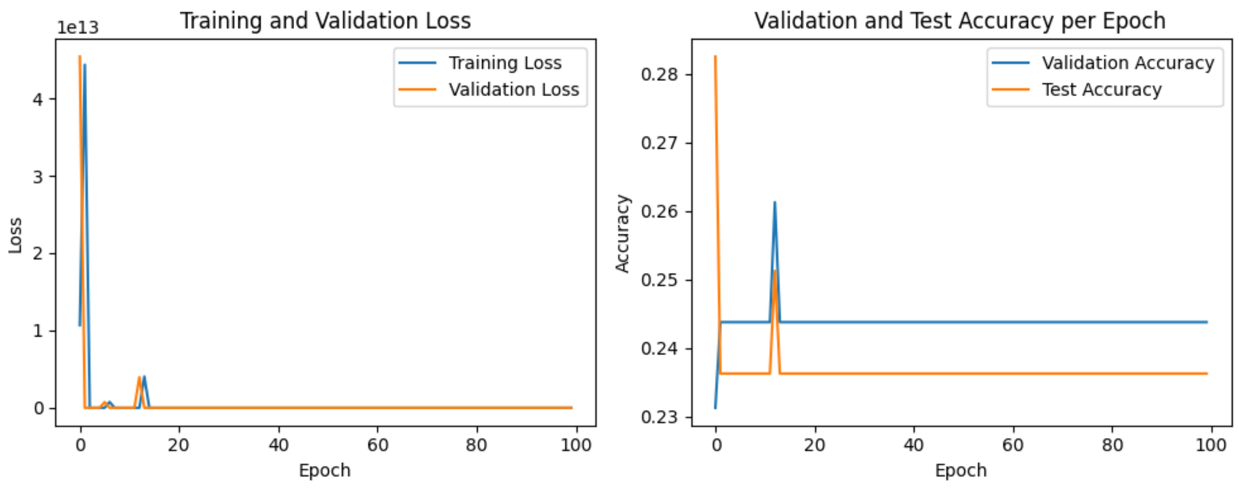
Graph 3: Training loss and accuracy of GAT on 4 classes

Accuracy of GAT and some more models fluctuated a lot during training although loss stayed stable after the first 100 iterations which suggests the model is overly sensitive to variations in the dataset. In case of attention models, their attention mechanisms might not be essential for the task, leading to over-parameterization, thus giving poorer results. It could also be the case that the dataset is not having enough complexity or structure to benefit from the inherent attention.

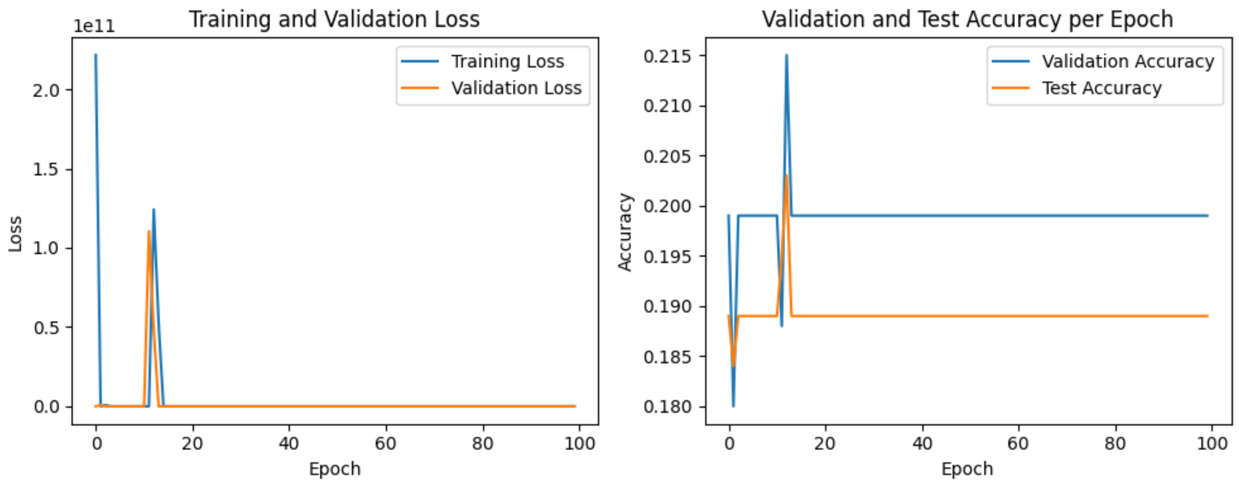


Graph 4: Training loss and accuracy of Attention Graph Sage net on 5 classes

The only model that came close to performance measure of the best GCN was Attention Graph Sage utilising AGNNConv operator. The results show that with a diverse dataset with multiple classes, attention mechanisms increase the performance.



Graph 5: Training loss and accuracy of GIN on 4 classes



Graph 6: Training loss and accuracy of GIN on 5 classes

The poorest performance in both cases (number of classes=4,5) was that of GINCN, which was experimented with different combinations of hyperparameters and the best results were obtained hidden_channels=64, num_layers=4, num_heads=4. The poor performance may be due to several factors, model's complexity not aligning with the dataset's characteristics, leading to limited adaptability or GINCN's assumption about data representation might not be matching the dataset structure, causing inefficiencies. Lastly, GINCN's expressive capabilities could be insufficient for the intricate relationships within music genres.

6. Discussion:

Despite employing various models, including GATs and complex architectures, the GCN models consistently outperformed others in the classification task which suggests that for GTZAN dataset, the simpler GCN architectures were more effective in capturing and leveraging the inherent relationships within the music data. The GCNs demonstrated better generalization, classification accuracy and other metrics ass compared to more complex models, showcasing the importance of model simplicity and relevance to the dataset's characteristics in achieving state-of-the-art performance. The dataset likely contains music genres that are distinguishable by simpler relationships or structures as GCNs are able to capture the relationships more effectively and as a result, complex models with attention mechanisms, such as GATs, may not be necessary or might even struggle to adapt to the dataset's characteristics.

Table 5 - Comparison of results with previous work

Model	**Number of classes**	**Accuracy (in %)**
GMM [1]	10	61
SVM and LDA [2]	10	78.5
CNN [4]	4	84
SVM adaboost [4]	10	81
Ensemble [5]	5	82.25
LSTM and SVM [6]	10	89
Neural Networks [7]	4	96
3 layer GCN	**4**	**99.875**
3 layer GCN	**5**	**97.5**

Experimenting with various architectures like Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and others required a lot of understanding of their working and tuning hyperparameters etc. Due to music data having high dimensionality models were prone to overfitting, balancing the tradeoff between model complexity and performance became a challenging task. Even during training, training the models on large datasets with complex architecturesdemanded significant computational resources and time often leading the system to crash even with advanced modern GPUs, to overcome which only a subset of total number of classes was taken into consideration.

7. Conclusion:

Graph Convolutional Networks (GCNs) with moderate depth (3 layers) outperformed all other architectures by achieving better results

Table 5 - Final optimal results

Model	Classes	Accuracy	Precision	Recall	F1-Score
3 layer gcn	4	99.875	0.998762	0.998894	0.998825
3 layer gcn	5	97.5	0.979167	0.974359	0.975312

and stability in comparison to deeper networks like GAT and multi-layer GCNs or GINCN. The GTZAN dataset does not fully benefit from the complexities of attention mechanisms (like GATs). Simpler models like GCNs seem more effective in capturing essential relationships for the GTZAN dataset. Complex models like GATs might be overparameterized for this particular dataset, indicating that the dataset's structure does not require intricate attention mechanisms. Due to the challenges in feature extraction, dataset variability, and computational intensity, future work could explore better feature representations, address dataset noise, and experiment with different architectures or model ensembles, or come up with better representation for the dataset using a new graph operator.

8. References

[1] Tzanetakis, George & Cook, Perry. (2002). Musical Genre Classification of Audio Signals. IEEE Transactions on Speech and Audio Processing. 10. 293 - 302. 10.1109/TSA.2002.800560.

[2] Tao Li and G. Tzanetakis, "Factors in automatic musical genre classification of audio signals," 2003 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (IEEE Cat. No.03TH8684), New Paltz, NY, USA, 2003, pp. 143-146, doi: 10.1109/ASPAA.2003.1285840.

[3] Li, Tom & Chan, Antoni & Chun, Andy. (2010). Automatic Musical Pattern Feature Extraction Using Convolutional Neural Network. Lecture Notes in Engineering and Computer Science. 2180.

[4] Chathuranga, Devindu & Jayaratne, Lakshman. (2013). Automatic Music Genre Classification of Audio Signals with Machine Learning Approaches. GSTF Journal on Computing (JoC). 3. 10.7603/s40601-013-0014-0.

[5] Rudra Chandra Gupta, Parth Rohilla, Harpreet Singh, Neeraj Kumar and Prashant Singh Rana, "Automated Music Genre Classification of Audio Signals using Ensemble Technique" in ICMLDS 2018, at Mahindra Ecole Centrale, Hyderabad.

[6] Fulzele, Prasenjeet & Singh, Rajat & Kaushik, Naman & Pandey, Kavita. (2018). A Hybrid Model for Music Genre Classification Using LSTM and SVM. 1-3. 10.1109/IC3.2018.8530557.

[7] Haggblade, Michael, Yang Hong, and Kenny Kao. "Music genre classification." *Department of Computer Science, Stanford University* 131 (2011): 132.

[8] Tang, C. P., Chui, K. L., Yu, Y. K., Zeng, Z., and Wong, K. H., "Music genre classification using a hierarchical long short term memory (LSTM) model", in *Third International Workshop on Pattern Recognition*, 2018, vol. 10828. doi:10.1117/12.2501763.

[9] Kipf, T. N., & Welling, M. (2016). Semi-Supervised Classification with Graph Convolutional Networks. *ArXiv*. /abs/1609.02907

[10] Hamilton, W. L., Ying, R., & Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. *ArXiv*. /abs/1706.02216

[11] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2017). Graph Attention Networks. *ArXiv*. /abs/1710.10903

[12] Thekumparampil, K. K., Wang, C., Oh, S., & Li, L. (2018). Attention-based Graph Neural Network for Semi-supervised Learning. *ArXiv*. /abs/1803.03735
[13] Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2018). How Powerful are Graph Neural Networks? *ArXiv*. /abs/1810.00826
[14] Chroma feature. (2023, July 27). In *Wikipedia*. https://en.wikipedia.org/wiki/Chroma_feature
[15] PyG Documentation latest release march 202 https://pytorch-geometric.readthedocs.io/en/latest/