Implementation Plan for Sentiment Classification Challenge

1. Framework and Environment Setup

Apple Silicon MPS with PyTorch: Set up a Python environment (using Conda or pipenv) with PyTorch >= 1.12 to ensure Apple Metal (MPS) backend support . Verify that PyTorch detects the MPS device (Apple M3 Max GPU) before training. For example:

```
import torch
if torch.backends.mps.is_available():
    device = torch.device("mps")  # Use Apple GPU
else:
    device = torch.device("cpu")
```

This moves tensors and models to the MPS device for acceleration . All model computations will then run on the Mac's GPU. Ensure macOS is up to date (>= 12.3) and that the Python environment is arm64 (Apple Silicon) so PyTorch was installed with MPS support . If using Conda, create an environment with Python 3.9+ and install PyTorch via the pytorch channel (which by default now includes MPS support in stable releases).

Dependencies and Requirements: Create a requirements.txt (or environment.yml for Conda) listing all necessary packages. Key dependencies include:

- **PyTorch** (with torchvision, torchaudio if needed) – core deep learning framework (with MPS support).
- **Transformers** (Hugging Face) – for loading pre-trained models and tokenizers.
- **Datasets** (Hugging Face) – to load the Financial Phrasebank dataset easily .
- **SentenceTransformers** – for sentence embedding models (which can also be installed via pip as sentence-transformers).
- **scikit-learn** – for metrics (accuracy, F1) and any tools like k-NN implementation or dimensionality reduction utilities (e.g., for t-SNE/UMAP if not using specialized libs).
- **UMAP** or **matplotlib**/**seaborn** – for embedding visualization (UMAP for dimension reduction, matplotlib for plotting).
- **OpenAI** (openai Python package) – if using the GPT-4 API for the bonus classification task.
- Any other utilities (pandas, numpy, etc.) as needed for data handling.

This requirements.txt will be part of the deliverables for reproducibility .

Leveraging 10 Worker Threads: Utilize the Apple M3's multi-core CPU by setting PyTorch DataLoader workers to 10. In the training code, when creating the DataLoader for training and validation sets, use DataLoader(dataset, batch_size=..., shuffle=..., num_workers=10) to spawn separate worker processes for data loading. This will significantly speed up data input pipeline, ensuring that the GPU (MPS) is fed with data continuously. The number 10 is chosen to match the available cores/threads on the M3 Max (adjust if needed based on performance monitoring). We will also pin memory (pin_memory=True) in DataLoaders if supported on Mac, to speed the transfer of batches to the GPU.

Efficient Training Configuration: We will use mixed precision if available for MPS (as of PyTorch 2.x, GPU half-precision on MPS might be limited, but we will check PyTorch docs for amp support on MPS). If supported, enabling automatic mixed precision can improve speed and memory usage. We also ensure to call torch.set_num_threads(10) if needed to align PyTorch's intra-op parallelism with the 10 CPU cores for any CPU-bound operations. Logging and checkpointing will be lightweight to avoid bottlenecks – for instance, use PyTorch Lightning or the Hugging Face Trainer API for a streamlined training loop, but with careful configuration to use the MPS device.

Finally, test the environment by running a small model's forward pass on MPS and measuring that the GPU is indeed utilized (the verification code above printing a tensor on 'mps:0' confirms that). With the environment ready, we proceed to data preparation and modeling.

2. Baseline Classification Model

Dataset Preparation and Exploratory Analysis: Start by loading the Financial Phrasebank dataset from Hugging Face . This dataset contains ~4,840 financial news sentences labeled as Positive, Negative, or Neutral . Each sentence has 5–8 human annotations; multiple versions of the dataset are available based on annotator agreement thresholds . We will use the 50% agreement configuration (which includes all 4,840 sentences with a majority vote label) as our full data pool, ensuring we have as much data as possible for semi-supervised experiments. We will also keep track of the agreement level for each instance (the dataset card indicates subsets at 50%, 66%, 75%, and 100% agreement).

Perform basic EDA: compute class distribution (how many positives, negatives, neutrals) and basic text length stats. Report these as part of the plan (e.g., a small table of class counts, and perhaps mean sentence length). This helps confirm if classes are imbalanced and informs choice of metrics (if one sentiment is much more frequent, accuracy could be less informative than F1).

Hierarchically Nested Data Splits: To simulate various amounts of labeled data, we will create a series of **nested training subsets** of increasing size . For example, we can define training set sizes like 100, 250, 500, and 1000 (approximately logarithmic increments up to the ~1000 limit as suggested). Construct these so that each larger training set contains all data from the smaller set (e.g., the 250-set includes the 100, the 500-set includes the 250, etc.) . This nesting ensures a fair "learning curve" comparison (performance should improve as more labeled data are cumulatively added, without randomness of different samples). The largest training set will be ~1000 instances, which is roughly 20% of the full dataset . The remaining ~3,840 instances (of the 50% agreement data) will initially be treated as **unlabeled pool** for semi-supervised labeling in later steps.

We will also reserve a **validation and test set**. Since no official split is given , we will create our own: for example, take ~20% of the data (~960 sentences) as a held-out test set for final evaluation, ensuring it includes a mix of all sentiment classes and agreement levels. We might further split a portion of the training data for validation (or use cross-validation) to tune hyperparameters . For simplicity, we can allocate another 10% (~480 sentences) as a validation set for model selection and hyperparameter tuning, leaving ~3,400 sentences for the pool of training+unlabeled. From that pool we extract the nested training sets (100, 250, 500, 1000) and regard the remainder in each case as unlabeled. **Important:** When constructing splits, ensure that the class distribution is roughly consistent across train/val/test, and that we shuffle the data before splitting (stratified sampling by sentiment label for fairness).

High vs. Low Annotator Agreement Analysis: To analyze the effect of label quality, we will leverage the agreement information. One approach: ensure that the smaller training set (e.g., the 100-sized) consists of high-agreement examples (e.g., all from the 100% agreement subset) to simulate a scenario of fewer but very reliable labels. Meanwhile, perhaps another training scenario could use the same number of examples but from lower-agreement instances to compare effect. The prompt specifically asks to evaluate subsets with different inter-annotator agreement levels, so we will incorporate this by evaluating model performance separately on high-agreement test samples vs low-agreement test samples. For instance, after training a model, we can compute accuracy/F1 on the subset of test sentences that had 100% annotator consensus vs. those with only 5/8 (~62%) agreement, to see if the model struggles more on ambiguous texts. We will **"analyze a scenario with low vs high inter-annotator agreement"** by such comparisons . Additionally, in the weak labeling stage, we will observe if using high-agreement seed labels yields better weak labels than using lower-agreement seeds (more on this in Section 3).

Model Selection: Choose two pre-trained Transformer-based text classifiers from Hugging Face as baselines . We aim for diversity in model size and possibly domain:

- *Model A:* **DistilBERT-base-uncased** – a compact, faster variant of BERT. DistilBERT has about 66M parameters and is lighter to fine-tune, which is beneficial given our hardware (M3 GPU is powerful but still not as high-memory as a large server GPU). It’s a general-purpose language model and will serve as a strong baseline for sentiment classification.

- *Model B:* **RoBERTa-base** (or **BERT-base-uncased**) – a full-size transformer (~110M parameters) with proven performance on many NLP tasks. RoBERTa-base is a robust choice for classification and often outperforms BERT due to its training regimen. This provides a comparison between a smaller vs larger model.

*(Alternatively, we could consider **BERTweet-base** (a RoBERTa-based model trained on tweet data) as mentioned, but since our data is formal financial news, a general model like RoBERTa or a finance-specific model might be more appropriate. If domain-specific models are allowed, **FinBERT** (a BERT model pre-trained on financial text) could be considered as well. However, to follow the guidelines of using general-purpose models , we will proceed with DistilBERT and RoBERTa.)*

Model Architecture: Both selected models will be used in a standard text classification setup. We will use the Hugging Face Transformers library’s AutoModelForSequenceClassification class to load each model with a classification head (a feed-forward layer on top of the [CLS] token or pooled output). The classification head will output 3 logits (for the three sentiment classes). We will initialize the models with pre-trained weights (distilbert-base-uncased, roberta-base, etc.) and a new classification layer. Since we have a moderate-size dataset, full fine-tuning (updating all transformer weights) is feasible, but we will monitor for overfitting on smaller training sizes.

Training Procedure: Fine-tune each model on the **full hard-labeled training data first (all ~1000 training examples)** to get a baseline performance . We will use that to decide which model is “best” (likely by validation F1 score) . Expected steps in training:

- Tokenize the input sentences using the model’s tokenizer (with padding/truncation to a reasonable length, e.g., 128 tokens).
- Use the DataLoader with batch_size (we might start around 16 or 32, adjusting based on MPS memory usage), and num_workers=10 as configured.
- Use an optimizer like AdamW (with weight decay for regularization) and a learning rate in the range 2e-5 to 5e-5 (typical for transformer fine-tuning). We will likely schedule the learning rate with warmup and decay (using Hugging Face’s get_linear_schedule_with_warmup) to stabilize training.
- Train for a limited number of epochs, e.g., 3-5 epochs for each model on the full training set, as Transformers often converge within a few epochs on small datasets. Monitor validation loss and stop early if overfitting.
- Compute **evaluation metrics** on the validation set: accuracy and F1-score (macro-averaged F1 across the three classes). F1 is important due to potential class imbalance and because it balances precision/recall; accuracy is straightforward to interpret for overall performance. We’ll track both.
- After initial training on the full ~1000, pick the model with higher validation performance (or perhaps consider efficiency vs performance tradeoff). For example, if RoBERTa-base gives slightly better accuracy than DistilBERT but is much slower, we

might still prefer it if performance gain is significant. Likely, RoBERTa-base will be the top performer. We will designate that as our **chosen baseline model** for further experiments.

Next, we **repeat fine-tuning for each hierarchically nested subset** using the selected best model (and possibly also observe the second model's performance for comparison). For each training subset size (100, 250, 500, 1000), we fine-tune the model from the pre-trained weights (fresh each time, to simulate training with exactly that many labels from scratch) and evaluate on the validation set. This yields a **learning curve of model performance vs. training data size** . We will plot this as **Figure 1** (as suggested in the challenge) to illustrate how additional labeled data improves the classifier . The expectation is a monotonically increasing curve that might plateau as it nears 1000.

To ensure results are robust, for each data size we might run 2-3 training runs with different random seeds (since with small data the variance can be high) and average the metrics. However, since the splits are nested (not random re-splits), variance is less of an issue here – each larger set strictly contains the smaller, so we can directly attribute differences to more data rather than data differences.

Evaluation on Annotator Agreement Subsets: As mentioned, we will evaluate how the model does on **high-agreement vs low-agreement subsets**. Concretely, we will take our final baseline model (trained on 1000) and test it on two slices of the test set: (a) sentences that had 100% annotators agreement, and (b) sentences with only the minimum agreement (in the 50% threshold set, that could be cases where only 4 of 8 annotators agreed, or generally those not included in the 75% or 100% sets). We will report metrics for each subset. This will tell us, for example, if the model achieves higher accuracy on clearly consensual cases (likely easier examples) and lower on contentious cases, which is valuable insight for error analysis . If we find large differences, it emphasizes the importance of label quality.

Baseline Results: We will compile the results of this section in a clear format. Likely a table of performance at each training size (rows: 100, 250, 500, 1000 training examples; columns: validation accuracy, validation macro-F1, etc.), and mention the chosen model. Additionally, Figure 1 (learning curve plot) will visualize the trend . We will also include a short discussion of any surprising findings (e.g., if diminishing returns after 500 examples, or if the smaller model outperformed larger on very tiny data due to less overfitting). This baseline establishes a reference for the improvements we expect with semi-supervised learning next.

(Possible extension: apply model quantization after training to reduce memory, as hinted in the challenge . We can mention trying post-training dynamic quantization on the DistilBERT model to see if we can compress it with minimal accuracy loss, since that was noted as a

consideration. This is optional and mainly for demonstrating awareness of deployment optimization.)

3. Embeddings and Weak Labelling

With the baseline in place, we move to semi-supervised techniques. The goal is to use the large pool of unlabeled sentences (those not in the labeled training set) to create **weak (pseudo) labels** based on similarity to the labeled examples . This leverages the idea that "similar texts have similar sentiments."

Selecting Sentence Embedding Models: We will generate fixed-size vector embeddings for each sentence using state-of-the-art **sentence transformers**, which are Transformer models fine-tuned specifically to produce semantically meaningful sentence encodings . At least two different pre-trained embedding models will be applied:

- **all-MiniLM-L6-v2:** a MiniLM-based SentenceTransformer model (6 layers, ~22M parameters, 384-dimensional embeddings). It's fast and has good performance on semantic similarity tasks, making it a strong candidate for embedding thousands of sentences efficiently.
- **paraphrase-mpnet-base-v2:** a RoBERTa (MPNet) based model that outputs 768-dimensional embeddings. This model is known to achieve very high performance on semantic textual similarity benchmarks. It is heavier than MiniLM but may produce higher-quality embeddings.

We load these via the SentenceTransformers library: e.g., SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2'). Using two different models allows us to compare embedding quality. We will embed **all sentences in the dataset** (both labeled and unlabeled) with each model and store the embeddings.

Embedding Quality Evaluation (Quantitative): Before using the embeddings for labeling, we will **explore them quantitatively** to ensure they capture useful semantic structure. Possible analyses without dimensionality reduction include:

- **Intra-class vs Inter-class Similarity:** Compute cosine similarities between sentence embeddings. We can take a sample of labeled sentences and calculate the average cosine similarity among sentences of the same sentiment versus the average similarity among sentences of different sentiments. If the embeddings are good for sentiment, we expect same-sentiment pairs to have higher similarity on average than cross-sentiment pairs. We will report these statistics (e.g., "Using paraphrase-mpnet embeddings, the average cosine similarity for sentences with the same label was 0.72 vs 0.45 for different labels, indicating decent clustering by sentiment.").

- **Nearest Neighbor Label Consistency:** For each labeled sentence, find its nearest neighbor (NN) in the embedding space (by cosine distance) and check if the NN has the same sentiment label. We can compute the proportion of cases where the NN label matches. A high percentage would mean embeddings cluster sentences by sentiment well. This effectively measures one-step KNN classifier accuracy on the labeled set itself. We will do this for both embedding models and possibly present it as a small table (Model vs. NN accuracy).
- **Case studies:** We can pick a couple of example sentences and list their top 3 most similar other sentences (by cosine similarity), showing their labels. Qualitatively, this can illustrate if similar content indeed corresponds to the same sentiment. For instance, if a positive sentence about profits finds as nearest neighbors other profit/growth-related sentences that are labeled positive, that's a good sign. If we find mismatched labels among nearest neighbors, that could either highlight ambiguous cases or limitations of embeddings.

Embedding Quality Evaluation (Qualitative): Additionally, we will **visualize the embeddings** in two ways: first, via simple similarity heatmaps or clustering without reduction. For example, we might select 50 random sentences and plot a heatmap of their cosine similarity matrix, then reorder by sentiment label to see if blocks (clusters) emerge by label. This is a way to "see" clustering without projection. We will also perform dimensionality reduction (next section in Bonus with UMAP) as an illustrative tool.

Weak Labeling Strategies: Using the computed embeddings, we develop multiple strategies to assign **weak labels** to the unlabeled data . At minimum, we will implement:

- **k-NN Based Labeling:** For each unlabeled sentence embedding, find the *k* nearest labeled sentence embeddings (from our current labeled training set) using cosine similarity. We might use k=5 (tunable). The labels of those neighbors are used to infer the label for the unlabeled point. We will experiment with **majority vote** (assign the class that appears most among the k neighbors) as well as **similarity-weighted vote** (weigh each neighbor's vote by its cosine similarity score so that closer neighbors count more). For example, if among 5 neighbors 3 are Neutral, 2 Positive, we label as Neutral; if ties, we could choose the class of the closest neighbor or decide no label. We will also consider different k values (like 3, 5, 10) and see which yields more accurate weak labels (we can validate on a subset of data for which we have true labels).
- **Similarity Thresholding:** As an alternative strategy, for each unlabeled sentence, find the most similar labeled sentence. If the cosine similarity is above a high threshold (e.g., >0.9), assign the unlabeled the same label; if it's below a lower threshold (<0.8), maybe label it as "uncertain" (or simply we could choose not to assign any label, though the challenge expects using all unlabeled data , so perhaps we always assign something). We can calibrate this threshold by looking at the distribution of cosine similarities for true same-label pairs vs different-label pairs (from earlier analysis). The idea is that if an unlabeled example is very close to a labeled example in embedding space, it's likely they share sentiment. This method might result in ignoring some unlabeled data if none are confidently similar to a label (which means we'd have fewer but high-precision pseudo-labels).

- **Clustering or Centroid Method (optional):** Another approach is to compute class centroids in embedding space from labeled data. Then label each unlabeled example with the class whose centroid it's closest to (this is like a 1-NN from centroids). This is simpler but essentially assumes roughly spherical, separated clusters for sentiments. It might not be as effective if classes overlap, but it's easy to implement and could be tried for comparison.
- **Self-training with Model (optional advanced):** Although not explicitly required, we might also try a classic self-training: train an initial classifier on the small labeled set, use it to predict labels for unlabeled, filter high-confidence predictions as pseudo-labels, and retrain. However, since the challenge specifically emphasizes embedding-based strategies, we'll focus on those first. (We may mention this in discussion as another approach to weak labeling, but likely stick to embedding KNN for implementation.)

For each strategy, we will **generate weak labels for the unlabeled dataset**. We should do this separately for each scenario of labeled set size. For example, if only 100 sentences are hard-labeled, we use those 100 as the neighbors to label the other ~3,740 unlabeled. If 250 are labeled, use those, etc. This way, we can later see how the number (and quality) of seed labels affects the weak labeling quality.

Weak Label Quality Evaluation: It's crucial to **evaluate the quality of these weak labels before using them for training** . We can do this by utilizing the ground truth that we actually have (though in a real scenario we wouldn't). Concretely:

- Take the pool of "unlabeled" data for which we generated weak labels, and compare the weak label to the true label (remember, we held out a test set separately, but here we can use the fact that in our dataset, what we call "unlabeled" still has an actual label from annotation – we just didn't use it for training). Compute accuracy of the weak labels and F1, to see how well the weak labeling strategy performs on its own. This essentially treats the weak label method as a classifier and measures its performance. We will do this for each strategy and each training size scenario. This addresses the prompt to *"compare results of sentiment classification with the weak labels directly (i.e., metrics for the weak labels)"* . Likely, with only 100 seed labels, the weak labels on the other ~3.7k sentences might be noisy (low accuracy), but as seed size grows to 1000, the weak labels could improve in quality.
- We will select the **most promising 2–5 techniques** and parameter choices based on these evaluations . For instance, we might find that k-NN with k=5 and weighted voting yields 70% weak label accuracy, whereas a simple 1-NN or threshold method yields 65%. We'll choose the top methods (say, k-NN and the thresholded approach if they are among best) to carry forward. If one method is clearly inferior, we'll drop it. The challenge expects 2-5, so we might keep two: e.g., weighted k-NN and thresholded similarity.
- We will also analyze if using **high-agreement labeled seeds** results in better weak labels than low-agreement seeds . To do this, we could conduct a controlled experiment: take 100 high-agreement labeled examples (100% consensus) vs 100 randomly chosen (which include lower consensus). Use both as seed sets to weak-label a fixed set of "unlabeled" sentences and compare weak label accuracy. This will

show how annotator noise in seed data affects the pseudo-labeling process. We anticipate that high-quality seeds improve weak labeling precision.

Throughout this section, results will be documented with both **tables and discussion**. For example, one table might list "Weak Label Strategy vs. Accuracy/F1 of pseudo-labels (for a given seed size)". We'll also include some **examples** of errors from weak labeling (to qualitatively assess where it goes wrong, e.g., an unlabeled sentence was given label Positive by KNN but actually was Negative – perhaps because it was lexically similar to a positive case but had a negation subtlety). These insights will guide how we use the weak labels in training.

(Note: All computations in this section can be done on CPU if needed, but since we have many embeddings, we can also utilize the MPS GPU for similarity computations by keeping embeddings in torch tensors on the GPU. The cosine similarities for thousands of points can be computed with matrix operations for speed. We must just be mindful of memory (embedding matrix of 4k x 768 is fine).)

4. Enhanced Model Training with Weak Labels

After identifying the best weak labeling approaches and generating pseudo-labeled data, we will **augment our training sets with these weak labels** and retrain the classifier to see if it improves performance . This mimics a semi-supervised learning scenario where we have a small set of gold labels and a large set of "free" but noisy labels.

Training Data Augmentation: For each hierarchically nested training size (100, 250, 500, 1000 hard labels), take all the remaining unlabeled examples (out of the ~3400 pool used) and add them to the training set with their weak labels. According to the instructions, we should *"always utilize all available unlabeled data for weak labelling"* to maximize the benefit. For example:

- When 100 are hard-labeled, we will have ~3,300 weak-labeled examples (if validation took ~500 and test ~960, then ~3,300 remain unlabeled in training pool). We add all 3,300 with their pseudo-labels to form a combined training set of ~3,400.
- When 250 are hard-labeled, add ~3,150 weak-labeled, total ~3,400, and so on. For the 1000 hard-labeled case, adding the remaining ~2,400 weak-labeled yields ~3,400 combined. (The exact counts depend on how we split earlier, but the principle holds.)

We will use the **best model from baseline (e.g., RoBERTa-base)** for all these experiments, to isolate the effect of data augmentation. The model will be initialized from pre-trained weights each time to avoid carrying over any bias.

Training Setup: The training procedure is similar to baseline, but now the dataset is much larger (thousands instead of hundreds). We will train for a few epochs, possibly more if needed, but we must watch out for the noise from weak labels. Overfitting to noisy labels could actually hurt performance on real data. Techniques to mitigate this:

- Use a smaller learning rate or fewer epochs when training on a lot of weak data, since the signal is noisy.
- Perhaps give lower weight to the loss from weak-labeled samples compared to hard-labeled ones. (We could implement a weighted loss where hard labels have weight 1.0 and weak labels weight 0.5, for instance, reflecting our confidence. This is an optional refinement if needed.)
- Shuffle the combined training data thoroughly so the model doesn't learn any ordering (mixing hard and weak examples).

We will train and then evaluate on the **held-out test set** (the same test set used for baseline) to see how performance changed. We do this for each size scenario:

- **100H+Weak vs 100H alone:** Did adding 3,300 pseudo-labeled examples boost accuracy/F1 compared to using 100 real ones only?
- **250H+Weak vs 250H**, etc.

We expect that with very few hard labels, the weak labels (though noisy) will significantly help the model generalize better, thus improving test performance – essentially *steepening* the learning curve. As the number of hard labels grows, the marginal benefit of weak labels might decrease (if the model is already quite good with 1000 real labels, adding pseudo-labels might give a smaller boost or even potentially slight harm if labels are wrong). We will quantify this.

Results and Comparison: We will prepare a **comparison table of model performances**:

Rows can be different training sizes (100, 250, 500, 1000). Columns could be:

- Baseline model accuracy/F1 (only hard labels)
- Semi-supervised model accuracy/F1 (hard + weak labels)
- Difference (improvement) in points.

We will also include the performance of the weak labeling method itself on the test (from section 3, how accurate the pseudo-labels were) as a reference. This addresses the task to *"compare how model performance changes with additional weak labels under different strategies, and compare to using weak labels directly"* . For instance, if the weak labeling method alone had 70% accuracy on test (just by nearest neighbors), and the trained model with those labels gets 75%, we see the model learned to generalize a bit better than raw pseudo-labels. On the other hand, if the model's performance is lower than the weak labels' inherent accuracy, it means the model might be getting confused by noise.

We will plot an **enhanced learning curve** as well, perhaps overlaying it on the baseline curve. This could be a graph of performance vs training size for baseline (hard-only) vs semi-supervised (hard+weak), illustrating the gap. Ideally, the curve with weak labels is always above the baseline curve, demonstrating the benefit of weak supervision .

During analysis, consider *annotator agreement effect*: If time permits, we might run the semi-supervised training in two modes – one where the initial labeled seeds were high-agreement examples vs one where they were mixed – to see if that influences the final model. The expectation is that starting with high-quality labels yields better final performance (because the model and pseudo-labels were grounded in less noisy truth). We will note such findings if observed.

5. Bonus Tasks

5.1 Few-Shot Sentiment Classification with GPT-4 (LLM In-Context Learning)

To provide a point of comparison, we will use a large language model (LLM) to perform sentiment classification via prompting, **without fine-tuning**. Specifically, we can use the GPT-4 API (if available) or GPT-3.5-turbo, etc., with **in-context learning**. The idea is to feed a prompt that explains the task and gives a few examples, then ask the model to label new sentences.

Prompt Design: We will create a prompt along these lines:

```
You are a financial sentiment analysis assistant.
Classify the sentiment of a given financial news sentence as Positive,
Negative, or Neutral.
Use the perspective of an investor reading the news.
Provide the label only.
```

```
Examples:
Sentence: "The company's profits surged by 50% this quarter."
Sentiment: Positive

Sentence: "The CEO resigned amidst a major scandal."
Sentiment: Negative

Sentence: "The company announced a new product line."
Sentiment: Neutral

Now classify the following sentence.
Sentence: "<new sentence here>"
Sentiment:
```

We will give GPT-4 maybe 3-5 shot examples (as above) covering each class, then append the test sentence and let it output a sentiment. We should ensure the examples reflect the Financial Phrasebank style (we can even use a few actual sentences from the dataset with their labels as context examples).

Using the API: Using OpenAI's Python API (with proper keys and rate limits in mind), we will send this prompt for a sample of test sentences (possibly the entire test set or at least a representative subset, say 200 sentences to limit cost). We parse the model's responses into labels and compare with ground truth.

Evaluation: Measure GPT-4's accuracy and F1 on these sentences. Prior research and intuition suggest GPT-4 should perform quite strongly, possibly near or above our fine-tuned smaller models, given it has seen a lot of language and can follow instructions. However, it might misclassify subtle cases or get some neutral vs positive distinctions wrong if the prompt/examples aren't perfect. We will tabulate the LLM's performance (and possibly compare to the best fine-tuned model's performance on the same subset). This addresses the bonus exploration of *"classification performance of an LLM with in-context learning"* .

We will also note **qualitatively** the LLM's behavior: Does it struggle with certain types of sentences (e.g., maybe it labels everything as positive due to optimistic bias, or it has trouble with domain-specific language)? If possible, we can also test GPT-4 with zero-shot (just an instruction and no examples) to see if the few-shot examples indeed improved its accuracy. This can be briefly discussed.

One more aspect: we will report the *cost and speed* considerations (e.g., "Using GPT-4 for 500 sentences cost X USD and took Y minutes"), since that feeds into the feasibility discussion. While GPT-4 might be accurate, it's not free nor instantaneous for large datasets, which is a point to mention in conclusions.

5.2 Embedding Visualization with Dimensionality Reduction

As a final analysis, we will produce visualizations of the sentence embeddings using a technique like **UMAP or t-SNE** . This is to glean insight into how sentences cluster in semantic space and whether sentiment separation is visible.

We will take the embeddings from one of our models (e.g., the **paraphrase-mpnet-base-v2** embeddings, since those are higher-dimensional and likely richer in information) or possibly the embeddings from the *fine-tuned classifier's penultimate layer* (to see how the model separates classes). For simplicity, using the pre-trained embeddings is fine.

UMAP Setup: Use UMAP from the umap-learn library to project embeddings down to 2D. We will experiment with hyperparameters: n_neighbors (maybe around 15) and min_dist (0.1 or so) to see a good separation. We'll color points by their true sentiment label (positive/negative/neutral) to see if distinct clusters form.

We might create two plots:

- UMAP of **original sentence-transformer embeddings** colored by label. If sentiment is a strong signal, we might see, for example, a cluster of positive sentences separate from negatives. If not, they may overlap significantly, indicating that raw semantic embeddings aren't purely sentiment-focused (which is plausible, as they capture general semantic similarity, not specifically sentiment).
- UMAP of **fine-tuned model embeddings** (like taking the [CLS] token output from the last hidden layer of our fine-tuned DistilBERT/RoBERTa for all test sentences). Since that model was trained to distinguish sentiment, its embedding space might show clearer separation between classes. This can be insightful to illustrate how task-specific fine-tuning reorganizes the semantic space.

We will include these plots in the report, each with a short caption. For example, *"Figure 2: UMAP visualization of sentence embeddings (paraphrase-mpnet-base-v2). Points are colored by sentiment label. We observe that positive (green) and negative (red) sentences form partially distinct clusters, while neutral (blue) points are spread out, often intermingling, suggesting neutrality is context-dependent."* We will discuss any patterns observed: perhaps sentences about profit and growth cluster (positive), those about losses cluster (negative), and neutral ones might be more scattered or form their own group if they are about mundane announcements.

If any particular cluster stands out (e.g., a cluster of neutral that are actually “not relevant from financial perspective” as per dataset definition), we’ll mention that. This qualitative analysis complements our quantitative results by giving an intuition of data distribution.

6. Evaluation and Conclusion

In this section, we synthesize all results, comparing the approaches in terms of performance, practicality, and the key question of **label efficiency**.

Comparative Analysis of Models: We will create a summary table comparing:

- **Baseline Fine-tuned Model (supervised)** – using 1000 hard labels.
- **Semi-Supervised Model** – using 1000 hard + ~2400 weak labels (the best strategy).
- **GPT-4 Few-shot** – on the full test set.

Metrics to compare will include accuracy and macro-F1 on the test set. This will let us see, for example, if our fine-tuned RoBERTa with 1000 labels achieved 85% accuracy, did using an additional 3000 weak labels push it to 90%? And how does GPT-4 perform – maybe GPT-4 gets 88% with prompting. If GPT-4 is similar or better, that’s notable given it required zero training. We will discuss these results in terms of trade-offs (e.g., GPT-4 doesn’t require manual labeling but costs API usage; fine-tuned models can be deployed to run locally on new data efficiently, etc.).

We’ll also explicitly address the *feasibility of weak labeling as a standalone approach*. That is: **Is the weak labeling strategy by itself sufficient to label data accurately, or is it crucial to use those labels to train a model?** For instance, if in Section 3 we found that weak labels had 70% accuracy and the model trained on them achieved 80%, then training a model clearly improved over using the noisy labels directly, justifying the extra step. Conversely, if weak labeling alone was say 85% accurate and the model stayed around 85-87%, it might imply one could almost just rely on the pseudo-labels as “good enough” without a complex model. We will make a conclusion on this with evidence. The question from the challenge is phrased as: *“Is training with weak labels worthwhile or is the weak label generation process sufficient to generate the labels directly?”* . Our results will allow us to answer that.

Time-Savings Estimate: One important deliverable is a **“time savings factor”** from using weak labeling . We interpret this as: how many fewer manual (hard) labels are needed to reach a certain performance when using weak labeling? We will choose a target performance level (likely the accuracy/F1 that the model achieved with the full 1000 labels in baseline). Then see how many labels are needed with and without weak data to reach that. For example, if baseline needed 1000 labeled to hit X% F1, but with weak labels we achieved X% F1 with only 500 labeled (plus the rest weak), then we saved 500 labels – a 50% reduction. We can express this as "Using our best semi-supervised method, the model required only 50% of the manual labels to achieve the same performance as the fully supervised model. This indicates a time-savings factor of 2 (half the labeling effort) for the desired accuracy."

We will base this estimate on our learning curves. Perhaps we'll find an even stronger effect at some mid-range. We will include a brief discussion that in real annotation hours, this could translate to significant cost savings, **with the caveat** that quality of weak labels and the domain specifics matter.

Key Findings: Summarize key takeaways:

- The improvement curve: e.g., "We observed that incorporating weak labels consistently improved the model's F1 by 5-10 points at low data regimes, and even at 1000 labels, gave a small boost of ~2 points, demonstrating the value of unlabeled data."
- The effect of annotator agreement: e.g., "High-consensus labeled examples led to more accurate weak labels and ultimately slightly better model performance than using low-consensus labels, highlighting the importance of label quality in semi-supervised learning. The model also performed substantially better on high-agreement test instances than low-agreement ones, indicating the latter remain challenging (ambiguous even for humans)."
- GPT-4 vs Fine-tuned: e.g., "GPT-4 in-context achieved comparable performance to our best fine-tuned model (~90% accuracy), showing the power of large LMs. However, using GPT-4 for all predictions is costly and may not be feasible for real-time classification of thousands of sentences, whereas our fine-tuned model runs locally on Apple Silicon in milliseconds per instance. This suggests a hybrid approach: GPT-4 could be used to bootstrap labels or for cases where the model is less confident."

Recommendation: Based on the above, we will answer whether weak labeling is a suitable strategy going forward. If our results show a strong benefit, we might recommend using weak labeling to **minimize manual labeling efforts**. For instance, "Our best semi-supervised model with 300 hard labels + weak labels matched the performance of a fully-supervised model with 1000 labels, saving 700 annotations (~70% of labeling work). Therefore, weak labeling is highly worthwhile in this dataset." We'll note any conditions (e.g., "This holds given that we had a large pool of unlabeled data and the assumptions of the embedding similarity method held true."). If weak labeling was less effective, we might conclude that a certain level of manual labels is still needed or that more advanced techniques (or more domain-specific embeddings) might be required to make weak supervision viable alone.

Finally, we wrap up with any reflections or possible next steps (like could we improve weak labels with iterative retraining, or how might this approach generalize to other tasks).

7. Deliverables and Project Structure

Jupyter Notebooks: The project will be organized into clear Jupyter notebooks for each part of the analysis . Likely notebooks include:

1. **Data Exploration and Preparation:** Loading the dataset, EDA, creating splits (hierarchical subsets, train/val/test), ensuring reproducibility (fix random seeds when sampling).
2. **Baseline Models Training:** Fine-tuning DistilBERT and RoBERTa on the data, recording results for each training size. This will include code for training (using Hugging Face Trainer or custom loops) and producing the learning curve plot.
3. **Embeddings and Weak Labeling:** Generating sentence embeddings, analyzing similarities (with code for computing and plotting similarity stats), implementing k-NN and other labeling, and evaluating their accuracy. This notebook will likely output the chosen weak label sets for each scenario.
4. **Semi-Supervised Model Training:** Using the augmented datasets (hard+weak labels) to train the classifier, then evaluating on test and comparing with baseline. This notebook generates the enhanced learning curve and comparative metrics.
5. **Bonus Analyses:** This might be split or combined. Possibly one notebook for GPT-4 experimentation (since it requires API calls – we might present results rather than executing live if API keys are sensitive). Another for Embedding Visualization (UMAP plots), or these could be sections in earlier notebooks.
6. **Conclusion and Discussion:** Not necessarily a separate notebook, but the reporting of results will be integrated in the above notebooks. We will ensure that all notebooks are run sequentially so that they produce a coherent story from data to conclusions .

Each notebook will be documented with Markdown explanations, following best practices (clear section headers, comments in code, and interpretation of outputs) . Before submission, we will run all notebooks from top to bottom to ensure reproducibility and that results match the described narrative . We will export these notebooks to HTML or PDF for an easy read-only view .

Code Repository: In addition to notebooks, we will provide a structured repository with modular code . Common functionality (to avoid cluttering notebooks) will be factored into python scripts or modules, for example:

- data_prep.py for dataset loading and custom splitting logic.
- models.py for any model wrapper or helper (though we largely use Hugging Face models directly, we might include a function to instantiate a model, or a training loop if not using Trainer).

- weak_labeling.py containing functions to compute embeddings, perform k-NN search (perhaps using Faiss or sklearn's NearestNeighbors for efficiency), apply labeling strategies, and evaluate them.
- train.py scripts for training baseline and semi-supervised models (if we want to allow running training outside notebooks, e.g., on a cluster or different machine, as hinted by the challenge). These could accept arguments like model name, data size, etc., enabling systematic experiments.
- Utility scripts for plotting (e.g., plot_utils.py for generating charts like the learning curve or UMAP scatter).

We will include a **README** explaining how to run the code, and the repository will contain the requirements.txt for setting up dependencies . The repository name will clearly indicate the challenge and author.

Diagrams and Visual Aids: To satisfy the traceability and clarity requirements , we will include a few diagrams/flowcharts in our documentation:

- A pipeline diagram illustrating the overall process (from data splitting -> baseline training -> embedding & weak labeling -> semi-supervised training -> evaluation). This can be a simple flow chart graphic embedded at the start of the report for an overview.
- Possibly a schematic of how k-NN labeling works (could be a small illustration, or just well-explained in text with an example).
- Plots and tables are also forms of deliverables: we already plan several (learning curves, bar charts comparing methods, UMAP plots). Each will be clearly titled and referred to in the text discussion .

Documentation of AI Tool Usage: As required, we will include a **section discussing the use of AI tools like ChatGPT/GPT-4 in our project** . This will be a 250-500 word section at the end of the analysis document (perhaps in the conclusion notebook or as a separate markdown file) detailing:

- Which parts of the work we used AI assistance for. For example, we might say we used ChatGPT to brainstorm approaches for weak labeling or to debug a PyTorch error, or used GitHub Copilot to accelerate writing some code functions.
- The prompting strategies we used and which were most effective . For instance, we can describe how we prompted GPT-4 for a few-shot classification by providing instructions and examples (as we did in Bonus 5.1), and note the outcome. Or if we used it to improve our code or writing, mention how prompts were phrased.
- An assessment of how these AI tools contributed to solving the task and to our learning (perhaps noting that using GPT-4 to label some data gave a quick benchmark, or that ChatGPT helped explain a concept we were unsure about, etc.). This reflective portion ensures we comply with the requirement to acknowledge and discuss AI tool usage .

After assembling all these deliverables, we will double-check that the **analysis is comprehensive and traceable**, meaning someone can follow from problem statement through methodology to results and conclusions, with all steps justified and reproducible . The final submission will include the notebook files (.ipynb), their HTML/PDF exports, the code repository (zipped or linked), and any additional write-up. The project title and author name will be clearly indicated on all documents for completeness .

By following this implementation plan, a single researcher on an M3 Max Mac can effectively carry out the sentiment classification mini-challenge, leveraging the hardware (with MPS acceleration and multi-threading) and modern NLP frameworks (Hugging Face Transformers and Sentence-Transformers) to explore both supervised and semi-supervised learning techniques on the Financial Phrasebank dataset . This plan ensures a step-by-step progression through data preparation, baseline modeling, innovative weak labeling, enhanced training, and thorough evaluation, culminating in actionable insights on the value of weak supervision for sentiment analysis. The results and discussion, supplemented by clear notebooks, code, tables, and figures, will address all requested aspects and provide a solid foundation for the final report.

Sources:

- Financial Phrasebank data description
- Mini-challenge instructions (hierarchical data splits, weak labeling, evaluation)
- PyTorch MPS usage guidelines (Apple Metal Performance Shaders backend)
- Course deliverables and tools usage requirements