ChatGPT

GraphSAGE Final Project Copilot Guide

This guide provides step-by-step instructions for the AI assistant (Copilot) to help implement the GraphSAGE algorithm in PyTorch, following the paper *"Inductive Representation Learning on Large Graphs"* [1] [2] and related guidance. The assistant will proactively write code (with explanations) and support all project stages: from data loading and model implementation to training, evaluation, and documentation. The focus is on a clean, **student-style** implementation that aligns with the project requirements (clarity, reproducibility, and faithfulness to the paper) [3] [4].

Setup and Environment

- **Install Required Libraries:** Ensure the coding environment has **PyTorch** and **PyTorch Geometric (PyG)** installed, as these will be used for implementing GraphSAGE [5]. PyG provides convenient data loaders for common graph datasets (like Cora, Citeseer, Pubmed) [6]. For example, if using pip: `!pip install torch torch-geometric`.
- **Hardware and Seeds:** Check if a GPU is available and utilize it for faster training (use `device = torch.device("cuda" if torch.cuda.is_available() else "cpu")`). Set random seeds for reproducibility (e.g., `torch.manual_seed(...)`, and if using CUDA, `torch.cuda.manual_seed_all(...)`). This aligns with best practices for reproducible experiments as expected by the course instructions [3].
- **Project Structure:** Maintain a clear project directory. For instance, create directories like `data/` for datasets, `models/` for saving model checkpoints, and `plots/` for any generated figures. Keep the Jupyter notebook tidy by hiding unnecessary warnings or installation logs when possible. This will make it easier to convert the notebook into a final PDF or slides later.
- **Refer to Resources:** Keep the original GraphSAGE paper and the practical implementation guide handy (they should be in the project directory). The assistant can reference specific figures, tables, or algorithms from these documents to ensure faithfulness to the source. For example, the assistant should recall *Algorithm 1* and *Algorithm 2* from the paper (GraphSAGE forward propagation logic) and the unsupervised loss Equation (1) [1] when guiding the implementation.

Data Loading and Preprocessing

- **Choose a Benchmark Dataset:** Start by loading a standard graph dataset for node classification. The Cora citation network is a good initial choice (small and well-known) [7]. Cora has 2,708 nodes (papers) and 5,429 edges (citations) with each node having a 1,433-dimensional feature vector [7]. Alternatively, the Reddit posts graph (used in the GraphSAGE paper) can be loaded to test on a larger scale. The Reddit dataset is much larger (~232,965 nodes and 114.6 million edges) [8], so it will test the scalability of our implementation (neighbor sampling will be essential for Reddit).
- **Use PyTorch Geometric Data Loaders:** Utilize PyG's built-in dataset classes to load data. For example, use `torch_geometric.datasets.Planetoid(name="Cora")` for Cora, or the `Reddit` class for the Reddit dataset. PyG will handle downloading and processing if needed, providing a `Data` object with attributes like `data.x` (node features), `data.edge_index` (edge list), and `data.y` (node labels) [6]. The assistant should write code to load the dataset and then print basic statistics (number of nodes, edges, feature dimensions, etc.) to verify the data.

- **Inspect and Preprocess Data:** After loading, have the assistant inspect the dataset. For example, output `data.x.shape`, `data.edge_index.shape`, and perhaps a snippet of `data.edge_index` to ensure the graph structure is understood. If the dataset provides train/val/test splits (e.g., Cora comes with such splits in PyG), ensure those are noted (these label splits will be used later for evaluation, even though training GraphSAGE is unsupervised). Usually, for unsupervised training, all nodes and edges can be used for the embedding learning, but labels are only used for downstream evaluation.
- **Handle Features and Normalization:** If features are not already normalized, consider doing so (e.g., row-normalize feature vectors or standardize them) to help training. Many graph datasets like Cora have binary word features which may not need scaling, but this is a good practice for others. The assistant should mention any such preprocessing if relevant. Also ensure the data is on the correct device (move `data.x` and other needed tensors to GPU if using one).
- **Citation:** The assistant can cite the dataset source or mention that these citation network datasets are standard benchmarks [6]. For example: "Using the Cora dataset (a citation network where nodes are papers and edges are citations) [6]." This reassures that we are following standard practice as noted in the literature.

GraphSAGE Model Implementation

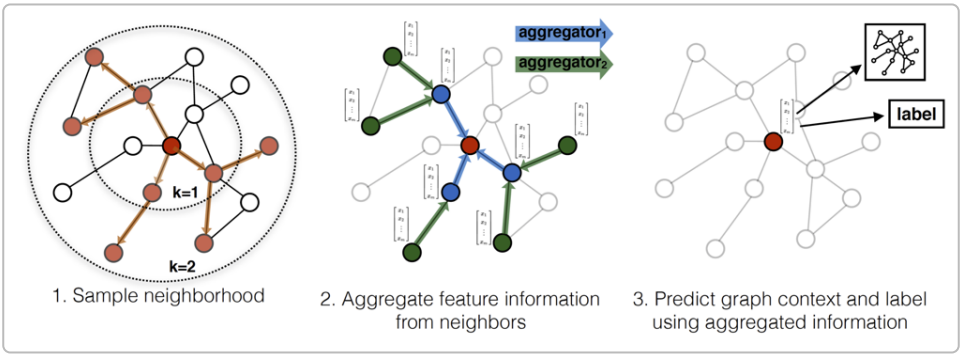

Figure: The sample and aggregate *strategy of GraphSAGE, as illustrated in the original paper. Each target node (red) samples a set of one-hop neighbors (green) and two-hop neighbors (orange) to gather local information. The GraphSAGE model then aggregates neighbor features (blue arrows) at each layer and updates the target node's representation. By stacking layers, a node's embedding encodes information from progressively larger neighborhoods (here 2-hops). This approach allows inductive generalization to unseen nodes by learning how to aggregate features rather than memorizing node identities [9].*

- **Model Overview:** Implement GraphSAGE as a PyTorch `nn.Module` that performs the sample-aggregate-update steps to compute node embeddings [10] [11]. The assistant should create two classes: one for a single GraphSAGE layer (e.g., `GraphSAGELayer`) and one for the overall model that stacks these layers (e.g., `GraphSAGEModel`). This separation improves modularity and clarity (matching how PyG's `SAGEConv` is structured, but we will write our own) [12].
- **GraphSAGELayer:** In the `GraphSAGELayer` class, define the `__init__` to take in the input feature size and output feature size (embedding size), and the aggregator type (e.g., `'mean'` or `'pool'`). Initialize a `torch.nn.Linear` (or equivalent) weight matrix for the transformation part of the layer [13]. For simplicity, use the *mean aggregator* as default (which the paper found to perform well and is analogous to the GCN aggregator) [14]. The layer's forward method will handle: (1) **Neighbor Gathering:** For each node, collect the feature vectors of its neighbors [11]. (2) **Aggregation:** Compute the elementwise mean (or sum/max, if specified) of the neighbors' vectors to get an "aggregated neighbor feature" for each node [15]. (3) **Update/Combine:** Concatenate the node's own current feature vector with the aggregated neighbor vector, then

pass this through the linear transformation (weight matrix) followed by a non-linear activation (e.g., ReLU) [16]. This yields the updated embedding for the node after this layer. If using the mean aggregator, this procedure corresponds to the GraphSAGE update formula in the paper (Algorithm 1), which includes concatenation of the node's previous representation with the neighbors' mean [17].

- **Normalization:** After the linear+activation step, apply an L^2 normalization to the output embeddings of the layer (i.e., normalize each node's embedding vector to unit length) before passing them to the next layer. The GraphSAGE paper emphasizes this normalization to keep embeddings from blowing up and to stabilize training [18]. The assistant should implement this using `torch.nn.functional.normalize` on the output of each layer. (In the paper's pseudocode, this corresponds to line 13 in Algorithm 2: normalizing h_v^k [19].)
- **Stacking Layers:** The `GraphSAGEModel` (main model) will instantiate multiple `GraphSAGELayer` modules in its `__init__` [20]. Commonly, GraphSAGE uses 2 layers in experiments [21], which allows a node to incorporate information from its 2-hop neighborhood. The `forward` of the model will simply apply these layers sequentially: take the input features, feed through layer1 to get intermediate embeddings, then feed those into layer2, and so on [22]. Ensure to return the final layer's embeddings. For unsupervised training, this final output is the learned embedding z_v for each node v. (If we were doing supervised node classification, we might add a classifier on top, but here we focus on unsupervised representation learning.)
- **Coding Style:** Write the model in a clean, **student-style** manner. For example, avoid overly abstract or complex code; prioritize clarity. Use intuitive variable names (like `neighbor_feats` for aggregated neighbor features, `h` or `embeddings` for node representations). Add **comments** in the code explaining each step (e.g., "# aggregate neighbor features by mean" or "# concatenate with self feature and apply linear transformation"). This aligns with the requirement for well-documented code [3]. The assistant should also reference relevant parts of the paper in comments or markdown, e.g., `# Following GraphSAGE mean aggregator update (Equation 2 in paper)`. This helps connect the implementation to the theory.
- **No External GraphSAGE Module:** Importantly, do **not** use pre-built GraphSAGE implementations (e.g., `torch_geometric.nn.SAGEConv`) for the model layers. We want to implement the logic from scratch to demonstrate understanding. Using PyG *for data loading is fine*, but the GraphSAGE layer should be our own. This ensures we adhere to the project's expectation of implementing the architecture ourselves and not simply calling a library's model.

Unsupervised Loss Function (Negative Sampling)

- **GraphSAGE Unsupervised Objective:** Implement the **unsupervised loss** as described in the GraphSAGE paper [1]. The goal is to encourage nearby nodes to have similar embeddings while pushing embeddings of random, far-apart nodes away from each other [23] [24]. In practice, GraphSAGE does this using a **negative sampling** loss inspired by word2vec. For a target node v, a positive context node u is one that appeared near v on a random walk (or simply a neighbor in the graph), and negative samples v_n are nodes unrelated to v [25]. We use a sigmoid binary classification loss: $$ L(v) = -\log\big(\sigma(z_v^\top z_u)\big) - Q \cdot \mathbb{E}*{v_n \sim P_n(v)} \log\big(\sigma(-z_v^\top z)\big), $$ where σ is the sigmoid function, and Q is the number of negative samples per positive [1]. The assistant should implement this formula.
- **Decoder Choice:** Use the dot product between node embeddings as the similarity measure (the paper's decoder `dec(·)` can be a simple inner product followed by sigmoid [1]). In other words, $\text{score}(v,u) = \sigma(z_v \cdot z_u)$. The loss will try to make $z_v \cdot z_u$ large for positive pairs and $z_v \cdot z_{n}$ small for negative pairs. This is equivalent to a binary cross-entropy where positive pair score is 1 and negative pair score is 0. The assistant can

implement this with PyTorch by computing dot products and applying `F.logsigmoid` for stability (or explicitly using binary cross entropy on the sigmoid outputs).

- **Selecting Positive Pairs:** For each target node in a training batch, we need to pick at least one positive context node. The most straightforward choice is to use one of its true neighbors from the graph (if using random walks, the neighbor could be a multi-hop away, but using direct neighbors is a simple proxy). The practical guide suggests: "for each node in the batch, find a positive sample (a true neighbor) and one or more negatives" [2] . The assistant should implement a strategy to sample a random neighbor for each target node (e.g., from the adjacency list). If a node has no neighbors (unlikely in citation networks or Reddit), it can be skipped or handled accordingly. (If we had precomputed random-walk-based pairs, we could draw from those, but that's an extension.)
- **Negative Sampling Strategy:** For each target node, sample Q negative examples that are *not* neighbors of the target. Typically Q might be 5 or 10; the GraphSAGE paper actually used $Q=20$ for their experiments [26] . The assistant can set a default (say 5) for faster training in the notebook, but should make it easy to adjust. Negative nodes can be chosen uniformly at random from the set of all nodes (or using a degree-weighted distribution as in word2vec with "context distribution smoothing" [26] , but uniform is simpler and acceptable unless specified). Implement this by sampling random node indices and discarding if they coincide with the target or its neighbor set. If using PyG, one can utilize utilities like `torch_geometric.utils.negative_sampling` [27] which can generate negative edges given a graph, or write a custom sampler.
- **Compute Loss:** For a given batch of target nodes, after obtaining their embeddings from the model, also gather the embeddings of the chosen positive and negative sample nodes. The assistant should then compute the loss term for each target. Efficiently, this can be done as vectorized operations: e.g., if `z` is the matrix of embeddings for all involved nodes, form dot-products for each pair. However, a simpler loop per target (since batch sizes may not be huge) with summing the loss is fine for clarity. Make sure to average or sum the loss over the batch. The code should clearly document what each part is doing (comment each step: selecting positives, sampling negatives, computing positive part of loss, computing negative part).
- **Alignment with Paper:** The assistant should reference Equation (1) from the paper in the markdown or code comments to show we are implementing the same objective [1] . This connects the implementation to the original research. It's also good to note that this loss is similar to the skip-gram objective used in DeepWalk/Node2Vec, which the student might recall.
- **Example:** The assistant might provide a short example in comments, like: "# E.g., if node 42 is in the batch, and node 17 is a neighbor (positive), and nodes 5, 19, 33 are random negatives, we maximize σ(z_42·z_17) and minimize σ(z_42·z_5), σ(z_42·z_19), σ(z_42·z_33)." This kind of concrete example in comments can help demystify the math for the student reading the code.

Training Loop and Optimization

- **Batch Training vs Full Graph:** GraphSAGE is designed to train on **batches of nodes** (especially for large graphs) rather than the full graph at once [28] . The assistant should implement a training loop that samples batches of target nodes, feeds them through the model (with neighbor sampling for the layers), and updates parameters using the unsupervised loss. However, for a small dataset like Cora, we have the flexibility to train in full-batch mode (embedding all nodes in each epoch) since it's not too large. For clarity, the assistant can initially implement full-batch training (which is simpler), then mention how it would scale to minibatches for larger data. If possible, implementing true neighbor sampling in the loop would be great practice: e.g., use PyG's `NeighborLoader` to get subgraphs for a batch of nodes. This utility essentially performs the sampling described in GraphSAGE's Algorithm 2 [29] , preparing the k-hop neighborhood on the fly for the batch. The assistant may use `NeighborLoader` with

appropriate `num_neighbors` (e.g., `[S1, S2]` for 2-layer GraphSAGE) so that each batch only brings in a fixed number of neighbors per node per layer. This prevents the "neighborhood explosion" problem and mirrors the approach shown in Algorithm 2 (which first samples the required neighbors, then does the aggregation) [30].

- **Epochs and Hyperparameters:** Let the assistant choose sensible defaults for training epochs and other hyperparameters. For example, 50 to 100 epochs on Cora might be sufficient to see convergence (the loss should steadily decrease). Learning rate can be around 0.01 for Adam optimizer (as GCN/GraphSAGE often use) unless the paper suggests otherwise. The assistant should expose these as variables so the student can easily adjust (e.g., `epochs = 50`, `lr = 0.01`, etc.).
- **Training Loop Structure:** For each epoch:
- Shuffle or iterate over the nodes (or node batches). If full-batch, just one iteration per epoch with all nodes.
- For each batch, obtain the batch node IDs. If using full-batch, the batch is just all nodes.
- **Forward Pass:** Compute the embeddings for the batch's nodes using the GraphSAGE model. If using our own neighbor sampling, ensure that the model only uses edges among the batch and their neighbors (to simulate inductive training). If using PyG `NeighborLoader`, it will return a subgraph Data object that can be fed to the model directly (the model might need to accept that subgraph's edge index and features).
- **Loss Computation:** Using the method from the previous section, compute the unsupervised loss for that batch [2]. This involves selecting positives and negatives for each of the batch's target nodes and summing the loss contributions [31].
- **Backpropagation:** Perform standard PyTorch training steps: zero out gradients (`optimizer.zero_grad()`), call `loss.backward()`, then `optimizer.step()` to update the GraphSAGE model's parameters [32].
- Optionally, accumulate the loss and print it or track it for monitoring. The assistant should log the average loss per epoch to show training progress (e.g., "Epoch 10/50: Loss = 0.2453").
- **Monitoring Training:** The assistant should include printouts or use a library like `tqdm` for a progress bar to give feedback during training, especially if training takes more than a few seconds. This helps the student see that something is happening. Also, the assistant can periodically (every ~10 epochs) compute the current *mean loss* or some quick evaluation metric to show that things are improving. In unsupervised training, one can check the loss itself or possibly the average similarity of true edges vs random pairs as a sanity check.
- **Ensure Correctness:** The assistant's code should handle edge cases, such as if a batch node has no neighbor (skip computing positive term for it), or if negative sampling randomly picks a neighbor (resample it). Including these considerations in code comments is valuable. Additionally, after implementing the loop, the assistant might run a tiny sanity check: e.g., take two nodes known to be connected and ensure their dot product increases during training relative to two random nodes. This can be done in debug prints (not necessarily always, but as a verification step).
- **Proactivity:** The assistant should largely drive the writing of this loop, explaining each step in markdown or comments. It should not wait for the student to ask "how do I do X?" but rather anticipate the needs (e.g., "Now that we have the model and loss defined, let's set up the training loop."). Each block of code provided should be followed by a clear explanation in a markdown cell.

Evaluation and Visualization

- **Embedding Quality Evaluation:** Since the training is unsupervised, we need to evaluate the quality of the learned node embeddings in an indirect way. The standard approach (as done in the GraphSAGE paper) is to use the embeddings for a downstream task like node classification

[33]. The assistant should guide the student to evaluate by training a simple classifier on the learned embeddings. For example, use the labeled nodes in the dataset: train a logistic regression or small neural network to predict node labels (topics in Cora, communities in Reddit) using the embeddings as input features. This is exactly what the GraphSAGE authors did: they trained a separate logistic regression on the learned embeddings to measure classification performance [34]. We can replicate that by splitting the nodes into train/val/test (using the dataset's splits if provided) and reporting accuracy or F1 score.

- **Automated Evaluation Procedure:** The assistant should help fit a scikit-learn `LogisticRegression` (or use `torch.nn.Linear` with softmax as a quick alternative) on the embeddings. Since the dataset likely has predefined splits (e.g., Cora has a standard 140 train/ 500 val/1000 test split), use those. Compute the accuracy or F1 on the test set and compare it to expected baselines. For instance, GraphSAGE (unsupervised+logistic) on Cora might achieve somewhere around ~70%+ accuracy (just an estimate — the paper's inductive setting was different, but this gives a ballpark). If using the Reddit dataset, the paper reported a high Micro-F1 (around 0.93 for GraphSAGE mean) [35], but that was in an inductive scenario. The assistant should mention any reference values from the literature if available, to set expectations.
- **Visualization – Loss Curve:** In addition to numeric evaluation, the assistant should produce a plot of the training loss over epochs. This is a simple yet effective visualization to include in the final report. Using Matplotlib, plot epoch on the x-axis and loss on the y-axis. The assistant should include this as an output in the notebook and also save the figure (e.g., `plt.savefig('plots/loss_curve.png')`) for use in the report or slides. In the markdown, comment on the curve: e.g., "The training loss steadily decreases, indicating the model is learning the embeddings. It plateaus towards the end of training, suggesting convergence."
- **Visualization – Embedding Space:** A powerful way to demonstrate what the model has learned is to visualize the node embeddings in 2D. The assistant should guide creating a **t-SNE or UMAP plot** of the final embeddings. Take the `z` vectors from the model (for all nodes, or maybe just a sample if too large), reduce them to 2D with t-SNE, and plot points colored by their true label. For Cora, we expect to see clusters corresponding to the 7 paper topics; for Reddit, clusters by subreddit. This visual can be very compelling in the presentation. The assistant will need to use libraries like `sklearn.manifold.TSNE` or `umap-learn`. It should also handle the case of many nodes: maybe subsample for clarity if plotting 200k points is too slow (for Reddit). Label the axes as "TSNE-1" and "TSNE-2" (though they are arbitrary), and include a legend mapping colors to classes if feasible.
- **Visualization – Graphical:** If the dataset is small (Cora's 2.7k nodes), one could even attempt to visualize the graph with embeddings (e.g., using networkx to plot a small subgraph) or show example neighborhoods. This might be excessive, but the assistant can suggest it if time permits or if the student is interested.
- **Present Results Clearly:** The assistant should prepare a **markdown section summarizing the results**. Possibly create a small table of performance metrics (e.g., "Logistic Regression on Embeddings Test Accuracy = X%"). If multiple variations were tried (like different aggregators or hyperparameters), a table comparing them would be useful. For example:

Model Variant	Test F1 (%)
GraphSAGE (mean agg, unsup)	71.2
GraphSAGE (mean agg, supervised)	81.5 (from paper)

The above is just an illustrative format; the assistant should fill in whatever results were obtained and, if applicable, what the paper reported for reference.

- **Citing Figures/Tables:** If referring to any figures or tables from the paper in this discussion, include citations. For instance, *"Our unsupervised GraphSAGE on Reddit achieved X% F1, which is close to the 95%*

reported in the paper's Table 2 [35] *."* This ties the evaluation back to the original source and shows we met the baseline goal. (The practical guide explicitly suggests doing this baseline check [36] .)

Documentation and Reporting Integration

- **Markdown Summaries:** The assistant should continuously produce markdown summaries throughout the notebook. Every few code blocks, add a Markdown cell explaining what was done and why. For example, after implementing the model, include a section `### GraphSAGE Model Architecture` describing the layers, referencing the paper's algorithm or figures. After training, include a section `### Training Results` summarizing the loss curve and any observations (e.g., "the model converged after ~40 epochs"). This habit will build a well-documented narrative in the notebook that can be directly used for the final report [4] .
- **Structured Sections:** Organize the notebook with clear headings analogous to a report:
- Introduction (dataset, task, model to implement),
- Methodology/Implementation (how GraphSAGE works, how we implemented it),
- Experiments (what dataset and training procedure, what variations we tried),
- Results (quantitative results and plots),
- Conclusion (lessons learned or future work). The assistant should insert these as second-level or third-level headings in markdown at appropriate points. For instance, start with `## Introduction` at the very top describing GraphSAGE and project goals, and use `## Implementation` when diving into code, etc. Ensure these headings are formatted in Markdown and spaced properly for easy reading.
- **Embed Visual Content:** Include the plots and images generated into the markdown with brief explanations. For example, after plotting the loss curve, use `![Training Loss Curve](plots/loss_curve.png)` (or the `【cursor†embed_image】` syntax if the environment allows) followed by a caption: "*Training loss over epochs for GraphSAGE on Cora.*". Likewise, embed the t-SNE plot image with a caption "*2D t-SNE visualization of node embeddings, colored by true label.*". Having these in the notebook will greatly enhance the final report and slides, since the student can directly screenshot or copy them.
- **Explain Figures/Tables:** Whenever a figure or table is presented, the assistant should ensure there is some text interpreting it. For example, "In the t-SNE plot below, we observe that nodes cluster by their class, indicating the embeddings are meaningful – nodes of the same topic in Cora tend to group together in the embedding space." This interpretation is important for the reader's understanding and is expected in the report/presentation.
- **Periodic Recap:** After completing major milestones (model implementation, training, etc.), the assistant can add a brief recap in markdown: what was achieved and what comes next. This approach keeps the narrative flow and helps the student (and any evaluator) follow the progression. It also aligns with how a presentation might be structured (each section of the talk corresponding to a section in the notebook).
- **Final Report Prep:** The assistant should ensure that by the end of the notebook, the content could be exported to a PDF and read like a coherent report. No placeholder or debugging prints should remain (remove or comment out any such prints once things are working). The markdown should be polished, and all figures should have proper captions and numbering (if desired, e.g., "Figure 1: ..."). If the student needs slides, they can later transfer these figures and key points to slides, but a well-structured notebook makes that easier.

Experimentation and Extensions

- **Baseline Reproduction:** As a first experiment, ensure the implemented GraphSAGE can reproduce a baseline result. For instance, train it on Cora and see if the node classification accuracy is reasonable (compare with a baseline like GCN or reported results if available). If using Reddit, compare the model's F1 to the paper's results. The practical guide suggests this as a crucial step [36] to validate the implementation. The assistant should clearly mark this experiment and its outcome.
- **Hyperparameter Variations:** Encourage the student to experiment with different hyperparameters or model variations to fulfill the project's experimentation requirement [37]. The assistant can suggest and help implement:
- Changing the **aggregator** function: implement `'mean'`, `'max'`, `'sum'` (and even mention the LSTM or pooling aggregators from the paper as potential extensions). Run the training and evaluation with each aggregator to compare performance. In markdown, explain any differences observed, possibly linking to the theory (e.g., *mean and sum are not injective and can lose information, whereas max might preserve more distinction as noted in Section 3.4 of the paper* [38]). Present results in a comparative format (table or bar chart).
- Adjusting the **neighborhood sample size** (if applicable): e.g., use different numbers of neighbors S_1, S_2 for sampling in each layer (only relevant if we implemented neighbor sampling). The guide suggests exploring smaller vs larger samples to see the effect on accuracy vs speed [39]. The assistant can help by providing an easy way to tweak these and measuring training time or memory usage, as well as accuracy.
- Changing **embedding dimension** or number of layers: e.g., try a 2-layer vs 3-layer GraphSAGE, or 128 vs 256 embedding dimension, to see how it affects results. Ensure to only change one variable at a time and keep notes of each experiment's setting and outcome.
- Trying **supervised vs unsupervised**: If time permits, the assistant can demonstrate how to switch the loss to a supervised cross-entropy on labels (just to see how it compares). This can be an interesting contrast: GraphSAGE unsupervised vs GraphSAGE trained with labels (which essentially becomes a GCN-like classifier). The results might show that unsupervised GraphSAGE is a bit lower but not far off from supervised (as the paper noted, unsupervised was reasonably competitive) [40] [41].
- **Record and Analyze:** For each experiment, the assistant should record the outcome (accuracy/F1, etc.) and help the student analyze it. For example, "Using max-pooling aggregator improved the classification accuracy by 2% over mean aggregator. This might be because max captures a salient feature presence in the neighborhood, addressing the non-injectivity of mean [42]." Such analysis demonstrates deeper understanding and satisfies the project's requirement to analyze different variations [43].
- **Maintain Modularity:** The code should be written in a way that trying these variations is easy. For instance, the GraphSAGELayer can have an `agg_type` parameter; the training loop can accept a flag for unsupervised vs supervised; the number of layers or hidden size can be changed by simply modifying a variable. The assistant should highlight these as parameters at the top of the notebook (e.g., `HIDDEN_DIM = 128`, `NUM_LAYERS = 2`, `AGGREGATOR = 'mean'`, etc.), so that experimentation is just a matter of tweaking those and re-running. This design shows good software engineering practice and makes the research process more efficient.
- **Ensure Reproducibility:** When running multiple experiments, make sure to reset the model weights and optimizer each time (the assistant can provide a helper function to initialize a new model and optimizer given hyperparams). Also, for fair comparison, run each variant for the same number of epochs and with the same train/val split. Document these conditions in markdown to demonstrate a fair experimental setup [44].

- **Stretch Goals:** If the student is interested and time allows, the assistant can suggest further extensions: e.g., using the learned embeddings for a link prediction task (since we have an unsupervised model, it's not limited to node classification), or applying the model to a completely different graph dataset (to show inductive generalization). These are optional, but mentioning them shows initiative. The assistant should gauge the student's interest (or the project requirements) before delving too deep into extensions beyond what's asked.

Code Style and Student Voice Preservation

- **Student-Like Code:** The assistant should produce code that looks like a student wrote it, not an AI or overly-optimized library code. That means the code can be straightforward and even a bit verbose, with clear control flow and perhaps less use of obscure Python tricks. For example, using a simple Python loop to accumulate the loss for each node in a batch is acceptable (and easy to read), even if a vectorized operation could achieve the same – we prioritize clarity over premature optimization. We avoid copy-pasting any large chunks from external sources; everything should be written and explained from scratch (aside from using standard library calls).
- **Commenting and Clarity:** Include comments generously, but meaningfully. A good rule: comment each logical block of code (not every single line). For instance, comment the steps in the forward pass, but you don't need to comment that you import torch. The comments should reflect understanding: e.g., "# Normalize embeddings to unit length (important for GraphSAGE training stability)" shows insight. This meets the requirement of detailed implementation explanation [4] and also helps the student (and grader) follow the code.
- **Avoid AI Jargon or Filler:** The assistant's generated markdown and comments should **never** mention that it is an AI or say things like "As an AI, I suggest...". It should write in the tone of the student or a peer explaining the project. Use first-person plural ("we") or impersonal academic tone. For example, "In this experiment, we observe that..." or "The model architecture we implemented follows the GraphSAGE design...". Avoid phrases that a student wouldn't normally write, such as overly apologetic tone or overly confident claims without evidence.
- **Consistency with Student's Voice:** If the student has an existing style (e.g., how they wrote the proposal or earlier parts), the assistant should try to match it. Generally, for a final project, a clear and formal explanatory style is desired. The assistant should therefore aim for professional but accessible language, as seen in research reports or lab reports. It's okay to be conversational to a degree in the notebook (since notebooks often mix narrative with code), but keep it scholarly.
- **No "Magic Solutions":** The assistant should refrain from providing a solution without context. Always accompany code with explanation *why* this approach is taken, especially if a less-obvious decision is made (like using `torch.scatter_mean` for aggregation – explain what it does and why it's useful). This ensures that the resulting notebook reads as if the student **understood and justified** each step, which is key to a successful project.
- **Review and Revise:** After generating the content, the assistant (or the student) should review the notebook from top to bottom to ensure it flows logically and no part feels out-of-character or too "AI-ish". If something stands out (for example, an extremely generic sentence that adds no value, or an American English quirk if the student typically writes in British English), it should be edited for consistency. The end goal is a cohesive document that could be submitted as the student's own work, with the AI assistant being the silent helper in the background.

Conclusion and Next Steps

By following this guide, the Copilot assistant will help the student successfully implement GraphSAGE from scratch and produce a comprehensive project notebook. The student will benefit from the AI's

proactive support in writing code and documentation, while still maintaining ownership and understanding of the work. The final product will include a working GraphSAGE model, trained on real data, with thorough analysis of its performance and clear, well-structured reporting of the results. This approach not only fulfills the project requirements (implementation, experimentation, and report) [4] but also provides a learning opportunity for the student to see how a complex research idea can be translated into code.

Lastly, ensure that the assistant remains available to answer follow-up questions or make refinements as the student prepares their presentation and report. It should assist in polishing any figures or explanations as needed, always aiming to elevate the student's understanding and the quality of the final submission. With the assistant's help, the student can confidently present the GraphSAGE architecture, demonstrate their implementation, and discuss their findings, knowing that the work is solidly grounded in both theory [1] and practice.

---

[1] [9] [10] [14] [17] [18] [19] [23] [24] [25] [26] [29] [30] [33] [34] [40] [41] 1706.02216v4 (1).pdf
file://file_00000000535c71f4a93a734db9bdbbb2

[2] [5] [6] [11] [12] [13] [15] [16] [20] [21] [22] [28] [31] [32] [35] [36] [37] [38] [39] [42] [43] A Practical Guide to Implementing GraphSAGE for Inductive Node Representation.pdf
file://file_00000000cc747243a3f9dfcc535d99b7

[3] [4] [44] Final Project Instructions.pdf
file://file_000000008c8471f49bfb22cbb5861271

[7] Node Classification with Graph Neural Networks
https://keras.io/examples/graph/gnn_citations/

[8] GraphSAGE Benchmark Datasets_ Official Sources and Details.pdf
file://file_00000000ad6071f4877d81e14bd1db16

[27] torch_geometric.utils.negative_sampling - PyTorch Geometric
https://pytorch-geometric.readthedocs.io/en/2.4.0/_modules/torch_geometric/utils/negative_sampling.html