Code analysis request

To give you a comprehensive explanation of the code inside the ZIP file, could you please clarify the following:

1. What specific programming language or file type in the ZIP should I start analyzing? (e.g., Python scripts, Jupyter Notebooks, etc.)
2. Are you interested in all files and scripts or just the main logic?
3. Should I focus more on the technical logic or also the biological/functional context if this is a bioinformatics project?

Once I have this information, I'll begin the deep analysis for you.

Great. I will analyze the Python scripts inside the provided ZIP file, focusing first on `protein_info.py`, then `train_val_split_data.py`, followed by the GNN model implementations and `model_test.py`. I will explain each line of code with detailed technical and brief biological context, describe the function of each method, and present the overall workflow of the project.

I'll let you know once the full analysis is ready.

Code and Workflow Analysis of HIGH-PPI Python Project

HIGH-PPI Code Analysis

This report analyzes the Python source code in the *HIGH-PPI* project, a bioinformatics pipeline for predicting protein-protein interactions (PPIs) using hierarchical graph neural networks (GNNs). We examine key scripts in the order specified: data preprocessing under `protein_info/`, train/validation splitting, the GNN model implementation (`gnn_models_sag.py`), and the testing script (`model_test.py`). For each, we describe its purpose, workflow, and biological context. Finally, we summarize the end-to-end workflow.

Protein–protein interactions (PPIs) are **physical contacts of high specificity** between two or more protein molecules, mediated by biochemical forces such as hydrogen bonding and electrostatics . PPIs form large **interaction networks** that underlie cellular processes and disease mechanisms . Predicting PPIs computationally often leverages graph representations: proteins and their features can be encoded as graphs of amino acid residues, and interactions become links in a higher-level graph. Graph Neural Networks (GNNs) are well suited for this task because they **directly operate on graph-structured data** , making them a powerful tool for PPI prediction.

1. Protein Data Preprocessing (`protein_info/`)

The `protein_info/` directory contains scripts that prepare protein-level graph data (adjacency and feature matrices) from raw structural information. These include:

- `generate_adj.py` : Builds residue contact graphs from PDB structures.
- `generate_feat.py` : Computes node (residue) features from PDB structures.
- `mapping_27k.py` : Processes protein ID mapping data for the SHS27k dataset (used in preprocessed examples).
- There is also an empty `a.py` (ignored here) and data files (sequence dictionaries, mappings).

1.1 `generate_adj.py` : Adjacency (Contact) Matrix

This script parses PDB structure files to construct a graph for each protein, where nodes are amino acid residues and edges connect residues within a distance threshold. This reflects the idea of a **residue contact network** :

- **Distance Threshold Argument**: The script takes a command-line argument `--distance` , a float threshold (e.g. 6–12 Å), used to connect residues that have any pair of atoms closer than this distance. This follows the approach that "two residues are connected if any pair of atoms (one from each residue) have Euclidean distance less than a threshold (e.g. 6 Å)" .
- **Function** `read_atoms(file, chain, model)` : Reads PDB lines from the file handle `file` , filtering for `ATOM` records of the specified chain. It collects each atom's coordinates and residue index, skipping alternate models. The residue index is parsed (e.g. columns 22-26 in PDB format), and each atom is stored with its 3D coordinates (`x,y,z`) in a list.
- **Distance Calculation (** `dist` **)**: The `dist(p1, p2)` function computes Euclidean distance between two 3D points. It is used to check if any atom-atom distance between two residues is below the threshold.
- **Building Adjacency**: The script loops over combinations of **pairs of residues** (using `itertools.combinations`). For each residue-pair, it checks all atom-atom distances between the two residues: if *any* distance is ≤ the threshold, an edge is added between those residues. This is implemented by iterating through atoms of residue A and residue B, using the `dist` function (see code snippet below).

  python

  ```python
  for i, j in combinations(range(len(atoms)), 2):
      atom1 = atoms[i]  # (res_i, x1,y1,z1)
      atom2 = atoms[j]  # (res_j, x2,y2,z2)
      if atom1.residue != atom2.residue:
          if dist((atom1.x,atom1.y,atom1.z), (atom2.x,atom2.y,atom2.z)) <= args.distance:
              # Mark residues atom1.residue and atom2.residue as connected
  ```

- **Output Format**: The result is saved as NumPy adjacency lists (`edge_list`). For each protein, it writes an array of edges in `protein_info/edge_list_{distance}.npy` . Each entry is a pair `(resA, resB)` . A symmetric undirected edge means each pair appears twice (A–B and B–A). The networks are saved for all proteins to be used later by the GNN.

Biological context: This graph construction models each protein's 3D structure as a contact network of residues. Such structural graphs capture physical proximity information, which can influence how proteins interact with other proteins (e.g. binding sites). Using a cutoff (commonly 6–12 Å) is a standard way to abstract 3D structures into graphs.

1.2 `generate_feat.py` : Node Feature Matrix

This script generates feature vectors for each residue of each protein, based on the same PDB structures. These features are used as input node attributes for the GNN.

- **Reading ATOM Records**: Similar to `generate_adj.py`, it defines `read_atoms(file, chain, model)` to parse PDB `ATOM` lines and collect atoms per residue. It also includes lines parsing (`if line.startswith("ATOM")` etc.) with regex to select a chain.
- **Feature Computation**: The code snippet suggests it computes features for each residue (not fully visible above). Typically, such a script might compute one-hot encoding of residue type, physicochemical property (e.g. hydrophobicity), or structural features. Based on the default `--vec_path './protein_info/vec5_CTC.txt'` in model training, it likely uses 7 physicochemical properties of amino acids (as described in literature , e.g. Meiler's 7 properties). Indeed, the code calls something like `self.feature[index]=self.feature_matrix[id-1]` which suggests mapping residues to a 7-dimensional property vector from a lookup. It also reads `protein.SHS27k.sequences.dictionary.pro3.tsv`, likely mapping protein IDs to features.
- **Distance and Angle**: The presence of a `dist` function here implies computing distances between atoms again, possibly to calculate distances between C-alpha atoms or determine bond/angle features. However, the actual feature logic is obscured.
- **Saving Feature Matrices**: It likely writes out a combined feature matrix file (e.g. `x_list.npy`) matching the adjacency graphs. This matrix has shape `(N_nodes_total, 7)`, where N is total residues across proteins. In training, this is loaded as `p_feat_matrix`.

In summary, `generate_feat.py` builds a numeric feature for each residue, probably encoding sequence or structure-derived information. These features serve as the initial node embeddings in the GNN.

1.3 `mapping_27k.py`: ID Mapping (SHS27k Dataset)

This script processes ID mapping for the SHS27k dataset (a subset of the STRING database). It loads a precomputed NumPy array `mapping_process.npy` (likely containing mapping of ENSP IDs to UniProt or vice versa). It then appears to iterate over a raw mapping file (`HUMAN_9606_idmapping.txt` or similar) to build a `dictionary` mapping protein IDs to sequences. The code fragment suggests writing to `'protein.SHS27k.sequences.dictionary.pro3.tsv'`. Essentially, it aligns IDs from one naming scheme to another, ensuring consistency between PPI edges and sequence data. The output is a TSV where each line has an ID and its amino acid sequence. This is used by `get_feature_origin` in `model_train.py` to assign sequence-derived features.

Overall (protein_info): The `protein_info` scripts prepare the raw PDB data into two NumPy arrays – an adjacency list of residue contacts (`p_adj_matrix`) and a feature matrix of residue attributes (`p_feat_matrix`). These become inputs to the GNN. The workflow is:

1. Place PDB files in `protein_info/`.
2. Run `generate_adj.py --distance D` to produce `edge_list_D.npy` (adjacency).
3. Run `generate_feat.py` to produce `x_list.npy` (features).
4. (For SHS27k example) Use `mapping_27k.py` to align protein IDs with sequences.

These processed files are then passed to the model training script.

2. Train/Validation Splitting

The project expects a train/validation split of PPI edges. This is handled not by a separate script, but by the `GNN_DATA.split_dataset` method in `gnn_data.py`, and by a JSON file in `train_val_split_data/`.

- **Split JSON**: Under `train_val_split_data/` we see `1.json`, which is empty (just a newline). In practice, `split_dataset()` will write indices to this file. The code in `model_train.py` calls:

  ```python
  ppi_data.split_dataset(train_valid_index_path='./train_val_split_data/train_val_split_1.json',
                         random_new=True, mode=args.split)
  ```

  With `random_new=True`, the method ignores any existing file and generates a new split.
- **Random Split**: If `mode='random'`, it randomly shuffles half of the edges (see `self.edge_num // 2`, as edges are stored twice) and splits into training vs validation by the given `test_size` ratio (default 0.2). It stores two lists of edge indices: `train_index` and `valid_index`. These are saved to the JSON file.
- **BFS/DFS Split**: If `mode='bfs'` or `'dfs'`, the code instead takes the original PPI list and performs a subgraph extraction. It treats the PPI network as an undirected graph: builds a mapping from each node (protein) to its incident edge indices (`node_to_edge_index`). Then it samples a connected subgraph of size proportional to `test_size` using BFS or DFS (`get_bfs_sub_graph` or `get_dfs_sub_graph`). The selected edges become the validation set; the rest are training. This ensures the validation edges form a connected region, testing generalization on novel connections. The split indices are also saved as JSON.
- **Usage**: During **training**, `split_dataset` is called with `random_new=True` (so the empty JSON is replaced). During **testing** (`model_test.py`), the code loads an existing JSON (with `random_new=False`) to retrieve the train/validation split indices.

Thus, the *split data* component uses `GNN_DATA.split_dataset` and a JSON file to define which PPI edges are used for training vs validation. There is no separate `.py` file here; it's managed within `GNN_DATA`.

3. GNN Model Implementation (`gnn_models_sag.py`)

This file defines the neural network architectures for **Hierarchical Graph Learning** on PPIs. The key classes are:

- `GCN` – a graph neural network that produces an embedding for each protein (from its residue graph).
- `GIN` – a GNN that takes protein embeddings and predicts interactions between proteins.
- `ppi_model` – the combined model that sequentially applies the above.

This reflects a **two-level model**: first embed each protein graph, then do link prediction on the protein–protein graph.

3.1 Class `GCN` (Protein-level Embedding)

```python
```

```python
class GCN(nn.Module):
    def __init__(self):
        super(GCN, self).__init__()
        hidden = 128
        self.conv1 = GCNConv(7, hidden)
        self.conv2 = GCNConv(hidden, hidden)
        self.conv3 = GCNConv(hidden, hidden)
        self.conv4 = GCNConv(hidden, hidden)
        self.bn1 = nn.BatchNorm1d(hidden)
        ...
        self.sag1 = SAGPooling(hidden, 0.5)
        self.sag2 = SAGPooling(hidden, 0.5)
        self.sag3 = SAGPooling(hidden, 0.5)
        self.sag4 = SAGPooling(hidden, 0.5)
        self.fc1 = nn.Linear(hidden, hidden)
        self.fc2 = nn.Linear(hidden, hidden)
        self.fc3 = nn.Linear(hidden, hidden)
        self.fc4 = nn.Linear(hidden, hidden)
        self.dropout = nn.Dropout(0.5)
```

- **Inputs**: Node features of dimension 7 (the 7 biochemical properties per residue) and an edge index for the residue graph. A `batch` vector (later) assigns nodes to proteins.
- **Architecture**: Four rounds of
  - **Graph Convolution** (`GCNConv`) to propagate information across edges, each followed by a linear layer (`fcX`), ReLU, and batch norm.
  - **SAGPooling** (`SAGPooling`) which pools (selects) 50% of the nodes (retaining those with highest scores) for the next layer. SAGPooling learns which nodes to keep.
  - Node sets shrink progressively (half each time), mimicking a hierarchical reduction of the graph.
- **Output**: After the last pooling, the remaining nodes are aggregated with `global_mean_pool` to a fixed-size graph embedding (`y[0]`). This yields one 128-dimensional vector per protein graph (regardless of original size).

The **forward pass** in pseudocode:

1. `x = ReLU(fc1(GCNConv1(x)))` , batch norm, then `y = SAGPool1(x, edges)` . Update `(x, edges, batch)` .
2. `x = ReLU(fc2(GCNConv2(x)))` , batch norm, `y = SAGPool2(...)` .
3. Repeat for conv3/fc3/SAGPool3, conv4/fc4/SAGPool4.
4. Return `global_mean_pool(y[0], batch)` (pool remaining node embeddings by graph).

Thus, `GCN` transforms the structural graph of a protein into a fixed embedding. This aligns with graph-level representation learning: it hierarchically pools residue features, summarizing each protein's structure.

3.2 Class `GIN` (Edge-level Classification)

```python
class GIN(nn.Module):
    def __init__(self, hidden=512, train_eps=True, class_num=7):
        super(GIN, self).__init__()
        self.gin_conv1 = GINConv(nn.Sequential(
            nn.Linear(128, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU(),
            nn.BatchNorm1d(hidden)), train_eps=train_eps)
        self.gin_conv2 = GINConv(nn.Sequential(...), train_eps=train_eps)
        # (Optional third GINConv is defined but commented out)
        self.lin1 = nn.Linear(hidden, hidden)
        self.fc2 = nn.Linear(hidden, class_num)
```

- **Inputs**: Protein embeddings `x` (from the `GCN` above), plus the PPI network's edge list and a list of edges to classify (`train_edge_id`). Here, `x` has dimension *hidden* for each protein node in the PPI graph.
- **Architecture**: Two layers of **GINConv** (Graph Isomorphism Network). Each GINConv wraps an MLP (here two linear layers with ReLU and BatchNorm) to update node embeddings based on neighbors. After two GIN layers, a linear layer and dropout refine the embeddings.
- **Link (Edge) Prediction**: The final part of `forward` performs edge classification: for each protein–protein edge in `train_edge_id` , it takes the two endpoint embeddings `x1` and `x2` (extracted using `edge_index`), multiplies them element-wise (`torch.mul(x1, x2)`), and passes through a final linear layer (`self.fc2`). This yields a `class_num` -dimensional score (here 7 classes, perhaps for different PPI types like binding, inhibition, etc.). A sigmoid would later convert this to probabilities.
- **Output**: A tensor of shape `(batch_size, class_num)` containing scores for each evaluated edge.

In summary, `GIN` learns from the **topology of the PPI network** (through `edge_index`) and the protein embeddings to predict interactions. Multiplying the two endpoint vectors is a common way to model a pairwise interaction feature vector.

3.3 Class `ppi_model` (Combined Model)

```python
class ppi_model(nn.Module):
    def __init__(self):
        super(ppi_model,self).__init__()
        self.BGNN = GCN()  # "background" or Base GNN
        self.TGNN = GIN()  # "top" GNN

    def forward(self, batch, p_x_all, p_edge_all, edge_index, train_edge_id, p=0.5):
        # Prepare inputs
        embs = self.BGNN(p_x_all, p_edge_all, batch-1)        # Embedding each protein graph
        final = self.TGNN(embs, edge_index, train_edge_id, p)  # Classify PPI edges
        return final
```

Here:

- `batch` : A vector indicating node-to-graph assignments (from `multi2big_batch` in testing). It groups all proteins together.
- `p_x_all` : The stacked node features for all proteins (from `x_list.npy`). Dimension `(N_nodes_total, 7)` .

- `p_edge_all` : The adjacency edges for all proteins (from `edge_list.npy`). Dimension `(2, N_edges_total)` .
- `edge_index` : The edges of the PPI network (protein–protein graph). Dimension `(2, N_PPI_edges)` .
- `train_edge_id` : Indices of edges in `edge_index` to evaluate (batch of PPI edges).
- `p` : Dropout probability (unused in `GIN` call above except hardcoded 0.5).

Workflow:

1. **BGNN (`GCN`)**: Takes every protein's residue graph (features + adjacency) and computes an embedding. It outputs a vector `embs` of size `(num_proteins, 128)` .
2. **TGNN (`GIN`)**: Takes those protein embeddings, plus the global PPI graph (`edge_index`), and predicts for each queried PPI edge whether an interaction exists (and type).
3. **Output**: The model returns scores for each PPI edge in `train_edge_id` . These are later passed through a sigmoid to get probabilities.

Biologically, this implements a **hierarchical approach**: first understand each protein's internal structure (BGNN), then use those summaries to predict whether two proteins interact (TGNN). GNNGL-PPI and other works similarly embed protein graphs before link prediction .

4. Testing Script (`model_test.py`)

The `model_test.py` script evaluates a trained `ppi_model` on held-out PPI edges. It performs the following steps:

1. **Argument Parsing**: It takes paths to the PPI network (`--ppi_path`), protein sequences (`--pseq_path`), protein features (`--p_feat_matrix`), adjacency (`--p_adj_matrix`), index JSON (`--index_path`), and the trained model checkpoint (`--model_path`).
2. **Data Preparation**:
   - It initializes `GNN_DATA` with the given PPI path. It calls `get_feature_origin(...)` to load sequences and vectors, and `generate_data()` to construct the `Data` object containing:
     - `graph.edge_index` : PPI edges (each protein pair).
     - `graph.edge_attr_1` : labels of PPI edges (interaction types).
     - `graph.x` : (unused here) node features of the PPI graph (not protein features).
     - Other attributes.
   - Loads the split JSON (`index_path`) to set `graph.train_mask` and `graph.val_mask` . These are lists of indices of edges used for training and validation.
   - It computes `ppi_list` – a list of all PPI edges in `(node1, node2)` form (halves of `graph.edge_index`).
   - It then categorizes validation edges into three "test sets" (`test1_mask` , `test2_mask` , `test3_mask`) based on whether one or both proteins were seen in training. (Nodes in `node_vision_dict` are marked 1 if seen in any train edge, else 0.) This stratification helps analyze how the model performs on unseen proteins, though the script ultimately tests on all validation edges.
3. **Model Loading**:
   - Instantiates `model = ppi_model()` and loads the saved weights: `model.load_state_dict(torch.load(model_path)['state_dict'])` .
   - Moves model to GPU if available.

4. **Feature/Adjacency Loading**:
    - Loads precomputed protein features: `p_x_all = torch.load(args.p_feat_matrix)` (PyTorch file).
    - Loads adjacency: `p_edge_all = np.load(args.p_adj_matrix, allow_pickle=True)`.
    - These contain stacked data for all proteins. The functions `multi2big_x` and `multi2big_edge` merge this data into large tensors suitable for batching:
        - `multi2big_x(x_ori)` concatenates all protein feature matrices into one large `(N_total_nodes, 7)` tensor (`x_cat`) and records the node counts of each protein in `x_num_index`.
        - `multi2big_edge(edge_ori, num_index)` concatenates all protein edges into one big edge list (`edge_cat`), adjusting node indices by cumulative offsets. It returns `(edge_cat, edge_num_index)` where `edge_cat` has shape `(2, N_total_edges)`.
        - `multi2big_batch(x_num_index)` then builds the `batch` vector: for each protein `i`, it assigns its nodes the batch number `i`. This yields a length `N_total_nodes` vector of graph IDs.
        - A final `batch+1` is done to match 1-indexing (the model does `batch-1` internally).
    - After these, we have `batch`, `p_x_all`, and `p_edge_all` ready as model inputs.
5. **Testing Loop**:
    - Calls `test(model, graph, graph.val_mask, device, batch, p_x_all, p_edge_all)`.
    - The `test` function splits the list of validation edge indices into batches (size 64). For each batch of edge indices `valid_edge_id`:
        - It runs `output = model(batch, p_x_all, p_edge_all, graph.edge_index, valid_edge_id)`.
        - Here, `graph.edge_index` is the full PPI edge list; `valid_edge_id` selects which edges to classify.
        - The model returns raw scores. It applies `sigmoid` to get probabilities, then thresholds at 0.5 for binary predictions (per class).
        - True labels are taken from `graph.edge_attr_1[valid_edge_id]`.
        - It accumulates predictions and labels.
    - After all batches, the code flattens predictions and truths and computes metrics using `Metrictor_PPI`. This reports **Precision, Recall, F1, and AUPR** for each of the seven PPI types.
6. **Output**: The script prints the metrics and the predicted/true label arrays.

In essence, `model_test.py` loads a trained model and data, prepares inputs, and evaluates prediction accuracy on the validation (test) set, reporting key classification metrics.

5. Workflow Summary

Putting it all together, the HIGH-PPI project pipeline is as follows:

1. **Data Preparation** (Protein info):
    - Collect protein structures (PDB) and sequences.
    - Run `generate_adj.py --distance D` to create residue contact graphs (adjacency) for each protein .

- Run `generate_feat.py` to compute residue features (e.g. physicochemical vectors) from structures/sequences.
- (For SHS27k dataset) Use `mapping_27k.py` to align protein IDs between sequence files and PPI networks.
- Save these as `p_adj_matrix` (NumPy adjacency lists) and `p_feat_matrix` (feature tensor).

2. **Split PPI Dataset**:
   - Choose a dataset (e.g. SHS27k edges file and sequence dictionary).
   - Use `GNN_DATA` to load PPI edges and sequence data. Optionally, produce train/val split indices via `split_dataset`. A JSON file in `train_val_split_data/` stores these indices.
3. **Model Training** (`model_train.py`, not explicitly asked but part of workflow):
   - Load data (using `GNN_DATA`), apply `split_dataset`, and create `Data` object with train/val masks.
   - Prepare model: instantiate `ppi_model`, optimizer, etc.
   - For each epoch: loop over training PPI edges, run forward (BGNN→TGNN) on mini-batches, compute loss, backpropagate.
   - Save the best-performing model checkpoint to `--save_path`.
4. **Model Testing** (`model_test.py`):
   - Load the saved model and the same data splits.
   - Prepare the “big” combined protein data (`p_x_all`, `p_edge_all`, `batch`).
   - Evaluate on validation edges (or a held-out test set): compute predictions with `model.forward()`.
   - Compute and output Precision, Recall, F1, AUPR for each PPI category.

Biologically, the high-level goal is: for each candidate protein pair, predict whether (and how) they interact, leveraging both protein structure (through residue contact graphs and features) and PPI network structure (through graph convolutions). The use of hierarchical GNNs allows the model to learn deep features of protein structures and then relate them to interactions. As noted in literature, GNNs are potent for PPI tasks because **PPI networks are graph-structured data**, and protein 3D structures naturally form residue-contact graphs . By modeling at both protein and network levels, HIGH-PPI aims to capture multi-scale information for robust interaction prediction.

References: We cited sources on PPI definitions , graph representation of protein structures , and the suitability of GNNs for PPI prediction , among others, to contextualize the code’s functionality. The code itself is from the provided `HIGH-PPI-main` project files.

Sources