Chongshen Yang(Student ID: 300414211) Kelvin Mock (Student ID: 300453668)

1.Self-training

Self-training is a semi-supervised learning approach where a single classifier is trained on a small set of labeled data and then used to assign pseudo-labels to unlabeled data. We experimented by comparing the accuracies of using 10% and 20% of the original dataset on the model respectively. These pseudo-labeled instances are then added to the training set, and the classifier is re-trained iteratively.

1. At first, the classifier will be trained on the labeled data.
2. Then it predicts labels for unlabeled instances.
3. After that, it selects high-confidence predictions (with probability >= 0.8). Those predicted pseudo-labeled instances will be added onto the train set.
4. Finally, those data are used to retrain the base classifier with this new expanded transit.

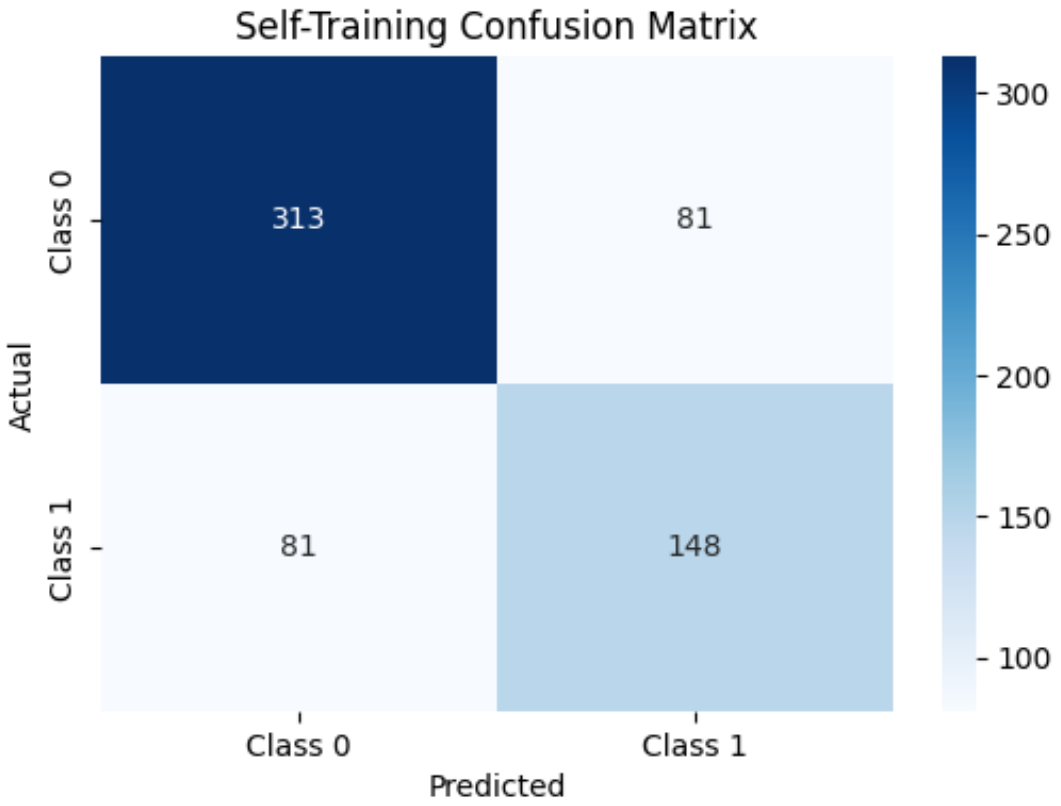
The above process will be repeated for `max_iter` number of times.

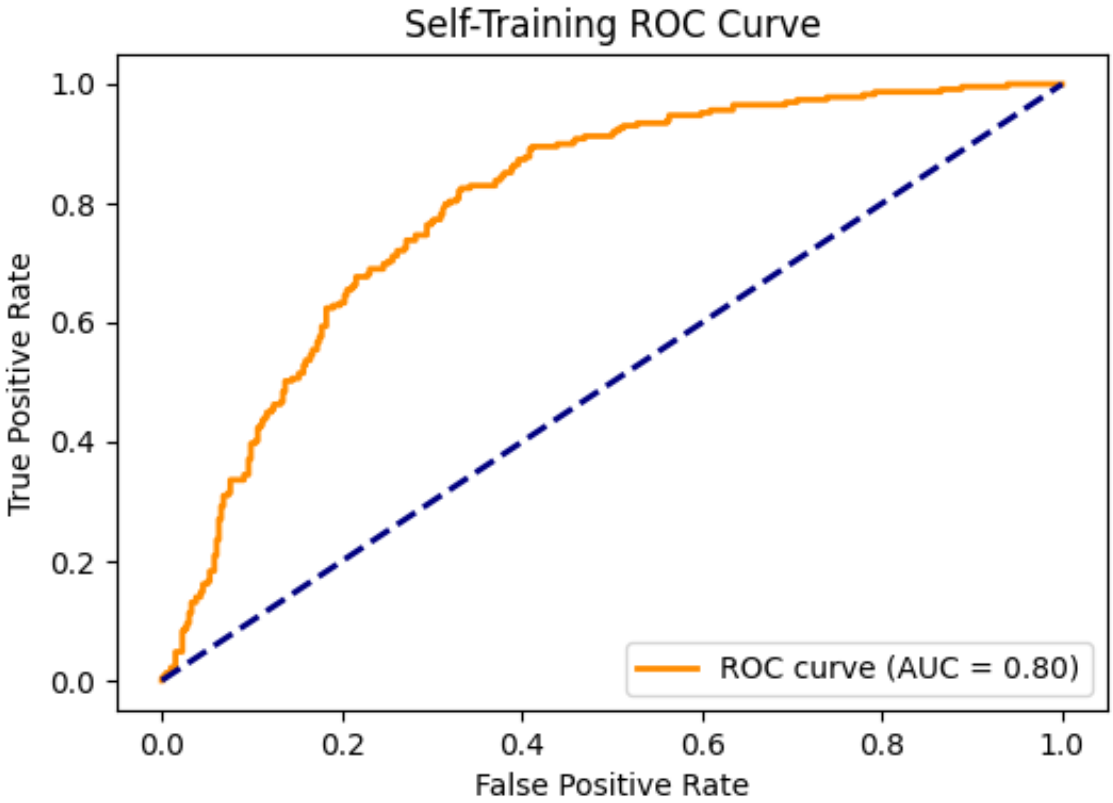
```
# Initialize base classifier
base_clf = GradientBoostingClassifier(random_state=42)

# Initialize Self-Training Classifier
self_training_clf = SelfTrainingClassifier(base_estimator=base_clf, threshold=0.8,
max_iter=10, verbose=True)

# Fit the model
self_training_clf.fit(X_train, y_train)

# Predictions
y_pred = self_training_clf.predict(X_test)
y_pred_proba = self_training_clf.predict_proba(X_test)[:, 1]
```


Self-Training Confusion Matrix
313
81
81
148
Class 0
Class 1
Actual
Class 0
Class 1
Predicted
300
250
200
150
100


Self-Training ROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (AUC = 0.80)

2. Co-training

Co-training is a semi-supervised learning method which involves complicated steps:

1. Split Features into 2 views: we shuffled the features and then split, for in the original order, all descriptive features are in the front, if we follow this, we will have a very bad result.
2. Initialize 2 classifiers, with 1 for each view.
3. Train both classifiers with the labeled instances.
4. Iteratively Pseudo-Label Unlabeled data:

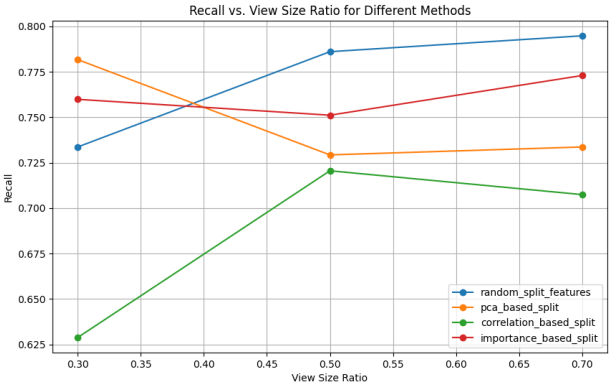
   a. Classifier1 labels unlabeled data for View2 (of the features set), and adds those newly labeled data to unlabeled data of View2, and

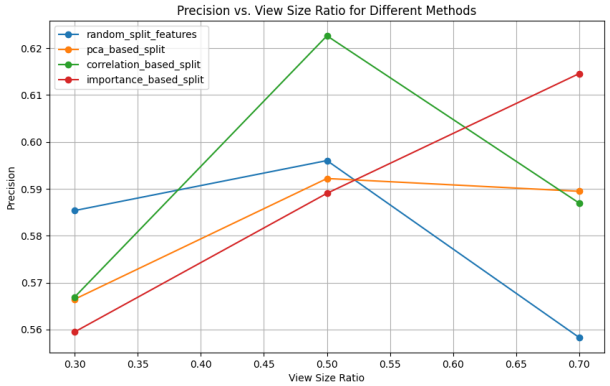
   b. Classifier2 labels unlabeled data for View1 (of the features set), and adds those newly labeled data to unlabeled data of View2

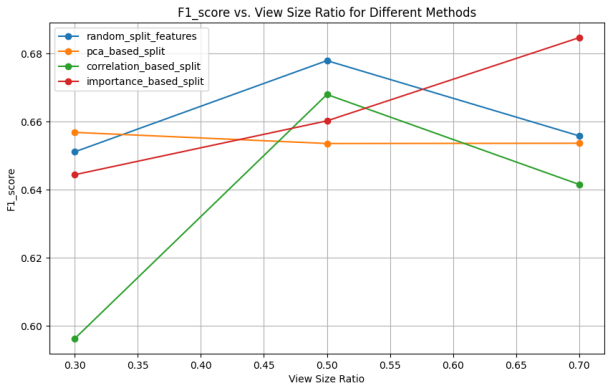
Co-Training works best when the two sets of features are different enough yet still relevant to the same prediction task — where they need to be complementary but not overlapping. However, in the original setup, all **descriptive features** were placed at the front, which led to poor results. This arrangement made it difficult for each classifier to learn effectively, as the features used by each weren't independent enough or failed to offer different perspectives on the data.

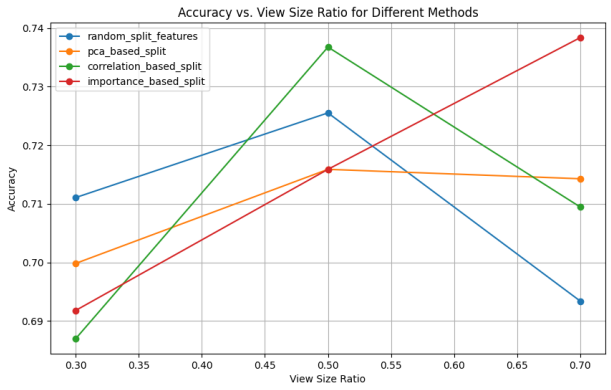
Recognizing this issue, I shuffled the features, which significantly improved the outcome compared to the unshuffled version. Next, I experimented with various feature-splitting methods, including **random splitting**, **PCA-based**, **importance-based**, and **correlation-based** approaches, along with different feature view sizes, to optimize the co-training effect.

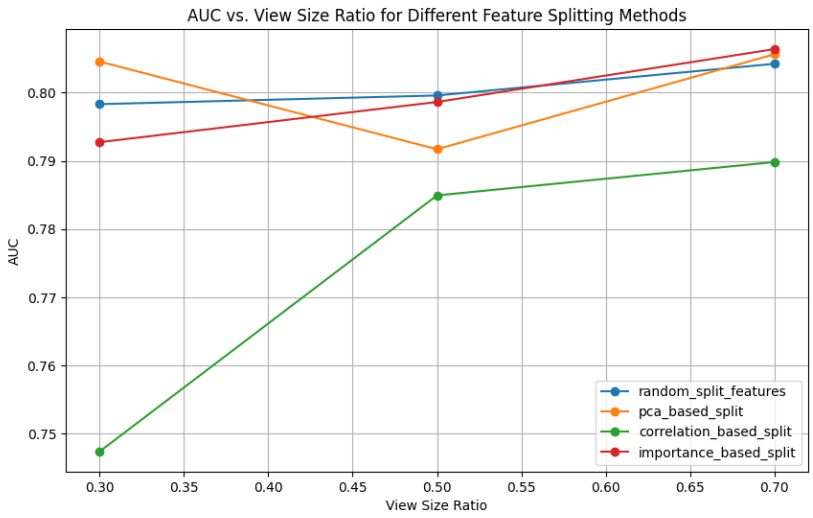
To determine how to create the feature view for each classifier, we conducted experiments to evaluate the impact of view size and feature combinations on co-training performance.





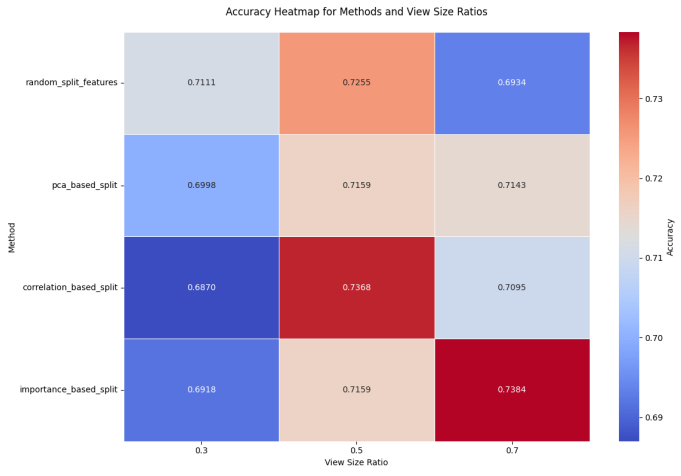


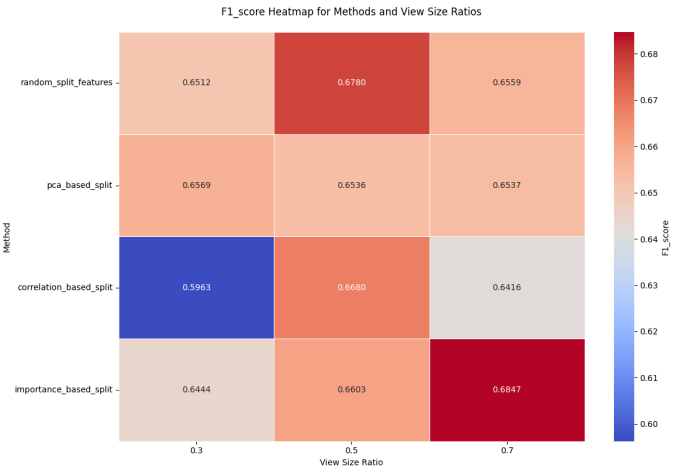


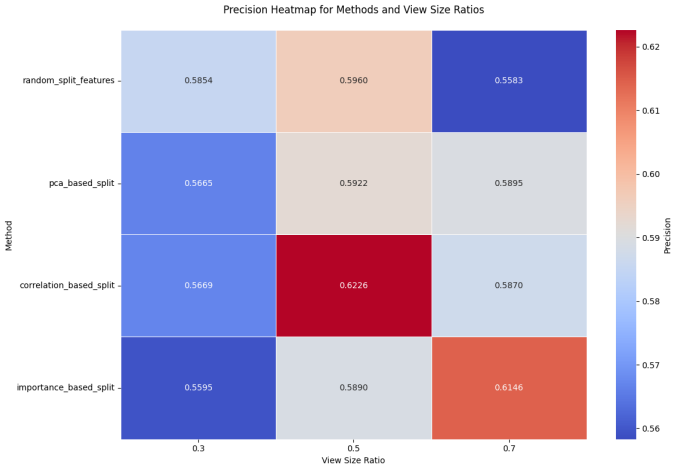


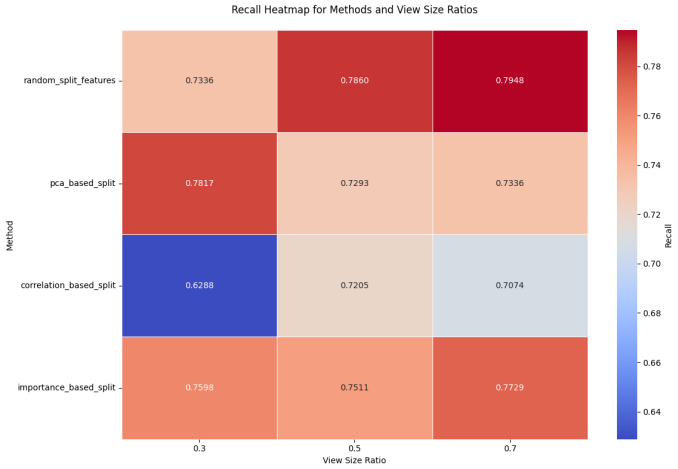
The following heatmaps illustrate the results across different metrics. We observed that the **importance-based split** with a **view size ratio of 0.7** consistently delivered the best average performance when considering the results comprehensively.

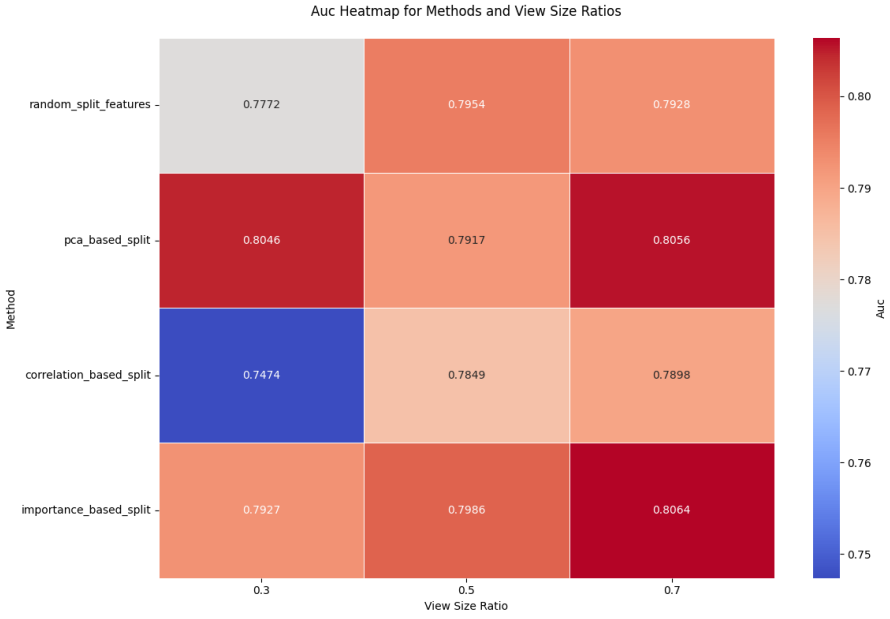
Based on these findings, we decided to adopt this configuration.

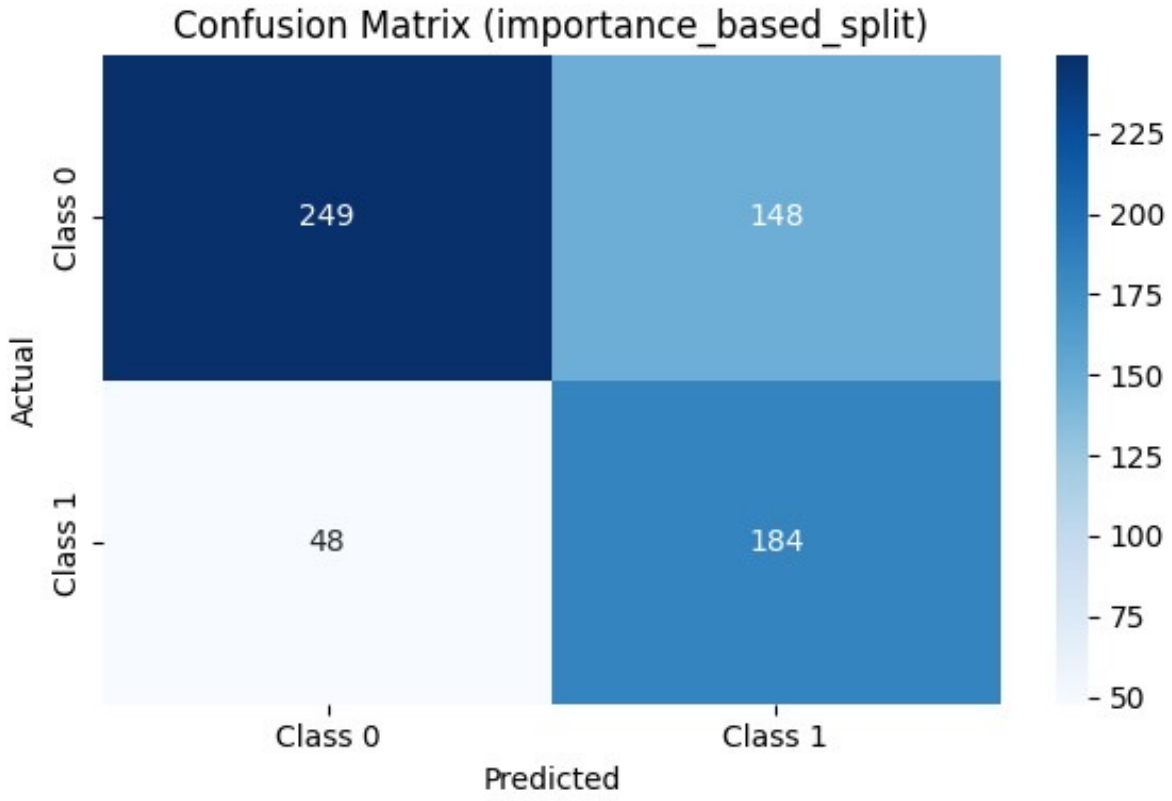


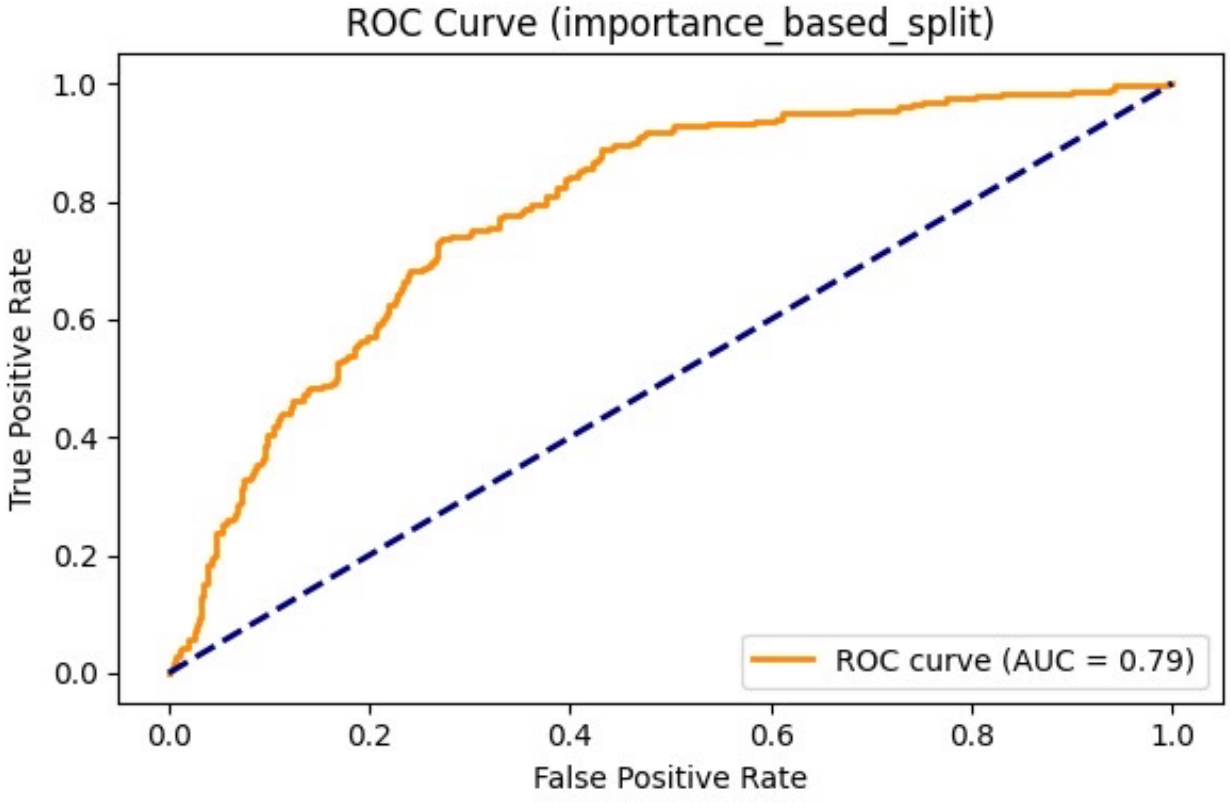



Precision Heatmap for Methods and View Size Ratios
random_split_features
pca_based_split
correlation_based_split
importance_based_split
0.5854
0.5960
0.5583
0.5665
0.5922
0.5895
0.5669
0.6226
0.5870
0.5595
0.5890
0.6146
Method
0.3
0.5
0.7
View Size Ratio
0.62
0.61
0.60
0.59
0.58
0.57
0.56
Precision


Recall Heatmap for Methods and View Size Ratios
random_split_features
pca_based_split
correlation_based_split
importance_based_split
0.7336
0.7860
0.7948
0.7817
0.7293
0.7336
0.6288
0.7205
0.7074
0.7598
0.7511
0.7729
Method
0.3
0.5
0.7
View Size Ratio
0.78
0.76
0.74
0.72
0.70
0.68
0.66
0.64
Recall

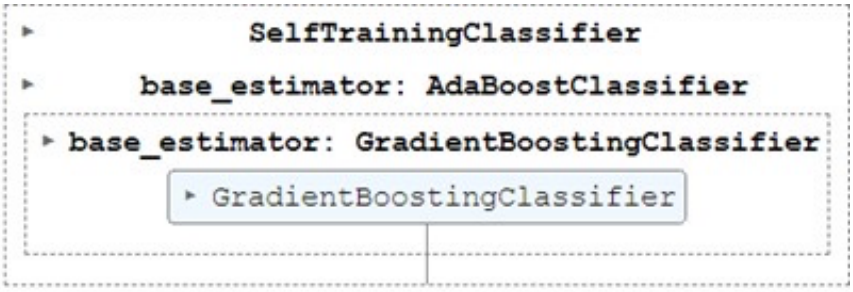

Auc Heatmap for Methods and View Size Ratios
random_split_features
pca_based_split
correlation_based_split
importance_based_split
0.7772
0.7954
0.7928
0.8046
0.7917
0.8056
0.7474
0.7849
0.7898
0.7927
0.7986
0.8064
Method
0.3
0.5
0.7
View Size Ratio
0.80
0.79
0.78
0.77
0.76
0.75
Auc





3. Semi-boosting

The core of the algorithm is a **combination** of AdaBoost and self-training, also known as an ensemble. It is also a semi-supervised learning technique. The structure of our implementation is shown below:



Base Estimator

Firstly, we have the **Gradient Boosting** classifier from scikit-learn with default parameters as the base estimator. The classifier is then trained with the entire training set (with labelled and unlabelled data). Pseudo-labels are applied to unlabelled samples.

AdaBoost Classifier:

After training the Gradient Boosting classifier, it is applied to **AdaBoost** with default parameters. The AdaBoost classifier serves the following purposes:

- To boost the performance of weak learners (here, a **Gradient Boosting** with **depth 1**), by iteratively focusing on hardly classified samples.
- To combine multiple weak classifiers to form a strong classifier.

Self-Training Classifier:

Finally, the ensemble has been further applied to a **self training classifier** from scikit-learn with also default parameters. It serves the following purposes:

- To leverage a base estimator (in this case, **AdaBoost**) to pseudo-label unlabeled instances.
- To iteratively train on the pseudo-labeled data in order to improve performance.

Initialize base estimator as a weak learner

```
base_estimator = GradientBoostingClassifier(max_depth=1, random_state=42)
```

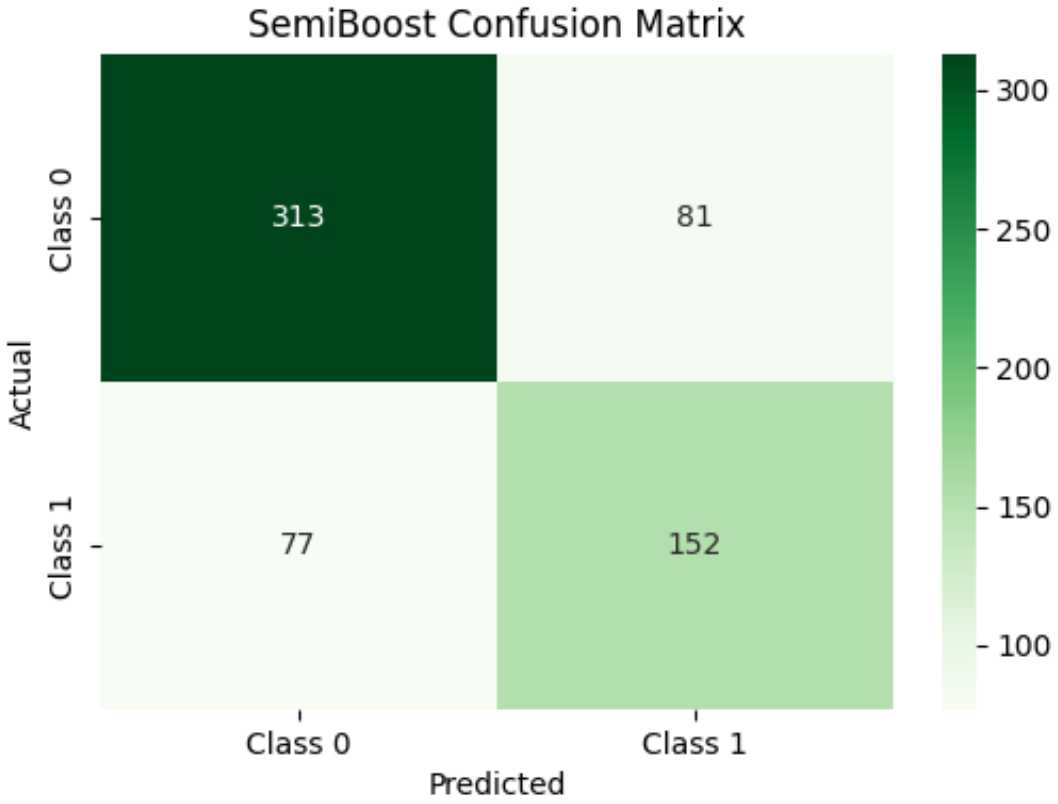
Initialize AdaBoost Classifier with the weak learner

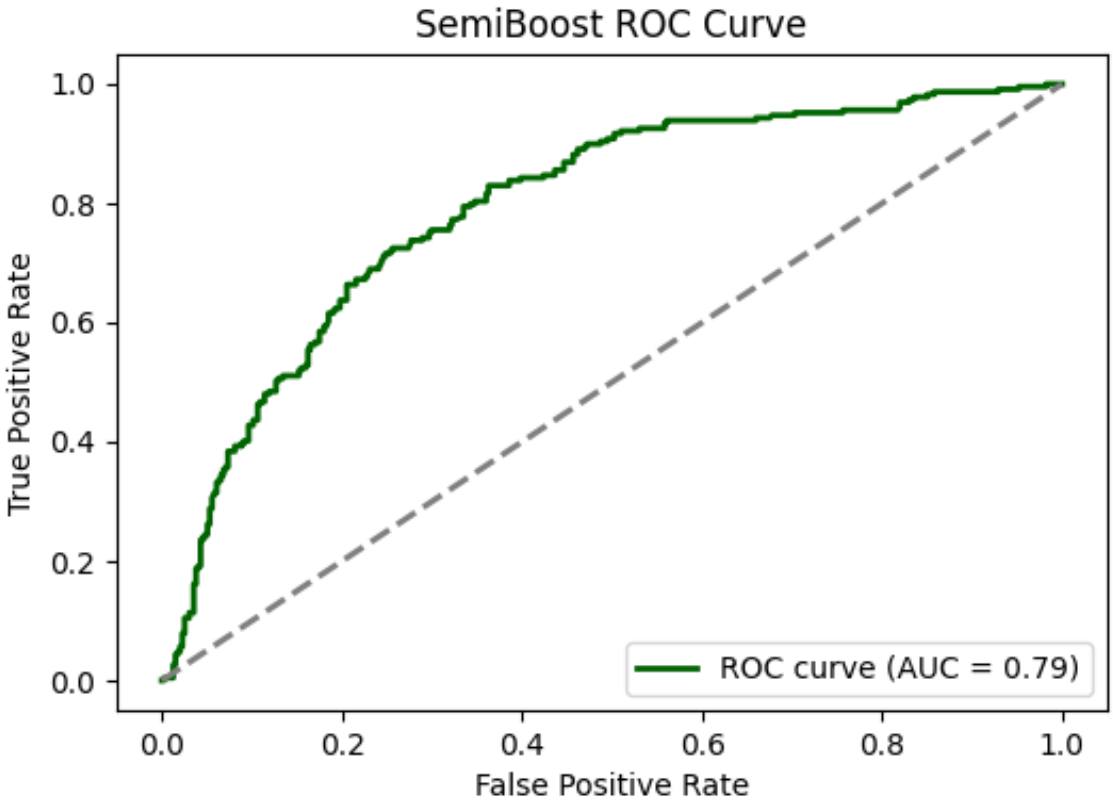
```
ada_clf = AdaBoostClassifier(estimator=base_estimator, n_estimators=50, random_state=42)
```

Initialize Self-Training Classifier with AdaBoost as base

```
self_training_clf = SelfTrainingClassifier(base_estimator=ada_clf, threshold=0.8, max_iter=10, verbose=True)
```

SemiBoost Test Accuracy: 0.7464
SemiBoost AUC-ROC Score: 0.7929





In short, our semi-boost classifier is an ensemble formed with Gradient Boosting algorithm at the base layer, then the Ada Boost algorithm, and the Self Training algorithm at the top layer.

4.Label spreading

Definition: Intrinsically semi-supervised learning methods are algorithms specifically designed to utilize both labeled and unlabeled data simultaneously during the training process. They inherently incorporate the structure and distribution of the unlabeled data to improve learning.

Label Spreading is a semi-supervised learning algorithm that **propagates** label information from labeled instances to unlabeled instances using a **similarity graph**. It is also more robust to noise.

This technique relies on the principle that points close to each other in the feature space are likely to have similar labels. Label Spreading is an intrinsically **semi-supervised** learning method because it directly incorporates both labeled and unlabeled data into a single model by constructing a graph where labels are propagated from labeled to unlabeled instances based on **data similarity**; unlike unsupervised pretraining, which separately learns representations from unlabeled data before supervised training, or semi-supervised ensembles, which combine multiple models, Label Spreading inherently leverages unlabeled data within its core algorithm without relying on separate pretraining phases or ensemble techniques.

In our implementation, we applied **Label Spreading** with an **RBF** (Radial Basis Function) kernel to construct the similarity graph. It iteratively spreads the label information **until convergence** or the maximum number of iterations is reached.

The algorithm constructs a graph where each node is a **data point**. Edges are **weighted** based on the similarity between data points.

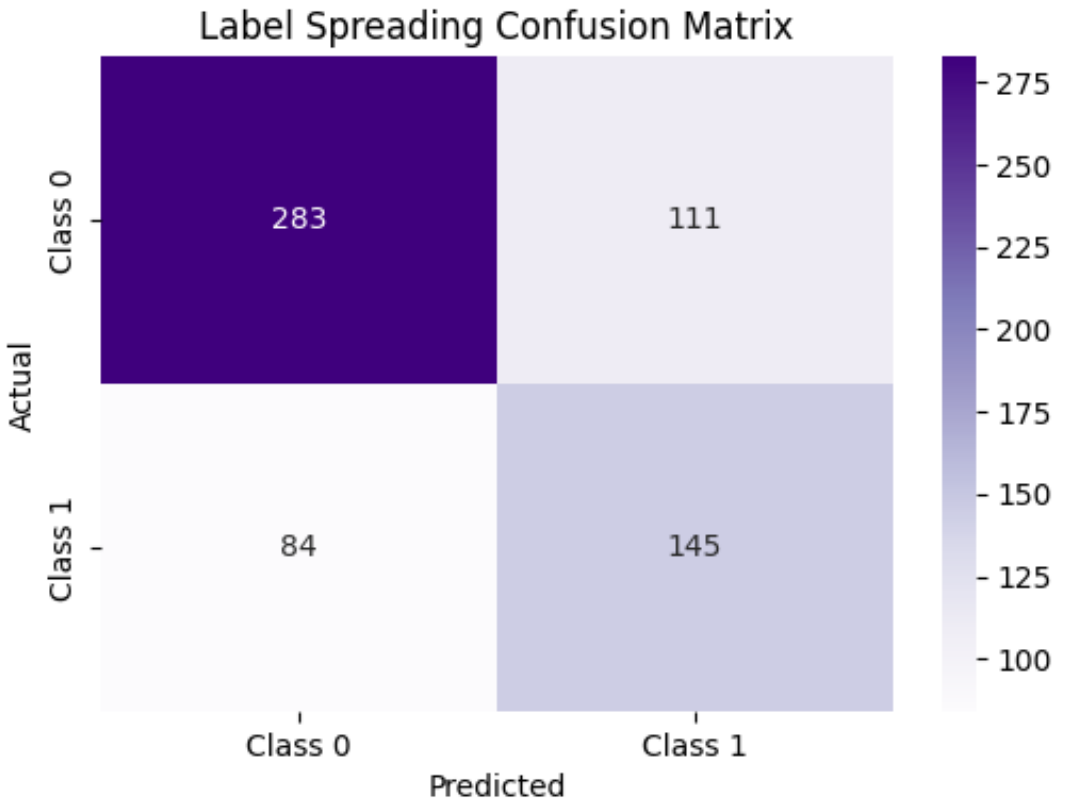
$$w_i j = exp(-\gamma * ||x_i - x_j||^2) \tag{1}$$

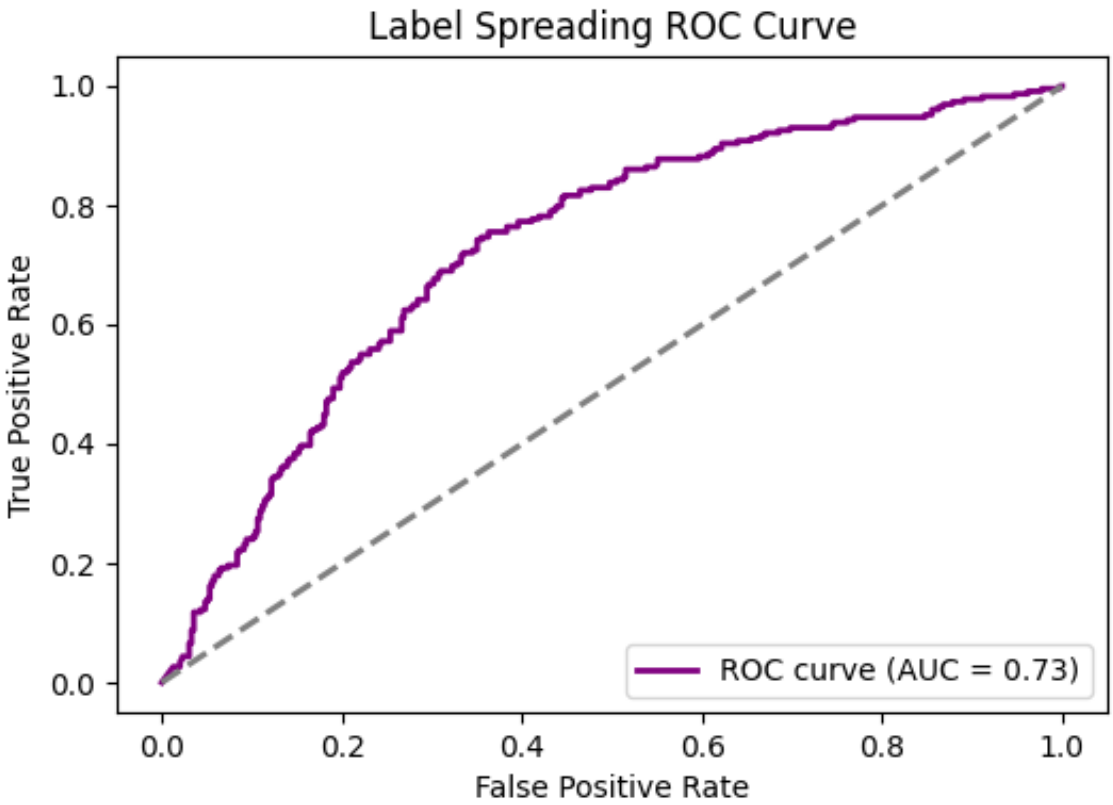
```
label_spread = LabelSpreading(kernel='rbf', gamma=20, max_iter=30)
```

Kernel: `'rbf'` is used to compute the similarity between points based on **distance**.

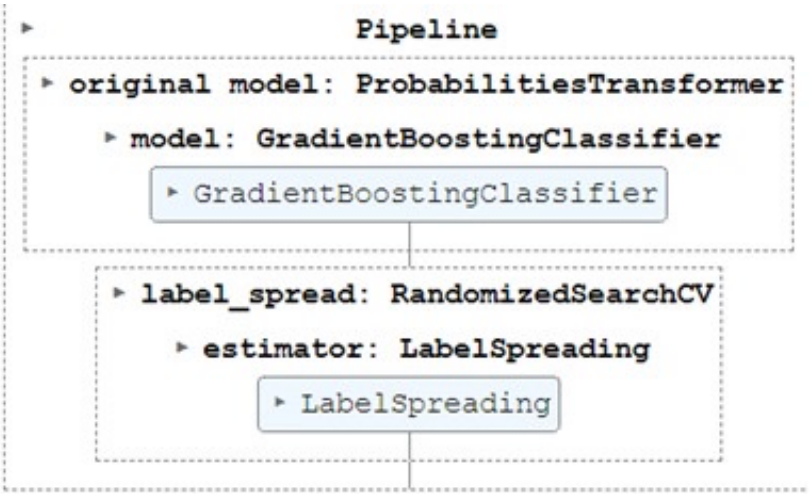
Gamma: Controls the **scale** of the RBF kernel (higher values focus on closer points).

Max Iterations: Limits the number of label propagation steps.





In our implementation, a complex structure is used, which is a **Pipeline** structure.



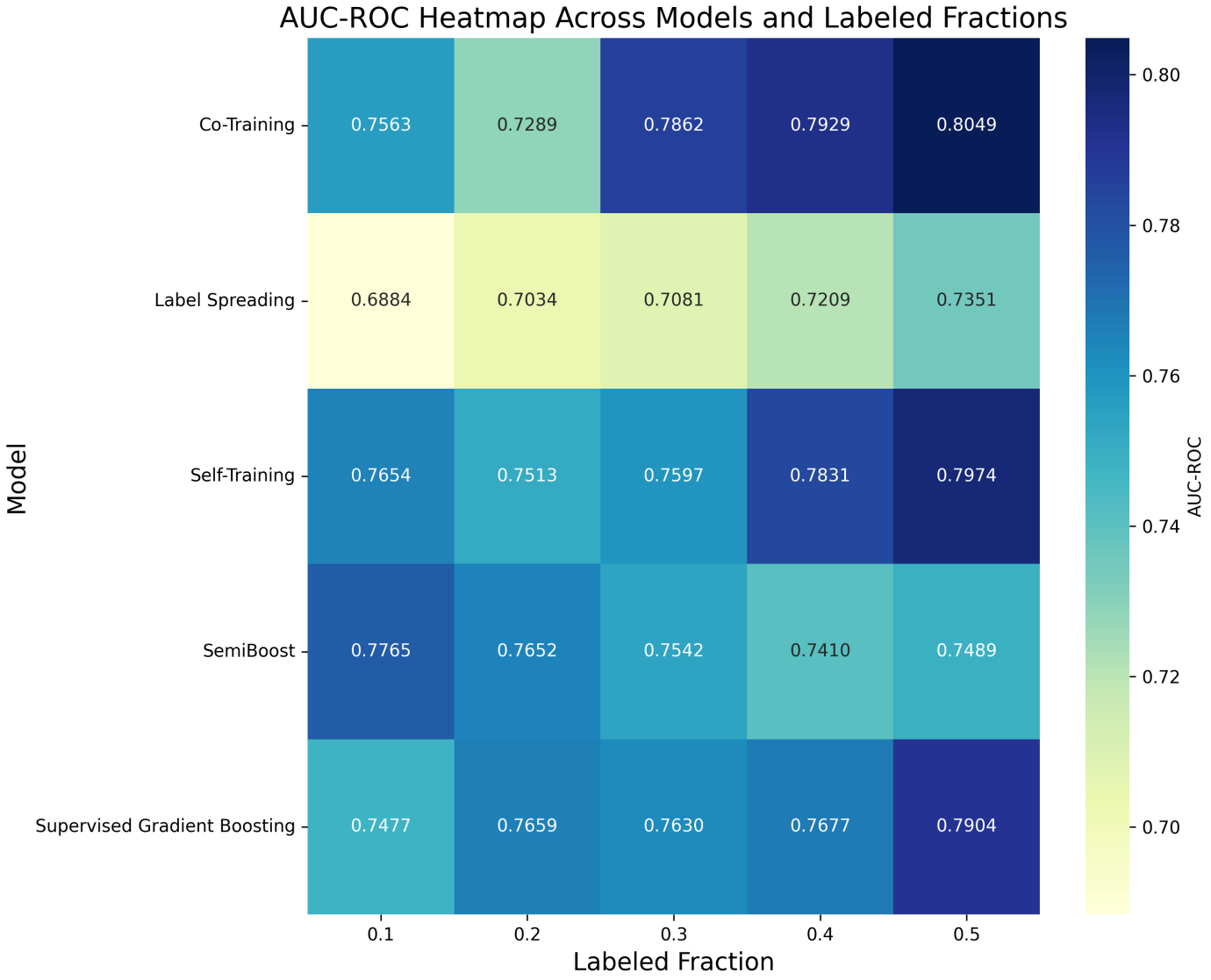
The pipeline consists of mainly 2 parts. The base estimator is the label spreading classifier **from scikit-learn**, which has been applied **randomized search** to have a set of optimized parameters. Another part of the label spreading classifier is a **self-defined class** which consists of a training function and a predicting function. The predicting function outputs a set of probabilities of each class for each sample (informally known as "predict_proba"). It is applied to the **pre-trained Gradient Boosting classifier** from scikit-learn with default parameters.

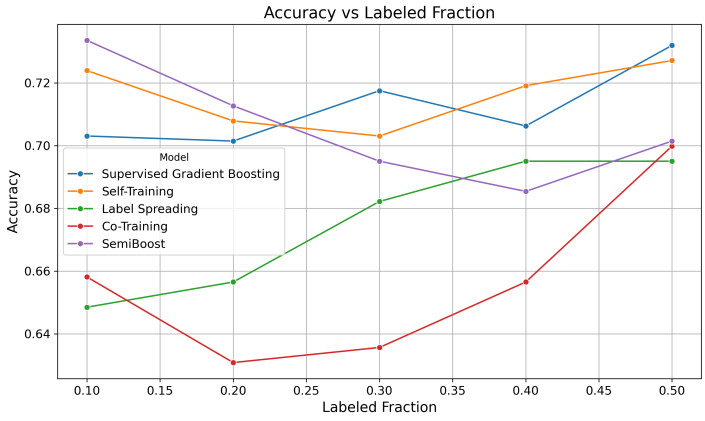
Evaluation

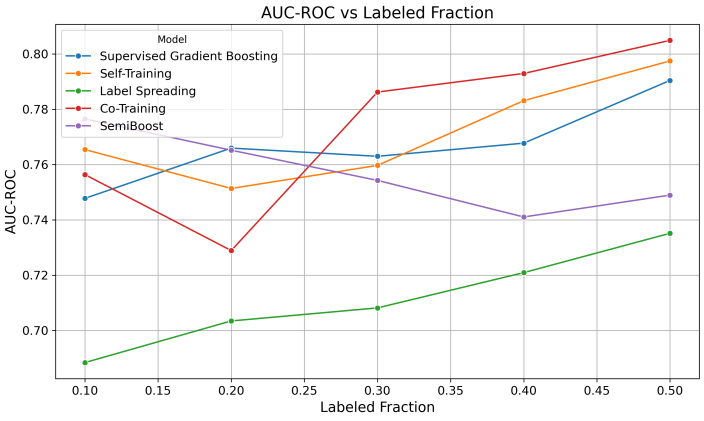
The baseline model (i.e., the **Supervised Gradient Boosting** classifier) and **Self-Training** consistently deliver the highest **accuracy** and **AUC (from the ROC curve)**, improving with more labeled data. **Label Spreading** shows steady progress but lags slightly, while **Co-Training** struggles with an erratic performance, especially at smaller labeled fractions.

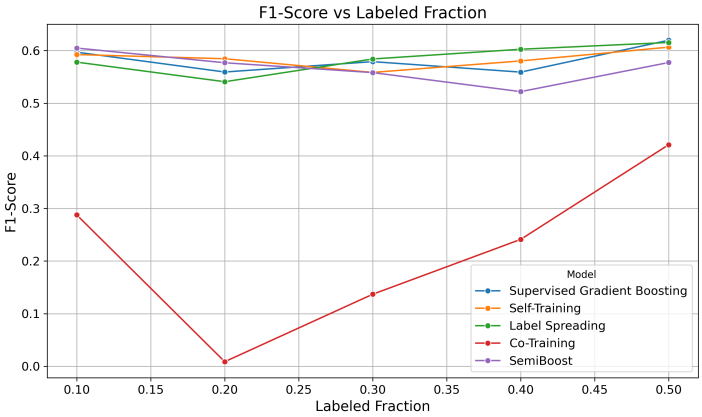
Precision and **F1-Score** are highest for the **Supervised Gradient Boosting**. **Self-Training** performs similarly well which can be an alternative to our semi-supervised learning problem. **Label Spreading** excels in **recall**, making it ideal for capturing positive instances, though its **precision** is weaker.

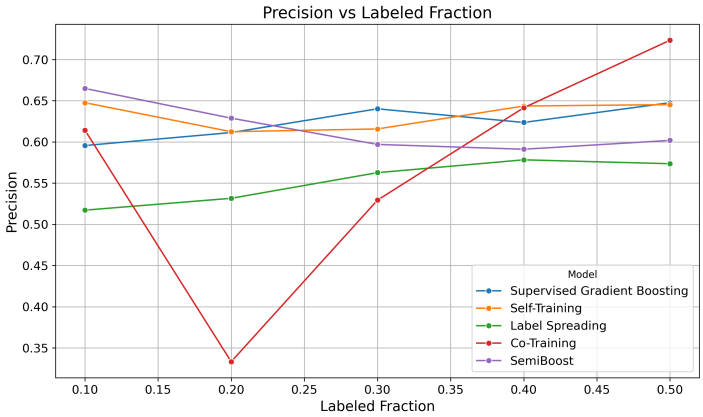
Overall, the **Supervised Gradient Boosting** is the most reliable. **Self-Training** is a strong semi-supervised option. **Label Spreading** is best suited for recall-focused tasks.

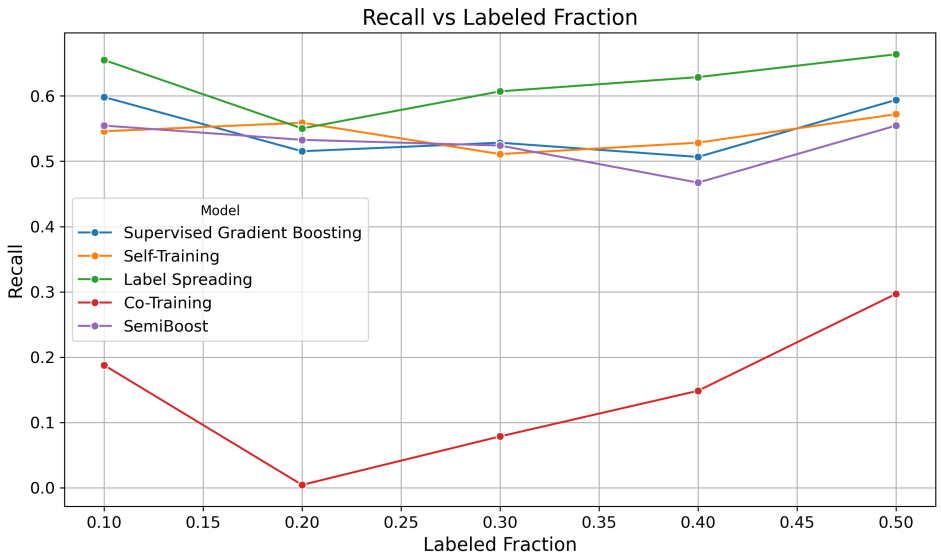

AUC-ROC Heatmap Across Models and Labeled Fractions
Co-Training
Label Spreading
Self-Training
SemiBoost
Supervised Gradient Boosting
0.7563
0.7289
0.7862
0.7929
0.8049
0.6884
0.7034
0.7081
0.7209
0.7351
0.7654
0.7513
0.7597
0.7831
0.7974
0.7765
0.7652
0.7542
0.7410
0.7489
0.7477
0.7659
0.7630
0.7677
0.7904
Model
0.1
0.2
0.3
0.4
0.5
Labeled Fraction
0.80
0.78
0.76
0.74
0.72
0.70
AUC-ROC


Accuracy vs Labeled Fraction
Accuracy
Labeled Fraction
Model
Supervised Gradient Boosting
Self-Training
Label Spreading
Co-Training
SemiBoost

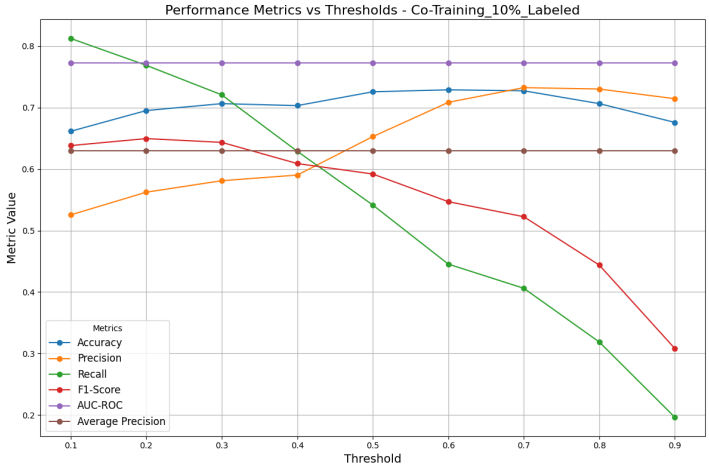

AUC-ROC vs Labeled Fraction
AUC-ROC
Labeled Fraction
Model
Supervised Gradient Boosting
Self-Training
Label Spreading
Co-Training
SemiBoost

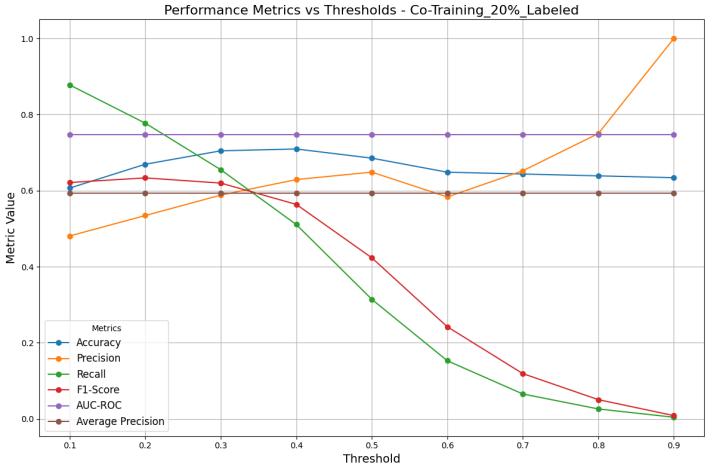


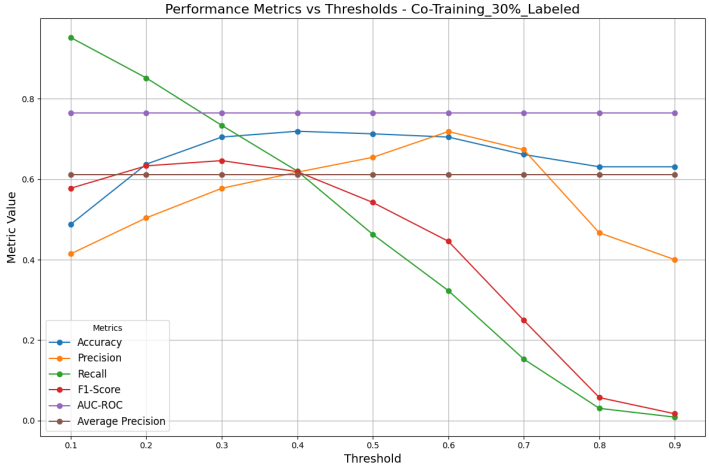


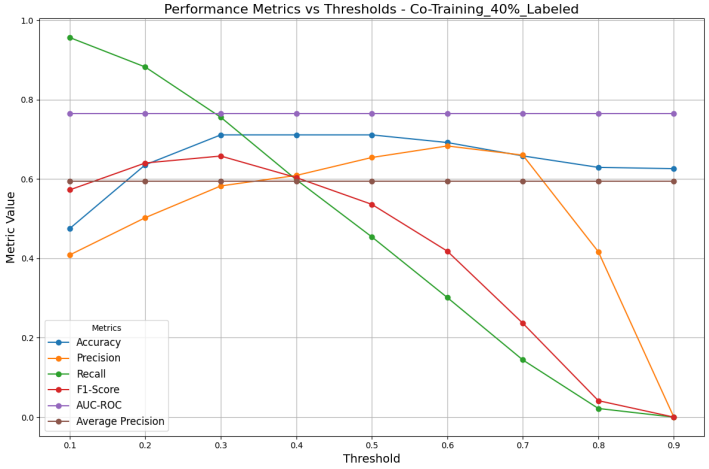


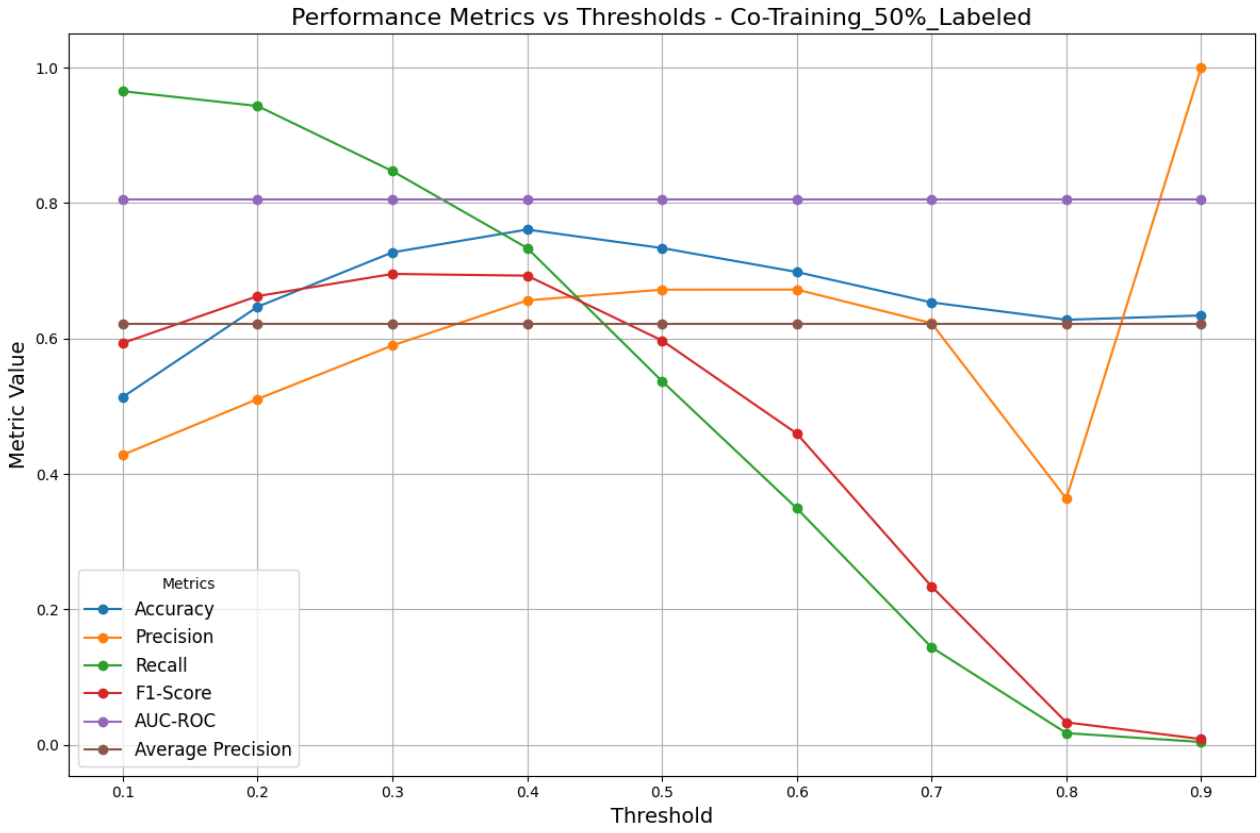
In the above plots, I observed that Co-Training achieved a high AUC-ROC value but exhibited low Recall and F1 scores, which initially perplexed me. To investigate this discrepancy, I conducted several experiments. Since AUC-ROC is calculated across all thresholds while Recall is evaluated only at a specific threshold, this suggests that Co-Training may perform better at certain thresholds than the one I initially set. I then tested Co-Training's performance under different settings. From the subsequent plots, we can see that when the threshold is around 0.3, the Co-Training method generally performs better, possibly because it assigns lower probability values. This explains why Co-Training typically has high AUC-ROC scores while Recall remains low, as AUC-ROC accounts for all threshold scenarios, whereas Recall is assessed only at the default threshold.


Performance Metrics vs Thresholds - Co-Training_10%_Labeled
Metric Value
Threshold
Metrics
Accuracy
Precision
Recall
F1-Score
AUC-ROC
Average Precision


Performance Metrics vs Thresholds - Co-Training_20%_Labeled
Metric Value
Threshold
Metrics
Accuracy
Precision
Recall
F1-Score
AUC-ROC
Average Precision


Performance Metrics vs Thresholds - Co-Training_30%_Labeled
Metric Value
Threshold
Metrics
Accuracy
Precision
Recall
F1-Score
AUC-ROC
Average Precision


Performance Metrics vs Thresholds - Co-Training_40%_Labeled
Metric Value
Threshold
Metrics
Accuracy
Precision
Recall
F1-Score
AUC-ROC
Average Precision


Performance Metrics vs Thresholds - Co-Training_50%_Labeled
Metric Value
Threshold
1.0
0.8
0.6
0.4
0.2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
Metrics
Accuracy
Precision
Recall
F1-Score
AUC-ROC
Average Precision