Configuration Manual

MSc Research Project
Data Analytics

Yash Bhargava
Student ID: x22220861

School of Computing
National College of Ireland

Supervisor: Dr. Giovani Estrada

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Yash Bhargava
Student ID:	x22220861
Programme:	Data Analytics **Year:** 2024
Module:	MSc Research Project
Lecturer:	Dr. Giovani Estrada
Submission Due Date:	12-08-2024
Project Title:	A study of Graph Neural Networks and Graph Attention Networks for Node Classification
Word Count:	807 **Page Count:** 24

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Yash Bhargava

Date: 11-08-2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Yash Bhargava
Student ID: x22220861

1 Introduction

For the successful completion of this study i.e., ***Improving Node Classification in Term-Document matrices using Advanced Graph Neural Networks (24)***. The author has used various system configurations, hardware and software requirements for the successful completion of this study.

The structure of this configuration manual is divided into many sections: such as in the section 2 all the system configuration requirements have been discussed. In the next section 3, how the project has been developed using user defined functions and classes for the layers and more. In the section 0, the report has discussed about the various preprocessing steps used for building the proposed models, in the section 0 the proposed baseline architectures in this study were discussed and lastly the references.

2 System Configuration

The following system configuration was used for the successful development of this project.

2.1 Hardware Requirements

Device	HP Laptop 15s-du3047TX
Operating System	Windows 11 x64-bit
RAM	8.00 GB
CPU	Intel® Core (TM) i5-1135G7 @ 2.40GHz
GPU	Intel® Iris Xe Graphics, NVIDIA GeForce MX350

2.2 Software Requirements

Software Used	Visual Studio Code
Language Used	Python
Others	MS Word, overleaf

3 Project Development

In this section, all the required python libraries and loading the data and getting into the right format of that data was showed

3.1 Importing the required Python libraries

```
import os
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import keras_tuner as kt
import seaborn as sns
from sklearn.metrics import confusion_matrix
import warnings
from sklearn.metrics import classification_report

warnings.filterwarnings("ignore")
pd.set_option("display.max_columns", 6)
pd.set_option("display.max_rows", 6)
np.random.seed(2)
```

3.2 Fetching the dataset using link and loading the dataset into notebook

```
zip_file = keras.utils.get_file(
    fname="cora.tgz",
    origin="https://linqs-data.soe.ucsc.edu/public/lbc/cora.tgz",
    extract=True,
)
data_dir = os.path.join(os.path.dirname(zip_file), "cora")
```

```
citations = pd.read_csv(
    os.path.join(data_dir, "cora.cites"),
    sep="\t",
    header=None,
    names=["target", "source"],
)
print("Citations shape:", citations.shape)
```

```
Citations shape: (5429, 2)
```

3.3 Exploratory Data Analysis

```
citations.sample(frac=1).head()
```

	target	source
2025	15429	217115
2832	34257	34266
2321	20924	289885
58	35	15670
2052	15987	523394

```
column_names = ["paper_id"] + [f"term_{idx}" for idx in range(1433)] + ["subject"]
papers = pd.read_csv(
    os.path.join(data_dir, "cora.content"),
    sep="\t",
    header=None,
    names=column_names,
)

print("Papers shape:", papers.shape)
```

```
Papers shape: (2708, 1435)
```

```
print(papers.sample(5).T)
```

```
             1050               1408     522            2549       2308
paper_id    32276             593329  358884          263553    1138027
term_0          0                  0       0               0          0
term_1          0                  0       0               0          0
term_2          0                  0       0               0          0
term_3          0                  0       0               0          0
...           ...                ...     ...             ...        ...
term_1429       0                  0       0               0          0
term_1430       0                  0       0               1          0
term_1431       0                  0       0               0          0
term_1432       0                  0       0               0          0
subject    Theory  Genetic_Algorithms  Theory  Neural_Networks  Case_Based

[1435 rows x 5 columns]
```

```
print(papers.subject.value_counts())
```

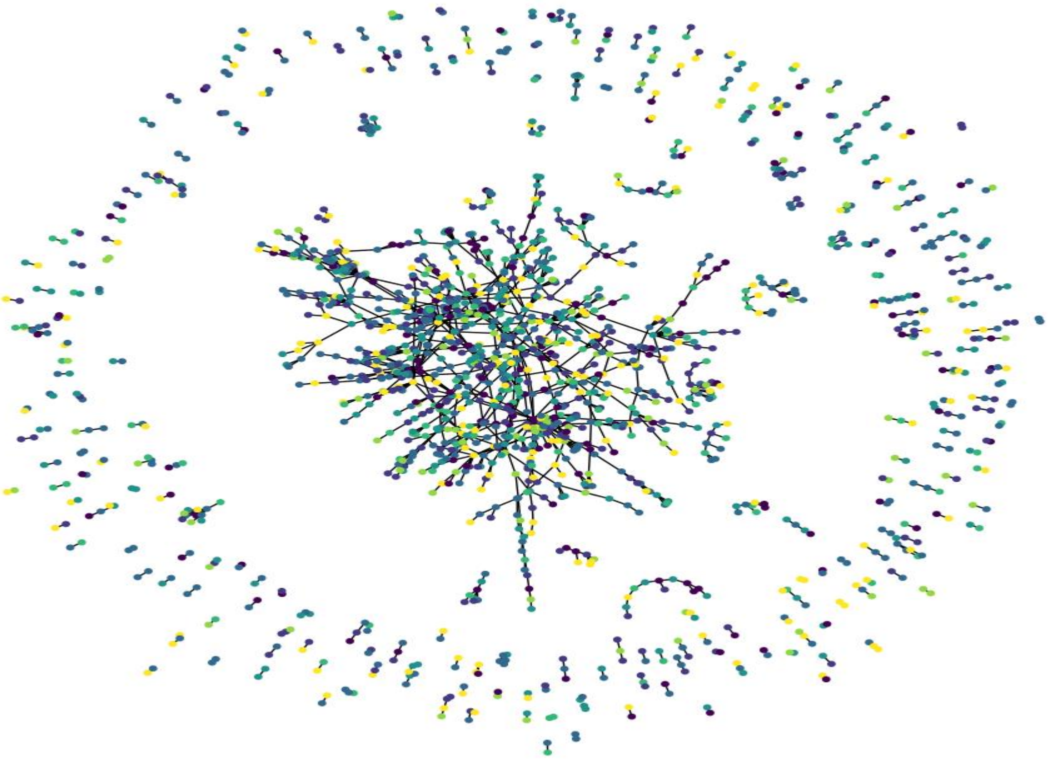
```
subject
Neural_Networks           818
Probabilistic_Methods     426
Genetic_Algorithms        418
Theory                    351
Case_Based                298
Reinforcement_Learning    217
Rule_Learning             180
Name: count, dtype: int64
```

```
class_values = sorted(papers["subject"].unique())
class_idx = {name: id for id, name in enumerate(class_values)}
paper_idx = {name: idx for idx, name in enumerate(sorted(papers["paper_id"].unique()))}

papers["paper_id"] = papers["paper_id"].apply(lambda name: paper_idx[name])
citations["source"] = citations["source"].apply(lambda name: paper_idx[name])
citations["target"] = citations["target"].apply(lambda name: paper_idx[name])
papers["subject"] = papers["subject"].apply(lambda value: class_idx[value])
```

3.4 Construction of the graphs

```
plt.figure(figsize=(10, 10))
colors = papers["subject"].tolist()
cora_graph = nx.from_pandas_edgelist(citations.sample(n=1500))
subjects = list(papers[papers["paper_id"].isin(list(cora_graph.nodes))]["subject"])
nx.draw_spring(cora_graph, node_size=15, node_color=subjects)
```



3.5 Train-test split and defining hyperparameters

```
train_data, test_data = [], []

for _, group_data in papers.groupby("subject"):
    # Select around 80% of the dataset for training.
    random_selection = np.random.rand(len(group_data.index)) <= 0.5
    train_data.append(group_data[random_selection])
    test_data.append(group_data[~random_selection])

train_data = pd.concat(train_data).sample(frac=1)
test_data = pd.concat(test_data).sample(frac=1)

print("Train data shape:", train_data.shape)
print("Test data shape:", test_data.shape)
```

```
Train data shape: (1365, 1435)
Test data shape: (1343, 1435)
```

```
hidden_units = [32, 32]
learning_rate = 0.01
dropout_rate = 0.5
num_epochs = 300
batch_size = 256
```

3.6 User defined function for training the models

```
def run_experiment(model, x_train, y_train):
    # Compile the model.
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate),
        loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=[keras.metrics.SparseCategoricalAccuracy(name="acc")],
    )
    # Create an early stopping callback.
    early_stopping = keras.callbacks.EarlyStopping(
        monitor="val_acc", patience=50, restore_best_weights=True
    )
    # Fit the model.
    history = model.fit(
        x=x_train,
        y=y_train,
        epochs=num_epochs,
        batch_size=batch_size,
        validation_split=0.15,
        callbacks=[early_stopping],
    )

    return history
```

3.7 User-defined function for plotting curves

```
def display_learning_curves(history):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    ax1.plot(history.history["loss"])
    ax1.plot(history.history["val_loss"])
    ax1.legend(["train", "test"], loc="upper right")
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")

    ax2.plot(history.history["acc"])
    ax2.plot(history.history["val_acc"])
    ax2.legend(["train", "test"], loc="upper right")
    ax2.set_xlabel("Epochs")
    ax2.set_ylabel("Accuracy")
    plt.show()
```

4 Data Preprocessing

4.1 Defining the function for creating feed forward network

```
def create_ffn(hidden_units, dropout_rate, name=None):
    fnn_layers = []

    for units in hidden_units:
        fnn_layers.append(layers.BatchNormalization())
        fnn_layers.append(layers.Dropout(dropout_rate))
        fnn_layers.append(layers.Dense(units, activation=tf.nn.gelu))

    return keras.Sequential(fnn_layers, name=name)
```

4.2 Train-test split

```
feature_names = list(set(papers.columns) - {"paper_id", "subject"})
num_features = len(feature_names)
num_classes = len(class_idx)

x_train = train_data[feature_names].to_numpy()
x_test = test_data[feature_names].to_numpy()

y_train = train_data["subject"]
y_test = test_data["subject"]
```

4.3 Fetching the shape of nodes and edges

```
# Create an edges array (sparse adjacency matrix) of shape [2, num_edges].
edges = citations[["source", "target"]].to_numpy().T
# Create an edge weights array of ones.
edge_weights = tf.ones(shape=edges.shape[1])
# Create a node features array of shape [num_nodes, num_features].
node_features = tf.cast(
    papers.sort_values("paper_id")[feature_names].to_numpy(), dtype=tf.dtypes.float32
)
# Create graph info tuple with node_features, edges, and edge_weights.
graph_info = (node_features, edges, edge_weights)

print("Edges shape:", edges.shape)
print("Nodes shape:", node_features.shape)
```

```
Edges shape: (2, 5429)
Nodes shape: (2708, 1433)
```

4.4 Definition of Graph Convolutional layers

In this section, we have used the code for defining the graph convolutional layers to build the proposed GNN architectures.

```
def create_gru(hidden_units, dropout_rate):
    inputs = keras.layers.Input(shape=(2, hidden_units[0]))
    x = inputs
    for units in hidden_units:
        x = layers.GRU(
            units=units,
            activation="tanh",
            recurrent_activation="sigmoid",
            return_sequences=True,
            dropout=dropout_rate,
            return_state=False,
            recurrent_dropout=dropout_rate,
        )(x)
    return keras.Model(inputs=inputs, outputs=x)
```

```
class GraphConvLayer(layers.Layer):
    def __init__(
        self,
        hidden_units,
        dropout_rate=0.2,
        aggregation_type="mean",
        combination_type="concat",
        normalize=False,
        *args,
        **kwargs,
    ):
        super().__init__(*args, **kwargs)

        self.aggregation_type = aggregation_type
        self.combination_type = combination_type
        self.normalize = normalize

        self.ffn_prepare = create_ffn(hidden_units, dropout_rate)
        if self.combination_type == "gru":
            self.update_fn = create_gru(hidden_units, dropout_rate)
        else:
            self.update_fn = create_ffn(hidden_units, dropout_rate)

    def prepare(self, node_repesentations, weights=None):
        # node_repesentations shape is [num_edges, embedding_dim].
        messages = self.ffn_prepare(node_repesentations)
        if weights is not None:
            messages = messages * tf.expand_dims(weights, -1)
        return messages
```

```python
    def aggregate(self, node_indices, neighbour_messages, node_repesentations):
        # node_indices shape is [num_edges].
        # neighbour_messages shape: [num_edges, representation_dim].
        # node_repesentations shape is [num_nodes, representation_dim]
        num_nodes = node_repesentations.shape[0]
        if self.aggregation_type == "sum":
            aggregated_message = tf.math.unsorted_segment_sum(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        elif self.aggregation_type == "mean":
            aggregated_message = tf.math.unsorted_segment_mean(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        elif self.aggregation_type == "max":
            aggregated_message = tf.math.unsorted_segment_max(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        else:
            raise ValueError(f"Invalid aggregation type: {self.aggregation_type}.")

        return aggregated_message

    def update(self, node_repesentations, aggregated_messages):

        if self.combination_type == "gru":
            h = tf.stack([node_repesentations, aggregated_messages], axis=1)
        elif self.combination_type == "concat":
            h = tf.concat([node_repesentations, aggregated_messages], axis=1)
        elif self.combination_type == "add":
            h = node_repesentations + aggregated_messages
        else:
            raise ValueError(f"Invalid combination type: {self.combination_type}.")

        node_embeddings = self.update_fn(h)
        if self.combination_type == "gru":
```

```python
            node_embeddings = tf.unstack(node_embeddings, axis=1)[-1]

        if self.normalize:
            node_embeddings = tf.nn.l2_normalize(node_embeddings, axis=-1)
        return node_embeddings

    def call(self, inputs):


        node_repesentations, edges, edge_weights = inputs
        # Get node_indices (source) and neighbour_indices (target) from edges.
        node_indices, neighbour_indices = edges[0], edges[1]
        # neighbour_repesentations shape is [num_edges, representation_dim].
        neighbour_repesentations = tf.gather(node_repesentations, neighbour_indices)

        # Prepare the messages of the neighbours.
        neighbour_messages = self.prepare(neighbour_repesentations, edge_weights)
        # Aggregate the neighbour messages.
        aggregated_messages = self.aggregate(
            node_indices, neighbour_messages, node_repesentations
        )
        # Update the node embedding with the neighbour messages.
        return self.update(node_repesentations, aggregated_messages)
```

4.5 Definition the class of Graph Attention Layers

In this section, we have used the code for defining the graph attention layers to build the proposed GAT architectures (24).

```
class GraphAttention(layers.Layer):
    def __init__(
        self,
        units,
        kernel_initializer="glorot_uniform",
        kernel_regularizer=None,
        **kwargs,
    ):
        super().__init__(**kwargs)
        self.units = units
        self.kernel_initializer = keras.initializers.get(kernel_initializer)
        self.kernel_regularizer = keras.regularizers.get(kernel_regularizer)

    def build(self, input_shape):
        self.kernel = self.add_weight(
            shape=(input_shape[0][-1], self.units),
            trainable=True,
            initializer=self.kernel_initializer,
            regularizer=self.kernel_regularizer,
            name="kernel",
        )
        self.kernel_attention = self.add_weight(
            shape=(self.units * 2, 1),
            trainable=True,
            initializer=self.kernel_initializer,
            regularizer=self.kernel_regularizer,
            name="kernel_attention",
        )
        self.built = True

    def call(self, inputs):
        node_states, edges = inputs
```

```
        # Linearly transform node states
        node_states_transformed = tf.matmul(node_states, self.kernel)

        # (1) Compute pair-wise attention scores
        node_states_expanded = tf.gather(node_states_transformed, edges)
        node_states_expanded = tf.reshape(
            node_states_expanded, (tf.shape(edges)[0], -1)
        )
        attention_scores = tf.nn.leaky_relu(
            tf.matmul(node_states_expanded, self.kernel_attention)
        )
        attention_scores = tf.squeeze(attention_scores, -1)

        # (2) Normalize attention scores
        attention_scores = tf.math.exp(tf.clip_by_value(attention_scores, -2, 2))
        attention_scores_sum = tf.math.unsorted_segment_sum(
            data=attention_scores,
            segment_ids=edges[:, 0],
            num_segments=tf.reduce_max(edges[:, 0]) + 1,
        )
        attention_scores_sum = tf.repeat(
            attention_scores_sum, tf.math.bincount(tf.cast(edges[:, 0], "int32"))
        )
        attention_scores_norm = attention_scores / attention_scores_sum

        # (3) Gather node states of neighbors, apply attention scores and aggregate
        node_states_neighbors = tf.gather(node_states_transformed, edges[:, 1])
        out = tf.math.unsorted_segment_sum(
            data=node_states_neighbors * attention_scores_norm[:, tf.newaxis],
            segment_ids=edges[:, 0],
            num_segments=tf.shape(node_states)[0],
        )
        return out
```

```
class MultiHeadGraphAttention(layers.Layer):
    def __init__(self, units, num_heads=8, merge_type="concat", **kwargs):
        super().__init__(**kwargs)
        self.num_heads = num_heads
        self.merge_type = merge_type
        self.attention_layers = [GraphAttention(units) for _ in range(num_heads)]

    def call(self, inputs):
        atom_features, pair_indices = inputs

        # Obtain outputs from each attention head
        outputs = [
            attention_layer([atom_features, pair_indices])
            for attention_layer in self.attention_layers
        ]
        # Concatenate or average the node states from each head
        if self.merge_type == "concat":
            outputs = tf.concat(outputs, axis=-1)
        else:
            outputs = tf.reduce_mean(tf.stack(outputs, axis=-1), axis=-1)
        # Activate and return node states
        return tf.nn.relu(outputs)

    def get_config(self):
        config = super(GraphAttentionNetwork, self).get_config()
        config.update({"units": self.units})
        return config
```

5 Model Architectures

5.1 Baseline Architecture

```python
class GNNNodeClassifier(tf.keras.Model):
    def __init__(
        self,
        graph_info,
        num_classes,
        hidden_units,
        aggregation_type="sum",
        combination_type="concat",
        dropout_rate=0.2,
        normalize=True,
        *args,
        **kwargs,
    ):
        super().__init__(*args, **kwargs)

        # Unpack graph_info to three elements: node_features, edges, and edge_weight.
        node_features, edges, edge_weights = graph_info
        self.node_features = node_features
        self.edges = edges
        self.edge_weights = edge_weights
        # Set edge_weights to ones if not provided.
        if self.edge_weights is None:
            self.edge_weights = tf.ones(shape=edges.shape[1])
        # Scale edge_weights to sum to 1.
        self.edge_weights = self.edge_weights / tf.math.reduce_sum(self.edge_weights)

        # Create a process layer.
        self.preprocess = create_ffn(hidden_units, dropout_rate, name="preprocess")
        # Create the first GraphConv layer.
        self.conv1 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv1",
        )
        # Create the second GraphConv layer.
        self.conv2 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv2",
        )
        # Create a postprocess layer.
        self.postprocess = create_ffn(hidden_units, dropout_rate, name="postprocess")
        # Create a compute logits layer.
        self.compute_logits = layers.Dense(units=num_classes, name="logits")

    def call(self, input_node_indices):
        # Preprocess the node_features to produce node representations.
        x = self.preprocess(self.node_features)
        # Apply the first graph conv layer.
        x1 = self.conv1((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x1 + x
        # Apply the second graph conv layer.
        x2 = self.conv2((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x2 + x
        # Postprocess node embedding.
        x = self.postprocess(x)
        # Fetch node embeddings for the input node_indices.
        node_embeddings = tf.gather(x, input_node_indices)
        # Compute logits
        return self.compute_logits(node_embeddings)
```

5.1.1 Summary of the Baseline Architecture

```python
gnn_model = GNNNodeClassifier(
    graph_info=graph_info,
    num_classes=num_classes,
    hidden_units=hidden_units,
    dropout_rate=dropout_rate,
    name="gnn_model",
)
node_indices = tf.constant([1, 10, 100], dtype=tf.int32)
output = gnn_model(node_indices)
print("GNN output shape:", output.shape)
gnn_model.summary()
```

```
GNN output shape: (3, 7)

Model: "gnn_model"
```

Layer (type)	Output Shape	Param #
preprocess (Sequential)	(2708, 32)	52,804
graph_conv1 (GraphConvLayer)	?	5,888
graph_conv2 (GraphConvLayer)	?	5,888
postprocess (Sequential)	(2708, 32)	2,368
logits (Dense)	(3, 7)	231

5.1.2 Evaluation of Baseline Architecture on test set

```python
x_test = test_data.paper_id.to_numpy()
_, test_accuracy = gnn_model.evaluate(x=x_test, y=y_test, verbose=0)
print(f"Test accuracy: {round(test_accuracy * 100, 2)}%")
```

```
Test accuracy: 69.17%
```

5.1.3 Prediction of randomly generated instances using baseline architecture

```python
print("Original node_features shape:", gnn_model.node_features.shape)
print("Original edges shape:", gnn_model.edges.shape)
gnn_model.node_features = new_node_features
gnn_model.edges = new_edges
gnn_model.edge_weights = tf.ones(shape=new_edges.shape[1])
print("New node_features shape:", gnn_model.node_features.shape)
print("New edges shape:", gnn_model.edges.shape)

logits = gnn_model.predict(tf.convert_to_tensor(new_node_indices))
probabilities = keras.activations.softmax(tf.convert_to_tensor(logits)).numpy()
display_class_probabilities(probabilities)
```

```
Original node_features shape: (2708, 1433)
Original edges shape: (2, 5429)
New node_features shape: (2715, 1433)
New edges shape: (2, 5478)
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
Instance 1:
- Case_Based: 28.55%
- Genetic_Algorithms: 48.27%
- Neural_Networks: 0.16%
- Probabilistic_Methods: 8.3%
- Reinforcement_Learning: 8.95%
- Rule_Learning: 0.45%
- Theory: 5.33%
```

5.2 Proposed GNN Node Classifier Architecture 1

```
class ProposedGNNNodeClassifier(tf.keras.Model):
    def __init__(
        self,
        graph_info,
        num_classes,
        hidden_units,
        aggregation_type="sum",
        combination_type="concat",
        dropout_rate=0.2,
        normalize=True,
        *args,
        **kwargs,
    ):
        super().__init__(*args, **kwargs)

        # Unpack graph_info to three elements: node_features, edges, and edge_weight.
        node_features, edges, edge_weights = graph_info
        self.node_features = node_features
        self.edges = edges
        self.edge_weights = edge_weights
        # Set edge_weights to ones if not provided.
        if self.edge_weights is None:
            self.edge_weights = tf.ones(shape=edges.shape[1])
        # Scale edge_weights to sum to 1.
        self.edge_weights = self.edge_weights / tf.math.reduce_sum(self.edge_weights)
```

```
        # Create a process layer.
        self.preprocess = create_ffn(hidden_units, dropout_rate, name="preprocess")
        # Create the first GraphConv layer.
        self.conv1 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv1",
        )
        # Create the second GraphConv layer.
        self.conv2 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv2",
        )
        # Create the third GraphConv layer.
        self.conv3 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv3",
        )
        # Create the fourth GraphConv layer.
        self.conv4 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
```

```
        self.conv5 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv5",
        )
        # Create the sixth GraphConv layer.
        self.conv6 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv6",
        )
        # Create the seventh GraphConv layer.
        self.conv7 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv7",
        )
        # Create the eighth GraphConv layer.
        self.conv8 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv8",
        )
    def call(self, input_node_indices):
        # Preprocess the node_features to produce node representations.
        x = self.preprocess(self.node_features)
        # Apply the first graph conv layer.
        x1 = self.conv1((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x1 + x
        # Apply the second graph conv layer.
        x2 = self.conv2((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x2 + x
        # Apply the third graph conv layer.
        x3 = self.conv3((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x3 + x
        # Apply the fourth graph conv layer.
        x4 = self.conv4((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x4 + x
        # Apply the fifth graph conv layer.
        x5 = self.conv5((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x5 + x
        # Apply the sixth graph conv layer.
        x6 = self.conv6((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x6 + x
        # Apply the sixth graph conv layer.
        x7 = self.conv7((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x7 + x
        # Apply the sixth graph conv layer.
        x8 = self.conv8((x, self.edges, self.edge_weights))
        # Skip connection.
        x = x8 + x
```

5.2.1 Summary of the Proposed GNN Architecture

```
proposed_gnn_model = ProposedGNNNodeClassifier(
    graph_info=graph_info,
    num_classes=num_classes,
    hidden_units=[100,100],
    dropout_rate=0.5,
    name="proposed_gnn_model",
)

node_indices = tf.constant([1, 10, 100], dtype=tf.int32)
output = proposed_gnn_model(node_indices)
#print("GNN output shape:", gnn_model([1, 10, 100]))
# Print the shape of the output
print("GNN output shape:", output.shape)

proposed_gnn_model.summary()
```

Model: "proposed_gnn_model"

Layer (type)	**Output Shape**	**Param #**
preprocess (Sequential)	(2708, 100)	159,632
graph_conv1 (GraphConvLayer)	?	52,400
graph_conv2 (GraphConvLayer)	?	52,400
graph_conv3 (GraphConvLayer)	?	52,400
graph_conv4 (GraphConvLayer)	?	52,400
graph_conv5 (GraphConvLayer)	?	52,400
graph_conv6 (GraphConvLayer)	?	52,400
graph_conv7 (GraphConvLayer)	?	52,400
graph_conv8 (GraphConvLayer)	?	52,400
postprocess (Sequential)	(2708, 100)	21,000
logits (Dense)	(3, 7)	707

Total params: 600,539 (2.29 MB)

Trainable params: 589,073 (2.25 MB)

Non-trainable params: 11,466 (44.79 KB)

5.2.2 Evaluation of Proposed GNN Architecture 1 on test data

```
x_test = test_data.paper_id.to_numpy()
_, test_accuracy = proposed_gnn_model.evaluate(x=x_test, y=y_test, verbose=0)
print(f"Test accuracy: {round(test_accuracy * 100, 2)}%")
```

```
Test accuracy: 70.96%
```

5.2.3 Prediction of randomly generated instances

```
print("Original node_features shape:", proposed_gnn_model.node_features.shape)
print("Original edges shape:", proposed_gnn_model.edges.shape)
proposed_gnn_model.node_features = new_node_features
proposed_gnn_model.edges = new_edges
proposed_gnn_model.edge_weights = tf.ones(shape=new_edges.shape[1])
print("New node_features shape:", proposed_gnn_model.node_features.shape)
print("New edges shape:", proposed_gnn_model.edges.shape)

logits = proposed_gnn_model.predict(tf.convert_to_tensor(new_node_indices))
probabilities = keras.activations.softmax(tf.convert_to_tensor(logits)).numpy()
display_class_probabilities(probabilities)
```

```
Original node_features shape: (2708, 1433)
Original edges shape: (2, 5429)
New node_features shape: (2715, 1433)
New edges shape: (2, 5478)
1/1 ━━━━━━━━━━━━━━━━━━━━ 3s 3s/step
Instance 1:
- Case_Based: 0.61%
- Genetic_Algorithms: 16.99%
- Neural_Networks: 2.71%
- Probabilistic_Methods: 1.59%
- Reinforcement_Learning: 2.08%
- Rule_Learning: 1.15%
- Theory: 74.88%
Instance 2:
- Case_Based: 0.03%
- Genetic_Algorithms: 96.73%
- Neural_Networks: 2.16%
- Probabilistic_Methods: 0.07%
- Reinforcement_Learning: 0.1%
- Rule_Learning: 0.05%
- Theory: 0.86%
```

5.3 Proposed GNN Node Classifier Architecture 2 (Hyperparameter Tuning)

```
def build_gnn_model(hp, graph_info, num_classes):
    hidden_units = [hp.Int(f'hidden_units_{i}', min_value=8, max_value=128, step=8) for i in range(2)]
    dropout_rate = hp.Float('dropout_rate', min_value=0.0, max_value=0.5, step=0.1)
    aggregation_type = hp.Choice('aggregation_type', values=['sum', 'mean', 'max'])
    combination_type = hp.Choice('combination_type', values=['concat', 'add'])
    normalize = hp.Boolean('normalize')

    model = ProposedGNNNodeClassifier(
        graph_info=graph_info,
        num_classes=num_classes,
        hidden_units=hidden_units,
        aggregation_type=aggregation_type,
        combination_type=combination_type,
        dropout_rate=dropout_rate,
        normalize=normalize,
    )

    model.compile(
        optimizer=tf.keras.optimizers.Adam(),
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=['accuracy']
    )

    return model
```

5.3.1 Random search using various different parameters and evaluated accuracy

```
tuner.search(
    x=x_train,
    y=y_train,
    epochs=100,  # You can adjust the number of epochs
    validation_data=(x_test, y_test)
)
```

```
Trial 15 Complete [00h 10m 45s]
val_accuracy: 0.7021593451499939

Best val_accuracy So Far: 0.7118391394615173
Total elapsed time: 01h 42m 21s
```

5.3.2 Building the model using best hyperparameters

```
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
early_stopping = keras.callbacks.EarlyStopping(
        monitor="val_accuracy", patience=50, restore_best_weights=True
    )
print(f"""
The hyperparameter search is complete.
The optimal number of hidden units is {[best_hps.get(f'hidden_units_{i}') for i in range(2)]}.
The optimal dropout rate is {best_hps.get('dropout_rate')}.
The optimal aggregation type is {best_hps.get('aggregation_type')}.
The optimal combination type is {best_hps.get('combination_type')}.
The optimal normalization setting is {best_hps.get('normalize')}.
""")

# Build the best model and train it
model = tuner.hypermodel.build(best_hps)
history = model.fit(
    x=x_train,
    y=y_train,
    epochs=300,
    validation_split = 0.2,
    validation_data=(x_test, y_test),
    callbacks = [early_stopping]
)
```

```
The hyperparameter search is complete.
The optimal number of hidden units is [88, 96].
The optimal dropout rate is 0.4.
The optimal aggregation type is sum.
The optimal combination type is add.
The optimal normalization setting is False.
```

5.4 Proposed GAT Node Classifier Architecture 1

```
class GraphAttentionNetwork(keras.Model):
    def __init__(
        self,
        node_states,
        edges,
        hidden_units,
        num_heads,
        num_layers,
        output_dim,
        **kwargs,
    ):
        super().__init__(**kwargs)
        self.node_states = node_states
        self.edges = edges
        self.preprocess = layers.Dense(hidden_units * num_heads, activation="relu")
        self.attention_layers = [
            MultiHeadGraphAttention(hidden_units, num_heads) for _ in range(num_layers)
        ]
        self.output_layer = layers.Dense(output_dim)

    def call(self, inputs):
        node_states, edges = inputs
        x = self.preprocess(node_states)
        for attention_layer in self.attention_layers:
            x = attention_layer([x, edges]) + x
        outputs = self.output_layer(x)
        return outputs

    def train_step(self, data):
        indices, labels = data

        with tf.GradientTape() as tape:
            # Forward pass
            outputs = self([self.node_states, self.edges])
            # Compute loss
            loss = self.compiled_loss(labels, tf.gather(outputs, indices))
        # Compute gradients
        grads = tape.gradient(loss, self.trainable_weights)
        # Apply gradients (update weights)
        optimizer.apply_gradients(zip(grads, self.trainable_weights))
        # Update metric(s)
        self.compiled_metrics.update_state(labels, tf.gather(outputs, indices))

        return {m.name: m.result() for m in self.metrics}

    def predict_step(self, data):
        indices = data
        # Forward pass
        outputs = self([self.node_states, self.edges])
        # Compute probabilities
        return tf.nn.softmax(tf.gather(outputs, indices))

    def test_step(self, data):
        indices, labels = data
        # Forward pass
        outputs = self([self.node_states, self.edges])
        # Compute loss
        loss = self.compiled_loss(labels, tf.gather(outputs, indices))
        # Update metric(s)
        self.compiled_metrics.update_state(labels, tf.gather(outputs, indices))

        return {m.name: m.result() for m in self.metrics}
```

5.4.1 Defining the hyperparameters for proposing the model

```
HIDDEN_UNITS = 128
NUM_HEADS = 8
NUM_LAYERS = 16
OUTPUT_DIM = len(class_values)

NUM_EPOCHS = 100
BATCH_SIZE = 32
VALIDATION_SPLIT = 0.15
LEARNING_RATE = 0.01
MOMENTUM = 0.9
```

```
loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = keras.optimizers.SGD(LEARNING_RATE, momentum=MOMENTUM)
accuracy_fn = keras.metrics.SparseCategoricalAccuracy(name="acc")
early_stopping = keras.callbacks.EarlyStopping(
    monitor="val_acc", min_delta=1e-5, patience=5, restore_best_weights=True
)
```

Proposed Graph Attention Network (GAT) 1

```
proposed_gat_model = GraphAttentionNetwork(
    node_states, edges, 64, NUM_HEADS, 8, OUTPUT_DIM
)
```

5.4.2 Training the model for 50 epochs and evaluating the accuracy

```
proposed_gat_model.compile(loss=loss_fn, optimizer=optimizer, metrics=[accuracy_fn])

history1 = proposed_gat_model.fit(
    x=train_indices,
    y=train_labels,
    validation_split=0.20,
    batch_size=128,
    epochs=50,
    callbacks=[early_stopping],
    verbose=2,
)

_, test_accuracy = proposed_gat_model.evaluate(x=test_indices, y=test_labels, verbose=0)

Architectur1 = proposed_gat_model.save("Architecture1.keras")
print("--" * 38 + f"\nTest Accuracy {test_accuracy*100:.1f}%")
```

```
Epoch 1/50
9/9 - 61s - 7s/step - acc: 0.1505 - loss: 0.1480 - val_acc: 0.2952 - val_loss: -2.9704e-02
Epoch 2/50
9/9 - 5s - 543ms/step - acc: 0.2733 - loss: -4.4616e-02 - val_acc: 0.3469 - val_loss: -6.2718e-02
Epoch 3/50
9/9 - 5s - 567ms/step - acc: 0.3416 - loss: -5.3086e-02 - val_acc: 0.4465 - val_loss: -5.1314e-02
Epoch 4/50
9/9 - 5s - 552ms/step - acc: 0.5365 - loss: -4.1785e-02 - val_acc: 0.3875 - val_loss: -4.6229e-02
Epoch 5/50
9/9 - 5s - 562ms/step - acc: 0.4451 - loss: -4.0861e-02 - val_acc: 0.4908 - val_loss: -5.1943e-02
Epoch 6/50
9/9 - 5s - 548ms/step - acc: 0.6223 - loss: -4.5263e-02 - val_acc: 0.5092 - val_loss: -5.1746e-02
```

```
Epoch 9/50
9/9 - 5s - 547ms/step - acc: 0.7027 - loss: -5.7091e-02 - val_acc: 0.6125 - val_loss: -6.7450e-02
Epoch 10/50
9/9 - 5s - 543ms/step - acc: 0.7350 - loss: -5.9200e-02 - val_acc: 0.6162 - val_loss: -6.8360e-02
Epoch 11/50
9/9 - 5s - 555ms/step - acc: 0.7590 - loss: -6.3579e-02 - val_acc: 0.6458 - val_loss: -7.6095e-02
Epoch 12/50
9/9 - 5s - 568ms/step - acc: 0.7710 - loss: -6.8950e-02 - val_acc: 0.6642 - val_loss: -7.1824e-02
Epoch 13/50
9/9 - 6s - 629ms/step - acc: 0.7886 - loss: -6.5241e-02 - val_acc: 0.6827 - val_loss: -8.1194e-02
Epoch 14/50
9/9 - 5s - 578ms/step - acc: 0.8070 - loss: -7.1984e-02 - val_acc: 0.6937 - val_loss: -7.5191e-02
Epoch 15/50
9/9 - 5s - 564ms/step - acc: 0.8246 - loss: -6.8213e-02 - val_acc: 0.7085 - val_loss: -8.1961e-02
Epoch 16/50
9/9 - 5s - 548ms/step - acc: 0.8458 - loss: -7.5021e-02 - val_acc: 0.7306 - val_loss: -8.2476e-02
Epoch 17/50
9/9 - 5s - 534ms/step - acc: 0.8587 - loss: -7.3587e-02 - val_acc: 0.7454 - val_loss: -8.3510e-02
Epoch 18/50
9/9 - 5s - 537ms/step - acc: 0.8735 - loss: -7.6461e-02 - val_acc: 0.7638 - val_loss: -8.4784e-02
Epoch 19/50
9/9 - 5s - 540ms/step - acc: 0.8827 - loss: -7.6195e-02 - val_acc: 0.7638 - val_loss: -8.4729e-02
Epoch 20/50
9/9 - 5s - 531ms/step - acc: 0.8929 - loss: -7.9091e-02 - val_acc: 0.7749 - val_loss: -8.7220e-02
Epoch 21/50
9/9 - 5s - 535ms/step - acc: 0.9012 - loss: -7.6614e-02 - val_acc: 0.7823 - val_loss: -9.2666e-02
Epoch 22/50
9/9 - 5s - 530ms/step - acc: 0.9058 - loss: -8.6179e-02 - val_acc: 0.7823 - val_loss: -9.0491e-02
Epoch 23/50
...
Epoch 32/50
9/9 - 5s - 537ms/step - acc: 0.9584 - loss: -7.9119e-02 - val_acc: 0.7860 - val_loss: -9.0686e-02
--------------------------------------------------------------------------
Test Accuracy 73.8%
```

5.4.3 Getting the model summary

odel: "graph_attention_network_2"

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(2708, 512)	734,208
multi_head_graph_attention_6 (MultiHeadGraphAttention)	?	263,168
multi_head_graph_attention_7 (MultiHeadGraphAttention)	?	263,168
multi_head_graph_attention_8 (MultiHeadGraphAttention)	?	263,168
multi_head_graph_attention_9 (MultiHeadGraphAttention)	?	263,168
multi_head_graph_attention_10 (MultiHeadGraphAttention)	?	263,168
multi_head_graph_attention_11 (MultiHeadGraphAttention)	?	263,168
multi_head_graph_attention_12 (MultiHeadGraphAttention)	?	263,168
multi_head_graph_attention_13 (MultiHeadGraphAttention)	?	263,168
dense_5 (Dense)	(2708, 7)	3,591

Total params: 5,686,288 (21.69 MB)

5.4.4 Prediction of randomly generated instances

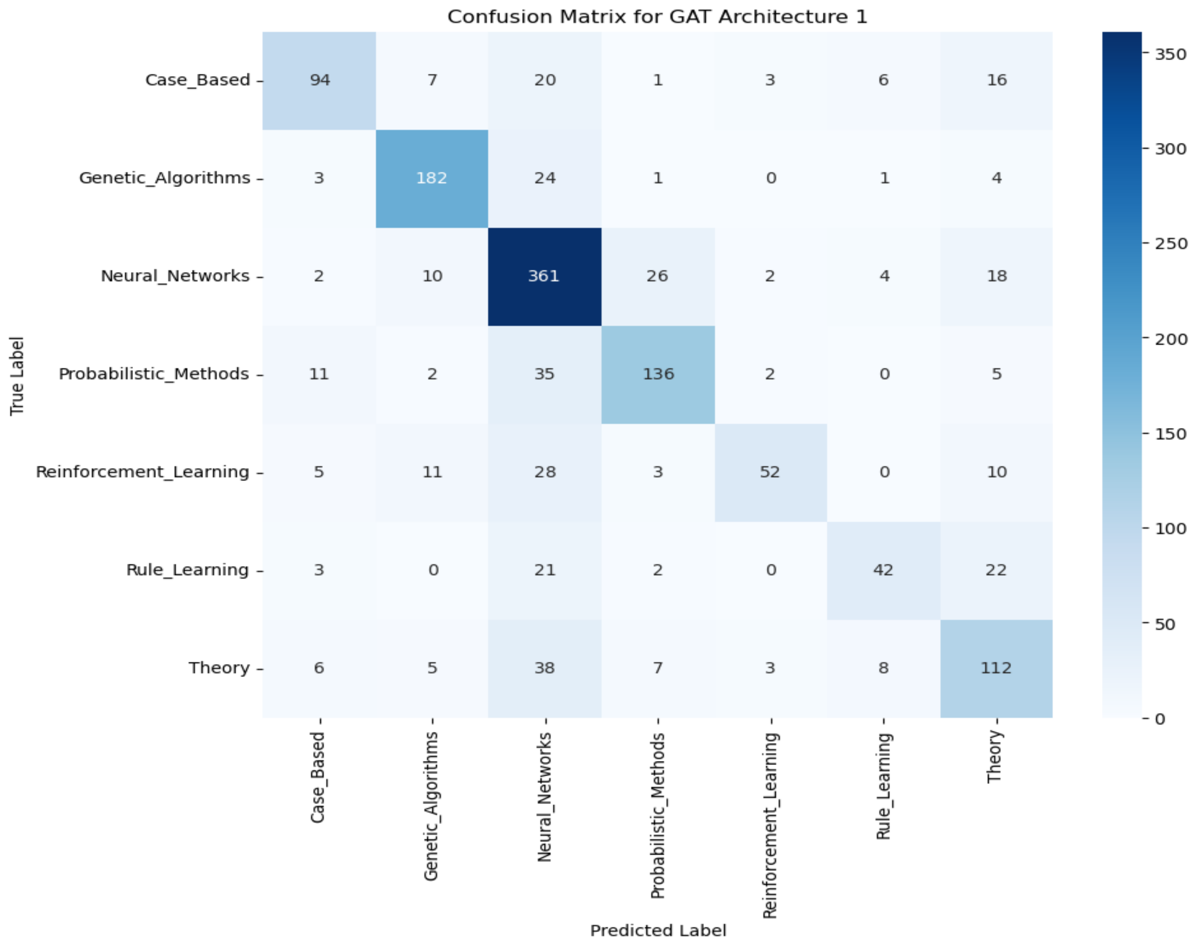
```
test_probs = proposed_gat_model.predict(x=test_indices)

mapping = {v: k for (k, v) in class_idx.items()}

for i, (probs, label) in enumerate(zip(test_probs[:10], test_labels[:10])):
    print(f"Instance {i+1}: {mapping[label]}")
    for j, c in zip(probs, class_idx.keys()):
        print(f"\tProbability of {c: <24} = {j*100:7.3f}%")
    print("---" * 20)
```

```
43/43 ━━━━━━━━━━━━━━━━━━━━ 15s 231ms/step
Instance 1: Probabilistic_Methods
        Probability of Case_Based               =   7.973%
        Probability of Genetic_Algorithms       =   4.577%
        Probability of Neural_Networks          =  55.207%
        Probability of Probabilistic_Methods    =  18.522%
        Probability of Reinforcement_Learning   =   4.390%
        Probability of Rule_Learning            =   1.621%
        Probability of Theory                   =   7.710%
------------------------------------------------------------
Instance 2: Genetic_Algorithms
        Probability of Case_Based               =   0.739%
        Probability of Genetic_Algorithms       =  98.151%
        Probability of Neural_Networks          =   0.003%
        Probability of Probabilistic_Methods    =   0.025%
        Probability of Reinforcement_Learning   =   0.331%
        Probability of Rule_Learning            =   0.673%
        Probability of Theory                   =   0.079%
```

5.4.5 Confusion Matrix of GAT Node Classifier 1



5.5 Proposed GAT Node Classifier Architecture 2

5.5.1 Defining the hyperparameters and training the model

```
loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
optimizer = keras.optimizers.SGD(LEARNING_RATE, momentum=MOMENTUM)
accuracy_fn = keras.metrics.SparseCategoricalAccuracy(name="acc")
early_stopping = keras.callbacks.EarlyStopping(
    monitor="val_acc", min_delta=1e-5, patience=5, restore_best_weights=True
)

proposed_gat_model2 = GraphAttentionNetwork(
    node_states, edges, 100, NUM_HEADS, 6, OUTPUT_DIM
)

proposed_gat_model2.compile(loss=loss_fn, optimizer=optimizer, metrics=[accuracy_fn])
history2= proposed_gat_model2.fit(
    x=train_indices,
    y=train_labels,
    validation_split=0.20,
    batch_size=128,
    epochs=50,
    callbacks=[early_stopping],
    verbose=2,
)


_, test_accuracy = proposed_gat_model2.evaluate(x=test_indices, y=test_labels, verbose=0)
Architectur2= proposed_gat_model2.save("Architectur2.keras")
print("--" * 38 + f"\nTest Accuracy {test_accuracy*100:.1f}%")
```

```
Epoch 1/50
9/9 - 41s - 5s/step - acc: 0.2392 - loss: -4.6040e-01 - val_acc: 0.3284 - val_loss: -4.5301e-01
```

5.5.2 Evaluation of Accuracy on test data

```
Epoch 10/50
9/9 - 7s - 782ms/step - acc: 0.7729 - loss: -5.2639e-01 - val_acc: 0.6863 - val_loss: -5.4391e-01
Epoch 11/50
9/9 - 10s - 1s/step - acc: 0.8033 - loss: -5.3761e-01 - val_acc: 0.7196 - val_loss: -5.6086e-01
Epoch 12/50
9/9 - 7s - 761ms/step - acc: 0.8255 - loss: -5.5878e-01 - val_acc: 0.7196 - val_loss: -5.7813e-01
Epoch 13/50
9/9 - 7s - 746ms/step - acc: 0.8440 - loss: -5.6832e-01 - val_acc: 0.7306 - val_loss: -5.8651e-01
Epoch 14/50
9/9 - 7s - 765ms/step - acc: 0.8495 - loss: -5.8499e-01 - val_acc: 0.7528 - val_loss: -6.0182e-01
Epoch 15/50
9/9 - 7s - 785ms/step - acc: 0.8744 - loss: -5.9753e-01 - val_acc: 0.7601 - val_loss: -6.1517e-01
Epoch 16/50
9/9 - 7s - 772ms/step - acc: 0.8827 - loss: -6.1134e-01 - val_acc: 0.7565 - val_loss: -6.2531e-01
Epoch 17/50
9/9 - 7s - 765ms/step - acc: 0.8920 - loss: -6.2304e-01 - val_acc: 0.7638 - val_loss: -6.3742e-01
Epoch 18/50
9/9 - 7s - 793ms/step - acc: 0.9040 - loss: -6.3177e-01 - val_acc: 0.7565 - val_loss: -6.5226e-01
Epoch 19/50
9/9 - 7s - 774ms/step - acc: 0.9077 - loss: -6.5390e-01 - val_acc: 0.7601 - val_loss: -6.6340e-01
Epoch 20/50
9/9 - 7s - 769ms/step - acc: 0.9187 - loss: -6.5619e-01 - val_acc: 0.7749 - val_loss: -6.7870e-01
Epoch 21/50
9/9 - 7s - 763ms/step - acc: 0.9215 - loss: -6.7070e-01 - val_acc: 0.7712 - val_loss: -6.7355e-01
Epoch 22/50
9/9 - 7s - 779ms/step - acc: 0.9298 - loss: -6.6926e-01 - val_acc: 0.7860 - val_loss: -6.8190e-01
Epoch 23/50
...
Epoch 30/50
9/9 - 7s - 770ms/step - acc: 0.9594 - loss: -7.2480e-01 - val_acc: 0.7860 - val_loss: -7.3934e-01
----------------------------------------------------------------------------
Test Accuracy 73.0%
```

5.5.3 Getting the summary of the model

```
proposed_gat_model2.summary()
```

```
Model: "graph_attention_network_3"
```

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(2708, 800)	1,147,200
multi_head_graph_attention_14 (MultiHeadGraphAttention)	?	641,600
multi_head_graph_attention_15 (MultiHeadGraphAttention)	?	641,600
multi_head_graph_attention_16 (MultiHeadGraphAttention)	?	641,600
multi_head_graph_attention_17 (MultiHeadGraphAttention)	?	641,600
multi_head_graph_attention_18 (MultiHeadGraphAttention)	?	641,600
multi_head_graph_attention_19 (MultiHeadGraphAttention)	?	641,600
dense_7 (Dense)	(2708, 7)	5,607

```
Total params: 10,004,816 (38.17 MB)

Trainable params: 5,002,407 (19.08 MB)
```

5.5.4 Prediction of randomly generated instances

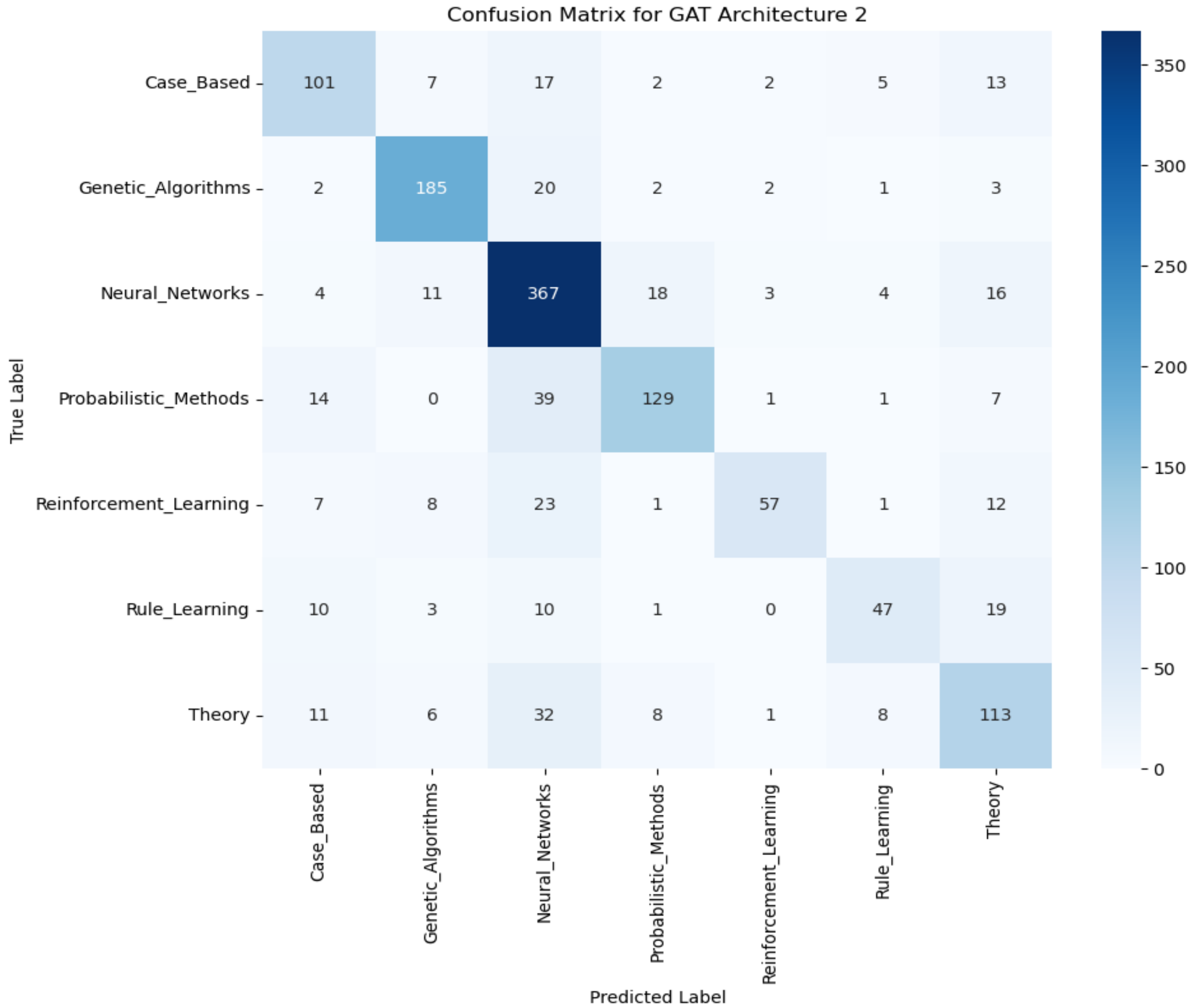
```
test_probs2 = proposed_gat_model2.predict(x=test_indices)

mapping = {v: k for (k, v) in class_idx.items()}

for i, (probs, label) in enumerate(zip(test_probs[:10], test_labels[:10])):
    print(f"Instance {i+1}: {mapping[label]}")
    for j, c in zip(probs, class_idx.keys()):
        print(f"\tProbability of {c: <24} = {j*100:7.3f}%")
    print("---" * 20)
```

```
43/43 ━━━━━━━━━━━━━━━━━━━━ 40s 491ms/step
Instance 1: Probabilistic_Methods
	Probability of Case_Based               =   7.973%
	Probability of Genetic_Algorithms       =   4.577%
	Probability of Neural_Networks          =  55.207%
	Probability of Probabilistic_Methods    =  18.522%
	Probability of Reinforcement_Learning   =   4.390%
	Probability of Rule_Learning            =   1.621%
	Probability of Theory                   =   7.710%
------------------------------------------------------------
Instance 2: Genetic_Algorithms
	Probability of Case_Based               =   0.739%
	Probability of Genetic_Algorithms       =  98.151%
	Probability of Neural_Networks          =   0.003%
	Probability of Probabilistic_Methods    =   0.025%
	Probability of Reinforcement_Learning   =   0.331%
	Probability of Rule_Learning            =   0.673%
	Probability of Theory                   =   0.079%
------------------------------------------------------------
```

5.5.5 Confusion Matrix of GAT Node Classifier 2



References

1. Bhargava, Y., 2024. Improving Node Classification in Term-Document Matrices Using Advanced Graph Neural Networks**.** MSc Research Project. National College of Ireland.

2. Sharma, P. (2024, June 25). *What are Graph Neural Networks, and how do they work?* Analytics Vidhya. https://www.analyticsvidhya.com/blog/2022/03/what-are-graph-neural-networks-and-how-do-they-work/#:~:text=Graph%20Neural%20Networks%20are%20topologies,edge%20prediction%2C%20and%20so%20on.