Machine learning models for hepatitis classification

1. Data and problem setup

The dataset contains clinical and laboratory measurements for patients with hepatitis. The target variable **TARGET** has two classes (**1 & 2**), representing different outcome categories. The goal is to build classification models to predict **TARGET** from **demographic**, **symptom**, **liver exam findings**, and **lab variables** (e.g., bilirubin, ALK Phosphate, SGOT, albumin, PROTIME).

I treated the problem as a supervised binary classification task. The provided Hepatitis-Train.csv was used for model fitting and cross-validation, and Hepatitis-Test.csv served as an independent test set for final evaluation.

Numeric variables (e.g., Age, Bilirubin, ALK Phosphate, SGOT, Albumin, PROTIME) were standardized with StandardScaler. **Categorical** variables (sex, symptoms, imaging findings, and Histology) were one-hot encoded using OneHotEncoder with drop="first". All models were implemented in scikit-learn using pipelines so that preprocessing and classification were combined in a single, consistent workflow.

2. Base classifiers and default results

I trained 4 different classifiers using default hyperparameters: Linear Support Vector Classifier (LinearSVC), Decision Tree (DecisionTreeClassifier), Random Forest (RandomForestClassifier), K-Nearest Neighbor (KNeighborsClassifier). Each classifier was fit on the training data and evaluated on the held-out test set. Performance was summarized using accuracy, weighted precision, weighted recall, and weighted F1 score.

3. Random Forest hyperparameter tuning

To improve performance, I performed hyperparameter tuning on the Random Forest classifier using randomized search (RandomizedSearchCV) with 5-fold cross-validation. I tuned at least the following hyperparameters:

n_estimators (number of trees, range: 50–300)
max_depth (tree depth, range: 2–20)
min_samples_split (2–10)
min_samples_leaf (1–10)
max_features ({"sqrt", "log2", 0.5, None})
bootstrap ({True, False})

The search used weighted F1 score as the optimization metric. After 40 random combinations, the best model had approximately the following hyperparameters:

n_estimators: 179
max_depth: 2
min_samples_split: 7
min_samples_leaf: 3
max_features: 'sqrt'
bootstrap: False

This tuned Random Forest was then refit on the full training data and evaluated on the test set. The tuned model's test accuracy, precision, recall, and F1 score are included in the table above under "Random Forest (tuned)."

Comparing the tuned model to the default Random Forest:

Accuracy: decrease by approximately 0.27
Precision: decrease by approximately 0.26
Recall: decrease by approximately 0.27

F1: decrease by approximately 0.33

The tuned Random Forest performed worse across all metrics because the dataset is small, making it easy for the random search to overfit patterns in the cross-validation folds that don't generalize to the test set. In contrast, the default Random Forest settings already provide a strong, stable bias–variance balance for this dataset size. As a result, the tuned model likely became too specialized to the training folds, leading to the observed drops in accuracy (-0.27), precision (-0.26), recall (-0.27), and F1 score (-0.33).

4. Feature importance analysis
Using the tuned Random Forest, I extracted impurity-based feature importances (feature_importances_) and mapped them back to the preprocessed feature names (numeric features plus one-hot encoded categorical variables). Features were ranked by importance, and the top five were:

- Albumin: 0.297
- Bilirubin: 0.148
- Ascites_yes: 0.145
- Varices_yes: 0.120
- PROTIME: 0.105

The top five features identified by the tuned Random Forest highlight a clinically coherent pattern. Albumin was by far the most influential predictor (0.297), reflecting its role as a key marker of liver synthetic function. Bilirubin (0.148) and PROTIME (0.105) also ranked highly, both of which capture liver impairment and disease severity. The presence of ascites (0.145) and varices (0.120) were similarly important, indicating that structural complications of chronic liver disease strongly contribute to the model's classification decisions. Overall, the model relies most heavily on features that reflect both functional decline and advanced liver.

5. Stacking with an MLP meta-classifier
To explore ensembling, I constructed a one-layer stacking model. The base learners were the four classifiers described earlier: LinearSVC (default), Decision Tree (default), Tuned Random Forest, KNN (default)

For stacking features, I used each base model's predicted class label on the training data as meta-features (4 columns: one per base classifier). These meta-features were then used to train a Multilayer Perceptron (MLPClassifier) with a single hidden layer of 16 neurons (ReLU activation, Adam optimizer).

On the test set, the stacked model achieved the metrics shown in the table above under Stacking (MLP). Compared to the best individual model:

Accuracy: The Stacking_MLP achieved 0.73, which is a decrease of 0.09 compared to KNN_default 0.82.
Precision: The Stacking_MLP achieved 0.73, which is a decrease of 0.09 compared to KNN_default 0.82.
Recall: The Stacking_MLP achieved 0.73, which is a decrease of 0.09 compared to KNN_default 0.82.
F1: The Stacking_MLP achieved 0.72, which is a decrease of 0.10 compared to KNN_default 0.82.

6. Reason for better performance
Overall, the tuned Random Forest and the stacking ensemble demonstrate that combining multiple views of the data can yield strong performance, but the **small dataset limits the amount of improvement we can reliably achieve without overfitting**.

The **reason** to get a **better result** is because **stronger model performance directly improves the reliability of the predictions**. In a **medical-style dataset** like this one, even **small gains** in accuracy, precision, recall, or F1 score **can translate into more consistent and trustworthy classifications.** Better results also mean the model is capturing the underlying patterns more effectively rather than relying on noise or chance. Even if improvements are small or hard to achieve with limited data, the goal is always to build the most generalizable and stable model possible.

Therefore, to get a better performance, I tried class imbalance and calibration, and using the SMOTE approach. In this implementation, SMOTE was added to the pipelines using SMOTE (random_state = 42). I didn't explicitly change or tune any of SMOTE's internal parameters (like k_neighbors). The hyperparameter tuning (RandomizedSearchCV) was focused on the RandomForestClassifier itself, not on SMOTE's parameters. As observed in the summary, applying SMOTE with these default settings didn't lead to a noticeable improvement in model performance on the test set for this particular dataset.

Key findings after SMOTE:

SMOTE-tuned Random Forest:
Hyperparameter tuning of the RandomForest with SMOTE resulted in a **RandomForest_SMOTE_tuned** model achieving an F1-score of 0.72 on the test set, which is identical to the non-SMOTE tuned version. The best cross-validation weighted F1 score during its **RandomizedSearchCV** was 0.838.

SMOTE-enabled Stacking Ensemble:
The Stacking MLP model, when utilizing SMOTE-enabled base models (**Stacking_MLP_SMOTE**), yielded an F1-score of 0.63, matching its non-SMOTE counterpart. A **ConvergenceWarning** for the MLPClassifier persisted even after increasing **max_iter** to 2000, indicating potential training challenges for the meta-learner.

Conclusion: NO Apparent Improvement from SMOTE
Across all evaluated models (LinearSVC, Decision Tree, RandomForest, KNN, tuned RandomForest, and Stacking MLP), the addition of SMOTE to the training pipeline did not lead to a measurable improvement in the F1-score on the test set, based on the rounded results.