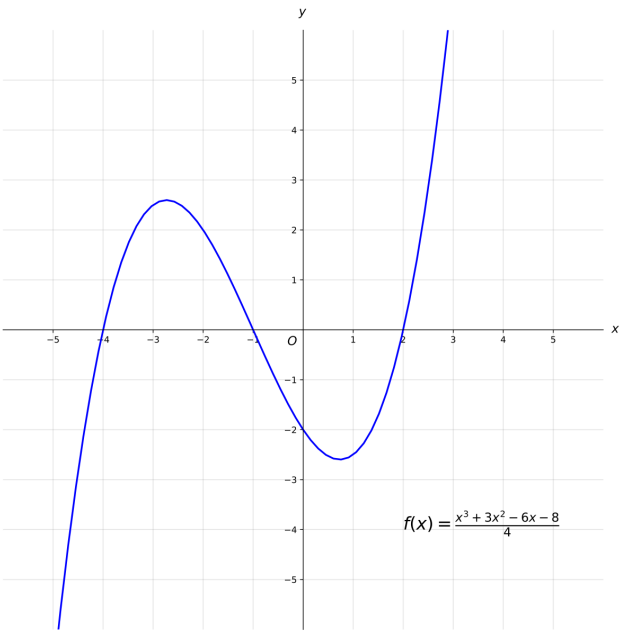
Report

Sirigiri Sai Keerthan-IMT2020511
Sougandh Krishna-IMT2020120

What is a graph?

A graph is a mathematical representation of relationships between a set of entities.



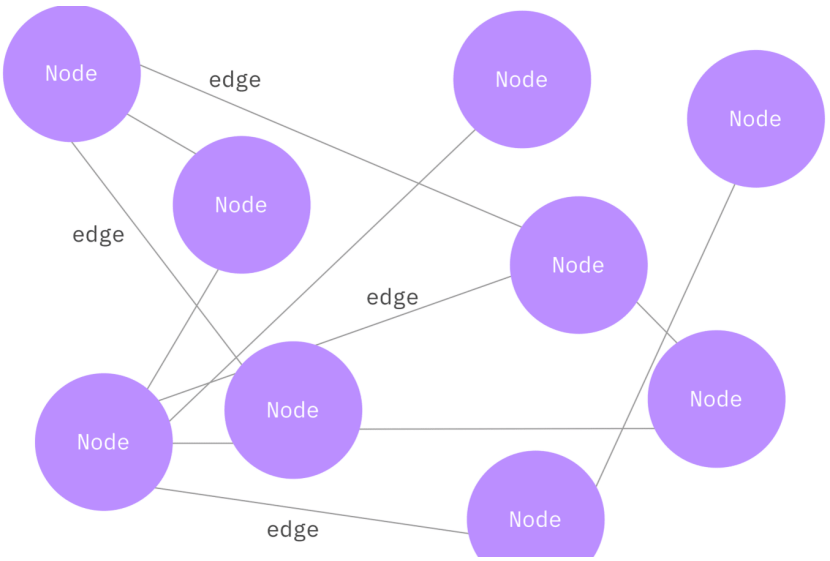
Node (or Vertex):

- A node represents an individual entity in a graph.Each node is typically assigned a unique identifier.
- Nodes can have associated attributes or features. In the context of GNNs, these features could be numerical values, categorical labels, or other relevant information about the entity represented by the node.

- In a social network graph, each node could represent an individual user. The node might have attributes such as age, gender, and interests

Edge:

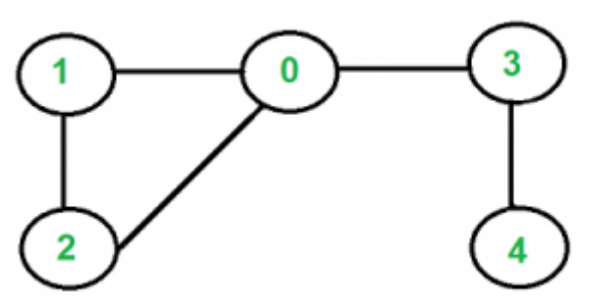
- An edge is a fundamental link or connection between two vertices in a graph. It represents a relationship, interaction, or association between the entities represented by the connected vertices.
- Similar to vertices, edges can also have associated attributes or weights. These attributes provide additional information about the relationship represented by the edge.
- For example, in a social network graph, an edge might have a weight indicating the strength of the friendship between two individuals.



Types of Graphs:

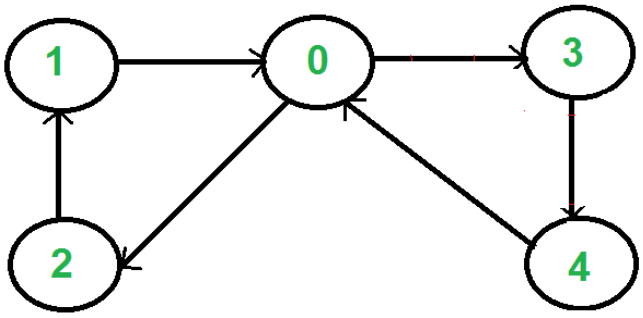
1.Undirected Graph:

In an undirected graph, edges have no direction, meaning the relationship between two vertices is mutual. If there is an edge between vertices *u* and *v*, it implies a connection between *u* and *v* and vice versa.



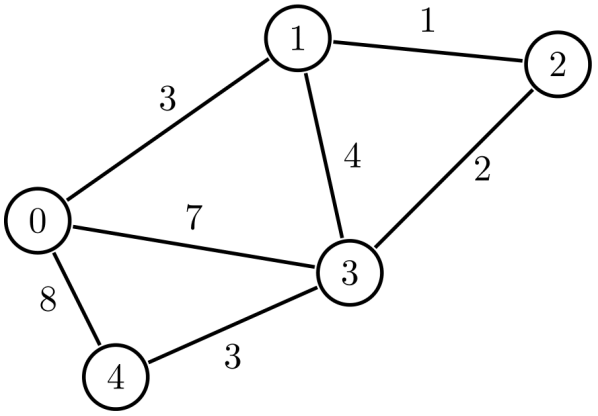
2.Directed Graph:

In a directed graph, edges have a specific direction. An edge from vertex *u* to vertex *v* indicates a one-way relationship from *u* to *v*. The reverse relationship from *v* to *u* may not be present unless explicitly specified.



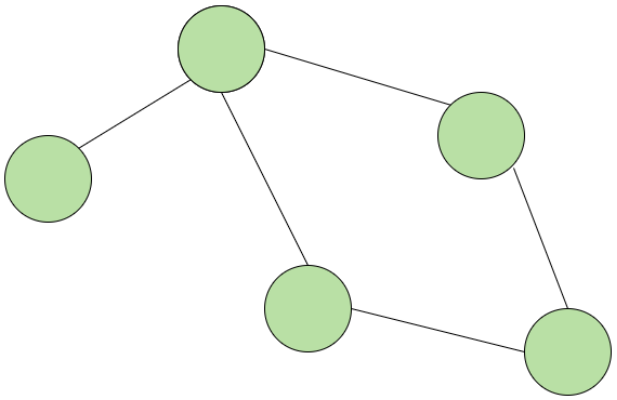
3.Weighted Graph:

In a weighted graph, edges have associated numerical values called weights. These weights can represent distances, costs, or any other quantitative measure associated with the relationship between vertices.



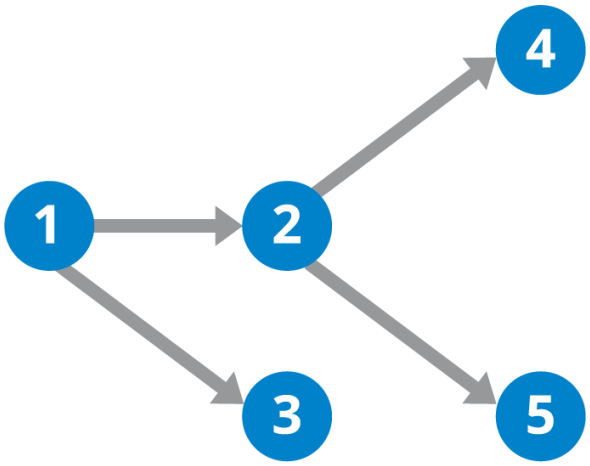
4.Cyclic Graphs:

A graph that contains cycles (loops) is called cyclic.



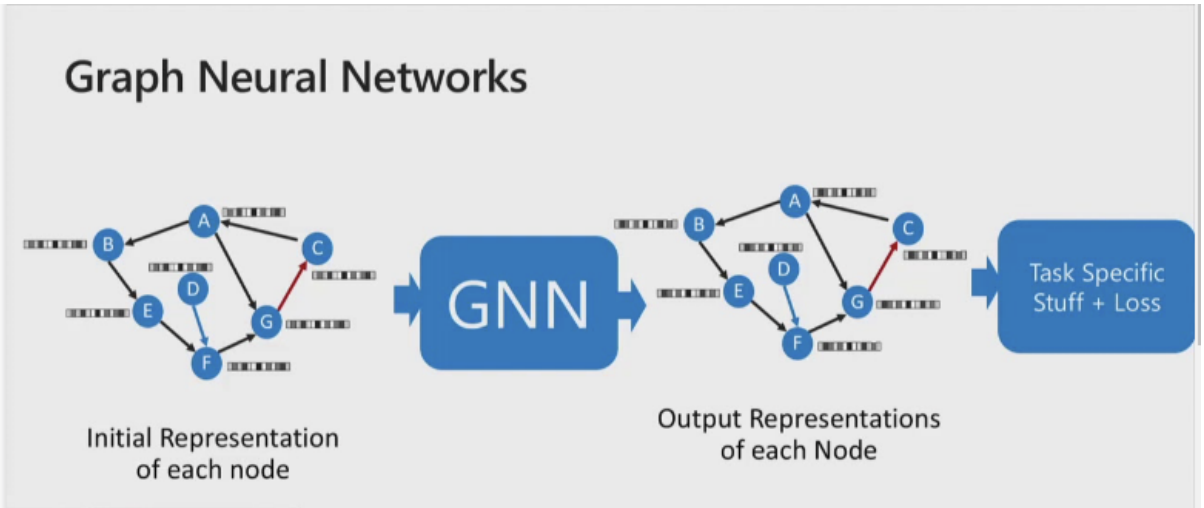
5 Acyclic Graphs

A graph without cycles is acyclic.



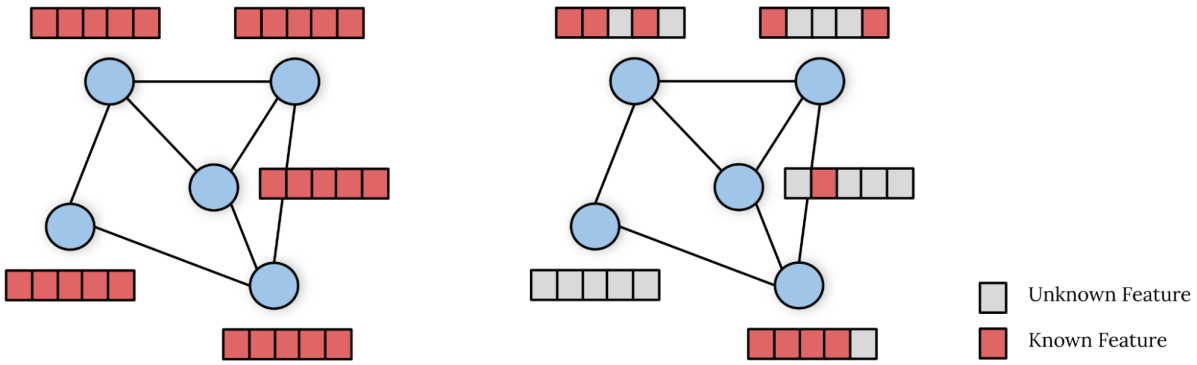
Graph Neural Networks

GNN stands for Graph Neural Network. It is a type of neural network designed to work with data structured as graphs. Graphs are mathematical structures that consist of nodes and edges. GNNs are particularly useful for tasks involving graph-structured data, such as social networks, recommendation systems, molecular chemistry, and more.



Key components of GNN

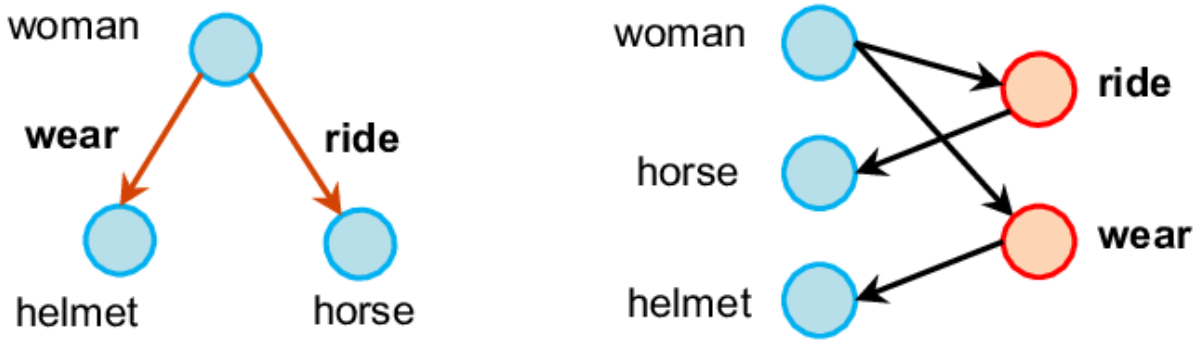
1. **Node Features**: Each node in the graph has associated feature vectors. These features can represent various attributes, such as user profiles in a social network or atom types in a chemical compound.



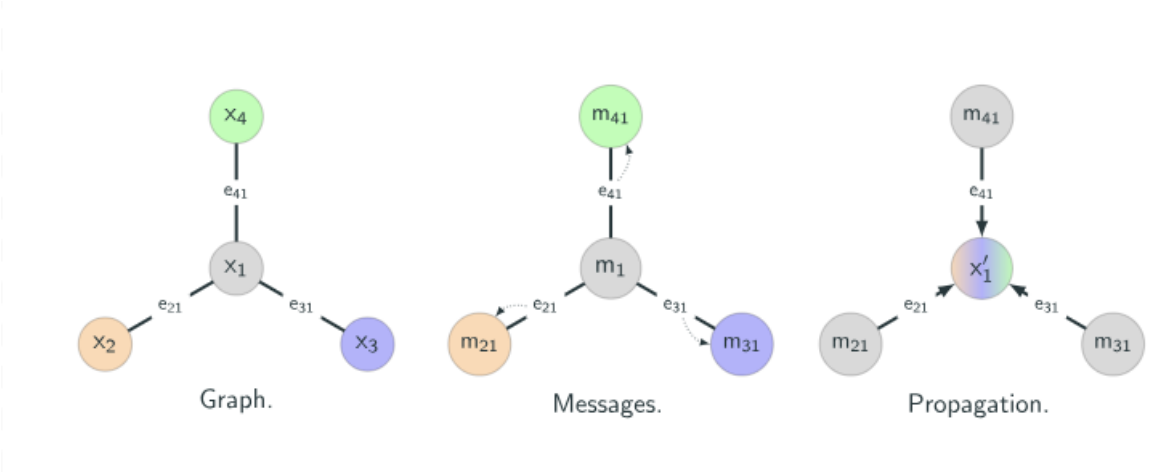
Graph with all features observed

Graph with partially missing features

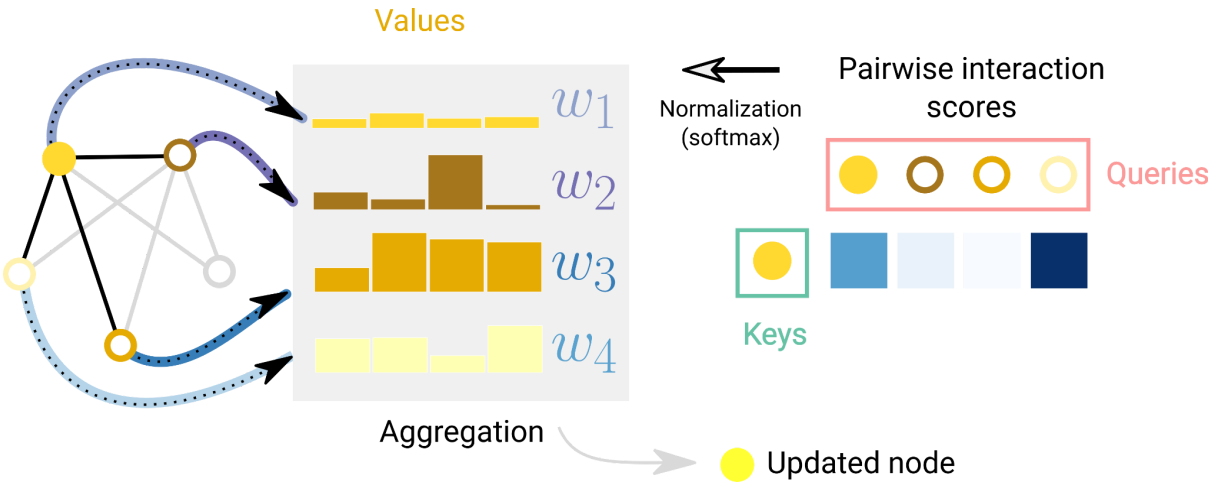
2. **Graph Structure**: The edges between nodes define the graph's structure. They represent relationships or connections between nodes and can be directed or undirected, weighted, or unweighted.



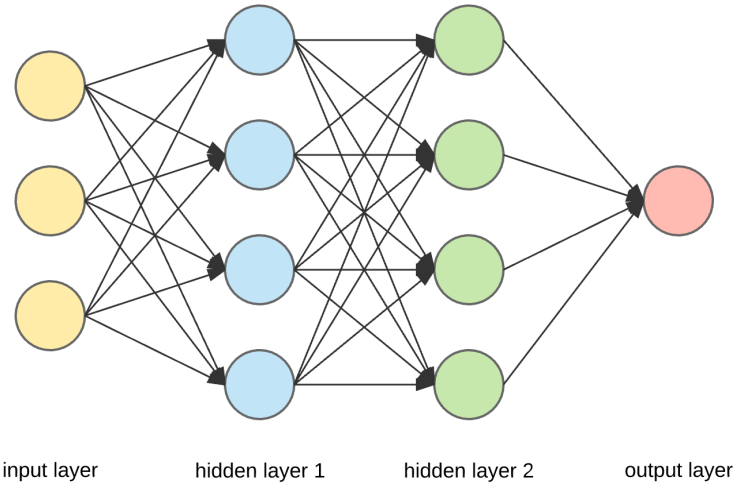
3. **Message Passing**: The core operation in GNNs is message passing, where information is exchanged between neighbouring nodes in the graph. This allows nodes to aggregate and update their features based on their connections.



4. **Aggregation Function**: An aggregation function combines the messages received from neighbours to compute an updated representation for each node. Common aggregation functions include mean, max-pooling, or weighted combinations.

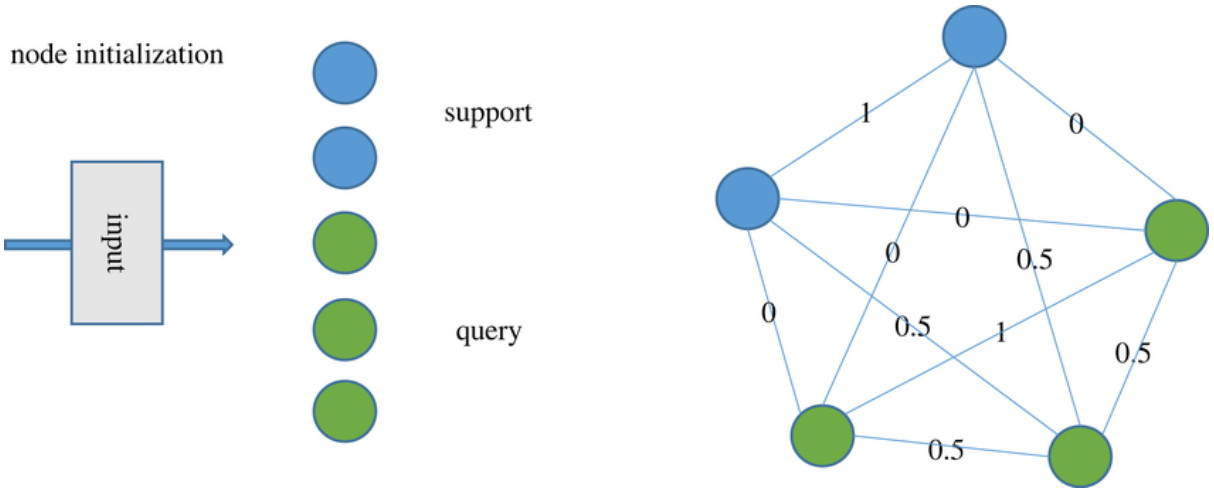


5. **Neural Network Layers**:GNNs typically incorporate neural network layers to refine node features.

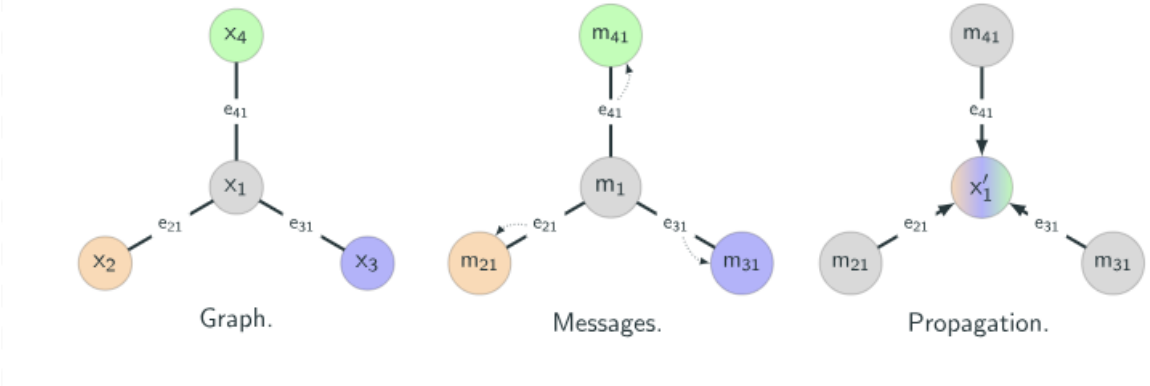


Working

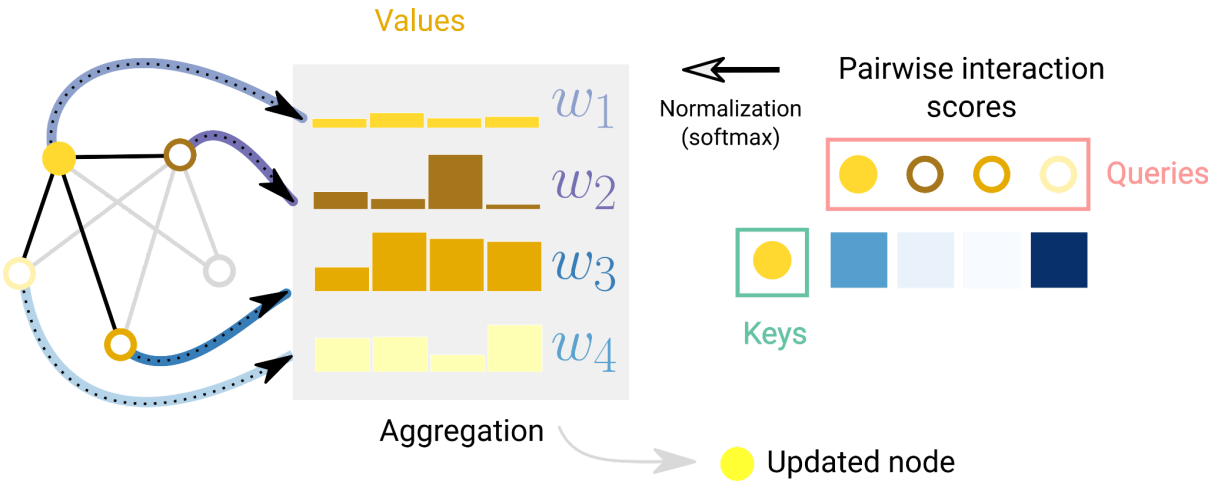
1. **Initialization**: Assign initial feature vectors to each node based on the problem domain.



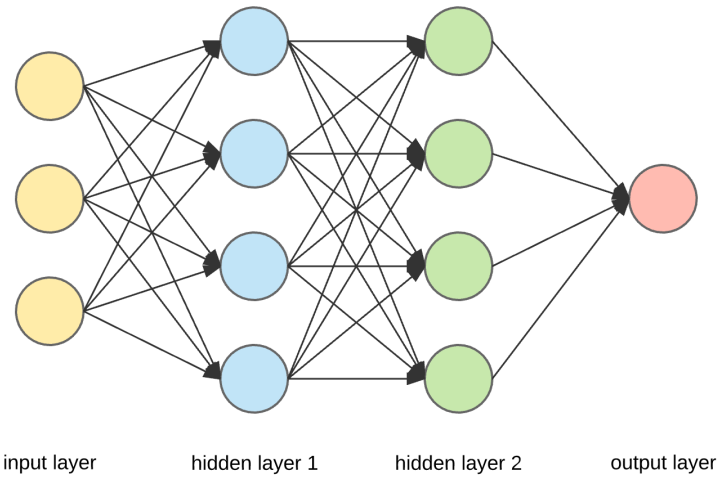
2. **Message Passing**: Iteratively, nodes exchange messages with their neighbours, aggregating information from their connections.



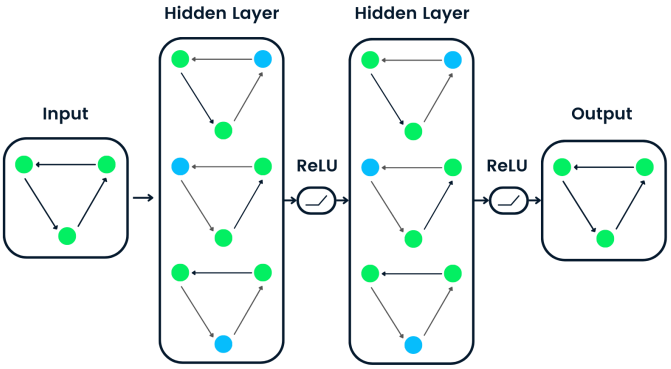
3. **Aggregation and Update**: The aggregated messages are combined using an aggregation function, and the node's feature vector is updated.



4. **Neural Network Layers**: Optionally, GNNs can include neural network layers for further refinement of node features.

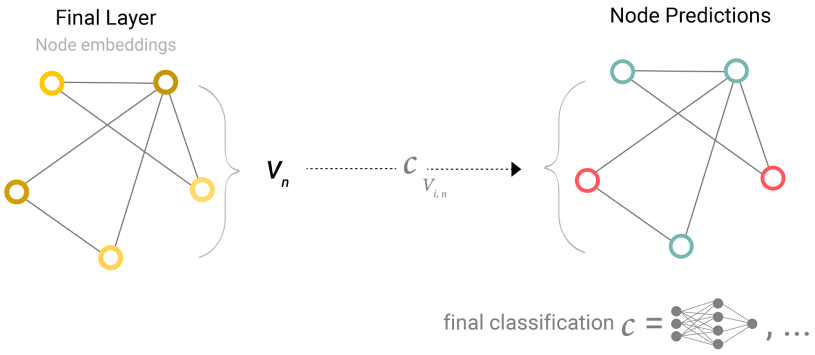


5. **Output**: The final node representations can be used for various downstream tasks, such as node classification, graph classification, or link prediction.



How do we make predictions in any of the tasks?

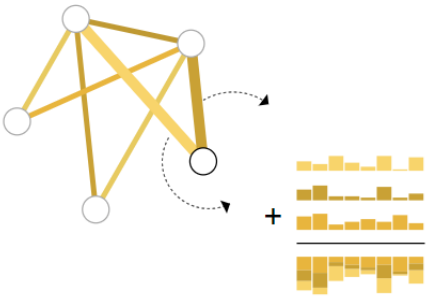
We will consider the case of binary classification, but this framework can easily be extended to the multi-class or regression case. If the task is to make binary predictions on nodes, and the graph already contains node information, the approach is straightforward — for each node embedding, apply a linear classifier.



However, it is not always so simple. For instance, you might have information in the graph stored in edges, but no information in nodes, but still need to make predictions on nodes. We need a way to collect information from edges and give them to nodes for prediction. We can do this by *pooling*. Pooling proceeds in two steps:

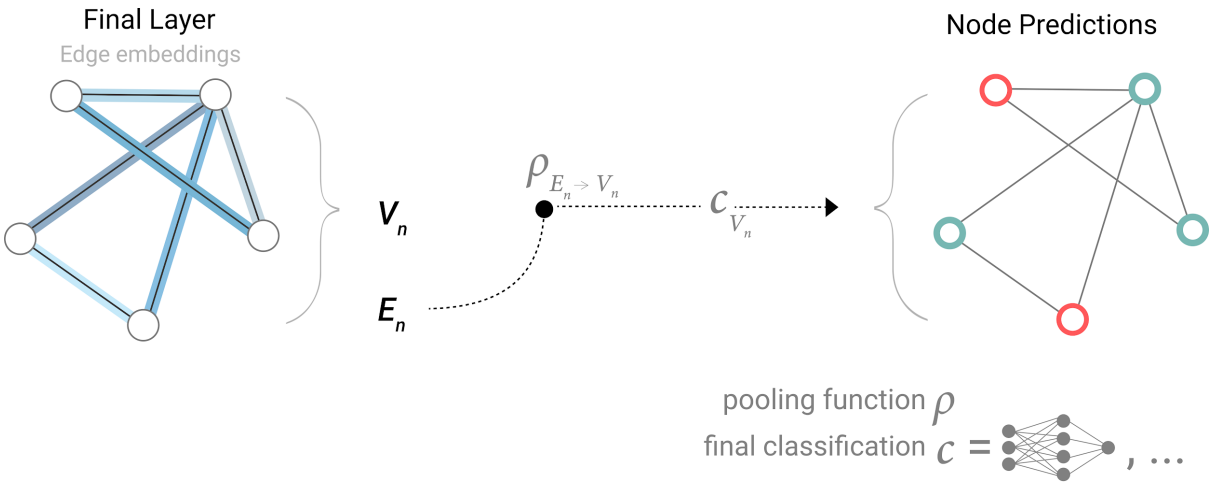
1. For each item to be pooled, *gather* each of their embeddings and concatenate them into a matrix.
2. The gathered embeddings are then *aggregated*, usually via a sum operation.

We represent the *pooling* operation by the letter ρ, and denote that we are gathering information from edges to nodes as $p_{E_n \to V_n}$.

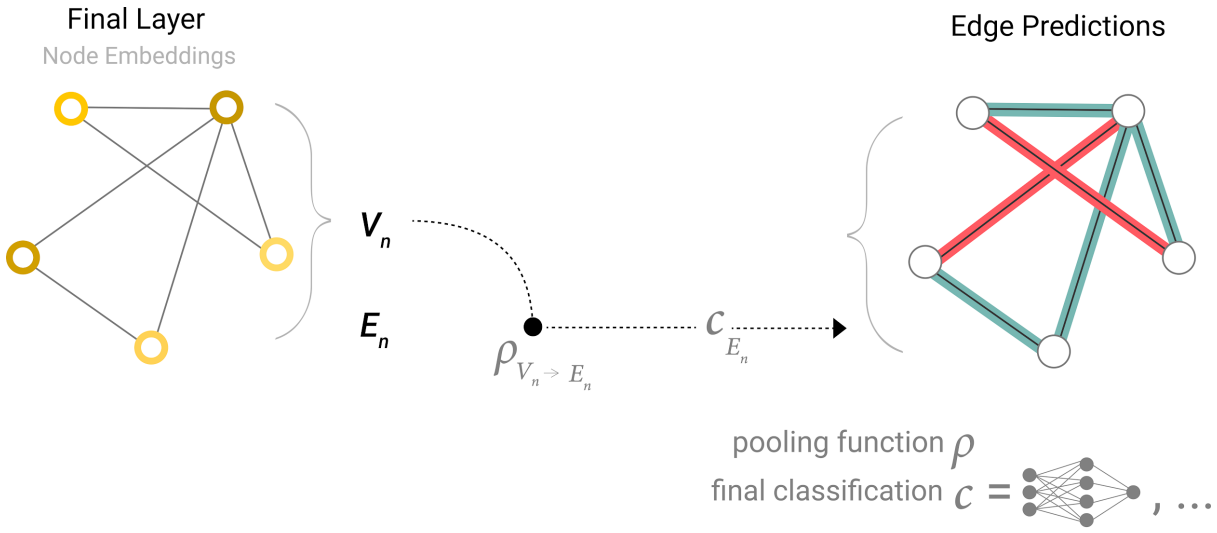


Aggregate information
from adjacent edges

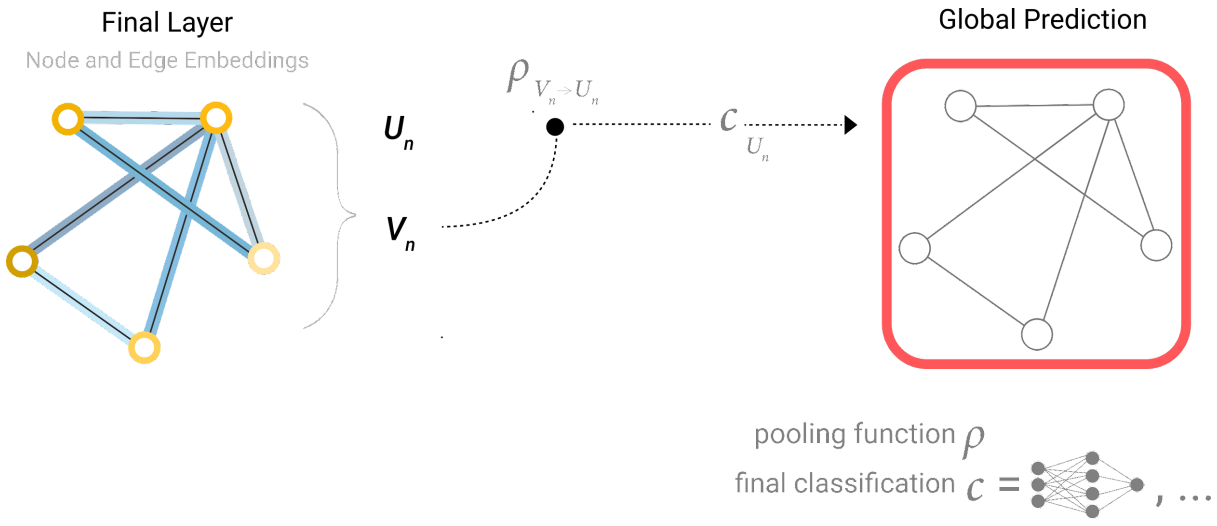
So If we only have edge-level features, and are trying to predict binary node information, we can use pooling to route (or pass) information to where it needs to go. The model looks like this.



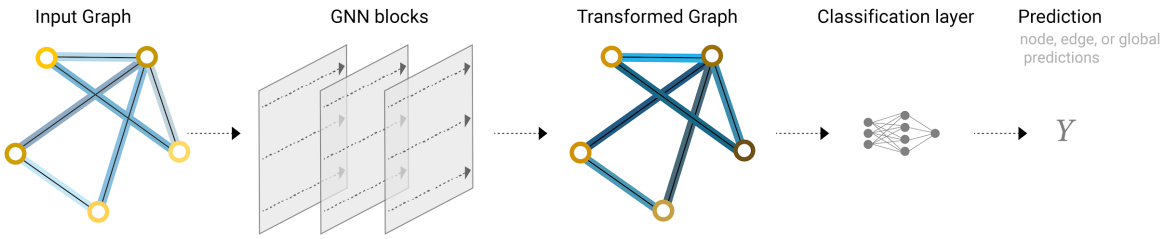
If we only have node-level features, and are trying to predict binary edge-level information, the model looks like this.



If we only have node-level features, and need to predict a binary global property, we need to gather all available node information together and aggregate them. This is similar to *Global Average Pooling* layers in CNNs. The same can be done for edges.



In our examples, the classification model “C” can easily be replaced with any differentiable model, or adapted to multi-class classification using a generalised linear model.



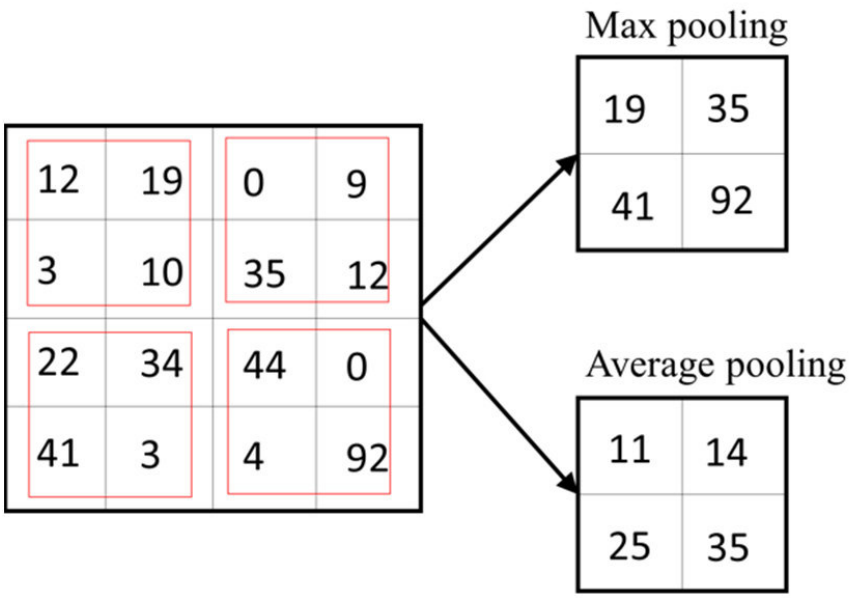
Types of Pooling:

1.Average Pooling

Average pooling computes the average of the elements present in the region of the feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.

2.Max pooling:

Max Pooling is a pooling operation that calculates the maximum value for patches of a feature map, and uses it to create a downsampled (pooled) feature map.

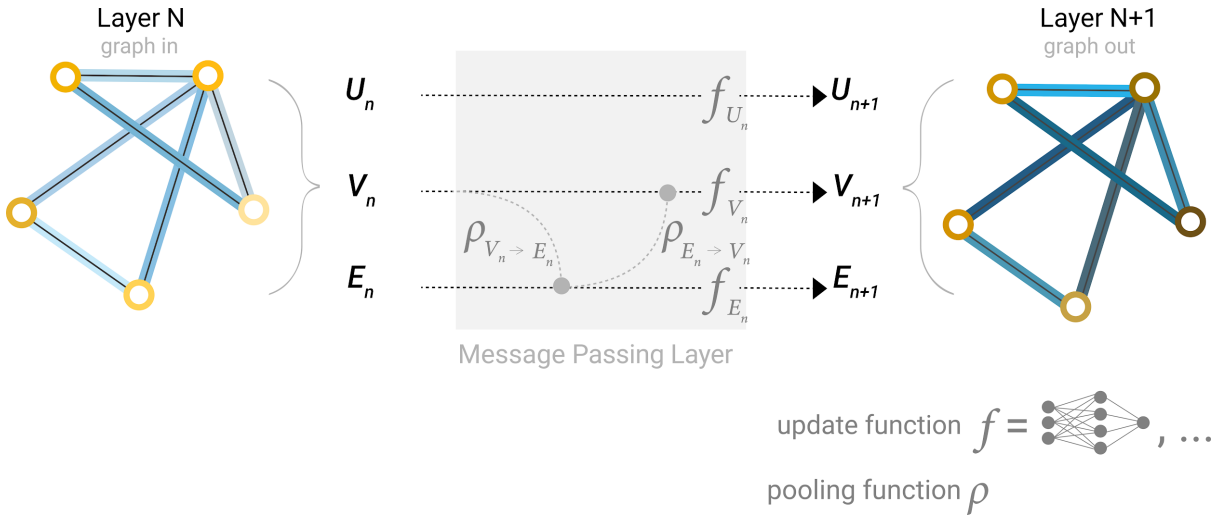


Learning edge representations

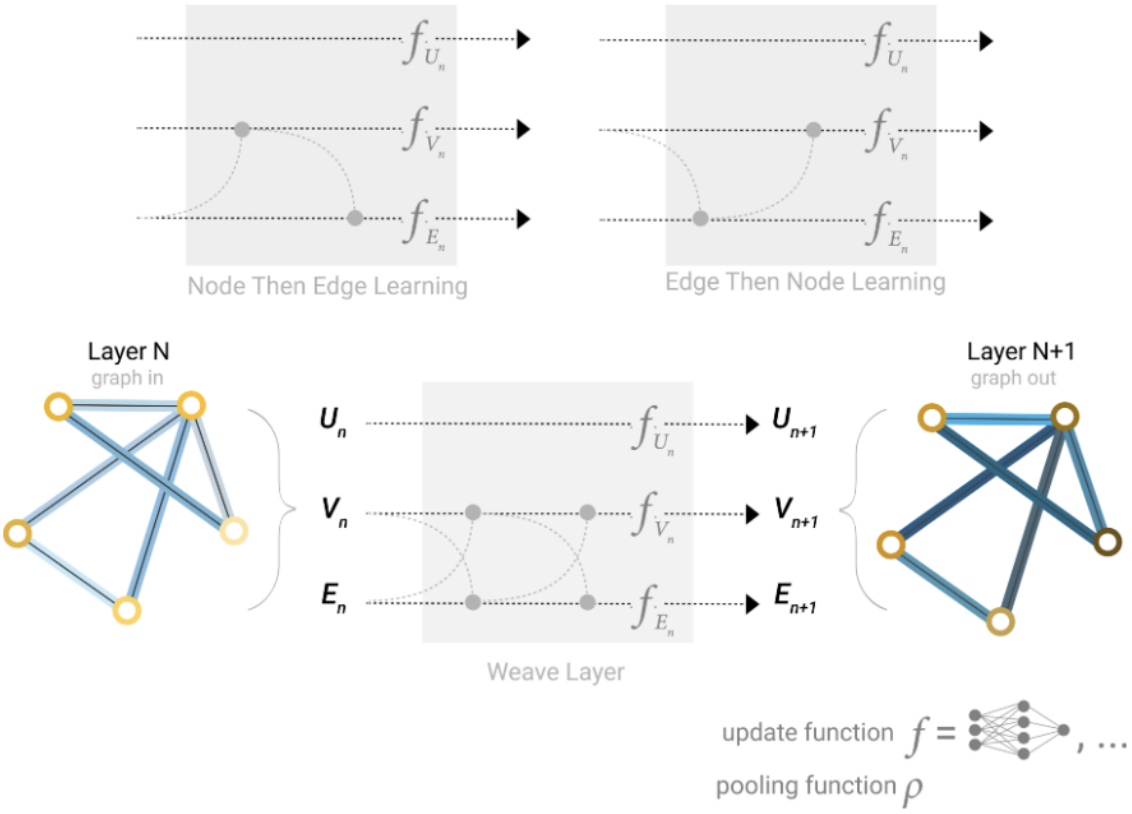
Our dataset does not always contain all types of information (node, edge, and global context). When we want to make a prediction on nodes, but our dataset only has edge information, we showed above how to use pooling to route information from edges to nodes, but only at the final prediction step of the model. We can share information between nodes and edges within the GNN layer using message passing.

We can incorporate the information from neighbouring edges in the same way we used neighbouring node information earlier, by first pooling the edge information, transforming it with an update function, and storing it.

However, the node and edge information stored in a graph are not necessarily the same size or shape, so it is not immediately clear how to combine them. One way is to learn a linear mapping from the space of edges to the space of nodes, and vice versa. Alternatively, one may concatenate them together before the update function.



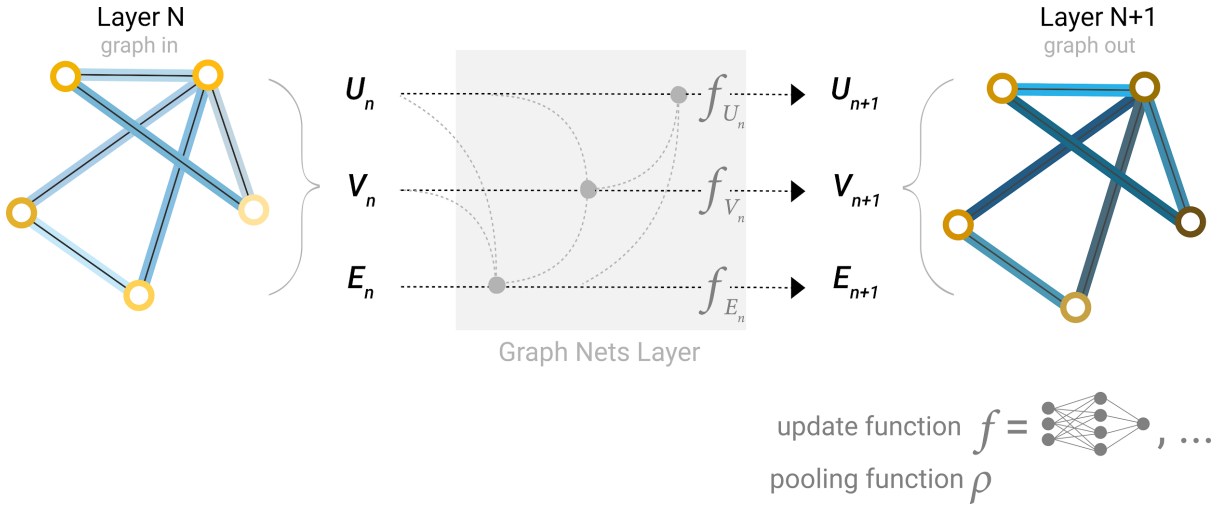
Which graph attributes we update and in which order we update them is one design decision when constructing GNNs. We could choose whether to update node embeddings before edge embeddings, or the other way around. This is an open area of research with a variety of solutions– for example we could update in a 'weave' fashion ,where we have four updated representations that get combined into new node and edge representations: node to node (linear), edge to edge (linear), node to edge (edge layer), edge to node (node layer).



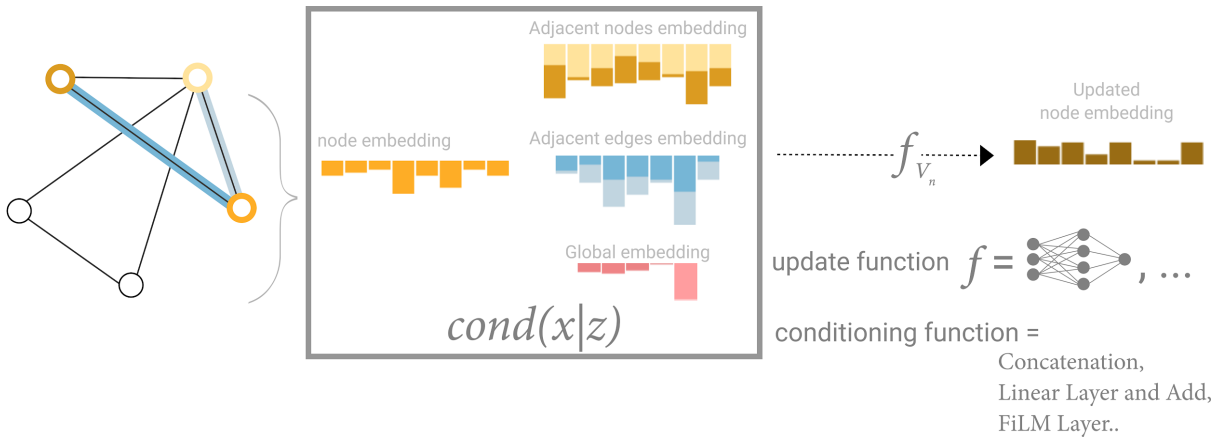
Adding global representations

There is one flaw with the networks we have described so far: nodes that are far away from each other in the graph may never be able to efficiently transfer information to one another, even if we apply message passing several times. For one node, If we have k-layers, information will propagate at most k-steps away. This can be a problem for situations where the prediction task depends on nodes, or groups of nodes, that are far apart. One solution would be to have all nodes be able to pass information to each other. Unfortunately for large graphs, this quickly becomes computationally expensive (although this approach, called 'virtual edges', has been used for small graphs such as molecules).

One solution to this problem is by using the global representation of a graph (U) which is sometimes called a master node or context vector. This global context vector is connected to all other nodes and edges in the network, and can act as a bridge between them to pass information, building up a representation for the graph as a whole. This creates a richer and more complex representation of the graph than could have otherwise been learned.



In this view all graph attributes have learned representations, so we can leverage them during pooling by conditioning the information of our attribute of interest with respect to the rest. For example, for one node we can consider information from neighbouring nodes, connected edges and the global information. To condition the new node embedding on all these possible sources of information, we can simply concatenate them. Additionally we may also map them to the same space via a linear map and add them or apply a feature-wise modulation layer, which can be considered a type of featurize-wise attention mechanism.



Types of GNN

The types of Graph Neural Networks (GNNs) can be categorised based on different criteria, including computation modules, graph types, and training types. Here are some types of GNNs based on these categories:

1. Computation Modules:

- Graph Convolutional Network (GCN): Utilises graph convolutional layers to aggregate information from neighbouring nodes.

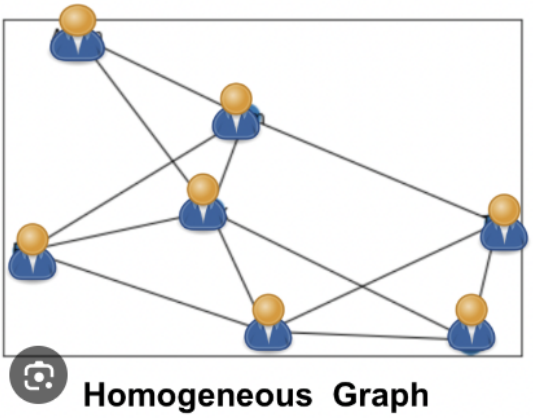
- Graph Attention Network (GAT): Incorporates an attention mechanism to assign different weights to neighbouring nodes during message passing ,[object Object],.

- Graph Recurrent Network (GRN): Uses recurrent neural network (RNN) units to capture temporal dependencies in dynamic graphs.

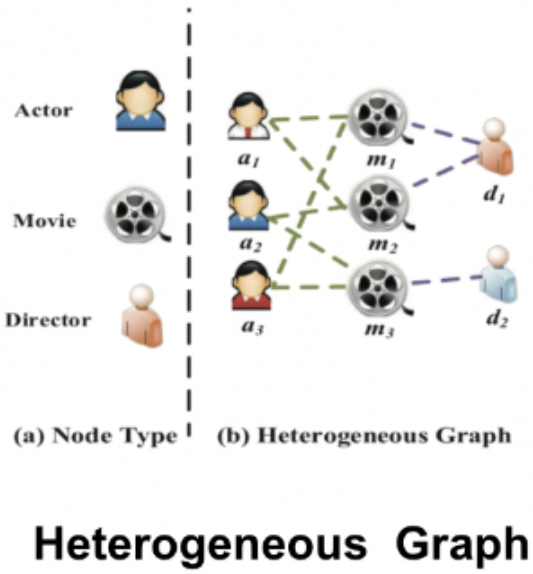
- Gated Graph Neural Network (GGNN): Employs gated recurrent units (GRUs) for message passing and update steps .

2. Graph Types:

- **Homogeneous Graphs**: GNNs designed for graphs where all nodes and edges belong to the same type.

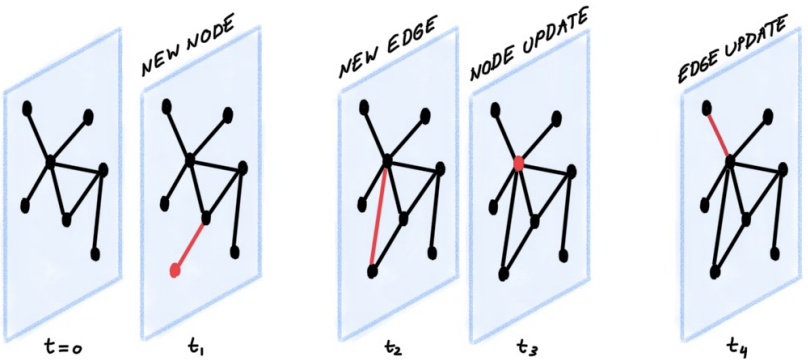


- **Heterogeneous Graphs**: GNNs tailored for graphs with nodes and edges of multiple types, such as social networks with different types of entities (users, products, etc.).



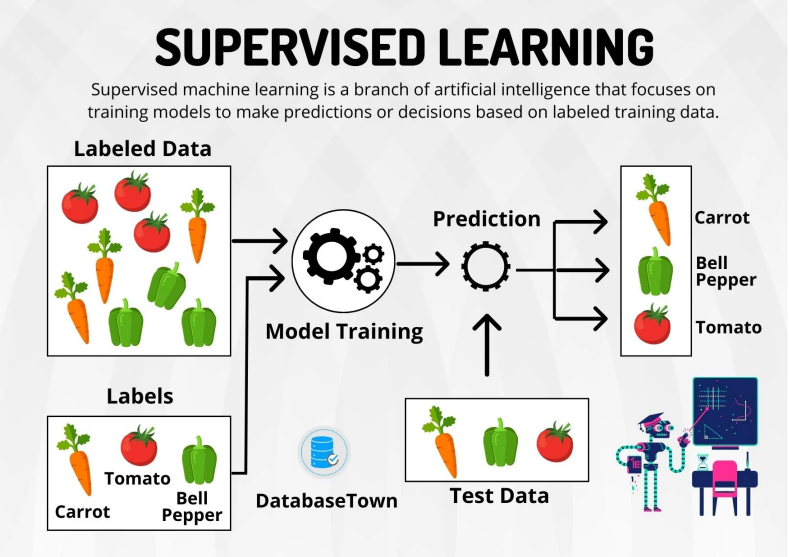
Heterogeneous Graph

- **Dynamic Graphs**: GNNs adapted to handle graphs that evolve over time, capturing temporal dependencies and changes in the graph structure.

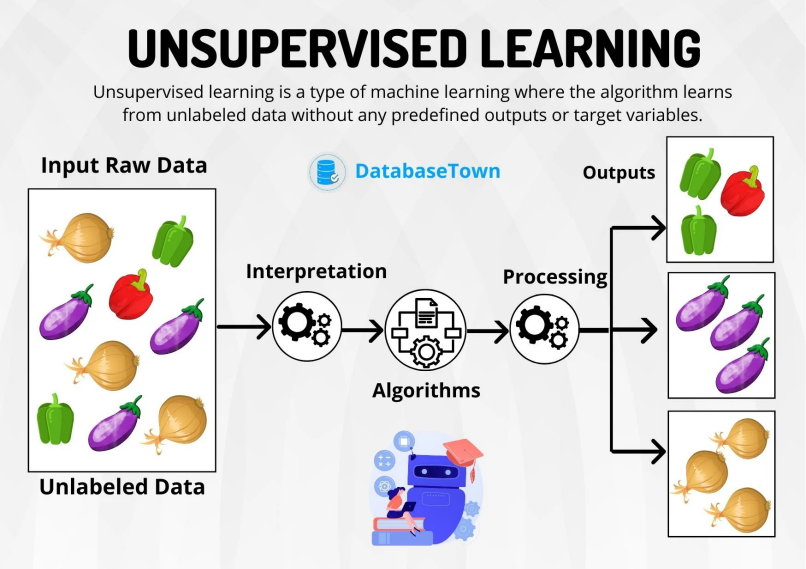


3. Training Types:

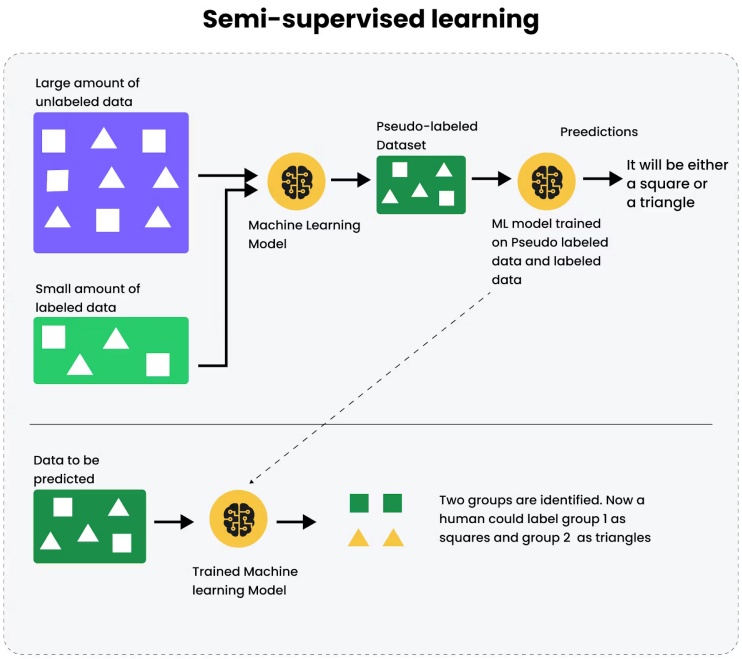
- **Supervised Learning**: GNNs trained on labelled data for tasks such as node classification, link prediction, and graph classification.



- **Unsupervised Learning**: GNNs trained on unlabeled data to learn meaningful representations of the graph structure.



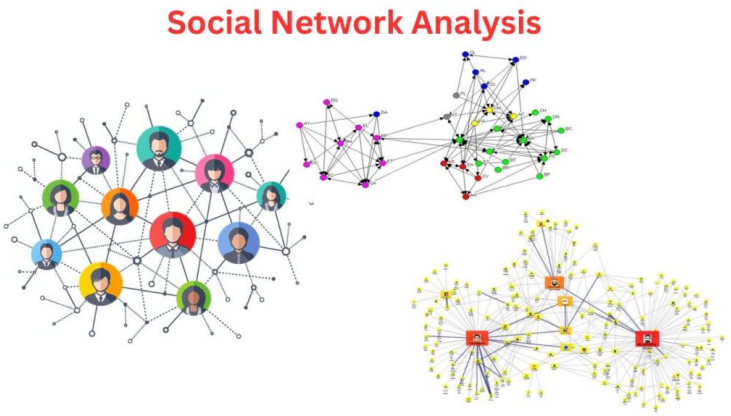
- **Semi-Supervised Learning**: GNNs trained on a combination of labelled and unlabeled data, leveraging both types of information for learning.



Real-world Examples

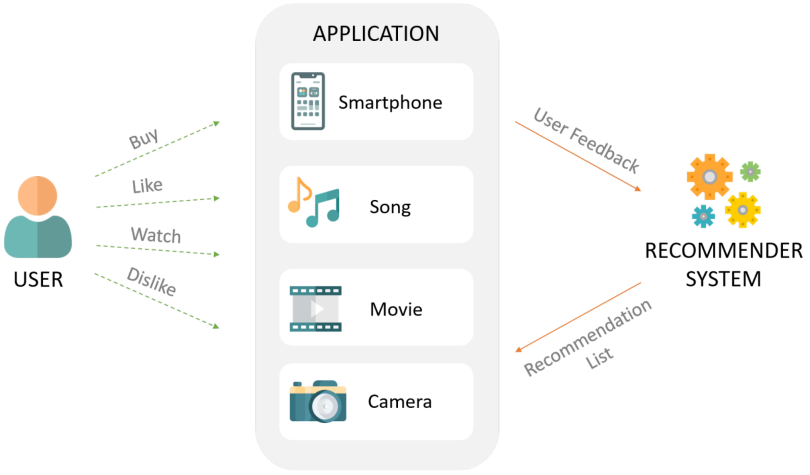
1. Social Network Analysis:

- **Application:** Social Media Platforms (Facebook, Twitter, LinkedIn)
- **Graph Representation:** Nodes represent individuals, and edges represent connections (friendship, family).

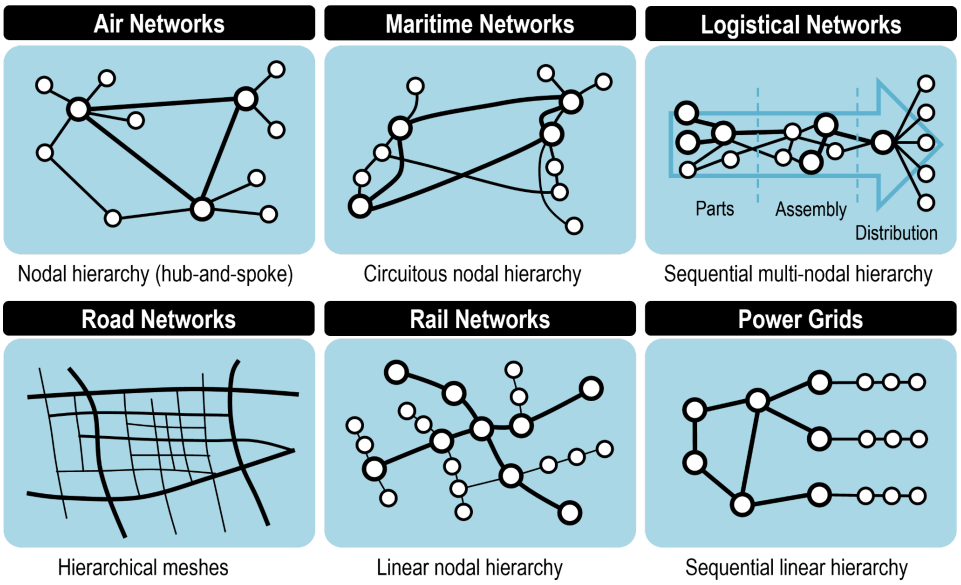


2. Recommendation Systems:

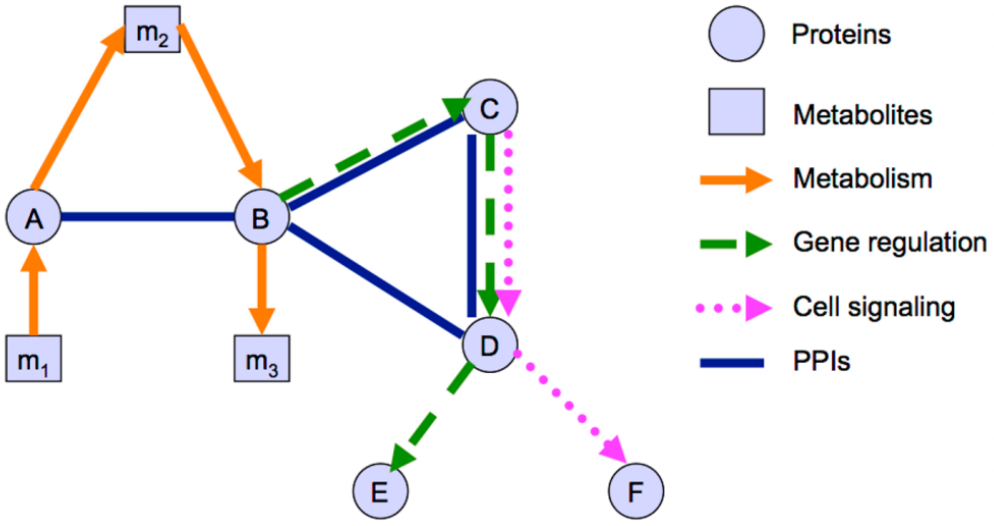
- **Application:** Netflix, Amazon, Spotify
- **Graph Representation:** Nodes represent users or items, and edges represent user-item interactions or similarities.



3. **Transportation Networks:**
   - **Application:** Google Maps, GPS Navigation Systems
   - **Graph Representation:** Nodes represent locations, and edges represent roads or transportation links.

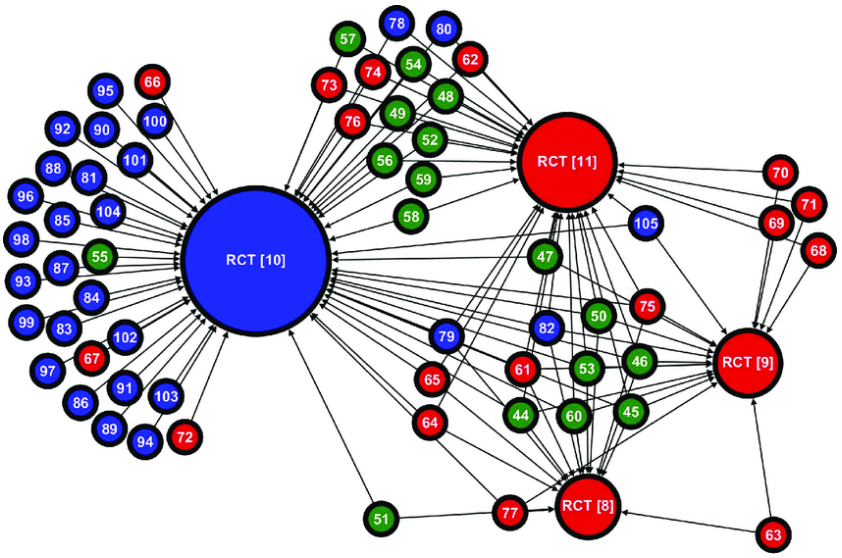


4. **Biological Networks:**
   - **Application:** Protein-Protein Interaction, Metabolic Pathways
   - **Graph Representation:** Nodes represent biological entities (proteins, genes), and edges represent interactions or relationships.



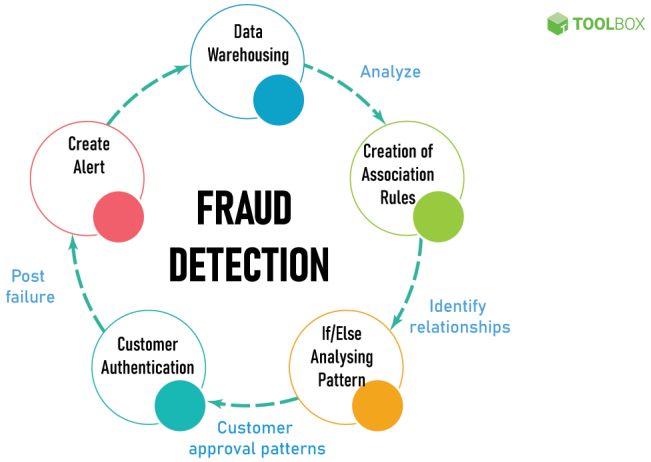
5. <u>Citation Networks</u>:

- **Application**: Academic Paper Citations
- **Graph Representation**: Nodes represent papers, and edges represent citations.



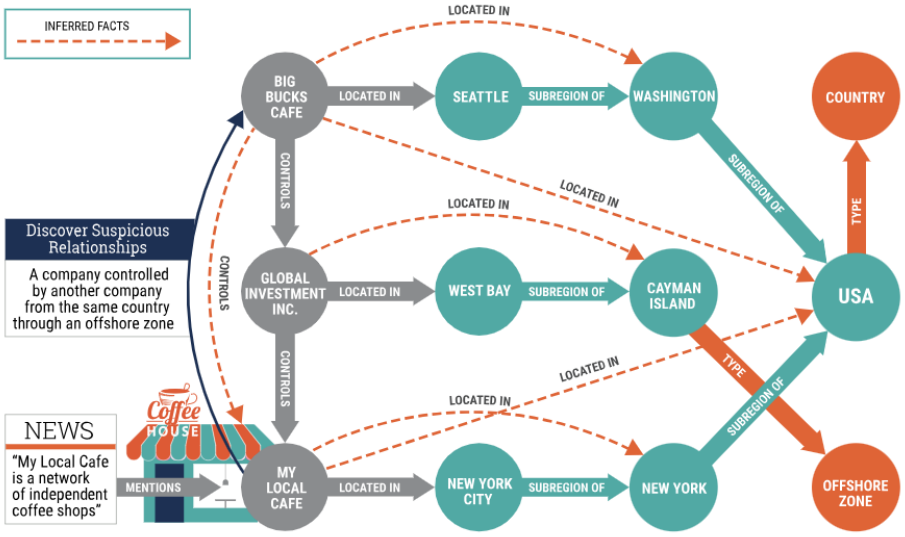
6. <u>Fraud Detection:</u>

- **Application:** Banking and Financial Transactions
- **Graph Representation**: Nodes represent accounts, and edges represent transactions or connections between accounts.



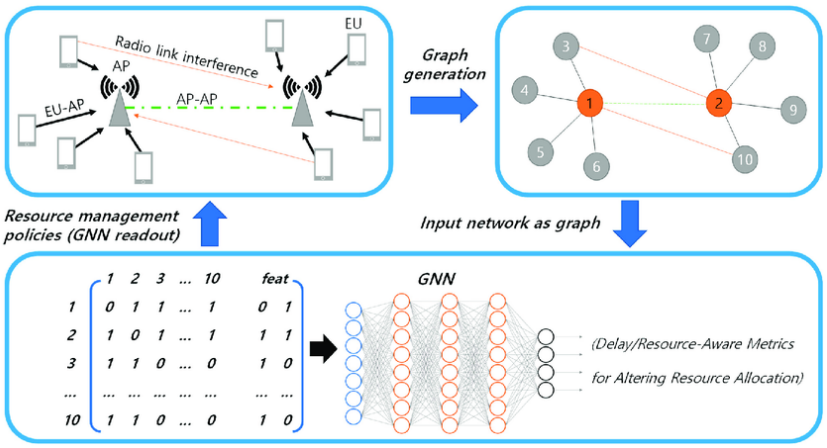
7. <u>Knowledge Graphs</u>:

- **Application**: Wikipedia, Google Knowledge Graph
- **Graph Representation**: Nodes represent entities (e.g., people, places), and edges represent relationships between entities.



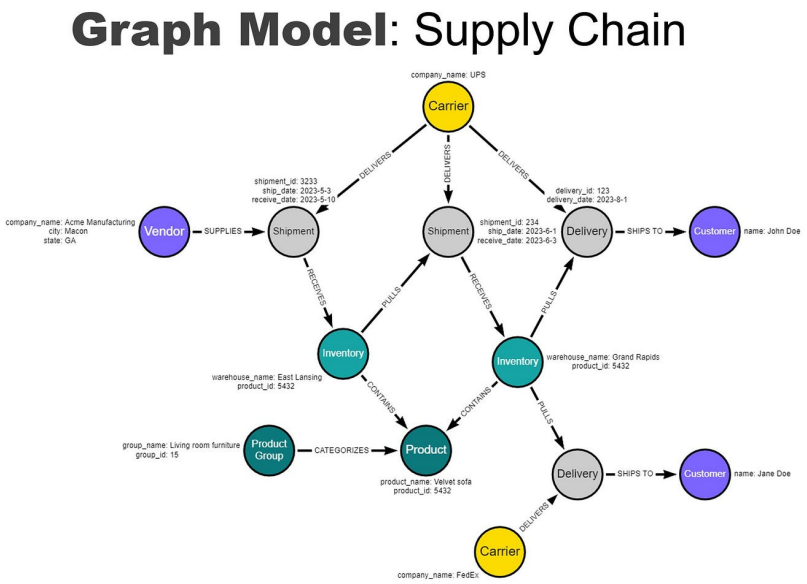
8. <u>Communication Networks</u>:

- **Application**: Email Communication, Telecommunications
- **Graph Representation**: Nodes represent communication entities (users, devices), and edges represent communication channels.



9. <u>Supply Chain Management</u>:

- **Application**: Logistics and Inventory Management
- **Graph Representation**: Nodes represent suppliers, warehouses, and retailers, and edges represent the flow of goods.

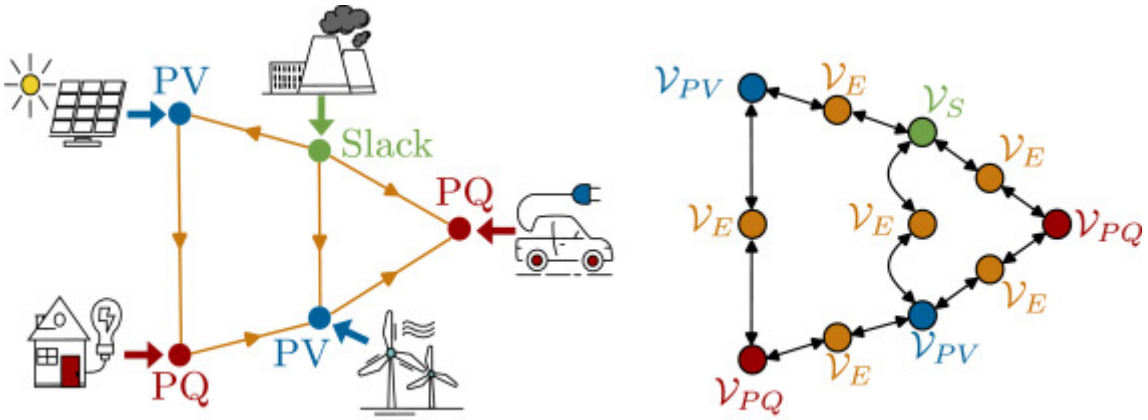


10. <u>Healthcare Networks</u>:

- **Application**: Patient Record Systems, Disease Spread Modeling
- **Graph Representation**:Nodes represent patients or healthcare entities, and edges represent interactions or dependencies.

11. <u>Power Grids</u>:

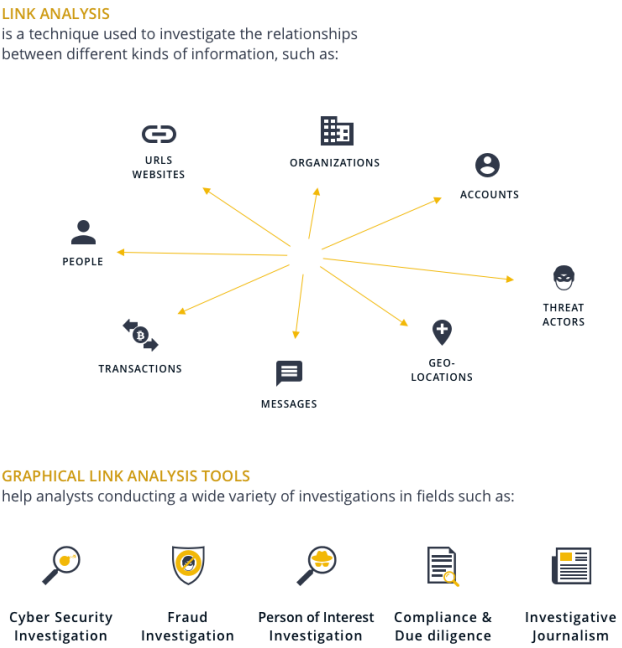
- **Application**: Electrical Power Distribution
- **Graph Representation**: Nodes represent power stations, substations, and consumers, and edges represent power lines.



(a) Power grid toy example (b) TGN representation

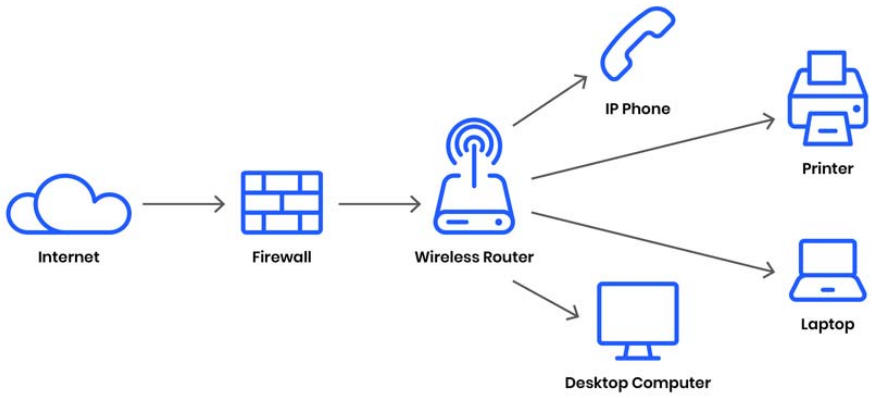
12. <u>Web Link Analysis</u>:

- **Application**: Search Engine Algorithms (e.g., Google PageRank)
- **Graph Representation**: Nodes represent web pages, and edges represent hyperlinks.



13. <u>Computer Networks</u>:

- **Application**: Internet Infrastructure, Network Security
- **Graph Representation**: Nodes represent devices (computers, routers), and edges represent network connections.



14. <u>Smart City Planning</u>:

- **Application**: Urban Planning, Traffic Management
- **Graph Representation**: Nodes represent city elements (roads, buildings, sensors), and edges represent connections or dependencies.

Implementation:

To see how GNN works, we did a simple experiment. We wrote a Python script that creates a graph, performs node embedding on it using its edge relations, trains a machine learning model on a large graph, and then makes predictions on relationships between nodes for some other graph.

Here's a step-by-step explanation:

Graph Creation:

A NetworkX graph is created with nodes representing individuals and edges representing different types of relationships: 'Family' (F), 'Colleague' (C), and 'No Relationship' (N).

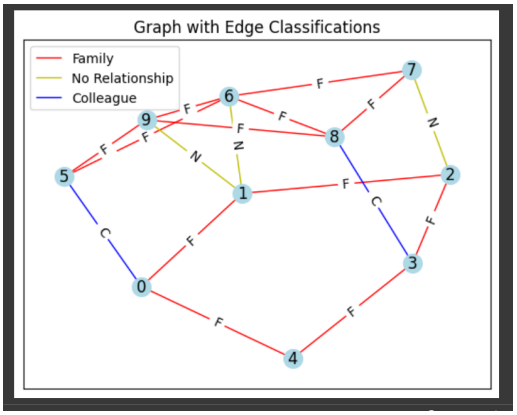
Nodes in the graph are structured into families to enhance the comprehensibility of the network. Specifically, family relationships are modelled as pentagons, where five nodes are interconnected in a cyclic manner. For example, node 0 is related to node 1 through a family connection, then 1 to 2 and this pattern continues until node 4 is connected back to node 0, forming a closed loop.

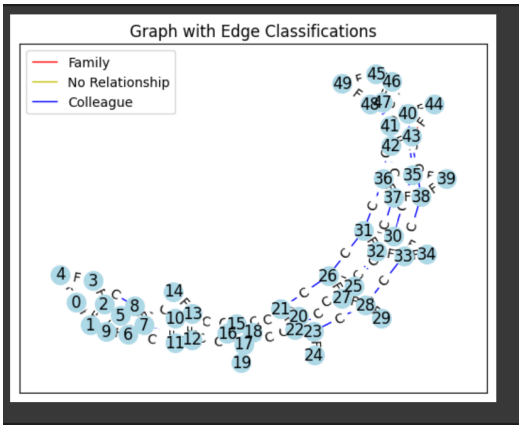
Multiple such family pentagons are created to provide a clear and structured representation of family relationships within the graph. To introduce additional complexity and realism, edges are established to represent colleague and no-relationship connections. These connections are formed both within individual families and between different family pentagons.

The process involves either hard-coding these relationships or randomly creating edges between nodes to simulate diverse social interactions. This approach results in a synthetic yet organised social

network graph, where family structures are explicitly defined, and various relationship types are introduced to train and test models for predicting connections within and between families.

This structured graph serves as a useful foundation for experimenting with machine learning models, particularly those focused on relationship prediction within social networks.





The first picture shows a larger set of 50 nodes that we used to train our model. These nodes are connected in a specific way to represent relationships. The second image is a simpler version with just 10 nodes, organised into two families. We made it simpler to help explain how we defined relationships in the graph.

Node Embedding:

Node embedding is a technique used to represent nodes in a graph as vectors in a continuous vector space. It transforms the discrete and sparse structure of a graph into a dense and continuous form, which is more amenable for machine learning algorithms.

We used the Node2Vec algorithm for node embeddings. Node2Vec is an algorithm designed for learning continuous representations (embeddings) of nodes in a graph. It is an extension of the Word2Vec model, commonly used for natural language processing tasks. Node2Vec explores the graph structure by generating random walks and learning embeddings that capture the structural and relational information of nodes. A random walk is a sequence of nodes traversed by starting from a certain node and randomly choosing neighbours to visit at each step. By exploring various random walks, Node2Vec captures both local and global graph structure.

Once random walks are obtained, Node2Vec treats them as sentences, where nodes are analogous to words. A Word2Vec model is then trained on these "sentences" to learn embeddings for each node.

Node embedding is powerful because it transforms the graph's topological information into a format that machine learning models can easily utilize, enabling the development of predictive models on graph-structured data.

The effectiveness of the node embeddings was assessed by examining the cosine similarities between nodes. Cosine similarity is a metric that quantifies the similarity between vectors. In this context, it was used to measure how closely related nodes were in the learned embedding space.

Nodes expected to be highly related, either due to a direct edge or a sequence of edges connecting them, displayed high cosine similarity

values. Specifically, for nodes with established connections, the cosine similarity exceeded 0.8, indicating a significant alignment in their embedding representations. Conversely, for nodes considered unrelated, the cosine similarity was notably lower, around 0.02. These findings affirm that the node embeddings successfully captured the relational nuances within the graph, effectively distinguishing between related and unrelated nodes based on their proximity in the learned vector space.

Train-Test Split:

Instead of using the built-in split function, we created a custom way to make sure each type of relationship gets a fair share in both the training and testing sets.

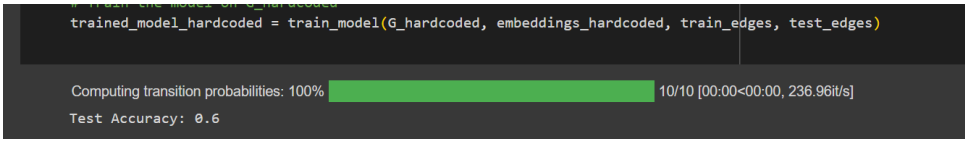
We started by extracting all the edges from the graph, including the relationship type information stored as attributes. The extracted edges are then separated into three lists based on their relationship type: Family ('F'), Colleague ('C'), and No Relationship ('N'). Then we did a train-test split for each relationship type using the train_test_split function from scikit-learn. The train and test edges for each relationship type are then concatenated to create the final train and test sets.

This custom-splitting approach helps our model understand the various types of connections in a balanced way.

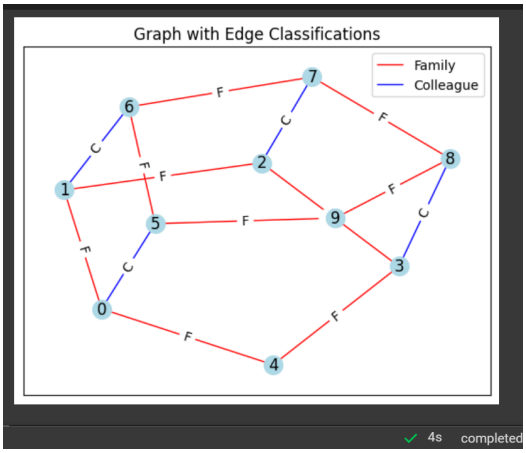
Model Training and Evaluation:

A logistic regression model was trained using the acquired node embeddings to predict relationships within the graph. The model was

fitted to the training set, and its accuracy was evaluated on the testing set. Notably, the highest accuracy achieved was 60% when considering a dataset with over 100 families.



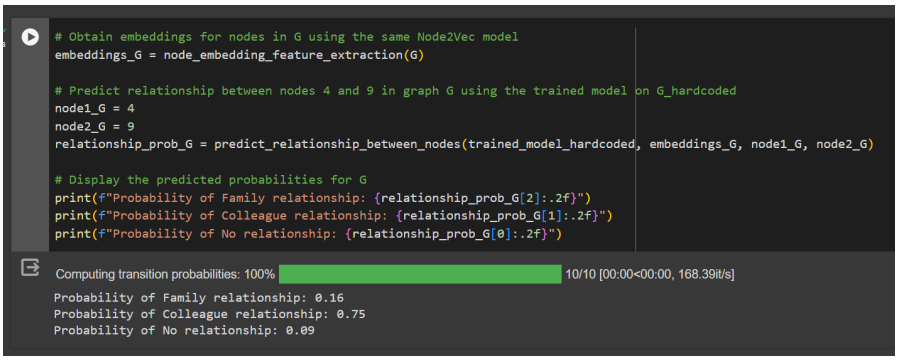
After training our model on a larger graph containing 100 families, we tested its predictive capabilities on a smaller graph and got decent results. But still there is a lot of scope for improvement. Here is an example:



Here, specifically, we aimed to predict the relationship between nodes 4 and 9, anticipating a colleague relationship based on our intuition and understanding of the graph structure.

So here to use the same model, the crucial step was to ensure that the node embeddings for the new graph come from the same embedding space. So we obtained node embeddings for the new graph using the same Node2Vec model that was initially trained on for our graph with 100 families. This ensures consistency in the embedding space. With the embeddings for the new graph in hand, we then used the logistic

regression model trained on our initial graph to predict edge relationships on the new graph. And this is the result we got:



The result was a 75% probability of a colleague relationship between nodes 4 and 9. This outcome aligns with our expectations, considering the prevalent colleague relationships among family members of node 4 and those of node 9.

Scope for Improvement:

1. **Fine-Tuning Model Hyperparameters:**

Experiment with different hyperparameter settings during the training phase, such as the dimensions of node embeddings, the length of random walks, and the number of walks per node. Fine-tuning these parameters could potentially lead to improved model performance.

2. **Exploration of Advanced Embedding Techniques:**

Explore other advanced graph embedding techniques beyond Node2Vec, such as GraphSAGE, DeepWalk, or Graph Attention Networks. Each method has its strengths and may capture different aspects of the graph structure more effectively.

3. **Trying Out New Models:**

Consider experimenting with entirely new graph-based models. Graph neural networks (GNNs) and Graph Isomorphism Networks (GIN) are examples of powerful models that can capture complex relationships in graph-structured data. Assessing their performance alongside traditional methods can provide valuable insights.

4. **Handling Imbalanced Data:**

Address potential class imbalance issues in the dataset. If certain relationship types are underrepresented, the model may not generalise well to these classes. Techniques such as oversampling, undersampling, or using class weights during training can help balance the learning process.

5. **Incorporation of Edge Features:**

Consider incorporating additional features related to edges, if available. Information such as the frequency of interaction, temporal aspects, or domain-specific features could provide valuable context for relationship prediction.

6. **Evaluation on Diverse Graph Structures:**

Assess the model's performance on a variety of graph structures beyond the initial cyclic pentagon family configuration. Evaluating the model on diverse graphs can help ensure its robustness across different scenarios.

7. **Optimization of Training Set Size:**

Investigate the impact of the training set size on model performance. Expanding the dataset or experimenting with different proportions of training data may uncover patterns and relationships that contribute to better generalisation.

8. **Ensemble Methods:**

Explore the use of ensemble methods by combining predictions from multiple models. Ensemble techniques, such as bagging or boosting, can enhance predictive accuracy by leveraging diverse model perspectives.

9. **Trying Out New Graph-Based Models:**

Experiment with entirely new graph-based models that have shown success in similar tasks. GNN architectures like Graph Convolutional Networks (GCNs) and Graph Attention Networks (GATs) could offer improved performance, especially in capturing complex relationships in large graphs.

10. **Fine-Grained Relationship Prediction:**

Consider refining the model to predict more fine-grained relationships within the 'Colleague' category. For example, distinguishing between different types of professional connections may lead to more nuanced and accurate predictions.

By systematically exploring new models alongside the suggested improvements, the overall predictive capability of the system can be enhanced, and a more comprehensive understanding of the underlying relationships within the graph can be achieved.

References:

https://distill.pub/2021/gnn-intro/

https://arxiv.org/ftp/arxiv/papers/1812/1812.08434.pdf

All the images are taken from google.