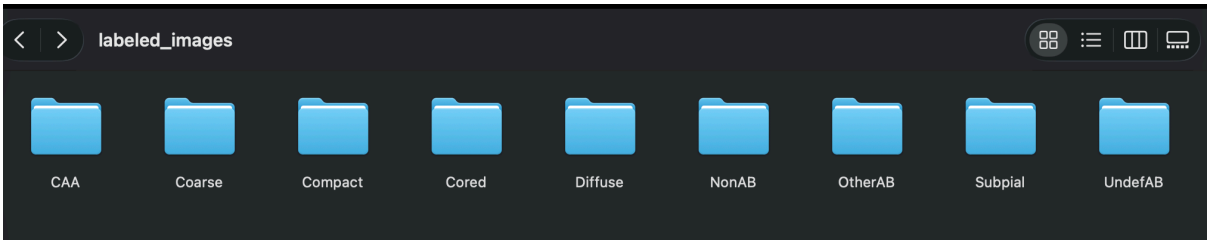
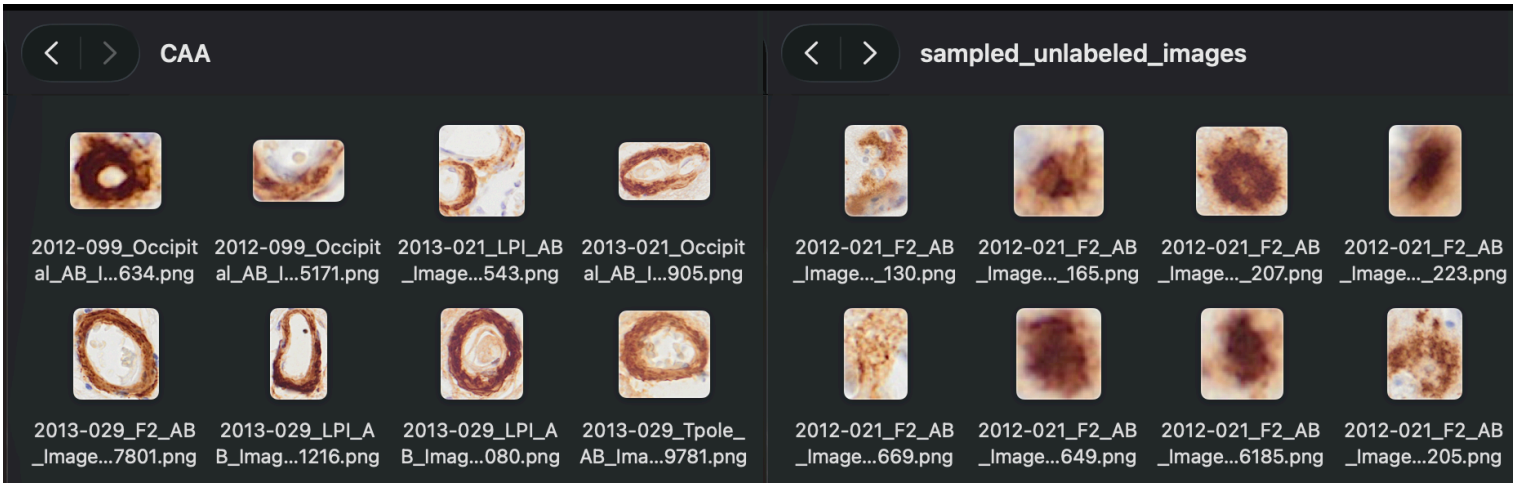
Honours Project Report

Already Implemented:

1- Data Preparation:

Segmented plaque images were extracted from large HDF5 files and organized into folders by label. For each plaque, additional morphological features (roundness and area) were extracted from the HDF5 metadata and stored in a data table alongside the corresponding image filename and index. The data table contains columns for Image (source HDF5 filename), Index (plaque identifier within the file), Roundness, Area, and Label (for labeled samples). Labeled images are stored in subdirectories named by their class label (e.g., Diffuse, Compact, Cored, Subpial, CAA), while unlabeled images are stored in a separate folder. Each segmented plaque is saved as a PNG file with the naming convention “{image_name}_index_{index}.png”. All images were downsampled to 224×224 pixels to ensure uniform dimensions for training. Given the large number of unlabeled plaques (approximately 4 million), a random sample of 10,000 unlabeled plaques was selected to create a manageable dataset for training and evaluation.





data_table_sampled_10k

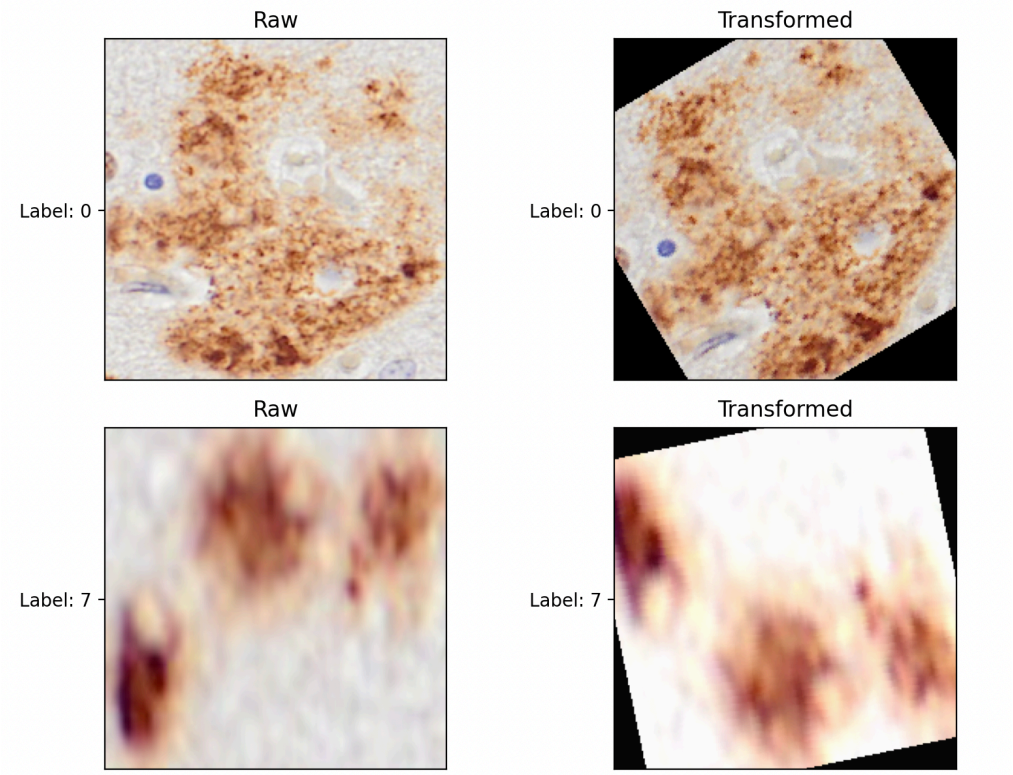
Image	Index	Roundness	Area	Label
Image_2014-010_Occipital_AB.hdf5	10985	0.3599173420565950	880.5664040000000	Diffuse
Image_2016-008_F2_BA4.hdf5	2444	0.5115943894106370	1326.6679960000000	Diffuse
Image_2019-025_LPI_AB.hdf5	10349	0.3008987961916100	1029.4421120000000	Diffuse
2018-014_F2_AB_Image.hdf5	9700	0.6879559486933610	347.3766520000000	Diffuse
Image_2017-027_F2_BA4.hdf5	9487	0.6519894635419590	256.83499600000000	Diffuse
Image_2014-079_F2_AB.hdf5	10812	0.37812738499065800	205.85839200000000	Diffuse
Image_2013-052_LPI_AB.hdf5	9203	0.4314351613764410	164.26628800000000	Diffuse
Image_2014-010_Occipital_AB.hdf5	11223	0.04234867976433140	10657.338504000000	Subpial
2015-067_F2_AB_Image.hdf5	4004	0.675487478161061	207.05960800000000	Compact
Image_2021-091_temporalA_AB4.hdf5	294	0.5810831902771800	2132.9091600000000	Cored

2- Data Preprocessing:

The next step was designing a data preprocessing pipeline. For this, **PlaqueDataset** module was implemented to handle image loading, normalization, transformation, and augmentation. The class accepts parameters for normalization statistics (mean and standard deviation), transformation pipelines.

The **PlaqueDataset** returns multiple outputs: the raw normalized image tensor, a stacked tensor of all transformed/augmented views, extra morphological features (if enabled), and the label. When multiple transforms are provided as a list, all transformed views are stacked along a new dimension, allowing simultaneous access to different augmentations of the same image. This design is useful for semi-supervised and self-supervised learning, where pairs of augmentations (e.g., weak and strong) are needed for consistency regularization.

To support supervised learning, an augmented variant called **PlaqueDatasetAugmented** module was implemented. This variant expands the dataset by treating each augmentation as a separate sample. Internally, it creates multiple **PlaqueDataset i**nstances one for each augmentation plus one for the raw image effectively multiplying the dataset size by (number_of_augmentations+1). This allows augmentations to serve as additional training samples that can be shuffled independently by the data loader, which is beneficial for supervised approaches. In contrast, the standard **PlaqueDataset** preserves the relationship between different views of the same image, which is essential for consistency-based semi-supervised methods. Both dataset classes support optional data preloading for faster training, on-the-fly transformation application, and configurable image downscaling methods (bilinear or nearest neighbor). The specific transformations used vary by architecture and training approach, and will be detailed in the respective sections.



3- Configuration Manager System

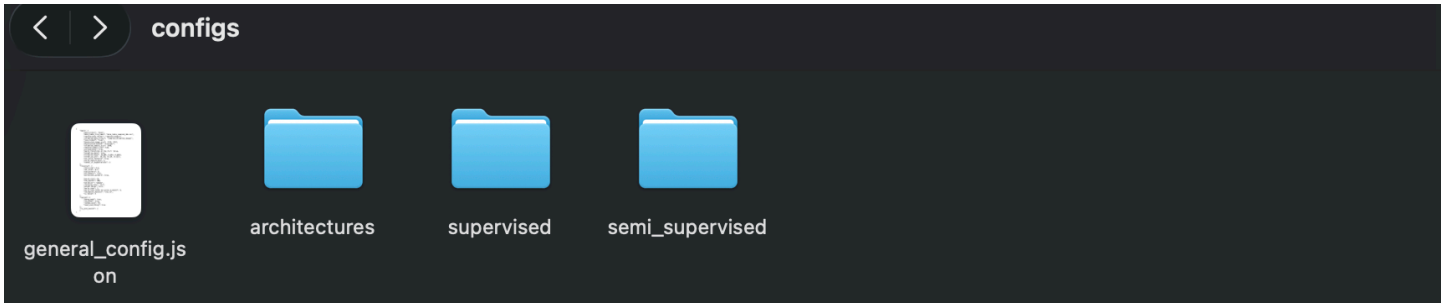
A configuration management system was implemented to centralize experiment settings and simplify parameter management. The **Config** class loads and merges JSON files from a structured directory hierarchy, providing a hierarchical, object-like interface.

The configuration directory is organized hierarchically. At the root level, general_config.json contains shared settings for data preprocessing, training hyperparameters, and system configuration. Subdirectories organize mode-specific and model-specific configurations: **supervised** folder contains supervised learning configurations, **semi_supervised** folder contains semi-supervised method configurations (e.g., Mean Teacher, FixMatch, Pi-Model), and **architecture** folder contains feature extractor and classifier configurations. Each model has its own configuration file, allowing independent parameter management.

The **Config** class recursively loads all JSON files in this structure and converts nested dictionaries into nested **Config** objects. This enables dot-notation access like **config.general_config.training.batch_size** instead of dictionary indexing, making the configuration more readable and easier to use throughout the codebase.

When loading configurations, the system automatically performs several post-processing steps. It loads label mappings from a CSV file and adds them to the config object. It generates a unique run ID (either from the SLURM job ID if available, or from the current timestamp). It automatically detects and sets the device to "cuda" if available, otherwise "cpu". Finally, it filters out irrelevant configuration sections based on the training mode (supervised, semi_supervised, or self_supervised), ensuring that only the relevant configurations are loaded for each experiment type.

This configuration system enables easy experimentation by allowing researchers to modify JSON files without changing code, and it ensures reproducibility by saving the complete configuration with each experiment run.



An example of general_config.json

configs > {} general_config.json > ...

```json
{
    "data": {
        "data_folder": "data",
        "data_table_file_name": "data_table_sampled_10k.csv",
        "labeled_data_folder": "labeled_images",
        "unlabeled_data_folder": "sampled_unlabeled_images",
        "runs_folder": "runs",
        "downscaled_image_size": [224, 224],
        "downscaling_method": "bilinear",
        "unlabeled_sample_size": 2000,
        "labeled_sample_size": 1e9,
        "preload_data": true,
        "apply_transforms_on_the_fly": false,
        "normalize_data": true,
        "normalize_mean": [0.485, 0.456, 0.406],
        "normalize_std": [0.229, 0.224, 0.225],
        "use_extra_features": true,
        "extra_feature_dim": 2,
        "number_of_augmentations": 3
    },
    "training": {
        "test_size": 0.2,
        "val_size": 0.1,
        "num_workers": 4,
        "pin_memory": true,
        "persistent_workers": true,

        "batch_size": 64,
        "num_epochs": 500,
        "optimizer": "adamw",
        "learning_rate": 1e-4,
        "weight_decay": 1e-5,
        "early_stop": 2,
        "early_stop_check_val_every_n_epoch": 5,
        "checkpoint_monitor": "val_f1",
        "cv_folds": 5
    },
    "system": {
        "debug_mode": true,
        "log_mode": true,
        "random_seed": 44,
        "seed_everything": true
    },
    "cv_grid_search": {
    }
}
```

And then further on in **main.py,** which is the entry point of the program, the **config_dir** is given as a parameter alongside the **train_mode** and the **run_mode,** and then a **Runner** class instance is created, which is responsible for connecting all the data preprocessing logic to the actual training logic and logging and saving the results. We will not go into the details of them in this report.

src > main.py > ...

```python
import argparse

from utils.logging_utils import print_log
from models.config import Config
from models.base_runner import BaseRunner

if __name__ == "__main__":
    # Parse arguments with config files
    parser = argparse.ArgumentParser(description="Plaque Analysis with Config Files")

    parser.add_argument(
        "--config_dir",
        type=str,
        default="configs",
        help="Directory containing config files",
    )
    parser.add_argument(
        "--train_mode",
        type=str,
        default="supervised",
        choices=["supervised", "semi_supervised", "self_supervised"],
        help="Training mode to use",
    )
    parser.add_argument(
        "--run_mode",
        type=str,
        default="single",
        choices=["single", "cross_validate", "optimize_hyperparameters"],
        help="Run mode to use",
    )
    # Parse arguments
    args = parser.parse_args()

    # Load and merge configurations
    config = Config.load_config(args.config_dir, args.train_mode)

    print_log(
        "Config: " + str(config),
        log_mode=config.general_config.system.log_mode,
        end="\n\n",
    )
    runner = BaseRunner.create_runner(args.train_mode, config)

    if args.run_mode == "single":
        runner.run_single_experiment()
    elif args.run_mode == "cross_validate":
        runner.cross_validate()
    elif args.run_mode == "optimize_hyperparameters":
        runner.optimize_hyperparameters()
    else:
        raise ValueError(f"Invalid run mode: {args.run_mode}")
```

4- Supervised base model:

A supervised baseline model was implemented to compare semi-supervised and self-supervised approaches. The implementation uses PyTorch Lightning for training, validation, and testing.

The architecture is modular with two components: a feature extractor and a classification head. The feature extractor supports multiple ResNet backbones (ResNet-18, ResNet-34, ResNet-50) using ImageNet pretrained weights. The final classification layer is removed, and the backbone is followed by a linear projection layer with ReLU activation and dropout regularization. The feature extractor can be frozen or fine-tuned and for this part we froze the feature extractor (resnet model) and only trained the linear head.

The classification head can be either a linear layer or a multi-layer perceptron (MLP) with configurable hidden layers and dropout. Optionally, morphological features (roundness and area) can be concatenated with the image features before classification, providing additional information to the model.

The model is implemented as a PyTorch Lightning module (**LightningSupervisedModule**) that encapsulates the feature extractor and classifier, handles training/validation/test steps, and manages metrics tracking. Training uses cross-entropy loss on labeled data only, with standard data augmentation and normalization techniques. This supervised baseline serves as the reference point for evaluating the performance improvements achieved by semi-supervised and self-supervised learning methods. The following section presents the performance results of this baseline model.

Results of 5-fold cross validation:

	precision	recall	f1-score	support
CAA	0.842 ± 0.179	0.915 ± 0.121	0.872 ± 0.063	16.2 ± 0.8
Coarse	0.791 ± 0.1	0.786 ± 0.137	0.785 ± 0.061	21.4 ± 0.98
Compact	0.883 ± 0.209	0.84 ± 0.185	0.854 ± 0.134	12.6 ± 0.98
Cored	0.728 ± 0.083	0.822 ± 0.247	0.77 ± 0.142	15.8 ± 0.8
Diffuse	0.894 ± 0.133	0.987 ± 0.054	0.937 ± 0.08	14.8 ± 0.8
NonAB	0.913 ± 0.221	0.973 ± 0.066	0.938 ± 0.129	14.6 ± 0.98
OtherAB	0.463 ± 0.689	0.195 ± 0.252	0.255 ± 0.308	6.2 ± 0.8
Subpial	0.766 ± 0.306	0.647 ± 0.179	0.698 ± 0.223	8.6 ± 0.98
UndefAB	0.95 ± 0.2	0.8 ± 0.339	0.858 ± 0.233	7.6 ± 0.98
macro avg	0.803 ± 0.152	0.774 ± 0.098	0.774 ± 0.109	117.8 ± 0.8
weighted avg	0.82 ± 0.1	0.822 ± 0.076	0.811 ± 0.082	117.8 ± 0.8

Transformation used to augment new labeled samples to training set:

src › supervised_runner.py › SupervisedRunner › _load_dataloaders

```
36   class SupervisedRunner(BaseRunner):
365      def _load_dataloaders(
375          train_transforms = trf.Compose(
376              [
377                  trf.RandomHorizontalFlip(p=0.5),
378                  trf.RandomVerticalFlip(p=0.5),
379                  trf.RandomRotation(degrees=(0, 90)),
380                  trf.ColorJitter(brightness=0.2, contrast=0.2),
381                  trf.ToTensor(),
382              ]
383          )
```

Configuration:

```
{
        supervised: {
                supervised_config: {
                        feature_extractor_name: resnet18,
                        classifier_name: linear,
                }
        },
        architectures: {
                classifiers_config: {
                        linear: {},
                },
                feature_extractors_config: {
                        resnet18: {
                                output_size: 64,
                                pretrained: True,
```

```
                freeze_feature_extractor: True,
                dropout_rate: 0.2,
            },
        }
    },
    general_config: {
        data: {
            unlabeled_sample_size: 2000,
            labeled_sample_size: 1000000000.0,
            preload_data: True,
            apply_transforms_on_the_fly: False,
            normalize_data: True,
            normalize_mean: [0.485, 0.456, 0.406],
            normalize_std: [0.229, 0.224, 0.225],
            use_extra_features: True,
            extra_feature_dim: 2,
            number_of_augmentations: 3
        },
        training: {
            test_size: 0.2,
            val_size: 0.1,
            batch_size: 64,
            num_epochs: 500,
            optimizer: adamw,
            learning_rate: 0.0001,
            weight_decay: 1e-05,
            early_stop: 2,
            early_stop_check_val_every_n_epoch: 5,
            cv_folds: 5
        },
        system: {
            random_seed: 44
        },
    },
}
```

5- Semi-supervised models:

5-1 PI model:

The PI Model (Π-Model) was implemented as a semi-supervised learning approach using consistency regularization. The idea is that different augmentations of the same image should produce similar predictions.

The implementation extends the base semi-supervised module and follows a two-stream training process. For labeled data, it computes a standard supervised loss using ground truth labels. For unlabeled data, it applies two augmentations to each image: a weak augmentation (minimal transformations) and a strong augmentation (more aggressive transformations). Both augmented versions are passed through the same model to obtain predictions.

The consistency loss measures the disagreement between the predictions on the weak and strong augmentations, encouraging the model to be invariant augmentation transformations, effectively learning from unlabeled data. The consistency loss can be computed using different metrics, such as mean squared error between the prediction distributions.

The total training loss combines the supervised loss on labeled data and the consistency loss on unlabeled data, weighted by a consistency coefficient. To prevent early instability, this coefficient starts small and gradually increases during training using a ramp-up schedule. This allows the model to first learn from labeled data before incorporating unlabeled data.

This approach leverages unlabeled data by enforcing prediction consistency across augmentations, improving generalization without requiring additional labeled examples.

Algorithm 1 Π-model pseudocode.

Require: x_i = training stimuli
Require: L = set of training input indices with known labels
Require: y_i = labels for labeled inputs $i \in L$
Require: $w(t)$ = unsupervised weight ramp-up function
Require: $f_\theta(x)$ = stochastic neural network with trainable parameters θ
Require: $g(x)$ = stochastic input augmentation function
for t in $[1, num\_epochs]$ **do**
  for each minibatch B **do**
    $z_{i \in B} \leftarrow f_\theta(g(x_{i \in B}))$ ▷ evaluate network outputs for augmented inputs
    $\tilde{z}_{i \in B} \leftarrow f_\theta(g(x_{i \in B}))$ ▷ again, with different dropout and augmentation
    $loss \leftarrow -\frac{1}{|B|} \sum_{i \in (B \cap L)} \log z_i[y_i]$ ▷ supervised loss component
      $+ w(t) \frac{1}{C|B|} \sum_{i \in B} ||z_i - \tilde{z}_i||^2$ ▷ unsupervised loss component
    update θ using, e.g., ADAM ▷ update network parameters
  end for
end for
return θ

Results of 5-fold cross validation:

	precision	recall	f1-score	support
CAA	0.874 ± 0.163	0.729 ± 0.146	0.791 ± 0.104	16.2 ± 0.8
Coarse	0.676 ± 0.056	0.813 ± 0.137	0.736 ± 0.054	21.4 ± 0.98
Compact	0.778 ± 0.089	0.732 ± 0.21	0.752 ± 0.131	12.6 ± 0.98
Cored	0.683 ± 0.17	0.76 ± 0.288	0.715 ± 0.195	15.8 ± 0.8
Diffuse	0.787 ± 0.17	0.972 ± 0.068	0.869 ± 0.122	14.8 ± 0.8
NonAB	0.707 ± 0.141	0.987 ± 0.054	0.821 ± 0.089	14.6 ± 0.98
OtherAB	0.4 ± 0.98	0.067 ± 0.164	0.114 ± 0.28	6.2 ± 0.8
Subpial	0.8 ± 0.8	0.275 ± 0.277	0.409 ± 0.41	8.6 ± 0.98
UndefAB	0.819 ± 0.298	0.686 ± 0.102	0.738 ± 0.115	7.6 ± 0.98
macro avg	0.725 ± 0.199	0.669 ± 0.049	0.66 ± 0.082	117.8 ± 0.8
weighted avg	0.738 ± 0.118	0.74 ± 0.044	0.714 ± 0.057	117.8 ± 0.8

src › semi_supervised_runner.py › SemiSupervisedRunner › _load_dataloaders

```
37   class SemiSupervisedRunner(BaseRunner):
371      def _load_dataloaders(
391          weak_transforms = trf.Compose(
392              [
393                  trf.RandomHorizontalFlip(p=0.5),
394                  trf.RandomVerticalFlip(p=0.5),
395                  trf.ToTensor(),
396              ]
397          )
398          strong_transforms = trf.Compose(
399              [
400                  trf.RandAugment(num_ops=2, magnitude=10),
401                  trf.ToTensor(),
402              ]
403          )
```

Note that all the different semi-supervised models introduced later use these transformations.

Configuration:

```
{
        semi_supervised: {
                pi_model_config: {


                },
                semi_supervised_config: {
                        model_name: pi_model,
                        feature_extractor_name: resnet18,
                        classifier_name: linear,
                        training: {
                                consistency_lambda_max: 0.1,
                                consistency_loss_type: mse,
                                ramp_up_epochs: 100,
```

```
                        ramp_up_function: linear
                }
        }
},
architectures: {
        classifiers_config: {
                linear: {},
        },
        feature_extractors_config: {
                resnet18: {
                        output_size: 64,
                        pretrained: True,
                        freeze_feature_extractor: True,
                        dropout_rate: 0.2,
                },
        }
},
general_config: {
        data: {
                unlabeled_sample_size: 2000,
                labeled_sample_size: 1000000000.0,
                preload_data: True,
                apply_transforms_on_the_fly: False,
                normalize_data: True,
                normalize_mean: [0.485, 0.456, 0.406],
                normalize_std: [0.229, 0.224, 0.225],
                use_extra_features: True,
                extra_feature_dim: 2,
                number_of_augmentations: 3
        },
        training: {
                test_size: 0.2,
                val_size: 0.1,
                batch_size: 64,
                num_epochs: 500,
                optimizer: adamw,
                learning_rate: 0.0001,
                weight_decay: 1e-05,
```

```
                early_stop: 2,
                early_stop_check_val_every_n_epoch: 5,
                cv_folds: 5
            },
            system: {
                random_seed: 44
            },
        },
}
```

5-2 FixMatch:

Following the PI model, FixMatch was implemented as a semi-supervised approach that combines pseudo-labeling with consistency regularization. It adds confidence-based filtering to focus on high-confidence unlabeled samples.

The approach uses a two-stage process for unlabeled data. First, weakly augmented images are passed through the model to obtain prediction probabilities. The maximum softmax probability is used as a confidence score. If this score exceeds a predefined threshold, the predicted class becomes a pseudo-label for that sample. This threshold acts as a quality filter, ensuring only confident predictions are used.

In the second stage, strongly augmented versions of the same images are passed through the model. The consistency loss is computed only for samples that passed the confidence threshold, using cross-entropy between the strong augmentation predictions and the pseudo-labels derived from the weak augmentations. Samples below the threshold are ignored, preventing the model from learning from uncertain predictions.

The total training loss combines the supervised loss on labeled data with the consistency loss on high-confidence unlabeled samples, weighted by a consistency coefficient that ramps up during training. This selective use of unlabeled data helps the model learn from reliable pseudo-labels while avoiding noise from uncertain predictions, improving generalization with limited labeled data.

Algorithm 1 FixMatch algorithm.

Input: Labeled batch $\mathcal{X} = \{(x_b, p_b) : b \in (1, \ldots, B)\}$, unlabeled batch $\mathcal{U} = \{u_b : b \in (1, \ldots, \mu B)\}$, confidence threshold τ, unlabeled data ratio μ, unlabeled loss weight λ_u.
$\ell_s = \frac{1}{B} \sum_{b=1}^{B} \mathrm{H}(p_b, \alpha(x_b))$ {*Cross-entropy loss for labeled data*}
for $b = 1$ **to** μB **do**
    $q_b = p_{\mathrm{m}}(y \mid \alpha(u_b); \theta)$ {*Compute prediction after applying weak data augmentation of* u_b}
end for
$\ell_u = \frac{1}{\mu B} \sum_{b=1}^{\mu B} \mathbb{1}\{\max(q_b) > \tau\} \mathrm{H}(\arg\max(q_b), p_{\mathrm{m}}(y \mid \mathcal{A}(u_b))$ {*Cross-entropy loss with pseudo-label and confidence for unlabeled data*}
return $\ell_s + \lambda_u \ell_u$

Results of 5-fold cross validation:

	precision	recall	f1-score	support
CAA	0.89 ± 0.095	0.778 ± 0.057	0.829 ± 0.046	16.2 ± 0.8
Coarse	0.691 ± 0.191	0.822 ± 0.245	0.742 ± 0.138	21.4 ± 0.98
Compact	0.821 ± 0.089	0.639 ± 0.362	0.705 ± 0.185	12.6 ± 0.98
Cored	0.638 ± 0.105	0.76 ± 0.356	0.685 ± 0.188	15.8 ± 0.8
Diffuse	0.847 ± 0.2	0.972 ± 0.068	0.903 ± 0.125	14.8 ± 0.8
NonAB	0.768 ± 0.083	0.987 ± 0.054	0.863 ± 0.042	14.6 ± 0.98
OtherAB	0.38 ± 0.742	0.167 ± 0.298	0.21 ± 0.351	6.2 ± 0.8
Subpial	0.767 ± 0.777	0.367 ± 0.395	0.49 ± 0.505	8.6 ± 0.98
UndefAB	0.88 ± 0.244	0.789 ± 0.129	0.826 ± 0.123	7.6 ± 0.98
macro avg	0.742 ± 0.183	0.698 ± 0.072	0.695 ± 0.107	117.8 ± 0.8
weighted avg	0.757 ± 0.114	0.757 ± 0.041	0.737 ± 0.066	117.8 ± 0.8

Configuration:


```
{
    semi_supervised: {
        fixmatch_config: {
            pseudo_label_confidence_threshold: 0.95
        },
        semi_supervised_config: {
            model_name: fixmatch,
            feature_extractor_name: resnet18,
            classifier_name: linear,
            training: {
                consistency_lambda_max: 0.1,
                consistency_loss_type: cross_entropy,
                ramp_up_epochs: 100,
                ramp_up_function: linear
            }
        }
    },
    architectures: {
        classifiers_config: {
            linear: {},
        },
        feature_extractors_config: {
            resnet18: {
                output_size: 64,
                pretrained: True,
```

```
                freeze_feature_extractor: True,
                dropout_rate: 0.2,
            },
        }
    },
    general_config: {
        data: {
            unlabeled_sample_size: 2000,
            labeled_sample_size: 1000000000.0,
            preload_data: True,
            apply_transforms_on_the_fly: False,
            normalize_data: True,
            normalize_mean: [0.485, 0.456, 0.406],
            normalize_std: [0.229, 0.224, 0.225],
            use_extra_features: True,
            extra_feature_dim: 2,
            number_of_augmentations: 3
        },
        training: {
            test_size: 0.2,
            val_size: 0.1,
            batch_size: 64,
            num_epochs: 500,
            optimizer: adamw,
            learning_rate: 0.0001,
            weight_decay: 1e-05,
            early_stop: 2,
            early_stop_check_val_every_n_epoch: 5,
            cv_folds: 5
        },
        system: {
            random_seed: 44
        },
    },
}
```

5-3 MeanTeacher:

After FixMatch, the Mean Teacher method was implemented as another semi-supervised approach, this time based on a teacher–student framework. The core idea is to use a slowly updated teacher model to provide stable targets for consistency regularization on unlabeled data.
In this setup, the student model is the main network that is trained with gradient descent, while the teacher model is a separate copy whose weights are updated as an exponential moving average (EMA) of the student's weights. At each training step, the student sees one augmented version of an unlabeled image (typically the stronger augmentation) and produces a prediction. The teacher sees another augmented version of the same image (typically the weaker augmentation) and produces its own prediction, but without gradients.
A consistency loss is then computed between the student and teacher predictions on these two views of the same unlabeled sample. This encourages the student to match the teacher's more stable predictions, even under stronger augmentations. The supervised loss on labeled data and the consistency loss on unlabeled data are combined, with the consistency term weighted by a coefficient that ramps up over time.
During training, the teacher's weights are continuously updated via EMA after each batch, gradually smoothing out the student's fluctuations. At the end of training, the teacher model is used for validation and inference, leveraging its more stable representation learned from both labeled and unlabeled data.

Algorithm 1: Mean Teacher Algorithm

Data: train set $(\mathcal{X}, \mathcal{Y})$, Unlabel data$(\mathcal{Z})$
Hyper parameters: $r, \alpha,\ p1,\ p2,\ \rho1,\ \rho2, epochs$;
Create Model : $student(\theta)$, $teacher(\theta')$;
Train *teacher* for 1 epoch;
while *epochs* **do**
  while *steps* **do**
    1: Insert noise with probability as per strategies ($p1,\ p2,\ \rho1,\ \rho2$) in $\mathcal{X}$ i.e. $\mathcal{X}_\eta$;
    2: $student(\mathcal{X}_\eta) = \mathcal{Y}_\eta$;
    3: Classification cost $(C(\theta))$ =Binary Cross Entropy($\mathcal{Y}$,$\mathcal{Y}_\eta$);
    4: Again create different noise data as mentioned in step 1 i.e. $\mathcal{X}_{\eta'}$;
    5: $teacher(\mathcal{X}_{\eta'}) = \mathcal{Y}_{\eta'}$;
    6: Calculate Consistency cost $J(\theta)$=Mean Squared Error($\mathcal{Y}_\eta$,$\mathcal{Y}_{\eta'}$);
    7: Calculate Overall cost $O(\theta) = \lambda C(\theta) + (1-\lambda)J(\theta)$;
    8: Calculate *gradients*, $O(\theta)$ w.r.t θ ;
    9: Apply *gradients* to θ;
    10: Update Exponential Moving average of θ to θ' i.e. $\theta'_t = \alpha\theta'_{t-1} + (1-\alpha)\theta_t$;
  end
end

Results of 5-fold cross validation:

	precision	recall	f1-score	support
CAA	0.872 ± 0.149	0.877 ± 0.072	0.873 ± 0.088	16.2 ± 0.8
Coarse	0.696 ± 0.144	0.879 ± 0.142	0.772 ± 0.077	21.4 ± 0.98
Compact	0.842 ± 0.206	0.718 ± 0.41	0.748 ± 0.235	12.6 ± 0.98
Cored	0.728 ± 0.102	0.772 ± 0.343	0.744 ± 0.211	15.8 ± 0.8
Diffuse	0.861 ± 0.208	0.987 ± 0.054	0.916 ± 0.122	14.8 ± 0.8
NonAB	0.834 ± 0.155	0.987 ± 0.054	0.902 ± 0.099	14.6 ± 0.98
OtherAB	0.433 ± 0.778	0.133 ± 0.249	0.196 ± 0.346	6.2 ± 0.8
Subpial	0.827 ± 0.517	0.369 ± 0.267	0.505 ± 0.335	8.6 ± 0.98
UndefAB	0.872 ± 0.215	0.743 ± 0.275	0.788 ± 0.116	7.6 ± 0.98
macro avg	0.774 ± 0.168	0.718 ± 0.084	0.716 ± 0.108	117.8 ± 0.8
weighted avg	0.786 ± 0.115	0.788 ± 0.071	0.765 ± 0.089	117.8 ± 0.8

Configuration:


```
{
    semi_supervised: {
        mean_teacher_config: {
            ema_decay: 0.99
        },
        semi_supervised_config: {
            model_name: mean_teacher,
            feature_extractor_name: resnet18,
            classifier_name: linear,
            training: {
                consistency_lambda_max: 0.1,
                consistency_loss_type: cross_entropy,
                ramp_up_epochs: 100,
                ramp_up_function: linear
            }
        }
    },
    architectures: {
        classifiers_config: {
            linear: {},
        },
        feature_extractors_config: {
```

```
                resnet18: {
                        output_size: 64,
                        pretrained: True,
                        freeze_feature_extractor: True,
                        dropout_rate: 0.2,
                },
        }
    },
    general_config: {
        data: {
                unlabeled_sample_size: 2000,
                labeled_sample_size: 1000000000.0,
                preload_data: True,
                apply_transforms_on_the_fly: False,
                normalize_data: True,
                normalize_mean: [0.485, 0.456, 0.406],
                normalize_std: [0.229, 0.224, 0.225],
                use_extra_features: True,
                extra_feature_dim: 2,
                number_of_augmentations: 3
        },
        training: {
                test_size: 0.2,
                val_size: 0.1,
                batch_size: 64,
                num_epochs: 500,
                optimizer: adamw,
                learning_rate: 0.0001,
                weight_decay: 1e-05,
                early_stop: 2,
                early_stop_check_val_every_n_epoch: 5,
                cv_folds: 5
        },
        system: {
                random_seed: 44
        },
    },
}
```

To be Implemented:

1- Improve the current Semi-supervised models: -> Deadline: 15 December

Following you can find some possible ways I plan to improve the models:

1- One challenge the semi-supervised models currently face is premature early stopping, caused by the consistency loss introducing noise into the validation loss. My goal is to explore potential improvements to address this issue.

2- Additionally, the reported test metrics show high variance sometimes up to 20% which is not ideal. One possible solution is to use an ensemble of models for each component, though this would increase training time linearly.

3- Another aspect worth investigating is the class imbalance problem. While the current implementation uses stratified splits to preserve class distributions across validation and test sets, the classification still relies on the argmax method. A potential improvement is to determine class-specific classification thresholds using the validation set, which may help mitigate the effects of imbalanced classes. This is the approach used in the original thesis paper and I believe it adds a lot to the performance and is easy to implement.

4- So far, the feature extractors have been frozen during training. One possible direction is to unfreeze the last few layers of the ResNet and fine-tune them.

2- Self-supervised models: -> Deadline: 31 January

The final phase of this project involves exploring **self-supervised learning** methods. I plan to investigate two complementary directions:

1. **Trainable self-supervised models** such as autoencoders and SimCLR, where the goal is to first learn meaningful image representations and then train a classifier on top. In this approach, we explicitly train a feature extractor in a self-supervised manner and then attach a classification head for downstream plaque classification.

2. **Pretrained state-of-the-art biomedical self-supervised models**, which can serve directly as feature extractors. Here, the focus will be on evaluating existing SSL models developed for histopathology or biomedical imaging, followed by training a lightweight classification head on the extracted features. One of the state of the art models in this area is **H-optimus** which is a a 1.1B parameter vision transformer trained with self-supervised learning on an extensive proprietary dataset of billions of histology images sampled from over 1 million slides of more than 800,000 patients. I believe extracting features shouldn't be challenging considering the nice documentation they have on huggingface.

```python
from huggingface_hub import login
import torch
import timm
from torchvision import transforms

# Login to the Hugging Face hub, using your user access token that can be found h
# https://huggingface.co/settings/tokens.
login()

model = timm.create_model(
    "hf-hub:bioptimus/H-optimus-1", pretrained=True, init_values=1e-5, dynamic_im
)
model.to("cuda")
model.eval()

transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(
        mean=(0.707223, 0.578729, 0.703617),
        std=(0.211883, 0.230117, 0.177517)
    ),
])

input = torch.rand(3, 224, 224)
input = transforms.ToPILImage()(input)

# We recommend using mixed precision for faster inference.
with torch.autocast(device_type="cuda", dtype=torch.float16):
    with torch.inference_mode():
        features = model(transform(input).unsqueeze(0).to("cuda"))
```

This stage is expected to be the most time-consuming, as it requires implementing a dedicated module for self-supervised training.

These deadlines take into account the upcoming Christmas break and my exam period from 10–30 January.

3- Hyperparameter optimization: -> Deadline: 15 February

Hyperparameter selection plays a crucial role in training our models. It spans both general hyperparameters such as learning rate and model-specific ones, including dropout rates for ResNet feature extractors or confidence thresholds in semi-supervised methods like FixMatch. To systematically search for an optimal configuration, I plan to employ **Optuna**, which integrates seamlessly with the PyTorch Lightning framework and supports efficient hyperparameter optimization.

However, this process presents notable challenges. The primary bottleneck is runtime: for context, a single supervised training run with 200 epochs and early stopping requires approximately **10 minutes** on the DAIC cluster. Extending this to a 5-fold cross-validation setup multiplies the runtime to **N × 50 minutes**, where *N* is the number of hyperparameter configurations being evaluated. Given the

compute limitations and queue times on the cluster, running full cross-validation for every trial is likely infeasible.

Because of these constraints, the hyperparameter tuning phase may need to rely on **single-fold** or **reduced-fold** validation rather than full cross-validation. The proposed timeline and deadlines take these computational bottlenecks into account.