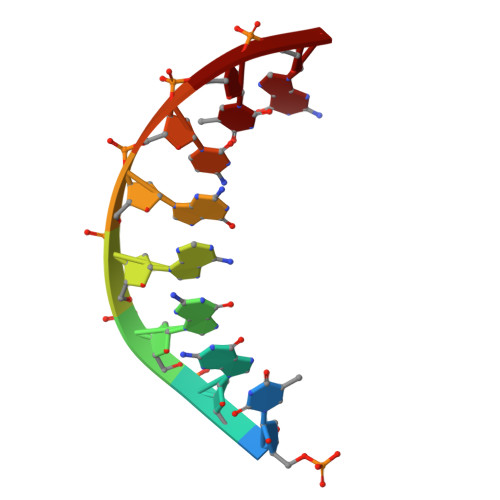
**PPI\_GNN**

PPI GNN is a graph-based approach using the 3D structures of proteins to assess the interaction between protein pairs.



**Step1: Creating the database of PPI**

1.1) Download data

* Positive interaction pairs: HuRI database’s H-I-05 was used for obtaining positive protein interaction pairs (<http://www.interactome-atlas.org/download>)
* Negative interaction pairs: Negatome database’s PDB-Stringent contains negative protein interaction pairs (<https://mips.helmholtz-muenchen.de/proj/ppi/negatome/>)
* In summary, interactions here which is our target variable is binary [0 or 1]

1.2) Convert the protein representation (Ensembl) to UniProt ID format

* The HuRI data needs to be converted from Ensembl to UniProt Swiss (UniProt KB) format. Using the ‘<https://www.uniprot.org/id-mapping>’ website converter, select the ‘tsv’ file containing all the protein Ensembl names, choose ‘Genome annotation databases -> Ensembl’ for the field ‘from’ and ‘UniProt -> UniProtKB’ for the ‘to’ field.
* Once the job is done, select the ‘Reviewed (Swiss-Prot)’ option in the top-left corner, to obtain the best 1-1 mapping. Next, select the ‘download’ option. Choose the Excel and non-compressed download option.
* Using the **‘1\_HuRI\_EnsemblToUniProt+Negatome+CombineDatasets’ ipynb** file here, convert and club all positive and negative protein interaction pairs in UniProt format, in a single CSV file.
* Add a column to indicate the interaction nature (positive is 1 and negative is 0)

1.3) Convert the UniProt IDs to PDB

* Using the ‘<https://www.uniprot.org/id-mapping>’ website converter, select the ‘txt’ file containing the protein UniProt IDs and the website converts it to PDB IDs. IT IS NOT A 1-1 MAPPING!
* Once the job is done, select the ‘download’ option. Choose the Excel and non-compressed download option.
* Using the **‘2\_UniProt\_to\_PDB\_ID’ ipynb** file here, convert the UniProt IDs to all possible PDB ID interaction pairs
* NOTE THAT NOT ALL UNIPROT IDS HAVE A PDB FILE (3D STRUCTURE). THIS IS AN ONGOING RESEARCH. FOR MORE INFORMATION IN THIS DIRECTION, SEE ALPHAFOLD BY GOOGLE DEEPMIND. Such cases are dropped from the study. We will consider interactions where both proteins’ PDB files are present.
* Some pairs may have been repeated and duplicates will be removed. On running the above script, an ‘interaction\_pdb\_db\_uniquepairs.csv’ will be created. This CSV file has 5 columns, ['protein1\_pdb', 'protein2\_pdb', 'protein1\_uniprot', 'protein2\_uniprot', 'interaction\_type']

1.4) Download the PDB files for all PDB IDs present

* One UniProt ID may have multiple PDB files. Since I am new to this domain, I consider all PDBs of a given UniProt ID show the same interaction with the proteins of another UniPort ID’s PDB files
* Next, run the **‘3\_PDB\_ID\_to\_3DStructures**.**ipynb’**. The first part of this ipynb is to download the required unique PDB files, along with checking the usage of space. The user can control this in two steps:
  + Since a given pair of UniProt IDs may have multiple interaction combinations (due to more than 1 PDB file per UniProt ID), the user can specify the maximum number of PDB pairs to be considered per pair of UniProt IDs
  + Since the PDB files may be large, the user can specify the total maximum space to be used by each PDB pair
* Once the user is satisfied, they can download the PDB files. Please note that not all proteins have a PDB file. If one of the proteins in the specified PDB pair is not present, then that pair is discarded.
* PLEASE NOTE THAT THE ABOVE STEPS WILL TAKE AROUND **3 hours** TO COMPLETE!
* The second part of the script is to construct PyGraphs of the proteins and this is done using the ‘Graphein’ library [<https://github.com/a-r-j/graphein>]

{Other useful links:

1. <https://github.com/a-r-j/graphein/issues/130>
2. <https://colab.research.google.com/github/a-r-j/graphein/blob/master/notebooks/atom_graph_tutorial.ipynb>

* Few things to note here:
  + We are using the graphein library to construct graphs from PDB files [On an informal note, uff thank god that a major part is taken care of!]
  + By default, function construct\_graph returns a networkx graph, not a torch geometric graph. We will use a function provided by them (Graphein) to do this [Discussed here]
  + Graphein offers multiple ways to construct a graph
    - Nodes: Can be either atoms or residues/amino acids
    - Based on what is the node, the node features change. Currently, we are keeping things super simple by making atoms as nodes and the amino acid or residue as the node feature [Encoded using Meilier’s method]
    - If we choose residues as the nodes, then we can use ProtBert or ESM embedding as node features
    - With the default settings, ‘format\_convertor = GraphFormatConvertor('nx', 'pyg', verbose='gnn', columns=None); print('Without columns mentioned:', vars(format\_convertor)), we get Without columns mentioned: {'src\_format': 'nx', 'dst\_format': 'pyg', 'columns': ['edge\_index', 'coords', 'name', 'node\_id'], 'type2form': {'atom\_type': 'str', 'b\_factor': 'float', 'chain\_id': 'str', 'coords': 'np.array', 'dist\_mat': 'np.array', 'element\_symbol': 'str', 'node\_id': 'str', 'residue\_name': 'str', 'residue\_number': 'int', 'edge\_index': 'torch.tensor', 'kind': 'str'}}
    - The above IS NOT WHAT WE WANT as node features or Meiler’s embedding are MISSING [Notice elements of ‘columns’ above, no mention of node features]
    - To make sure we have both node features (Meilers embeddings) and edge\_index, make sure params\_dict = {'src\_format': 'nx', 'dst\_format': 'pyg', 'columns': ['edge\_index', 'coords', 'name', 'node\_id', 'meiler']}; format\_convertor = GraphFormatConvertor(\*\*params\_dict). When you do this, the PyTorch graph will have Data(edge\_index=[2, 2\*number\_of\_edges], node\_id=[number\_of\_nodes], coords=[number\_of\_nodes, 3], name=name\_of\_protein, num\_nodes=number\_of\_nodes, x=[number\_of\_nodes, 7])
    - Note that Meiler’s embeddings are a static 7-feature embedding:
      * Encodes physicochemical properties: hydrophobicity, volume, polarity, etc.
      * Static, i.e., not context aware [think of it like word embeddings which are static]. A given amino acid has a standard 7D representation that is uniform throughout [universal]
    - When we run the above command, actually there is no direct X in it, it’s under the name of ‘meiler’. We need to manually do:

pyg\_graph.x = pyg\_graph.meiler

* + - To know more about the PDB to networkx graph and convert networkx to pyg\_graph function (you can’t use default functions from torch\_geometric or networkx for this, we will use the function from graphein) read:   
      a) <https://graphein.ai/notebooks/pscdb_baselines.html?highlight=graphformatconvertor>  
      b) <https://graphein.ai/notebooks/atom_graph_tutorial.html>
    - Make sure the pyg\_graph has two tensors, X and edge\_index. If yes, then we are all set to convert all the PDB files we have and save them as a .pt file!
    - In summary:
      * Nodes: Atoms
      * Node features: Meiler embeddings
      * Edges and Edge features: Using ‘add\_atomic\_edges‘ from Graphein library
    - We also will create metadata.csv which will contain only interactions for which the graphs exist for both proteins [Used in the next step]

We have our dataset!

**Step2: Modeling PPIGNN**

For this part, run the ‘**4\_GNN\_part.ipynb**’ notebook

2.1) Load dataset

* In this step, we load the metadata for protein-protein interactions (PPI), filter it to ensure valid structural graphs exist for both proteins in a pair, and then perform a **stratified train-test split** based on **UniProt IDs** to ensure no data leakage and reliable evaluation on unseen protein pairs.
* We begin by loading a CSV file (metadata.csv) that contains information about protein pairs (protein1\_pdb, protein2\_pdb) and their interaction types (e.g., binding vs non-binding). This metadata includes both PDB and UniProt identifiers for the proteins.
* Filter for Available Graph Files:
  + Graphs are precomputed and stored as separate files.
  + We scan the graph directory to get a list of all graph files available.
  + Only pairs where **both protein1\_pdb and protein2\_pdb** have corresponding graph files are retained.
  + This ensures that we only train on and evaluate pairs that can be represented as input graphs.
  + The filtered data frame is saved as filtered\_interactions.csv.

2.2) Stratified Train-Test Split (No Data Leakage)

* To prevent data leakage and ensure that the model is tested on **entirely different protein pairs**, we:
  + Construct a **canonical pair ID** using **UniProt IDs** (not PDBs), ensuring consistent identification of a protein pair regardless of order.
  + Drop duplicate pair IDs and perform a **stratified split** based on the interaction type to preserve class distribution.
  + Use 80% of the data for training and 20% for testing.
  + After splitting, we map the original full metadata back to create train\_df and test\_df.
  + This setup ensures that the model is not evaluated on any protein pair (or reversed pair) it has already seen during training, making for a fair and generalizable evaluation.

2.3) Model building and training

### Model Architecture: PPIGCN: The architecture is designed to handle *pairs* of protein graphs. Each protein in the pair is encoded via a Graph Neural Network (GNN), and their representations are combined to predict the likelihood of interaction. **Key points of the model:**

### Each protein graph is passed through a **Graph Convolutional Network (GCN)** to extract features.

### The final node-level features are aggregated (e.g., by mean pooling) into a graph-level embedding.

### The two protein embeddings are concatenated and passed through a feedforward neural network to classify interaction.

### Training & Validation Loop with Cross-Validation: You perform **5-fold stratified cross-validation** to evaluate the model performance across different splits. The interaction labels are stratified to preserve the class distribution in each fold. The workflow for each fold is described below:

### Split the dataset into training and validation sets (with StratifiedKFold).

### Load graph pairs and labels using your custom PPI\_GraphDataset class.

### Initialize the model and move it to GPU (if available).

### Use **Binary Cross Entropy Loss (BCELoss)** as the criterion.

### Use **Adam optimizer** to update the weights.

### At each epoch:

### Run the training loop and accumulate predictions.

### Run validation loop and compute evaluation metrics: AUROC, Precision, Recall, and F1 Score

* The function test\_model is designed to evaluate a trained model on a test dataset. It performs inference using the model, generates predictions, and computes metrics for evaluating the model's performance in terms of binary classification

**CHALLENGES FACED**

1. Gathering the perfect dataset: I searched from many sources and finally chose the above two
2. Understanding that the two datasets were in two different protein formats. I then made sure to convert them to UniProt
3. UniProt to PDB was another challenge, especially in terms of ‘web-scrapping’ them, checking their sizes as this project is supposed to be a ‘Colab free’-friendly project
4. Getting the right library to convert the PDBs to graphs was another big challenge. I searched for methods/libraries to convert a PDB file to a graph and finally found graphein to work so far; tried ‘proteingraph’ and DGLbut both didn't work Initially I tried making my own graphs but realized it was not as straightforward as it seemed and would require ready to use libraries keeping the 'biology' intact

### **Future Work:**

1. **Expand dataset sources**: Investigate and integrate more diverse and large-scale datasets for both positive and negative interactions to create a more robust and generalized model.
2. **Incorporate graph edges-associated data**: Add edge features the **bond type** (single, double, etc.) and **ring status** (whether the bond is part of a ring or not) will be **edge features** in your protein graph [<https://graphein.ai/notebooks/atom_graph_tutorial.html>]  
   **from** graphein.protein.edges.atomic **import** add\_atomic\_edges**,** add\_bond\_order**,** add\_ring\_status

params\_to\_change **=** **{**"granularity"**:** "atom"**,** "edge\_construction\_functions"**:** **[**add\_atomic\_edges**,** add\_bond\_order**,** add\_ring\_status**]}**

config **=** ProteinGraphConfig**(\*\***params\_to\_change**)**

g **=** construct\_graph**(**config**=**config**,** pdb\_code**=**"3eiy"**)**

1. **Change nodes from atoms to residues:** Following the tutorial here [<https://graphein.ai/notebooks/residue_graphs.html>]
2. **Incorporate AlphaFold structures**: Integrating AlphaFold-predicted protein structures for cases where PDB data is missing, thereby enhancing the dataset and covering more protein interactions.
3. **Try ProtBERT for node features**: Explore using **ProtBERT**, a language model specifically trained on protein sequences, to generate embeddings for residues or amino acids as node features, possibly improving graph representations and model accuracy.
4. **Use attention-based GNNs**: Experiment with **attention-based Graph Neural Networks** (e.g., Graph Attention Networks, GAT), which may better capture important relationships between proteins, allowing the model to focus on more relevant interaction sites or structures.
5. **Incorporate multi-modal data**: Experiment with incorporating additional sources of data, like molecular weight, and gene expression data, to improve the model’s ability to predict protein-protein interactions. [For more information on this: <https://graphein.ai/notebooks/residue_graphs.html>; see <https://graphein.ai/notebooks/residue_graphs.html#Built-in-Graph-Annotation-Functions> and <https://graphein.ai/notebooks/residue_graphs.html#Built-in-Graph-Annotation-Functions>]
6. **Use dynamic graph models**: Explore temporal changes in protein interactions over time and how dynamic graphs could improve the understanding of protein interaction evolution or changes in disease states.