Comparative Analysis of Machine Learning Models for Thyroid Cancer Recurrence Prediction

Anay Aggarwal Ekam Kaur Susie Lu

2024-06-20

This study assesses the efficacy of machine learning algorithms—Artificial Neural Network, K-Nearest Neighbors, Support Vector Machine, Logistic Regression, Extreme Gradient Boosting, and Random Forest—in predicting recurrence in Differentiated Thyroid Cancer (DTC). Using a comprehensive dataset from the UCI Machine Learning Repository, we conduct a comparative analysis of these algorithms based on key performance metrics. Our investigation seeks to identify the best model for predicting DTC recurrence, aiming to advance personalized treatment strategies and improve clinical outcomes in thyroid cancer care.

1 Introduction

Thyroid cancer is among the most prevalent endocrine malignancies worldwide, with differentiated thyroid cancer (DTC) representing the majority of cases [1]. The management and prognosis of thyroid cancer critically depend on the timely and accurate prediction of cancer recurrence. Traditional approaches, primarily based on clinical and pathological parameters, often lack the precision required for personalized treatment planning [2]. With advancements in machine learning (ML) techniques, there's a burgeoning interest in leveraging these analytical tools to enhance the predictive accuracy regarding cancer recurrence, thereby improving the efficacy of therapeutic interventions and follow-up strategies [3].

This paper explores the application of six distinct machine learning algorithms—Artificial Neural Network (ANN), Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Logistic Regression (LR), and the ensemble learning methods – Random Forest (RF) and Extreme Gradient Boosting (XGBoost) — to predict cancer recurrence in patients with differentiated thyroid cancer. Utilizing the differentiated thyroid cancer recurrence dataset from the UCI Machine Learning Repository [4], this study aims to compare the predictive performance of

these algorithms in a clinical setting. The dataset comprises patient demographic information, clinical features, and pathological details, providing a solid foundation for predictive modeling.

ANNs, celebrated for their capability to model complex nonlinear relationships through interconnected processing elements, offer powerful tools for medical prediction tasks [5]. SVMs, recognized for their effectiveness in high-dimensional spaces and versatility in handling both linear and nonlinear data, present a robust methodology for classification challenges [6]. KNN, a straightforward yet efficient algorithm based on feature space proximity, offers an intuitive approach to classification by leveraging the similarity between cases [7]. LR, known for its simplicity and efficient training, offers a intuitive way for determining the decision boundary between linearly separable data points, and directly outputs the predicted probabilities for each class [8]. RF, an ensemble learning method that uses bagging on classification trees, proves to be efficient and accurate in handling high-dimensional and non-linear data [9]. XGBoost, an ensemble learning method that uses boosting (meaning that the algorithm sequentially generates new trees based on previous training output), is very fast and robust for a variety of tasks [10].

By applying all six machine learning approaches to the differentiated thyroid cancer recurrence dataset, this study seeks to identify the most effective model for predicting cancer recurrence, thus contributing to the ongoing efforts to enhance patient outcomes in thyroid cancer management.

Through a rigorous evaluation of model performance based on accuracy, precision, recall, and specificity, this paper aims to shed light on the strengths and limitations of each algorithm in the context of thyroid cancer recurrence prediction. Furthermore, the study discusses the implications of these findings for clinical practice, emphasizing the potential of machine learning to revolutionize cancer care by enabling more accurate and personalized risk assessments [11]

2 Notation

Suppose we have a quantitative response Y and p different predictors $X_1, X_2, \dots, X_p$. We assume that there is some relationship between Y and $X = (X_1, X_2, \dots, X_p)$, which can be written in the general form

$$Y = f(X) + \epsilon.$$

Here f is some fixed but unknown function of $X_1, X_2, \dots, X_p$ and ϵ is a random error term, which is independent of X and has mean zero. Our models will not be concerned with the form of f but rather will try to formulate an estimate $\hat{f}$ of f that in turn produces an estimate $\hat{Y}$ of Y. The formulation of interest becomes

$$\hat{Y} = \hat{f}(X),$$

where $\hat{f}$ is treated as a black box and the mean-zero error ϵ is dropped.

Suppose the outcome is the set with classes $1, 2, \ldots, n$. The Bayes Classifier assigns each observation to the most likely class, given its predictor values. In other words, we assign class $j \in \{1, 2, \ldots, n\}$ to the test observation x_0 if

$$Pr(Y = j|X = x_0) = \max_i Pr(Y = i|X = x_0)$$

The Bayes classifier produces the lowest possible test error rate called the Bayes error rate. In other words, the Bayes classifier minimizes the test error defined by

$$\frac{1}{2}\sum_{i=1}^{n} I(y_i \neq \hat{y}_i),$$

where $I(y_i \neq \hat{y}_i)$ is an indicator variable that equals 1 if $y_i \neq \hat{y}_i$ and zero if $y_i = \hat{y}_i$ (i.e., the ith observation was classified correctly). The Bayes decision boundary is determined by those observations which conditional probability is exactly 0.5.

3 Data Analysis

Our study focuses on the "Differentiated Thyroid Cancer Recurrence" dataset [4] hosted by the UCI Machine Learning Repository. The UCI Machine Learning Repository offers a wide array of datasets used for empirical analysis in machine learning and data mining [12]. Established by the University of California, Irvine, this repository facilitates academic and educational pursuits by providing free access to datasets that cover various domains. As of March, 2024, it hosts and maintains over 600 datasets.

The "Differentiated Thyroid Cancer Recurrence" dataset encompasses 383 samples or observations, and a range of 17 variables pertinent to thyroid cancer, including patient demographics, clinical features, and pathological details, all aimed at elucidating patterns associated with cancer recurrence.

We will employ six distinct modeling methods to analyze our dataset: Artificial Neural Network (ANN), K-Nearest Neighbors (KNN), Support Vector Machine (SVM), Logistic Regression (LR), Random Forest (RF), and Extreme Gradient Boosting (XGBoost). Each of these methods brings unique strengths to the analysis, with ANN providing deep learning capabilities, KNN offering simplicity and ease of interpretation, SVM delivering powerful discriminative classification, LR providing an intuitive and easily trainable implementation, and the ensemble methods RF and XGBoost offering highly robust and accurate tree algorithms – thereby encompassing a comprehensive approach to predicting cancer recurrence in the studied dataset.

To prepare our data for modeling, we fix a typographical error, remove duplicate observations, and transform categorical variables into factors.

```r
#' Load raw data.
cleaned_data <-
  readr::read_csv(here::here('data/raw-data.csv')) |>
  dplyr::distinct() |>
  dplyr::rename(`Hx Radiotherapy` = 'Hx Radiothreapy') |>
  dplyr::mutate(Gender = ifelse(Gender == 'F', 'Female', 'Male')) |>
  dplyr::mutate(
    Gender = factor(Gender, levels = c('Female', 'Male')),
    Smoking = factor(Smoking, levels = c('Yes', 'No')),
    `Hx Smoking` = factor(`Hx Smoking`, levels = c('Yes', 'No')),
    `Hx Radiotherapy` = factor(`Hx Radiotherapy`, levels = c('Yes', 'No')),
    `Thyroid Function` = factor(
      `Thyroid Function`,
      levels = c('Euthyroid', 'Clinical Hyperthyroidism',
                 'Subclinical Hyperthyroidism', 'Clinical Hypothyroidism',
                 'Subclinical Hypothyroidism')),
    `Physical Examination` = factor(`Physical Examination`,
                                    levels = c('Normal', 'Diffuse goiter',
                                               'Single nodular goiter-right',
                                               'Single nodular goiter-left',
                                               'Multinodular goiter')),
    Adenopathy = factor(Adenopathy,
                        levels = c('No', 'Right', 'Left', 'Bilateral',
                                   'Posterior', 'Extensive')),
    Pathology = factor(
      Pathology,
      levels = c('Papillary', 'Micropapillary', 'Follicular',
                 'Hurthel cell')),
    Focality = factor(Focality, levels = c('Uni-Focal', 'Multi-Focal')),
    `T` = factor(`T`, levels = c('T1a', 'T1b', 'T2', 'T3a', 'T3b', 'T4a',
                                 'T4b')),
    N = factor(N, levels = c('N0', 'N1b', 'N1a')),
    M = factor(M, levels = c('M0', 'M1')),
    Stage = factor(Stage, levels = c('I', 'II', 'III', 'IVA', 'IVB')),
    Response = factor(
      Response,
      levels = c('Excellent', 'Biochemical Incomplete',
                 'Structural Incomplete', 'Indeterminate')),
    Risk = factor(Risk, levels = c('Low', 'Intermediate', 'High')),
    Recurred = factor(Recurred, levels = c('Yes', 'No'))
  )
```

After removing duplicates, our data has 364 observations. Out of the 17 variables, 16 will be used as features, leaving `Recurred` as the target variable to be predicted. Among the patients, there is a significant disparity between males and females: 293(80.5%) are females and 71(19.5%) are males. Males are about evenly distributed in terms of cancer recurrence with 59.2% total recurred cases. On the other hand, females are not evenly distributed in terms of cancer recurrence with 22.5% total recurred cases (see Figure 1).

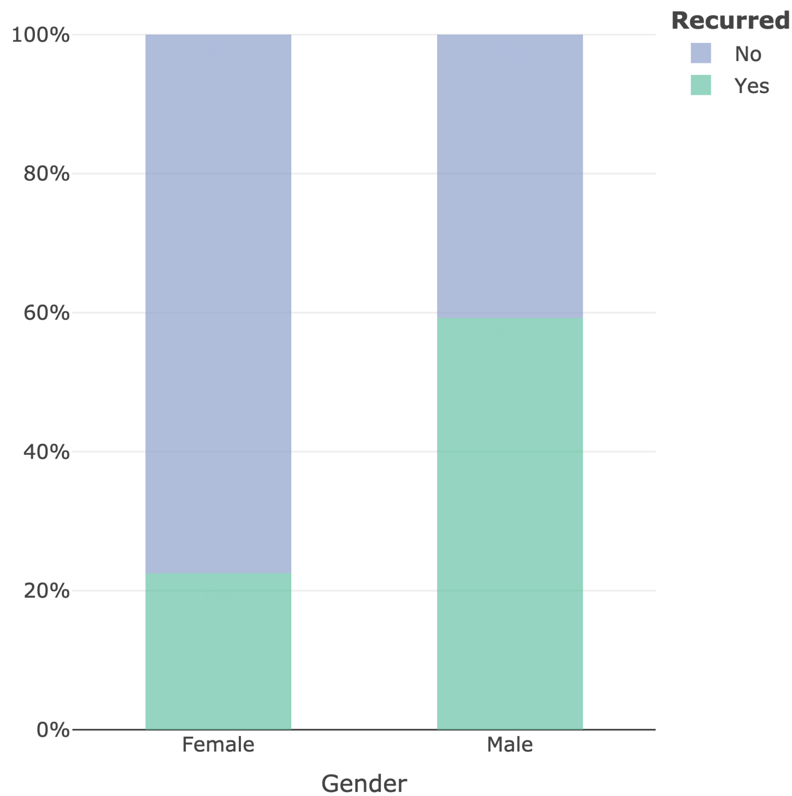

Figure 1: Gender Distribution by Cancer Recurrence.

The distribution of `Age` by cancer recurrence is shown in Figure 2. Note that, in general, older patients are more likely to recur.

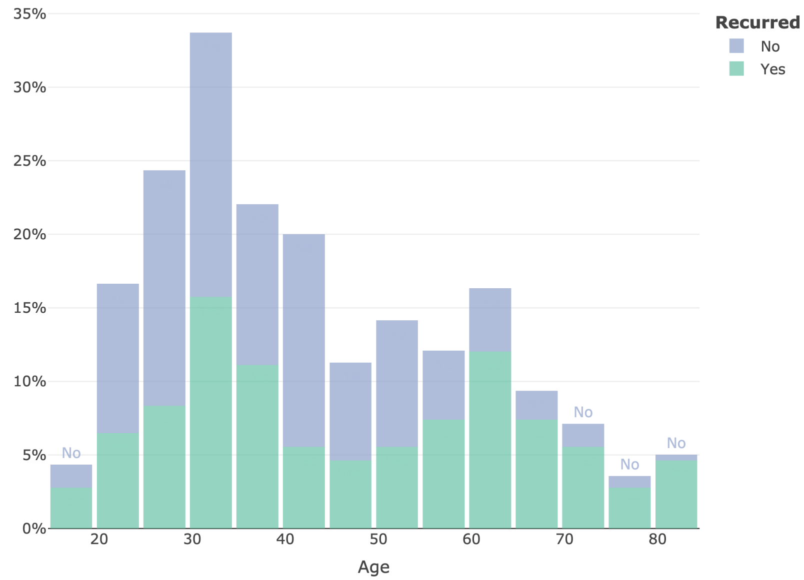

Figure 2: Age Distribution by Cancer Recurrence

Besides `Age`, the rest of the features are categorical. One interesting categorical feature is `Adenopathy`. It represents the presence of swollen lymph nodes during physical examination. The different adenopathy types observed are no adenopathy, anterior right, anterior left, bilateral (i.e., both sides of the body), posterior, and extensive (i.e., involves all the locations). Note the high correlation between swollen lymph nodes and DTC recurrence rate (see Figure 3).

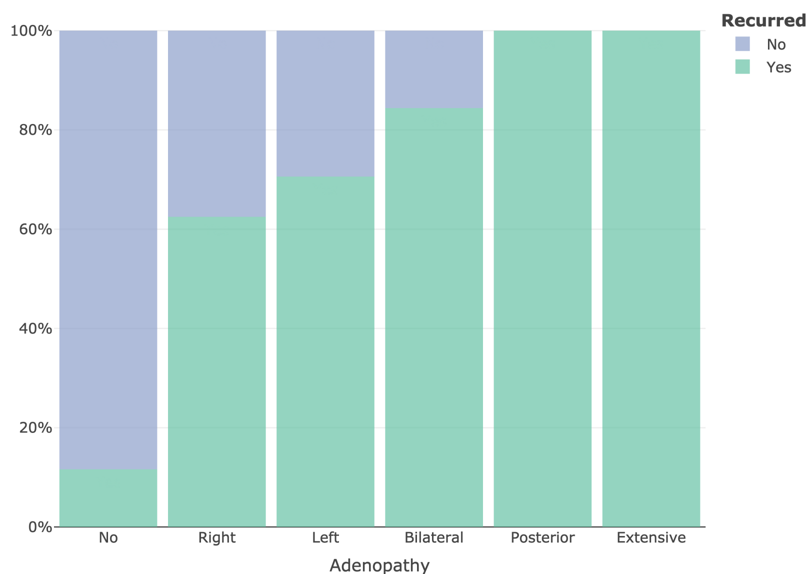

Figure 3: Adenopathy Distribution by cancer recurrence.

A summary of all the features and their categories are shown in Table 1.

Table 1: Feature Names and their Distinct Values

Feature	Values
Gender	Female, Male
Smoking	Yes, No
Hx Smoking	Yes, No
Hx Radiotherapy	Yes, No
Thyroid Function	Euthyroid, Clinical Hyperthyroidism, Subclinical Hyperthyroidism, Clinical Hypothyroidism, Subclinical Hypothyroidism
Physical Examination	Normal, Diffuse goiter, Single nodular goiter-right, Single nodular goiter-left, Multinodular goiter
Adenopathy	No, Right, Left, Bilateral, Posterior, Extensive
Pathology	Papillary, Micropapillary, Follicular, Hurthel cell
Focality	Uni-Focal, Multi-Focal
Risk	Low, Intermediate, High

T	T1a, T1b, T2, T3a, T3b, T4a, T4b
N	N0, N1b, N1a
M	M0, M1
Stage	I, II, III, IVA, IVB
Response	Excellent, Biochemical Incomplete, Structural Incomplete, Indeterminate

4 Model Training

Let us split the cleaned data set into a training (75%) set and a test (25%) set using a random generator. The training data will be further separated into 10 folds for cross-validation.

```
# Split data into training and test sets.
set.seed(314)
data_split <- rsample::initial_split(cleaned_data)
train_data <- rsample::training(data_split)
test_data <- rsample::testing(data_split)

# Split the training data into 10-folds for cross-validation.
set.seed(3145)
data_cross_val <- rsample::vfold_cv(train_data)

# Set aside the outcome column of the sample test data.
test_outcome <- factor(test_data$Recurred)
```

The `recipes` package is useful to create a blueprint of the pre-processing steps that will be applied to our data during model training. We use this package to specify that

- the minimum number of features with absolute correlations less that 0.6 should be removed,
- the numeric features should be normalized, and
- the categorical variables should be transformed into numerical variables.

```
# Create recipe for the data prep.
data_rec <- recipes::recipe(Recurred ~ ., data = train_data) |>
  recipes::step_corr(threshold = 0.6) |>
  recipes::step_normalize(recipes::all_numeric()) |>
  recipes::step_dummy(recipes::all_nominal_predictors())
```

4.1 K-Nearest Neighbors

4.1.1 Model Description

The K-Nearest Neighbors (KNN) algorithm is a nonparametric method used for classification. It classifies a given sample based on the proximity to the training data. The algorithm determines the class of a point X by identifying the most common class label among its k nearest neighbors, where k is a predetermined hyperparameter. Unlike other algorithms, the KNN classifier does not involve training a model; instead, it memorizes the training data, making it a "lazy" algorithm.

The primary hyperparameters of the KNN algorithm are k, the distance measure, and the weight function. Common distance measures include Euclidean distance, Manhattan distance, and Minkowski distance.

Choosing the optimal value for k is crucial and involves balancing the bias-variance tradeoff. A small k results in low bias and high variance. Low bias means the model captures the complexity of the training data very well, but high variance means the model is highly sensitive to the specifics of the training data, often leading to overfitting and higher test errors. As k increases, the model averages over more neighbors, which smooths out the predictions and reduces the model's sensitivity to individual data points, thus reducing variance. Therefore, a large k results in high bias and low variance. The model may become too simplistic, leading to higher bias, but it becomes less sensitive to the training data, making it more robust to noise and better at generalizing to new data.

To avoid classification ties, it is advisable to select k appropriately. For binary classification, this typically means choosing an odd k. Additionally, to enhance model flexibility, a weighted version of KNN can be employed, where the influence of each of the k nearest neighbors is weighted inversely by their distance to the test point. We will tune these three parameters below.

4.1.2 Model Workflow

Below we create a KNN model specification and workflow indicating the model hyperparameters: a number of neighbors (i.e., k), a weight function, and a distance function. To optimize our model, we will use the `tune::tune()` function to find optimal values of these parameters based on model accuracy.

```
# Create model specification.
knn_model_spec <-
  parsnip::nearest_neighbor(
    neighbors = tune::tune(),
    dist_power = tune::tune(),
```

```r
    weight_func = tune::tune()
  ) |>
  parsnip::set_mode('classification') |>
  parsnip::set_engine('kknn')

# Create model workflow.
knn_workflow <- workflows::workflow() |>
  workflows::add_model(knn_model_spec) |>
  workflows::add_recipe(data_rec)
```

4.1.3 Model Tuning and Fitting

Next, we run our prepared workflow. To speed up the computation, we utilize parallel computing, distributing the tasks across multiple cores.

We fine-tune the model hyperparameters (namely k, the distance function, and the weight function) using the 10-fold cross-validation setup. We then select the best model based on accuracy.

```r
#' Check number of available cores.
cores_no <- parallel::detectCores() - 1

#' Start timer.
tictoc::tic()

# Create and register clusters.
clusters <- parallel::makeCluster(cores_no)
doParallel::registerDoParallel(clusters)

# Fine-tune the model params.
knn_res <- tune::tune_grid(
  object = knn_workflow,
  resamples = data_cross_val,
  control = tune::control_resamples(save_pred = TRUE)
)

# Select the best fit based on accuracy.
knn_best_fit <-
  knn_res |>
  tune::select_best(metric = 'accuracy')

# Finalize the workflow with the best parameters.
```

```
knn_final_workflow <-
  knn_workflow |>
  tune::finalize_workflow(knn_best_fit)

# Fit the final model using the best parameters.
knn_final_fit <-
  knn_final_workflow |>
  tune::last_fit(data_split)

# Stop clusters.
parallel::stopCluster(clusters)

# Stop timer.
tictoc::toc()
```

```
17.607 sec elapsed
```

4.1.4 Model Performance

We then apply our selected model to the test set. The final metrics are given in Table 2.

```
# Use the best fit to make predictions on the test data.
knn_pred <-
  knn_final_fit |>
  tune::collect_predictions() |>
  dplyr::mutate(truth = factor(.pred_class))
```

Table 2: KNN Performance Metrics: Accuracy, Precision, Recall, and Specificity.

Metric	Value
Accuracy	90.1
Precision	76.9
Recall	87.0
Specificity	91.2

4.2 Support Vector Machine

4.2.1 Model Description

Support Vector Machines (SVM) are powerful supervised learning algorithms used for both classification and regression tasks. For classification, SVM works by finding the hyperplane that best separates data points of different classes in a high-dimensional space. The optimal hyperplane is determined by maximizing the margin between the closest points of the classes, known as support vectors.

SVM is particularly effective in high-dimensional spaces and is useful when the number of dimensions exceeds the number of samples. It can employ various kernel functions—such as linear, polynomial, and radial basis function (RBF)—to handle non-linear classification by mapping input features into higher-dimensional spaces.

The most commonly used kernel in SVM is the Radial Basis Function (RBF) kernel, also known as the Gaussian kernel. The RBF kernel maps input features into an infinite-dimensional space, allowing SVM to create complex decision boundaries. The RBF kernel function is defined as:

$$K(X_i, X_j) = e^{-\frac{||X_i - X_j||^2}{2\sigma^2}}$$

where X_i and X_j are the input feature vectors, and σ is a parameter that determines the spread of the kernel and controls the influence of individual training samples.

4.2.2 Model Workflow

We will create an SVM model specification and workflow, indicating the model hyperparameters σ (or `rbf_sigma`) and `cost`. The `rbf_sigma` parameter controls the influence of individual training examples, while the `cost` parameter controls the trade-off between achieving a low training error and a low testing error, which affects the model's ability to generalize. To optimize our model, we will use the `tune::tune()` function to find the optimal values of these parameters in terms of model accuracy.

```
# Create model specification.
svm_model_spec <-
  parsnip::svm_rbf(
    cost = tune::tune(),
    rbf_sigma = tune::tune()
  ) |>
  parsnip::set_engine('kernlab') |>
  parsnip::set_mode('classification')

# Create model workflow.
```

```
svm_workflow <- workflows::workflow() |>
  workflows::add_model(svm_model_spec) |>
  workflows::add_recipe(data_rec)
```

4.2.3 Model Tuning and Fitting

As we did for KNN, we use parallel computing to fine-tuning our model using the 10-fold cross-validation we set up earlier. We end this section by selecting the best model based on accuracy.

```
#' Check number of available cores.
cores_no <- parallel::detectCores() - 1

#' Start timer.
tictoc::tic()

# Create and register clusters.
clusters <- parallel::makeCluster(cores_no)
doParallel::registerDoParallel(clusters)

# Fine-tune the model params.
svm_res <- tune::tune_grid(
  object = svm_workflow,
  resamples = data_cross_val,
  control = tune::control_resamples(save_pred = TRUE)
)

# Select the best fit based on accuracy.
svm_best_fit <-
  svm_res |>
  tune::select_best(metric = 'accuracy')

# Finalize the workflow with the best parameters.
svm_final_workflow <-
  svm_workflow |>
  tune::finalize_workflow(svm_best_fit)

# Fit the final model using the best parameters.
svm_final_fit <-
  svm_final_workflow |>
  tune::last_fit(data_split)
```

```r
# Stop clusters.
parallel::stopCluster(clusters)

# Stop timer.
tictoc::toc()
```

```
21.555 sec elapsed
```

4.2.4 Model Performance

We then apply our selected model to the test set. The final metrics are given in Table 3.

```r
# Use the best fit to make predictions on the test data.
svm_pred <-
  svm_final_fit |>
  tune::collect_predictions() |>
  dplyr::mutate(truth = factor(.pred_class))
```

Table 3: SVM Performance Metrics: Accuracy, Precision, Recall, and Specificity.

Metric	Value
Accuracy	78.0
Precision	23.1
Recall	100.0
Specificity	76.5

4.3 Artificial Neural Network

4.3.1 Model Description

Artificial Neural Networks (ANNs) are a class of machine learning algorithms inspired by the structure and function of the human brain. They consist of interconnected layers of nodes, or neurons, which process input data to perform tasks such as classification, regression, and pattern recognition. ANNs are particularly effective for complex tasks like image and speech recognition, natural language processing, financial forecasting, and medical diagnosis.

An ANN is composed of multiple layers, including an input layer, one or more hidden layers, and an output layer. The input layer receives the raw data, the hidden layers process the data through various transformations, and the output layer produces the final prediction or

classification. Each connection between neurons has an associated weight, and each neuron has a bias term. These parameters are adjusted during the training process to minimize the error in predictions.

The training process of an ANN involves forward propagation, where input data is passed through the network layer by layer. Each neuron applies an activation function to compute its output, introducing non-linearity to help the network learn complex patterns. The loss, or error, between the network's output and the true target values is calculated using a loss function. Through backpropagation, the loss is propagated backward through the network, and the weights and biases are adjusted using an optimization algorithm like gradient descent.

ANNs offer significant advantages, including flexibility in modeling complex relationships and the ability to scale for large datasets and intricate tasks. Their ability to learn and generalize from data makes them powerful tools in various applications, driving advancements in fields ranging from technology and finance to healthcare and beyond.

4.3.2 Model Workflow

Let us start by specifying the ANN model and creating the model workflow. Specifically, we will define a multilayer perceptron model (i.e., a single-layer, feed-forward neural network). The key parameters we will set include the number of epochs (or training iterations), the number of hidden units, the penalty (or weight decay), and the learning rate.

```r
# Create model specification.
ann_model_spec <-
  parsnip::mlp(
    epochs = tune::tune(),
    hidden_units = tune::tune(),
    penalty = tune::tune(),
    learn_rate = 0.1
  ) |>
  parsnip::set_engine('nnet') |>
  parsnip::set_mode('classification')

# Create model workflow.
ann_workflow <- workflows::workflow() |>
  workflows::add_model(ann_model_spec) |>
  workflows::add_recipe(data_rec)
```

4.3.3 Model Tuning and Fitting

We will proceed to tune all the parameters except for the learning rate. This is because the `nnet` package does not support tuning the learning rate.

```r
#' Check number of available cores.
cores_no <- parallel::detectCores() - 1

#' Start timer.
tictoc::tic()

# Create and register clusters.
clusters <- parallel::makeCluster(cores_no)
doParallel::registerDoParallel(clusters)

# Fine-tune the model params.
ann_res <- tune::tune_grid(
  object = ann_workflow,
  resamples = data_cross_val,
  control = tune::control_resamples(save_pred = TRUE)
)

# Select the best fit based on accuracy.
ann_best_fit <-
  ann_res |>
  tune::select_best(metric = 'accuracy')

# Finalize the workflow with the best parameters.
ann_final_workflow <-
  ann_workflow |>
  tune::finalize_workflow(ann_best_fit)

# Fit the final model using the best parameters.
ann_final_fit <-
  ann_final_workflow |>
  tune::last_fit(data_split)

# Stop clusters.
parallel::stopCluster(clusters)

# Stop timer.
tictoc::toc()
```

```
19.291 sec elapsed
```

4.3.4 Model Performance

We then apply our selected model to the test set. The final metrics are given in Table 4.

```
# Use the best fit to make predictions on the test data.
ann_pred <-
  ann_final_fit |>
  tune::collect_predictions() |>
  dplyr::mutate(truth = factor(.pred_class))
```

Table 4: ANN Performance Metrics: Accuracy, Precision, Recall, and Specificity.

Metric	Value
Accuracy	92.3
Precision	84.6
Recall	88.0
Specificity	93.9

4.4 Logistic Regression

4.4.1 Model Description

Logistic Regression (LR) is a supervised learning algorithm widely used for classification problems. It is particularly effective for binary classification tasks, where the outcome variable can take one of two possible values. The model predicts the probability that a given input belongs to a specific class by applying the logistic (sigmoid) function, which transforms a linear combination of input features into a probability value between 0 and 1.

For binary classification, the logistic function is defined as $\sigma(\hat{Y}_i) = 1/(1 + e^{-\hat{Y}_i})$ where $\hat{Y}$ is a linear combination of the input features. The probability of the outcome i being the positive class (represented as 1) is given by:

$$\sigma(\hat{Y}_i) = \sigma(\beta_0 + \beta_1 X_{1,i} + \beta_2 X_{2,i} + \ldots + \beta_p X_{p,i}),$$

where β_0 is the intercept, and $\beta_1, \beta_2, \ldots, \beta_p$ are the coefficients corresponding to the input features $X_1, X_2, \ldots, X_p$. These coefficients are estimated using the method of maximum likelihood estimation (MLE), which maximizes the likelihood of the observed data.

LR can also be extended to handle multi-class classification problems through multinomial logistic regression. In this case, the model uses the softmax function to generalize to multiple classes. The softmax function is an extension of the logistic function for multiple classes and is defined as,

$$Pr(Y = j|X = x_0) = \frac{e^{\ _j x_0}}{\sum_{i=1}^{n} e^{\ _i \cdot x_0}}$$

where x_0 is an observation, n is the number of classes, and $_j$ is the coefficient vector for class j.

The primary advantage of LR is its interpretability. Each coefficient indicates the change in the log-odds of the outcome for a one-unit change in the corresponding predictor variable. This provides clear insights into the influence of each predictor on the probability of the outcome. Despite its simplicity, LR is a powerful tool for both binary and multi-class classification, making it suitable for a wide range of applications where the relationship between the predictors and the log-odds is approximately linear.

4.4.2 Model Workflow

In this section, we will train our LR model and find the optimal values for the model parameters. The key parameter we will optimize is the penalty parameter, which refers to the regularization term added to the loss function to prevent overfitting. We will find the optimal penalty value to improve model performance. Additionally, we will set mixture = 1 to apply Lasso regularization, which helps in potentially removing irrelevant predictors and choosing a simpler model.

```
# Create model specification.
lr_model_spec <-
  parsnip::logistic_reg(
    penalty = tune(),
    mixture = 1) |>
  parsnip::set_mode('classification') |>
  parsnip::set_engine('glmnet')

# Create model workflow.
lr_workflow <- workflows::workflow() |>
  workflows::add_model(lr_model_spec) |>
  workflows::add_recipe(data_rec)
```

4.4.3 Model Tuning and Fitting

```
#' Check number of available cores.
cores_no <- parallel::detectCores() - 1

#' Start timer.
tictoc::tic()

# Create and register clusters.
clusters <- parallel::makeCluster(cores_no)
doParallel::registerDoParallel(clusters)

# Fine-tune the model params.
lr_res <- tune::tune_grid(
  object = lr_workflow,
  resamples = data_cross_val,
  control = tune::control_resamples(save_pred = TRUE)
)

# Select the best fit based on accuracy.
lr_best_fit <-
  lr_res |>
  tune::select_best(metric = 'accuracy')

# Finalize the workflow with the best parameters.
lr_final_workflow <-
  lr_workflow |>
  tune::finalize_workflow(lr_best_fit)

# Fit the final model using the best parameters.
lr_final_fit <-
  lr_final_workflow |>
  tune::last_fit(data_split)

# Stop clusters.
parallel::stopCluster(clusters)

# Stop timer.
tictoc::toc()
```

```
15.21 sec elapsed
```

4.4.4 Model Performance

We then apply our selected model to the test set. The final metrics are given in Table 5.

```
# Use the best fit to make predictions on the test data.
lr_pred <-
  lr_final_fit |>
  tune::collect_predictions() |>
  dplyr::mutate(truth = factor(.pred_class))
```

Table 5: Logistic Regression Performance Metrics: Accuracy, Precision, Recall, and Specificity.

Metric	Value
Accuracy	93.4
Precision	84.6
Recall	91.7
Specificity	94.0

4.5 Extreme Gradient Boosting

4.5.1 Model Description

Extreme Gradient Boosting (XGBoost) is an advanced implementation of gradient boosting designed to enhance performance and speed. It builds upon the principles of gradient boosting to provide a highly efficient, flexible, and portable library that supports both regression and classification tasks. XGBoost has become one of the most popular machine learning algorithms due to its high performance and scalability.

XGBoost operates by sequentially adding decision trees to an ensemble. Each tree is built to correct the errors of the previous trees in the ensemble. The process begins with an initial model, typically a simple model such as the mean of the target variable. At each subsequent step, a new decision tree is added to the model to predict the residuals (errors) of the previous trees. Each tree is built by optimizing an objective function that combines a loss function and a regularization term. The regularization term helps prevent overfitting by penalizing the complexity of the model. After each tree is added, the residuals are updated. The new tree aims to minimize these residuals, improving the overall model's performance.

The node splitting in each tree is guided by an objective function, which typically involves minimizing a loss function (such as mean squared error for regression or log loss for classification) while including a regularization term. The final prediction is the sum of the predictions from all the trees in the ensemble, effectively reducing variance. This process is depicted in the attached flowchart, showing how each tree contributes to the final model.

XGBoost has several key advantages. It incorporates both L1 (Lasso) and L2 (Ridge) regularization to prevent overfitting and manage model complexity. The algorithm supports parallel processing, significantly speeding up the training process. XGBoost can handle missing values internally, making it robust to incomplete datasets. Additionally, users can define custom objective functions and evaluation metrics, allowing for flexibility in optimization.

4.5.2 Model Workflow

To effectively train our XGBoost model and find the optimal hyperparameters, we will set up a workflow that includes model specification and data preprocessing. The hyperparameters to be tuned include:

- `tree_depth`: Controls the maximum depth of each tree, impacting the model's complexity.
- `min_n`: Specifies the minimum number of observations that must exist in a node for a split to be attempted, preventing overly specific branches and encouraging generalization.
- `loss_reduction`: Sets the minimum reduction in the loss function required to make a further partition on a leaf node, helping to control overfitting by making the algorithm more conservative.
- `sample_size`: Determines the fraction of the training data used for fitting each individual tree, introducing randomness and preventing overfitting.
- `mtry`: Sets the number of features considered when looking for the best split, adding variability to enhance generalization.
- `learn_rate`: Also known as the shrinkage parameter, controls the rate at which the model learns. Smaller learning rates can lead to better performance by allowing the model to learn more slowly and avoid overfitting.

```
# Create model specification.
xgboost_model_spec <-
  boost_tree(
    trees = 1000,
    tree_depth = tune(),
    min_n = tune(),
    loss_reduction = tune(),
    sample_size = tune(),
    mtry = tune(),
    learn_rate = tune()
  ) |>
  set_engine('xgboost') |>
  set_mode('classification')

# Create model workflow.
```

```
xgboost_workflow <- workflows::workflow() |>
  workflows::add_model(xgboost_model_spec) |>
  workflows::add_recipe(data_rec)
```

4.5.3 Model Tuning and Fitting

```
#' Check number of available cores.
cores_no <- parallel::detectCores() - 1

#' Start timer.
tictoc::tic()

# Create and register clusters.
clusters <- parallel::makeCluster(cores_no)
doParallel::registerDoParallel(clusters)

# Fine-tune the model params.
xgboost_res <- tune::tune_grid(
  object = xgboost_workflow,
  resamples = data_cross_val,
  control = tune::control_resamples(save_pred = TRUE)
)
```

```
# Select the best fit based on accuracy.
xgboost_best_fit <-
  xgboost_res |>
  tune::select_best(metric = 'accuracy')

# Finalize the workflow with the best parameters.
xgboost_final_workflow <-
  xgboost_workflow |>
  tune::finalize_workflow(xgboost_best_fit)

# Fit the final model using the best parameters.
xgboost_final_fit <-
  xgboost_final_workflow |>
  tune::last_fit(data_split)

# Stop clusters.
parallel::stopCluster(clusters)
```

```
# Stop timer.
tictoc::toc()
```

```
23.54 sec elapsed
```

4.5.4 Model Performance

We then apply our selected model to the test set. The final metrics are given in Table 6.

```
# Use the best fit to make predictions on the test data.
xgboost_pred <-
  xgboost_final_fit |>
  tune::collect_predictions() |>
  dplyr::mutate(truth = factor(.pred_class))
```

Table 6: XGBoost Performance Metrics: Accuracy, Precision, Recall, and Specificity.

Metric	Value
Accuracy	91.2
Precision	73.1
Recall	95.0
Specificity	90.1

4.6 Random Forest

4.6.1 Model Description

Random forest is an ensemble learning method that constructs multiple decision trees during training and outputs the mode of the classes (classification) or mean prediction (regression) of the individual trees. This method is particularly effective for classification problems, such as the one we are dealing with in the Thyroid dataset where the target variable is categorical. Each decision tree in a random forest splits the predictor space into distinct regions using recursive binary splits. For instance, a tree might first split based on whether Age < 35 and then further split based on whether Gender $=$ Female to predict cancer recurrence. These splits are chosen to minimize a specific error criterion, such as the Gini index or entropy [13].

A significant limitation of individual decision trees is their high variance; small changes in the training data can lead to very different tree structures. Random forest addresses this by using bagging, where multiple trees are trained on different bootstrap samples of the data. The final

prediction is made by aggregating the predictions of all the trees, typically through majority voting in classification problems. This process reduces variance because the average of many uncorrelated trees' predictions is less variable than the prediction of a single tree.

Random forest further reduces correlation between trees by selecting a random subset of predictors to consider for each split, rather than considering all predictors. Typically, for classification problems, this subset size is approximately $\sqrt{p}$, where p is the total number of predictors. This random selection of features ensures that the trees are less similar to each other, which reduces the correlation between their predictions and leads to a greater reduction in variance. By combining bagging with feature randomness, random forests create robust models that are less prone to overfitting and provide better generalization to new data.

4.6.2 Model Workflow

In this section, we will set up a workflow to train our Random Forest model. The goal is to optimize the following hyperparameters to achieve the best performance on our classification task:

- `trees`: This parameter specifies the total number of trees to be grown in the forest. Tuning the number of trees can help ensure that the model is robust and neither overfitting nor underfitting the data.
- `min_n`: This parameter sets the minimum number of observations required in a terminal node. Tuning `min_n` helps control the size of the trees, affecting the model's ability to generalize to new data.

```
# Create model specification.
rf_model_spec <-
  parsnip::rand_forest(
    trees = 500,
    min_n = tune::tune()
  ) |>
  parsnip::set_engine('ranger') |>
  parsnip::set_mode('classification')

# Create model workflow.
rf_workflow <- workflows::workflow() |>
  workflows::add_model(rf_model_spec) |>
  workflows::add_recipe(data_rec)
```

4.6.3 Model Tuning and Fitting

```
#' Check number of available cores.
cores_no <- parallel::detectCores() - 1

#' Start timer.
tictoc::tic()

# Create and register clusters.
clusters <- parallel::makeCluster(cores_no)
doParallel::registerDoParallel(clusters)

# Fine-tune the model params.
rf_res <- tune::tune_grid(
  object = rf_workflow,
  resamples = data_cross_val,
  control = tune::control_resamples(save_pred = TRUE)
)

# Select the best fit based on accuracy.
rf_best_fit <-
  rf_res |>
  tune::select_best(metric = 'accuracy')

# Finalize the workflow with the best parameters.
rf_final_workflow <-
  rf_workflow |>
  tune::finalize_workflow(rf_best_fit)

# Fit the final model using the best parameters.
rf_final_fit <-
  rf_final_workflow |>
  tune::last_fit(data_split)

# Stop clusters.
parallel::stopCluster(clusters)

# Stop timer.
tictoc::toc()
```

```
21.431 sec elapsed
```

4.6.4 Model Performance

We then apply our selected model to the test set. The final metrics are given in Table 7.

```
# Use the best fit to make predictions on the test data.
rf_pred <-
  rf_final_fit |>
  tune::collect_predictions() |>
  dplyr::mutate(truth = factor(.pred_class))
```

Table 7: Random Forest Performance Metrics: Accuracy, Precision, Recall, and Specificity.

Metric	Value
Accuracy	94.5
Precision	84.6
Recall	95.7
Specificity	94.1

5 Model Comparison

To assess model performance, we will compare the accuracy, precision, recall, and specificity metrics of our models. These metrics are defined as follows.

- Accuracy: proportion of correct predictions for the test data or $\frac{TN+TP}{TN+TP+FN+FP}$.
- Precision: proportion of correctly classified positive observations among all observations that are classified as positive by the model or $\frac{TP}{TP+FP}$.
- Recall: proportion of correctly classified positive observations among all actual positive observations or $\frac{TP}{TP+FN}$.
- Specificity: proportion of correctly classified negative observations among all actual negative observations or $\frac{TN}{TN+FP}$.

We evaluated the performance of our models on the Thyroid dataset using the metrics shown in Table 8.

Table 8: Performance Comparison: Accuracy, Precision, Recall, and Specificity.

Metric	Model	Value
Accuracy	Random Forest	94%

Precision	ANN, Logistic Regression, Random Forest	85%
Recall	SVM	100%
Specificity	Random Forest	94%

The Random Forest model emerged as the top performer in terms of overall accuracy and specificity, achieving a 94% accuracy rate and a 94% specificity rate, indicating its robustness in correctly identifying negative cases. The precision metric, though the lowest among the metrics, was 85% across the Artificial Neural Network, Logistic Regression, and Random Forest models, reflecting their ability to correctly classify positive cases. The Support Vector Machine model stood out in terms of recall, achieving a perfect score of 100%, suggesting its effectiveness in capturing all positive cases. These results highlight the Random Forest model as the most balanced and reliable for this classification task, combining high accuracy and specificity with competitive precision, while the SVM model excels in recall, making it particularly suitable for ensuring that all positive cases are identified.

6 Conclusion

This study aimed to evaluate and compare the effectiveness of various machine learning models in predicting the recurrence of Differentiated Thyroid Cancer (DTC). By examining key metrics such as accuracy, precision, recall, and specificity, we assessed the performance of models including Artificial Neural Network (ANN), Logistic Regression (LR), K-Nearest Neighbors (KNN), Support Vector Machine (SVM), Random Forest (RF), and Extreme Gradient Boosting (XGBoost).

Our findings indicate that the Random Forest model is the most robust and balanced classifier for this task. It achieved the highest accuracy and specificity rates, both at 94%, demonstrating its reliability in correctly identifying both positive and negative cases. This suggests that Random Forest is highly effective in distinguishing between patients who will and will not experience a recurrence of thyroid cancer.

In terms of precision, the ANN, Logistic Regression, and Random Forest models all achieved an 85% precision rate. This shows that these models are equally competent in accurately predicting positive cases among those identified as positive, minimizing false positives.

The SVM model excelled in recall, achieving a perfect score of 100%. This indicates that SVM is exceptionally effective at capturing all actual positive cases, making it a critical tool when the primary goal is to ensure that no positive cases are missed. This is particularly important in medical diagnostics where missing a positive case can have serious implications.

Overall, while the Random Forest model provides the best balance of performance across all metrics, the SVM model's outstanding recall rate highlights its utility in scenarios where it

is crucial to identify all positive cases. These results underscore the importance of selecting the appropriate model based on the specific needs of the task. For balanced performance and overall robustness, Random Forest is recommended. However, for applications where recall is paramount, SVM is the superior choice.

Future work can explore the integration of these models in a hybrid approach, leveraging the strengths of each to further improve prediction accuracy and reliability. Additionally, investigating the impact of different feature engineering techniques and more sophisticated hyperparameter tuning methods may yield further enhancements in model performance.

These findings contribute valuable insights into the application of machine learning in medical diagnostics, particularly for predicting the recurrence of DTC, and pave the way for more personalized and accurate treatment strategies.

7 Informed Consent

We used anonymous data for modeling and no consent was required for conducting this study.

8 Institutional Review Board Statement

Not applicable

9 Funding

There was no funding for conducting this study.

10 Data Availability Statement

The data is available at the UC Irvine Machine Learning Repository [12]. For more information, please refer to [4].

11 Acknowledgments

We thank our PRIMES mentor, Dr. Marly Gotti, for her guidance throughout the reading and research periods. We are also grateful to the PRIMES organizers for making this program possible.

12 Conflicts of Interest

None

13 Abbreviations

The following abbreviations are used in this manuscript:

- KNN: K-Nearest Neighbors
- SVM: Support Vector Machine
- RBF: Radial Basis Function
- ANN: Artificial Neural Network
- NNs: Neural Network(s)
- RF: Random Forest
- LR: Logistic Regression
- XGBoost: Extreme Gradient Boosting
- DTC: Differentiated Thyroid Cancer
- ML: Machine Learning

References

[1] Pellegriti, G., Frasca, F., Regalbuto, C., Squatrito, S. and Vigneri, R. (2013). Worldwide increasing incidence of thyroid cancer: Update on epidemiology and risk factors. *Journal of Cancer Epidemiology* **2013** 1–10.

[2] Bhattacharya, S., Mahato, R. K., Singh, S., Bhatti, G. K., Mastana, S. S. and Bhatti, J. S. (2023). Advances and challenges in thyroid cancer: The interplay of genetic modulators, targeted therapies, and AI-driven approaches. *Life Sciences* **332** 122110.

[3] NM, X., L, W. and C., Y. (2022). Improving the diagnosis of thyroid cancer by machine learning and clinical data. *Scientific report* **12** 11143.

[4] Borzooei, S. and Tarokhian, A. (2023). Differentiated thyroid cancer recurrence. *UCI Machine Learning Repository*.

[5] Abiodun, O. I., Jantan, A., Omolara, A. E., Dada, K. V., Mohamed, N. A. and Arshad, H. (2018). State-of-the-art in artificial neural network applications: A survey. *Heliyon* **4** e00938.

[6] Cervantes, J., Garcia-Lamont, F., Rodríguez-Mazahua, L. and Lopez, A. (2020). A comprehensive survey on support vector machine classification: Applications, challenges and trends. *Neurocomputing* **408** 189–215.

[7] Syriopoulos, P. K., Kotsiantis, S. B. and Vrahatis, M. N. (2022). Survey on KNN methods in data science. In *Learning and intelligent optimization* (D. E. Simos, V. A. Rasskazova, F. Archetti, I. S. Kotsireas and P. M. Pardalos, ed) pp 379–93. Springer International Publishing, Cham.

[8] Maalouf, M. (2011). Logistic regression in data analysis: An overview. *International Journal of Data Analysis Techniques and Strategies* **3** 281–99.

[9] Genuer, R., Poggi, J.-M. and Tuleau-Malot, C. (2010). Variable selection using random forests. *Pattern Recognition Letters* **31** 2225–36.

[10] Anju and Sharma, N. (2018). Survey of boosting algorithms for big data applications. *International Journal of Engineering Research and Technology (IJERT)* **5**.

[11] Kourou, K., Exarchos, T. P., Exarchos, K. P., Karamouzis, M. V. and Fotiadis, D. I. (2015). Machine learning applications in cancer prognosis and prediction. *Computational and Structural Biotechnology Journal* **13** 8–17.

[12] Kelly, R., Markelle andLongjohn and Nottingham, K. The UCI machine learning repository.

[13] G. James, T. H., D. Witten and Tibshirani, R. (2021). *An introduction to statistical learning*. Springer Texts in Statistics.