Graph Representation Learning for Binding Pocket Prediction in Proteins

A Thesis

submitted to
Indian Institute of Science Education and Research Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

by

Karampuri Yash Sunil

Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

May, 2025

Supervisor: Prof. Arnab K. Laha

Certificate

This is to certify that this dissertation entitled Graph Representation Learning for Binding Pocket Prediction in Proteins towards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by Karampuri Yash Sunil at the Indian Institute of Management Ahmedabad under the supervision of Prof. Arnab K. Laha, Professor, Department of Operations and Decision Sciences, during the academic year 2024-2025.

Arnab Kumar Laha

Prof. Arnab K. Laha
Supervisor
IIM Ahmedabad

Prof. M.S. Madhusudhan
Expert
IISER Pune

Committee:

Prof. Arnab K. Laha

Prof. M.S. Madhusudhan

To Maa, Pappa and Samiksha didi

Declaration

I hereby declare that the matter embodied in the report entitled Graph Representation Learning for Binding Pocket Prediction in Proteins are the results of the work carried out by me at the Department of Operations and Decision Sciences, Indian Institute of Management, Ahmedabad, under the supervision of Prof. Arnab K. Laha and the same has not been submitted elsewhere for any other degree.

Karampuri Yash Sunil
Enr. No. 20201105
BS-MS IISER Pune
Date: 25th March 2025

Acknowledgments

I would like to express my deepest gratitude to my advisor, Prof. Arnab Kumar Laha, for his invaluable guidance, encouragement, and constant support throughout my time at IIM Ahmedabad. His insightful feedback and thoughtful suggestions greatly shaped the direction of my work. I am also sincerely thankful to my expert, Prof. M. S. Madhusudhan, for his constructive remarks and valuable inputs that enriched this thesis. Working under him in the COSPI Lab during my undergraduate studies taught me how to conduct quality research with rigor and integrity. I am deeply grateful to both my advisor and expert for the knowledge, guidance, and support they have provided me—both in academic pursuits and beyond, which has had a lasting impact on my academic and personal growth.

I extend my heartfelt gratitude to my parents and sister for their unconditional love, patience, and unwavering support throughout this journey. Their encouragement has been the foundation of my strength, enabling me to overcome challenges and remain committed to my work. My sister has been a constant source of inspiration, teaching me the importance of perseverance, embracing failure, and trying again with renewed determination. My mother has always motivated me to aim higher and pursue my ambitions with confidence. My father, through his wisdom and outlook on life, has shown me the value of leading a happy and fulfilled life. I am deeply indebted to them for their faith in me.

I am equally grateful to my friends, who stood by me during difficult times with their understanding and companionship. Without their support, this thesis would not have been possible. I am also thankful to IISER Pune for shaping my character and boosting my self-esteem through its vibrant academic programs, extracurricular activities, and sports. My experiences there have been instrumental in my development as both a researcher and an individual.

Abstract

This study explores several key concepts in graph-based learning and applies them to the problem of ligand-binding pocket prediction and clustering on protein surfaces. First, we investigate graph embedding techniques, scalable feature learning with algorithms like Node2vec, and graph representation learning methods. We then explore neighborhood reconstruction methods and how multi-relational data and knowledge graphs can be used, building a solid foundation for applying graph-based techniques to biological data. Next, we focus on the problem of predicting and clustering ligand-binding pockets on protein surfaces. Using a graph-based approach, we first generate a set of evenly spaced points on the protein's Solvent Accessible Surface (SAS) with a fast algorithm from the CDK library. For each point, we calculate feature descriptors based on the local chemical environment, including properties of solvent-exposed atoms, distance-weighted properties of nearby atoms (within 6A), and other neighborhood features. These descriptors are used to predict ligandability scores through Graph Neural Networks (GNN) and Graph Convolutional Networks (GCN). Points with high ligandability scores are then clustered using single-linkage clustering with a 3A cut-off to form pocket predictions. The predicted pockets are ranked by their cumulative ligandability scores. This method provides an efficient framework for identifying potential ligand-binding pockets, contributing to drug discovery and protein-ligand interaction studies.

Contents

Introduction

Graphs are extensive data structures and a global language used to describe complex systems. Generally, a graph is a collection of objects (nodes) with a set of interactions between pairs of these objects (edges). Consider encoding a biological domain as a graph; nodes can be used to represent proteins, and edges can be used to represent biological interactions (like kinetic interactions).

Graphs can be used to set up a mathematical foundation that can be built upon to analyze, comprehend, and learn from the complex systems in the real world. With the rise of social media platforms, large scientific projects to map biological interactions, food chains, databases of molecular structures, and billions of connected devices, researchers have plenty of graph data to study. The challenge is figuring out how to make the most of this data.

Challenges of Graph Deep Learning: Networks are inherently complex structures with arbitrary and intricate topologies, which lack the spatial locality seen in simpler structures like grids. They do not have a fixed node ordering or reference point and are often dynamic in nature, featuring multimodal attributes. Different tasks can be performed on networks, such as predictions at the node level, edge level, community (subgraph) level, and graph level.

Node-level tasks include node classification, where, for example, the category of a protein in a protein-protein interaction network is predicted. Node regression involves predicting the value of a property, such as the toxicity of a chemical compound in molecular graphs. Node clustering is used to group similar proteins in an interaction network for functional analysis. Another important thing possible is to rank nodes, like how Google ranks websites based on how many other websites link to them. Node recommendation, such as suggesting friends in a social network based on mutual connections, is another common application. Finally, node

anomaly detection, such as identifying fraudulent users in a transaction network, focuses on spotting unusual patterns.

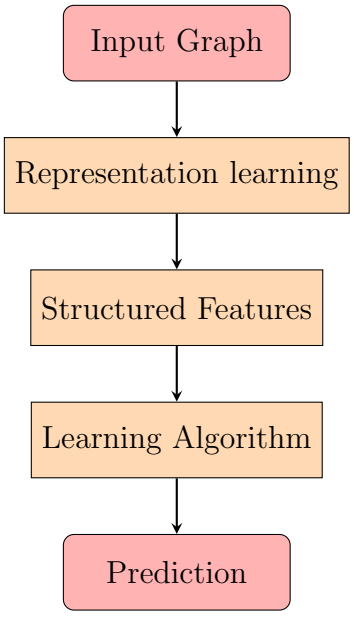
At the edge level, tasks like edge attribute prediction, which involves predicting the weight or strength of connections between nodes based on interaction history, are important. Edge classification categorizes the type of relationship between two nodes, such as friendship or collaboration. Edge recommendation aims to suggest potential new connections, particularly in knowledge graphs. Edge importance ranking helps prioritize critical routes in transportation networks, while edge anomaly detection identifies suspicious connections in cybersecurity networks. Strategies for edge deletion or addition are essential in optimizing financial networks, and edge dynamics prediction is used to anticipate changes in connections over time, such as in evolving social or biological networks. Such tasks, performed by Graph Neural Networks, unlock powerful insights into complex systems, which enables better analysis, understanding, and learning from real-world data.

Graph Neural Networks (GNNs) have been extensively utilized in various subgraph-level and graph-level tasks. At the subgraph level, GNNs can be employed for tasks such as subgraph classification, where the goal is to classify entire subgraphs, like identifying communities or cliques in social networks. Another important task is subgraph matching, which involves finding specific patterns or motifs within larger graphs, as seen in biological networks. Subgraph generation is useful for producing new subgraphs that share structural properties with a query subgraph, helping in synthetic data generation and augmentation. GNNs also support subgraph querying, enabling complex data retrieval by finding instances that match specific subgraph patterns in large knowledge graphs. Subgraph sampling helps reduce computational complexity by extracting representative subgraphs for statistical analysis. Additionally, GNNs can predict subgraph dynamics, such as forecasting changes in community structures over time in dynamic networks. Lastly, subgraph anomaly detection identifies abnormal patterns within subgraphs that differ from expected norms, which is useful in areas like fraud detection.

At the graph level, GNNs perform tasks like graph classification, where entire graphs are classified based on their structure, such as classifying molecular graphs by toxicity levels. Graph regression involves predicting scalar values associated with graphs, like estimating the total energy of molecules. Graph generation allows the creation of new graphs with similar properties to input graphs, which is especially valuable in drug discovery for generating

new chemical structures. Graph comparison measures the similarity between graphs, for instance, comparing social networks by their community structures. Graph clustering groups graphs with similar structures, helping to identify patterns and support exploratory analysis. GNNs are also useful for predicting changes in graph dynamics, such as forecasting shifts in a network's connectivity over time. Lastly, graph anomaly detection identifies outlier graphs that deviate from the norm, which can be helpful in network data analysis and fraud detection.

Traditional ML for Graphs:



Graph Representation Learning:

Goal: Effective graph feature learning for ML with graphs.

$$f : u \rightarrow \mathbb{R}$$

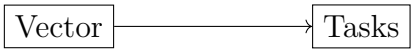
where, R is an embedding representing feature.

Why Embedding?

Task: Mapping Nodes into an Embedding Space The primary objective is to project nodes into an embedding space where the similarity between their embeddings reflects their structural or relational similarity in the network. For instance: - If two nodes are connected by an edge, their embeddings should be positioned close to each other in the embedding space.

This process serves several purposes: - Encoding the network's structural information. - Facilitating various downstream prediction tasks.

—



—

Setup Consider an undirected graph defined as follows: - $\mathcal{V}$ represents the set of vertices. - $\mathbf{A}$ denotes the binary adjacency matrix. - For simplicity, no node features or additional information are utilized.

Example: - Vertex set: $\mathcal{V} = \{1, 2, 3, 4\}$ - Adjacency matrix:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Node Embedding in Vector Spaces Embedding nodes into a low-dimensional vector space is a fundamental technique in graph analysis. The goal is to map nodes such that their proximity in the embedding space reflects their similarity in the original graph. For example,

if two nodes are connected by an edge, their embeddings should be close to each other. This process encodes the network's structure in a way that supports efficient downstream tasks, such as link prediction or node classification.

The dot product of two node embeddings serves as a measure of their similarity in the embedding space, approximating their relationship in the original graph. This ensures that the network's relational structure is preserved in a computationally efficient form.

Embedding Process The embedding process consists of two main components: 1. **Encoder**: Maps each node to a d-dimensional vector in the embedding space. 2. **Similarity Function**: Defines how relationships in the embedding space correspond to those in the original graph. Typically, the dot product of embeddings is used to measure similarity.

The process is unsupervised, meaning it does not rely on node labels or features. Instead, it focuses solely on capturing the graph's structural properties, making the embeddings versatile and applicable to a wide range of tasks.

Defining Node Similarity A critical consideration in node embedding is the definition of node similarity. Should nodes be considered similar if they: - Are directly connected by an edge? - Share common neighbors? - Occupy similar structural roles in the network?

The choice of similarity metric significantly influences how the embeddings reflect the graph's structure. Different embedding methods may emphasize different aspects of similarity, depending on the specific application.

Chapter 1

Preliminaries

Permutation Invariance and Equivariance

Initial Approach: A straightforward method for defining deep neural networks on graphs involves directly using the adjacency matrix. For example, to generate a graph embedding, one could flatten the adjacency matrix and pass it through a Multi-Layer Perceptron (MLP):

$$\mathbf{z}_{\mathcal{G}} = \mathrm{MLP}\left(\mathbf{P}_{[1]} \oplus \mathbf{P}_{[2]} \oplus \cdots \oplus \mathbf{P}_{[|\mathcal{V}|]}\right),$$

where $\mathbf{P}_{[i]} \in \mathbb{R}^{|\mathcal{V}|}$ represents the i-th row of the adjacency matrix.

The Issue: This approach has a significant limitation: it depends on the arbitrary ordering of nodes in the adjacency matrix. Consequently, the model lacks **permutation invariance**, a fundamental requirement for graph-based neural networks.

Key Desideratum: For neural networks operating on graphs, it is essential to exhibit either **permutation invariance** or **permutation equivariance**. Formally, for a function f that takes an adjacency matrix $\mathbf{A}$ as input, the following properties must hold:

$$f(\mathbf{S}\mathbf{A}\mathbf{S}^\top) = f(\mathbf{A}) \quad \text{(Permutation Invariance)},$$

$$f(\mathbf{S}\mathbf{A}\mathbf{S}^\top) = \mathbf{S}f(\mathbf{A}) \quad \text{(Permutation Equivariance)},$$

where $\mathbf{S}$ is a permutation matrix.

Interpretation: - **Permutation Invariance**: The function f should produce the same output regardless of the ordering of rows and columns in the adjacency matrix. - **Permutation Equivariance**: If the adjacency matrix is permuted, the output of f should be permuted in a corresponding manner.

1.1 Basic Framework of Graph Neural Networks (GNNs)

Structural Information

Graph Neural Networks (GNNs) capture structural information about graphs through node embeddings, denoted as $\mathbf{l}_u^{(t)}$, which encode details about node degrees within a t-hop neighborhood. For instance, in molecular graphs, degree information can help identify atom types and structural motifs such as benzene rings.

Feature Information

In addition to structural data, GNN embeddings also incorporate features from their t-hop neighborhoods. This feature aggregation is analogous to how Convolutional Neural Networks (CNNs) gather information from spatial patches in images, whereas GNNs perform this operation based on local graph neighborhoods.

Basic GNN Framework

GNNs operate through iterative message-passing steps, which involve two key operations: **UPDATE** and **AGGREGATE**. The fundamental message-passing equation is expressed as:

$$\mathbf{l}_u^{(t)} = \sigma\left(\mathbf{W}_{\text{self}}^{(t)}\mathbf{l}_u^{(t-1)} + \mathbf{W}_{\text{neigh}}^{(t)} \sum_{v \in \mathcal{N}(u)} \mathbf{l}_v^{(t-1)} + \mathbf{b}^{(t)}\right),$$

where $\mathbf{W}_{\text{self}}^{(k)}$ and $\mathbf{W}_{\text{neigh}}^{(k)}$ are trainable weight matrices, and σ represents a non-linear activation function such as ReLU.

1.2 Node-Level vs. Graph-Level Definitions

The operations defined for individual nodes can be generalized to the graph level. The graph-level update equation is given by:

$$\mathbf{L}^{(t)} = \sigma\left(\mathbf{A}\mathbf{H}^{(t-1)}\mathbf{W}_{\text{neigh}}^{(t)} + \mathbf{H}^{(t-1)}\mathbf{W}_{\text{self}}^{(t)}\right),$$

where $\mathbf{L}^{(t)}$ is the matrix of node representations, and $\mathbf{A}$ is the adjacency matrix of the graph.

Self-loops in GNNs

Incorporating self-loops into the graph simplifies the message-passing process by allowing each node to aggregate its own features. This can be expressed as:

$$\mathbf{l}_u^{(t)} = \text{AGGREGATE}\left(\{\mathbf{l}_v^{(t-1)} \mid \forall v \in \mathcal{N}(u)\} \cup \{u\}\right).$$

This approach eliminates the need for a separate update function but may reduce the model's expressiveness.

When self-loops are included, the graph-level update simplifies to:

$$\mathbf{L}^{(k)} = \sigma\left((\mathbf{A} + \mathbf{I})\mathbf{L}^{(k-1)}\mathbf{W}^{(k)}\right).$$

Neighborhood Normalization

Summing neighbor embeddings during aggregation can lead to instability, particularly for nodes with varying degrees. A common solution is to normalize the aggregation by averaging:

$$\mathbf{s}_{\mathcal{N}(u)} = \frac{\sum_{v \in \mathcal{N}(u)} \mathbf{l}_v}{|\mathcal{N}(u)|},$$

or by using symmetric normalization, which helps mitigate numerical instabilities during optimization.

Graph Convolutional Networks (GCNs)

GCNs employ symmetric-normalized aggregation combined with self-loop updates. The message-passing function is defined as:

$$\mathbf{l}_u^{(t)} = \sigma\left(\mathbf{W}^{(t)}\left(\sum_{v\in\mathcal{N}(u)\cup\{u\}} \frac{\mathbf{l}_v}{\sqrt{|\mathcal{N}(u)|\cdot|\mathcal{N}(v)|}}\right)\right).$$

Normalization Considerations

While normalization enhances stability and performance, it can obscure structural information, making it difficult to distinguish nodes of varying degrees. The decision to normalize depends on the specific application and the relative importance of node features versus structural information.

1.3 Over-smoothing in GNNs

A significant challenge in deeper GNNs is *over-smoothing*, where node representations converge to similar values as the number of layers increases, leading to a loss of distinct node features.

The influence of node u on node v is quantified using the Jacobian matrix:

$$\mathcal{I}_T(u,v) = \mathbf{1}^\top\left(\frac{\partial \mathbf{l}_v^{(T)}}{\partial \mathbf{l}_u^{(0)}}\right)\mathbf{1},$$

where $\mathcal{I}_T(u,v)$ measures how the initial embedding of node u impacts the final representation of node v.

Alleviating Over-smoothing

To mitigate over-smoothing, techniques such as concatenation-based skip connections can be employed. These methods retain information from previous message-passing rounds by directly incorporating earlier node representations into the update step:

$$\text{UPDATE}_{\text{concat}}(\mathbf{l}_u, \mathbf{s}_{\mathcal{N}(u)}) = \text{UPDATE}_{\text{base}}(\mathbf{l}_u, \mathbf{s}_{\mathcal{N}(u)}) \oplus \mathbf{l}_u.$$

Skip Connections in Graph Neural Networks

Skip connections are techniques used in GNNs to enhance information flow and address issues like over-smoothing. Two common methods are discussed below.

Concatenation-based Skip Connections

This approach concatenates the node's previous representation with the output of the base update function:

$$\text{UPDATE}_{\text{concat}}(\mathbf{l}_v, \mathbf{s}_{\mathcal{N}(v)}) = \text{UPDATE}_{\text{base}}(\mathbf{l}_v, \mathbf{s}_{\mathcal{N}(v)}) \oplus \mathbf{l}_v,$$

where $\oplus$ denotes the concatenation operation.

Linear Interpolation-based Skip Connections

This method uses linear interpolation between the base update function and the node's previous representation:

$$\text{UPDATE}_{\text{interpolate}}(\mathbf{l}_v, \mathbf{s}_{\mathcal{N}(v)}) = \boldsymbol{\alpha}_1 \odot \text{UPDATE}_{\text{base}}(\mathbf{l}_v, \mathbf{s}_{\mathcal{N}(v)}) + \boldsymbol{\alpha}_2 \odot \mathbf{l}_v,$$

where $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2 \in [0, 1]^d$ are gating vectors, and $\odot$ denotes elementwise multiplication.

Practical Considerations

Skip connections significantly alleviate over-smoothing and enhance optimization stability, similar to residual connections in CNNs. These techniques are particularly effective for moderately deep GNNs (2–5 layers) and in homophilic tasks, where predictions for a node are closely related to the features of its neighbors.

Chapter 2

Graph Representation learning

2.1 Encoder-Decoder Framework

Node embedding methods aim to encode nodes as low-dimensional vectors that capture their position in the graph and the structure of their local neighborhood. This involves mapping nodes to a latent space where geometric relationships correspond to connections (e.g., edges) in the original graph. Node embeddings can be understood using an encoder-decoder framework, which consists of two main operations:

- **Encoder**: Maps each node to a low-dimensional vector (embedding).
- **Decoder**: Uses these embeddings to reconstruct information about the original graph.

The Encoder

The encoder function maps each node $b \in V$ to a vector embedding $z_b \in \mathbb{R}^d$, where z_b is the embedding for node b. Formally, the encoder is defined as:

$$\text{enc} : V \to \mathbb{R}^d$$

This means the encoder generates node embeddings based on node IDs. A common approach, known as shallow embedding, simply looks up the node embedding in a matrix $Z \in \mathbb{R}^{|V| \times d}$, where $Z[b]$ is the row corresponding to node b:

$$\text{enc}(b) = Z[b]$$

While shallow embedding is often used, the encoder can also incorporate node features or local graph structure to create embeddings.

The Decoder

The decoder's role is to reconstruct graph properties from the node embeddings. For example, it might predict the neighbors $N(a)$ of a node a or its adjacency matrix row $A[a]$.

A typical decoder is pairwise, predicting relationships between pairs of nodes, and is defined as:

$$\text{dec} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+$$

This predicts the similarity between two nodes, for instance, whether they are neighbors. The decoder reconstructs relationships between nodes a and b from their embeddings z_a and z_b and minimizes reconstruction loss:

$$\text{dec}(\text{enc}(a), \text{enc}(b)) = \text{dec}(z_a, z_b) \approx S[a, b]$$

Here, $S[a, b]$ is a similarity measure between nodes, such as the adjacency matrix entry $A[a, b]$, which indicates whether two nodes are neighbors.

Optimizing an Encoder-Decoder Model

The goal of optimizing the encoder-decoder model is to minimize the reconstruction loss, which compares the estimated similarity between nodes with the true similarity. This is

done by minimizing an empirical loss function L over a set of node pairs D:

$$L = \sum_{(a,b)\in D} \ell(\text{dec}(z_a, z_b), S[a, b])$$

where $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a loss function that measures the difference between the predicted similarity $\text{dec}(z_a, z_b)$ and the true similarity $S[a, b]$. Depending on the type of decoder and similarity function used, this loss function can be mean squared error or a classification loss like cross-entropy. The objective is to train the encoder and decoder to accurately reconstruct the relationships between nodes in the training set D. This optimization is typically performed using stochastic gradient descent (SGD).

Overview of the Encoder-Decoder Approach

The encoder-decoder framework helps define and compare different node embedding methods by focusing on three key components:

1. The decoder function.
2. The graph-based similarity measure.
3. The loss function.

Several well-known node embedding methods, which use the shallow encoding approach, can be described using this framework. In the following sections, we will explore two main groups of node embedding methods:

- **Matrix factorization-based methods**: These approaches are closely related to spectral clustering.
- **Random walk-based methods**: These are inspired by techniques from natural language processing and are also theoretically connected to spectral graph theory.

2.2 Background and Traditional Approaches

Graph statistics and kernel methods generalize node-level features to graph-level statistics, forming the basis for graph-based machine learning. Heuristic statistics and graph properties serve as features in traditional pipelines.

Node Degree

The degree of node u is the number of incident edges:

$$d_u = \sum_{v \in V} A[u, v]$$

Different notions of degree (e.g., in-degree, out-degree) arise from summing over matrix rows or columns.

Node Centrality

Eigenvector centrality extends degree centrality by considering a node's neighbors' importance. It is defined as:

$$e_u = \frac{1}{\lambda} \sum_{v \in V} A[u, v] e_v, \quad \forall\, u \in V$$

In vector form:

$$\lambda e = Ae$$

Using the Perron-Frobenius theorem, the eigenvector of the largest eigenvalue gives positive centrality values. Power iteration estimates e:

$$e^{(t+1)} = Ae^{(t)}$$

Starting with $e^{(0)} = (1, 1, \ldots, 1)^\top$, $e^{(1)}$ gives node degrees, and $e^{(t)}$ represents the number

of length-t paths to each node.

Betweenness Centrality

Measures how often a node appears on shortest paths:

$$x_i = \sum_{s,t} \frac{n_i^{st}}{g_{st}}$$

where $n_i^{st} = 1$ if i lies on the shortest path between s and t, and g_{st} is the total number of such paths.

Closeness Centrality

Measures a node's average shortest path distance to others:

$$l_i = \frac{1}{n} \sum_j d_{ij}, \quad C_i = \frac{n}{\sum_j d_{ij}}$$

Clustering Coefficient

Quantifies local connectivity:

$$c_u = \frac{|\{(v_1, v_2) \mid (v_1, v_2) \in \varepsilon, v_1, v_2 \in N(u)\}|}{\binom{d_u}{2}}$$

where the numerator counts edges among node u's neighbors, and the denominator represents all possible neighbor pairs. A coefficient of 1 means a fully connected neighborhood.

Graph-Level Features and Graph Kernels:

Graph-level classification aims to classify entire graphs, such as predicting molecular solubility from molecular graph structures.

Graph kernel methods define explicit graph features or implicit similarity functions for

machine learning models. We focus on explicit feature extraction rather than implicit similarity measures.

Bag of Nodes:

A simple approach to defining graph-level features is aggregating node statistics, such as histograms of degrees, centralities, and clustering coefficients. While useful, this method captures only local properties and may overlook global graph characteristics.

Graphlets and Path-Based Methods:

Graphlets, small subgraph structures, serve as features by counting their occurrences in a graph. The graphlet kernel enumerates all possible substructures of a fixed size and counts their occurrences, though this is computationally challenging.

Path-based methods offer an alternative by analyzing the types of paths in a graph. This approach retains structural information while avoiding the combinatorial challenges of graphlet enumeration.

Neighborhood Overlap Detection:

Standard graph statistics fail to capture node relationships, which is crucial for link prediction. We quantify node similarity using neighborhood overlap measures. The simplest measure counts shared neighbors:

$$S[a, b] = |M(a) \cap M(b)|$$

where $S[a, b]$ is the similarity between nodes a and b, and S summarizes all pairwise similarities. The probability of an edge (a, b) depends on this overlap:

$$P(A[a,b]=1) \propto S[a,b]$$

A threshold determines edge existence based on node similarity measures from training data.

Local Overlap Measures:

Local overlap statistics depend on the number of shared neighbors:

$$|M(a) \cap M(b)|$$

One such measure is the Sorenson index:

$$S_{\text{Sorenson}}[a,b] = \frac{2\,|M(a) \cap M(b)|}{d_a + d_b}$$

which normalizes by the sum of node degrees to prevent bias toward high-degree nodes. Other measures include:

Salton index:

$$S_{\text{Salton}}[a,b] = \frac{2\,|M(a) \cap M(b)|}{\sqrt{d_a d_b}}$$

Jaccard index:

$$S_{\text{Jaccard}}[a,b] = \frac{|M(a) \cap M(b)|}{|M(a) \cup M(b)|}$$

which account for overlap while reducing degree bias. Some measures weigh shared neighbors differently:

Resource Allocation (RA) index:

$$S_{\text{RA}}[b_1,b_2] = \sum_{a \in M(b_1) \cap M(b_2)} \frac{1}{d_a}$$

Adamic-Adar (AA) index:

$$S_{\mathrm{AA}}[b_1, b_2] = \sum_{a \in M(b_1) \cap M(b_2)} \frac{1}{\log(d_a)}$$

Both emphasize shared low-degree neighbors, often more informative.

Different similarity measures are suited for various networks:

- Dense Graphs or Homogeneous Networks: Jaccard and Salton indices offer balanced connectivity measures.
- Sparse Graphs or Social Networks: The Sorenson-Dice index emphasizes meaningful shared connections.
- Biological Networks or Complex Systems: RA and AA indices highlight hidden connections.
- Recommender Systems: The Adamic-Adar index balances common neighbors while adjusting for node popularity.

Global Overlap Measures:

Local overlap measures serve as effective heuristics for link prediction and often rival deep learning methods. However, they focus only on local node neighborhoods, which limits their applicability.

For example, two nodes might lack local overlap but still belong to the same community. Global overlap statistics address such broader relationships.

Katz Index:

The Katz index is a fundamental global overlap measure that counts paths of all lengths between node pairs:

$$S_{\text{Katz}}[a, b] = \sum_{j=1}^{\infty} \beta^j A^j[a, b]$$

where $\beta \in \mathbb{R}^+$ controls the weight given to shorter versus longer paths. Choosing $\beta < 1$ downweights longer paths.

The Katz index can be seen as a geometric series of matrices, common in graph analysis. The following theorem provides a general result:

Theorem: Let Y be a real square matrix with largest eigenvalue λ_1. Then:

$$(J - Y)^{-1} = \sum_{j=0}^{\infty} Y^j \quad \text{if and only if} \quad \lambda_1 < 1 \text{ and } (J - Y) \text{ is invertible.}$$

Proof: Define $s_M = \sum_{j=0}^{M} Y^j$, then:

$$Y s_M = Y \sum_{j=0}^{M} Y^j = \sum_{j=1}^{M+1} Y^j$$

which leads to:

$$s_M - Y s_M = \sum_{j=0}^{M} Y^j - \sum_{j=1}^{M+1} Y^j$$

$$s_M(J - Y) = J - Y^{M+1}$$

$$s_M = (J - Y^{M+1})(J - Y)^{-1}$$

If $\lambda_1 < 1$, then $\lim_{M \to \infty} Y^M = 0$, leading to:

$$\lim_{M \to \infty} s_M = (J - Y)^{-1}$$

Thus, the Katz index is given by:

$$S_{\text{Katz}} = (J - \beta A)^{-1} - J$$

where $S_{\text{Katz}} \in \mathbb{R}^{|M| \times |M|}$ stores node similarity values.

Leicht, Holme, and Newman (LHN) Similarity:

The Katz index favors high-degree nodes due to their participation in many paths. The LHN similarity normalizes observed path counts by their expected values under a random model:

$$S_{\text{LHN}}[a, b] = \frac{A[a, b]}{\mathbb{E}[A[a, b]]}$$

Using the configuration model, which preserves degree distribution:

$$\mathbb{E}[A[a, b]] = \frac{d_a d_b}{2m}$$

where $m = |\epsilon|$ is the total edge count. This suggests edge likelihood between a and b scales with their degree product.

For paths of length 2:

$$\mathbb{E}[A^2[a, b]] = \frac{d_a d_b}{(2m)^2} \sum_{c \in M} (d_c - 1) d_c$$

which considers paths through intermediate nodes c, correcting for prior edge use.

For longer paths, expected counts become complex. Using the adjacency matrix's largest eigenvalue λ_1:

$$\mathbb{E}[A^j[a,b]] \approx \frac{d_a d_b \lambda_1^{j-1}}{2m}, \quad j > 2$$

The LHN index is thus:

$$S_{\text{LHN}}[a,b] = J[a,b] + \frac{2m}{d_a d_b} \sum_{j=0}^{\infty} \beta^j \lambda_1^{1-j} A^j[a,b]$$

where J is the identity matrix. The matrix solution is:

$$S_{\text{LHN}} = 2\beta m \lambda_1 D^{-1} \left(J - \frac{\beta}{\lambda_1} A \right)^{-1} D^{-1}$$

where D is a diagonal degree matrix. Unlike the Katz index, the LHN similarity gives high scores only to node pairs with more paths than expected, providing a more balanced similarity measure.

Random Walk-based Global Similarity Measures:

Instead of counting exact paths in the graph, we measure node similarity using random walks. A well-known approach is extitPersonalized PageRank, a variant of PageRank, which estimates the probability of a random walk from node a reaching node b.

We define the stochastic matrix P as:

$$P = AD^{-1}$$

where A is the adjacency matrix and D is the diagonal matrix of node degrees. The stationary distribution q_a is given by:

$$q_a = cPq_a + (1 - c)e_a$$

where e_a is a one-hot vector marking the starting node a, and $c \in (0, 1)$ is the restart probability, controlling how often the random walk returns to a.

Without restart ($c = 1$), the probabilities converge to eigenvector centrality. With restart, we obtain a node-specific importance measure as the walk consistently returns to a.

The closed-form solution is:

$$q_a = (1 - c)(I - cP)^{-1}e_a$$

where I is the identity matrix. The node-node similarity is then defined as:

$$S_{\text{RW}}[a, b] = q_a[b] + q_b[a]$$

This measure reflects the likelihood of reaching b from a via a random walk and vice versa.

2.3 Graph Laplacians and Spectral Methods

We examine clustering nodes and deriving low-dimensional embeddings using Laplacians, alternative representations of the adjacency matrix with useful mathematical properties.

Unnormalized Laplacians

The unnormalized Laplacian is:

$$L = D - A$$

where D is the degree matrix and A is the adjacency matrix.

- L is symmetric ($L^\top = L$) and positive semi-definite:

$$y^\top L y \geq 0, \quad \forall y \in \mathbb{R}^{|V|}$$

- For all $y \in \mathbb{R}^{|V|}$,

$$y^\top L y = \sum_{(a,b)\in\epsilon} (y[a] - y[b])^2$$

- L has $|V|$ non-negative eigenvalues:

$$0 = \lambda_{|V|} \leq \lambda_{|V|-1} \leq \cdots \leq \lambda_1$$

Theorem

The geometric multiplicity of eigenvalue 0 in L equals the number of connected components.

Proof

For an eigenvector e of eigenvalue 0,

$$e^\top L e = 0$$

implies:

$$\sum_{(a,b)\in\epsilon} (e[a] - e[b])^2 = 0$$

which means $e[a] = e[b]$ for all connected nodes. If the graph is fully connected, the eigenvector is a constant vector of ones. If there are K components, L is block diagonal:

$$L = \begin{bmatrix} L_1 & & \dots & \\ & L_2 & \dots & \\ \vdots & \vdots & \ddots & \vdots \\ & & \dots & L_K \end{bmatrix}$$

Each block contributes an eigenvector for eigenvalue 0, serving as an indicator for nodes in that component.

Normalized Laplacians

Common normalized variants include:

1. Symmetric normalized Laplacian:

$$L_{\text{sym}} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$$

2. Random walk Laplacian:

$$L_{\text{RW}} = D^{-1} L$$

Both retain key properties, but differ algebraically. The 0-eigenvalue theorem holds exactly for L_{RW} and with scaling for L_{sym}.

Graph Cuts and Clustering

Eigenvectors of Laplacians indicate connected components. We generalize this to fully connected graphs for clustering.

A cluster cut is defined using partitions $A_1, \dots, A_K$:

$$\text{cut}(A_1, \dots, A_K) = \frac{1}{2} \sum_{k=1}^{K} |\{(a, b) \in \epsilon : a \in A_k, b \in A'_k\}|$$

Minimizing the cut alone tends to create small clusters, so we use:

1. Ratio Cut:

$$\text{RatioCut}(A_1, \ldots, A_K) = \frac{1}{2} \sum_{k=1}^{K} \frac{|\{(a,b) \in \epsilon : a \in A_k, b \in A'_k\}|}{|A_k|}$$

2. Normalized Cut (NCut):

$$\text{NCut}(A_1, \ldots, A_K) = \frac{1}{2} \sum_{k=1}^{K} \frac{|\{(a,b) \in \epsilon : a \in A_k, b \in A'_k\}|}{\text{vol}(A_k)}$$

where $\text{vol}(A) = \sum_{a \in A} d_a$. The NCut balances clusters based on node volume.

Spectral Clustering Using Graph Laplacians

The second-smallest eigenvector of the graph Laplacian can be used to divide nodes into clusters. This idea extends to forming multiple clusters by considering the smallest K eigenvectors of the Laplacian. The general process is as follows:

1. Identify the K smallest eigenvectors (excluding the smallest one): $e_{|V|-1}, e_{|V|-2}, \ldots, e_{|V|-K}$.
2. Construct a matrix $W \in \mathbb{R}^{|V| \times (K-1)}$ where each column is an eigenvector from step 1.
3. Represent each node by the corresponding row in matrix W, i.e., $z_a = W[a]$ for each node $a \in V$.
4. Perform K-means clustering on the node embeddings z_a for all $a \in V$.

This method, initially explained for two clusters, can also be applied to the normalized Laplacian, and the approach is generalized for K clusters. Spectral clustering uses the graph Laplacian's spectrum to group nodes, providing a well-founded approximation for optimal clustering. It also has theoretical links to random walks on graphs and graph signal processing.

Heterogeneous Graphs

In heterogeneous graphs, nodes belong to different types, forming distinct groups:

$$V = V_1 \cup V_2 \cup \cdots \cup V_k, \quad V_i \cap V_j = \emptyset \quad \forall\, i \neq j$$

Edges follow specific rules based on node types. For instance, in a biomedical graph, drug and disease nodes connect via "treatment" edges, while "polypharmacy side-effects" edges link drug nodes.

Multiplex Graphs

A multiplex graph consists of k layers, each representing a unique relation. Every node appears in all layers, with intra-layer edges defining interactions within a layer and inter-layer edges linking the same node across layers.

For example, in a transportation network, nodes represent cities, layers correspond to transport modes (e.g., air, train), intra-layer edges connect cities via the same mode, and inter-layer edges allow switching transport modes within a city.

Chapter 3

Random-walk Based Graph Embedding and Node2Vec

Graph embedding techniques represent graph-structured data in low-dimensional spaces while preserving their structural properties. These embeddings enable efficient computation for downstream tasks such as node classification, clustering, and link prediction. Despite recent advances in representation learning, many methods struggle to capture the diverse connectivity patterns found in complex networks.

Node2vec addresses some of these challenges by mapping nodes into low-dimensional spaces using biased random walks. This flexible approach balances between preserving community structure (homophily) and capturing structural roles (structural equivalence), resulting in task-independent node representations that generalize well to various networks.

However, important questions remain. Existing methods often fail to explain how embeddings capture multi-scale structural information and how they can be optimally used for tasks like link prediction. Supervised methods require task-specific features that are costly to design, while unsupervised approaches, though scalable, often lack flexibility and generalization.

This chapter introduces a novel analytical framework for graph embedding to address these issues. The framework leverages random walks and consists of three components: a random walk strategy, a similarity metric, and an embedding generation algorithm. To-

gether, they offer a systematic method for learning embeddings that capture both local and global graph structures.

Additionally, we propose techniques to combine embeddings across multiple scales, improving performance in tasks such as link prediction. In particular, embeddings based on autocovariance similarity, combined with dot product ranking, outperform existing methods using Pointwise Mutual Information (PMI) similarity by up to 100%.

Method

Consider an undirected graph $G = (V, E)$ with n nodes and m edges. The graph is represented by a symmetric matrix $\mathbf{A}$, where $\mathbf{A}_{ab}$ denotes the weight of the edge between nodes a and b. If an edge exists, $\mathbf{A}_{ab}$ is positive; otherwise, it is zero. The degree of a node u is defined as the sum of the weights of its connected edges, given by $\deg(a) = \sum_v \mathbf{A}_{ab}$.

A node embedding is a function ϕ that maps each node b to a vector in $\mathbb{R}^d$. These embeddings are organized into a matrix $\mathbf{A}$, where each row corresponds to the embedding vector of a node. Some approaches also generate an additional matrix $\mathbf{b}$. In random-walk-based methods, embeddings are designed such that nodes with similar properties have closely aligned vectors, and their similarity is quantified by the dot product of these vectors.

3.1 Random Walk Process

A random walk on a graph can be modeled as a Markov chain, where the walker transitions between nodes B. The probability of moving from node a to node b depends solely on the walker's current position and is determined by the adjacency matrix $\mathbf{P}$. In a standard random walk, this transition probability is proportional to the edge weight $\mathbf{P}_{ab}$ and the degree of node a:

$$f(x(k+1) = b \mid x(k) = a) = \frac{\mathbf{P}_{ab}}{\deg(a)}$$

where $x(k)$ represents the walker's position at time k. These transition probabilities are

collectively represented in a transition matrix $\mathbf{T}$:

$$\mathbf{T} = \mathbf{D}^{-1}\mathbf{P}$$

where $\mathbf{D}$ is the degree matrix of the graph. For a connected, non-bipartite graph, the random walk converges to a unique stationary distribution $\boldsymbol{\pi}$:

$$\pi_a = \frac{\deg(a)}{\sum_b \deg(b)}$$

Standard random walks are effective for exploring neighborhoods in undirected connected graphs. For directed graphs, PageRank is often employed instead of standard random walks to ensure the process reaches a stable state.

Similarity Function

A node similarity metric is a function $\varphi : V \times V \to \mathbb{R}$ that quantifies the similarity between two nodes based on their topological properties. High positive values indicate strong similarity, while high negative values suggest dissimilarity.

Random-walk-based similarity metrics utilize the likelihood of a walker transitioning between pairs of nodes. Random walks are advantageous because they can capture similarities at multiple scales, such as local or global levels. This is achieved through a Markov time parameter τ, which controls the number of steps in the walk. This paper investigates how varying Markov time scales influence embeddings and how they can be leveraged to enhance performance across different tasks.

Two random-walk-based similarity functions are discussed: Pointwise Mutual Information (PMI) and Autocovariance. PMI is widely used in graph embeddings and forms the basis of methods like word2vec and DeepWalk. Autocovariance, on the other hand, is commonly employed for detecting communities at multiple scales. The paper further demonstrates the effectiveness of autocovariance-based embeddings for tasks involving edge-level details.

Pointwise Mutual Information (PMI)

Let $Y_b(k) \in \{0, 1\}$ be an indicator variable representing whether the walker is at node b at time k. The Pointwise Mutual Information (PMI) between being at node a at time k and being at node b at time $k + \tau$ is defined as:

$$W_{ab}(\tau) = \text{PMI}(Y_a(k) = 1, Y_b(k + \tau) = 1) = \log \frac{q(Y_a(k) = 1, Y_b(k + \tau) = 1)}{q(Y_a(k) = 1)p(Y_b(k + \tau) = 1)}$$

PMI provides a non-linear perspective on node proximity based on random walks. For a random walk that converges to a steady state with distribution $\boldsymbol{\pi}$, PMI can be computed using $\boldsymbol{\pi}$ and the transition probability over τ steps:

$$W_{ab}(\tau) = \log(\pi_a q(y(k + \tau) = b \mid y(k) = a)) - \log(\pi_a \pi_b)$$

Here, $W_{ab}(\tau)$ ranges from $-\infty$ to $-\log(\pi_b)$. In matrix form, the PMI matrix is expressed as:

$$W(\tau) = \log(\mathbf{\Pi M}^\tau) - \log(\boldsymbol{\pi}\boldsymbol{\pi}^T)$$

where $\mathbf{\Pi} = \text{diag}(\boldsymbol{\pi})$ and $\log(\cdot)$ is applied element-wise. The PMI matrix is symmetric due to the reversibility of undirected random walks. Methods such as LINE and DeepWalk, which utilize random walks for embeddings, effectively incorporate PMI into their models:

$$W_{\text{LINE}} = W(1) - \log b$$

$$W_{\text{DW}} = \log \left(\frac{1}{T} \sum_{\tau=1}^{T} \exp(W(\tau)) \right) - \log b$$

where b is the number of negative samples and T is the context window size. LINE

directly employs PMI at Markov time 1, while DeepWalk averages PMI matrices from time 1 to T to achieve a smoother approximation. Detailed proofs are provided in the Appendix.

Autocovariance

Autocovariance measures the joint variability of $Y_a(k)$ and $Y_b(k+\tau)$ over a time interval τ:

$$W_{ab}(\tau) = \text{cov}(Y_a(k), Y_b(k+\tau)) = \mathbb{E}[(Y_a(k) - \mathbb{E}[Y_a(k)])(Y_b(k+\tau) - \mathbb{E}[Y_b(k+\tau)])]$$

This metric quantifies how the probabilities of the walker visiting nodes a and b co-vary over time τ. For a random walk with steady-state probabilities $\boldsymbol{\pi}$:

$$W_{ab}(\tau) = \pi_a p(y(k+\tau) = b \mid y(k) = a) - \pi_a \pi_b$$

where $W_{ab}(\tau)$ ranges from $-\pi_a\pi_b$ to $\pi_a(1-\pi_b)$. In matrix form, it is given by:

$$W(\tau) = \boldsymbol{\Pi}\mathbf{N}^\tau - \boldsymbol{\pi}\boldsymbol{\pi}^T$$

The autocovariance matrix resembles the PMI matrix but omits the logarithmic transformation. This distinction is significant for graph embeddings: PMI is a non-linear metric associated with the sigmoid function commonly used in deep learning, whereas autocovariance is linked to a piecewise linear function.

Embedding Algorithm

The goal of an embedding algorithm is to generate vector representations that align with a specified similarity metric. The optimization problem is formulated as:

$$\mathbf{V}^* = \arg\min_{\mathbf{V}} \sum_{a,b} (\mathbf{v}_a^T \mathbf{v}_b - W_{ab})^2 = \arg\min_{\mathbf{V}} \|\mathbf{V}\mathbf{V}^T - \mathbf{W}\|_F^2$$

Here, $\mathbf{v}_a^T \mathbf{v}_b$ represents the similarity between nodes in the embedding space, $\mathbf{W}$ is the similarity matrix, and $\|\cdot\|_F$ denotes the Frobenius norm. The following sections will discuss two techniques to solve this embedding problem.

Matrix Factorization

Matrix factorization is a technique used to find optimal vector representations by approximating $\mathbf{R}$ with $\mathbf{V}\mathbf{V}^T$, where $\text{rank}(\mathbf{V}\mathbf{V}^T) = d$. Since $\mathbf{R}$ is symmetric, the optimal solution $\mathbf{V}^*$ can be obtained using Singular Value Decomposition (SVD). According to the Eckart-Young-Mirsky theorem, if $\mathbf{W} = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^T$ is the SVD of $\mathbf{W}$, then:

$$\mathbf{V}^* = \mathbf{S}_d \sqrt{\mathbf{\Lambda}_d}$$

Unlike traditional spectral methods, factorization does not rely on the graph Laplacian. Direct SVD of $\mathbf{W}$ has a computational complexity of $O(n^3)$, which is impractical for most scenarios. However, for sparse graphs, scalable factorization techniques can be employed. For autocovariance, the Lanczos method can be applied, reducing the complexity to $O(nd^2 + md\tau)$ by leveraging sparse matrix-vector multiplications. For PMI, a spectral sparsifier of the similarity matrix can be constructed, and Randomized SVD can be utilized, achieving a complexity of $O(em\tau \log n + emd + nd^2 + d^3)$, where $em = O(m\tau)$ represents the number of non-zero entries in the sparsifier.

Sampling

Sampling-based methods generate embeddings by maximizing the likelihood of samples obtained from a random-walk process. Each sample i consists of a sequence $\langle b_1^{(i)}, b_2^{(i)}, \ldots, b_L^{(i)} \rangle$ of length L, where the initial node $b_1^{(i)}$ follows a distribution $p(b)$, assumed here as π_b. From each sample, pairs $(b_s^{(i)}, b_{s+\tau}^{(i)})$ are extracted, forming a multiset D of node pairs (a, b).

Unlike matrix factorization, sampling methods produce two matrices, $\mathbf{A}$ and $\mathbf{B}$, representing source and target embeddings. This approach, introduced by word2vec, is more efficient. We focus on negative sampling, which uses t negative samples to enhance embedding quality. Let z be a binary variable indicating whether $(a, b) \in D$. The log-likelihood of the corpus D is given by:

$$\ell = \sum_{a,b} \#(a, b) \left(\log(p(z = 1|a, b)) + t\mathbb{E}_{w\sim\pi} \log(p(z = 0|a, w)) \right)$$

where $\#(a, b)$ is the count of pair (a, b) in D. Using PMI, the conditional probability is modeled via the sigmoid function: $p(z = 1|a, b) = \sigma(\mathbf{a}_a^T \mathbf{b}_b)$ and $p(z = 0|a, b) = \sigma(-\mathbf{a}_a^T \mathbf{b}_b)$. For autocovariance, we define:

$$p(z = 1|a, b) = \rho\left(\frac{\mathbf{a}_a^T \mathbf{b}_b + \pi_a \pi_b}{\mathbf{a}_a^T \mathbf{b}_b + (t + 1)\pi_a \pi_b} \right)$$
$$p(z = 0|a, b) = \rho\left(\frac{\pi_a \pi_b}{\mathbf{a}_a^T \mathbf{b}_b + (t + 1)\pi_a \pi_b} \right)$$

where $\rho(x)$ is piecewise linear: 0 if $x < 0$, x if $0 \leq x \leq 1$, and 1 otherwise.

Theorem 2.1. For sufficiently large dimensionality d ($d = \Omega(n)$), the embedding maximizing likelihood satisfies:

$$\mathbf{a}_a^T \mathbf{b}_b = \frac{1}{t}(\pi_a p(x(s + \tau) = b \mid x(s) = a) - \pi_a \pi_b)$$

Distributed Representations of Words and Phrases

The Skip-gram model efficiently learns high-quality vector representations, capturing both syntactic and semantic relationships. The referenced paper introduces improvements enhancing vector quality and training efficiency, including subsampling frequent words for speedup and negative sampling as a simple alternative to hierarchical softmax.

3.2 The Skip-gram Model

The objective is to find word representations useful for predicting nearby words. Given a sequence $w_1, w_2, \ldots, w_U$, the goal is to maximize:

$$\frac{1}{U}\sum_{r=1}^{U}\sum_{-c\leq j\leq c, j\neq 0}\log p(v_{r+j}|v_r)$$

where c is the context window size. Larger c improves accuracy but increases training cost. The probability is defined via softmax:

$$p(v_P|v_I) = \frac{\exp({\mathbf{b}'_{v_P}}^\top \mathbf{b}_{v_I})}{\sum_{v=1}^{V}\exp({\mathbf{b}'_v}^\top \mathbf{b}_{v_I})}$$

where $\mathbf{b}_v$ and $\mathbf{b}'_v$ are input and output vector representations. However, computing gradients is costly as it scales with vocabulary size V, often 10^5 to 10^7.

Hierarchical Softmax

Hierarchical softmax approximates softmax efficiently by reducing the number of evaluations from V to approximately $\log_2(V)$. It represents output words as leaves in a binary tree, where each inner node defines conditional probabilities for its children.

For a word v, let $n(v,k)$ be the k-th node on its path from root, with $L(v)$ denoting path length. The probability is given by:

$$p(v|v_I) = \prod_{j=1}^{L(w)-1}\sigma\left(n(v,k+1) = \mathrm{ch}(n(v,j))\cdot {\mathbf{b}'_{n(v,j)}}^\top \mathbf{b}_{v_I}\right)$$

where $\sigma(x) = 1/(1+\exp(-x))$. It follows that:

$$\sum_{v=1}^{V} p(v|v_I) = 1.$$

This reduces computation from $O(V)$ to $O(\log V)$. Unlike standard Skip-gram, which assigns two representations per word, hierarchical softmax assigns one to words and one to inner nodes.

Tree structure significantly affects efficiency. Mnih and Hinton explored various tree constructions; here, a binary Huffman tree is used, assigning shorter codes to frequent words, improving training speed.

Negative Sampling

Negative Sampling (NEG) is an alternative to hierarchical softmax, based on Noise Contrastive Estimation (NCE) in language modeling. NCE posits that a good model should distinguish real data from noise using logistic regression, favoring real data.

While NCE approximately maximizes the softmax log probability, the Skip-gram model prioritizes learning quality vector representations. Thus, NCE can be simplified without compromising vector quality. The objective function for NEG is:

$$\log \sigma({\mathbf{v}'_{v_r}}^\top \mathbf{v}_{v_I}) + \sum_{i=1}^{k} \mathbb{E}_{v_i \sim P_n(v)} \left[\log \sigma(-{\mathbf{v}'_{v_i}}^\top \mathbf{v}_{v_I}) \right]$$

where $\sigma(x) = \frac{1}{1+\exp(-x)}$ is the sigmoid function. The first term models the probability of the target word v_r, while the summation represents negative samples from noise distribution $P_n(v)$. The objective is to distinguish v_r from noise using logistic regression with k negative samples per data point.

Experiments suggest setting k between 5–20 for small datasets and 2–5 for large ones. Unlike NCE, NEG does not require explicit probabilities of the noise distribution, only samples. Moreover, while NCE maximizes softmax log probability, this is less crucial for the

Skip-gram model.

Both methods depend on a noise distribution $P_n(v)$. Empirical results show that raising the unigram distribution to the $\frac{3}{4}$ power (i.e., $U(v)^{3/4}/Z$) outperforms unigram and uniform distributions in all tested tasks.

Subsampling of Frequent Words

Large corpora contain frequent words like "in," "the," and "a," which appear millions of times but carry little semantic value. For instance, the Skip-gram model benefits from co-occurrences like "France" and "Paris" but gains little from "France" and "the" since "the" appears with almost every word. Moreover, frequent words' vector representations stabilize after limited training.

To address this, we apply subsampling, discarding each word v_i with probability:

$$P(v_i) = 1 - \sqrt{\frac{l}{f(v_i)}}$$

where $f(v_i)$ is the word frequency and l is a threshold, typically 10^{-5}. This method aggressively down-samples high-frequency words while preserving frequency rank order. Though heuristically derived, it proves highly effective. Subsampling speeds up learning and enhances rare word representations.

Empirical Results

Negative Sampling consistently outperforms Hierarchical Softmax on analogical reasoning tasks and slightly surpasses Noise Contrastive Estimation (NCE). Additionally, subsampling significantly accelerates training and improves rare word representations.

3.3 Feature Learning Framework

Feature learning in networks is framed as a maximum likelihood optimization problem for any network, whether directed or undirected, weighted or unweighted. The objective is to map nodes to feature representations for use in downstream prediction tasks. This mapping, represented as a matrix g of size $|V| \times d$, maximizes the log-probability of observing a network neighborhood $NS(a)$ for a node a based on its feature representation:

$$\max_{g} \sum_{a \in V} \log \Pr(NS(a) \mid g(a)).$$

Two assumptions simplify the optimization:

1. **Conditional Independence:** The likelihood of observing each neighborhood node is independent, given by:

$$\Pr(NS(a) \mid g(a)) = \prod_{m_i \in NS(a)} \Pr(m_i \mid g(a)).$$

2. **Symmetry in Feature Space:** Nodes influence each other symmetrically, modeled as:

$$\Pr(m_i \mid g(a)) = \frac{\exp(g(m_i) \cdot g(a))}{\sum_{b \in V} \exp(g(b) \cdot g(a))}.$$

With these assumptions, the objective simplifies to:

$$\max_{g} \sum_{a \in V} \left(-\log Z_a + \sum_{m_i \in NS(a)} g(m_i) \cdot g(a) \right),$$

where the per-node partition function Z_a is approximated using negative sampling:

$$Z_a = \sum_{b \in V} \exp(g(a) \cdot g(b)).$$

The optimization problem is solved using stochastic gradient ascent. Inspired by the Skip-gram model in natural language processing, this approach requires a richer notion of neighborhood for networks, achieved through a randomized procedure that samples diverse neighborhoods for each node.

Classic Search Strategies

The process of generating a neighborhood $NS(u)$ for a source node u involves sampling sets of k nodes, where k is a predefined size. Two main sampling strategies are explored:

- **Breadth-first Sampling (BFS):** This strategy samples immediate neighbors of the source node u. It provides a microscopic view of the neighborhood, emphasizing structural equivalence, where nodes with similar roles in the network are embedded closely together.

- **Depth-first Sampling (DFS):** DFS samples nodes sequentially at increasing distances from the source node u. It explores a larger part of the network, providing a macroscopic view of the neighborhood that is crucial for inferring communities based on homophily, where nodes within the same community are interconnected.

These sampling strategies influence the learned representations differently:

- BFS tends to capture structural equivalence by focusing on nearby nodes that repeat in the sampled neighborhoods, reducing variance.

- DFS explores larger network regions, capturing macroscopic views that are essential for understanding homophily but can introduce higher variance due to distant nodes sampled.

Both strategies play a significant role in determining the nature of learned representations, reflecting the dual aspects of homophily and structural equivalence observed in real-world networks.

3.4 Node2vec

To improve neighborhood sampling beyond the limitations of BFS and DFS, we introduce a flexible biased random walk strategy. This approach allows us to smoothly transition between exploring neighborhoods in a BFS or DFS fashion.

Random Walks:

- **Fixed-Length Random Walk:** Starting from a source node u, nodes c_i are chosen sequentially over a fixed length l based on transition probabilities ρ_{vx}.
- **Search Bias ρ:** The random walk can be influenced by systemic biases p and q to guide transitions between nodes v and x, combining edge weights w_{vx} with the bias terms. This combination is crucial for exploring diverse network neighborhoods, accommodating both structural equivalence and homophily.

Equations:

- The unnormalized transition probability ρ_{wy} is:

$$\rho_{wy} = \begin{cases} w_{wy} & \text{if } (w, y) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

  where w_{wy} is the edge weight between nodes w and y.

- A 2nd-order random walk with parameters a and b modifies the transition probability:

$$\rho_{wy} = \rho'_{wy} \cdot w_{wy},$$

  where ρ'_{wy} is adjusted based on a and b:

$$\rho'_{wy} = \begin{cases} \frac{1}{a} & \text{if } d_{tty} = 0, \\ 1 & \text{if } d_{tty} = 1, \\ \frac{1}{b} & \text{if } d_{tty} = 2. \end{cases}$$

Here, d_{tty} is the shortest path distance between nodes t and y.

This method balances local structure with broader network exploration.

Parameter a: Controls the likelihood of revisiting a node. A high a ($a > \max(b, 1)$) reduces backtracking and increases exploration. The effective complexity is

$$O\left(\frac{l}{k(l-k)}\right)$$

per sample. For instance, in Figure 1, a random walk $u, t4, t5, t6, t8, t9$ of length $l = 6$ gives:

$$NT(u) = \{t4, t5, t6\}, \quad NS(t4) = \{t5, t6, t8\}, \quad NT(t5) = \{t6, t8, t9\}.$$

Sample reuse may introduce bias but enhances efficiency by reducing redundancy in 2-hop sampling. A low a ($a < \min(b, 1)$) leads to backtracking (Figure 2), keeping the walk near u.

Parameter b: Differentiates between "inward" and "outward" nodes. If $b > 1$, the walk stays near t, resembling BFS locality sampling. If $b < 1$, it prefers distant nodes, akin to DFS exploration. Since ρ_{wy} depends on the previous node t, the walk follows a 2nd-order Markov process.

The graph2vec algorithm

Algorithm: The graph2vec algorithm

GenerateEmbeddings (Graph $H = (\mathcal{N}, \mathcal{E}, \mathcal{W})$, Embedding size d, Walks per node γ, Walk length λ,

$\theta = \text{ComputeTransitionWeights}(H; \alpha; \beta)$

$H' = (\mathcal{N}, \mathcal{E}, \theta)$

Initialize trajectories as Empty

for iter = 1 to γ do

for all nodes $v \in \mathcal{N}$ do

path = graph2vecWalk($H'; v; \lambda$)

Append path to trajectories

ϕ = OptimizeEmbeddings($\kappa, d,$ trajectories)

return ϕ

graph2vecWalk (Graph $H' = (\mathcal{N}, \mathcal{E}, \theta)$, Start node v, Path length λ)

Initialize path as [v]

for step = 1 to λ do

current = path[-1]

$\mathcal{N}_{\text{current}}$ = FetchNeighbors($current, H'$)

w = SampleNeighbor($\mathcal{N}_{\text{current}}, \theta$)

Append w to path

return path

The pseudocode for node2vec, given in Algorithm, involves offsetting bias from the start node u by simulating r random walks of length l for each node. During these walks, nodes are sampled using precomputed transition probabilities ρ_{vx} for a 2nd order Markov chain, ensuring efficient $O(1)$ time complexity for node selection using alias sampling. The algorithm's three phases—preprocessing to compute transition probabilities, random walk simulations, and SGD-based optimization—are executed sequentially and can be parallelized for scalability.

Learning Edge Features

The `graph2vec` algorithm provides a semi-supervised approach to learning expressive node representations. However, many tasks, such as link prediction, require modeling relationships between node pairs rather than individual nodes. Given that random walks naturally capture network connectivity, we extend them to pairs of nodes using a bootstrapping approach over individual node embeddings.

For two nodes u and v, we define a binary operator δ that combines their feature vectors $\mathbf{h}(u)$ and $\mathbf{h}(v)$ into a joint representation $\mathbf{g}(u, v)$, where $\mathbf{g} : V \times V \to \mathbb{R}^{d_0}$ and d_0 is the embedding dimension of the node pair. The operator should generalize to all node pairs, including those without existing edges, ensuring applicability to link prediction, where both true and false edges appear in the test set.

Multi-label Classification

For link prediction, we construct a dataset by randomly removing 50% of edges while preserving network connectivity. Negative samples are generated by selecting an equal number of node pairs without direct edges.

Since existing feature learning algorithms have not been directly applied to link prediction, we compare `graph2vec` against heuristic scores based on node neighborhoods. The authors evaluated these methods on:

- **Facebook:** A user network with 4,039 nodes and 88,234 edges, where edges represent friendships.
- **Protein-Protein Interaction (PPI):** A Homo Sapiens PPI network with 19,706 nodes and 390,633 edges, where edges denote biological interactions.
- **arXiv ASTRO-PH:** A collaboration network of 18,722 scientists, with 198,110 edges representing co-authorship.

Experimental Results

The optimal parameters α and β for `graph2vec` are omitted for clarity. Across datasets, learned representations significantly outperform heuristic baselines. Notably, on arXiv ASTRO-PH, `graph2vec` improves AUC by 12.6% over Adamic-Adar.

Among feature learning methods, `graph2vec` consistently outperforms DeepWalk and LINE, achieving AUC gains of up to 3.8% and 6.5%, respectively, with the best choice of binary operators. While Weighted-L1 and Weighted-L2 occasionally favor LINE, the Hadamard operator with `graph2vec` is the most stable and performs best on average.

Chapter 4

Multi-relational Data and Knowledge Graphs

4.1 Knowledge Graph Completion

Knowledge graphs are multi-relational graphs $G = (V, E)$, where edges are tuples $e = (a, \tau, b)$. Each edge describes a specific relation $\tau \in \mathcal{T}$ between two nodes a and b. These graphs are called knowledge graphs because the tuples express factual relationships between nodes. For instance, in a biomedical knowledge graph, an edge (a, TREATS, b) could indicate that the drug (node a) treats a disease (node b).

The primary objective in knowledge graph completion is to predict missing edges or relations, although node classification tasks may also be performed using multi-relational graphs.

Reconstructing Multi-Relational Data

Embedding knowledge graphs can be treated as a reconstruction problem. The goal is to reconstruct the relationships between nodes based on their embeddings p_a and p_b. The challenge lies in handling multiple edge types (relations), so the decoder must account for

both node embeddings and relation types.

The multi-relational decoder is defined as:

$$\text{dec} : \mathbb{R}^d \times \mathcal{R} \times \mathbb{R}^d \to \mathbb{R}^+$$

The output $\text{dec}(p_a, \tau, p_b)$ represents the likelihood that the relation τ exists between nodes a and b.

A simple example of a multi-relational embedding model is RESCAL, where the decoder is defined as:

$$\text{dec}(a, \mathcal{R}, b) = p_a^\top \mathcal{R}_\tau p_b$$

Here, $\mathcal{R}_\tau \in \mathbb{R}^{d \times d}$ is a learnable matrix specific to relation τ. The embedding matrix P and relation matrices $\mathcal{R}_\tau$ are trained by minimizing a reconstruction loss:

$$L = \sum_{a \in V} \sum_{b \in V} \sum_{\tau \in \mathcal{R}} \left\| p_a^\top \mathcal{R}_\tau p_b - A[a, \mathcal{R}_\tau, b] \right\|^2$$

where $A \in \mathbb{R}^{|V| \times |\mathcal{R}| \times |V|}$ is the adjacency tensor of the multi-relational graph. Optimizing this loss is essentially performing tensor factorization.

4.2 Loss Functions, Decoders, and Similarity Functions

The decoder provides a score for node embeddings, the similarity function defines the node-node relationship being decoded, and the loss function measures the difference between the decoded output and the actual similarity.

In multi-relational settings, various decoders and loss functions are used, and most methods define similarity based on the adjacency tensor. These methods aim to reconstruct the immediate multi-relational neighbors from low-dimensional embeddings because defin-

ing higher-order relationships in multi-relational graphs is challenging. The primary goal of these methods is relation prediction.

Loss Functions

The key components of a multi-relational node embedding method are the **decoder** and the **loss function**. The goal is to decode the adjacency tensor from low-dimensional node embeddings, but simple reconstruction loss has some drawbacks.

Drawbacks of Simple Reconstruction Loss

- **Computationally expensive**: The nested sums in reconstruction loss require $O(|V|^2|\mathcal{R}|)$ operations. For large graphs, this is not feasible.
- **Not suited for binary comparison**: Since the adjacency tensor usually contains binary values, mean-squared error (MSE) is not ideal. MSE is more appropriate for regression, whereas edge prediction is more like a classification problem.

Cross-Entropy with Negative Sampling

A more efficient loss for multi-relational graphs is **cross-entropy loss with negative sampling**. It is formulated as:

$$L = \sum_{(a,\tau,b)\in E} -\log(\sigma(\text{dec}(p_a, \tau, p_b))) - \gamma \mathbb{E}_{b_n \sim P_{n,a}(V)} \left[\log(\sigma(-\text{dec}(p_a, \tau, p_{b_n})))\right]$$

Where:

- σ is the logistic function,
- $P_{n,a}(V)$ is the negative sampling distribution over nodes V,
- γ is a hyperparameter.

This loss is similar to *node2vec* but applies to general multi-relational decoders. The goal is to predict whether an edge exists between two nodes using both positive and negative examples:

- **Positive examples**: Existing edges, for which we want the score to be high.
- **Negative examples**: Randomly chosen pairs that are not connected, for which we want the score to be low.

The cross-entropy loss measures the log-likelihood of correctly predicting the existence (or non-existence) of an edge. The expectation over negative samples is estimated using a Monte Carlo approximation. A more compact form of the loss is:

$$L = \sum_{(a,\tau,b)\in E} \left(-\log(\sigma(\text{dec}(p_a, \tau, p_b))) - \sum_{b_n \in P_{n,a}} \log(\sigma(-\text{dec}(p_a, \tau, p_{b_n}))) \right)$$

Max-Margin Loss

Another popular loss for multi-relational embeddings is the **max-margin loss** (also called hinge loss):

$$L = \sum_{(a,\tau,b)\in E} \sum_{b_n \in P_{n,a}} \max(0, -\text{dec}(p_a, \tau, p_b) + \text{dec}(p_a, \tau, p_{b_n}) + \Delta)$$

Here, we are comparing the score of a true pair with a negative pair using *contrastive estimation*. If the score for the true pair is larger by at least Δ (the margin), the loss is zero. Otherwise, the loss encourages increasing the score difference. The max-margin loss compares the direct output of the decoders rather than treating the task as binary classification.

Chapter 5

An overview of Molecular Docking

Molecular docking is a computational technique that predicts the preferred binding orientation of a ligand to a receptor, forming a stable complex. This helps estimate key parameters such as binding free energy, affinity, and stability using scoring functions.

Docking is popular in discovery of drugs to predict how small molecules interact with biological targets like proteins, carbohydrates, and nucleic acids, aiding structure-based drug design. The goal is to find an optimized conformation that minimizes the system's free energy (ΔG_{bind}), which includes van der Waals (ΔG_{vdw}), hydrogen bonding (ΔG_{hbond}), desolvation (ΔG_{desolv}), electrostatics (ΔG_{elec}), torsional energy (ΔG_{tor}), total internal energy (ΔG_{total}), and unbound system energy (ΔG_{unb}).

Successful docking requires structural databases for target identification and computational tools for ligand evaluation. Various algorithms rank ligands based on their interactions, streamlining drug discovery.

5.1 Computational Approaches

In silico methods must be fast and robust for effective target recognition. Docking-based virtual screening prioritizes active candidates from large molecular databases, aiding drug discovery. Significant progress has been made in scoring and docking protocols, improv-

ing receptor-ligand docking predictions. This section outlines key computational coupling approaches, their types and applications.

Molecular Docking Approaches

Molecular docking follows two primary approaches:

Simulation Approach: This method separates ligand and target molecules, allowing the ligand to explore conformational space before binding to the target pocket. Moves involve internal (torsional) or external (rotational/translational) adjustments, with each pose evaluated based on total system energy. While accurate, this approach is computationally intensive. Grid-based tools and optimization methods have improved efficiency.

Shape Complementarity Approach: Here, docking is based on surface structural matching between ligand and target. The target's solvent-accessible surface is compared with the ligand's shape, enabling rapid scanning of thousands of molecules in seconds. Hydrophobicity considerations further refine docking accuracy, making this approach fast and effective.

Types of Docking

Docking simulations employ various search algorithms such as genetic, fragment-based, Monte Carlo, and molecular dynamics algorithms;high-throughput tools like DOCK, GOLD, FlexX, and ICM are widely used. Depending on the simulation's objective, docking can be:

- Flexible ligand docking: The ligand is flexible, while the target remains rigid.
- Rigid body docking: Both ligand and target are rigid.
- Flexible docking: Both molecules are flexible, capturing complex interactions more realistically.

Applications of Molecular Docking

Molecular docking is a crucial tool in modern research, providing insights before experimental validation. It plays a key role in drug discovery by predicting interactions between small molecules (ligands) and target proteins, aiding in enzyme activation or inhibition. Major applications include:

Lead Optimization

Docking predicts optimal ligand orientations and binding modes within target molecules, guiding the development of more potent, selective, and efficient drug analogs.

Hit Identification

By integrating docking with scoring functions, large molecular databases can be screened *in silico* to identify potent drug candidates for specific targets.

Drug-DNA Interaction Studies

Chemotherapeutic drugs often target nucleic acids, but their cytotoxic mechanisms remain unclear. Molecular docking helps predict drug binding to DNA, aiding in understanding anticancer mechanisms. Insights from these studies establish correlations between molecular structure and cytotoxicity, informing rational drug design with improved efficacy and reduced side effects.

Graph-based Molecular Generation

Motif-based methods treat molecules as collections of functional substructures, ensuring chemical validity while generating new molecules. Jin et al. employ a variational autoencoder (VAE) to construct molecular graphs from motif-based scaffolds, later integrating adversarial training for diverse outputs. Hierarchical encoders extend this to polymers, while

reinforcement learning framework optimize pharmacochemically relevant fragments. Jin et al. introduce multi-property rationales, further refined by an Expectation-Maximization algorithm. Other works leverage Markov Chain Monte Carlo sampling, differentiable optimization, and neural network-based molecule edits.

Non-motif approaches generate molecules atom-by-atom. Zhou et al. apply double Q-learning, avoiding pre-training constraints. Shi et al. develop a flow-based autoregressive model for parallelized structure generation. Bayesian optimization and evolutionary algorithms further refine molecule discovery.

For 3D molecule generation, conditional VAEs and diffusion models predict molecular conformations. Energy-based methods optimize 3D structures via gradient descent, while autoregressive flow models ensure geometric consistency. These advancements improve molecular design by integrating graph-based learning with geometric and probabilistic modeling.

5.2 Graph-based Approaches

Graph Convolutional Networks (GCNs)

Graph Convolutional Networks (GCNs) serve as a baseline for molecular representation learning due to their efficiency and scalability. These models learn node states by aggregating messages from neighboring nodes. A hierarchical GCN model enhances molecular graph representations by preserving spatial information through radial basis function layers, enabling robust distance tensors for effective atom-wise and pair-wise interactions.

Pre-training Strategies for GNNs

Labeled molecular datasets are often limited, making supervised training prone to overfitting. Pre-training strategies leverage large-scale unlabeled data to enhance generalization. The key challenge lies in selecting effective tasks that align with downstream objectives. Improper task selection can lead to negative transfer, reducing performance on specific tasks.

Contrastive Learning for Molecular Graphs

Graph contrastive learning has emerged as an effective GNN pre-training approach. GraphCL maximizes mutual information between augmented graph representations, improving feature extraction. Wang et al. extend this idea with MolCLR, introducing molecular-specific augmentations such as bond deletions to capture chemical interactions more effectively.

Despite these advancements, the effectiveness of data augmentation remains debated. Li et al. argue that random atomic modifications can disrupt molecular structures. Alternative methods, such as the N-gram graph representation, aim to preserve molecular integrity by embedding short walks within graphs.

Altae et al. propose a hybrid architecture combining iterative refinement with GCNs, enabling one-shot learning to reduce data dependency in molecular property predictions.

Chapter 6

P2Rank: Machine Learning Based Tool for Prediction of Ligand Binding Sites from Protein Structure

6.1 Background and Motivation

Ligand Binding Site (LBS) prediction plays a key role in understanding protein function and drug discovery, including prediction of drug side effects, prioritization of docking and virtual screening. LBS prediction is a fast, standalone, and user-friendly tool that does not rely on large template libraries of known protein-ligand complexes.

Existing Approaches

LBS prediction methods employ various algorithmic strategies, including geometric, energetic, conservation-based, template-based, and machine learning approaches. Many state-of-the-art methods combine these strategies for enhanced accuracy. Below are some notable tools:

- **Geometric Methods: Fpocket** is a widely used geometric approach utilizing Voronoi

tessellation. It efficiently identifies binding sites but produces many candidates without optimal ranking.

- **Consensus-Based Methods: MetaPocket 2.0** combines predictions from multiple algorithms, selecting the top-ranked binding sites from each tool, improving accuracy over individual methods.
- **Energetic Methods: SiteHound** uses a probe-grid system to estimate interaction energies around the protein surface. It is available both as a web server and a stand-alone tool.
- **Machine Learning-Based Methods:DeepSite** employs deep learning on a voxelized 3D space to predict binding sites but is only accessible as a web server.

Challenges and Future Directions

While reported identification success rates for these tools are high, independent benchmarks indicate that many methods fail to generalize well to unseen proteins. This highlights the need for more accurate and robust approaches for LBS prediction.

6.2 Stand-alone Tools vs Web Servers

Many ligand binding site (LBS) prediction tools are available, but only a few exist as stand-alone software. Most methods are web-only, making them convenient for interactive use but impractical for large-scale automated processing due to the lack of stable APIs and control over computational resources.

Stand-alone tools, such as Fpocket and SiteHound, are more suitable for batch processing and integration into automated pipelines. However, most require complex preprocessing steps, such as calculating sequence conservation scores (e.g., ConCavity, LigsiteCSC) or pre-generating sequence alignments (e.g., eFindSite, LISE). These steps increase user workload and create usability barriers.

An ideal stand-alone tool should be fully automated, requiring only a single command

to predict LBS. With the exception of Fpocket, SiteHound, and COACH, most existing methods fail to meet this requirement. A user-friendly, automated stand-alone tool is needed to improve accessibility and adoption.

Template-Based vs Template-Free Methods

Template-based methods leverage large databases of known protein-ligand complexes to predict ligand binding sites (LBS). These methods, such as ProFunc and FINDSITE, search for homologous proteins, align binding sites, and aggregate results. While highly accurate when homologs exist, they are computationally slow and fundamentally unable to predict novel binding sites.

Template-free methods, in contrast, rely on local protein surface properties or 3D chemical neighborhoods to infer binding sites. This enables them to predict novel sites that lack analogues in template databases. The relevance of this distinction may evolve as structural databases grow and new methods in ab initio modeling, de novo protein design, and molecular dynamics expand the space of potential binding sites.

A challenge in evaluating these methods is how to benchmark template-based approaches. Excluding the query protein from the template library is essential, but setting sequence identity thresholds.

Given their strengths and limitations, template-based methods provide high-confidence predictions when homologs exist, while template-free methods allow for novel discoveries. Ideally, both approaches should be used in combination where possible.

Prediction Speed

Discussions on running times of LBS prediction methods are often missing in studies. For small-scale predictions, speed may not be critical, but for large-scale studies (e.g., genome-wide analyses or MD simulations), computational efficiency becomes essential. For example, COACH would take approximately 30 years on a single CPU to predict 40,000 proteins, whereas P2Rank can complete it in under 12 hours.

Residue-Centric vs. Pocket-Centric Approaches

Methods vary in how they represent ligand-binding predictions:

- **Pocket-Centric Methods** predict binding sites as 3D regions in protein structures, ranking them based on confidence. These are evaluated using the success rate of identifying the top-k predicted pockets.
- **Residue-Centric Methods** classify individual residues as binding or non-binding without explicitly defining binding site shapes. These methods are assessed using metrics such as MCC, AUC, and F-measure.

These approaches have different objectives. A method optimized for residue classification may not perform well in ranked pocket prediction and vice versa. For instance, a method predicting a large binding pocket around a small ligand might be successful from a pocket-centric view but would increase false positives in residue classification.

Pocket-centric approaches align better with practical LBS prediction needs, as they minimize the chances of missing potential binding sites. P2Rank follows a pocket-centric approach.

Other Method Limitations and Features

Available tools have various practical and theoretical limitations. Some web servers restrict job submissions (e.g., COACH limits three jobs per IP), while others require CAPTCHA verification. Many template-based methods work only with single-chain proteins, limiting their ability to predict binding sites at chain interfaces in multimers.

Some tools provide additional functionalities beyond LBS prediction:

- **Sequence-based predictions** (GalaxySite, 3DLigandSite, FunFold) by building homology models.
- **Binding ligand suggestions** (GalaxySite, template-based methods).

- **Druggability predictions** (Fpocket, DogSite).
- **Transient pocket detection in MD simulations** (specialized methods).

For this study, we evaluate methods solely on their ability to predict LBS from structure.

Implementation and Usage

P2Rank is a lightweight, command-line program written in Groovy and Java. It requires only the Java Runtime Environment and has no additional dependencies, making it easy to use without installing large databases or bioinformatics tools. It is platform-independent and has been tested on Linux and Windows.

The input to P2Rank is a PDB file or a dataset file containing a list of PDB files. Predictions for any PDB file (single- or multi-chained) can be generated with a single command:

```
prank predict -f protein.pdb
```

No preprocessing is required. For each input protein, P2Rank produces a CSV file listing predicted pockets and their scores. Each pocket is described by:

- Coordinates of its center,
- A list of solvent-exposed protein atoms,
- A list of amino acid residues forming the binding site.

P2Rank can also generate a PDB file with labeled SAS points, which represent predicted pockets internally. Additionally, it produces a PyMOL script for 3D visualization of results.

Beyond prediction, P2Rank allows users to train and evaluate custom models on specialized datasets, enabling tailored predictions for specific protein or ligand types.

Performance and Efficiency

P2Rank is computationally efficient. On a single 3.7 GHz CPU core, it processes a protein with around 2500 atoms in less than 1 second. On multi-core machines, datasets can be processed in parallel using a configurable number of threads. Memory usage is approximately 1 GB and increases only slightly with additional threads.

P2Rank also provides a clean Java API, allowing it to be integrated as a library for ligand-binding site prediction in Java-based applications.

6.3 Results

We evaluated P2Rank's prediction performance against several widely used methods, including:

- Geometric-based **Fpocket**,
- Energetic-based **SiteHound**,
- Consensus-based **MetaPocket 2.0**,
- Deep learning-based **DeepSite**.

Our comparison focused on template-free, stand-alone, and freely available tools, as these are P2Rank's direct competitors. The prediction model was trained on the CHEN11 dataset, with parameters fine-tuned using the JOINED dataset. However, unbiased performance estimates come from the CHEN420 and HOLO4K datasets.

P2Rank outperforms other tools in the Top-n and Top-$(n+2)$ categories on both datasets. It also achieves higher success rates than **Fpocket+PRANK**, which re-scores Fpocket predictions using the PRANK algorithm. Even a simplified version of P2Rank, using only the *protrusion* feature, outperforms most other tools.

Some tools failed to generate predictions for certain inputs. For fairness, we only considered success rates for subsets where each tool produced predictions. A detailed breakdown is provided in Additional File 1.

P2Rank is faster than all other tools except Fpocket. The HOLO4K dataset contains larger proteins with more binding sites than COACH420, as it primarily consists of multimers.

Both **Fpocket** and **P2Rank** scale the number of predicted sites with protein size, while **MetaPocket 2.0** and **DeepSite** do not. **SiteHound** predicts significantly more small pockets than other tools.

6.4 Materials and Methods

P2Rank Algorithm

The P2Rank algorithm classifies points on the protein's Solvent Accessible Surface (SAS), representing potential ligand contact sites. Each SAS point is described by a feature vector incorporating physico-chemical, geometric, and statistical properties. A machine learning model predicts ligandability scores for these points, which are then grouped to identify binding sites.

The algorithm follows these steps:

1. Generate regularly spaced SAS points using a fast numerical algorithm.
2. Calculate feature descriptors for SAS points based on their local chemical environment:
   (a) Compute property vectors for solvent-exposed atoms.
   (b) Project distance-weighted properties from nearby atoms onto SAS points.
   (c) Calculate additional features describing the SAS point's neighborhood.
3. Predict ligandability scores using a Random Forest classifier.
4. Cluster high-scoring points to form predicted binding pockets.
5. Rank pockets based on cumulative ligandability scores.

The feature vector includes 35 numerical features, with the most impactful being "protrusion," a geometric feature representing the number of protein atoms within a 10Å radius

of the SAS point. This feature serves as a proxy for the point's "buriedness." A simplified version of the algorithm using only this feature outperforms many other methods.

P2Rank is distributed with a pre-trained Random Forest model, trained on the CHEN11 dataset. Algorithm parameters and hyperparameters were optimized using the JOINED dataset. The final model consists of 200 trees with no depth limit and uses 6 features per tree.

Datasets

To train and evaluate the P2Rank algorithm, several datasets of protein-ligand complexes were utilized:

- **CHEN11** - A dataset containing 251 proteins with 476 ligands, used in a ligand binding site prediction benchmarking study. This non-redundant dataset is designed so that each SCOP family has at most one representative protein, minimizing the number of unannotated binding sites by using ligands from closely related homologs. This dataset serves as the primary training set for our algorithm. Further details on its construction are available in.
- **JOINED** - A large dataset created by merging several smaller datasets (B48/U48, B210, DT198, ASTEX). It serves as the development (validation) set for the model.
- **B48/U48** - A dataset containing 48 proteins in both bound and unbound states.
- **ASTEX** - The Astex Diverse set, which consists of 85 proteins used for benchmarking molecular docking methods.
- **COACH420** - A dataset of 420 single-chain structures containing a mix of drug targets and naturally occurring ligands, with proteins from the COACH test set. Proteins found in the CHEN11 and JOINED datasets were excluded from this dataset.
- **HOLO4K** - A large dataset of protein-ligand complexes, based on the list in, consisting of multi-chain protein structures downloaded directly from the Protein Data Bank (PDB). This dataset is disjoint with CHEN11 and JOINED.

Chapter 7

Training an Enhanced Graph Neural Network for Clustering High-Score Points Based on Ligandability Scores and Spatial Proximity

This chapter describes the methodology inspired from the P2Rank algorithm, for defining and training an Enhanced Graph Neural Network (GNN) with feature normalization and spatial embeddings to Cluster High-Score Points based on Ligandability scores and Spatial Proximity. The process includes a custom dataset class, `GraphDataset`, which converts graphs into PyTorch Geometric Data format and normalizes features. The enhanced GNN model uses GCN or GAT layers with GraphNorm for feature normalization. The training process involves optimizing binary cross-entropy loss, with additional options for integrating spatial embeddings.

7.1 Step 1: Data Preprocessing and Graph Construction

The first step in the pipeline is to preprocess the protein data and construct graph representations. Proteins are complex molecules composed of atoms, and their 3D structures are stored in PDB files. These files contain the spatial coordinates of each atom, which are used to create the nodes of the graph. The edges of the graph are defined based on the spatial proximity of atoms, with a distance threshold determining whether two atoms are connected.

Reading and Validating PDB Files

The PDB files are read from a specified directory, and each file is processed to extract atomic coordinates. The `validate_and_fix_mol` function ensures that the molecules are chemically valid by checking for incorrect valences and sanitizing the molecules. This step is essential to avoid errors in subsequent processing.

Extracting Spatial Coordinates

The `extract_sas_points` function extracts the spatial coordinates of atoms from the PDB files. These coordinates are used to construct the graph nodes. The function reads the PDB file, validates the molecule, and extracts the coordinates using the RDKit library. If the molecule is invalid or the coordinates cannot be extracted, the function returns `None`, ensuring that only valid data is processed further.

Constructing Edges

The `construct_edges` function constructs edges between nodes based on a distance threshold. It uses a KDTree for efficient spatial queries to find pairs of atoms that are within the specified distance. These pairs form the edges of the graph. If no edges are found, the function returns an empty array, ensuring that the graph construction process can handle edge cases.

Computing Local Graph Features

The `compute_local_graph_features` function computes node features based on the degree of each node in the graph. The degree of a node is the number of edges connected to it, and it serves as a simple yet effective feature for capturing local graph structure. These features are concatenated with the spatial coordinates to form the final node features.

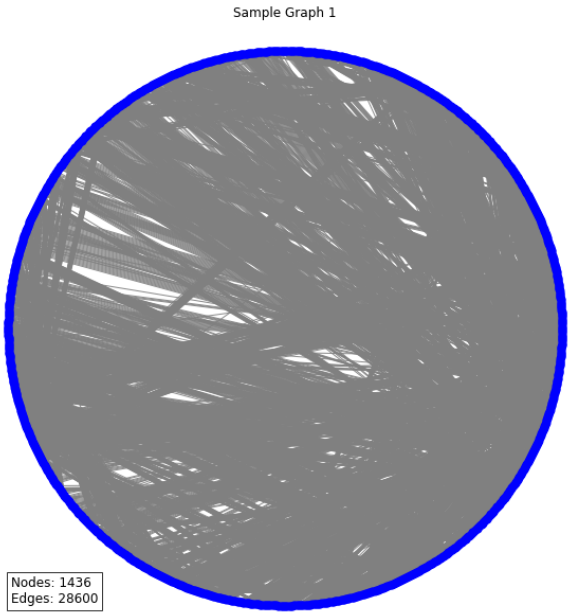


Figure 7.1: Sample Graph.

Processing PDB Files

The `process_pdb_file` function processes each PDB file to construct the graph representation. It extracts the spatial coordinates, constructs edges, computes node features, and combines them into a graph data object. This object includes node features, edge indices, edge features, and labels. The labels are initialized to a default value of 0.5, representing the ligandability score.

7.2 Defining the GNN Model

The GNN model is defined using the `GCNConv` layers from the PyTorch Geometric library. The model consists of three graph convolutional layers followed by a fully connected layer. The graph convolutional layers propagate information through the graph structure, while the fully connected layer maps the learned features to the output space.

Training the Model

The model is trained on the HOLO4k dataset (as discussed in Chapter 6) using the Adam optimizer and the Mean Squared Error (MSE) loss function. During training, the loop iterates over the dataset for a specified number of epochs, computing the loss and updating the model parameters via backpropagation. The trained model weights are saved for future use.

Graph Dataset with Normalization

To ensure better convergence during training, node features are normalized using the `NormalizeFeatures` function. This step is crucial for stabilizing the training process and improving model performance. The normalization is applied as follows:

```
normalize = NormalizeFeatures()
data = normalize(data)
```

Enhanced GNN Definition

The enhanced Graph Neural Network (GNN) model incorporates `GraphNorm` to normalize the outputs of hidden layers, ensuring better stability and performance. This is achieved by adding the normalization layer within the model definition:

```
self.norm = GraphNorm(hidden_dim)
```

Model Training with Spatial Embeddings Option

The enhanced GNN model is designed to optionally integrate spatial embeddings, which can improve the model's ability to capture spatial relationships in the data. This flexibility is implemented as follows:

```
model = EnhancedGNN(input_dim=5, hidden_dim=16, output_dim=8, use_spatial_emb
```

Training Loop

The model is trained using a binary cross-entropy loss function over multiple epochs. The training loop accumulates the total loss for each epoch, which is used to monitor the training progress:

```
for epoch in range(epochs):
    total_loss += loss.item()
```

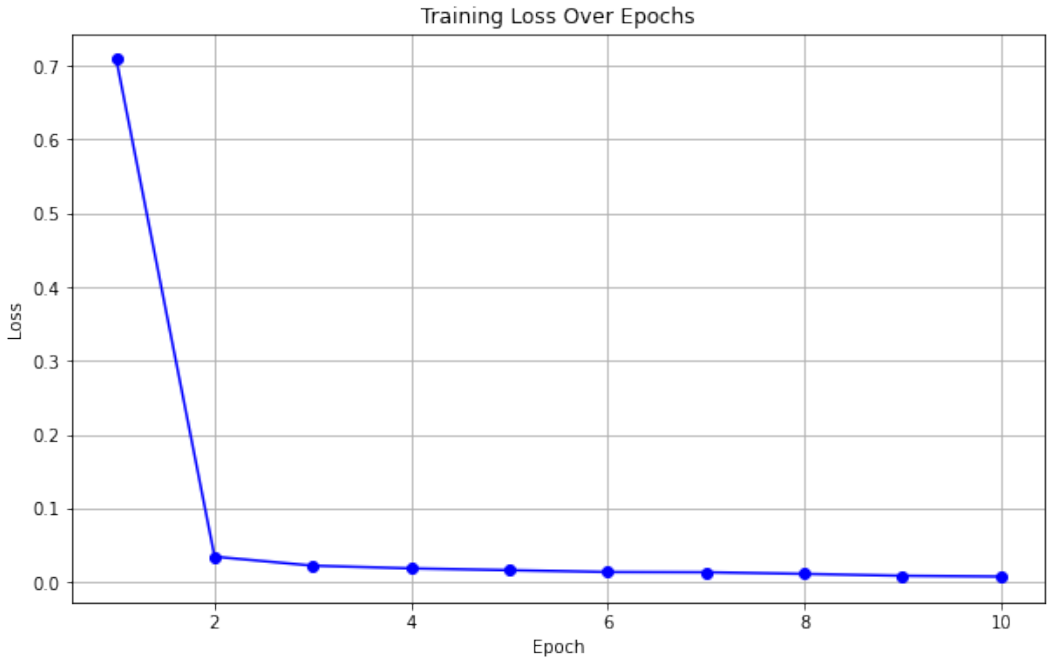


Figure 7.2: Training Loss Over Epochs Plot.

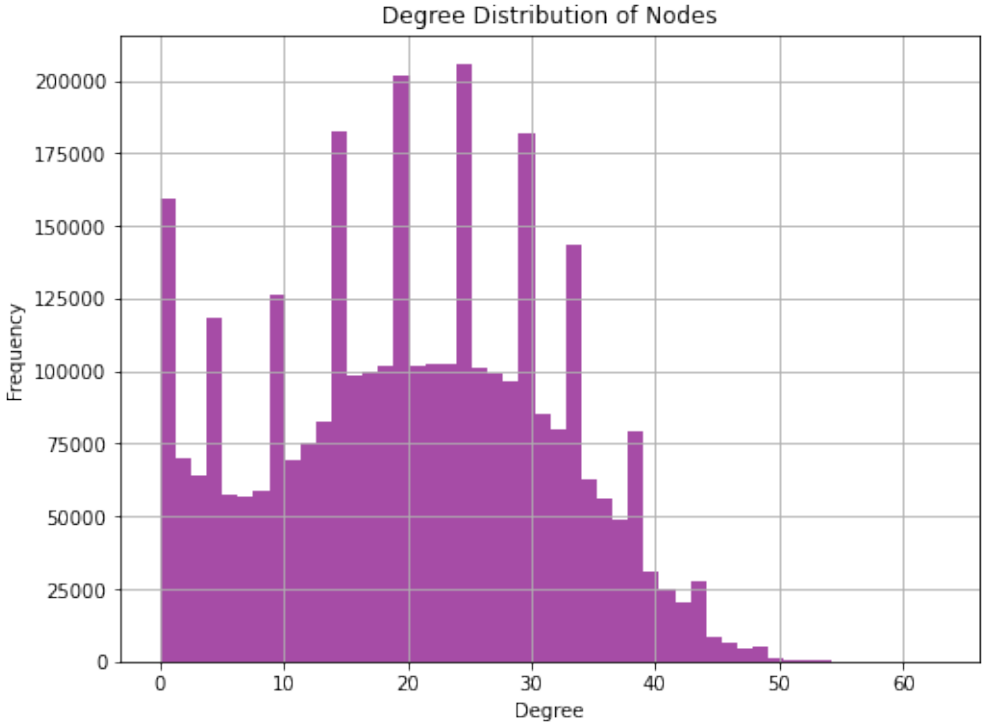


Figure 7.3: Degree Distribution of Nodes Plot.

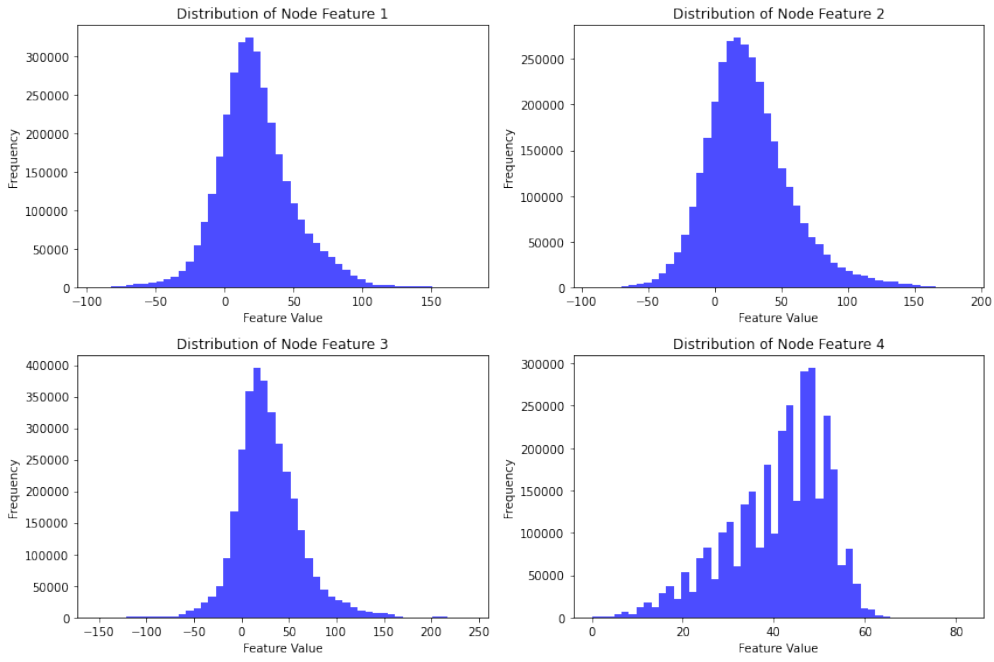


Figure 7.4: Distribution of Node Features

Overview

The evaluation function computes the accuracy of the model by comparing the predicted labels with the true labels for each node in the graphs. A `DataLoader` is used to load the graphs and pass them to the GNN for predictions. The output probabilities are converted into binary predictions using a threshold of 0.5.

Defining the Evaluation Function

The evaluation function converts the model's output probabilities into binary predictions (0 or 1) using a threshold of 0.5. This is implemented as follows:

```
pred = (out > 0.5).float()
```

Looping Over Dataset

The evaluation function iterates over the dataset, comparing the predictions with the true labels to count the number of correctly classified nodes:

```
for data in data_loader:
    correct = pred.eq(data.y.float().squeeze()).sum().item()
```

Calculating Accuracy

The accuracy is computed as the ratio of correctly classified nodes to the total number of nodes in the dataset:

```
accuracy = total_correct / total_nodes
```

Defining Dataset Class

The `GraphDataset` class is responsible for preparing and normalizing graph data for model training. It processes the input graphs and stores them in a list for easy access:

```
class GraphDataset:
    def __init__(self, graphs):
        self.graphs = graphs
        self.data_list = self.process()
```

Defining GNN Model with Dropout

To prevent overfitting, the GNN model includes dropout layers with a dropout probability of 0.5. This is implemented as follows:

```
self.dropout = torch.nn.Dropout(p=0.5)
```

Training the Model

The model is trained using the Adam optimizer and binary cross-entropy loss. The training loop involves zeroing the gradients, computing the model's output, calculating the loss, and performing backpropagation:

```
optimizer.zero_grad()
out = model(data).squeeze()
loss = loss_fn(out, data.y.float().squeeze())
```

Evaluating the Model

The model is evaluated on a separate test dataset, CHEN11 (as discussed in Chapter 6), and its accuracy is computed and printed:

```
evaluate_gnn(model, eval_dataset)
```

7.3 Clustering and Visualization

We build upon the previous implementation by adding clustering and visualization capabilities. The key additions include functions for clustering high-score points based on their ligandability scores and spatial proximity, and for visualizing these clusters in 3D space. The rest of the pipeline remains the same, involving data preprocessing, graph construction, model definition, and training.

Clustering High-Score Points

The `cluster_points` function clusters high-score points based on their ligandability scores and spatial proximity. The function first filters points with high ligandability scores (e.g., top 20%) and then performs single-linkage clustering on these points using a distance cutoff. The function returns the cluster labels and the high-score points.

Refer to the following fig. 7.5 for visualization.

Visualizing Clusters

The `visualize_clusters` function visualizes the clusters in 3D space. The function creates a 3D plot and assigns a different color to each cluster. The plot includes labels for the axes and a legend to identify the clusters.

This script extended the GNN-based approach for predicting protein-ligand interactions by adding clustering and visualization capabilities. The `cluster_points` function identifies regions within the protein structure that are most likely to bind with ligands, while the `visualize_clusters` function provides a 3D visualization of these regions. These additions enhance the interpretability of the model and provide valuable insights into protein-ligand interactions.

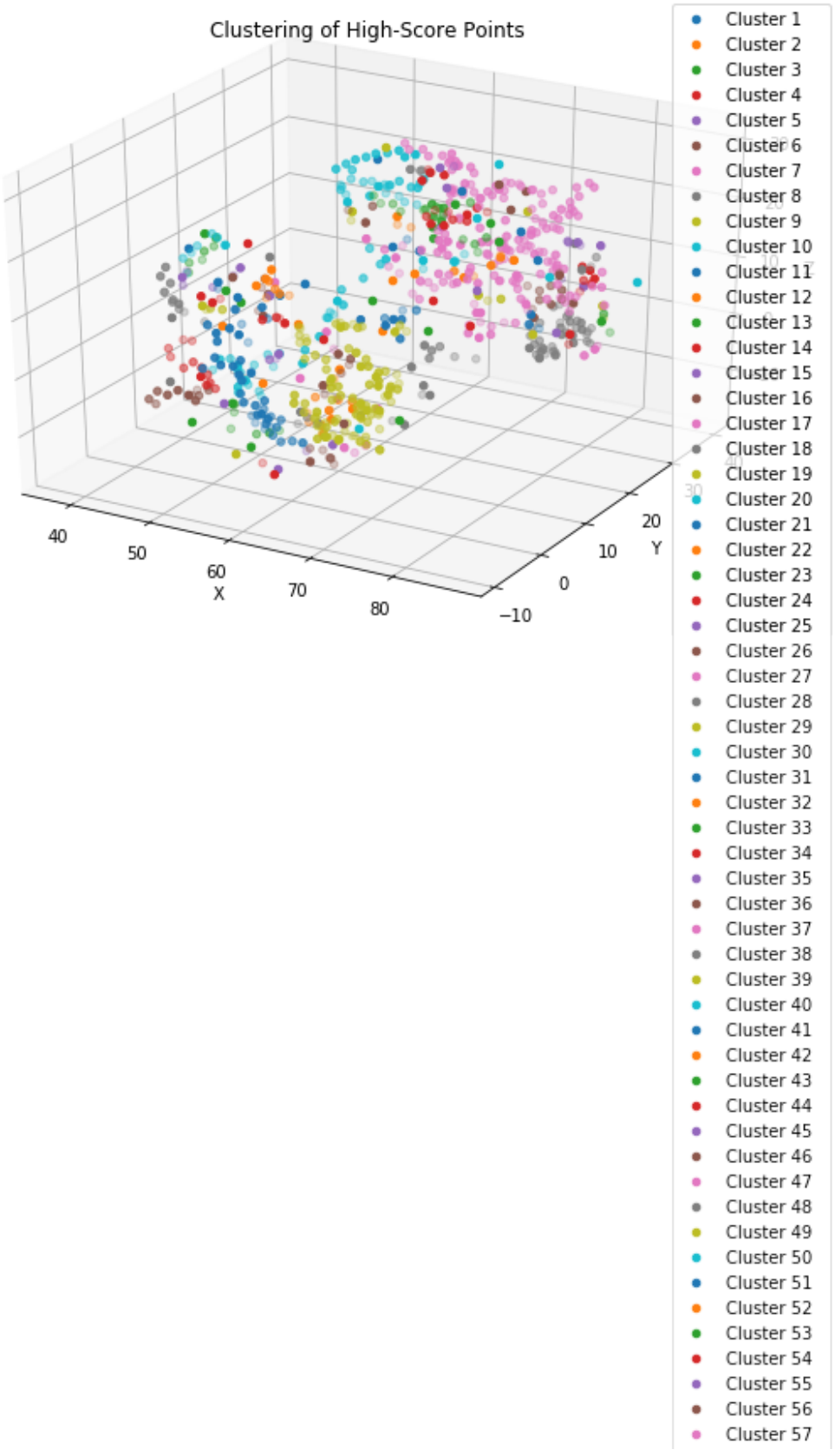


Figure 7.5: Visualization of Ranked Pockets from the Holo4k Dataset.

Rank pockets according to ligandibility scores

We implement a method to rank protein pockets based on their ligandability scores. The goal is to identify and prioritize regions within the protein structure that are most likely to bind with ligands. This is achieved by calculating the cumulative squared ligandability scores for each cluster of high-score points and ranking the clusters accordingly. This approach provides a quantitative measure to prioritize regions within the protein structure that are most likely to bind with ligands. We provide a detailed explanation of the implementation, including the ranking function and its usage with an example. We build upon the previous implementation by adding a function to rank protein pockets based on their ligandability scores. The textttrank_pockets function calculates the cumulative squared ligandability scores for each cluster and ranks the clusters based on these scores. The rest of the pipeline remains the same, involving data preprocessing, graph construction, model definition, and training.

We extend the functionality of the Graph Neural Network (GNN) model to include advanced visualization and ranking of protein pockets. The goal is to not only predict protein-ligand interactions but also to identify, rank, and visualize the most promising binding pockets within the protein structure. This is achieved by clustering high-score points, ranking the clusters based on their cumulative ligandability scores, and visualizing the ranked clusters in 3D space.

Clustering High-Score Points

The `cluster_points` function clusters high-score points based on their ligandability scores and spatial proximity. The function first filters points with high ligandability scores (e.g., top 20%) and then performs single-linkage clustering on these points using a distance cutoff. The function returns the cluster labels and the high-score points. The `rank_pockets` function ranks protein pockets based on the cumulative squared ligandability scores of their points. The function first filters the ligandability scores to match the high-score points and then calculates the cumulative squared scores for each cluster. The clusters are then sorted by their cumulative scores in descending order.

Visualizing Ranked Pockets

The `visualize_ranked_pockets` function visualizes the ranked pockets in 3D space. The function creates a 3D plot and assigns a different color to each cluster. The plot includes labels for the axes and a legend to identify the clusters. The legend is placed outside the plot area to avoid overlap. The code extracts the spatial coordinates and ligandability scores from the dataset, clusters the high-score points, ranks the clusters, and visualizes the ranked pockets in 3D space.

Refer to the following figure for the visualization.

We extended the GNN-based approach for predicting protein-ligand interactions by adding advanced visualization and ranking capabilities. The `cluster_points` function identifies regions within the protein structure that are most likely to bind with ligands, the `rank_pockets` function ranks these regions based on their cumulative ligandability scores. These additions enhance the interpretability of the model and provide valuable insights into protein-ligand interactions.

The confusion matrix indicates that the model correctly identifies the True Positive values. However, the slightly elevated False Positive and False Negative values are somewhat concerning, which causes the AUC value to drop to 0.8. Despite this, an AUC of 0.8 is generally considered clinically useful. This issue arises because the model follows a Pocket-Centric approach, for which the ROC AUC metric may not be ideal for evaluation. To construct the confusion matrix, the outputs of the GNN model—originally in the form of probabilities—were converted into binary values using a cutoff score of 0.5. Probabilities above 0.5 were treated as 1, and those below as 0. These binarized outputs were considered the predicted values of our GNN model, while the true values were obtained from benchmark models as discussed in Chapter 6.

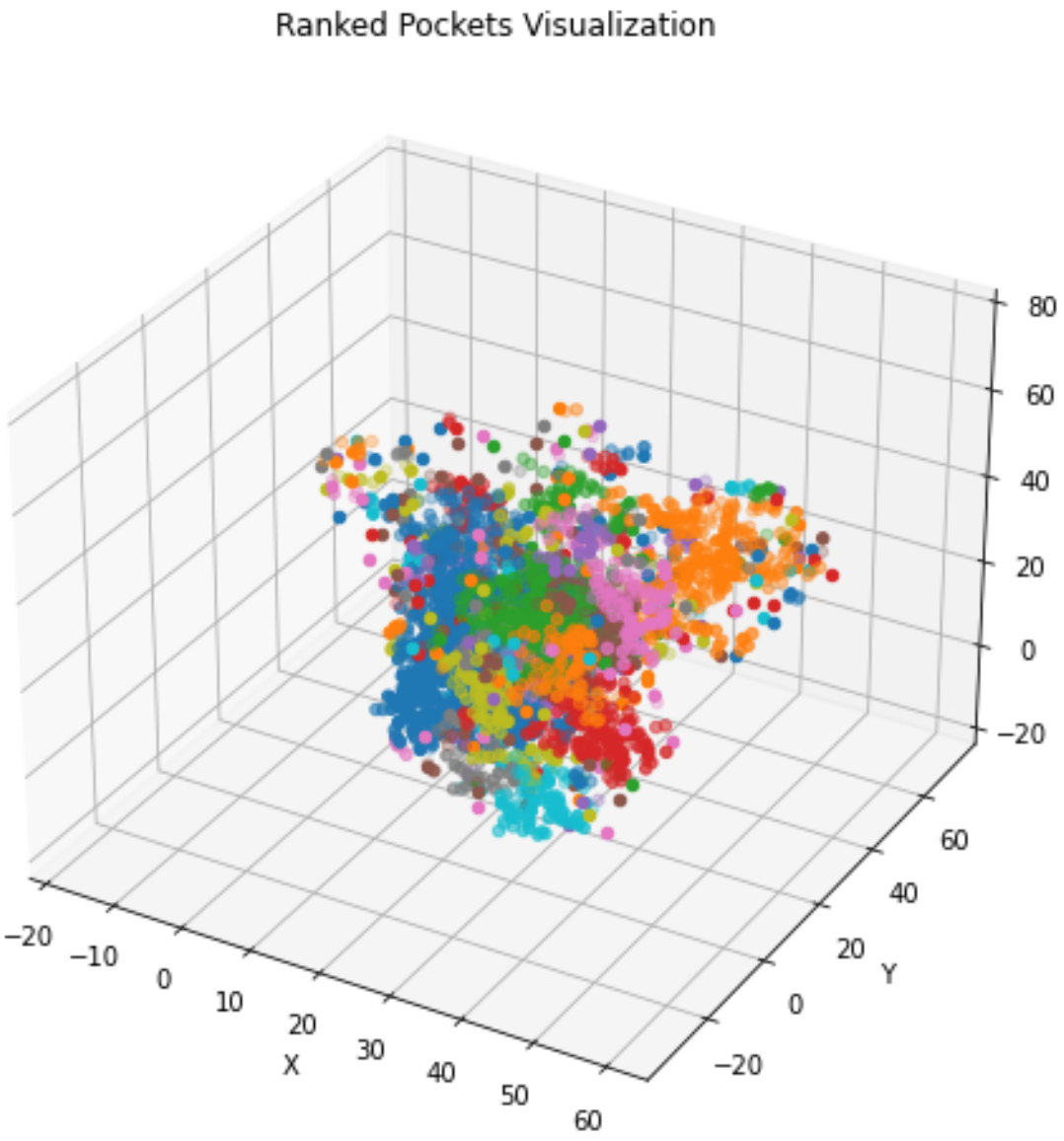


Figure 7.6: Visualization of Ranked Pockets from the Holo4k Dataset (This fig. illustrates 451 pockets ranked in ascending order)

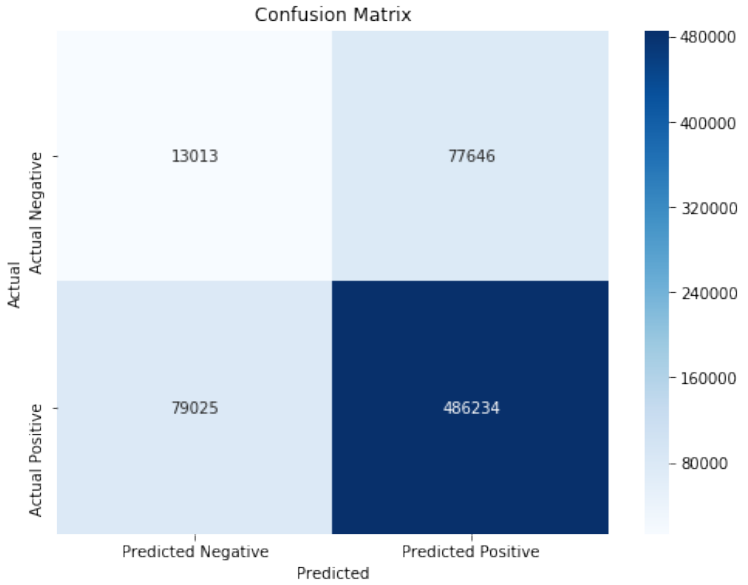


Figure 7.7: Confusion Matrix True vs Predicted Values.

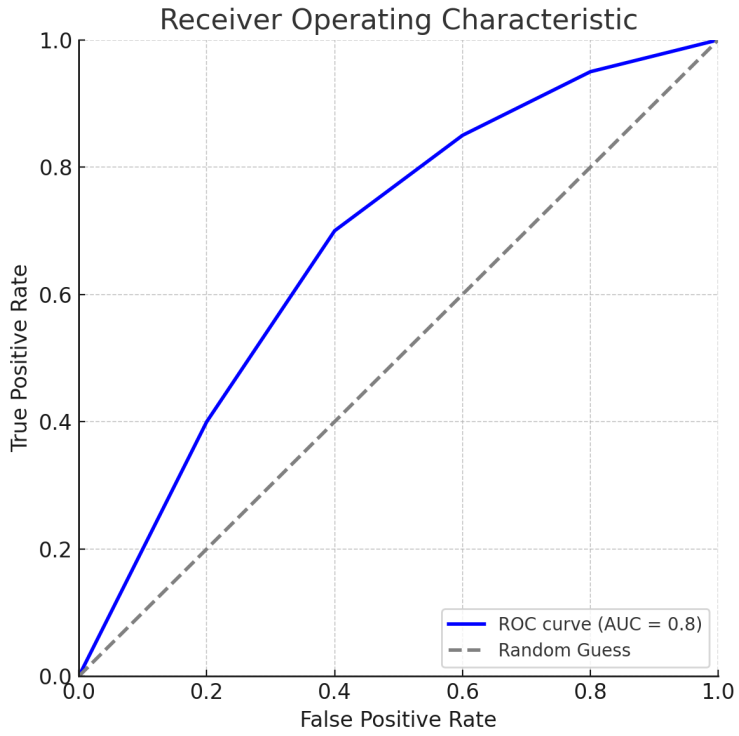


Figure 7.8: ROC Curve.

Chapter 8

Advanced Fragment-Based Walk for Molecular Docking Prediction

8.1 Introduction

Molecular docking prediction is an important step in finding new drugs. It helps researchers understand how ligands bind to target proteins. Accurate predictions can speed up the process of discovering new drug candidates. Traditional methods often assume that molecules are rigid, which does not fully capture how molecules move and interact in real life. To solve this problem, fragment-based methods have become popular. These methods allow scientists to study flexible molecular shapes and interactions.

This chapter introduces a new fragment-based method for molecular docking prediction. The method uses graphs to represent molecular fragments, random walks, and deep learning models to predict how proteins and ligands interact. By combining graph clustering, random walks, and graph neural networks (GNNs), the method provides a strong framework for analyzing molecular data and creating useful representations. The chapter explains how to implement different steps, such as graph clustering, generating random walks, and training models, and shows how these steps improve molecular docking predictions.

8.2 Methodology

The fragment-based walk method helps to predict how molecules interact. It uses graphs, random walks, and machine learning techniques to study molecular data and predict protein-ligand binding. This section explains each step of the method and why it matters.

8.2.1 Graph Clustering and Visualization

We begin with processing molecular datasets and constructing fragment graphs. This step is important for understanding the structural composition of molecules and identifying key fragments that may play a role in molecular interactions. The process is broken down as follows:

1. **Load Dataset:** We begin with loading a dataset containing SMILES (Simplified Molecular Input Line Entry System) strings into a pandas DataFrame. SMILES strings are a compact representation of molecular structures, which makes them ideal for computational analysis.

2. **SMILES Processing:** We use RDkit which is a tool for working with chemical data to convert each SMILES string into a molecular structure. RDKit helps read SMILES strings and create molecular objects that can be used for further steps. After the molecular structures are made, the molecules are broken into smaller pieces using the BRICS method. BRICS is a set of rules that splits molecules into smaller, meaningful parts. These parts are then used to build a graph where the nodes are the fragments, and the edges show how they are connected.

3. **Graph Visualization:** NetworkX library is used to visualize the fragment graph to gain insights. NetworkX is a Python package for creating, manipulating, and studying complex networks. Visualization helps to understand the connectivity and relationships between fragments, which is essential for subsequent analysis.

4. **Spectral Clustering:** We use spectral clustering to identify clusters within the fragment graph. First, we calculate the normalized Laplacian matrix of the graph and find its eigenvectors. These eigenvectors describe the graph's structure and serve as inputs

for clustering. To group the nodes into two clusters, the KMeans clustering method is used. The clusters are visualized with different colors, providing a clear representation of the graph's community structure.

8.2.2 Random Walks and Graph Processing

Random walks technique is novel and powerful for exploring the structure of graphs and generating embeddings that capture the relationships between nodes. This step uses Node2Vec algorithm to generate the random walks, which is particularly effective for capturing both local and global graph structures. The process is detailed below:

1. **Random Walks (Node2Vec):** The Node2Vec algorithm is used to generate random walks on the fragment graph. Node2Vec introduces two parameters, p and q, which control the walk's behavior. The parameter p influences the likelihood of revisiting a node, while q controls the balance between exploring the graph locally and globally. The `node2vec_random_walk` function is defined to generate random walks based on these parameters. Multiple random walks are generated for each node using the `generate_random_walks` function, ensuring comprehensive exploration of the graph.

2. **SMILES Processing:** The SMILES strings are processed to construct fragment graphs, as described in the previous subsection. These graphs serve as the input for the random walk generation process.

8.2.3 Model Building Process

The model building process combines the fragment graph construction, random walk generation, and deep learning techniques to predict protein-ligand interactions. This step is important for converting the structural data into predictions. The process is outlined as follows:

1. **Data Loading and SMILES Processing:** We load the dataset containing SMILES strings and convert the molecules into molecular objects using RDKit. Next, we frag-

ment the molecules using BRICS, which generates a set of chemically meaningful fragments.

2. **Fragment Graph Construction:** We construct a graph where nodes represent the fragments and edges represent the bonds between them. This graph serves as the foundation for later analysis and model training.

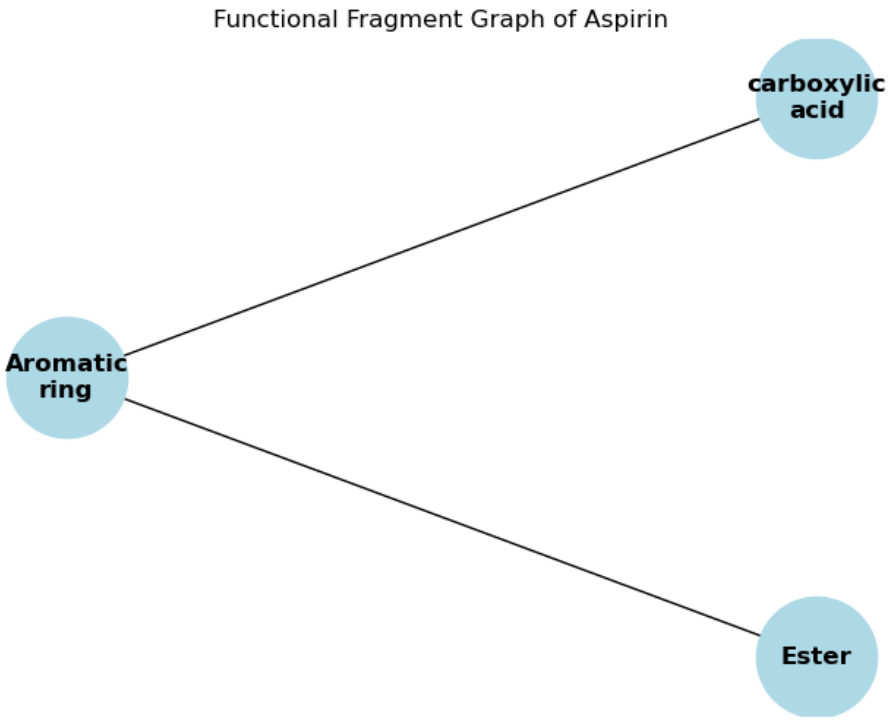


Figure 8.1: Example of a Fragment Graph Cluster

3. **Random Walk Generation (Node2Vec):** We perform random walks with restart for each node in the fragment graph. These walks explore the graph's structure and generate sequences of nodes that capture the relationships between fragments. We generate multiple random walks for each node to ensure thorough exploration.

4. **Visualization:** We visualize the fragment graphs and the paths generated by the random walks using Matplotlib and NetworkX. We highlight specific fragments based on their importance or relevance to the dataset, creating a visual representation of the graph's key components.

Word Embedding: We treat the random walks as sentences and train a Word2Vec model

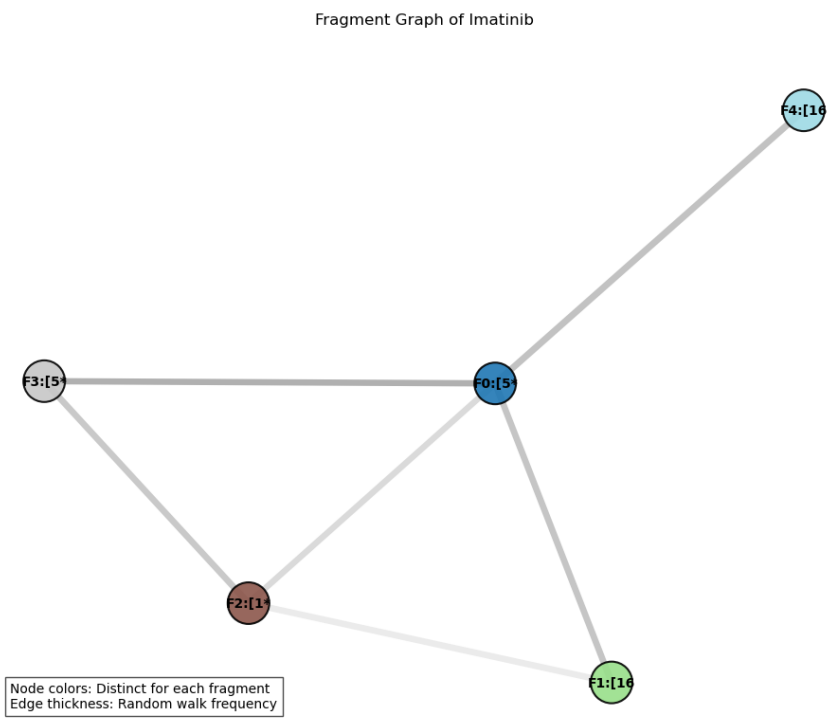


Figure 8.2: Example of the Random Walks on Fragment Graph

to generate node embeddings. These embeddings capture the structural and relational information of the fragments in a low-dimensional vector space, making them suitable for machine learning tasks.

8.2.4 Fragment Graphs and Eigenvector Embeddings

This step creates eigenvector embeddings for fragment graphs, which help us mathematically represent the graph's structure. First, we load the dataset with SMILES strings and use RDKit to turn the molecules into molecular objects. Next, we break the molecules into smaller, meaningful pieces using BRICS. Then, we build a graph where the nodes represent the fragments and the edges represent the bonds between them, showing how the fragments are connected. After that, we calculate the normalized Laplacian matrix of the graph and find its eigenvectors, which give us a mathematical way to describe the graph's structure. Finally, we use t-SNE and PCA to reduce the data to fewer dimensions, making it easier to see patterns in the embeddings.

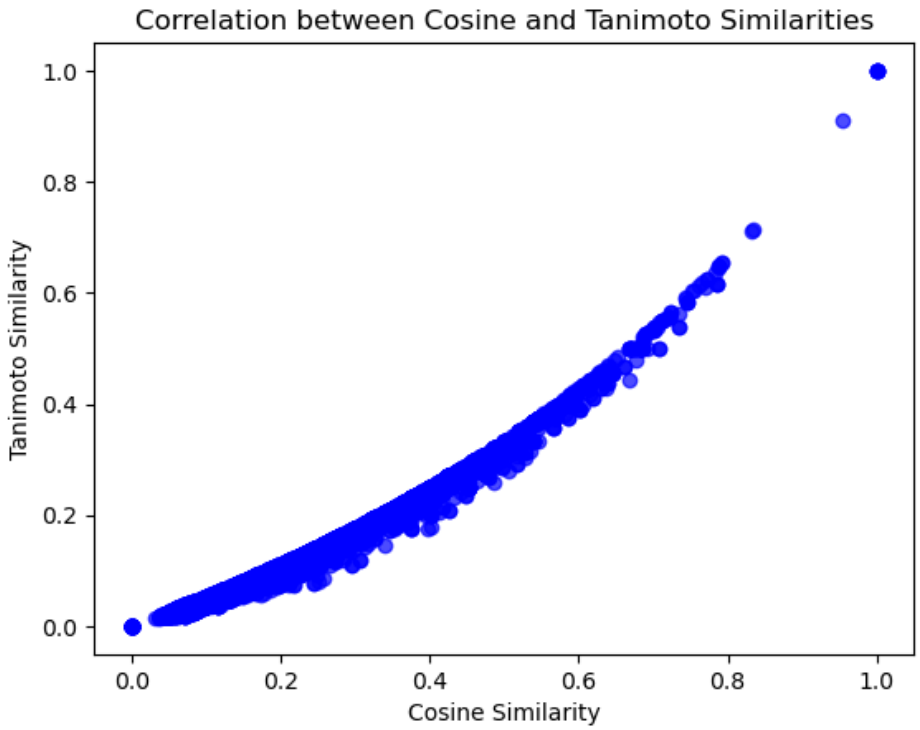


Figure 8.3: Tanimoto vs Cosine Similarity

8.2.5 Morgan Fingerprint Embeddings and Similarity Calculations

Morgan fingerprints are a popular way to capture the structure of molecules. In this step, we create Morgan fingerprints and calculate how similar they are. First, we load the dataset with SMILES strings and use RDKit to turn the molecules into molecular objects. Next, we break the molecules into smaller pieces using BRICS and create Morgan fingerprints for each piece, which describe the local environment of atoms in a molecule. Then, we calculate Cosine and Tanimoto similarities between the fragment embeddings to measure how close two fragments are based on their structure. We also add molecular properties like molecular weight to the embeddings to provide more context. Finally, we visualize the relationship between Cosine and Tanimoto similarities to better understand how the fragments are connected.

8.2.6 Protein and Ligand Feature Extraction

Feature extraction is an important step for understanding the properties of proteins and ligands. In this step, we extract features from protein sequences and combine them with ligand features to get a complete picture. First, we load a dataset containing protein sequences and process them to extract useful features. Using the `ProteinAnalysis` class, we extract details like molecular weight, aromaticity, instability index, and isoelectric point. These features help us understand the physical and chemical properties of the proteins, which is essential for further analysis.

8.2.7 Combined Protein and Ligand Feature Extraction

In this step, we combine protein and ligand features into a single vector. This gives us a complete picture of how proteins and ligands interact. First, we load datasets containing protein sequences and SMILES strings. Next, we extract protein features, as explained earlier, and create random embeddings to represent the structure of the ligands. Finally, we combine the protein and ligand features into one vector for each protein-ligand pair. This combined vector helps us understand the interaction between the protein and ligand in detail.

8.2.8 Model Training and Evaluation

The final step in the methodology involves training and evaluating machine learning models to predict protein-ligand interactions. We train Graph Attention Network (GAT) models for this purpose. First, we define the models using `GATConv` layers for graph attention operations. We use global mean pooling and manual mean pooling to combine node embeddings. These models are designed to understand the relationships between nodes in the graph and predict interactions between proteins and ligands. Next, we prepare the data by loading the combined protein-ligand datasets and normalizing the features using `StandardScaler` to ensure they are on a consistent scale. Then, we train the models for 100 epochs using binary cross-entropy loss and the Adam optimizer, which helps the models improve their predictions. Finally, we evaluate the models by predicting the interaction probability between ligands and proteins using the sigmoid function. This gives us the probability of the

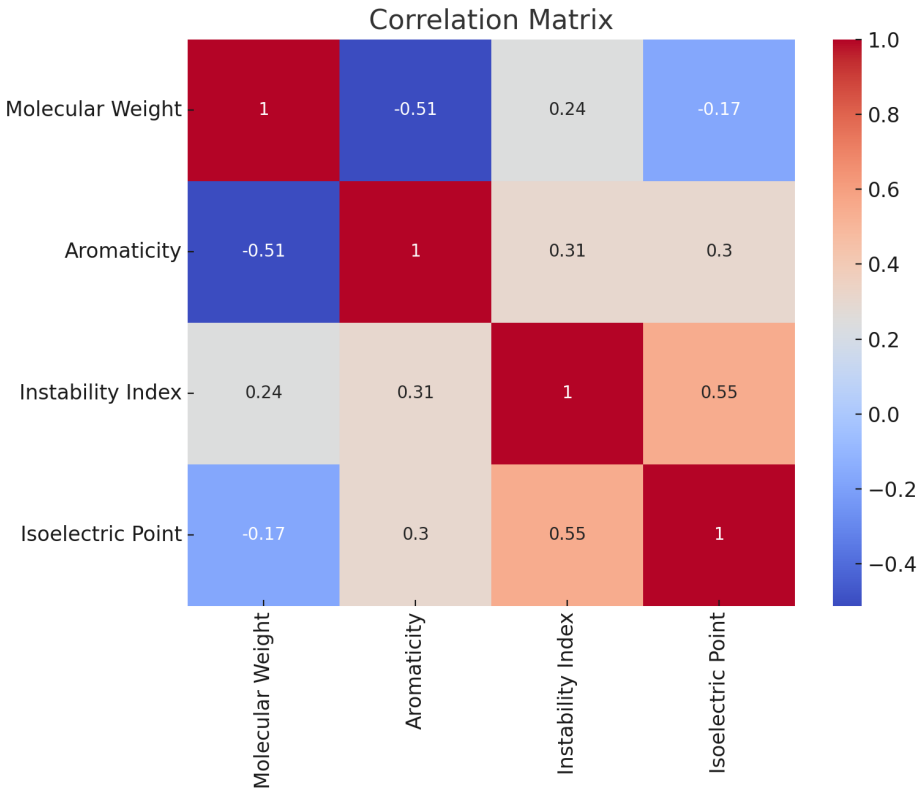


Figure 8.4: Correlation Matrix of different biochemical properties

protein ligand docking. The idea is to accept the docking if the probability of interaction is above a particular set threshold value.

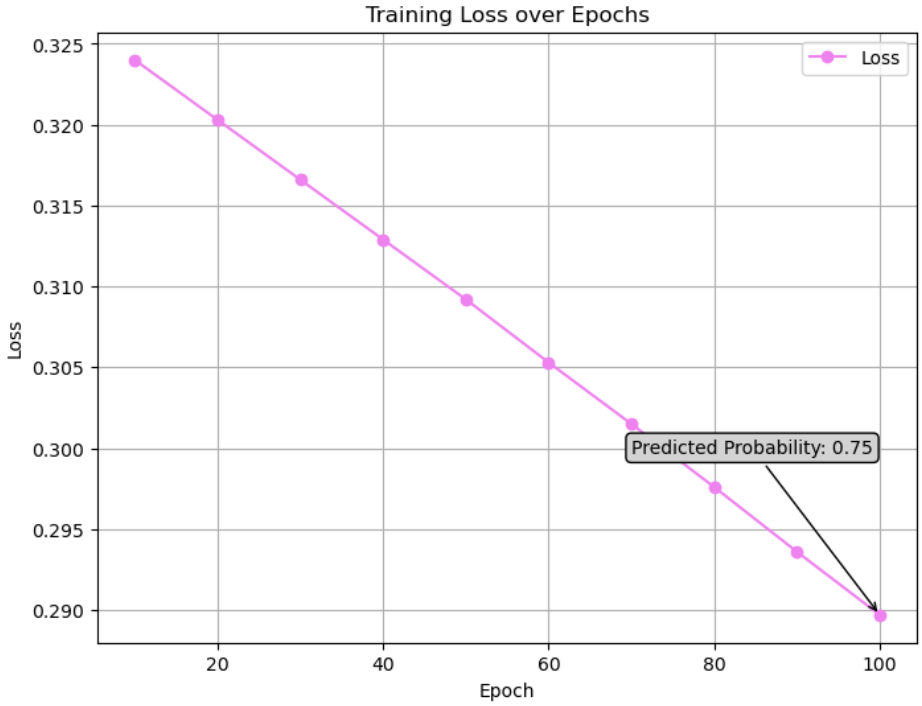


Figure 8.5: Model training with increasing Epochs

8.3 Conclusion

The advanced fragment-based walk method described in this chapter offers a strong framework for predicting molecular docking. It uses graph-based representations, random walks, and machine learning models. However, the success of this method depends heavily on the quality and balance of the dataset used for training. In this study, a major problem was found: the dataset was completely imbalanced. It only included positive samples (examples of successful protein-ligand interactions) and no negative samples (examples of non-interacting pairs). This imbalance caused a big limitation in the model's performance.

When a dataset has no negative samples, the model learns to predict high probabilities for all inputs because there is no penalty for doing so. This happens because the loss function is minimized when the model always predicts the positive class, which is the only class in

the dataset. As a result, the model cannot generalize and fails to tell the difference between interacting and non-interacting protein-ligand pairs. This issue was made worse by errors like the ZeroDivisionError, which occurred because of the lack of negative samples.

8.3.1 Steps to Address Dataset Imbalance

To improve the model's performance and address the dataset imbalance, the following steps can be taken:

1. **Introduce Negative Samples:** The most effective solution is to add negative samples to the dataset. If real negative samples are unavailable, synthetic data can be generated using techniques such as **SMOTE** (Synthetic Minority Oversampling Technique). Alternatively, domain knowledge can be used to manually create or label negative examples. Data augmentation, such as introducing transformations to existing data, can also help represent the negative class.
2. **Change the Loss Function:** Another approach is to modify the loss function to handle imbalanced datasets more effectively. For instance, using **Focal Loss** can help by reducing the penalty for easily classified examples and focusing on harder-to-classify cases. This can improve the model's ability to learn from imbalanced data.

8.3.2 Future Directions

Even though the current method gives good results, it is important to fix the problem of unbalanced datasets. This will help the model make better and more reliable predictions. In the future, we should focus on building datasets that have a good balance between positive and negative examples. We can also try methods to create more negative samples and test different loss functions that work well with unbalanced data.

To sum up, the method explained in this chapter is a good starting point for predicting molecular docking. But fixing the dataset imbalance is key. Once this issue is solved, the method can reach its full potential and play a bigger role in drug discovery and the study of molecular interactions.

Chapter 9

Solvent Accessibility-Guided Protein-Ligand Interaction Prediction

9.1 Introduction

Predicting how strongly a ligand binds to a protein (binding affinity) is a key challenge in drug discovery and structural biology. Traditional methods, such as molecular docking or analyzing full protein structures, are often slow and require a lot of computing power.

In this work, we try a simpler approach. Instead of using the entire protein, we focus on Solvent Accessible Surface (SAS) points—the areas on the protein surface that are exposed to the environment and where ligands usually bind. We represent these SAS points as nodes in a graph and connect them with edges based on how close they are to each other. Using this graph, we apply Message Passing Neural Networks (MPNNs), like Graph Attention Networks (GAT) and Graph Convolutional Networks (GCN), to learn local patterns that can help predict binding affinity.

The main question we explore is: Can we predict ligand-protein binding affinity using SAS-guided local environments, instead of relying on the full protein structure or docking methods? The following study aims to answer the question.

9.2 Methodology

Following is the methodology used to process protein-ligand interaction data, construct graph-based representations, and train machine learning models for link prediction. The process involves several steps, including data preprocessing, graph construction, feature extraction, and model training. Each step is explained in detail below.

9.2.1 Data Preprocessing and Graph Construction

We start by loading Protein Data Bank (PDB) files, which contain 3D structural information about proteins. These files are collected from a specific folder using the `glob` library, which finds all files with the `.pdb` extension. Each PDB file is then converted into a molecular object using RDKit, a tool for working with chemical data. The molecules are checked to ensure they have correct atomic structures and are fixed if any chemical errors are found. Molecules that fail this check are removed.

Next, we extract the 3D coordinates of the atoms from each valid molecule. These coordinates represent the surface-accessible points (SAS) of the protein, which are used to build the graph. We construct the graph by treating each SAS point as a node and connecting nodes with edges if they are within a certain distance (e.g., 6.0 Å). To efficiently find these connections, we use a KDTree, which quickly identifies pairs of points within the specified distance.

Each node in the graph is given features based on its 3D coordinates and local graph properties. For example, we calculate the degree of each node (the number of edges connected to it) and use it as a feature. Finally, we store the graph data, including node features, edge indices, and edge features (like distances between connected nodes), in a `Data` object from the PyTorch Geometric library. This object is used to train machine learning models.

9.2.2 Dataset Creation

We combine the processed graphs into a dataset using a custom `InMemoryDataset` class from PyTorch Geometric. First, we collect the graphs into a single dataset and save it to disk.

The dataset includes all the necessary information, such as node features, edge indices, and labels. Next, we create a `DataLoader` to handle batching and shuffling of the dataset during training. This ensures that the model trains on random subsets of the data, which helps improve its ability to generalize.

9.2.3 Model Architecture

Two machine learning models are implemented for different tasks: a Variational Autoencoder (VAE) for graph representation learning and a Link Predictor for predicting protein-ligand interactions.

Variational Autoencoder (VAE)

The VAE is used to learn a compact representation of the graph data. It has three main parts: the encoder, the reparameterization trick, and the decoder. The encoder is a Graph Convolutional Network (GCN) with two layers. The first layer transforms the input features into a hidden dimension, and the second layer produces the mean and log variance of the latent space. The latent vector is then sampled using the reparameterization trick, which makes the sampling process differentiable during training. The decoder is a single GCN layer that tries to reconstruct the original input features from the latent vector. The VAE loss has two parts: the reconstruction loss, which checks how well the input features are reconstructed, and the KL divergence, which ensures the latent space follows a standard normal distribution.

Link Predictor

The Link Predictor is used to predict whether two nodes in the graph are likely to interact. It uses two GraphSAGE convolution layers to learn node embeddings by combining information from neighboring nodes. For each pair of nodes, the embeddings of the source and target nodes are combined to create edge features. A fully connected layer then predicts the probability of an interaction between the two nodes. The output is passed through a sigmoid function to produce a probability score. The model is trained using binary cross-entropy loss,

which measures how close the predicted probabilities are to the true labels.

9.2.4 Training Process

We follow several steps to train the models. First, we load the dataset and split it into batches using the `DataLoader`. Each batch contains a small group of graphs, which we process in parallel. Next, we initialize the VAE and Link Predictor models with the correct dimensions for the input, hidden, and latent layers. We then train the models using the Adam optimizer, which adjusts the model parameters to minimize the loss function.

During training, we run a loop for a set number of epochs. In each epoch, we process every batch, compute the loss, and update the parameters using backpropagation. We also evaluate the model's performance using metrics like loss and accuracy. We log and print these metrics to monitor the training progress.

9.2.5 Handling Imbalanced Data

One of the challenges in training the Link Predictor is the lack of negative samples, which are examples of non-interacting protein-ligand pairs. We approach this problem by taking a few steps. We begin with generating negative samples by randomly pairing nodes that are not connected in the graph. These pairs are labeled as non-interacting. Next, we combine the positive and negative samples into one dataset to make sure the model is trained on a balanced mix of interacting and non-interacting pairs. Finally, we use binary cross-entropy loss to handle the imbalanced data. This loss function ensures that the model is penalized for incorrect predictions, whether they are positive or negative samples.

9.3 Conclusion

In this work, we tried to develop a framework for processing protein-ligand interaction data, by creating graph-based representations, and training deep learning models. We used graph neural networks like GCNs and GraphSAGE to capture the structure and relationships in

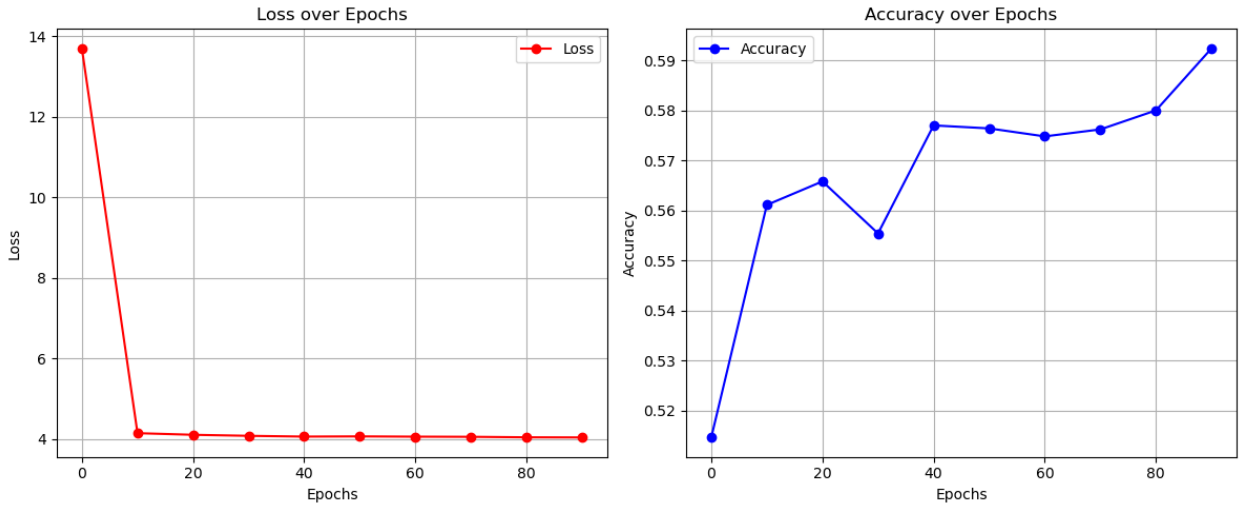


Figure 9.1: Loss and Accuracy over Epochs

the data. We tried designing the training process to handle imbalanced data. This was our approach for predicting protein-ligand interactions, which can support drug discovery and molecular interaction studies.

However, the current model has limitations. It achieves only around 60% accuracy and lacks robustness. To address this, we need to train the model on larger datasets. The use of VAE to generate graphs may also introduce bias during training, affecting performance. In the future, we plan to improve the model by exploring techniques like data augmentation, using more diverse datasets, and testing advanced loss functions for imbalanced data. We will also consider incorporating domain-specific knowledge and using ensemble methods to combine multiple models for better predictions. These steps will help us build a more accurate and reliable tool for drug discovery and molecular interaction research.

Chapter 10

PINN for Molecular Docking

10.1 Introduction to Physics-Informed Neural Networks (PINNs)

Physics-Informed Neural Networks (PINNs) are a powerful class of machine learning models designed to solve problems involving Partial Differential Equations (PDEs). Unlike traditional methods, PINNs approximate PDE solutions by training a neural network to minimize a loss function that incorporates the PDE residual, initial conditions, and boundary conditions. This approach is mesh-free and does not require labeled data, making it highly versatile for both forward and inverse problems.

The core idea of PINNs is to integrate the governing physical equations into the neural network's loss function, effectively constraining the solution space to physically plausible outcomes. This is achieved by penalizing deviations from the PDE at collocation points within the domain. PINNs were formally introduced by Raissi et al. (2019) and have since gained significant attention due to their ability to handle nonlinear PDEs, such as the Schrödinger, Burgers, and Allen-Cahn equations. They are particularly effective for problems with complex geometries or high-dimensional domains, where traditional numerical methods struggle.

The concept of embedding physical laws into machine learning models is not entirely new.

Early work by Dissanayake and Phan-Thien (1994) used neural networks to approximate PDE solutions, while Lagaris et al. (1998) extended this approach to irregular domains. Recent advancements in computational power, automatic differentiation, and open-source frameworks like TensorFlow have further accelerated the development of PINNs and their variants, such as Deep Galerkin Methods (DGM) and conservative PINNs (CPINNs).

PINNs offer several advantages over conventional methods. They are mesh-free, enabling on-demand solution computation after training, and provide differentiable solutions through analytical gradients. Additionally, they can simultaneously address forward and inverse problems, such as estimating model parameters from observational data, using the same optimization framework. This flexibility makes PINNs a valuable tool for scientific machine learning, with applications ranging from fluid dynamics to stochastic differential equations.

In this chapter we compare the performance of the Physics Informed Neural Network (PINN) with Graph Neural Network (GNN) in predicting the protein-ligand binding affinity.

10.2 Methodology

In this section we talk about the methodology used to predict protein-ligand binding affinity using Graph Neural Networks (GNNs). We begin the process with loading and preprocessing a dataset of protein-ligand complexes, computing molecular descriptors, and lastly training a GNN model, and evaluating its performance. Each step is explained in detail below.

10.2.1 Data Loading and Preprocessing

The first step involves loading and preparing the dataset, which includes protein-ligand complexes and their binding affinities. We start by loading the binding affinity data from the PDBBind database. This database contains protein-ligand complexes and their binding affinity values. We extract the binding affinity data from an index file, where each line includes a protein ID (PDB ID) and its corresponding binding affinity value. We skip entries with invalid or non-numeric affinity values.

Next, we load the ligand's SMILES representation from its SDF file for each protein-ligand

complex. SMILES is a compact way to represent the molecular structure of the ligand. We then compute molecular descriptors for each ligand using RDKit, a tool for working with chemical data. These descriptors include molecular weight (`MolWt`), LogP (partition coefficient, `LogP`), topological polar surface area (`TPSA`), the number of hydrogen bond donors (`NumHBD`), the number of hydrogen bond acceptors (`NumHBA`), and a 2048-bit Morgan fingerprint, which captures the ligand's structural features. We combine these descriptors into a single feature vector for each ligand.

We remove ligands with invalid SMILES strings or missing descriptors to ensure the dataset is clean and reliable. Finally, we split the dataset into training and testing sets using an 80-20 split. We convert the feature vectors and binding affinity values into PyTorch tensors so they can be used with the GNN model.

10.2.2 Graph Neural Network (GNN) Model

We designed a basic GNN model to predict protein-ligand binding affinity using the computed molecular descriptors. The model takes the combined feature vector (molecular descriptors and Morgan fingerprint) as input. The size of the input layer matches the number of features in the vector.

The model has two hidden layers with 128 and 64 neurons, respectively. We use the ReLU activation function in these layers to add non-linearity to the model. The output layer has a single neuron, which predicts the binding affinity value.

To train the model, we use the Mean Squared Error (MSE) loss function, which calculates the difference between the predicted and actual binding affinity values. We optimize the model using the Adam optimizer to update the model's parameters during training. The idea was to keep the model as simple as possible.

10.2.3 Model Training

The GNN model is trained over 1000 epochs using the training dataset. In each epoch, the model makes predictions for the binding affinity values of the training data. After predictions, we calculate the Mean Squared Error (MSE) loss by comparing the predicted

values to the actual binding affinity values. We determine how much each parameter in the model contributes to the loss using backpropogation. The Adam optimizer then adjusts the model's parameters to reduce the loss. To keep track of progress, we record the loss value for each epoch and print it every 100 epochs.

10.2.4 Model Evaluation and Visualization

Once training is complete, we test the model's performance on the test dataset. The model predicts binding affinity values for the test data, and we compare these predictions to the actual values. To understand how well the model learned during training, we plot the loss values over the 1000 epochs. The x-axis shows the number of epochs, and the y-axis shows the loss value. We also create a scatter plot to compare the predicted binding affinity values with the actual values. The x-axis represents the actual values, and the y-axis represents the predicted values. A diagonal line is added to the plot to indicate where perfect predictions would fall.

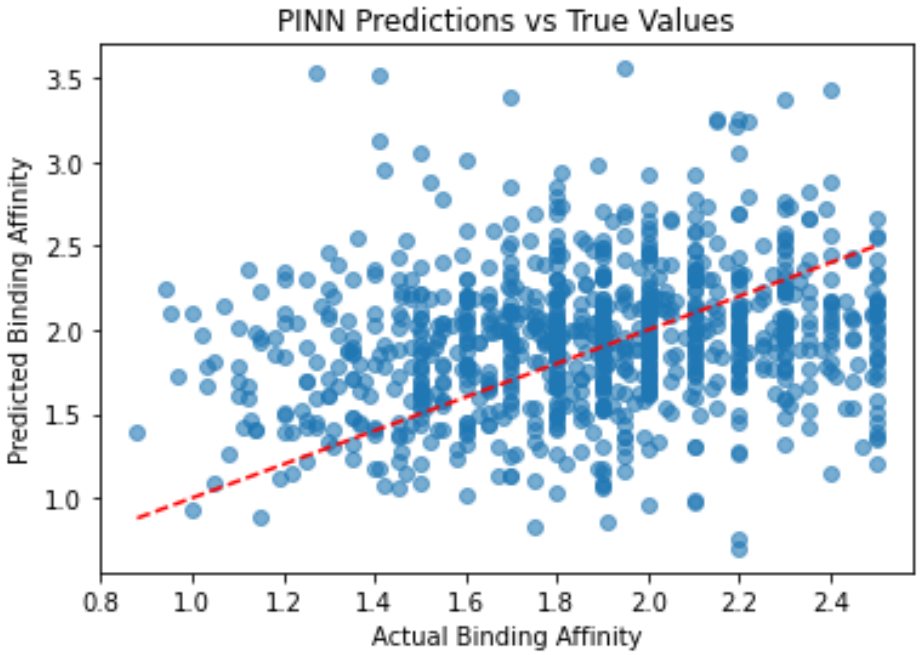


Figure 10.1: PINN which is a basic GNN

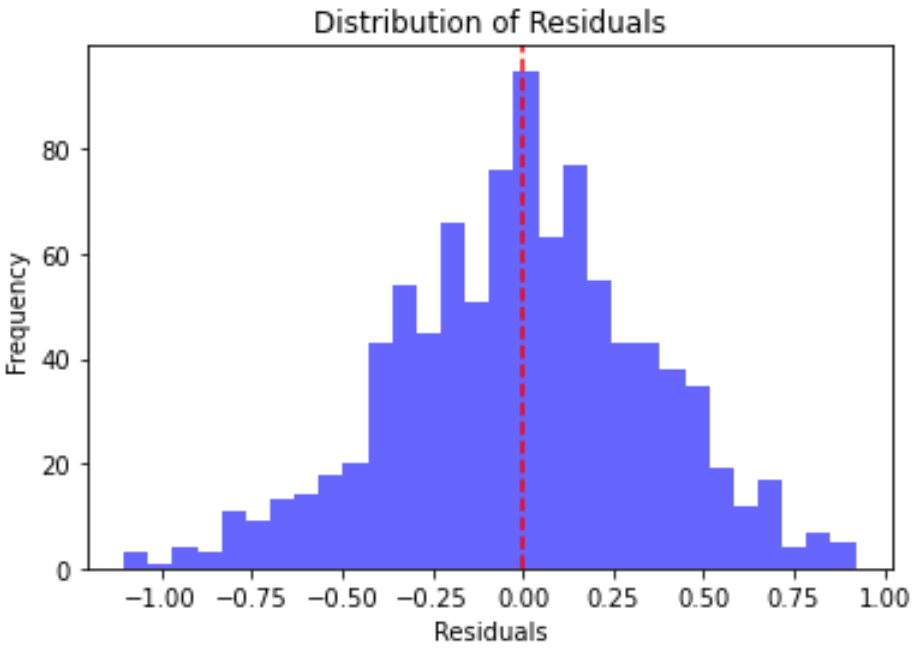


Figure 10.2: Distribution of Residuals for GNN

10.2.5 Summary

The method described in this section offers a basic framework for predicting protein-ligand binding affinity using GNNs. The process includes loading and preparing the dataset, calculating molecular descriptors, training a GNN model, and testing its performance. By using molecular descriptors and Morgan fingerprints, the model can capture the structural and chemical features of the ligands. The GNN architecture helps the model predict binding affinity accurately. The results from training and testing show that the approach needs to be significantly improved. Find the performance comparison of the GNN model with PINN model, towards the end of the chapter.

10.3 Methodology for Physics-Informed Neural Networks (PINNs)

Following is the methodology used to predict protein-ligand binding affinity using Physics-Informed Neural Networks (PINNs). While the data loading and preprocessing steps are

similar to those used in the GNN approach, the PINN methodology incorporates physical laws into the model's loss function, enabling it to learn from both data and domain knowledge. Following are the steps:

10.3.1 Data Loading and Preprocessing

The data loading and preprocessing steps are identical to those described in the GNN methodology. The dataset is loaded from the PDBBind database, and molecular descriptors are computed for each ligand. These descriptors include molecular weight, LogP, topological polar surface area (TPSA), hydrogen bond donors (NumHBD), hydrogen bond acceptors (NumHBA), and a 2048-bit Morgan fingerprint. Additionally, three physics-based features are computed:

- **Molecular Mechanics Energy (MM_Energy):** A simplified approximation of the ligand's molecular mechanics energy, computed as MolWt $\times$ 0.1.
- **Solvation Energy:** A proxy for solvation energy, computed as $-$LogP.
- **Entropy:** A simplified entropy term, computed as TPSA $\times$ 0.05.

These physics-based features are concatenated with the molecular descriptors to form the input feature vector. The dataset is split into training and testing sets, and the features are standardized using `StandardScaler` to ensure consistent scaling.

10.3.2 Physics-Informed Neural Network (PINN) Model

1. **Model Architecture:** The PINN consists of a fully connected neural network with four layers:
    - An input layer with dimensionality equal to the number of features.
    - Two hidden layers with 256 and 128 neurons, respectively, using ReLU and LeakyReLU activation functions.
    - An output layer with a single neuron to predict binding affinity.

Additionally, the model includes three learnable parameters (a, b, and c) that scale the contributions of the physics-based features (MM_Energy, Solvation_Energy, and Entropy) to the predicted binding affinity.

2. **Physics-Informed Loss Function:** The loss function combines the Mean Squared Error (MSE) between the predicted and actual binding affinity values with a physics-based regularization term. The regularization term penalizes deviations from the physical relationship:

$$a \cdot \text{MM\_Energy} + b \cdot \text{Solvation\_Energy} + c \cdot \text{Entropy} \approx \text{Predicted Affinity}.$$

The total loss is computed as:

$$\text{Loss} = \text{MSE}(\text{Predicted Affinity}, \text{Actual Affinity}) + 0.05 \cdot \text{Physics Regularization}.$$

This ensures that the model's predictions are consistent with both the data and the underlying physical laws.

3. **Training Process:** The model is trained using the Adam optimizer. A learning rate scheduler reduces the learning rate by a factor of 0.5 every 200 epochs to improve convergence. The training data is divided into mini-batches of size 64, and the model is trained for 1000 epochs. The loss is tracked and printed every 100 epochs to monitor progress.

10.3.3 Model Evaluation and Visualization

After training, we test the model's performance on the test dataset. First, the trained model predicts the binding affinity values for the test data. Next, we plot the training loss history to see how well the model learned over time. The x-axis shows the number of epochs, and the y-axis shows the loss value. Finally, we create a scatter plot to compare the predicted binding affinity values with the actual values. A diagonal line is added to the plot to show where perfect predictions would fall. The x-axis represents the actual values, and the y-axis represents the predicted values.

10.3.4 Differences Between PINNs and GNNs

Following are the key differences between PINNs and GNNs:

- **Incorporation of Physical Laws:** PINNs explicitly incorporate physical laws into the loss function, making sure that the model's predictions are consistent with domain knowledge. GNNs, on the other hand, rely solely on data-driven learning and do not enforce physical constraints.

- **Loss Function:** PINNs use a composite loss function that includes both data-driven (MSE) and physics-based terms. GNNs typically use a purely data-driven loss function, such as MSE or cross-entropy.

- **Model Interpretability:** PINNs provide interpretable parameters (e.g., a, b, and c) that quantify the contributions of physical features to the predictions. GNNs, while powerful, are often treated as black-box models with limited interpretability.

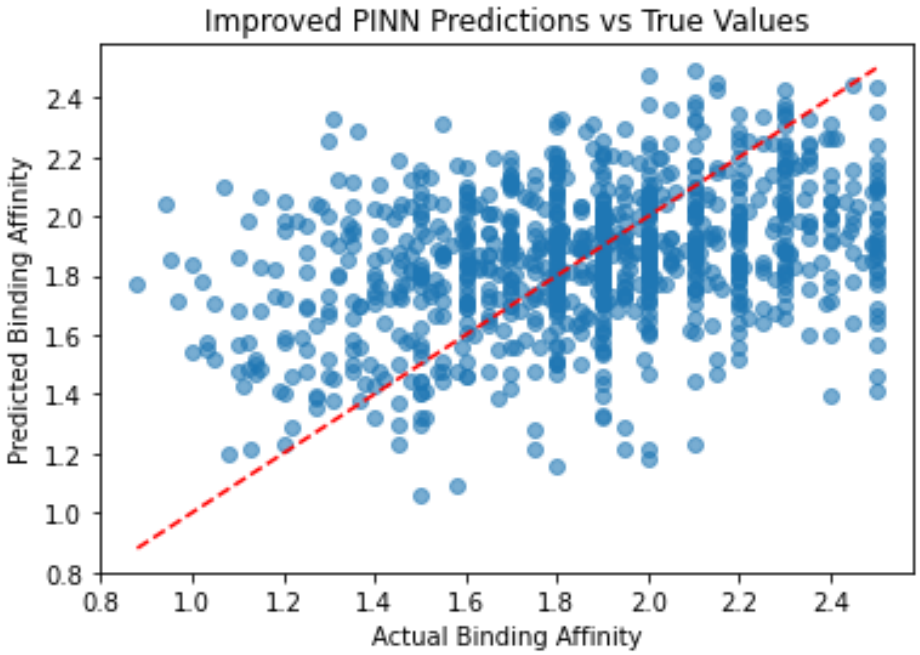


Figure 10.3: PINN vs True Values

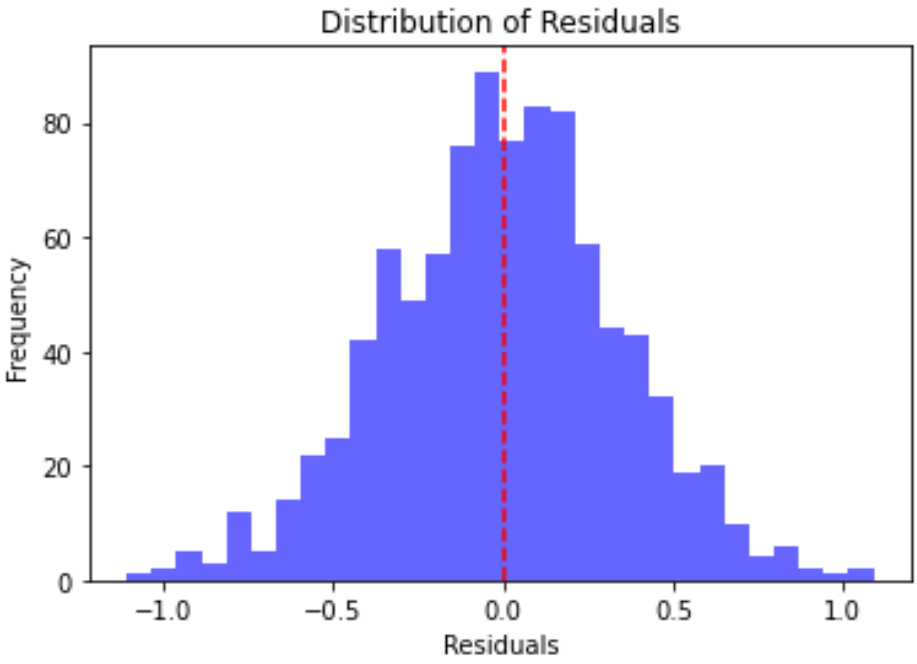


Figure 10.4: Distribution of Residuals for PINN

10.4 Conclusion

The method described in this section provides a complete framework for predicting protein-ligand binding affinity using Physics-Informed Neural Networks (PINNs). By including physical laws in the loss function, the PINN model makes predictions that are both data-driven and consistent with scientific principles. The use of physics-based features and regularization terms improves the model's interpretability and ability to generalize, making it a useful tool for drug discovery and studying molecular interactions. The training and evaluation results show that this approach works well, especially when compared to traditional GNNs.

For GNNs, the results were:

- MAE: 0.2782
- RMSE: 0.3528
- R^2: -0.0367

For PINNs, the results were:

- MAE: 0.2726
- RMSE: 0.3447
- R^2: 0.0106

The improvement in the R^2 value for PINNs shows that incorporating physical laws helps the model make better predictions. This makes PINNs a stronger choice for tasks like drug discovery and molecular interaction studies.

It is important to note that this study used very basic neural network model. The goal was to compare a basic Graph Neural Network (GNN) model to a basic PINN model. The accuracy and results can be significantly improved by making the following adjustments:

Using advanced activation functions can help reduce validation loss. Choosing a more robust loss function, especially for datasets with outliers, can enhance performance. Techniques like learning rate scheduling or alternative optimizers can also lead to better results. Additionally, improving the model architecture by adding batch normalization, dropout, or increasing the depth of the network can further boost performance. These improvements can help make the models more accurate and reliable for predicting protein-ligand binding affinity.

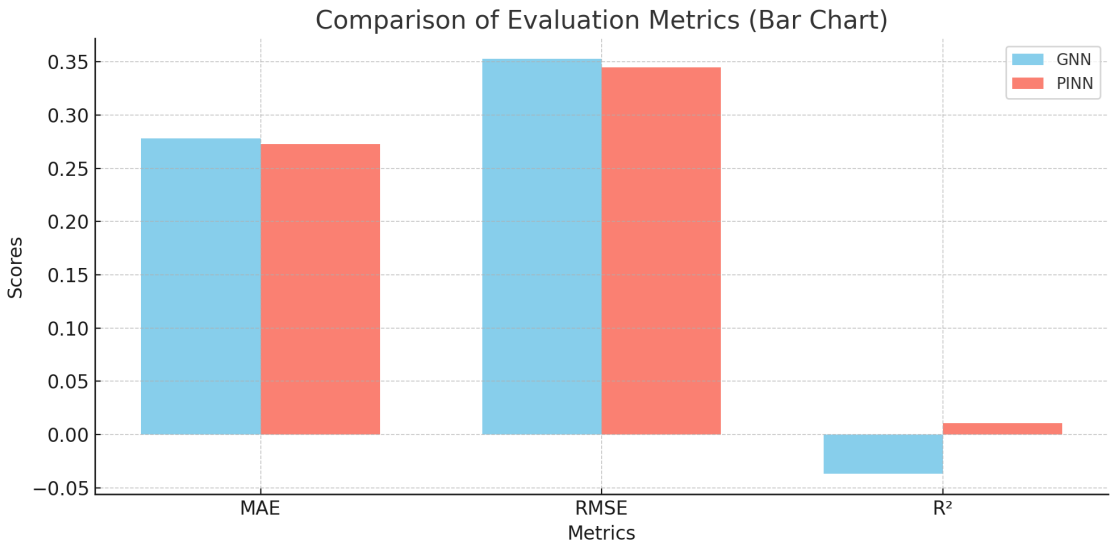


Figure 10.5: Evaluation Metrics (GNN vs PINN)

Chapter 11

Conclusion

In this thesis, we explored the fundamental concepts and methodologies underlying Graph Neural Networks (GNNs). We began by examining how GNNs operate at both the node and graph levels, along with common challenges such as over-smoothing, which can reduce the effectiveness of deeper models.

We then delved into graph representation learning techniques, including the encoder-decoder framework and random-walk-based methods like Node2Vec, which are useful for learning meaningful node and graph embeddings. Additionally, we investigated the role of multi-relational data and knowledge graphs, which are essential for representing complex relationships in structured data.

Building on this foundation, we applied these techniques to a real-world problem in molecular docking. We reviewed existing computational methods in the field and focused on graph-based approaches. In particular, we introduced P2Rank, a machine learning tool that predicts ligand-binding sites on protein structures.

In the final part of the thesis, we designed and trained an improved GNN model to group and rank high-scoring points based on ligandability scores and spatial closeness. We used a function called `cluster_points` to group points that were both close together and had high ligandability scores. Then, we ranked these groups using the `rank_pockets` function, which added up the ligandability scores to find the most promising pockets. Finally, we used the `visualize_ranked_pockets` function to create 3D visualizations, making it easier to see and

understand the results.

In addition to the core components of the thesis, we conducted a series of supplementary studies and developed various Python scripts to explore alternative methodologies and insights. While not central to the main body of the thesis, these studies address specific challenges and offer innovative perspectives that contribute to the broader goal of advancing drug discovery. Each approach presents opportunities for further refinement and has the potential to inspire future research directions.

The primary motivation behind these supplementary efforts was to deepen our understanding of the key concepts discussed and to creatively apply them in solving real-world problems. The insights gained through these studies serve as a foundation for ongoing and future exploration in this field.

All Python scripts used throughout this research, including those related to the supplementary studies, are available in the following GitHub repository: https://github.com/YashKarampuri/MS-Thesis-Supplementary.

Bibliography

[1] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[2] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems*, 30, 2017.

[3] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S. Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2020.

[4] William L. Hamilton. *Graph Representation Learning*. Morgan & Claypool Publishers, 2020.

[5] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 855–864, 2016.

[6] Zexi Huang, Arlei Silva, and Ambuj Singh. A broader picture of random-walk-based graph embedding. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 685–695, 2021.

[7] Bing-Jie Sun, Huawei Shen, Jinhua Gao, Wentao Ouyang, and Xueqi Cheng. A non-negative symmetric encoder-decoder approach for community detection. In *Proceedings of the 2017 ACM Conference on Information and Knowledge Management*, pages 597–606, 2017.

[8] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. *Advances in Neural Information Processing Systems*, 27, 2014.

[9] Xu K, Hu W, Leskovec J, Jegelka S. How powerful are graph neural networks?. arXiv preprint arXiv:1810.00826. 2018 Oct 1.

[10] Chris McCormick. Word2vec tutorial-the skip-gram model. Apr-2016. [Online]. Available: `http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model`, 2016.

[11] Veličković P, Cucurull G, Casanova A, Romero A, Lio P, Bengio Y. Graph attention networks. arXiv preprint arXiv:1710.10903. 2017 Oct 30.

[12] Qiu J, Dong Y, Ma H, Li J, Wang K, Tang J. Network embedding as matrix factorization: Unifying deepwalk, line, pte, and node2vec. InProceedings of the eleventh ACM international conference on web search and data mining 2018 Feb 2 (pp. 459-467).

[13] Moore NS, Cyr EC, Ohm P, Siefert CM, Tuminaro RS. Graph neural networks and applied linear algebra. SIAM Review. 2025 Mar 31;67(1):141-75.

[14] Clemens Isert, Kenneth Atz, and Gisbert Schneider. Structure-based drug design with geometric deep learning. *Current Opinion in Structural Biology*, 79:102548, 2023.

[15] Rocío Mercado, Tobias Rastemo, Edvard Lindelöf, Günter Klambauer, Ola Engkvist, Hongming Chen, and Esben Jannik Bjerrum. Graph networks for molecular design. *Machine Learning: Science and Technology*, 2(2):025023, 2021.

[16] Jonathan M. Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andres Cubillos-Ruiz, Nina M. Donghia, Craig R. MacNair, Shawn French, Lindsey A. Carfrae, Zohar Bloom-Ackermann, et al. A deep learning approach to antibiotic discovery. *Cell*, 180(4):688–702, 2020.

[17] Farzan Soleymani, Eric Paquet, Herna Viktor, Wojtek Michalowski, and Davide Spinello. Protein–protein interaction prediction with deep learning: A comprehensive review. *Computational and Structural Biotechnology Journal*, 20:5316–5341, 2022.

[18] Bara A. Badwan, Gerry Liaropoulos, Efthymios Kyrodimos, Dimitrios Skaltsas, Aristotelis Tsirigos, and Vassilis G. Gorgoulis. Machine learning approaches to predict drug efficacy and toxicity in oncology. *Cell Reports Methods*, 3(2), 2023.

[19] Agarwal S, Mehrotra RJ. An overview of molecular docking. JSM chem. 2016 May;4(2):1024-8.

[20] Jiang H, Wang J, Cong W, Huang Y, Ramezani M, Sarma A, Dokholyan NV, Mahdavi M, Kandemir MT. Predicting protein–ligand docking structure with graph neural network. Journal of chemical information and modeling. 2022 Jun 14;62(12):2923-32.

[21] Hu W, Liu Y, Chen X, Chai W, Chen H, Wang H, Wang G. Deep learning methods for small molecule drug discovery: A survey. IEEE Transactions on Artificial Intelligence. 2023 Mar 3;5(2):459-79.

[22] Krivák R, Hoksza D. P2Rank: machine learning based tool for rapid and accurate prediction of ligand binding sites from protein structure. Journal of cheminformatics. 2018 Dec;10:1-2.

[23] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. Advances in neural information processing systems. 2017;30.

[24] Hu W, Liu Y, Chen X, Chai W, Chen H, Wang H, Wang G. Deep learning methods for small molecule drug discovery: A survey. IEEE Transactions on Artificial Intelligence. 2023 Mar 3;5(2):459-79.

[25] Chen LY, Li YP. Machine Learning Applications in Chemical Kinetics and Thermochemistry. InMachine Learning in Molecular Sciences 2023 Oct 2 (pp. 203-226). Cham: Springer International Publishing.

[26] Gan S, Cosgrove DA, Gardiner EJ, Gillet VJ. Investigation of the use of spectral clustering for the analysis of molecular data. Journal of chemical information and modeling. 2014 Dec 22;54(12):3302-19.

[27] Yang Z, Ding M, Huang T, Cen Y, Song J, Xu B, Dong Y, Tang J. Does negative sampling matter? a review with insights into its theory and applications. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2024 Feb 29.

[28] Soleymani F, Paquet E, Viktor H, Michalowski W, Spinello D. Protein–protein interaction prediction with deep learning: A comprehensive review. Computational and Structural Biotechnology Journal. 2022 Jan 1;20:5316-41.

[29] Murphy R, Srinivasan B, Rao V, Ribeiro B. Relational pooling for graph representations. InInternational Conference on Machine Learning 2019 May 24 (pp. 4663-4673). PMLR.

[30] Cuomo S, Di Cola VS, Giampaolo F, Rozza G, Raissi M, Piccialli F. Scientific machine learning through physics–informed neural networks: Where we are and what's next. Journal of Scientific Computing. 2022 Sep;92(3):88.

[31] Roy AA, Dhawanjewar AS, Sharma P, Singh G, Madhusudhan MS. Protein Interaction Z Score Assessment (PIZSA): an empirical scoring scheme for evaluation of protein–protein interactions. Nucleic acids research. 2019 Jul 2;47(W1):W331-7.