ML 2 Assignments

Daniel Rodinger

15. February 2025

1 Assignment 4

1.1 CNN

Convolution in Image Processing

In image processing, **convolution** is a fundamental operation used to extract spatial features from images. It applies a small filter (or kernel) to an input image, computing a weighted sum of pixel values. The output is called a **feature map**. Mathematically, the **discrete 2D convolution** is defined as:

$$C(i,j) = (I * F)(i,j) = \sum_m \sum_n I(m,n)F(i-m, j-n) \tag{1}$$

where:

- $I(m,n)$ is the input image (pixel intensities),
- $F(i-m, j-n)$ is the filter (kernel),
- $C(i,j)$ is the resulting feature map.

Cross-Correlation in CNNs

In **Convolutional Neural Networks (CNNs)**, the operation used is actually **cross-correlation**. In cross-correlation, the filter is **not flipped**, meaning it directly slides over the image without reversing its structure:

$$C(i,j) = \sum_m \sum_n I(i+m, j+n)F(m,n) \tag{2}$$

Since CNNs **learn the filters automatically**, flipping is unnecessary, and cross-correlation is computationally simpler while achieving the same effect.

Effect of Convolution on Image Size

Applying a 3×3 filter to a 5×5 input without padding reduces the output size:

$$\text{Output Size} = \text{Input Size} - \text{Filter Size} + 1$$

$$\text{Output Size} = 5 - 3 + 1 = 3$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix} * \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \rightarrow \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

Padding

Padding prevents size reduction by adding extra border pixels.

For a 3×3 filter, the required padding is:

$$P = \frac{\text{Filter Size} - 1}{2} = \frac{3-1}{2} = 1$$

With padding, the input becomes:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 & 9 & 10 & 0 \\ 0 & 11 & 12 & 13 & 14 & 15 & 0 \\ 0 & 16 & 17 & 18 & 19 & 20 & 0 \\ 0 & 21 & 22 & 23 & 24 & 25 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

After convolution, the output remains 5×5 instead of 3×3.

Stride

Stride determines how much the filter moves at each step.

$$\text{Output Size} = \frac{\text{Input Size} - \text{Filter Size} + 2P}{S} + 1$$

For a 5×5 input and 3×3 filter:

- Stride $S = 1 \rightarrow$ Output 3×3
- Stride $S = 2 \rightarrow$ Output 2×2

Larger strides lead to faster computations but lower resolution.

$$S = 1 : \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

$$S = 2 : \begin{bmatrix} x_{11} & x_{13} \\ x_{31} & x_{33} \end{bmatrix}$$

Key Layers in CNNs

Pooling Layer

Pooling reduces the spatial dimensions of feature maps, making the model more computationally efficient and invariant to small translations.

Max Pooling

- A 2×2 filter with stride 2 takes the **maximum** value in each region.
- Reduces size while preserving important features.

$$\text{Output Size} = \frac{\text{Input Size} - \text{Pool Size}}{\text{Stride}} + 1$$

Activation Functions

ReLU (Rectified Linear Unit)

$$f(x) = \max(0, x)$$

Why is the derivative important?

- The derivative determines how the network updates weights during backpropagation.
- If the derivative is too small, learning slows down.
- Prevent Vanishing Gradient.

Fully Connected Layer and Dropout

After feature extraction, the final layers are **fully connected layers**.

- Each neuron is connected to every neuron in the previous layer.
- Maps extracted features to final class predictions.

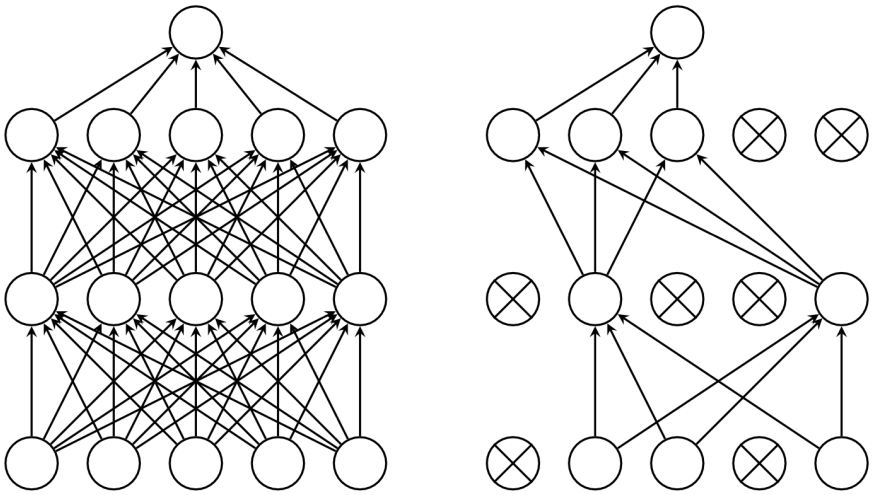


Figure 1: Dropout Regularization: During training, dropout randomly deactivates neurons to improve generalization and reduce overfitting.

Filters

2 Assignment 5 RNN

Recurrent Neural Networks (RNNs) are specialized neural architectures for handling **sequential data**. Unlike standard feedforward networks, RNNs maintain a hidden state h_t that captures information from previous time steps:

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$
$$y_t = W_{hy}h_t + b_y$$

Key Points:

- Weights are shared across all time steps.
- Can handle variable-length sequences.
- Useful for time series, language modeling, and more.
- Still trained via backpropagation, specifically Backpropagation Through Time (BPTT).

$$E = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}; \; S = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}; \; B = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

(a) Original (b) Kanten (Edges) (c) Schärfen (Sharpen) (d) Gauß-Filter (Blur)

Figure 2: Effects of different convolution filters applied to an image. (a) Original image, (b) Edge detection E, (c) Sharpening using the sharpening filter S, (d) Gaussian blur using the Gaussian filter B. Each filter enhances specific features by convolving the image with the corresponding kernel.

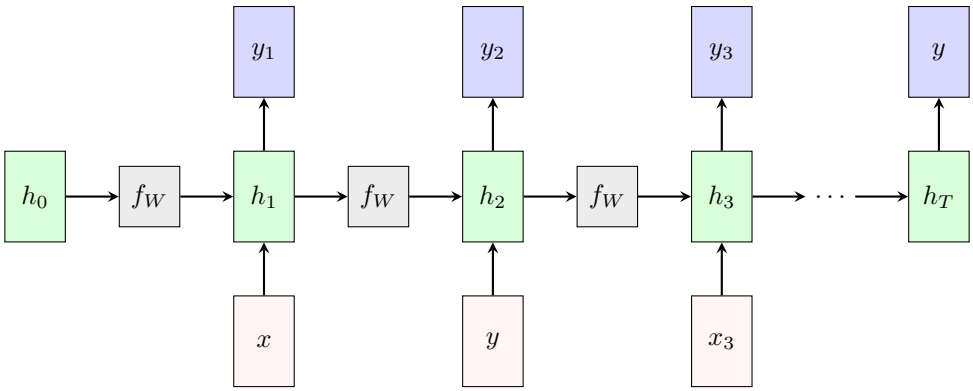


Figure 3: This diagram illustrates a Recurrent Neural Network (RNN) unrolled over time. Hidden states ($h_0 ... h_t$) maintain information between steps, while the same weight function (f_W) is applied. The network processes sequential inputs (x_1, x_2, x_3) while preserving context through hidden states.

Challenges of RNNs

Vanishing and Exploding Gradients

- **Vanishing Gradient:** Repeated multiplication by small numbers leads gradients to shrink to zero.
- **Exploding Gradient:** Repeated multiplication by large weights causes gradients to blow up.

Backpropagation Through Time (BPTT)

1. Perform forward pass through entire sequence
2. Calculate loss at each time step: $L_t(\hat{y}_t, y_t)$
3. Compute total loss: $L = \sum_{t=1}^{T} L_t$
4. Backpropagate gradients through time
5. Update weights: $W \leftarrow W - \eta \frac{\partial L}{\partial W}$

Transformers

Transformers overcome the sequential processing limitations of RNNs:

- Use **self-attention** to capture dependencies between all tokens in a sequence *in parallel*.
- **No recurrence:** enables full **parallelization during training**.
- Efficient at modeling both **local and global** contextual relationships.

Self-Attention Mechanism (Pseudo-Code)

```
for each token x_i in input sequence:
    for each token x_j in input sequence:
        attention_score = softmax((x_i · x_j) / sqrt(d_k))
    x_i' = sum over j of attention_score * V_j
```

Note: Each token learns to attend to all others via learned attention weights, improving contextual understanding.

3 Assignment 5: Logistic Regression

Logistic regression is a classification algorithm that predicts the probability of an observation belonging to a particular class.

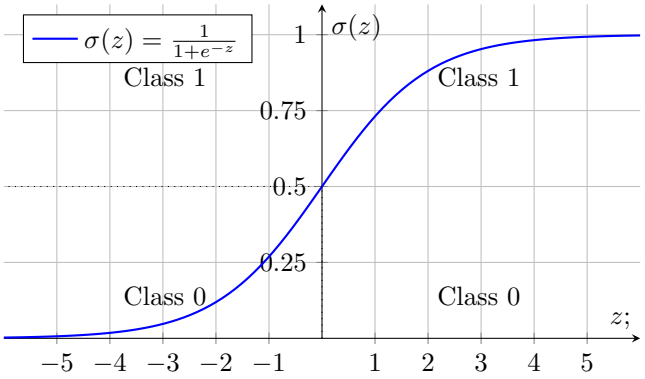


Figure 4: The sigmoid function maps the linear combination of features to a probability between 0 and 1.

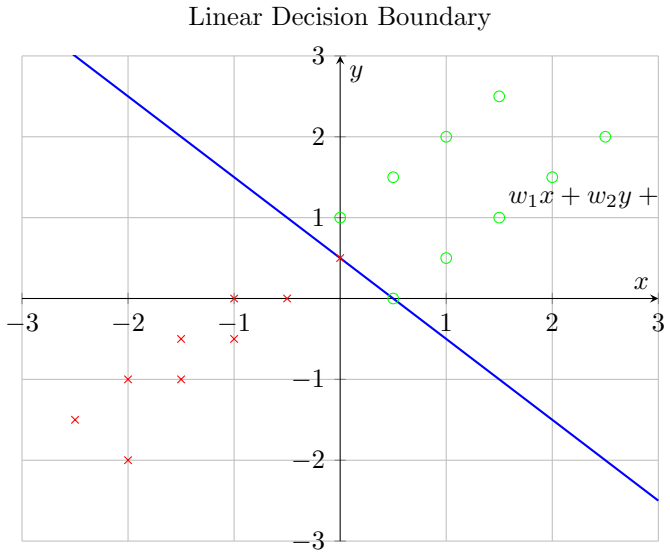


Figure 5: In logistic regression, the decision boundary is a hyperplane that separates the classes.

This plot illustrates the linear boundary (blue line) created in a 2D feature space. The boundary is defined by the equation $w_1x + w_2y + b = 0$, where w_1 and w_2 are feature weights. Green circles represent the positive class while red x marks represent the negative class. The model classifies points on one side of the boundary as class 1 (probability > 0.5) and points on the other side as class 0 (probability < 0.5). Logistic regression can effectively separate classes when they are linearly separable.

The formula $w_1x + w_2y + b = 0$ represents the decision boundary in logistic regression:

- w_1 and w_2 are weights (or coefficients) learned during training that determine the slope and orientation of the boundary
- x and y are the input features (the two dimensions in our 2D space)
- b is the bias term (or intercept) that shifts the boundary away from the origin

In vector form, this can be written as $\mathbf{w}^\top \mathbf{x} + b = 0$, where $\mathbf{w} = [w_1, w_2]$ and $\mathbf{x} = [x, y]$.
The model predicts:

- Class 1 when $w_1x + w_2y + b > 0$ (sigmoid output > 0.5)

- Class 0 when $w_1x + w_2y + b < 0$ (sigmoid output < 0.5)

The weights determine the importance of each feature in making the prediction. Larger absolute values of weights indicate higher importance of the corresponding features.

Log-Odds Interpretation

Logistic regression models the **log-odds** (logit) of the probability of the positive class as a linear function of the input features:

$$\log\left(\frac{P(y=1 \mid \mathbf{x})}{1-P(y=1 \mid \mathbf{x})}\right) = \mathbf{w}^\top \mathbf{x} + b$$

- The left-hand side is the **logarithm of the odds**, the log of the ratio of the probability of class 1 to class 0.
- The right-hand side is a **linear combination** of the input features, just like in linear regression.
- This means each feature contributes additively to the **log-odds** of the prediction.
- A positive weight increases the odds of class 1; a negative weight decreases them.
- The final probability is computed using the **sigmoid** function to map log-odds into the range $[0, 1]$.

A linear model can produce any real number, but probabilities must stay between 0 and 1. Logistic regression solves this by modeling the log-odds, which can take any real value.
Log-odds perspective makes logistic regression highly interpretable in fields where understanding the effect of each feature in probabilistic terms is essential.

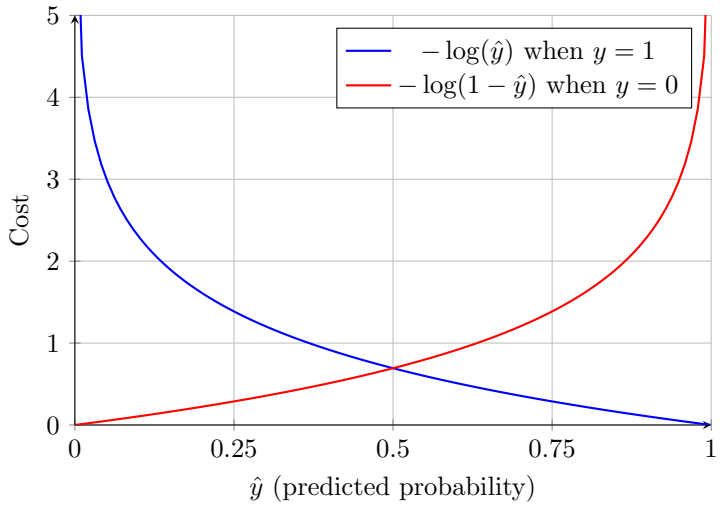


Figure 6: The cross-entropy loss function penalizes incorrect predictions more heavily as they move away from the true label.

This figure illustrates the cross-entropy loss function used in logistic regression. The graph shows two components of the loss function:

- Blue curve: $-\log(\hat{y})$ used when the true label $y = 1$
- Red curve: $-\log(1 - \hat{y})$ used when the true label $y = 0$

Key observations:

- The loss approaches infinity as predictions get further from the true label
- For $y = 1$, the penalty increases exponentially as $\hat{y}$ approaches 0
- For $y = 0$, the penalty increases exponentially as $\hat{y}$ approaches 1
- The loss is minimal when predictions match the true labels ($\hat{y} \approx 1$ when $y = 1$, and $\hat{y} \approx 0$ when $y = 0$)

This asymmetric penalty structure encourages the model to make confident and accurate predictions by severely penalizing high-confidence incorrect predictions, which makes it particularly suitable for classification tasks.

4 Ensemble Learning Methods

Ensemble learning combines multiple models to improve prediction performance beyond what any single model could achieve.

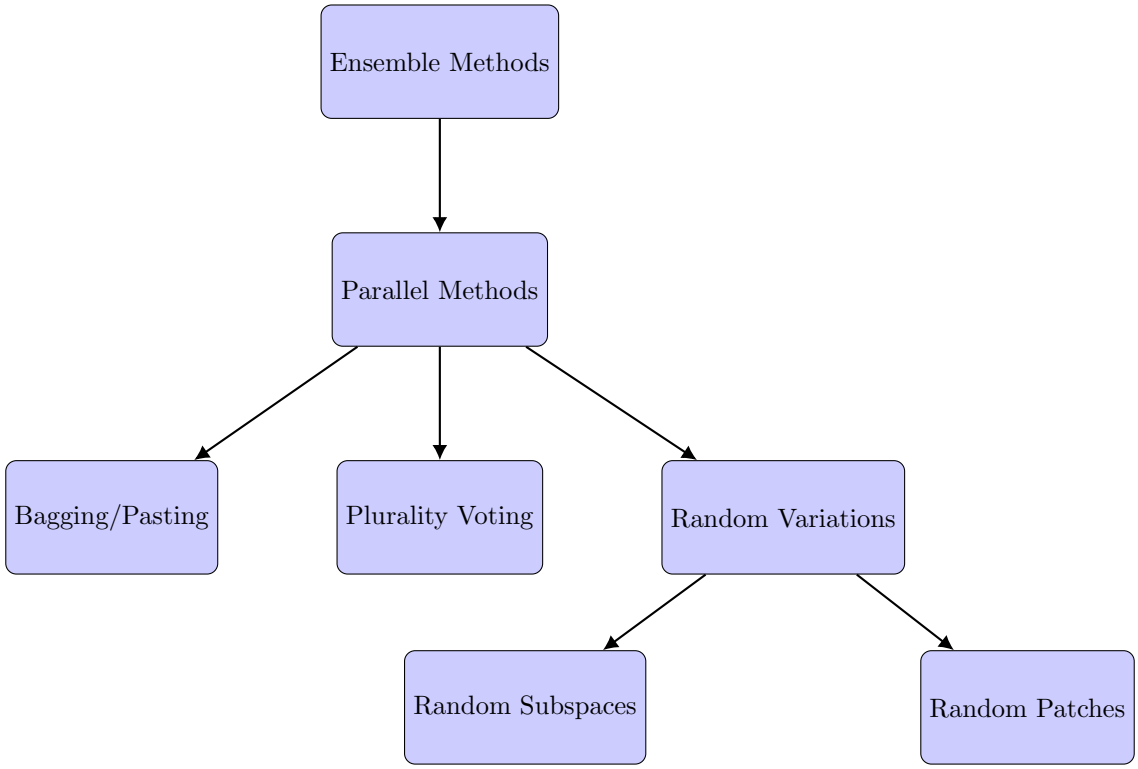


Figure 7: Taxonomy of parallel ensemble methods

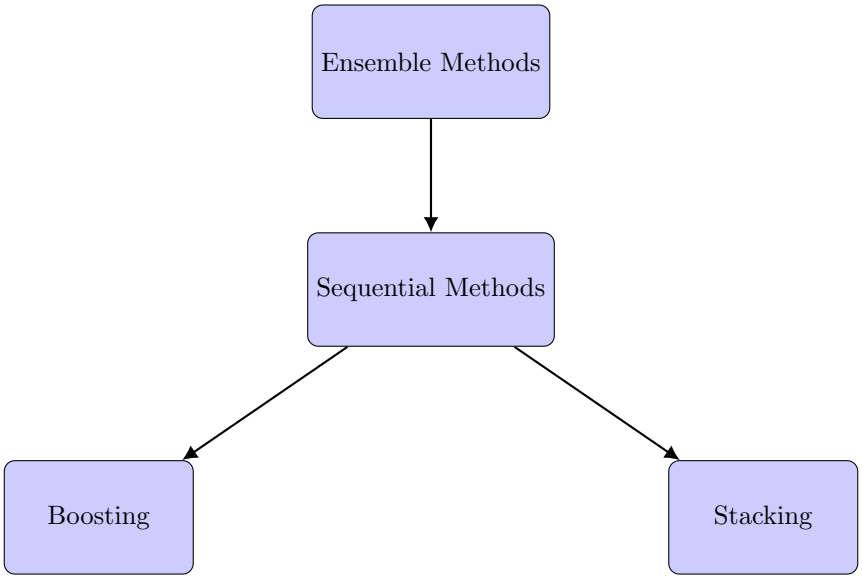


Figure 8: Taxonomy of sequential ensemble methods

Method	Approach	Key Characteristics
Plurality Voting	Models independently predict; majority wins	Simple; effective when errors uncorrelated
Bagging	Bootstrap sampling with replacement	Reduces variance; Random Forest is an example
Pasting	Sampling without replacement	Similar to bagging but without duplicates
Random Subspaces	Full dataset with random feature subsets	Effective for high-dimensional data
Random Patches	Bootstrap samples + random feature subsets	Combines benefits of bagging and subspaces
Stacking	Meta-learner combines model predictions	Learns optimal way to weight model outputs
Boosting	Sequential training focusing on errors	Reduces bias; AdaBoost and XGBoost examples

Table 1: Comparison of ensemble methods

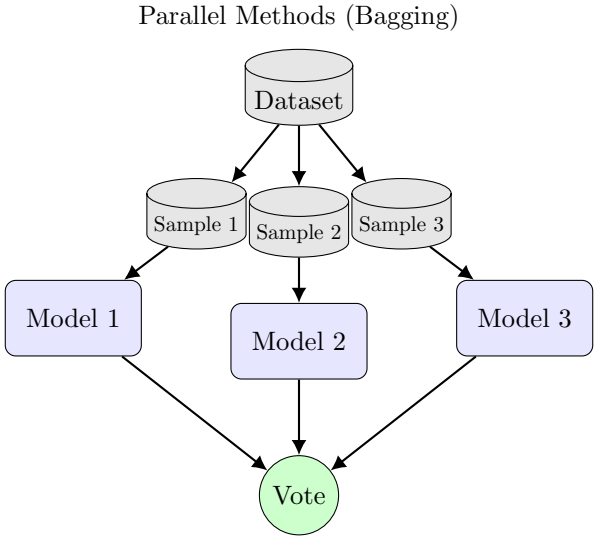


Figure 9: Parallel ensemble method (Bagging)

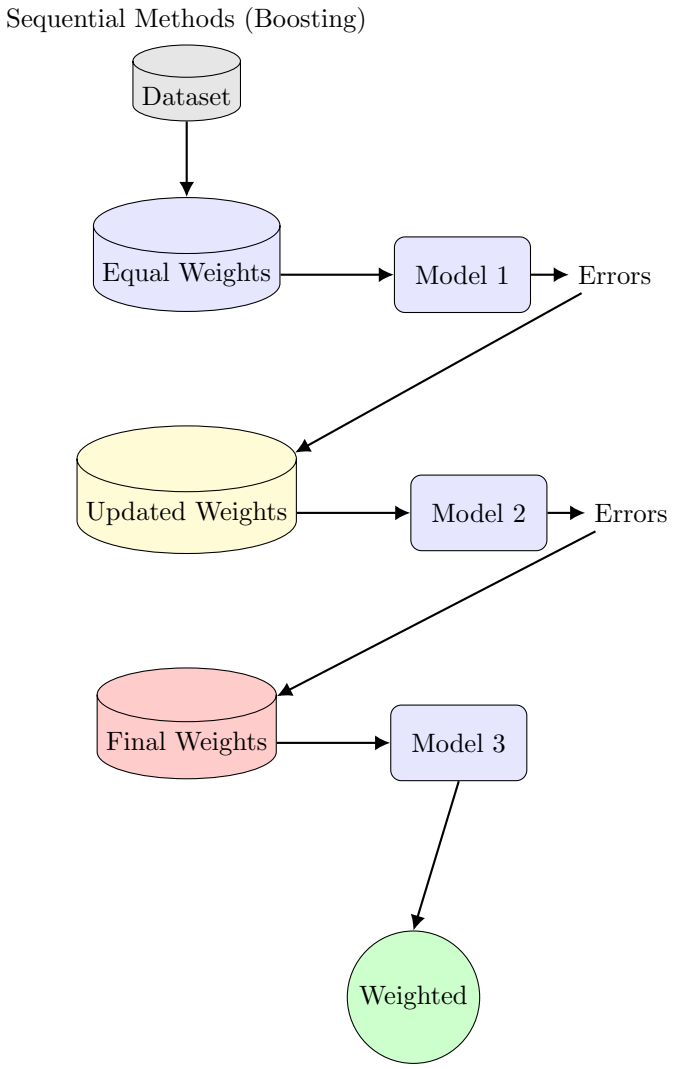


Figure 10: Sequential ensemble method (Boosting)

Key Differences:

- **Parallel Methods** (Bagging, Random Subspaces, Plurality): Models trained independently, primarily reduce variance, can be parallelized.
- **Sequential Methods** (Boosting, Stacking): Models depend on previous models, can reduce both bias and variance, must be trained in sequence.

5 Random Forest

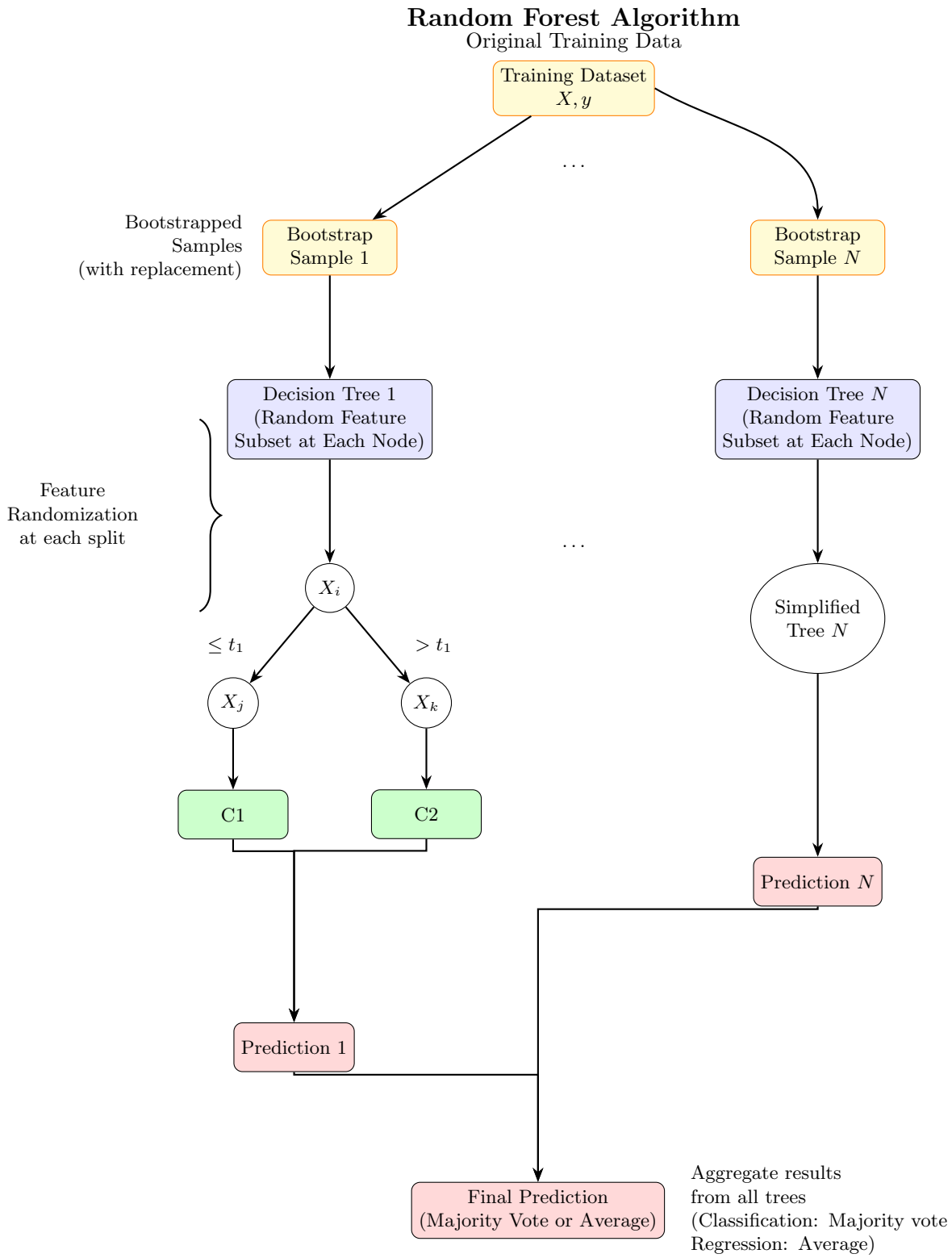


Figure 11: Visualization of Random Forest algorithm showing bootstrap sampling, feature randomization at each split, individual decision trees, and final aggregation of predictions

Table 2: Comparison of Ensemble Methods

Feature	**Bagging**	**Random Forest**	**Boosting**	**Stacking**
Base Model Training	Independent parallel training	Independent parallel training with feature randomization	Sequential training based on previous errors	Independent training of base models
Sampling Strategy	Bootstrap samples with replacement	Bootstrap samples with replacement	Weighted sampling focusing on difficult examples	Usually uses full dataset for all models
Feature Selection	Uses all features	Random subset of features at each split	Uses all features	Uses all features
Model Diversity	Through different data subsets	Through different data subsets AND feature subsets	Through sequential error correction	Through different algorithm types
Aggregation Method	Average (regression) or majority vote (classification)	Average (regression) or majority vote (classification)	Weighted sum based on performance	Meta-learner trained on base models' outputs
Main Goal	Reduce variance	Reduce variance and correlation between predictors	Reduce bias	Improve overall prediction performance
Typical Base Models	Decision trees	Decision trees	Shallow decision trees (stumps)	Various algorithms
Parallel Computing	Highly parallelizable	Highly parallelizable	Sequential (less parallelizable)	Partially parallelizable
Sensitivity to Noisy Data/Outliers	Moderately robust	Highly robust	Sensitive (can overfit to noise)	Depends on base models and meta-learner
Interpretability	Low	Low	Very low	Very low
Computational Cost	Moderate	Moderate to high	High	Very high
Hyperparameter Sensitivity	Low	Moderate	High	Very high
Examples	Bagged decision trees	Random Forest	AdaBoost, Gradient Boosting	Stacked generalization

6 Assignment 9

6.1 Ada Boost

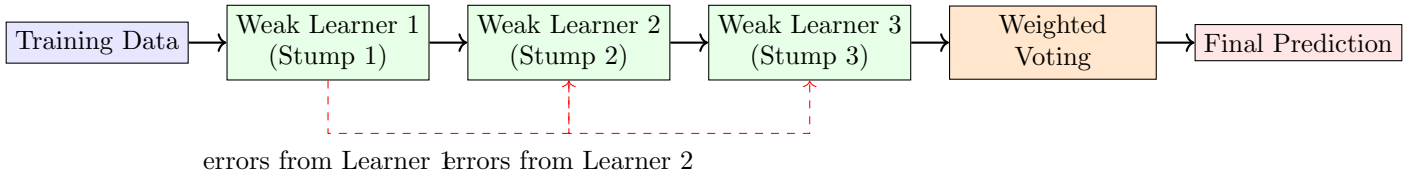


Figure 12: Illustration of the AdaBoost training process: Sequentially training weak learners focused on previous errors and combining them via weighted voting.

6.2 Gradient Boosting

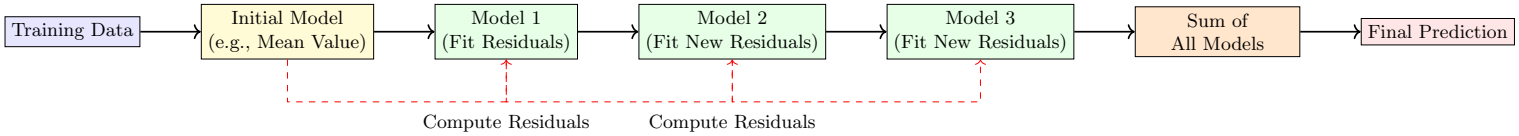


Figure 13: Gradient Boosting sequentially fits models to correct the residual errors made by previous models. The final prediction is the sum of all models.

Assignment 10

DBSCAN is a density-based clustering algorithm. It groups together points that are closely packed (with many nearby neighbours) and marks points that lie alone in low-density regions as outliers.

Instead of a visual representation, I provide the following **pseudo-code**, which more clearly expresses the step-by-step logic of the algorithm. This format is chosen because DBSCAN is fundamentally procedural and for me difficult to summarize effectively in a static diagram.

Key Parameters:

- `eps` – radius around a point to consider as neighbourhood
- `minPts` – minimum number of points in a neighbourhood to be considered a core point

Pseudo-Code:

```
DBSCAN(X, eps, minPts):
    label = 0
    for each point x in X:
        if x is not visited:
            mark x as visited
            neighbours = regionQuery(x, eps)
            if len(neighbours) < minPts:
                mark x as noise
            else:
                label += 1
                expandCluster(x, neighbours, label, eps, minPts)

expandCluster(x, neighbours, label, eps, minPts):
    assign label to x
    for each point n in neighbours:
        if n is not visited:
            mark n as visited
            n_neighbours = regionQuery(n, eps)
            if len(n_neighbours) >= minPts:
                neighbours += n_neighbours
        if n not yet assigned to any cluster:
            assign label to n

regionQuery(x, eps):
    return all points in X within distance eps of x
```

Summary: DBSCAN starts from unvisited points and uses density (based on `eps` and `minPts`) to determine whether a point is a core point. It then expands clusters by recursively visiting all density-reachable neighbours. Points with insufficient neighbours are marked as noise. This algorithm effectively detects arbitrarily shaped clusters and handles outliers naturally.

This simplified version shows the essence of DBSCAN without optimization details. It highlights how clusters are formed by recursively expanding dense neighbourhoods, a behavior that is best conveyed through logic flow rather than geometry.

7 Support Vector Machines

7.1 Introduction

The fundamental idea behind SVMs is to find an optimal hyperplane that separates different classes in the feature space while maximizing the margin between them.

7.2 The Optimal Hyperplane

Given a linearly separable dataset, there are infinitely many hyperplanes that can separate the two classes. However, SVM selects the hyperplane that maximizes the margin, the distance between the hyperplane and the closest data points from each class.

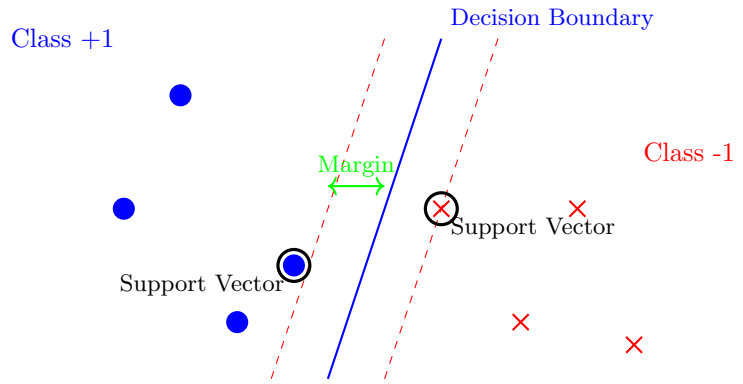


Figure 14: SVM finds the hyperplane that maximizes the margin between classes

7.2.1 Support Vectors

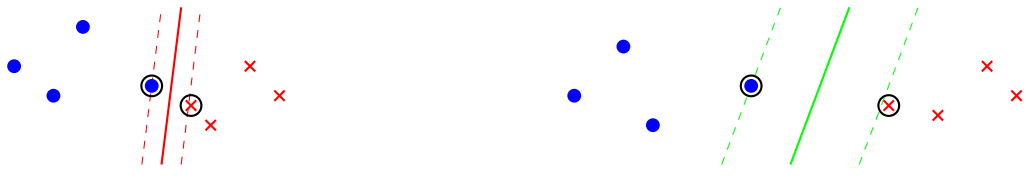
Support vectors are the data points that lie closest to the decision boundary. These points are critical because:

- They define the position and orientation of the optimal hyperplane
- Removing any support vector would change the decision boundary
- Only support vectors matter for the final classifier, other points can be removed without affecting the model

7.3 Advantages of Maximum Margin

The maximum margin approach provides several benefits:

1. **Generalization**: A larger margin typically leads to better performance on unseen data
2. **Uniqueness**: The optimal hyperplane is unique for linearly separable data
3. **Robustness**: The classifier is less sensitive to small perturbations in the training data



Small Margin
(Poor Generalisation)

Large Margin
(Better Generalisation)

Figure 15: Comparison of margin sizes and their impact on generalization

7.4 Key Properties

- **Sparsity**: Only support vectors contribute to the final decision function
- **Convex Optimization**: The SVM optimization problem has a unique global optimum

- **Geometric Interpretation**: Clear geometric meaning as margin maximization
- **Extensibility**: Can be extended to non-linear cases using kernel methods