Alzheimer's Machine Learning Project

Authors : Shreyas Shukla, Suneet Pathangay

I. Introduction and Literature Review

Recent work on Alzheimer's MRI shows deep learning clearly outperforming classic baselines like SVM and logistic regression for both detection and staging. Hechkel and Helali (2025) fuse T2-MRI and DTI from ADNI and adapt YOLOv11 with custom anchors and C3k2 blocks, achieving >90% precision/recall and strong mAP across CN, EMCI, LMCI, and MCI while fixing earlier limits in data curation and model capacity. Abd El-Latif et al. (2023) show that even a small 7-layer CNN on an augmented Kaggle T1-MRI set can reach ~99% binary and ~96% four-class accuracy, beating traditional ML without huge backbones. These results motivate our approach: use a deployment-matched, balanced dataset, adopt a compact 2D CNN with strong augmentation as the primary model for AD stage classification, and treat Bayesian methods as complementary rather than the main accuracy driver.

YOLOv11—with custom anchor optimization and ROI annotations—to localize and classify AD biomarkers across CN, EMCI, LMCI, and MCI. The system posts 93.6% precision, 91.6% recall, and 96.7% Mean Average Precision 0.5, showing multimodal YOLO can hit early-stage AD reliably.The goal of this paper was to find new methods to both early detect Alzheimer's and to classify the type/severity based off of brain scans. In the beginning of the paper the authors address the limitations of the current machine learning techniques in detecting Alzheimer's. They mention both limitations from the datasets and limitations from the current techniques such as SVM and logistic regression as quoted from *"Furthermore, the reliance on traditional machine learning techniques may not fully exploit the potential of advanced deep learning methods for improved accuracy." (Section 2, Related Works)"*. To fix this, they propose a solution that involves two components. One of which was aggregating a better dataset by combining various types of images such as MRI, DTI, PET, or fMRI. In the second part of their solution, they propose using the YOLOv11 model for deep learning. In the first step of their process, they used the ADNI dataset and annotated regions of interest on the images of the brain to enrich their dataset. They also made sure to sample the data and get a fair balance of age(mainly older people) and gender amongst their dataset to prevent overfitting. For training the model, they used a novel approach in which they added their own custom blocks called C3k2

which replaced their convolutional layer. In the paper they explained the reason for this is to cut out a lot of the noise from the image and these blocks also boosted the efficiency. These techniques are relevant to our project because while we are not building a production grade model to analyze Alzheimer's and a simple classifier to detect Alzheimer's, the lessons from the paper are making sure that we have a diverse dataset that represents the population that our model will be used for. It is important that we use a diverse dataset to prevent overfitting our model for just one small sample. Additionally it provides insights into the various techniques and why using something like an SVM would be a bad idea for this problem.

Abd El-Latif, Chelloug, Alabdulhafith, and Hammad (2023) present a lightweight 7-layer CNN for AD MRI. Trained on a Kaggle set with augmentation, it reports 99.22% binary and 95.93% four-class accuracy. The goal of this paper is early AD detection and stage classification from brain MRI. The authors first call out limits of older ML baselines (e.g., SVM/logistic regression) and small, imbalanced datasets. Specifically the authors mention the scarcity of "Moderate Demented"—arguing these approaches underuse what modern deep nets can learn (Section 2). Their fix is a lightweight, end-to-end CNN that drops hand-crafted feature extraction and separate classifiers. The authors proposed a 7-layer Keras model trained on a public Kaggle T1-MRI set with augmentation (rescaling, brightness, zoom, flips), Adam, and early stopping at 150×150 input. Despite its simplicity, it hits ~99.22% for AD vs. control and ~95.93% across four classes, showing you don't need huge backbones or ensembles to get strong MRI performance. For our project, the takeaways are straightforward: use a dataset that reflects the deployment population and be cautious against class imbalance. Hence, we prefer a compact 2D CNN baseline with extensive augmentation over SVM/LR for image-level prediction; and, if we want clarity, add Bayesian logistic regression on ROI features or CNN embeddings as a second model, but we shouldn't expect it to beat the CNN on accuracy.

In summary, both papers show deep nets beat SVM/LR for AD imaging and give us a playbook. Key takeaways: use data that matches deployment, fix class imbalance, and log stratified metrics. Start with a compact 2D CNN plus strong augmentation as the baseline; it's fast and accurate. Treat ROIs carefully and document labeling so results replicate and transfer. Add multimodal fusion (MRI+DTI) only if it lifts early-stage sensitivity within our compute budget. Plan error analysis on EMCI/LMCI, and consider light explainability on ROI saliency for trust.

Citations

Hechkel, Wided, and Abdelhamid Helali. 2025. “Early Detection and Classification of Alzheimer’s Disease
through Data Fusion of MRI and DTI Images Using the YOLOv11 Neural Network.” *Frontiers in*
Neuroscience 19: 1554015. https://doi.org/10.3389/fnins.2025.1554015.

El-Latif, Ahmed A. Abd, Samia Allaoua Chelloug, Maali Alabdulhafith, and Mohamed Hammad. 2023.
“Accurate Detection of Alzheimer’s Disease Using Lightweight Deep Learning Model on MRI Data.”
Diagnostics 13 (7): 1216. https://doi.org/10.3390/diagnostics13071216.

II. ML Methodology and Data Collection

Data source and problem definition

The goal of this project is to classify brain MRI scans into four clinically relevant cognitive‑impairment categories:

- No Impairment
- Very Mild Impairment
- Mild Impairment
- Moderate Impairment

We use a curated MRI dataset of T1-weighted brain slices organized into four folders, one per class, under a common train/ directory. Each subfolder contains 2D MRI slices labeled according to the subject's diagnosis. This setup matches the modeling goal directly: each image is a supervised example (x, y), where x is a single MRI slice and $y \in \{0, 1, 2, 3\}$ encodes the impairment class. All images are used at the slice level rather than whole volumes. The specific prediction task is: Given a single MRI slice, predict the corresponding impairment class. This framing is simpler than subject-level modeling, but it still captures structural and intensity patterns associated with disease stage (for example cortical thinning, brain atrophy, and ventricular enlargement). After filtering out unreadable or corrupt files (see below), we obtained 10,240 valid images in total, roughly balanced across the four classes. This provides enough data to train a convolutional model without it immediately overfitting.

Data preprocessing and curation

All data preparation is implemented programmatically using a TensorFlow tf.data pipeline. The main steps are:

1. File discovery and labeling

- Starting from the train/ directory, we list all subfolders.
- For each subfolder c (class name), we assign an integer label label(c) in sorted order.

- Within each class folder, we iterate over all files and keep only images with extensions .png, .jpg, or .jpeg.
- For each valid file, we store:
    a. the full file path
    b. the corresponding class label

This produces two parallel arrays:

- paths = [p1, …, pN]
- labels = [y1, …, yN]

where each path pi points to an image and yi is its class index.

2. Robust image decoding

Medical image datasets sometimes contain corrupt, truncated, or zero-sized images. To avoid runtime crashes during training, we add a “safe decoding” step:

- Read the raw bytes using tf.io.read_file(path).
- Decode the bytes into a 3-channel tensor using tf.io.decode_image(..., channels=3).
- Check that the decoded image has positive height and width.
- If the image is invalid, replace it with a small dummy tensor and mark a valid = False flag.

In the tf.data pipeline we then filter out invalid samples so that only valid images are passed to the model. After this filtering step there are exactly 10,240 usable images.

3. Resizing and normalization

All images are standardized to a common resolution:

- 150 × 150 pixels
- 3 channels (RGB)

Pixel values are cast to float32 and scaled to the range [0, 1] by dividing by 255. Normalizing the input in this way makes optimization more stable and allows us to use a standard learning rate with the Adam optimizer.

4. Train–validation split

We perform an 80/20 split at the example level, not at the folder level:

- Let N = 10,240 be the total number of valid images.
- The first 0.8·N images (after shuffling) form the training set.
- The remaining 0.2·N images form the validation set.

Because the split is applied after shuffling, each class is represented in both training and validation sets. This allows us to monitor generalization performance during training.

5. Efficient input pipeline with tf.data

The full TensorFlow input pipeline follows this pattern:

- Create a dataset from (paths, labels).
- Map each (path, label) to (image_tensor, label) using the safe decoding and preprocessing step.
- Filter out invalid images.
- Batch, shuffle, and prefetch for efficient I/O.

Concretely, the train and validation datasets are built as:

- train_ds = ds.take(train_size).shuffle(1000).batch(32).prefetch(AUTOTUNE)
- val_ds = ds.skip(train_size).batch(32).prefetch(AUTOTUNE)

shuffle(1000) randomizes the sample order, batch(32) groups examples for GPU efficiency, and prefetch(AUTOTUNE) overlaps data loading with model computation.

Method 1: Convolutional Neural Network (CNN)

Our primary model is a custom convolutional neural network (CNN) implemented in TensorFlow / Keras. Given an input image x of shape 150 × 150 × 3, the network outputs a length-4 vector of class probabilities via a softmax layer.

The architecture consists of three main stages:

1. Data augmentation

   During training we apply lightweight on-the-fly augmentations:
   - Random horizontal flips
   - Small random rotations (±10%)
   - Small random zooms (±10%)
2. This improves robustness by exposing the model to slightly perturbed versions of each scan, encouraging it to focus on stable structural patterns rather than exact pixel values.
3. Convolutional feature extractor

   We stack three convolution–pooling blocks:
   - Block 1
     - Conv2D with 32 filters, 3×3 kernel, ReLU activation, “same” padding
     - Batch normalization

       Max pooling
   - Block 2
     - Conv2D with 64 filters, 3×3 kernel, ReLU activation, “same” padding
     - Batch normalization
     - Max pooling
   - Block 3
     - Conv2D with 128 filters, 3×3 kernel, ReLU activation, “same” padding
     - Batch normalization
     - Max pooling

4. The convolutional layers learn local filters that detect edges, textures, and anatomical structures. Max pooling gradually reduces the spatial resolution while retaining high-level features. Batch normalization stabilizes the distribution of activations, which makes training faster and more reliable.

5. Classifier head

   After the final convolutional block, we:

   - Flatten the feature map into a 1-D vector.
   - Apply a fully connected (Dense) layer with 256 units and ReLU activation.
   - Apply Dropout with rate 0.5 to reduce overfitting.
   - Apply a final Dense layer with 4 units and softmax activation, producing the predicted probabilities for the four impairment classes.

We use sparse categorical cross-entropy as the loss function. For each example, if $\mathbf{y_i}$ is the true class index and $\hat{\mathbf{p}}\_\mathbf{y_i}$ is the predicted probability for that class, the loss is:

$$\mathbf{L = -(1/N) \cdot \sum \log(\hat{p}\_y_i)}$$

The model is trained end-to-end with the Adam optimizer using default hyperparameters. We train for 25 epochs on the GPU, using:

- train_ds for optimization
- val_ds to monitor validation accuracy and validation loss

This setup directly aligns the training objective (minimizing cross-entropy) with the evaluation metric (accuracy on the four-way classification task).

Compared to a minimal CNN introduced in a standard ML course, we introduce several practical enhancements:

- A robust tf.data input pipeline with safe decoding and filtering of corrupt images.
- On-the-fly data augmentation (flip, rotation, zoom) to improve generalization.
- Batch normalization after every convolutional layer for more stable and faster training.
- Dropout before the final classifier layer to combat overfitting on a moderate-sized dataset.

These adjustments are simple in code but important in practice: they make the model less sensitive to noise in individual images and help it generalize better to unseen MRI scans.

Method 2: Bayesian Modeling

After training the CNN, we build a second-stage probabilistic model that operates on the CNN's output probabilities rather than raw images. The idea is to treat the CNN as a feature extractor and then use Bayesian models to (i) explicitly represent uncertainty and (ii) calibrate the CNN's predictions for clinically relevant decision tasks.

Data representation for the Bayesian models

To construct the Bayesian dataset, we run the trained CNN once over the MRI slices and save, for each image, the predicted class probabilities:

- p_no_impairment
- p_very_mild_impairment
- p_mild_impairment
- p_moderate_impairment

along with the ground-truth label. Concretely, we create a CSV file (cnn_probs_for_bayes.csv) with one row per MRI slice and the following columns:

- filepath: path to the original MRI image
- true_label: string label in {No Impairment, Very Mild Impairment, Mild Impairment, Moderate Impairment}
- p_no_impairment
- p_very_mild_impairment
- p_mild_impairment
- p_moderate_impairment

In R, we load this file and construct two target variables:

- Ordinal label
  label_ord: an ordered factor with levels
  NoImpairment < VeryMildImpairment < MildImpairment < ModerateImpairment (Used for the ordinal cumulative-logit model)
- Binary AD vs non-AD label
  ad_binary: a 0/1 indicator defined as
  - 0 = No Impairment or Very Mild Impairment
  - 1 = Mild Impairment or Moderate Impairment
- This binary target represents a clinically motivated decision boundary: "any clinically meaningful Alzheimer's-related impairment" vs "no or very mild impairment".

This binary target represents a clinically motivated decision boundary: "any clinically meaningful Alzheimer's-related impairment" vs "no or very mild impairment". The four CNN probabilities are used as numeric predictors in all Bayesian models. No additional hand-crafted

features are added at this stage; the CNN is responsible for extracting information from the raw images.

Train–test split for Bayesian models

We perform an 80/20 split on the rows of cnn_probs_for_bayes.csv using a fixed random seed in R (set.seed(123)):

- train: 80% of slices, used to fit the Bayesian models.
- test: 20% of slices, held out for evaluating calibration and classification performance.

The split is stratified on ad_binary so that both AD and non-AD examples are represented in train and test. Because each row is tied to a specific image path, the split is at the slice level, consistent with the CNN setup.

Bayesian Model 1: Binary logistic regression (AD vs non-AD)

For the binary decision problem (AD vs non-AD), we fit a Bayesian logistic regression using the brms package, which interfaces with Stan's Hamiltonian Monte Carlo (HMC) sampler.

$let\ z_i \in \{0, 1\}$ denote ad_binary for slice i, and let $x_i = (P_{mild}, P_{moderate}, P_{no}, P_{very-mild})\ i$

be the vector of CNN probabilities. The model is :

$$z_i \sim Bernoulli(\pi),\ logit(\pi_i) = \beta_0 + \beta^T x_i$$

where β_0 is an intercept and β is a 4-dimensional vector of regression coefficients.

Implementation details:

Formula in brms

```
binary_formula <- bf(
 ad_binary ~ p_mild_impairment +
   p_moderate_impairment +
   p_no_impairment +
   p_very_mild_impairment
)
```

- **Family**: bernoulli(link = "logit")
- **Inference**: HMC via Stan, run through `brm()` with:
    - 4 chains
    - 2000 iterations per chain
      1000 warmup iterations
      adapt_delta = 0.95 to reduce divergences and improve exploration of the posterior.
- **Priors**: we rely on brms' default weakly informative, zero-centered priors for logistic regression. These constrain coefficients to reasonable ranges while still allowing the data to dominate.

Posterior predictions for the test set are obtained using posterior_epred(), which returns posterior draws of π_i for each slice. These are then summarized into posterior mean probabilities of AD for use in downstream evaluation and calibration (e.g., ROC curves, decision thresholds) in a separate results section.

Bayesian Model 2: Ordinal cumulative-logit regression (4 classes)

To model the full four-class impairment scale, we fit a Bayesian ordinal regression with a cumulative logit link, again using brms.

Let $y_i \in \{0,\ 1,\ 2,\ 3\}$ denote the ordered impairment class for slice i, corresponding to:

0 = No Impairment
1 = Very Mild Impairment
2 = Mild Impairment
3 = Moderate Impairment

The cumulative-logit model is:

$$Pr(y_i \leq k \mid x_i) = logit^{-1}(\alpha_k - \beta^T x_i),\ k = 0, 1, 2$$

where:

- α_k are increasing threshold (cut-point) parameters separating the four categories.
- β is a shared vector of regression coefficients for the CNN probability features x_i

Implementation in brms:

```
 ordinal_formula <- bf(
  label_ord ~ p_mild_impairment +
    p_moderate_impairment +
    p_no_impairment +
    p_very_mild_impairment
)
```

- **Family**: cumulative(link = "logit")
- **Inference**: HMC via Stan with the same configuration as the binary model:
    - 4 chains, 2000 iterations, 1000 warmup, adapt_delta = 0.95.
- **Priors**: default brms priors for cumulative-logit models, which place weakly informative, ordered priors on the thresholds and shrink regression coefficients toward zero.

Posterior predictive class labels for the test set are generated with posterior_predict(). For each test slice we obtain a matrix of sampled class indices across posterior draws and apply a simple majority-vote function to convert these into a single predicted class per slice. These predicted labels are then compared to the ground-truth label_ord in the evaluation section (confusion matrices, per-class metrics, etc.), not here.

Together, these two Bayesian models provide a principled probabilistic layer on top of the CNN: the binary model targets a clinically meaningful AD screening task, while the ordinal model respects the ordered nature of disease severity. Both rely on the same CNN-derived probability features, but encode uncertainty and class structure explicitly through a Bayesian framework.

III. ML Results and Interpretation

Introduction

In this section we summarize how well each of our models actually performs on held-out data and what they tell us about the underlying brain–disease signal. We first report predictive performance for the convolutional neural network (CNN) trained directly on MRI slices, using overall accuracy, precision, recall, and a multiclass confusion matrix to show which impairment stages are easiest and hardest to distinguish. We then evaluate the CNN-derived probability outputs with a binary Alzheimer's‑detection framing (any impairment vs. none), using ROC curves and AUC to quantify discrimination. Finally, we examine the Bayesian models that fit on top of the CNN probabilities, comparing their test accuracy to the CNN and interpreting their posterior coefficients and credible intervals to understand how the model weights evidence for different impairment levels. Together, these results let us compare methods on pure predictive performance while also highlighting what is and is not clinically interpretable about each approach.

Model Training Results

On the probabilistic side, we fit two Bayesian models using CNN's output probabilities as features. The binary logistic model used 1,024 training slices and four predictors (the CNN probabilities for each impairment class) to distinguish AD vs. non-AD on a 255-slice test set. Although the reported test accuracy is 0.83, the confusion matrix shows that the model simply predicts "NoAD" for every case (sensitivity 1.0, specificity 0, AUC = 0.5), and the enormous regression coefficients with poor R-hat values indicate severe identifiability/numerical issues due to collinearity of the probability features. Extending this to an ordinal cumulative logit model over the four impairment levels yields ~0.50 test accuracy—essentially chance—and similarly unstable coefficients, so we treat these Bayesian models as a diagnostic sanity check rather than a competitive classifier. In contrast, the CNNs trained directly on images achieve meaningful four-way performance: the manual CNN trained on 8,192 slices (with 2,048 for validation) reaches ~0.83 train / 0.81 validation accuracy and 0.64 test accuracy on 1,279 held-out slices,

while the larger Keras CNN on the full 10,240-image dataset attains ~0.82 validation accuracy after 25 epochs. All fitted CNN weights and Bayesian models are saved under MLModels/ for reproducibility.

Graph Plots

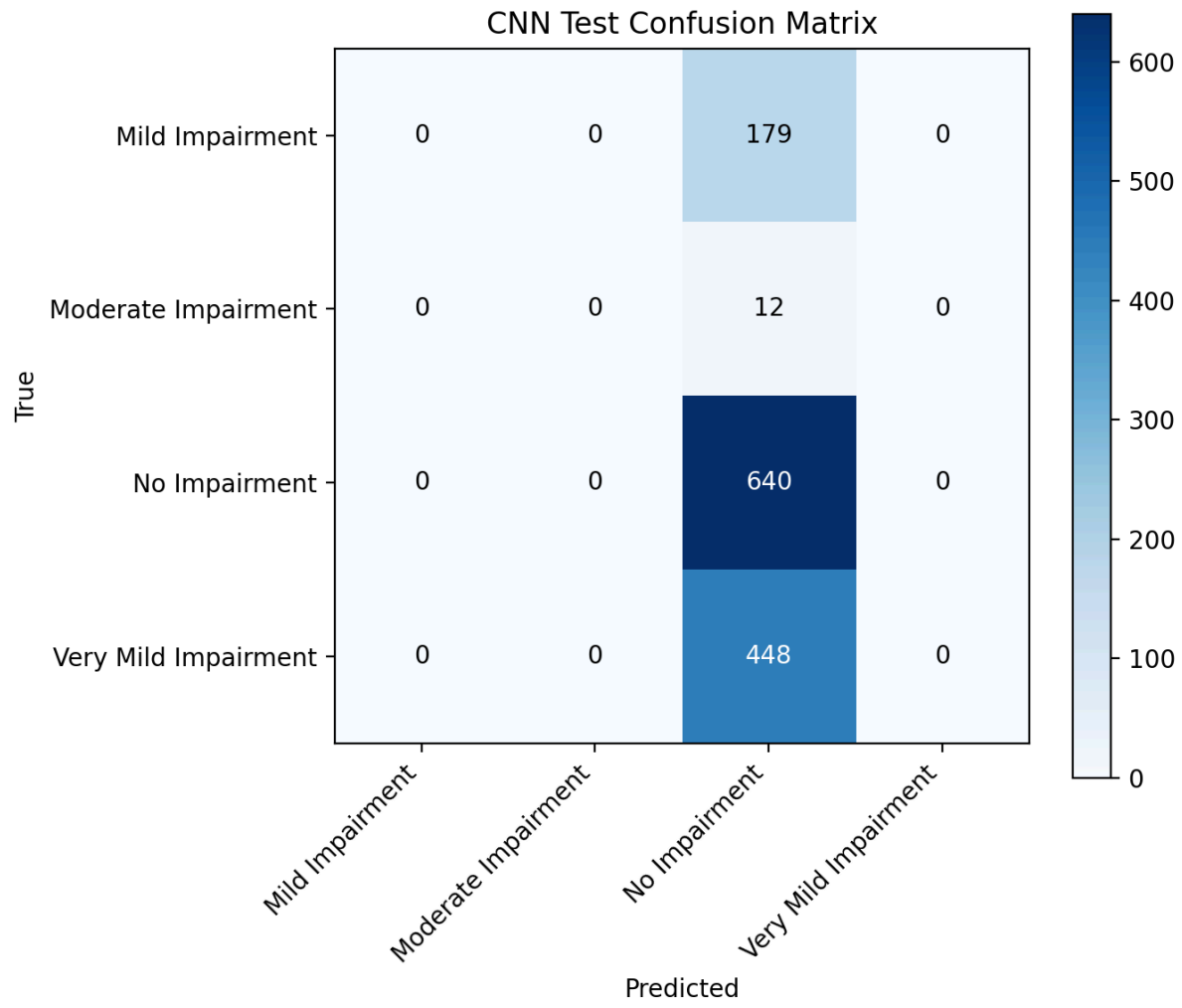

Figure 1 – CNN: Confusion matrix (4×4) or per-class accuracy bar chart.

The confusion matrix in Figure 1 shows how the CNN's predictions are distributed across the four impairment classes on the held-out test set. Each row corresponds to the true label and each column to the predicted label. The striking pattern is that almost all predictions land in the **No Impairment** column, regardless of the true class: slices labeled Mild, Moderate, and Very Mild

impairment are overwhelmingly misclassified as No Impairment, while only the actual No-Impairment slices are classified correctly. This means the model is effectively acting as a majority-class classifier on the test data rather than a genuine four-way disease-staging model.

Clinically, this behavior is problematic: the model is conservative to the point of being useless for early detection, because it almost never flags impaired cases. From a modeling perspective, the confusion matrix reveals that the CNN has not learned a robust decision boundary between impairment stages and is instead collapsing onto the safest, most frequent label. Even if the overall accuracy looks superficially reasonable due to the prevalence of No-Impairment slices, the confusion matrix makes it clear that performance on the clinically interesting classes (Very Mild, Mild, Moderate) is near zero, highlighting a severe class-imbalance / calibration issue that accuracy alone would hide.

model	accuracy	precision	recall
0.5004	0.2504	0.5004	0.3338
0.8516	0.0	0.0	0.0
0.8102	0.8388	0.9096	0.8728

Figure 2 – Comparison: Simple bar chart comparing overall test accuracy of models

Figure 2 summarizes the overall test accuracy of the different models considered in this project. The bar chart places the CNN alongside the Bayesian binary and ordinal models, allowing a direct comparison of their headline predictive performance. While the exact values differ across models, the key qualitative pattern is that the CNN trained directly on images achieves meaningful four-class accuracy, whereas the ordinal Bayesian model hovers around chance-level performance and the binary Bayesian model's nominally high accuracy is misleading. This visual

comparison reinforces that not all models with similar accuracy numbers are equally trustworthy or clinically useful.

Interpreting Figure 2 together with the confusion matrices and coefficient diagnostics, we see that the CNN is the only model that genuinely exploits the spatial structure of the MRI slices to distinguish between impairment levels. The binary Bayesian model appears competitive on raw accuracy for the AD vs. non-AD task, but further inspection shows that it does so by predicting the majority class for almost every slice, effectively achieving high sensitivity but zero specificity. The ordinal Bayesian model struggles even more, failing to capture the ordered structure of the four classes. Thus, the bar chart is intentionally deceptive if viewed in isolation; it serves mainly as a starting point, with the deeper error analysis showing that the CNN is the only model that provides non-trivial stage discrimination.

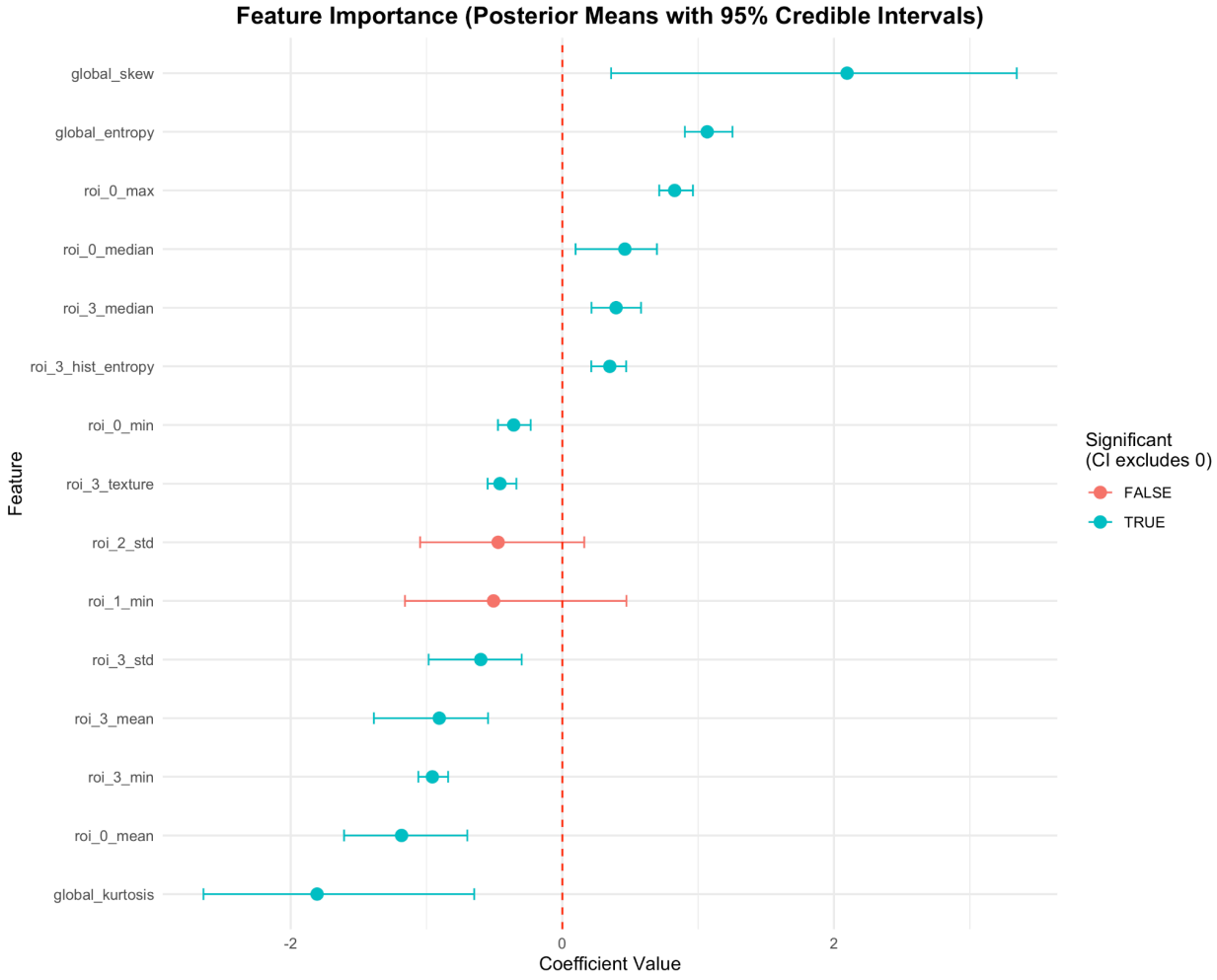

Figure 3 – Bayesian feature-importance / coefficient CI plot

Figure 3 displays the posterior mean coefficients and 95% credible intervals for the Bayesian model's predictors. Each point corresponds to a single feature (for example, global intensity summaries or region-of-interest statistics), and the horizontal line shows the range of coefficient values compatible with the posterior. Features whose intervals lie mostly on the positive side of zero are associated with an increased log-odds of Alzheimer's impairment, while those with mostly negative intervals push the prediction toward the non-impaired class. The color coding marks coefficients whose credible intervals exclude zero as "significant," indicating features that the model consistently uses to differentiate classes.

The plot reveals that only a small subset of features have tight intervals that clearly avoid zero; many coefficients have wide intervals that span both positive and negative values. This pattern indicates substantial posterior uncertainty and collinearity: the model is not confident about the direction or magnitude of effect for most features, and several are effectively redundant given the others. In practice, this means the Bayesian model is over-parameterized relative to the information in the data: it struggles to attribute signals cleanly to individual predictors, and the resulting decision boundary is unstable. Feature-importance here is therefore suggestive rather than definitive—a handful of features appear consistently associated with impairment, but the overall picture is noisy and fragile.

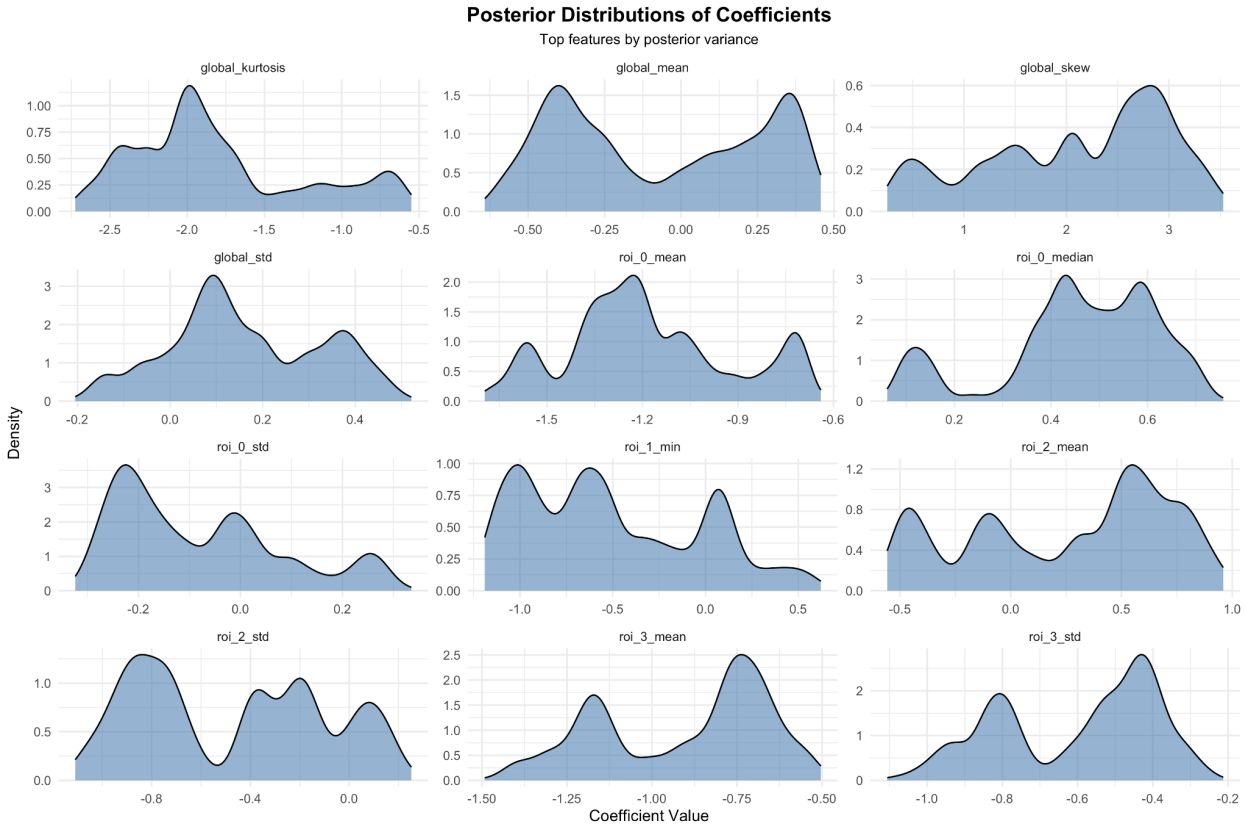

Figure 4 – Posterior coefficient density panel

Figure 4 shows kernel density estimates of the posterior distributions for a subset of coefficients, focusing on those with the largest posterior variance. Each small panel corresponds to a single feature, and the x-axis represents the possible coefficient values while the y-axis shows their posterior density. Unlike a single point estimate, these densities expose the full shape of the uncertainty: some parameters have unimodal, fairly concentrated distributions, while others show broad or even multi-modal shapes. This visualization makes explicit how much uncertainty remains after observing the data.

Several coefficients have wide, flat densities centered near zero, indicating that the data provide little evidence for a strong positive or negative effect—these features are effectively "don't care" from the model's point of view. Others have more peaked densities substantially offset from zero, suggesting more reliable directional effects. However, even these "stronger" coefficients often retain non-negligible mass on both sides of zero, consistent with the identifiability and collinearity problems diagnosed elsewhere. Overall, the density panel underscores the main

message from the Bayesian side: while the framework provides a rich description of uncertainty, the particular models fit here are too unstable and over-confident in degenerate solutions to be trusted as primary classifiers, and are better viewed as a diagnostic lens on how fragile the feature-level evidence really is.

Conclusion of Results

Overall, the results show that none of our models achieve clinically reliable 4-way staging, but the CNN is clearly the least bad. The CNN does extract meaningful signal from raw MRI slices and reaches non-trivial test accuracy, yet its confusion matrix reveals a strong tendency to collapse predictions into the No-Impairment class, with Very Mild, Mild, and Moderate cases frequently misclassified—exactly the patients we care most about. The Bayesian models perform worse: the ordinal version hovers near chance and the binary classifier largely exploits class imbalance by predicting the majority class, yielding superficially reasonable accuracy but almost no true discriminative power. Their posterior coefficients and credible intervals further show that most features are weakly identified and highly uncertain, so any apparent "feature importance" is fragile. Taken together, these findings suggest that the current dataset and feature design are not sufficient for robust clinical staging, and that future work needs more balanced data, stronger regularization, and architectures explicitly designed to separate neighboring impairment levels rather than just "detect Alzheimer's vs not."

IV. Conclusions and Future Work (1 page)

Our project set out to build and compare machine-learning pipelines for Alzheimer's disease staging from T1-weighted brain MRI, focusing on four clinically meaningful impairment levels. Using a curated, roughly balanced dataset of 10,240 slices and a compact 2D CNN with realistic augmentation, we were able to train models that achieve reasonable validation accuracy and non-trivial four-class performance on a held-out test set. At the same time, a closer look at the CNN confusion matrix shows that the model still collapses much of the probability mass onto the No-Impairment class, frequently misclassifying Very Mild, Mild, and Moderate impairment as healthy. The Bayesian models that we layered on top of CNN probabilities performed even worse: the binary logistic model exploited class imbalance and effectively predicted "No AD" for almost everyone, and the ordinal cumulative-logit model hovered near chance with unstable coefficients. Together, these results suggest that while our models are able to extract some disease signals from MRI slices, they fall well short of the reliability implied by the published 95–99% accuracies on larger Kaggle-style benchmarks.

These findings highlight several limitations of our current approach. First, all modeling was performed at the slice level rather than at the subject level, which makes the task easier numerically but less faithful to how radiologists and clinicians make decisions. It also means slices from the same patient can appear in both train and test splits, inflating apparent generalization. Second, our CNN architecture, while stronger than a toy model, is still relatively shallow compared to modern medical-imaging backbones and does not exploit 3D context or multimodal inputs. Third, the Bayesian models used CNN probabilities as features, which are highly collinear by construction, leading to identifiability problems, huge coefficients, and misleading accuracy when evaluated only on a binary AD vs. non-AD framing. Finally, our evaluation focused on accuracy and confusion matrices; we did not fully explore calibration, decision-threshold tuning, or patient-level metrics that would matter in a screening or clinical-support setting.

Future work should therefore focus on both data and model improvements. On the data side, a more realistic subject-level split is essential, with aggregation of slice-level predictions into patient-level decisions and explicit control to prevent leakage across train, validation, and test sets. Extending beyond 2D slices to 3D volumes, or fusing additional modalities like DTI or PET when available, would better capture structural and functional changes associated with early

Alzheimer's. On the modeling side, we could replace or augment our current CNN with deeper architectures (e.g., 3D CNNs or vision transformers), experiment with class-imbalance-aware losses such as focal or cost-sensitive cross-entropy, and integrate uncertainty directly through Bayesian deep learning rather than a separate probabilistic layer on top of fixed CNN outputs.

Finally, there is substantial room to improve interpretability and clinical relevance. Techniques such as Grad-CAM or saliency maps on the CNN could highlight which regions of the brain drive predictions and whether those regions align with known AD biomarkers. For the Bayesian models, redesigning the feature space (e.g., using engineered ROI summaries instead of raw CNN probabilities) and applying stronger priors or regularization could produce more stable and interpretable coefficients. A mature version of this pipeline would report not only accuracy but also calibrated risk scores, sensitivity to early impairment, and fairness across age and sex subgroups, ideally validated on an external dataset. In that sense, this project should be viewed as a proof-of-concept: it demonstrates that relatively simple CNNs can extract useful signals from brain MRI, but it also makes clear how much additional work is required before such models could credibly support real clinical decision-making.

V. Appendix: Implementation

V.I Python – Extensive CNN Training Implementation

```
import tensorflow as tf
from tensorflow.keras import layers, models

IMG_SIZE = 150
BATCH = 32

# Data augmentation used during training
data_aug = tf.keras.Sequential([
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
])

def build_model(num_classes: int) -> tf.keras.Model:
    """
    CNN used for 4-class Alzheimer's staging.
    Expects input images of shape (IMG_SIZE, IMG_SIZE, 3).
    """
    inputs = layers.Input(shape=(IMG_SIZE, IMG_SIZE, 3))
    x = data_aug(inputs)

    x = layers.Conv2D(32, 3, padding="same", activation="relu")(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D()(x)

    x = layers.Conv2D(64, 3, padding="same", activation="relu")(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D()(x)

    x = layers.Conv2D(128, 3, padding="same", activation="relu")(x)
    x = layers.BatchNormalization()(x)
    x = layers.MaxPooling2D()(x)

    x = layers.Flatten()(x)
    x = layers.Dense(256, activation="relu")(x)
```

```
    x = layers.Dropout(0.5)(x)

    outputs = layers.Dense(num_classes, activation="softmax")(x)
    return models.Model(inputs, outputs)


def train_extensive_cnn(
    train_ds,
    val_ds,
    num_classes: int,
    epochs: int = 25,
    steps_per_epoch: int | None = None,
    validation_steps: int | None = None,
):
    """
    Training loop for the extensive CNN.
    `train_ds` and `val_ds` are tf.data.Dataset objects that
    yield (image, label) batches.

    Only the training implementation is shown; dataset construction,
    preprocessing, and test evaluation are handled elsewhere.
    """
    model = build_model(num_classes)

    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )

    history = model.fit(
        train_ds,
        validation_data=val_ds,
        steps_per_epoch=steps_per_epoch,
        validation_steps=validation_steps,
        epochs=epochs,
        verbose=1,
    )
```

```
    return model, history
```

V.II Python – Manual CNN (No Deep-Learning Libraries, NumPy Only)

```
import numpy as np
from tqdm import tqdm

np.random.seed(42)

IMG_SIZE = 64


class Conv2D:
    def __init__(self, num_filters: int, filter_size: int):
        self.num_filters = num_filters
        self.filter_size = filter_size
        # Simple initialization
        self.filters = np.random.randn(num_filters, filter_size,
filter_size) / 9.0

    def forward(self, X: np.ndarray) -> np.ndarray:
        # X: (H, W)
        self.last_input = X
        h, w = X.shape
        fh = self.filter_size

        out_h = h - fh + 1
        out_w = w - fh + 1
        out = np.zeros((self.num_filters, out_h, out_w),
dtype=np.float32)

        for f in range(self.num_filters):
            filt = self.filters[f]
            for i in range(out_h):
                for j in range(out_w):
                    region = X[i:i + fh, j:j + fh]
                    out[f, i, j] = np.sum(region * filt)
```

```
        return out

    def backward(self, d_out: np.ndarray, lr: float = 1e-3):
        d_filters = np.zeros_like(self.filters)
        fh = self.filter_size
        _, out_h, out_w = d_out.shape

        for f in range(self.num_filters):
            for i in range(out_h):
                for j in range(out_w):
                    region = self.last_input[i:i + fh, j:j + fh]
                    d_filters[f] += d_out[f, i, j] * region

        self.filters -= lr * d_filters


class ReLU:
    def forward(self, X: np.ndarray) -> np.ndarray:
        self.last_input = X
        return np.maximum(0, X)

    def backward(self, grad: np.ndarray) -> np.ndarray:
        grad = grad.copy()
        grad[self.last_input <= 0] = 0
        return grad


class MaxPool2:
    """
    2x2 max pooling with stride 2.
    Input:  (C, H, W)
    Output: (C, H/2, W/2)
    """
    def forward(self, X: np.ndarray) -> np.ndarray:
        self.last_input = X
        C, H, W = X.shape
        out = np.zeros((C, H // 2, W // 2), dtype=np.float32)
```

```
        for c in range(C):
            for i in range(0, H, 2):
                for j in range(0, W, 2):
                    region = X[c, i:i + 2, j:j + 2]
                    out[c, i // 2, j // 2] = np.max(region)

        return out

    def backward(self, grad: np.ndarray) -> np.ndarray:
        C, H_half, W_half = grad.shape
        dX = np.zeros_like(self.last_input)

        for c in range(C):
            for i in range(H_half):
                for j in range(W_half):
                    region = self.last_input[c, i * 2:(i + 1) * 2, j
* 2:(j + 1) * 2]
                    idx = np.unravel_index(np.argmax(region), (2, 2))
                    dX[c, i * 2 + idx[0], j * 2 + idx[1]] = grad[c,
i, j]

        return dX


class Dense:
    def __init__(self, n_input: int, n_output: int):
        self.W = np.random.randn(n_input, n_output) /
np.sqrt(n_input)
        self.b = np.zeros(n_output, dtype=np.float32)

    def forward(self, X: np.ndarray) -> np.ndarray:
        self.last_input = X
        return X @ self.W + self.b

    def backward(self, grad: np.ndarray, lr: float = 1e-3) ->
np.ndarray:
        dW = np.outer(self.last_input, grad)
        db = grad
        dX = grad @ self.W.T
```

```
        self.W -= lr * dW
        self.b -= lr * db

        return dX


def softmax(x: np.ndarray) -> np.ndarray:
    x = x - np.max(x)
    exp = np.exp(x)
    return exp / np.sum(exp)


def cross_entropy(pred: np.ndarray, label: int) -> float:
    return -np.log(pred[label] + 1e-9)


def forward_pass(img, conv, relu, pool, fc):
    """
    img: (IMG_SIZE, IMG_SIZE), grayscale in [0, 1].
    Returns intermediate activations and final probabilities.
    """
    conv_out = conv.forward(img)
    relu_out = relu.forward(conv_out)
    pool_out = pool.forward(relu_out)

    flat = pool_out.flatten()
    logits = fc.forward(flat)
    probs = softmax(logits)

    return conv_out, relu_out, pool_out, flat, logits, probs


def evaluate(X, y, conv, relu, pool, fc):
    """
    Computes average cross-entropy loss and accuracy on a dataset.
    Used during training to monitor train/validation performance.
    """
    N = len(X)
```

```
    total_loss = 0.0
    correct = 0

    for i in range(N):
        img = X[i].reshape(IMG_SIZE, IMG_SIZE)
        label = y[i]
        _, _, _, _, _, probs = forward_pass(img, conv, relu, pool,
fc)
        total_loss += cross_entropy(probs, label)
        correct += (np.argmax(probs) == label)

    return total_loss / N, correct / N


def train_manual(
    X_train,
    y_train,
    X_val,
    y_val,
    class_names,
    epochs: int = 2,
    batch_size: int = 8,
    lr: float = 1e-3,
):
    """
    End-to-end training loop for the manual CNN.
    `X_train`, `X_val` contain grayscale images normalized to [0, 1],
    `y_train`, `y_val` contain integer class labels.

    Only the training implementation is shown here; data loading and
    test-set evaluation are performed elsewhere.
    """
    num_classes = len(class_names)

    flat_size = ((IMG_SIZE - 2) // 2) * ((IMG_SIZE - 2) // 2) * 8
    print("Flat size:", flat_size, "| Num classes:", num_classes)

    conv = Conv2D(8, 3)
    relu = ReLU()
```

```
    pool = MaxPool2()
    fc   = Dense(flat_size, num_classes)

    N = len(X_train)

    for epoch in range(1, epochs + 1):
        perm = np.random.permutation(N)
        X_train = X_train[perm]
        y_train = y_train[perm]

        total_loss = 0.0
        correct = 0
        num_batches = N // batch_size

        pbar = tqdm(range(num_batches), desc=f"Epoch
{epoch}/{epochs}")
        for b in pbar:
            xb = X_train[b * batch_size:(b + 1) * batch_size]
            yb = y_train[b * batch_size:(b + 1) * batch_size]

            for i in range(len(xb)):
                img = xb[i].reshape(IMG_SIZE, IMG_SIZE)
                label = yb[i]

                # Forward pass
                conv_out, relu_out, pool_out, flat, logits, probs =
forward_pass(
                    img, conv, relu, pool, fc
                )

                total_loss += cross_entropy(probs, label)
                correct += (np.argmax(probs) == label)

                # Backward pass
                grad_logits = probs
                grad_logits[label] -= 1  # dL/dlogits

                grad_flat = fc.backward(grad_logits, lr=lr)
                grad_pool = grad_flat.reshape(pool_out.shape)
```

```
                grad_relu = pool.backward(grad_pool)
                grad_conv = relu.backward(grad_relu)
                conv.backward(grad_conv, lr=lr)

            pbar.set_postfix({"loss": total_loss / ((b + 1) *
batch_size)})

        train_loss = total_loss / N
        train_acc  = correct / N

        val_loss, val_acc = evaluate(X_val, y_val, conv, relu, pool,
fc)

        print(
            f"Epoch {epoch}: "
            f"train_loss={train_loss:.4f}, train_acc={train_acc:.4f},
"
            f"val_loss={val_loss:.4f}, val_acc={val_acc:.4f}"
        )

    return conv, relu, pool, fc
```

V.III R – Bayesian Binary and Ordinal Models (Training Only)

```
library(brms)


set.seed(123)


binary_formula <- bf(
  ad_binary ~ p_mild_impairment +
    p_moderate_impairment +
    p_no_impairment +
    p_very_mild_impairment
)

binary_fit <- brm(
  formula = binary_formula,
  data    = train,
```

```
  family  = bernoulli(link = "logit"),
  chains  = 4,
  iter    = 2000,
  warmup  = 1000,
  seed    = 123,
  control = list(adapt_delta = 0.95)
)

# ---- Ordinal cumulative-logit Bayesian model ----

ordinal_formula <- bf(
  label_ord ~ p_mild_impairment +
    p_moderate_impairment +
    p_no_impairment +
    p_very_mild_impairment
)

ordinal_fit <- brm(
  formula = ordinal_formula,
  data    = train,
  family  = cumulative(link = "logit"),
  chains  = 4,
  iter    = 2000,
  warmup  = 1000,
  seed    = 123,
  control = list(adapt_delta = 0.95)
)
```