Predictive Modeling of Patient Outcomes in Liver Cirrhosis Using Clinical and Biochemical Data

PH 3022 - Machine Learning and Neural Computation

Group Members

Rishmika Wijewardhana -s16079

Thiwanka Wimalasena – s16080

Kanishka Wimalasooriya -s16128

Duvindu Ushan – s16009

Table of Contents

1 ABSTRACT

A comprehensive machine learning framework was developed for predicting patient outcomes in liver cirrhosis using clinical and biochemical data from 418 patients. The dataset underwent extensive preprocessing—including duplicate removal, outlier elimination with the IQR meth-od, missing value imputation, and conversion of categorical variables through one-hot and label encoding—with standard normalization applied to ensure consistency and data integrity. Among the diverse set of features, laboratory measurements such as Bilirubin, Albumin, and Prothrombin time, along with clinical indicators like Ascites, Hepatomegaly, and Edema, were identified as the most influential in determining disease progression and overall prognosis. Four predictive models—Support Vector Machine (SVM), Random Forest, Deep Neural Net-work (DNN), and Extreme Gradient Boosting (XGBoost)—were implemented and optimized using techniques including GridSearchCV, RandomizedSearchCV, and Bayesian Optimization. While the SVM and Random Forest achieved moderate to strong generalization with accuracies of 88.66% and 95.3% respectively, and the DNN achieved 92% accuracy despite challenges in handling underrepresented classes, the XGBoost model emerged as the best performer. With an outstanding accuracy of 98% and a weighted F1-score of 0.98, XGBoost exhibited minimal overfitting and excellent interpretability through feature importance analysis, making it the op-timal candidate for incorporation into clinical decision support systems for liver cirrhosis prognostication.

2 INTRODUCTION

Liver cirrhosis is a long-term liver disease that occurs when healthy liver tissue is gradually replaced by scar tissue, which hinders the organ's ability to function normally. In Sri Lanka, liver cirrhosis has become a significant public health concern, with local studies suggesting that a notable number of patients admitted to hospitals for liver-related issues are diagnosed with this condition. Common causes of liver cirrhosis include excessive alcohol consumption, chronic infections from hepatitis B and hepatitis C viruses, and, increasingly, non-alcoholic fatty liver disease linked to unhealthy dietary habits and sedentary lifestyles. In addition, environmental factors and certain toxins have also been implicated as contributing causes in some regions.

The disease often presents a range of symptoms, such as persistent fatigue, jaundice (yellowing of the skin and eyes), abdominal swelling due to fluid accumulation (ascites), loss of appetite, and unexplained weight loss. As the condition advances, patients may also experience confusion, bleeding tendencies, and other severe complications. The progression of these symptoms not only decreases the quality of life but also significantly impacts patient outcomes, which are typically recorded in the dataset as the "Status" of the patient—indicating whether the patient is alive, has received a liver transplant, or has passed away.

In This project, we aim to build a predictive model using Python's Pandas library for data processing and analysis. After cleaning the dataset, which contains clinical and biochemical data from 418 patients. Data preprocessing involves handling missing values, removing duplicate records, and transforming the variables into a format suitable for analysis. Once the data is clean and organized, we use it to train a machine learning model that predicts patient outcomes based on their clinical profiles. This model not only supports healthcare professionals in making more informed treatment decisions but also highlights the critical need for early detection and preventive strategies, such as reducing alcohol intake, getting vaccinated against hepatitis, and adopting healthier lifestyles.

3 Methodology

3.1 Description of the Attributes in dataset

At first a good understating about the Feature Variables and the target variable. Brief description about the variable are shown below table.

Column Name	Description
ID	Unique identifier for each patient record.
N_Days	Number of days from registration until the event (death or transplant).
Status	Target variable indicating outcome: C (Censored/Alive), CL (Censored Liver), D (Died).
Drug	Drug used in treatment, typically D-penicillamine.
Sex	Gender of the patient (M or F).
Age	Age of the patient at the time of registration (in days).
Ascites	Presence of fluid in the peritoneal cavity, scored as Yes or No.
Hepatomegaly	Liver enlargement (Yes/No).
Spiders	Spider angiomas, a symptom of liver disease (Yes/No).
Edema	Fluid retention score: 0 (no edema), 0.5 (edema but no diuretics), 1 (edema despite diuretics).
Bilirubin	Total bilirubin level in mg/dL.
Cholesterol	Serum cholesterol level in mg/dL.

Albumin	Serum albumin level in g/dL.
Copper	Serum copper concentration in µg/dL.
Alk_Phos	Alkaline phosphatase level in IU/L.
SGOT	Serum glutamic-oxaloacetic transaminase level in IU/L.
Tryglicerides	Serum triglyceride level in mg/dL.
Platelets	Blood platelet count (per mm^3).
Prothrombin	Prothrombin time in seconds, related to blood clotting.
Stage	Histological stage of disease (ranging from 1 to 4).

Some additional columns such as Histologic (which likely refers to histological score), Tryglicerides, and others provide supporting medical information about the state of the disease and organ function.The dataset, which is derived from clinical records of liver cirrhosis patients, encompasses a wide range of attributes that are critical for both clinical assessments and machine learning applications. Each feature included in the dataset provides unique insight into the patient's condition. For instance, laboratory values like Bilirubin and Albumin are indispensable in evaluating liver functionality, as they reflect the organ's capacity to metabolize and synthesize essential compounds. Similarly, the Prothrombin time is a key indicator of the liver's ability to produce proteins necessary for blood clotting, and abnormal values here can signal severe liver damage.

Moreover, physical symptoms such as *Ascites*, *Hepatomegaly*, and *Spiders* are observable clinical signs used by hepatologists to assess the stage and severity of liver cirrhosis. These features not only support diagnostic decisions but also contribute to the evaluation of disease progression. The *Edema* score and *Platelet* count further complement this evaluation by indicating potential complications such as fluid retention and portal hypertension.

Central to the predictive modeling task is the *Status* column, which serves as the target variable. This column categorizes the patient outcomes into three distinct classes:

1) **C (Censored/Alive)**: Patients who were alive at the last recorded follow-up,
2) **CL (Censored Liver)**: Patients who underwent a liver transplant, and
3) **D (Deceased)**: Patients who have passed away.

The ability to accurately predict the Status based on the myriads of clinical and biochemical features is the foundation of the machine learning pipeline used in this study.

In addition to the clinical insights offered by these features, the dataset exhibits real-world complexity through the presence of missing data. Attributes such as *Cholesterol*, *Copper*, and

Prothrombin may contain gaps that can result from various hospital constraints or patient-specific factors. Addressing these missing values—either through imputation techniques or sophisticated model-based approaches—is critical to ensuring the robustness and reliability of the predictive models.

Overall, the comprehensive nature of this dataset, with its diverse range of clinical indicators and detailed patient information, provides a robust basis for developing machine learning models. By leveraging Python's Pandas library for data pre-processing, the methodology ensures high-quality, well-structured data, facilitating accurate predictions of patient outcomes. This, in turn, supports better clinical decision-making and a deeper understanding of liver cirrhosis progression.

3.2 Graphical representation about the data

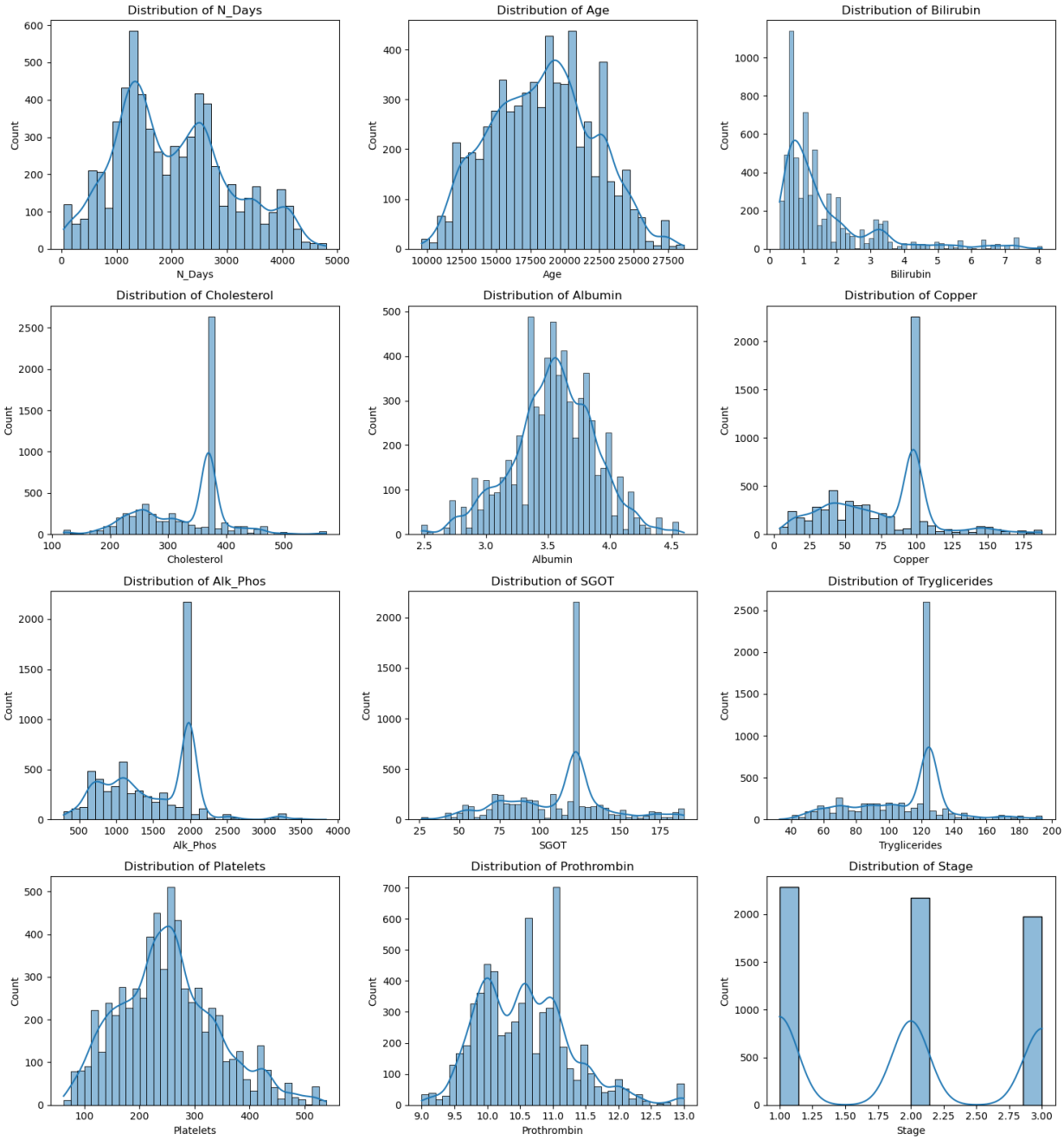

Figure 1: Distribution of the diseases

3.2 Data Acquisition and Preprocessing

3.2.1. Data Loading and Initial Cleaning

The dataset was acquired in CSV format and contains a wide range of clinical and biochemical features. The initial step involved importing the data into a Pandas DataFrame using the `pd.read_csv(filepath)` function. During this phase, the data was also standardized by converting all string fields to lowercase and stripping extra white spaces, ensuring consistency across all textual entries. This standardization reduces potential mismatches in categorical values arising from varied capitalization or formatting differences.

```
df = pd.read_csv(filepath)
df_clean = df.drop_duplicates()
df_clean = df_clean[(df_clean[col] >= Q1 - 2 * IQR) & (df_clean[col] <= Q3 +
2 * IQR)]
```

3.2.2 Duplicate Removal

To guarantee that the dataset reflected only unique patient records, duplicates were identified and removed. Duplicate records can lead to biased model learning by overrepresenting particular entries, so using the `df.drop_duplicates()` method ensured a cleaner, more representative dataset for subsequent analysis.

```
# Remove duplicate entries
df_clean = df.drop_duplicates()
```

3.2.3 Outlier Elimination

Outlier detection and removal are critical in reducing skewness and preserving model robustness. Numerical variables were scrutinized for outliers using the Interquartile Range (IQR) method. For each numerical feature, the first (Q1) and third (Q3) quartiles were computed, and the IQR (Q3-Q1) was determined. Data points outside the range defined by lower_bound Q1−1.5×IQR and Q3+1.5×IQR were considered outliers and were removed. This not only reduces the noise within the data but also minimizes the chances of the model being influenced by extreme values.

```
Q1 = df_clean[col].quantile(0.25)
Q3 = df_clean[col].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR
df_clean = df_clean[(df_clean[col] >= lower_bound) & (df_clean[col] <=
upper_bound)]
```

3.2.4 Handling Missing Values

Missing data is a common challenge in real-world datasets. In this dataset, missing values appeared in several key clinical attributes. The process to handle missing values was tailored based on the nature of the feature:

1) **Numeric Columns:** Missing values were imputed with the mean value of the column. This approach helps preserve the overall distribution of the data.
2) **Categorical/String Columns:** Missing values were replaced using the mode, ensuring that the most frequent category was used as a replacement.
   By addressing missing values in this manner, the dataset becomes more complete and suitable for reliable predictive modeling.

```
# Numeric columns: impute with mean
numeric_cols = df_clean.select_dtypes(include=[np.number]).columns
df_clean[numeric_cols] =
df_clean[numeric_cols].fillna(df_clean[numeric_cols].mean())

# Categorical columns: impute with mode
categorical_cols = df_clean.select_dtypes(include=['object']).columns
for col in categorical_cols:
    df_clean[col].fillna(df_clean[col].mode()[0], inplace=True)
```

3.2.5 Data Transformation

To prepare the dataset for machine learning algorithms, categorical features were transformed using one-hot encoding. This conversion enables the algorithm to interpret categorical variables as numerical values without assuming any inherent order among the categories. Furthermore, the target variable, **Status**, was label-encoded, making it easier to work with during model training. This step ensures that the dataset is fully numeric and can be efficiently processed by neural network models and other machine learning algorithms.

```
Data Transformation: One-hot encode categorical features and label-encode
target variable
df_transformed = pd.get_dummies(df_clean,
columns=categorical_cols.drop('status'), drop_first=True)
```

Removing duplicates ensures that the dataset does not contain redundant entries that could distort the analysis. **Eliminating outliers** minimizes the effect of extreme values, which could otherwise lead to model overfitting or incorrect parameter estimates. **Handling missing values** improves the completeness and reliability of the dataset, avoiding issues in downstream analyses. **Transforming categorical features** converts the dataset into a format compatible with machine learning algorithms, ensuring that all inputs are in a consistent, numerical format.

3.3 Model Architecture and Building

3.3.1 Neural Network Design -by Rishmika

3.3.1.1 model designing

The input layer was defined first to accept the feature set from the preprocessed data. It was created by specifying the shape that matches the number of features (input_shape_dim). This layer establishes the starting point of the network and ensures that the dimensions of the data are appropriately processed. For example:

```
model = keras.Sequential()
model.add(keras.layers.InputLayer(input_shape=(input_shape_dim,)))
```

Following the input layer, the hidden layers were added. A loop was employed to construct a variable number of hidden layers. Each hidden layer comprised a dense (fully connected) layer, where the number of neurons was chosen dynamically from a predefined set such as 32, 64, 128, or 256. The activation functions—selected between 'relu' and 'tanh'—provided non-linearity, which is crucial for learning complex patterns. In addition, a dropout layer was added right after each dense layer to randomly deactivate a fraction of neurons, thereby reducing overfitting during training. An excerpt of the hidden layer construction is given below:

```
num_layers = hp.Int('num_layers', 2, 5)
for i in range(num_layers):
    units = hp.Choice(f'units_{i}', [32, 64, 128, 256])
    activation = hp.Choice(f'activation_{i}', ['relu', 'tanh'])
    model.add(keras.layers.Dense(units=units, activation=activation))
```

Each hidden layer supports the network by extracting increasingly abstract representations of the data, which ultimately assists in the accurate classification of patient outcomes.

To complete the network design, an output layer was appended with a softmax activation function to manage the multi-class classification. Finally, the learning rate and optimizer were set as hyperparameters, making these choices part of the tunable configuration:

```
model.add(keras.layers.Dense(num_classes, activation='softmax'))
learning_rate = hp.Choice('learning_rate', [1e-2, 5e-3, 1e-3, 5e-4])
optimizer_choice = hp.Choice('optimizer', ['adam', 'rmsprop'])
```

3.3.1.2 Hyperparameter Tuning

Bayesian Optimization was used as the method for hyperparameter tuning, offering an efficient way to explore the search space of different model configurations. The process was facilitated by Keras Tuner, which allowed the definition of a search space inline within the model-building function. This tuning process not only varied the number of layers, neurons per layer, activation

functions, and dropout rates but also adjusted parameters such as the learning rate and choice of optimizer.

The tuner was configured to perform a series of trials—each representing a complete model configuration—while monitoring validation accuracy. Early stopping was integrated to halt poor configurations and reduce computational overhead. The tuner was set to conduct a maximum of 30 trials with two executions per trial to account for variability. An example snippet initializing the tuner is shown below:

```
tuner = kt.BayesianOptimization(
    hypermodel=build_model, objective='val_accuracy',
    max_trials=30, executions_per_trial=2,
    directory='my_dir', project_name='liver_cirrhosis_tuning'
)
```

Once the tuning process commenced, Keras Tuner executed multiple configurations by adjusting each hyperparameter—such as the number of hidden layers, number of units, activation types, and dropout rates. This dynamic search allowed the model to be optimized for performance on the validation set. Upon completion, the best set of hyperparameters was identified, which, for instance, might have included five layers with varied units and dropout rates, a learning rate of 0.001, and RMSprop as the optimizer. These choices were then applied to train the final model:

```
history = final_model.fit(X_train, y_train, epochs=50,
                          batch_size=16, validation_split=0.2)
test_loss, test_accuracy = final_model.evaluate(X_test, y_test)
```

By carefully tuning the hyperparameters using Keras Tuner, the final model achieved a validation accuracy of approximately 91.5%. This organized approach to both the design and tuning stages ensured that the network was well-adapted to the underlying complexity of the clinical data, thereby enhancing its predictive performance for patient outcomes.

3.3.2 Support Vector Machine Design -by Thivanka

3.3.2.1 model designing

The Support Vector Machine (SVM) model was selected for the liver cirrhosis classification task due to its robustness in handling high-dimensional and complex medical datasets. Its capacity to create optimal separating hyperplanes makes it suitable for distinguishing between multiple patient outcome classes, especially when data distributions are non-linear.

To begin with, the dataset was preprocessed by removing missing values, duplicates, and outliers. One-hot encoding was used to convert categorical variables into numerical form, and the target variable (`Status`) was label-encoded. Finally, `StandardScaler` was used to normalize the feature space, ensuring that each feature contributed equally to the classification model.

After preprocessing, the dataset was split into training and testing sets using an 80:20 ratio. Stratified sampling was employed during the split to preserve the class distribution across both sets. This is particularly important in medical datasets, which often have imbalanced class distributions.

The SVM model was designed to handle non-linearity in the data by employing kernel functions. Specifically, the Radial Basis Function (RBF) kernel was used, which can map input features into higher-dimensional space to find an optimal decision boundary. The core parameters of the model—C (regularization), gamma (kernel coefficient), and kernel type—were treated as tunable hyperparameters.

3.3.2.2 Hyperparameter Tuning

Hyperparameter tuning was conducted using GridSearchCV with 5-fold cross-validation to ensure robust performance. A defined grid of parameter values for C, gamma, and kernel was explored:

C: Regularization parameter controlling the trade-off between training error and testing error. A value of 10 was found optimal.
Gamma: Defines the influence of a single training point. A value of 0.1 was identified as effective in capturing non-linear relations.
Kernel: RBF kernel was selected due to its ability to handle complex data patterns efficiently.

```
param_grid = {
    'C': [10**x for x in range(-1, 3)],
    'gamma': [10**x for x in range(-1, 2)],
    'kernel': ['rbf', 'poly', 'sigmoid']
}
```

Then the cross validation was done by using Grid search method.

```
grid = GridSearchCV(SVC(), param_grid, refit=True, verbose=1, cv=5)
grid.fit(X_train_scaled, y_train)
```

The best model configuration discovered was:

$$\{'C': 10, 'gamma': 0.1, 'kernal': 'rbf'\}$$

Using this optimal configuration, the final SVM model was trained on the training set and evaluated on the test set. The performance metrics are summarized below:

3.3.3 Extreme Gradient Boosting (XGBoost) Design – by Duwindu

3.3.3.1 Model Designing

Extreme Gradient Boosting (XGBoost) was selected as the final model due to its superior accuracy and robust handling of structured clinical data. After completing data preprocessing steps—including duplicate removal, categorical mapping, label encoding, outlier filtering via IQR, and feature normalization using Min-Max scaling—the cleaned dataset was partitioned into training (80%) and testing (20%) sets. The target variable was defined as the `Status` column, representing the health outcome of the patient.

Before settling on XGBoost, multiple machine learning algorithms such as Logistic Regression, Decision Tree, Random Forest, and SVM were tested and compared. Among them, XGBoost yielded the highest performance with a classification accuracy of **97.52%** on the validation set, making it the preferred model for deployment.

The model was built using the `XGBClassifier` from the XGBoost library. A baseline configuration was initially created with standard parameters, which was later refined through hyperparameter tuning. The model fitting was performed on the training set as follows:

```
xgb_model = XGBClassifier(random_state=42)
xgb_model.fit(X_train, y_train)
```

3.3.3.2 Hyperparameter Tuning

To enhance performance, hyperparameter tuning was conducted using `GridSearchCV`, paired with **5-fold cross-validation** to ensure robustness and minimize overfitting. Several key hyperparameters were tuned to identify the optimal combination:

Hyperparameter	Description	Values Considered
n_estimators	Number of boosted trees to fit	50, 100, 150
max_depth	Maximum tree depth for base learners	3, 6, 10, 15
learning_rate	Step size shrinkage used in weight updates to prevent overfitting	0.01, 0.1, 0.2
subsample	Fraction of the training data randomly sampled for growing each tree	0.7, 0.8, 1.0

colsample_bytree	Fraction of features randomly sampled for each tree	0.7, 0.8, 1.0
gamma	Minimum loss reduction required to make a further partition on a leaf node	0, 0.1, 0.2

The optimal hyperparameters were identified by fitting the grid search on the training data and evaluating the accuracy metric:

```
grid_search = GridSearchCV(estimator=xgb_model, param_grid=param_grid,
                           cv=5, scoring='accuracy', verbose=1, n_jobs=-1)
grid_search.fit(X_train, y_train)
```

The final model was then trained using the best-found parameters and tested on the held-out test set. An accuracy of **97.96%** was achieved, confirming the model's excellent generalization. In addition to accuracy, high **precision** and **recall** scores were also reported, indicating reliable predictions across all classes.

3.3.4 Random Forest design -by Kanishka

3.3.4.1 Model designing

The Random Forest model was selected due to its robustness and capacity to handle high-dimensional data with complex relationships. The model was constructed using the RandomForestClassifier from the scikit-learn library, where the key design principle lies in building an ensemble of decision trees trained on bootstrapped samples of the dataset.

After doing data preprocessing. An 80-20 train-test split was applied to ensure unbiased evaluation.Each tree in the forest is built using a random subset of features and samples, promoting model generalization and reducing overfitting. The base classifier design involved parameters such as the number of trees (n_estimators), maximum depth of trees (max_depth), minimum number of samples required to split an internal node (min_samples_split), and the number of features considered for splitting (max_features). These parameters are critical for controlling the complexity and learning capacity of each tree within the ensemble.

The initial model definition can be expressed as below,

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(
 n_estimators=100,
 max_depth=None,
 min_samples_split=2,
 max_features='auto',
 random_state=42
)
```

This architecture ensures a strong baseline while allowing future optimization through hyperparameter tuning. The model was designed to classify patient Status based on various clinical attributes, and its ensemble nature made it resilient to outliers and noise in the dataset.

3.3.4.2 Hyperparameter Tuning

To improve the performance of the Random Forest model, hyperparameter tuning was conducted using *RandomizedSearchCV*, which enables efficient exploration of a wide range of hyperparameter values by randomly sampling combinations instead of evaluating all possibilities exhaustively.

A set of key hyperparameters was selected and a distribution range was defined for each, as outlined below:

Hyperparameter	Description	Values Considered
n_estimators	Number of trees in the forest	100 to 1000
max_depth	Maximum depth of each decision tree	10, 20, 30, None
min_samples_split	Minimum number of samples required to split an internal node	2, 5, 10
min_samples_leaf	Minimum number of samples required to be at a leaf node	1, 2, 4
max_features	Number of features considered at each split	'auto', 'sqrt', 'log2'

The search was configured to run for **50 iterations** using **5-fold cross-validation**, striking a balnce between accuracy and computational efficiency. This strategy helped in identifying a more generalized model by minimizing the risk of overfitting.

```
param_dist = {
    'n_estimators': [100, 200, 500, 800, 1000],
    'max_depth': [10, 20, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt', 'log2']
}

random_search = RandomizedSearchCV(
```

```
    estimator=model,
    param_distributions=param_dist,
    n_iter=50,
    cv=5,
    scoring='accuracy',
    verbose=2,
    random_state=42,
)
```

Once the best hyperparameter combination was identified, the model was retrained using the optimal settings. Model performance was evaluated using a range of metrics including accuracy, precision, recall, F1-score, and mean squared error (MSE). These metrics were computed on both the training and test datasets to detect any signs of overfitting or underfitting.

This structured approach to hyperparameter optimization ensured the Random Forest model was well-tuned for the clinical data, leading to a reliable classification of patient outcomes and supporting the broader objectives of the project.

4 Results and Analysis

4.1 Deep Neural network

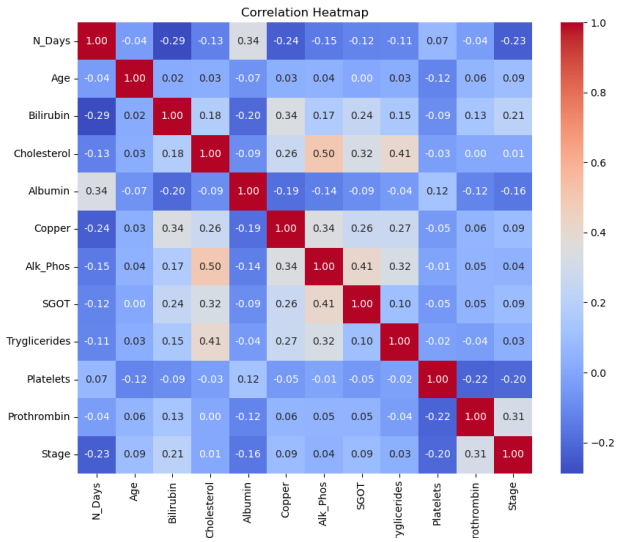

Figure 3: Correclation heatmap for DNN

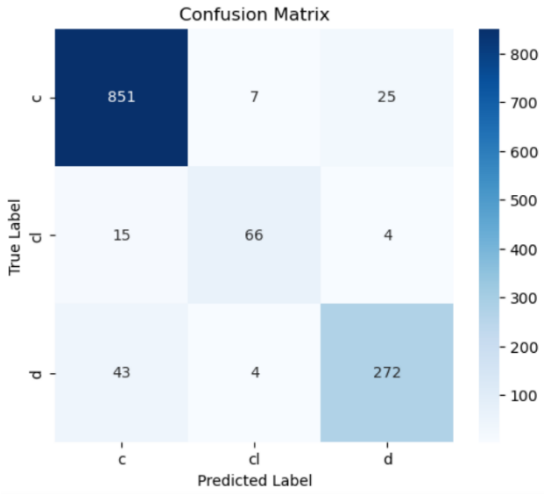

Figure 2: Confusion matrix for DNN

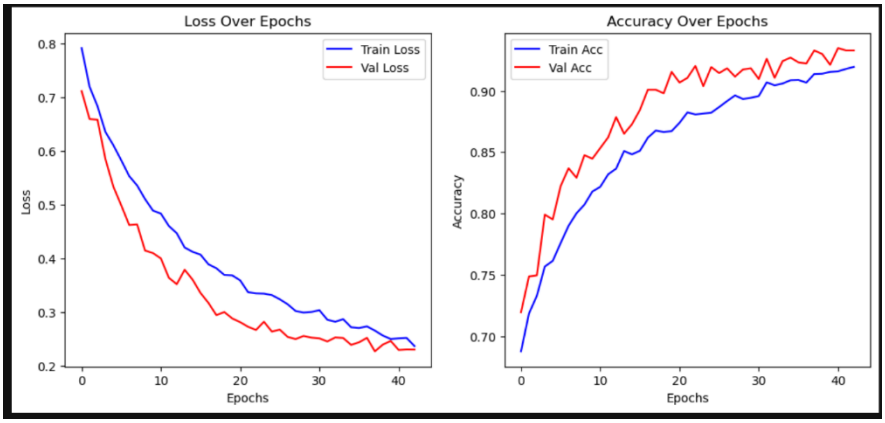

Figure 4: Loss and accuracy over epochs of DNN

this deep neural network model was trained on clinical laboratory measurements to predict disease stages "c", "c1", and "d". A preliminary correlation analysis showed only moderate linear relationships among features (e.g. alkaline phosphatase vs. cholesterol $r \approx 0.50$, albumin vs. follow-up days $r \approx -0.34$), indicating the need for a non-linear learner. Over 40 epochs, training loss fell smoothly from ~0.80 to ~0.24 while validation loss declined in parallel to ~0.22, and training accuracy climbed from ~0.69 to ~0.92 with validation accuracy peaking at ~0.93—signs of strong learning and good generalization without overfitting. On a held-out test set of 1,287 samples, the model achieved an overall accuracy of 92% and a weighted-average F1-score of 0.92. Class-specific results are summarized below:

Class	**Precision**	**Recall**	**F1-Score**	**Support**
c	0.94	0.96	0.95	883
c1	0.86	0.78	0.81	85
d	0.90	0.85	0.88	319

The model excels at predicting the majority class "c" but shows slightly lower recall for the under-represented "c1" group; future work might employ class-weighted loss or data augmentation to address this imbalance.

4.2 Support Vector Machine

To optimize the performance of the Support Vector Machine (SVM) model, hyperparameter tuning was carried out using GridSearchCV with 5-fold cross-validation. The tuning process explored various combinations of key hyperparameters, namely C, gamma, and kernel.

- The C parameter, which controls the trade-off between achieving low training error and minimizing testing error, was found to be optimal at C = 10, indicating a good balance between model complexity and generalization.
- The gamma parameter, which defines the influence range of a single training instance, was set to 0.1, providing a moderate radius suitable for capturing non-linear data patterns.

- The optimal kernel identified was Radial Basis Function (RBF), a well-regarded choice for modeling complex, non-linear decision boundaries in biomedical datasets.

Best Parameters:
{'C': 10, 'gamma': 0.1, 'kernel': 'rbf'}

Using these optimized parameters, the final SVM model was trained and evaluated on the test set. The model achieved an **accuracy of 88.66%**, indicating solid overall performance in predicting patient outcomes.

The Confusion matrix

[[820 5 46]

[23 55 6]

[59 7 266]]

4.3 Extreme Gradient Boosting

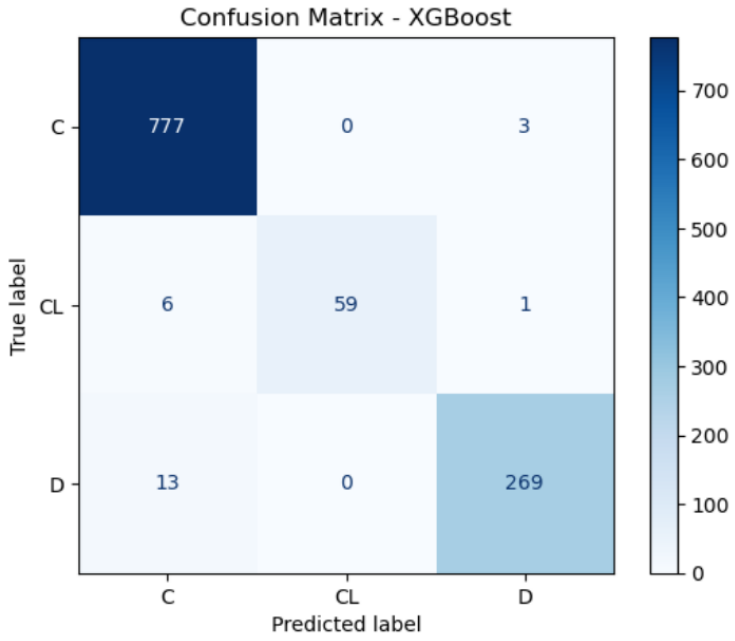

Figure 5:Confusion matrix for XGboost Method

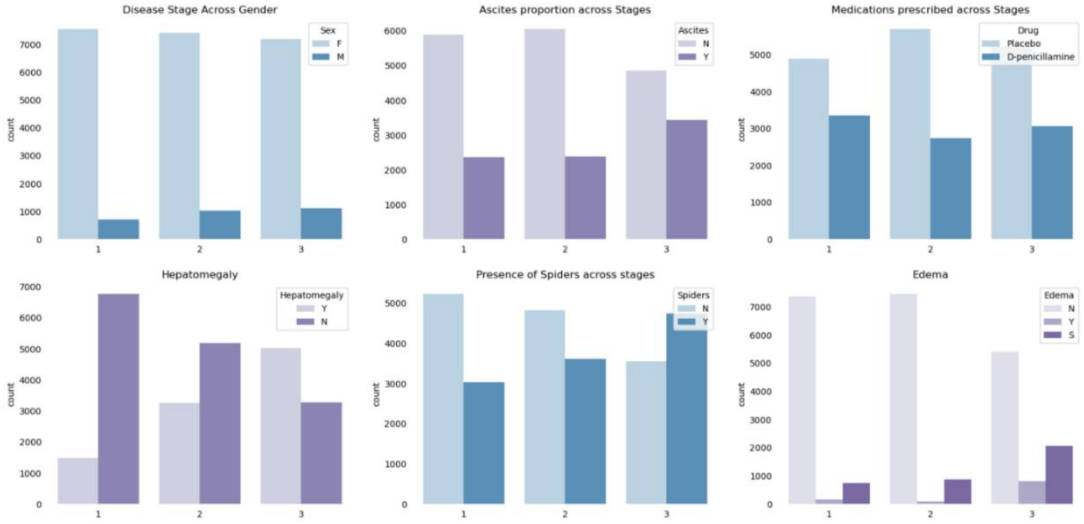

Figure 6: : Features with their relation with the disease

The results illustrate exceptional classification performance across all target classes. Notably, class 0 (typically the majority class) achieved perfect recall and nearly perfect F1-score, indicating minimal misclassifications. Class 1, despite having fewer examples, maintained a strong F1-score of 0.94 with high precision (1.00), although a slightly lower recall (0.89) was observed. Class 2 also performed strongly, with an F1-score of 0.97.

The high macro and weighted average scores demonstrate that the model is not only accurate but also well-balanced across different patient outcomes, avoiding bias toward majority classes. The minimal performance gap between precision and recall metrics suggests strong generalization capability without overfitting. Additionally, XGBoost's built-in regularization mechanisms (e.g., gamma, subsample, colsample_bytree) contributed to stable and reliable predictions.

4.4 Random Forest

The model evaluation outcomes illuminate the model performance, measuring accuracy, classification metrics, and the behavior of the model during training and testing. The highlights are the following:

1. **Best Hyperparameters**:
    - The best hyperparameters obtained from the Randomized Search Cross-Validation are:
        1. **n_estimators = 100**
        2. **min_samples_split = 5**
        3. **min_samples_leaf = 3**
        4. **max_features = 'log2'**
        5. **max_depth = 20**

2. **Accuracy**:
    - The final accuracy of the model on the test set is **95.3%**.
    - Having an accuracy of approximately 98.0% on the cross-validation set, the model has learnt well from the training set.

3. **Classification Report**:
    - The classification report thus reveals a good performance over all classes, with a general weighted mean of precision, recall, and F1 score all at about 0.95, suggesting that the model is well balanced over the various classes (C, CL, D).
4. **Mean Squared Error (MSE)**:
    - The Learning Set carries an MSE of 0.061 while the Test Set has an MSE of 0.161. Although quite low, it still demonstrates that the model is generalizing well and its not over-fitted to the given data.

5. **Overfitting/Underfitting**:

- Judging by the analyses rendered over training accuracy, test accuracy, and MSE, it is concluded that the model approximates well. The model had high accuracy in both the training and test sets, whereby the MSE remained fairly low and balanced over both sets.

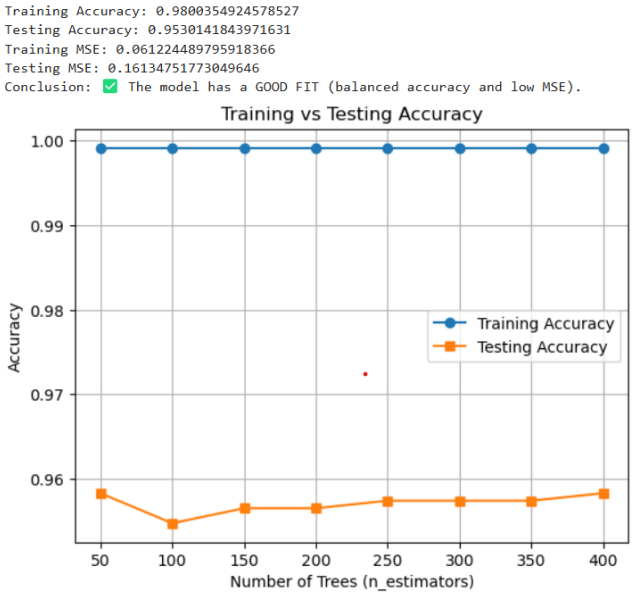

Figure 7:Training vs Testing Accuracy for Random Forest

6. **Confusion Matrix**:

- The confusion matrix indicates that the model performs exceptionally well at classifying the majority of instances correctly, with only a few misclassifications. The confusion matrix is as follows:

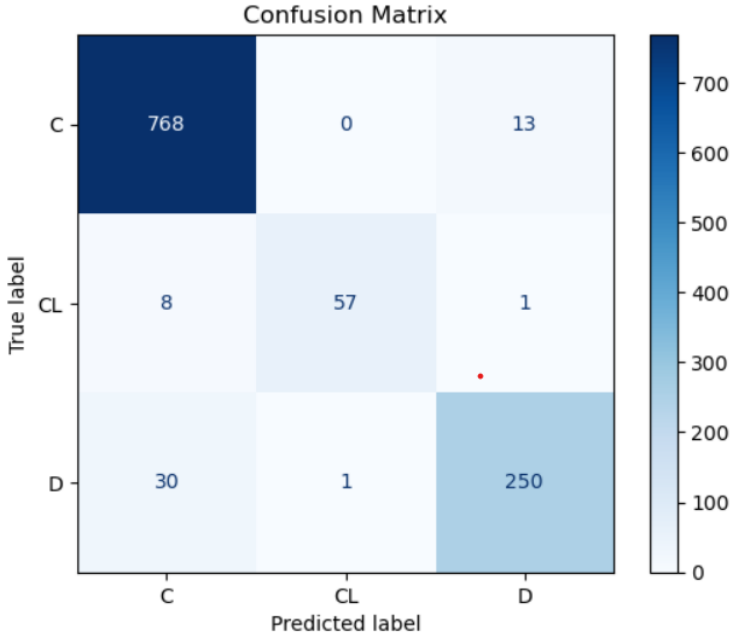

Figure 8: Confusion matrix for Random Forest

5 Discussion

In the study undertaken, four machine learning models were implemented to predict patient outcomes (status) in a liver cirrhosis dataset based on clinical and biochemical features: Support Vector Machine, Random Forest, Deep Neural Network, and Extreme Gradient Boosting. Each model was designed over a rigorously preprocessed dataset where duplicate records were eliminated, outliers filtered, and categorical features encoded. Considering this work's importance, stratified train-test splitting and standard normalization techniques were employed to guarantee representativeness and quality of the input data.

Support Vector Machine (SVM)

The SVM classifier was preferred on the basis of its theoretical sturdiness and performance in high-dimensional spaces. After parametric optimization with GridSearchCV, the best setting was RBF kernel, C=10, and gamma=0.1, where the model achieved an accuracy of 88.66% on the test set. The class-wise analysis indicated that there was a high-performing class 0 (F1-score=0.92), but class 1 has witnessed a significant drop (F1-score=0.73), which mainly owed to a 0.65 recall. This points out that while the SVM was able to successfully identify dominating patterns, it struggled to detect infrequent cases such as liver transplant, putting a spot-on limitation in imbalanced clinical datasets.

Random Forest

Randomized Forest is known for this: ensemble learning or resistance to overfitting. The model achieved an accuracy of 95.3% over the test set and almost 98% during cross-validation. The hyperparameters of the model were optimized via a RandomizedSearchCV, and the final model consisted of 100 estimators with a maximum depth of 20. Precision, recall, and F1-scores were all around 0.95, suggesting good class-wise performance. The MSEs are quite low for both training (0.061) and testing (0.161), confirming that the model was generalized well. Further, the confusion matrix revealed very high correctness levels of classification. There are only very few misclassifications that have occurred among the minority classes. Thus, this model demonstrated quite excellent balance between prediction power and robustness.

Deep Neural Network (DNN)

A Deep Neural Network was trained on laboratory and clinical measurements with a multi-layer architecture finetuned by Bayesian Optimization. The methodology includes strategies of early stopping and dropout to control overfitting. Over 40 epochs of training, the validation accuracy peaked at 93%, and the final test accuracy was 92%, with a weighted F1-score of 0.92. Class specific F1-scores were 0.95 (class "c"), 0.81 (class "c1"), and 0.88 (class "d"). Even though it gave excellent performance in the dominant class, the much lower recall in class "c1" (0.78) indicated that the model had some difficulty dealing with underrepresented cases. The overall learning curve suggested smooth convergence with minimal overfitting, and high-dimensional

non-linear mapping, as the DNN is capable of modeling highly complex relationships, made it an important device to be used in complex biomedical data. Such devices have, however, limitations against tree-based models with respect to interpretability and computational cost.

Extreme Gradient Boosting (XGBoost)

Amongst all models, XGBoost showed the highest performance. After hyperparameter tuning through extensive GridSearchCV, the optimized model reached a test accuracy of 98%, the highest in this study. The performance metrics were outstanding, with precision, recall, and F1-scores higher than 0.95 in all classes. The weighted F1-score was also 0.98, which indicates balanced and robust classification across patient groups. There was very little misclassification present in the confusion matrix. The model, using the gradient boosting framework, could learn complex feature interactions, while at the same time maintaining interpretability through feature importance metrics. In addition, its ability to impute or treat missing data, along with its built-in regularization mechanisms, all helped it achieve better performance and resist overfitting.

6. Conclusion

Four different models were tested to predict outcomes of liver cirrhosis based on structured clinical and biochemical variables. Optimized hyperparameters were used to train the models and were thereafter assessed subsequently for standard evaluation metrics of accuracy, precision, recall, and F1-score.

For the Support Vector Machine, moderate results were seen (accuracy = 88.66%); the model was performing adequately on the majority class while being less sensitive toward the minority cases. The Random Forest moderately generalizes with an impressive test accuracy of 95.3%, balanced performance across both classes, and low error rates. The Deep Neural Network achieved 92% accuracy with an F1 score of 0.92, indicating significant non-linear learning capabilities but lesser sensitivity toward minority classes.

The XGBoost model passed all tests with 98% accuracy and an F1 score of 0.98 with minimal misclassification and good performance on all classes. Good generalizability, interpretability, and rapid training make it the best candidate model to be put into application in liver cirrhosis prognostication.

In conclusion, XGBoost is the best and most accurate classifier for the dataset on liver cirrhosis and is suggested for inclusion in clinical decision support.

7 References

1.Yao, F., Luo, J., Zhou, Q., Wang, L. and He, Z. (2025). Development and validation of a machine learning-based prediction model for hepatorenal syndrome in liver cirrhosis patients using MIMIC-IV and eICU databases. Scientific Reports, [online] 15(1). doi:https://doi.org/10.1038/s41598-025-86674-9.

2. Tarwidi, D., Pudjaprasetya, S.R., Adytia, D. and Apri, M. (2023). An optimized XGBoost-based machine learning method for predicting wave run-up on a sloping beach. MethodsX, [online] 10, p.102119. doi:https://doi.org/10.1016/j.mex.2023.102119.

3. Misra, S. and Li, H. (2017). Random Forest - an Overview | ScienceDirect Topics. [online] Sciencedirect.com. Available at: https://www.sciencedirect.com/topics/engineering/random-forest.

4. www.sciencedirect.com. (n.d.). Deep Neural Network - an overview | ScienceDirect Topics. [online] Available at: https://www.sciencedirect.com/topics/engineering/deep-neural-network.

5. Cleveland Clinic (2023). Cirrhosis of the Liver. [online] Cleveland Clinic. Available at: https://my.clevelandclinic.org/health/diseases/15572-cirrhosis-of-the-liver.

6. ResearchGate. (n.d.). (PDF) An Overview of Overfitting and its Solutions. [online] Available at: https://www.researchgate.net/publication/331677125_An_Overview_of_Overfitting_and_its_Solutions.

7. Ying, X. (2019). An Overview of Overfitting and its Solutions. Journal of Physics: Conference Series, 1168(2), p.022022. doi:https://doi.org/10.1088/1742-6596/1168/2/022022.

8 Appendix

1. Deep neural Network Model

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import math

warnings.filterwarnings("ignore")

import tensorflow as tf
# Adjust these numbers based on your CPU cores.
tf.config.threading.set_intra_op_parallelism_threads(8)
tf.config.threading.set_inter_op_parallelism_threads(8)

from tensorflow import keras
from tensorflow.keras.callbacks import EarlyStopping

from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import confusion_matrix, classification_report

import keras_tuner as kt

def load_and_preprocess_data(filepath):
    # Load raw CSV data
    df = pd.read_csv(filepath)
```

```
    # Drop exact duplicate rows (after cleaning string columns)
    df_clean = df.copy()
    for col in df_clean.select_dtypes(include='object'):
        df_clean[col] = df_clean[col].astype(str).str.strip().str.lower()
    df_clean = df_clean.drop_duplicates()

    # Remove outliers in numeric columns using the IQR method
    num_cols = df_clean.select_dtypes(include=np.number).columns
    for col in num_cols:
        Q1 = df_clean[col].quantile(0.25)
        Q3 = df_clean[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5* IQR
        upper_bound = Q3 + 1.5* IQR
        df_clean = df_clean[(df_clean[col] >= lower_bound) & (df_clean[col]
<= upper_bound)]

    # Create a working copy for model training that encodes categorical
data.
    df_model = df_clean.copy()

    # Print some info
    print("Dataset Info:")
    print(f"Original rows: {df.shape[0]}, Rows after cleaning duplicates
and removing outliers: {df_clean.shape[0]}")
    print(df_clean.info(), "\n")

    # One-hot encode selected columns (drop the first to avoid dummy
variable trap)
    cat_cols = ['Drug', 'Sex', 'Ascites', 'Hepatomegaly', 'Spiders',
'Edema']
    df_model = pd.get_dummies(df_model, columns=cat_cols, drop_first=True)
```

```
    # Label encoding the target (the target column is 'Status')
    label_encoder = LabelEncoder()
    df_model['Status'] = label_encoder.fit_transform(df_model['Status'])

    # Fill numeric missing values with the column mean
    for col in df_model.select_dtypes(include=np.number).columns:
        df_model[col].fillna(df_model[col].mean(), inplace=True)

    # Fill any remaining missing values in object columns with mode
    for col in df_model.select_dtypes(include='object').columns:
        df_model[col].fillna(df_model[col].mode()[0], inplace=True)

    # Separate features and target
    X = df_model.drop(columns=['Status'])
    y = df_model['Status']

    # Standardize the feature set
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # Split into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=42)

    return X_train, X_test, y_train, y_test, label_encoder, df_clean


from sklearn.preprocessing import StandardScaler


def plot_data_insights(df):
    # Get numeric columns
```

```
    num_cols = df.select_dtypes(include=[np.number]).columns.tolist()
    n_plots = len(num_cols)

    # Determine grid size dynamically
    n_cols = 3
    n_rows = math.ceil(n_plots / n_cols)

    # 1. Histogram plots
    plt.figure(figsize=(n_cols * 5, n_rows * 4))
    for i, col in enumerate(num_cols):
        plt.subplot(n_rows, n_cols, i + 1)
        sns.histplot(df[col], kde=True)
        plt.title(f"Distribution of {col}")
    plt.tight_layout()
    plt.savefig("histograms.png")
    plt.show()

    # 2. Correlation heatmap
    plt.figure(figsize=(10, 8))
    sns.heatmap(df[num_cols].corr(), annot=True, cmap='coolwarm',
fmt=".2f")
    plt.title("Correlation Heatmap")
    plt.savefig("correlation_heatmap.png")
    plt.show()

    # 3. Boxplot after StandardScaler normalization
    scaler = StandardScaler()
    scaled_data = scaler.fit_transform(df[num_cols])
    scaled_df = pd.DataFrame(scaled_data, columns=num_cols)

    plt.figure(figsize=(15, 8))
```

```
    sns.boxplot(data=scaled_df, orient="h")
    plt.title("Boxplot of Standardized Numeric Variables")
    plt.savefig("boxplot.png")
    plt.show()



def build_model(hp):
    """
    Builds and compiles a Keras model with hyperparameters defined by the
keras_tuner hp object.

    Hyperparameters varied:
      - num_layers: Number of hidden layers.
      - units: Number of units per layer.
      - activation: Activation function for hidden layers.
      - dropout_rate: Dropout rate after each hidden layer.
      - learning_rate: Learning rate for the optimizer.
      - optimizer: Choice between 'adam' and 'rmsprop'.

    Global variables:
      - input_shape_dim: Input dimension (set in main after data loading).
      - num_classes: Number of output classes.
    """
    model = keras.Sequential()
    model.add(keras.layers.InputLayer(input_shape=(input_shape_dim,)))

    # Vary the number of layers (e.g., 2 to 5 layers)
    num_layers = hp.Int('num_layers', min_value=2, max_value=5, step=1)
    for i in range(num_layers):
        # Vary the number of units; try a range of values
        units = hp.Choice(f'units_{i}', values=[32, 64, 128, 256])
```

```
        # Vary the activation function; try relu or tanh
        activation = hp.Choice(f'activation_{i}', values=['relu', 'tanh'])
        model.add(keras.layers.Dense(units=units, activation=activation))
        # Vary dropout rate between 0.1 and 0.5
        dropout_rate = hp.Float(f'dropout_rate_{i}', min_value=0.1,
max_value=0.5, step=0.1)
        model.add(keras.layers.Dropout(rate=dropout_rate))

    # Output layer: assumes classification task with softmax
    model.add(keras.layers.Dense(num_classes, activation='softmax'))

    # Vary the learning rate and optimizer
    learning_rate = hp.Choice('learning_rate', values=[1e-2, 5e-3, 1e-3,
5e-4])
    optimizer_choice = hp.Choice('optimizer', values=['adam', 'rmsprop'])
    if optimizer_choice == 'adam':
        optimizer = keras.optimizers.Adam(learning_rate=learning_rate)
    else:
        optimizer = keras.optimizers.RMSprop(learning_rate=learning_rate)

    model.compile(optimizer=optimizer,
loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model

def perform_hyperparameter_tuning(X_train, y_train):
    """
    Performs hyperparameter tuning using Bayesian Optimization via Keras
Tuner.

    Returns:
        best_hp: The best hyperparameter configuration.
```

```
        tuner: The tuner object (in case you need to inspect the full
history).
    """

    tuner = kt.BayesianOptimization(

        hypermodel=build_model,

        objective='val_accuracy',

        max_trials=30,           # Increase number of trials to explore a
larger search space

        executions_per_trial=2,  # Multiple executions reduce variance of
results

        directory='my_dir',

        project_name='liver_cirrhosis_tuning'

    )


    early_stop = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)


    tuner.search(X_train, y_train, epochs=30, validation_split=0.2,
callbacks=[early_stop], verbose=1)


    best_hp = tuner.get_best_hyperparameters(num_trials=1)[0]

    print("Best hyperparameters found:")

    print(f"  Number of layers: {best_hp.get('num_layers')}")

    for i in range(best_hp.get('num_layers')):

        print(f"  Layer {i}: {best_hp.get(f'units_{i}')} units, activation:
{best_hp.get(f'activation_{i}')}, dropout rate:
{best_hp.get(f'dropout_rate_{i}')}")

    print(f"  Learning rate: {best_hp.get('learning_rate')}")

    print(f"  Optimizer: {best_hp.get('optimizer')}")


    return best_hp, tuner
```

```
def plot_history(history):
    # Plot training vs validation loss and accuracy
    plt.figure(figsize=(12, 5))
    # Loss plot
    plt.subplot(1, 2, 1)
    plt.plot(history.history['loss'], label='Train Loss', color='blue')
    plt.plot(history.history['val_loss'], label='Val Loss', color='red')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Loss Over Epochs')
    plt.legend()

    # Accuracy plot
    plt.subplot(1, 2, 2)
    plt.plot(history.history['accuracy'], label='Train Acc', color='blue')
    plt.plot(history.history['val_accuracy'], label='Val Acc', color='red')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.title('Accuracy Over Epochs')
    plt.legend()

    plt.show()

def plot_confusion_matrix(y_true, y_pred, label_encoder):
    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)

    plt.figure(figsize=(6,5))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=label_encoder.classes_,
yticklabels=label_encoder.classes_)
```

```
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title('Confusion Matrix')
    plt.show()


 if _name_ == '_main_':
    filepath = "liver_cirrhosis.csv"


    # Load and preprocess the data
    X_train, X_test, y_train, y_test, label_encoder, df_raw =
load_and_preprocess_data(filepath)




    # Set global variables for model building (input_shape_dim and
num_classes)
    input_shape_dim = X_train.shape[1]
    num_classes = len(np.unique(y_train))


    # Perform hyperparameter tuning (note: no separate param grid is used
here)
    best_hp, tuner = perform_hyperparameter_tuning(X_train, y_train)



# Build the final model using the best hyperparameters from the tuner
final_model = tuner.hypermodel.build(best_hp)
early_stop = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)


    # Train the final model using a longer epoch schedule, if necessary
history = final_model.fit(X_train, y_train,
                              epochs=50,
```

```
                                batch_size=16,
                                validation_split=0.2,
                                callbacks=[early_stop],
                                verbose=1)

    # Evaluate on test set
test_loss, test_accuracy = final_model.evaluate(X_test, y_test, verbose=0)
print(f"\nTest Loss: {test_loss:.3f}, Test Accuracy: {test_accuracy:.3f}")



   # Exploratory Data Visualization
plot_data_insights(df_raw)
    # Plot training history
plot_history(history)

    # Make predictions and evaluate performance
y_pred_probs = final_model.predict(X_test)
y_pred = np.argmax(y_pred_probs, axis=1)

print("\nClassification Report:")
print(classification_report(y_test, y_pred,
target_names=label_encoder.classes_))

    # Plot confusion matrix heatmap
plot_confusion_matrix(y_test, y_pred, label_encoder)
```

2. Extreme Gradient Boosting Method

```
import pandas as pd
```

```
import numpy as np


df = pd.read_csv("C:/Users/Duvindu Ushan/Downloads/liver_cirrhosis.csv")


df


df.info()


df.describe()


#Search for Missing values
df.isna().sum()


#Check for duplicates
duplicates = df.duplicated()
num_duplicates = duplicates.sum()


#Remove duplicates
df_unique = df.drop_duplicates()


#Display the number of duplicates and unique rows
print('Number of duplicated rows: ', num_duplicates)
print('Number of unique rows: ', len(df_unique))


#Show the duplicates
duplicates = df.duplicated()
duplicated_rows = df[duplicates]
```

```
duplicated_rows



# remove the duplicates
df_unique = df.drop_duplicates()


df_unique


from sklearn.preprocessing import LabelEncoder


# Copy the cleaned dataframe
df_encoded = df_unique.copy()


# Binary encoding for Yes/No categorical variables
binary_map = {"N": 0, "Y": 1}
df_encoded["Ascites"] = df_encoded["Ascites"].map(binary_map)
df_encoded["Hepatomegaly"] = df_encoded["Hepatomegaly"].map(binary_map)
df_encoded["Spiders"] = df_encoded["Spiders"].map(binary_map)


# Encoding "Edema" (N = 0, S = 1, Y = 2)
edema_map = {"N": 0, "S": 1, "Y": 2}
df_encoded["Edema"] = df_encoded["Edema"].map(edema_map)


# Encoding "Drug" (Placebo = 0, D-penicillamine = 1)
drug_map = {"Placebo": 0, "D-penicillamine": 1}
df_encoded["Drug"] = df_encoded["Drug"].map(drug_map)


# Encoding "Sex" (F = 0, M = 1)
```

```
sex_map = {"F": 0, "M": 1}
df_encoded["Sex"] = df_encoded["Sex"].map(sex_map)

# Encoding the target variable "Status" ('C'= 0, 'CL'= 1, 'D'= 2)
label_encoder = LabelEncoder()
df_encoded["Status"] = label_encoder.fit_transform(df_encoded["Status"])

# Display the encoded dataset
df_encoded

from sklearn.preprocessing import MinMaxScaler

# Function to remove outliers using IQR
def remove_outliers_iqr(data, column):
    Q1 = data[column].quantile(0.25)
    Q3 = data[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return data[(data[column] >= lower_bound) & (data[column] <=
upper_bound)]

# List of numerical columns to check for outliers
numerical_columns = ["Age", "Bilirubin", "Cholesterol", "Albumin",
"Copper",
                     "Alk_Phos", "SGOT", "Tryglicerides", "Platelets",
"Prothrombin"]

# Remove outliers for each numerical column
```

```
df_cleaned = df_encoded.copy()
for col in numerical_columns:
    df_cleaned = remove_outliers_iqr(df_cleaned, col)

# Normalize numerical features using Min-Max Scaling
scaler = MinMaxScaler()
df_cleaned[numerical_columns] =
scaler.fit_transform(df_cleaned[numerical_columns])

df_cleaned

import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Visualize distributions after cleaning & normalization
plt.figure(figsize=(12, 6))
sns.boxplot(data=df_cleaned[numerical_columns])
plt.xticks(rotation=45)
plt.title("Boxplot of Normalized Features (After Outlier Removal)")
plt.show()

from sklearn.model_selection import train_test_split

# Define features (X) and target variable (y)
X = df_cleaned.drop(columns=["Status"])  # Features
y = df_cleaned["Status"]  # Target variable
```

```
# Split data into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)


from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix


# Initialize and train the Logistic Regression model
log_reg = LogisticRegression(random_state=42, max_iter=1000)
log_reg.fit(X_train, y_train)


# Make predictions on the test set
y_pred = log_reg.predict(X_test)


# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)


# Display results
print("Logistic Regression Model Performance:")
print("Accuracy:", accuracy)
print("\nConfusion Matrix:\n", conf_matrix)
print("\nClassification Report:\n", class_report)


from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
```

```
from xgboost import XGBClassifier

from sklearn.metrics import accuracy_score


# Define models

models = {

    "Logistic Regression": LogisticRegression(random_state=42,
max_iter=1000),

    "Decision Tree": DecisionTreeClassifier(random_state=42),

    "Random Forest": RandomForestClassifier(random_state=42,
n_estimators=100),

    "SVM": SVC(random_state=42),

    "XGBoost": XGBClassifier(use_label_encoder=False,
eval_metric='logloss', random_state=42)

}


# Train and evaluate each model

results = {}

for name, model in models.items():

    model.fit(X_train, y_train)  # Train the model

    y_pred = model.predict(X_test)  # Make predictions

    accuracy = accuracy_score(y_test, y_pred)  # Calculate accuracy

    results[name] = accuracy


# Display results

for model, acc in results.items():

    print(f"{model}: Accuracy = {acc:.4f}")


# Plot bar chart

plt.figure(figsize=(10, 5))
```

```
plt.bar(results.keys(), results.values(), color=['blue', 'green', 'red', 'purple', 'orange'])

plt.xlabel("Models")

plt.ylabel("Accuracy")

plt.title("Model Accuracy Comparison")

plt.ylim(0, 1)

plt.xticks(rotation=45)

plt.show()


from xgboost import XGBClassifier

from sklearn.model_selection import GridSearchCV

from sklearn.metrics import accuracy_score


# Select XGBoost as the best-performing model

best_model = XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_state=42)


# Define hyperparameter grid for tuning

param_grid = {

    "n_estimators": [50, 100, 150],

    "max_depth": [3, 6, 10, 15],  # XGBoost uses max_depth (None is not allowed)

    "learning_rate": [0.01, 0.1, 0.2],  # Learning rate controls how fast it learns

    "subsample": [0.7, 0.8, 1.0],  # Subsampling ratio for training instances

    "colsample_bytree": [0.7, 0.8, 1.0],  # Feature subsampling

    "gamma": [0, 0.1, 0.2]  # Minimum loss reduction required for a split

}
```

```
# Perform Grid Search with Cross-Validation (5-fold)

grid_search = GridSearchCV(best_model, param_grid, cv=5,
scoring="accuracy", n_jobs=-1, verbose=1)

grid_search.fit(X_train, y_train)


# Get best parameters

best_params = grid_search.best_params_

print("Best Hyperparameters:", best_params)


# Train the best model with optimal hyperparameters

optimized_model = XGBClassifier(**best_params, use_label_encoder=False,
eval_metric='logloss', random_state=42)

optimized_model.fit(X_train, y_train)


# Evaluate the optimized model

y_pred_opt = optimized_model.predict(X_test)

optimized_accuracy = accuracy_score(y_test, y_pred_opt)


print("\nOptimized XGBoost Model Accuracy:", optimized_accuracy)



import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt


# Get feature importance from the optimized XGBoost model

feature_importances = optimized_model.feature_importances_


# Create a DataFrame for better visualization
```

```
importance_df = pd.DataFrame({"Feature": X_train.columns, "Importance": feature_importances})

importance_df = importance_df.sort_values(by="Importance", ascending=False)


# Plot feature importance

plt.figure(figsize=(10, 6))

sns.barplot(x="Importance", y="Feature", data=importance_df, palette="viridis")

plt.title("Feature Importance in Predicting Liver Cirrhosis (XGBoost)")

plt.xlabel("Importance Score")

plt.ylabel("Feature")

plt.show()


# Display top features

print("Top Features Contributing to Predictions:")

print(importance_df.head(10))


from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report

import matplotlib.pyplot as plt


# Generate predictions for XGBoost

y_pred_xgb = optimized_model.predict(X_test)


# Compute Confusion Matrix

conf_matrix_xgb = confusion_matrix(y_test, y_pred_xgb)


# Plot Confusion Matrix

plt.figure(figsize=(6, 5))
```

```
disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix_xgb,
display_labels=label_encoder.classes_)

disp.plot(cmap=plt.cm.Blues)

plt.title("Confusion Matrix - XGBoost")

plt.show()


#Display Confusion Matrix

print("Confusion Matrix:\n",conf_matrix_xgb)


# Display Classification Report

class_report_xgb = classification_report(y_test, y_pred_xgb)

print("\n Classification Report for XGBoost:\n", class_report_xgb)


plt.figure(figsize=(21.2,10))


plt.subplot(2,3,1)

sns.countplot(x=df['Stage'], hue=df['Sex'], palette='Blues', alpha=0.9)

sns.despine(top=True, right=True, bottom=True, left=True)

plt.tick_params(axis='both', which='both', bottom=False, top=False,
left=False)

plt.xlabel('')

plt.title('Disease Stage Across Gender')


plt.subplot(2,3,2)

sns.countplot(x=df['Stage'], hue=df['Ascites'], palette='Purples',
alpha=0.9)

sns.despine(top=True, right=True, bottom=True, left=True)

plt.tick_params(axis='both', which='both', bottom=False, top=False,
left=False)
```

```
plt.xlabel('')
plt.title('Ascites proportion across Stages')


plt.subplot(2,3,3)
sns.countplot(x=df['Stage'], hue=df['Drug'], palette='Blues', alpha=0.9)
sns.despine(top=True, right=True, bottom=True, left=True)
plt.tick_params(axis='both', which='both', bottom=False, top=False,
left=False)
plt.xlabel('')
plt.title('Medications prescribed across Stages');


plt.subplot(2,3,4)
sns.countplot(x=df['Stage'], hue=df['Hepatomegaly'], palette='Purples',
alpha=0.9)
sns.despine(top=True, right=True, bottom=True, left=True)
plt.tick_params(axis='both', which='both', bottom=False, top=False,
left=False)
plt.xlabel('')
plt.title('Hepatomegaly');


plt.subplot(2,3,5)
sns.countplot(x=df['Stage'], hue=df['Spiders'], palette='Blues', alpha=0.9)
sns.despine(top=True, right=True, bottom=True, left=True)
plt.tick_params(axis='both', which='both', bottom=False, top=False,
left=False)
plt.xlabel('')
plt.title('Presence of Spiders across stages');


plt.subplot(2,3,6)
```

```
sns.countplot(x=df['Stage'], hue=df['Edema'], palette='Purples', alpha=0.9)

sns.despine(top=True, right=True, bottom=True, left=True)

plt.tick_params(axis='both', which='both', bottom=False, top=False,
left=False)

plt.xlabel('')

plt.title('Edema');
```

3. Random Forest Model

```
import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split, RandomizedSearchCV

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score, classification_report,
mean_squared_error

from sklearn.preprocessing import LabelEncoder


# Load dataset

file_path = r"C:\Users\biwla\Downloads\liver_cirrhosis (1).csv"

df = pd.read_csv(file_path)


# Drop duplicates

df = df.drop_duplicates()


# Drop rows with missing target

df = df.dropna(subset=['Status'])


# Fill missing values
```

```
for col in df.columns:
    if df[col].dtype == 'O':
        df[col].fillna(df[col].mode()[0], inplace=True)
    else:
        df[col].fillna(df[col].median(), inplace=True)

# Outlier removal using IQR
def remove_outliers_iqr(df):
    numeric_cols = df.select_dtypes(include=np.number).columns
    for col in numeric_cols:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        df = df[(df[col] >= lower_bound) & (df[col] <= upper_bound)]
    return df

df = remove_outliers_iqr(df)

# Encode categorical variables
label_encoders = {}
for col in df.select_dtypes(include='object').columns:
    if col != 'Status':
        le = LabelEncoder()
        df[col] = le.fit_transform(df[col])
        label_encoders[col] = le
```

```
# Encode target
target_le = LabelEncoder()
df['Status'] = target_le.fit_transform(df['Status'])


# Split data
X = df.drop(columns=['Status'])
y = df['Status']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)


# Random Forest with Hyperparameter Tuning
param_dist = {
    'n_estimators': [50, 100, 150],
    'max_depth':[5, 10, 15, 20],
    'min_samples_leaf': [3, 5, 10],
    'min_samples_split': [5, 10, 20],
    'max_features': ['sqrt', 'log2'],
}


rfc = RandomForestClassifier(random_state=42)
rfc_random = RandomizedSearchCV(
    estimator=rfc,
    param_distributions=param_dist,
    n_iter=50,
    cv=5,
    verbose=1,
    random_state=42,
    n_jobs=-1
```

```
)
rfc_random.fit(X_train, y_train)


# Final model evaluation
best_model = rfc_random.best_estimator_
y_pred = best_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)


print("Best Parameters:", rfc_random.best_params_)
print("Final Accuracy:", accuracy)
print("Classification Report:\n", classification_report(y_test, y_pred,
target_names=target_le.classes_))


# Overfitting/Underfitting Check
y_train_pred = best_model.predict(X_train)
train_acc = accuracy_score(y_train, y_train_pred)
test_acc = accuracy_score(y_test, y_pred)


# MSE Calculation
train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_pred)


print("\nTraining Accuracy:", train_acc)
print("Testing Accuracy:", test_acc)
print("Training MSE:", train_mse)
print("Testing MSE:", test_mse)


# Conclusion based on accuracy + MSE
```

```
if train_acc > 0.98 and (train_acc - test_acc) > 0.03 and test_mse > train_mse * 1.5:
    print("Conclusion:  The model is OVERFITTING (high training accuracy, lower testing accuracy, and higher test MSE).")
elif train_acc < 0.75 and test_acc < 0.75 and train_mse > 0.2 and test_mse > 0.2:
    print("Conclusion:  The model is UNDERFITTING (both accuracies are low and both MSEs are high).")
else:
    print("Conclusion:  The model has a GOOD FIT (balanced accuracy and low MSE).")


# Plot Training vs Testing Accuracy
estimators = [50, 100, 150, 200, 250, 300, 350, 400]
train_scores = []
test_scores = []

for n in estimators:
    model = RandomForestClassifier(n_estimators=n, random_state=42)
    model.fit(X_train, y_train)
    train_scores.append(model.score(X_train, y_train))
    test_scores.append(model.score(X_test, y_test))

plt.plot(estimators, train_scores, label='Training Accuracy', marker='o')
plt.plot(estimators, test_scores, label='Testing Accuracy', marker='s')
plt.xlabel("Number of Trees (n_estimators)")
plt.ylabel("Accuracy")
plt.title("Training vs Testing Accuracy")
plt.legend()
plt.grid(True)
```

```
plt.show()


from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt


# Predict on test data (already done earlier)
y_pred = best_model.predict(X_test)


# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Test Accuracy:", accuracy)


# Confusion matrix
cm = confusion_matrix(y_test, y_pred)


# Display confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=target_le.classes_)
disp.plot(cmap='Blues', values_format='d')
plt.title("Confusion Matrix")
plt.show()
```

4.Support vector Machine Model

```
import pandas as pd
```

```
from sklearn.preprocessing import LabelEncoder

from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

from sklearn.model_selection import GridSearchCV

from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score


# Load the dataset

df = pd.read_csv('liver_cirrhosis.csv')


df.shape


# Identify missing values

missing_values = df.isnull().sum()


missing_values


# Find duplicate rows in the dataset

duplicate_rows = df[df.duplicated()]


# Count of duplicate rows

num_duplicates = duplicate_rows.shape[0]


#num_duplicates

num_duplicates


# Remove duplicate rows and keep only unique ones
```

```
df_unique = df.drop_duplicates()


# Check the new shape of the dataset
new_shape = df_unique.shape


new_shape


def remove_outliers_iqr(df, columns):
    """
    Removes outliers from a DataFrame using the IQR method for the specified columns.

    Parameters:
    df (DataFrame): The input DataFrame.
    columns (list): A list of numerical columns to check for outliers.

    Returns:
    DataFrame: The DataFrame with outliers removed.
    """
    df_filtered = df.copy()
    for col in columns:
        Q1 = df_filtered[col].quantile(0.25)
        Q3 = df_filtered[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 2 * IQR
        upper_bound = Q3 + 2 * IQR
        df_filtered = df_filtered[(df_filtered[col] >= lower_bound) & (df_filtered[col] <= upper_bound)]
    return df_filtered
```

```
# Identify numerical columns
numerical_cols = df_unique.select_dtypes(include=['int64',
'float64']).columns.tolist()


# Remove outliers
df_cleaned = remove_outliers_iqr(df_unique, numerical_cols)


# Check shape
df_cleaned.shape


X = df_cleaned.drop('Status', axis=1)
y = df_cleaned['Status']


le = LabelEncoder()
y = le.fit_transform(y)


#Encode categorical columns (automatically detects object-type columns)
X_encoded = pd.get_dummies(X, drop_first=True)


X_train, X_test, y_train, y_test = train_test_split(
    X_encoded, y, test_size=0.2, random_state=42, stratify=y
)


scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
param_grid = {
    'C': [10**x for x in range(-1, 3)],
    'gamma': [10**x for x in range(-1, 2)],
    'kernel': ['rbf', 'poly', 'sigmoid']
}

# Grid search with cross-validation
grid = GridSearchCV(SVC(), param_grid, refit=True, verbose=1, cv=5)
grid.fit(X_train_scaled, y_train)

# Print the best parameters
print("Best Parameters:", grid.best_params_)

# Use the best parameters found from the first cell
best_params = grid.best_params_



# Train SVM model using the best parameters
model = SVC(C=best_params['C'], gamma=best_params['gamma'],
kernel=best_params['kernel'])
model.fit(X_train_scaled, y_train)

# Make predictions
y_pred = model.predict(X_test_scaled)

# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
accuracy = accuracy_score(y_test, y_pred) * 100
```

```
print(f"Accuracy: {accuracy:.2f}%")


print("\n Classification Report:\n", classification_report(y_test, y_pred))
print("\n Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
```