Comparative Analysis of Machine Learning Models for Thyroid Cancer Recurrence Prediction

Anay Aggarwal   Marly Gotti   Ekam Kaur   Susie Lu

TABLE OF CONTENTS

ABSTRACT

Background: Thyroid cancer is one of the most common endocrine malignancies, with Differentiated Thyroid Cancer (DTC) accounting for the majority of cases. Accurate prediction of cancer recurrence is essential for improving personalized treatment and patient outcomes. This study compares six machine learning algorithms—Artificial Neural Network (ANN), K-Nearest Neighbors (KNN), Support Vector Machine (SVM), Logistic Regression (LR), Random Forest (RF), and Extreme Gradient Boosting (XGBoost)—to identify the best model for predicting DTC recurrence.

Methods: We conducted a comparative analysis using a dataset from the UCI Machine Learning Repository, which includes demographic, clinical, and pathological data for thyroid cancer patients. Each algorithm was evaluated on key performance metrics, including accuracy, precision, recall, and specificity. Feature selection techniques, such as Principal Component Analysis (PCA) and Feature Importance Analysis (FIA), were applied to identify the most significant features influencing recurrence.

Results: Among the models tested, Random Forest achieved the highest overall accuracy and specificity, while SVM with the polynomial kernel excelled in recall, ensuring all positive cases were captured. Feature selection highlighted “Response”, “N”, “T”, “Risk”, and “Age” as the most impactful variables, contributing to model improvement and enhanced interpretability.

Conclusions: The Random Forest model demonstrates robust predictive power and is a strong candidate for clinical applications in DTC recurrence prediction, with potential to support more tailored treatment strategies. The study underscores the role of machine learning in advancing cancer care through improved predictive accuracy and personalized risk assessment.

1. INTRODUCTION

Thyroid cancer is among the most prevalent endocrine malignancies worldwide, with differentiated thyroid cancer (DTC) representing the majority of cases [15]. Effective management and prognosis of thyroid cancer rely heavily on timely and accurate prediction of recurrence. Traditional approaches, primarily based on clinical and pathological parameters, often lack the precision required for personalized treatment planning [3]. 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 [13].

This paper examines the effectiveness 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 stand out for their capacity to model intricate nonlinear relationships, making them highly effective in tackling complex medical prediction tasks [1]. In contrast, SVMs are particularly valued for their robustness in high-dimensional spaces, where they adeptly handle both linear and nonlinear data, proving highly suitable for classification challenges [5]. KNN provides a more intuitive approach by classifying cases based on feature-space proximity, efficiently leveraging the similarity between data points [17].

When it comes to simplicity and efficiency, LR remains a go-to choice, especially for determining decision boundaries in linearly separable data, with the added advantage of directly outputting class probabilities [12]. For high-dimensional, non-linear data, RF offers a powerful solution; by employing bagging across multiple classification trees, it achieves both accuracy and resilience [7]. Finally, XGBoost builds on previous models through sequential boosting, resulting in a fast, robust algorithm that's adaptable to a wide range of predictive tasks [2].

This study identifies the most effective model among six machine learning algorithms for predicting recurrence in DTC patients, contributing to improved 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. We further improve our six models by conducting feature selection via principal component analysis (PCA) and feature importance analysis (FIA). 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 $X_1, X_2, \ldots, X_p$. We assume that there is some relationship between Y and $X = (X_1, X_2, \ldots, X_p)$, which can be written in the general form

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

Here f is some fixed but unknown function of $X_1, X_2, \ldots, X_p$ and ϵ is a random error term, which is independent of X and has mean zero.

Our models do not focus on the exact form of f; instead, they estimate $\hat{f}$ 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

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

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 [10]. 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 includes 383 observations and 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 in a feature name, 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).

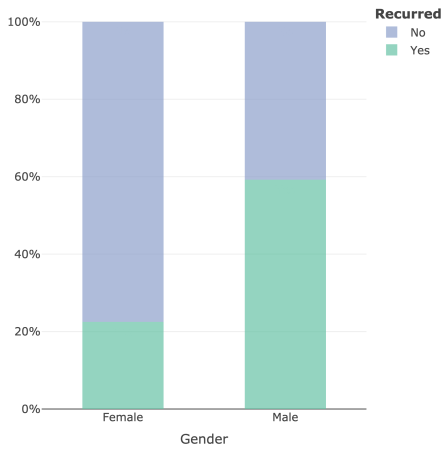

Figure 1: Gender distribution by cancer recurrence.

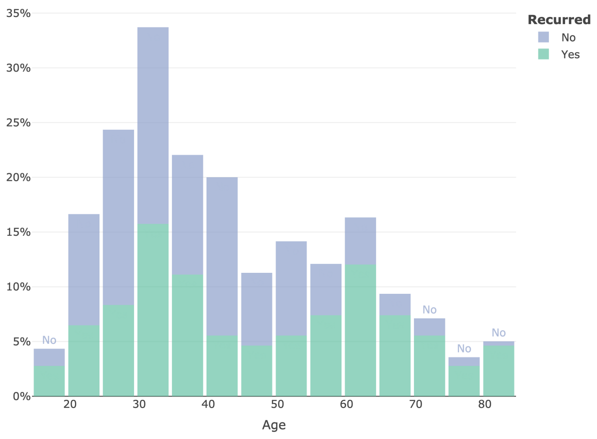

Figure 2: Age distribution by cancer recurrence.

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

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).

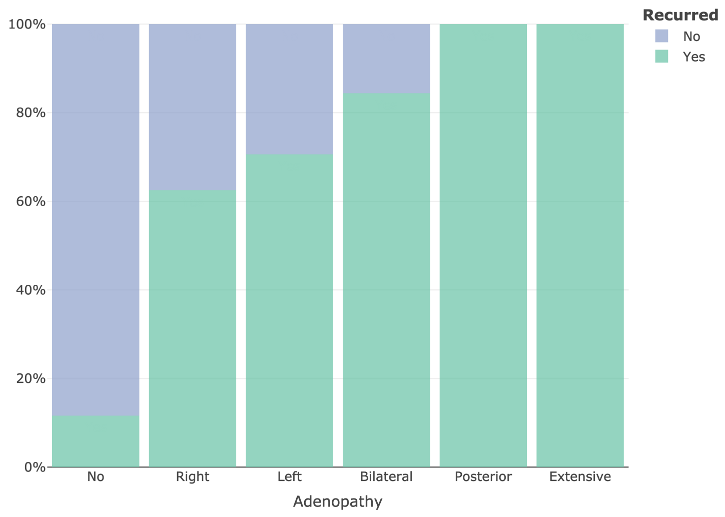

Figure 3: Adenopathy distribution by cancer recurrence.

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 dataset 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.

```r
# 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 than 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_dummy(recipes::all_nominal_predictors()) |>
  recipes::step_normalize(recipes::all_numeric_predictors()) |>
  recipes::step_corr(threshold = 0.6)
```

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 its 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(),
    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 to distribute 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.

```
#' 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.
```

```r
parallel::stopCluster(clusters)

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

```
10.017 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.

```r
# 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 (SVMs) are powerful supervised learning algorithms used for both classification and regression tasks. In the context of classification, SVMs aim to find the optimal hyperplane that best separates data points of different classes in a high-dimensional space. This optimal hyperplane is determined by maximizing the margin, or the distance, between the closest points of the opposing classes. These closest points are referred to as support vectors.

SVMs are particularly effective in high-dimensional spaces and are useful when the number of dimensions exceeds the number of samples. They are capable of handling non-linear classification tasks by employing various kernel functions—such as linear, polynomial, and radial basis function—to map input features into higher-dimensional spaces where they may become linearly separable.

The most commonly used kernel in SVMs 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 SVMs 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.

Another widely used kernel is the polynomial kernel, which maps input features into a higher-dimensional space with polynomials of degree d. The polynomial kernel function is defined as

$$K(X_i, X_j) = (X_i \cdot X_j)^d,$$

where X_i and X_j are the input feature vectors and d is the degree of the polynomial.

4.2.2. Model Workflow

In this section, we will construct two SVM models to explore different kernel functions: one utilizing a Radial Basis Function (RBF) kernel and the other employing a polynomial kernel.

We'll begin by focusing on the RBF kernel SVM. The accompanying code outlines the model specification and workflow. The critical hyperparameters for this model are `rbf_sigma` (or σ) and `cost`. The `rbf_sigma` parameter governs the extent to which individual training examples influence the decision boundary, while the cost parameter manages the trade-off between fitting the training data well (low training error) and avoiding overfitting, which impacts the model's ability to generalize to new, unseen data.

To fine-tune our RBF kernel SVM, we'll leverage the `tune::tune()` function. This will systematically search for the optimal combination of `rbf_sigma` and `cost` values that yields the highest accuracy on our validation set.

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

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

Subsequently, we will shift our attention to the polynomial kernel SVM. Similar to the RBF kernel model, we'll define the model specification and establish a workflow tailored to this kernel function. The polynomial kernel introduces its own set of hyperparameters, notably the `degree` which controls the complexity of the decision boundary. We'll again employ hyperparameter tuning to identify the optimal configuration that maximizes the model's performance.

```r
# Create model specification.
svm_poly_model_spec <-
  parsnip::svm_poly(
    cost = tune::tune(),
    degree = tune::tune()
  ) |>
  parsnip::set_engine('kernlab') |>
  parsnip::set_mode('classification')

# Create model workflow.
svm_poly_workflow <-
  workflows::workflow() |>
  workflows::add_model(svm_poly_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 models 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 each model's parameters.
svm_rbf_res <-
  tune::tune_grid(
    object = svm_rbf_workflow,
    resamples = data_cross_val,
    control = tune::control_resamples(save_pred = TRUE)
  )
svm_poly_res <-
  tune::tune_grid(
    object = svm_poly_workflow,
    resamples = data_cross_val,
    control = tune::control_resamples(save_pred = TRUE)
  )

# Select the best fit based on accuracy for each model.
svm_rbf_best_fit <-
  svm_rbf_res |>
  tune::select_best(metric = 'accuracy')
svm_poly_best_fit <-
  svm_poly_res |>
  tune::select_best(metric = 'accuracy')

# Finalize each model's workflow by selecting the corresponding best fit
svm_rbf_final_workflow <-
  svm_rbf_workflow |>
```

```r
  tune::finalize_workflow(svm_rbf_best_fit)
svm_poly_final_workflow <-
  svm_poly_workflow |>
  tune::finalize_workflow(svm_poly_best_fit)

# Find the last fit for each model.
svm_rbf_final_fit <-
  svm_rbf_final_workflow |>
  tune::last_fit(data_split)
svm_poly_final_fit <-
  svm_poly_final_workflow |>
  tune::last_fit(data_split)

# Stop clusters.
parallel::stopCluster(clusters)

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

```
12.846 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_rbf_pred <-
  svm_rbf_final_fit |>
  tune::collect_predictions() |>
  dplyr::mutate(truth = factor(.pred_class))
```

Table 3: RBF SVM performance metrics: Accuracy, Precision, Recall, and Specificity

Metric	Value
Accuracy	78.0%
Precision	23.1%
Recall	100.0%
Specificity	76.5%

We also apply our selected SVM with polynomial kernel to the test set. The final metrics are given in Table 4.

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

The SVM based on the RBF kernel has a perfect recall of 100%, so it excels at capturing all actual

Table 4: Polynomial SVM performance metrics: Accuracy, Precision, Recall, and Specificity

Metric	Value
Accuracy	94.5%
Precision	84.6%
Recall	95.7%
Specificity	94.1%

positive cases, even though its precision metric is relatively low. On the other hand, when it is more important to have high percentages for all four metrics, the SVM based on the polynomial kernel would be better.

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()
```

```
11.658 sec elapsed
```

4.3.4. 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.
ann_pred <-
  ann_final_fit |>
  tune::collect_predictions() |>
  dplyr::mutate(truth = factor(.pred_class))
```

Table 5: 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:

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

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,

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

where x_0 is an observation, n is the number of classes, and β_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::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)
```

```r
# 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()
```

```
9.044 sec elapsed
```

4.4.4. Model Performance

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

```r
# 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 6: LR 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.

```r
# 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

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

```r
# 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()
```

```
13.955 sec elapsed
```

4.5.4. 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.
xgboost_pred <-
  xgboost_final_fit |>
  tune::collect_predictions() |>
  dplyr::mutate(truth = factor(.pred_class))
```

Table 7: 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 [8].

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.

```r
# 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

```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)
```

```r
# 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()
```

```
10.637 sec elapsed
```

4.6.4. Model Performance

We then apply our selected model to the test set. The final metrics are given in Table 8.

```r
# 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 8: RF performance metrics: Accuracy, Precision, Recall, and Specificity

Metric	Value
Accuracy	94.5%
Precision	84.6%
Recall	95.7%
Specificity	94.1%

5. FEATURE SELECTION

We apply two feature selection techniques to identify the critical predictors of DTC. In this way, we can improve our six machine learning models by running them on the selected features only and also reduce overfitting. This is especially important for models such as KNN that are sensitive to high-dimensional data. Further, since the models rely on fewer features, they can be more easily applied on real-world data in which some patients can have incomplete clinical or pathological information.

5.1. Principal Components Analysis

Our first main feature selection technique is Principal Components Analysis (PCA), which is an approach to encode most of the variability in a dataset with a smaller number of dimensions. PCA was first introduced by K. Pearson in 1901 [14] and finds many applications in machine learning and statistics [9]. Suppose that there are originally p features. The first principal component is the linear combination of the original features that has the largest variance. The second principal component is the linear combination that has the largest variance among all linear combinations that are not correlated with the first principal component; thus, it encodes most of the variance that is not explained by the first principal component. The third through the pth principal components are calculated similarly.

Since PCA can only be applied to numerical variables, we first use one-hot encoding to transform all categorical variables into numerical variables.

```r
# Create PCA recipe.
pca_recipe <- cleaned_data |>
  recipes::recipe(Recurred ~ .) |>
  recipes::step_dummy(recipes::all_nominal_predictors()) |>
  recipes::step_normalize(recipes::all_numeric_predictors()) |>
  recipes::step_corr(threshold = 0.6) |>
  recipes::step_pca(recipes::all_numeric_predictors())

# Estimate parameters from the `recipes` package.
pca_prep <- recipes::prep(pca_recipe)

# Extract the PCs.
extract_pcs <- parsnip::tidy(pca_prep, 4)
```

Figure 4 displays the top five features contributing to each of the first four principal components. The PCA results indicate that features such as Risk (Low, Intermediate), Response (Structural Incomplete), Adenopathy (No), History of Radiotherapy (Yes), Stage (IVB), Pathology (Papillary), and T Stage (T2, T3a) are significant for the first three principal components.

In the next step, we visualize the first two dimensions of the PCA analysis. Figure 5 shows partial separation between recurrence statuses along the first two principal components, with some clustering patterns observed. However, there is some overlap between ‘Yes’ and ‘No’ groups, suggesting that these two components alone may not fully differentiate recurrence status.

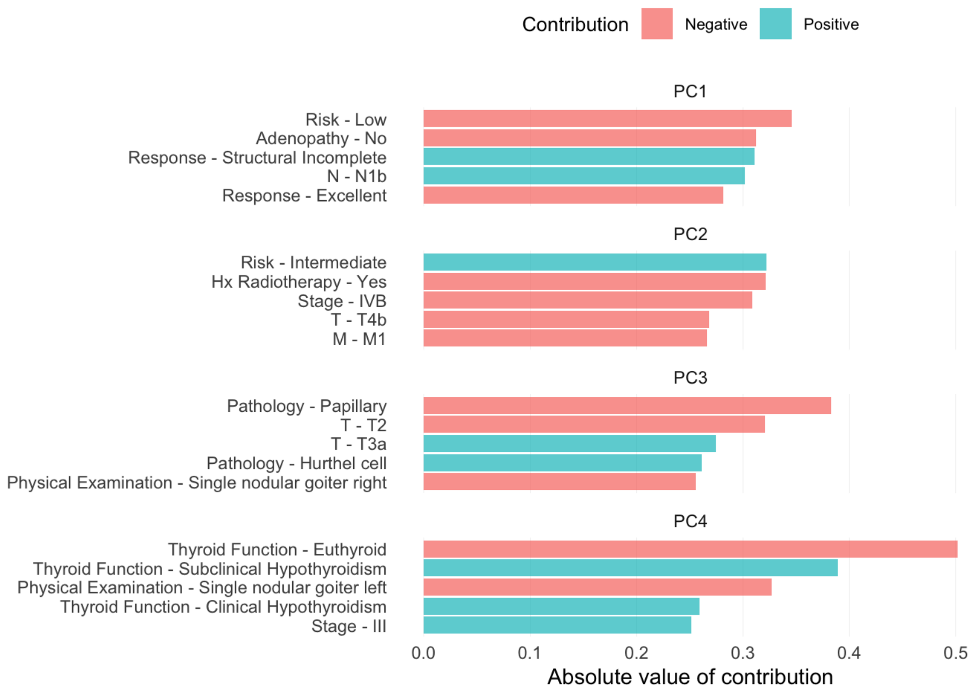

Figure 4: Top features for the first three principal components.

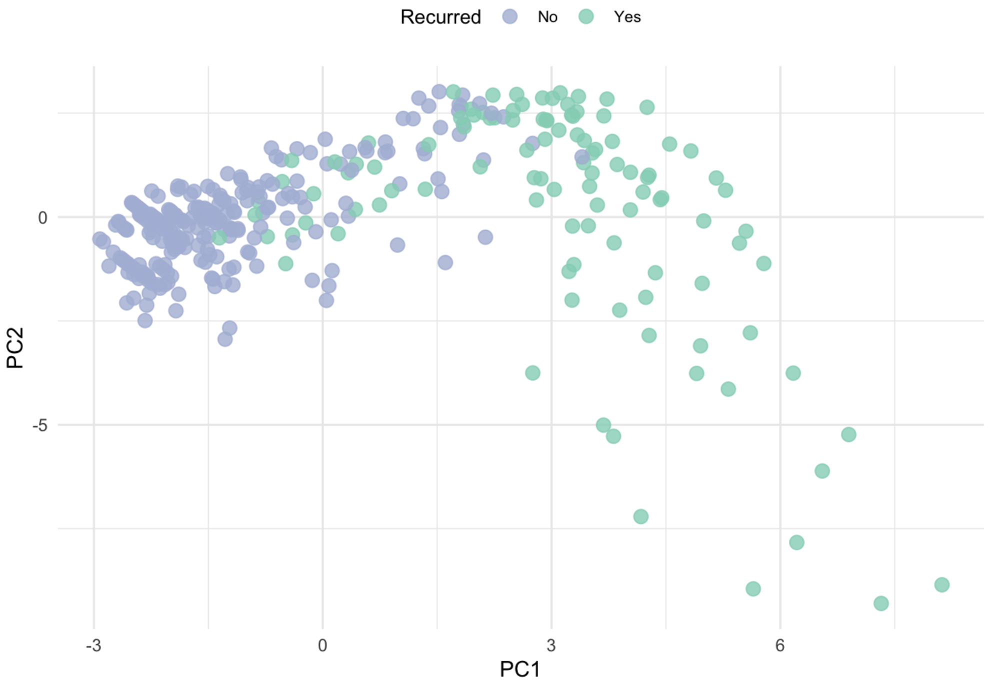

Figure 5: First two principal components.

5.2. Factor Analysis of Mixed Data

Another feature selection technique is Factor Analysis of Mixed Data (FAMD), a hybrid of Principal Component Analysis (PCA) and Multiple Correspondence Analysis (MCA) [6]. While PCA is directly applicable only to datasets with numerical features, and MCA is suitable for datasets with categorical features, FAMD can be applied to datasets containing both numerical and categorical features.

Suppose that the dataset includes K numerical features $(k_1, \ldots, k_K)$ and Q categorical features $(q_1, \ldots, q_Q)$. For any linear combination z of features, we use $r^2(z, k_i)$ to denote the squared correlation coefficient between z and k_i, and $\eta^2(z, q_i)$ to denote the squared correlation ratio between z and q_i.

PCA on the K quantitative features seeks for the linear combination of features that yields the maximal $\sum_{i=1}^{K} r^2(z, k_i)$; MCA on the Q qualitative features seeks for the linear combination of features that yields the maximal $\sum_{i=1}^{Q} \eta^2(z, q_i)$; thus, FAMD on all $K + Q$ features seeks for the linear combination of features that yields the maximal $\sum_{i=1}^{K} r^2(z, k_i) + \sum_{i=1}^{Q} \eta^2(z, q_i)$.

We start by separating the data into categorical and numerical variables, then apply FAMD using the `PCAmix` function from the `PCAmixdata` R package.

```r
split_data <-
  cleaned_data |>
  dplyr::select(-Recurred) |> # Remove targeting feature.
  PCAmixdata::splitmix()

PCA_mix_result <- PCAmixdata::PCAmix(
  X.quanti = split_data$X.quanti, # Categorical variables.
  X.quali = split_data$X.quali, # Numerical variables.
  rename.level = TRUE,
  graph = FALSE
)
```

The `PCAmix` output shows that the first three FAMD dimensions contribute 26.17% of the total variance (see Table 9).

Table 9: Proportion of variance explained by the first three dimensions of FAMD

Dimension	Proportion	Cumulative
Dimension 1	13.85%	13.85%
Dimension 2	7.41%	21.26%
Dimension 3	4.91%	26.17%

Figure 6 shows the distribution of data points along the first two principal components (i.e., Dimension 1 and Dimension 2), color-coded by the recurrence status. It indicates that the first two dimensions capture some of the variance associated with recurrence status, though there is also a fair overlap. Similar results can be observed in Figure 7, which shows the distribution of data points along the first three principal components (i.e., Dimension 1, Dimension 2, and Dimension 3), color-coded by the recurrence status. In addition, Figure 8 shows the top contributors to each of the first four dimensions of FAMD.

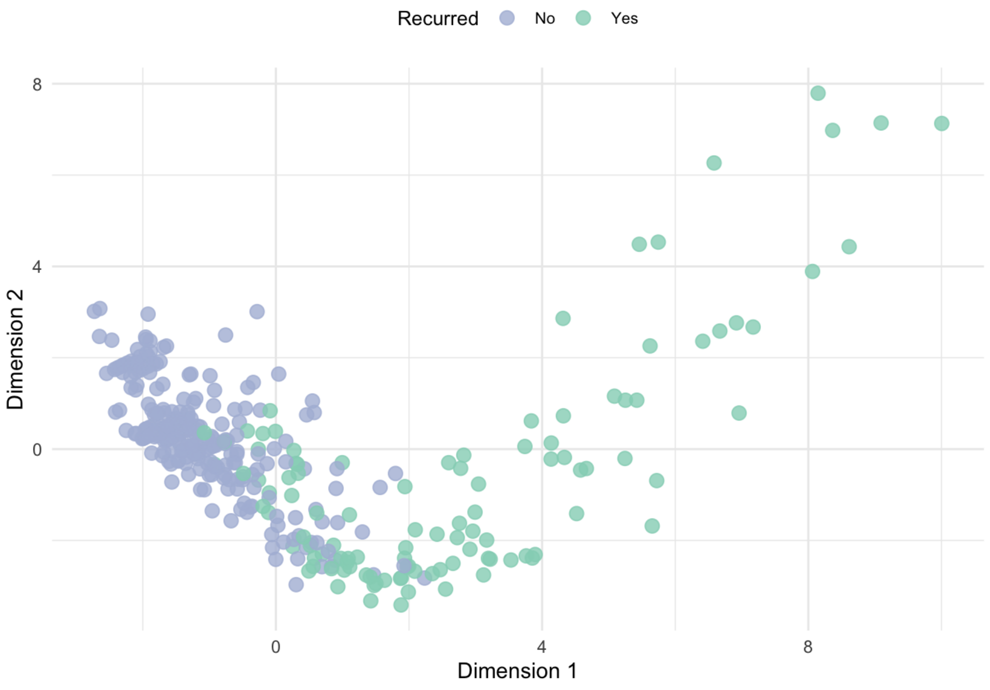

Figure 6: First two dimensions of the `PCAmix` analysis.

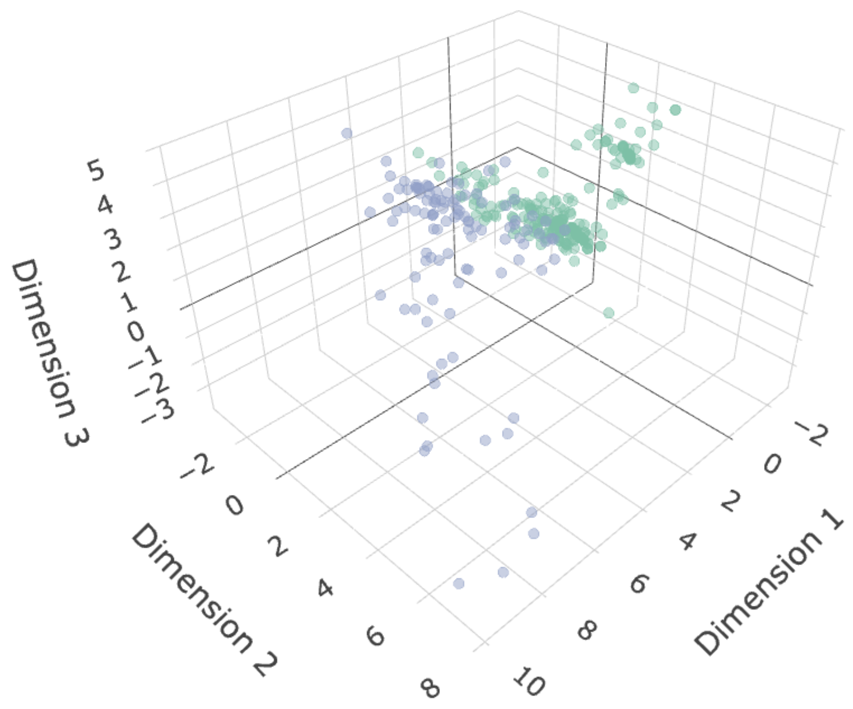

Figure 7: First three dimensions of the PCAmix results by recurrence status.

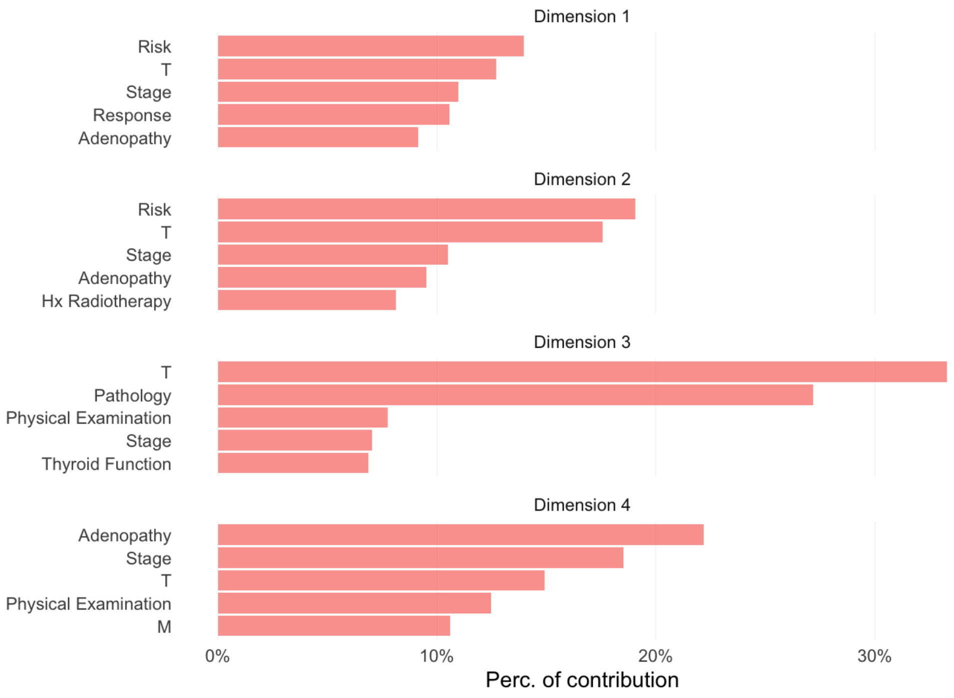

Figure 8: Top contributors to the first four dimensions of FAMD.

These plots suggest that, overall, Risk, T, Stage, and Adenopathy are significant features.

5.3. Feature Importance Analysis

The third feature selection technique we will apply is Feature Importance Analysis (FIA). We use FIA because it helps us understand *why* a model makes its predictions. Unlike PCA and FAMD, which reduce dimensionality by transforming features into uncorrelated components that capture the most variance in the data, FIA assesses the contribution of individual features to a model's predictions. Moreover, FIA is useful for model improvement: by identifying the most impactful features, it allows us to focus our efforts on the data that truly matters. Additionally, FIA aids in detecting overfitting: features with surprisingly high importance might indicate that the model is latching onto irrelevant details in the training data that do not generalize well to unseen data. By identifying such features, we can potentially adjust the model to reduce overfitting [16]. We use FIA to better understand the random forest model, one of the better-performing models on the DTC dataset. Specifically, we employ two types of FIA methods: Impurity Importance and Permutation Importance.

Impurity Importance

Impurity importance, also known as Mean Decrease in Impurity (MDI) or Gini importance, measures the importance of a feature based on the total reduction of the criterion (impurity) brought by that feature. It is not very computationally expensive and gives an importance score for each feature as part of the tree-building process. However, it may be biased towards features with more categories or continuous features. The idea is the following: let our random forest consist of T decision trees. For each tree t, we consider every node j where a split is made on feature f. We need the impurity function I, which measures the quality of a split at a node. It quantifies how mixed the classes are (for classification tasks) or the variance within the node (for regression tasks). Lower impurity indicates purer nodes. We introduce the following variables:

- $I(p(j))$ is the impurity of the parent node
- $I(l(j))$ and $I(r(j))$ are the impurities of the left and right child nodes, respectively.
- $w_{l(j)}$ and $w_{r(j)}$ are the proportions of samples in the left and right child nodes, respectively.

We now calculate the decrease in impurity for the node j as

$$\Delta I_j = I(p(j)) - [w_{l(j)} I(l(j)) + w_{r(j)} I(r(j))].$$

For each feature f, we sum the impurity decreases for all nodes where f is used for splitting. Call the set of such nodes inside a tree t as $n(t, f)$. We then normalize the sum. The result is the following:

$$\text{MDI}(f) = \frac{1}{T} \sum_{t=1}^{T} \sum_{j \in n(t,f)} \Delta I_j.$$

This gives us our impurity importance.

Permutation Importance

Permutation importance, also known as Mean Decrease in Accuracy (MDA), measures the importance of a feature by evaluating the decrease in the model's performance when the feature's values are randomly shuffled. It provides a more unbiased estimate of feature importance by directly assessing the impact of feature shuffling on model performance. It can be more computationally intensive since it requires model evaluation on permuted datasets. The idea is as follows: We let M be the baseline performance metric, i.e. the model's performance on a validation set. Then, for each feature f, we create a permuted version of the validation set by randomly shuffling the values of f while fixing the other feature values. We then evaluate the model's performance on this permuted set to get a new metric M_f. The permutation importance of a feature f is then simply the decrease in performance metric:

$$\text{MDA}(f) = M - M_f.$$

We will use accuracy for our performance metric. In this case, a higher MDA value indicates that the feature is more important (because permuting it significantly reduces the model's accuracy).

Analysis

Let us first compute the impurity importance. In order to do this, we need to redefine our model workflow and tune our new model. The outptut of the analysis can be see in Figure 9.

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

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

```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.
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)

# Extract workflow from fitted model
ip_plot <- extract_workflow(rf_final_fit) |>
  extract_fit_parsnip(final_forest) |>
  vip::vip(num_features = 16)

# Stop clusters.
parallel::stopCluster(clusters)

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

```
10.874 sec elapsed
```

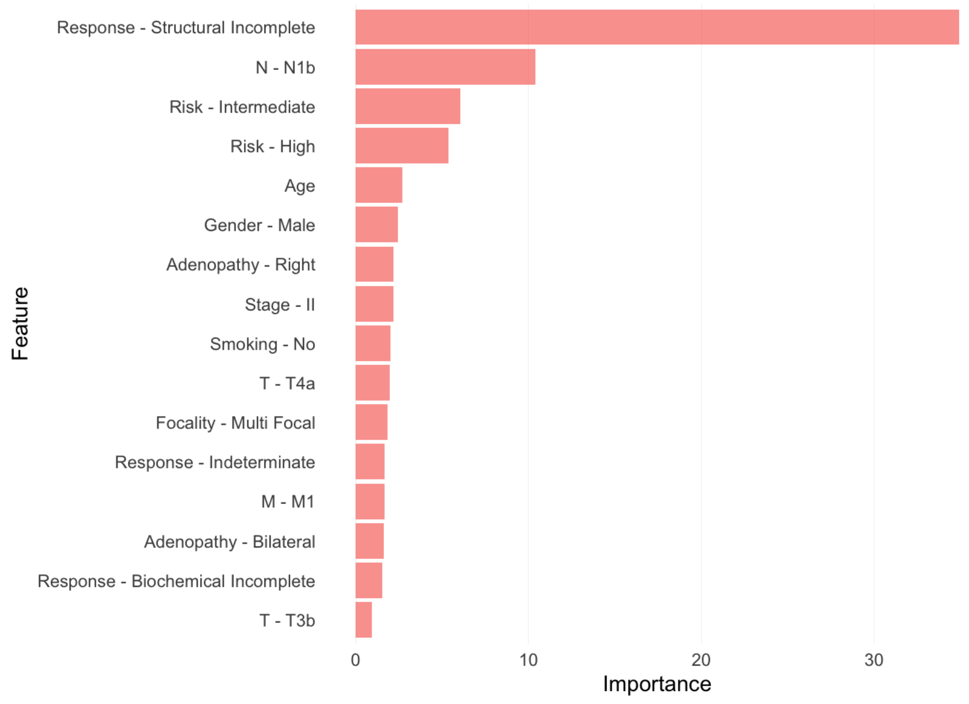

Figure 9: Feature analysis by impurity importance.

Now let us do the same for permutation importance. The output of the analysis can be seen in Figure 10.

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

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

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

#' Start timer.
tictoc::tic()

# Create and register clusters.
```

```r
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)

# Extract workflow from fitted model
pi_plot <- extract_workflow(rf_final_fit) |>
  extract_fit_parsnip(final_forest) |>
  vip::vip(num_features = 16)

# Stop clusters.
parallel::stopCluster(clusters)

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

```
11.814 sec elapsed
```

Results

We notice a very interesting phenomenon, confirmed by both of the plots: the `Response - Structural Incomplete` feature plays a much bigger part in the role of the prediction of our random forest model than any of the other features. Medically, this means that our model is eager to predict DTC when there is evidence of cancer cells upon imaging, but no detectable thyroglobulin (a protein produced by the thyroid gland). Intuitively, this should indeed correspond to high risk of DTC. However, the nontrivial result that these plots uncover is that this feature is significantly more important than other important features identified in our EDA (e.g. the risk assessment). The next most important features seem to be the `N - N1b` feature and the risk assessment.

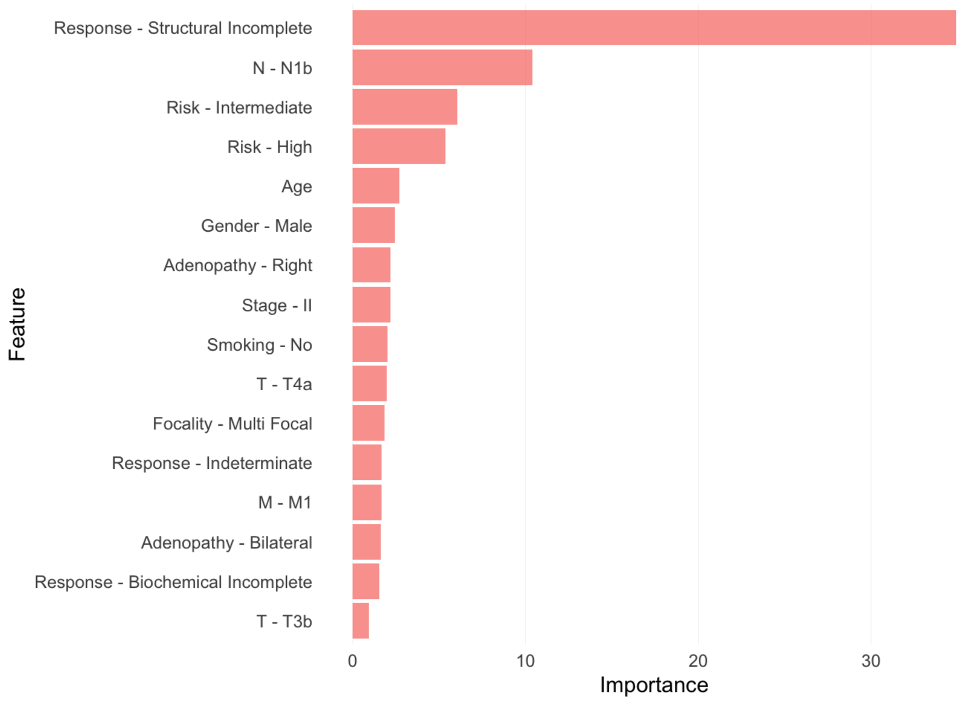

Figure 10: Feature analysis by permutation importance.

5.4. Combined Results of PCA, FAMD, and FIA

Comparing the results of PCA/FAMD and FIA, the following features (out of the sixteen total features) are the most important: Response, N, T, Risk, and Age. In the next section, we will retrain our six machine learning models on this reduced feature space and compare the resulting performance with the performance on the entire feature space.

6. 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 first evaluate the performance of our six models that use the entire feature space via the metrics shown in Table 10.

The Random Forest model emerged as the top performer on the test set in terms of overall accuracy and specificity, achieving a 94% accuracy rate and a 94% specificity rate, indicating its robustness

Table 10: Performance Comparison: Accuracy, Precision, Recall, and Specificity

Metric	Model	Value
Accuracy	SVM-poly, Random Forest	94%
Precision	SVM-poly, ANN, Logistic Regression, Random Forest	85%
Recall	SVM-rbf	100%
Specificity	SVM-poly, Random Forest	94%

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.

We next compare the performance of our six models when trained on the subset of features identified by PCA, FAMD, and FIA: Response, N, T, Risk, and Age [see Table 11]. The training and testing procedures remain the same, but the models are now trained on the reduced feature space.

Table 11: Performance Comparison (with PCA/FAMD/FIA): Accuracy, Precision, Recall, and Specificity

Metric	Model	Value
Accuracy	ANN, Logistic Regression	95%
Precision	SVM RBF, SVM Poly, ANN, Logistic Regression	85%
Recall	ANN, Logistic Regression	96%
Specificity	ANN, Logistic Regression	94%

6.1. Model Performances

6.1.1. Correlation Matrix

We first examine how the predictions of our models vary across the test observations. We visualize the correlation between the prediction vectors of each pair of models using a heatmap [11].

We observe that the correlation between the SVM predictions and the rest of the models is notably lower. This indicates that SVMs approach to predicting DTC differs from the other models.

Notice that this is closely related to SVM having a recall of 100%, but an extremely low precision of 23.1%. On the test set, the SVM model predicts all the true positives but also has a large false positive rate of 23.5%. In comparison to the predictions of the other models, we notice that there is not a single test case for which SVM predicts negative but another model predicts positive – all disparities are where SVM predicts positive but other models predict negative. For each model, there are between 14 and 20 test cases out of the 91 total test cases for which SVM predicts positive but the other model predicts negative.

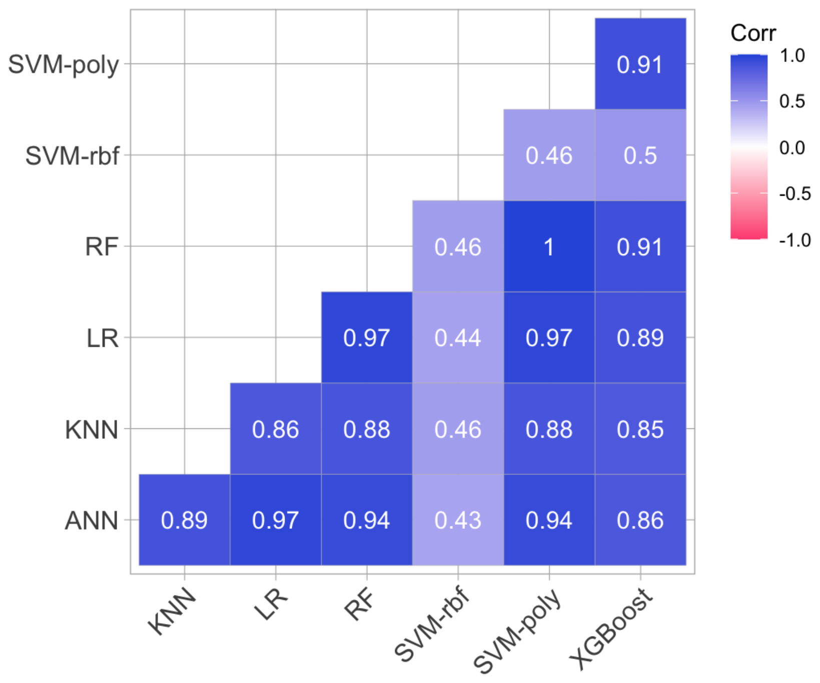

Figure 11: Correlation heatmap of model predictions.

We will explore this further in the individual model analysis section.

6.1.2. Bayesian Model Comparison

We now delve deeper into the comparison of model performances to understand if the apparent best-performing models (RF and SVM with the polynomial kernel) are truly superior. We leverage the resampling results (cross-validation performance metrics) obtained during model tuning to approximate their performance in a Bayesian framework. While not identical to test performance, we expect these resampling metrics to provide a reasonable estimation.

Bayesian model comparison allows us to draw more intuitive and practical inferences than traditional frequentist analysis. Let's briefly outline the mathematical underpinnings.

We employ a Bayesian ANOVA model to generate distributions of the metric estimate for each model. A standard ANOVA model (expressed as a linear regression) for model comparison would be:

$$y = \beta_0 + \beta_1 m_1 + \cdots + \beta_k m_k,$$

where y is the metric to be predicted, β_0 denotes the metric estimate for the base model (any model can be chosen), m_i is an indicator variable for model i, and β_i represents the difference in metric estimate between model i and the base model. We aim to assess whether any of the β_is is 0.

To account for potential dependencies between metric outputs and individual resamples (resample-to-resample effect), we modify the model equation:

$$y = \beta_{i0} + \beta_1 m_1 + \cdots + \beta_k m_k.$$

Note that only the intercept term β_{i0} varies across resamples, assuming that the differences in metric estimates (compared to the base model) remain consistent across resamples.

In the Bayesian context, each β_i has a prior distribution representing our initial beliefs. We use default uninformative priors. As data is processed, these prior distributions are updated based on likelihood estimates, resulting in posterior distributions for each parameter. The `tidyposterior` library handles these calculations.

To formally compare posterior distributions, we examine the distribution of differences and calculate the Region of Practical Equivalence (ROPE). ROPE estimates the probability that two models are practically equivalent. We define a *practical effect size*, representing the difference in metrics we consider negligible. We then calculate the percentage of the difference distribution falling within this interval. A high percentage indicates that the models' performance in that metric is not practically different.

We compare our models using the following three metrics:

- **ROC-AUC:** Area under the Receiver Operating Characteristic curve, measuring the trade-off between true positive rate and false positive rate.
- **Accuracy:** Proportion of correctly classified observations.
- **Brier Score:** A cost function assessing the calibration of probabilistic predictions. It quantifies the mean squared difference between predicted probabilities and actual outcomes:

$$BS = \frac{1}{n}\sum_{i=1}^{n}(p(x_i) - y_i)^2,$$

  where $p(x_i)$ denotes the model's predicted probability of a positive class for observation x_i, y_i is the true class label for observation x_i, and n is the total number of observations. A lower Brier score indicates better calibration.

We proceed to compare each of these metrics, one at a time. Let us start by analyzing the ROC-AUC metric.

In Figure 12 we can see a comparison of the ROC-AUC metrics for each model per resampling fold.

Ideally, these lines should be parallel. While this is not the case here, the lines do appear to have a similar trend.

Next, Figure 13 and Figure 14 show the actual posterior distributions for the ROC-AUC metric of each model.

The Random Forest model appears to have the highest ROC-AUC metric, though ANN, SVM-rbf, SVM-poly, and LR all appear close seconds.

Precisely, Table 12 illustrates the ROPE measures of each pairwise model comparison where we use a practical effect size of 0.02 (ie. the probability that the difference in ROC-AUC of two models is at most 0.02).

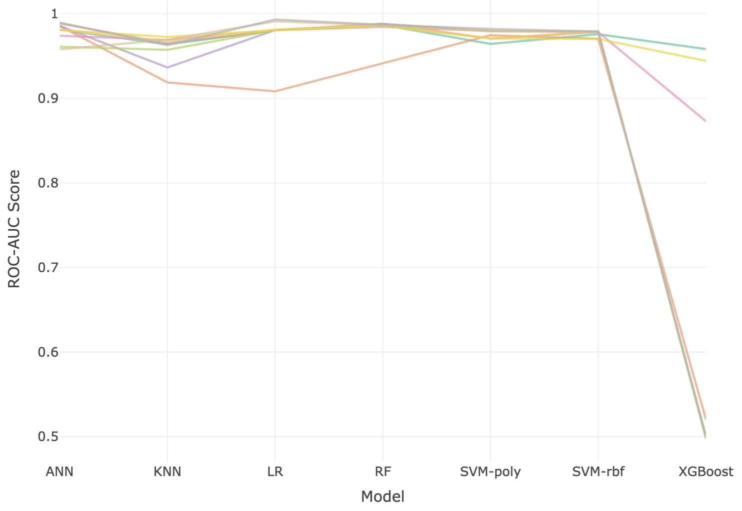

Figure 12: ROC-AUC scores for each model per resampling fold.

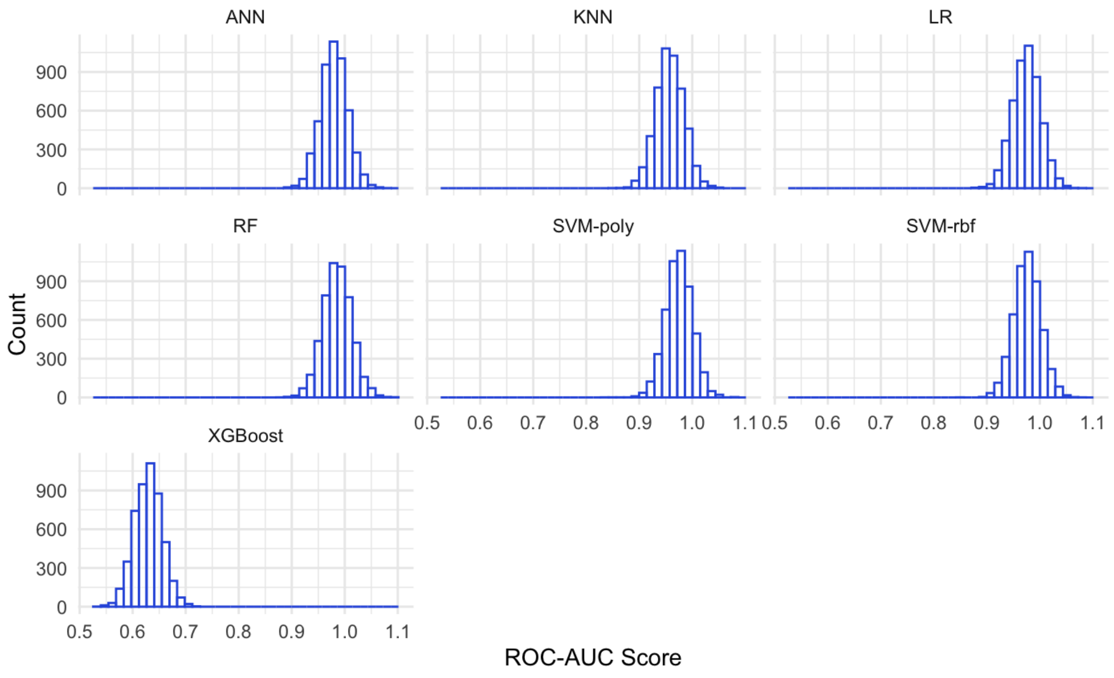

Figure 13: Posterior distributions of ROC-AUC scores for each model.

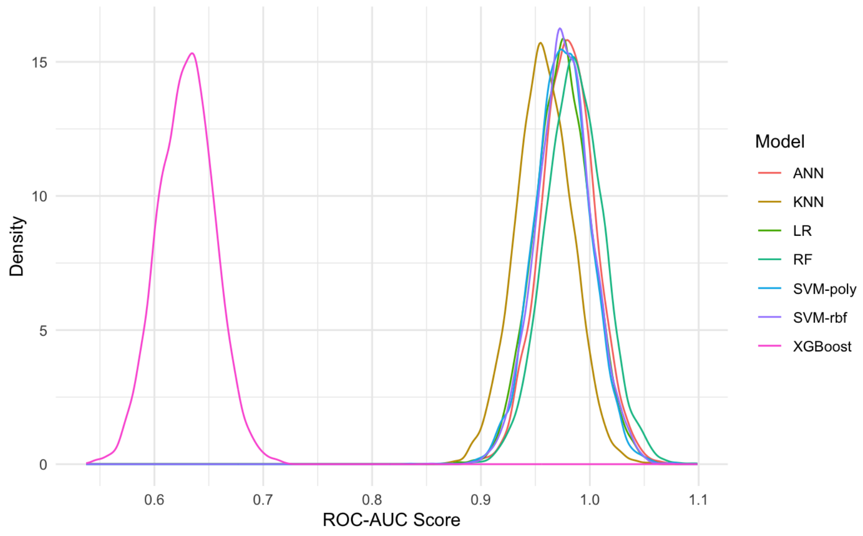

Figure 14: Comparison of ROC-AUC score distributions across models.

Table 12: ROPE Measure for Each Pairwise Model (ROC-AUC)

Contrast	Practical (-)	Practical (=)	Practical (+)
ANN vs RF	35.24%	41.06%	23.7%
KNN vs RF	58.6%	32.62%	8.78%
LR vs RF	39.82%	40.98%	19.2%
RF vs XGBoost	0%	0%	100%
SVM-poly vs RF	39.98%	40.74%	19.28%
SVM-rbf vs RF	38.16%	41.28%	20.56%

In the ROPE values frame, we see that the Random Forest appears much better than the XGB model – though the results are less conclusive in comparison to the ANN, KNN, LR, SVM-rbf, SVM-poly models.

As a secondary test, we also run the analysis using the accuracy metric. The process is identical, and in Figure 15 we can see the posterior distributions of the accuracy of all models.

Similarly as before, in Table 13 we can see the ROPE measure of each pairwise model comparison using a practical effect size of 0.02.

Table 13: ROPE Measure for Each Pairwise Model (Accuracy)

Contrast	Practical (-)	Practical (=)	Practical (+)
ANN vs RF	36%	46.54%	17.46%
KNN vs RF	82.42%	16.28%	1.3%
LR vs RF	31.76%	47.88%	20.36%
RF vs XGBoost	0%	0%	100%
SVM-poly vs RF	54.28%	36.64%	9.08%
SVM-rbf vs RF	100%	0%	0%

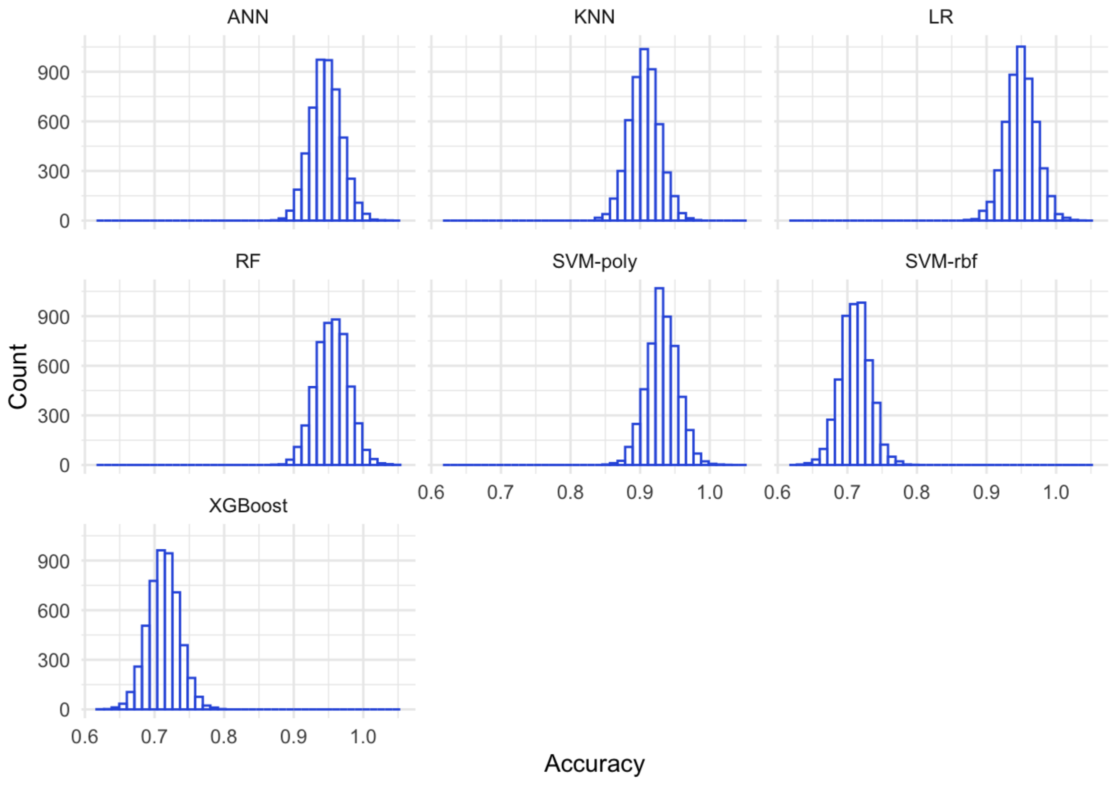

Figure 15: Posterior distributions of the models' accuracy.

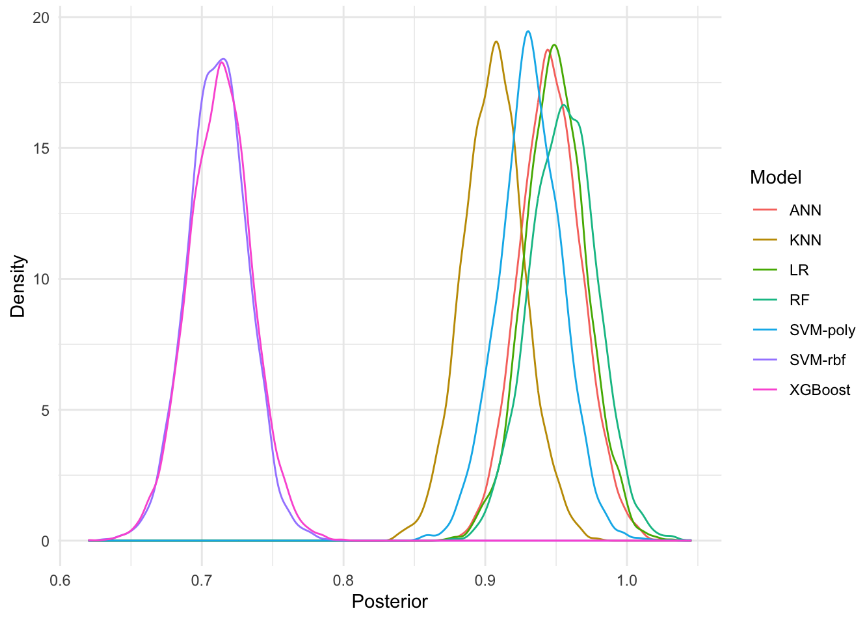

Figure 16: Comparison of the accuracy distributions across models.

In the accuracy metric, we see that Random Forests largely surpasses the KNN, SVM-rbf, and XGBoost models – but the results are again less conclusive for the ANN, LR, and SVM-poly models.

Lastly, we look at the Brier score posterior distributions of the models in [16].

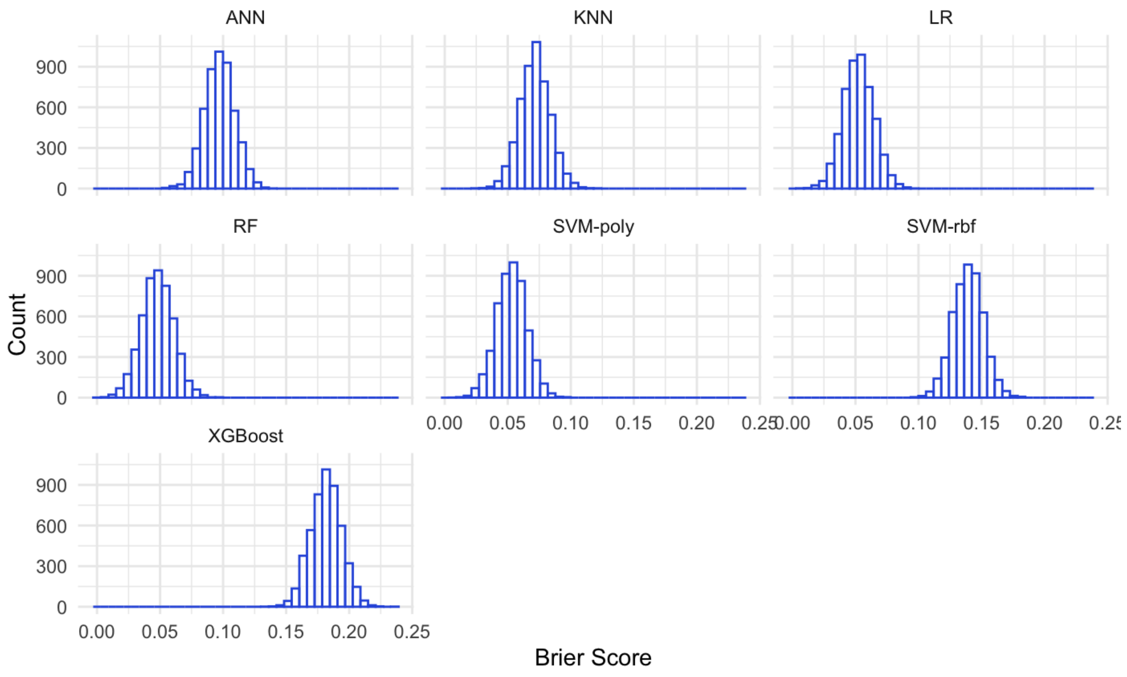

Figure 17: Brier score posterior distributions for each model.

The Random Forest distribution is the one to the far left. RF appears largely better than ANN, SVM, and XGBoost models in the Brier score – though the model appears comparable to LR and SVM-poly. The ROPE probabilities are in Table 14.

Table 14: ROPE Measure for Each Pairwise Model (Brier)

Contrast	Practical (-)	Practical (=)	Practical (+)
ANN vs RF	0%	4.02%	95.98%
KNN vs RF	0.68%	40.14%	59.18%
LR vs RF	7.76%	73.98%	18.26%
RF vs XGBoost	100%	0%	0%
SVM-poly vs RF	6.96%	74.24%	18.8%
SVM-rbf vs RF	0%	0%	100%

Indeed, we see that RF is practically equivalent to LR and SVM-poly with high probability in the Brier Loss, and largely superior to the ANN, SVM-rbf, and XGBoost models – though the results are less conclusive for KNN.

Hence, RF largely surpasses KNN, SVM-rbf, ANN, and XGBoost in at least one metric, but not LR or SVM-poly. This suggests that RF, LR, SVM-poly are likely comparable in predicting DTC recurrence (given the practical effect size) despite the slightly differing test metrics – though it is

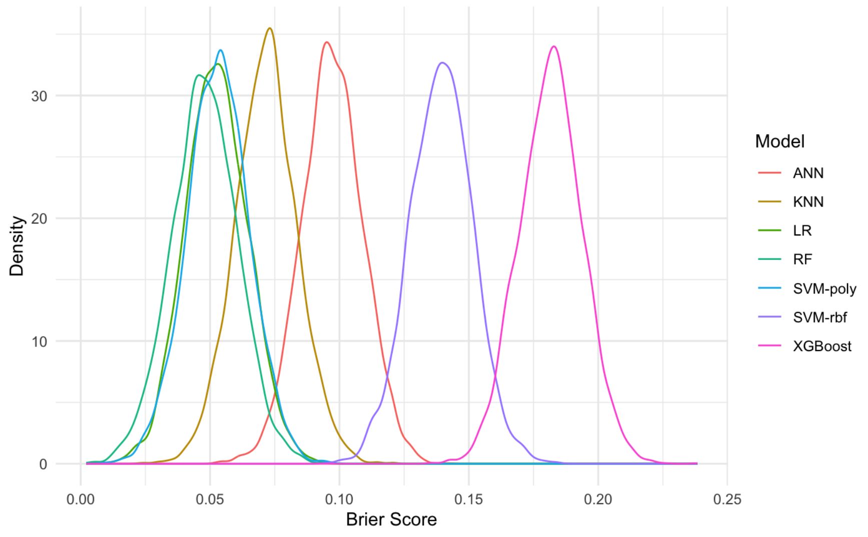

Figure 18: Posterior distributions of Brier losses for each model.

worth noting that RF did surpass all the other models in the posterior distribution itself for all metrics.

7. 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.

8. INFORMED CONSENT

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

9. INSTITUTIONAL REVIEW BOARD STATEMENT

Not applicable.

10. FUNDING

There was no funding for conducting this study.

11. DATA AVAILABILITY STATEMENT

The data is available at the UC Irvine Machine Learning Repository [10]. For more information, please refer to [4].

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
- PCA: Principal Components Analysis
- FAMD: Factor Analysis of Mixed Data
- FIA: Feature Importance Analysis

14. ACKNOWLEDGMENTS

This paper is the result of a research collaboration made possible through MIT PRIMES. We are immensely grateful to the MIT PRIMES organizers for making possible the research collaboration that led to the present paper.

REFERENCES

[1] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, 2018.

[2] Anju and N. Sharma. Survey of boosting algorithms for big data applications. *International Journal of Engineering Research and Technology (IJERT)*, 5, 04 2018.

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

[4] S. Borzooei and A. Tarokhian. Differentiated thyroid cancer recurrence, 2023.

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

[6] M. Chavent, V. Kuentz, A. Labenne, and J. Saracco. Multivariate analysis of mixed data: The r package pcamixdata. *Electronic Journal of Applied Statistical Analysis*, 15(3):606–645, 2022.

[7] R. Genuer, J.-M. Poggi, and C. Tuleau-Malot. Variable selection using random forests. *Pattern Recognition Letters*, 31(14):2225–2236, 2010.

[8] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning*. Springer Texts in Statistics, 2021.

[9] Jolliffe, Ian T. Principal Component Analysis. *Springer Series in Statistics, Springer-Verlag*, 2002.

[10] R. Kelly, Markelle andLongjohn and K. Nottingham. The uci machine learning repository.

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

[12] M. Maalouf. Logistic regression in data analysis: An overview. *International Journal of Data Analysis Techniques and Strategies*, 3:281–299, 07 2011.

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

[14] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(11):559–572, 1901.

[15] G. Pellegriti, F. Frasca, C. Regalbuto, S. Squatrito, and R. Vigneri. Worldwide increasing incidence of thyroid cancer: update on epidemiology and risk factors. *Journal of Cancer Epidemiology*, 2013:1–10, 6 2013.

[16] E. Scornet. Trees, forests, and impurity-based variable importance in regression. In *Annales de l'Institut Henri Poincare (B) Probabilites et statistiques*, volume 59, pages 21–52. Institut Henri Poincaré, 2023.

[17] P. K. Syriopoulos, S. B. Kotsiantis, and M. N. Vrahatis. Survey on knn methods in data science. In D. E. Simos, V. A. Rasskazova, F. Archetti, I. S. Kotsireas, and P. M. Pardalos, editors, *Learning and Intelligent Optimization*, pages 379–393, Cham, 2022. Springer International Publishing.