An Efficient Computational Risk Prediction Model of Heart Diseases Based on Dual-Stage Stacked Machine Learning Approaches

Problem Statement:

The study addresses the increasing global mortality due to cardiovascular diseases (CVD). It aims to develop an efficient machine learning model that accurately predicts the risk of heart diseases to assist in early diagnosis and reduce fatalities.

Approach:

The researchers built a dual-stage stacked machine learning model using patient data and five classifiers. The three best-performing models (RandomForest, ExtremeGradientBoost, DecisionTree) were fine-tuned and combined in a stacking technique to enhance prediction accuracy. Hyperparameter tuning was done using RandomizedSearchCV and GridSearchCV. The model was cross-validated and tested on two datasets for robustness.

Outcome:

The model achieved high accuracy (96%), excellent recall (0.98), and a low false-negative rate (below 1%). It demonstrated strong reliability and performed well even with different datasets, highlighting its potential for heart disease risk prediction. Future improvements could involve feature selection and deep learning models.

Dual-Stage Stacked Machine Learning Approach is a method that involves two main steps for improving model performance by leveraging multiple machine learning models. Here's how it works:

1. First Stage (Base Models)

- Multiple machine learning models (classifiers) are trained independently using the same dataset.
- In this study, five classifiers were used, and the top-performing ones (e.g., RandomForest, ExtremeGradientBoost, DecisionTree) were selected.
- Each model makes its own predictions based on the input data.

2. Second Stage (Stacking)

- The predictions from the first stage (from the individual models) are then combined as new inputs for another model (meta-model).
- This meta-model (stacked model) learns from the output of the first-stage models and makes the final prediction.
- Stacking helps to combine the strengths of different models, leading to more accurate and robust predictions.

This dual-stage stacking approach allows for the integration of multiple models' insights, leading to improved accuracy and a more generalized solution. It also reduces overfitting and leverages the diverse strengths of various algorithms.

The models used in the **dual-stage stacked machine learning approach** in the study are:

1. **Random Forest (RF)** – A popular ensemble learning method using multiple decision trees.
2. **Extreme Gradient Boosting (XGB)** – A powerful boosting algorithm that builds models sequentially to correct errors.
3. **Decision Tree (DT)** – A simple tree-based model used for classification and regression tasks.

These three models were fine-tuned and selected for the stacking ensemble in the second stage to achieve better prediction accuracy.

In the **dual-stage stacked machine learning approach**, the **base models** are the individual models trained during the first stage. In this study, the base models used are:

1. **Random Forest (RF)**
2. **Extreme Gradient Boosting (XGB)**
3. **Decision Tree (DT)**
4. **Support Vector Machine (SVM)**
5. **K-Nearest Neighbors (KNN)**

These models generate initial predictions, and the outputs are then combined in the second stage using the stacking technique to improve the final prediction performance. The top three models (RF, XGB, DT) were chosen for stacking in this case.

Effective Feature Engineering Technique for Heart Disease Prediction With Machine Learning

Problem Statement:

Heart failure is a chronic condition affecting millions worldwide, and early detection is crucial to save lives. The study aims to develop a machine learning model that can accurately predict heart failure using patient health data.

Approach:

The researchers used nine machine learning algorithms and introduced a new feature engineering technique called **Principal Component Heart Failure (PCHF)**. This technique selects the eight most important health features from patient data to improve prediction accuracy. The models were evaluated through cross-validation, and their performances were compared.

Outcome:

The **Decision Tree (DT)** model outperformed the others, achieving a perfect accuracy score of 100%. The proposed method shows great promise in early heart failure detection, offering a powerful tool for medical diagnosis.

It looks like you have implemented various machine learning models, including Naive Bayes, Support Vector Machine (SVM), K-Nearest Neighbor (KNN), Gradient Boosting, and AdaBoost classifiers, and evaluated them using metrics such as accuracy, precision, recall, F1 score, and confusion matrices. Additionally, you've utilized cross-validation with different values for `k` (3, 5, and 7 splits) to validate model performance.

You also generated bar plots to visually compare performance metrics across classifiers.

Here are a few improvements and suggestions for better organization and understanding:

1. Data Preprocessing

Before applying the models, it's crucial to check and handle missing values and perform feature scaling for models like SVM and KNN, which are sensitive to scales.

```
python
Copy code
# Check for missing values and handle if any
df.isnull().sum()

# Scaling the data (since SVM and KNN can be sensitive to feature scales)
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

2. Organize Model Training and Evaluation

Instead of repeating code for each classifier, you can create a function to train, predict, and evaluate models.

```
python
Copy code
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score, confusion_matrix

def evaluate_model(model, X_train, X_test, y_train, y_test):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_pred)
    cm = confusion_matrix(y_test, y_pred)

    print(f'Accuracy: {accuracy:.4f}')
    print(f'Precision: {precision:.4f}')
    print(f'Recall: {recall:.4f}')
    print(f'F1 Score: {f1:.4f}')
    print(f'ROC AUC: {auc:.4f}')
    print('Confusion Matrix:')
    print(cm)
```

You can then call this function for each classifier:

python
Copy code
```
# Naive Bayes
classifier = GaussianNB()
evaluate_model(classifier, X_train, X_test, y_train, y_test)

# SVM
svm = SVC()
evaluate_model(svm, X_train, X_test, y_train, y_test)

# KNN
knn = KNeighborsClassifier(n_neighbors=3)
evaluate_model(knn, X_train, X_test, y_train, y_test)

# Gradient Boosting
gbc = GradientBoostingClassifier()
evaluate_model(gbc, X_train, X_test, y_train, y_test)

# AdaBoost
abc = AdaBoostClassifier(n_estimators=100, random_state=0)
evaluate_model(abc, X_train, X_test, y_train, y_test)
```

3. Cross-Validation

You can also create a function to handle cross-validation for each model, which makes it easier to compare performance across different `k` values.

python
Copy code
```
def cross_val_evaluate(model, X, y, k):
    k_folds = KFold(n_splits=k)
    scores = cross_val_score(model, X, y, cv=k_folds)
    predicted = cross_val_predict(model, X, y, cv=k)

    accuracy = accuracy_score(y, predicted)
    f1 = f1_score(y, predicted, average='weighted')
    precision = precision_score(y, predicted, average='weighted')
    recall = recall_score(y, predicted, average='weighted')

    print(f'Cross-Validation (k={k}):')
    print(f'Accuracy: {accuracy:.4f}')
    print(f'Precision: {precision:.4f}')
    print(f'Recall: {recall:.4f}')
    print(f'F1 Score: {f1:.4f}')
    print('Scores:', scores)
    print('-' * 40)
```

Use this function to cross-validate each model:

python
Copy code
```
# Naive Bayes
cross_val_evaluate(classifier, X_scaled, y, 5)

# SVM
cross_val_evaluate(svm, X_scaled, y, 5)

# KNN
cross_val_evaluate(knn, X_scaled, y, 5)
```

```
# Gradient Boosting
cross_val_evaluate(gbc, X_scaled, y, 5)

# AdaBoost
cross_val_evaluate(abc, X_scaled, y, 5)
```

4. Visualizations

Your visualizations for comparing metrics across classifiers are well done. You can further improve by adding axis labels, gridlines, and ensuring the colors are distinct for better readability.

Let me know if you'd like further refinements or additional insights!

It looks like you have implemented various machine learning models, including Naive Bayes, Support Vector Machine (SVM), K-Nearest Neighbor (KNN), Gradient Boosting, and AdaBoost classifiers, and evaluated them using metrics such as accuracy, precision, recall, F1 score, and confusion matrices. Additionally, you've utilized cross-validation with different values for `k` (3, 5, and 7 splits) to validate model performance.

You also generated bar plots to visually compare performance metrics across classifiers.

Here are a few improvements and suggestions for better organization and understanding:

1. Data Preprocessing

Before applying the models, it's crucial to check and handle missing values and perform feature scaling for models like SVM and KNN, which are sensitive to scales.

```
python
Copy code
# Check for missing values and handle if any
df.isnull().sum()

# Scaling the data (since SVM and KNN can be sensitive to feature scales)
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

2. Organize Model Training and Evaluation

Instead of repeating code for each classifier, you can create a function to train, predict, and evaluate models.

```
python
Copy code
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score, confusion_matrix

def evaluate_model(model, X_train, X_test, y_train, y_test):
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    auc = roc_auc_score(y_test, y_pred)
    cm = confusion_matrix(y_test, y_pred)

    print(f'Accuracy: {accuracy:.4f}')
    print(f'Precision: {precision:.4f}')
    print(f'Recall: {recall:.4f}')
    print(f'F1 Score: {f1:.4f}')
    print(f'ROC AUC: {auc:.4f}')
    print('Confusion Matrix:')
    print(cm)
```

You can then call this function for each classifier:

python
Copy code
```
# Naive Bayes
classifier = GaussianNB()
evaluate_model(classifier, X_train, X_test, y_train, y_test)

# SVM
svm = SVC()
evaluate_model(svm, X_train, X_test, y_train, y_test)

# KNN
knn = KNeighborsClassifier(n_neighbors=3)
evaluate_model(knn, X_train, X_test, y_train, y_test)

# Gradient Boosting
gbc = GradientBoostingClassifier()
evaluate_model(gbc, X_train, X_test, y_train, y_test)

# AdaBoost
abc = AdaBoostClassifier(n_estimators=100, random_state=0)
evaluate_model(abc, X_train, X_test, y_train, y_test)
```

3. Cross-Validation

You can also create a function to handle cross-validation for each model, which makes it easier to compare performance across different `k` values.

python
Copy code
```
def cross_val_evaluate(model, X, y, k):
    k_folds = KFold(n_splits=k)
    scores = cross_val_score(model, X, y, cv=k_folds)
    predicted = cross_val_predict(model, X, y, cv=k)

    accuracy = accuracy_score(y, predicted)
    f1 = f1_score(y, predicted, average='weighted')
    precision = precision_score(y, predicted, average='weighted')
    recall = recall_score(y, predicted, average='weighted')

    print(f'Cross-Validation (k={k}):')
    print(f'Accuracy: {accuracy:.4f}')
    print(f'Precision: {precision:.4f}')
    print(f'Recall: {recall:.4f}')
    print(f'F1 Score: {f1:.4f}')
    print('Scores:', scores)
    print('-' * 40)
```

Use this function to cross-validate each model:

python
Copy code
```
# Naive Bayes
cross_val_evaluate(classifier, X_scaled, y, 5)

# SVM
cross_val_evaluate(svm, X_scaled, y, 5)

# KNN
cross_val_evaluate(knn, X_scaled, y, 5)
```

```
# Gradient Boosting
cross_val_evaluate(gbc, X_scaled, y, 5)

# AdaBoost
cross_val_evaluate(abc, X_scaled, y, 5)
```

4. Visualizations

Your visualizations for comparing metrics across classifiers are well done. You can further improve by adding axis labels, gridlines, and ensuring the colors are distinct for better readability.

The project you've created is a comprehensive comparison of machine learning models using the **Heart Disease dataset**. The key objective is to assess and compare the performance of multiple classification algorithms, namely Naive Bayes, Support Vector Machine (SVM), K-Nearest Neighbor (KNN), Gradient Boosting Classifier (GBC), and AdaBoost, by evaluating their accuracy, precision, recall, and F1 score. Let's go through it step by step:

Step 1: Import Libraries and Load the Dataset

First, you import the necessary libraries for data manipulation, visualization, and machine learning model evaluation, such as `pandas`, `numpy`, `seaborn`, `matplotlib`, and Scikit-learn's `metrics` and `model_selection` modules. You also load the **Heart Disease dataset** into a pandas DataFrame.

- **Libraries**: Used for visualization, model training, and evaluation.
- **Dataset**: Contains information about patients' heart conditions, including features like `age`, `sex`, `cp`, `trestbps`, etc.

Step 2: Data Preprocessing

You prepare your feature matrix `X` and target variable `y`:

- **Feature columns**: Age, sex, chest pain type (`cp`), resting blood pressure (`trestbps`), cholesterol levels (`chol`), and more.
- **Target column**: The presence or absence of heart disease.

You split the dataset into **training (70%) and test sets (30%)** using the `train_test_split` function to ensure that you have data for training and testing the models.

Step 3: Model Training and Evaluation

You implement and evaluate the following classifiers:

1. **Naive Bayes** (GaussianNB)
2. **Support Vector Machine** (SVC)
3. **K-Nearest Neighbor** (KNN)
4. **Gradient Boosting Classifier** (GBC)
5. **AdaBoost Classifier**

For each model, the following steps are taken:

- **Fit the model** using the training data (`X_train`, `y_train`).
- **Predict** the test data (`X_test`).
- Calculate evaluation metrics:
  - **Accuracy**: Measures the percentage of correct predictions.
  - **ROC AUC Score**: Area Under the Receiver Operating Characteristic curve.
  - **Precision**: Measures how many selected items are relevant.
  - **Recall**: Measures how many relevant items are selected.
  - **F1 Score**: Harmonic mean of precision and recall.
  - **Confusion Matrix**: To visualize the performance across true/false positives and negatives.

Each model's performance metrics are printed to assess how well they predict heart disease.

Step 4: Visualizing Results

You create bar plots to compare the performance of the classifiers in terms of:

- **Precision, Recall, and F1 Score**: This shows how each model performs across these metrics.
- **Accuracy**: Bar chart visualizing the accuracy of each classifier.

Step 5: Cross-Validation

You use **K-Fold Cross-Validation** for more robust performance estimation. By splitting the dataset into `k` subsets (folds), each model is trained and tested `k` times. You use `k=3`, `k=5`, and `k=7` to evaluate models multiple times.

The average performance (accuracy, precision, recall, and F1 score) across all the folds is calculated and printed.

Step 6: Cross-Validation for Each Classifier

The cross-validation process is applied to:

- **Naive Bayes** (GaussianNB)
- **Support Vector Machine** (SVC)
- **K-Nearest Neighbor** (KNN)
- **Gradient Boosting Classifier** (GBC)
- **AdaBoost**

Each classifier is evaluated with `k=3`, `k=5`, and `k=7` cross-validation, and the corresponding performance metrics (accuracy, precision, recall, and F1 score) are printed and plotted.

Step 7: Final Visualizations with Cross-Validation

You create another set of bar plots to visualize how each classifier performs after applying cross-validation (`k=5` and `k=7`). You compare the performance in terms of precision, recall, F1 score, and accuracy for different values of `k`.

Outcome:

The final outcome of your project is a comprehensive performance comparison of various machine learning models on the **Heart Disease dataset**. You observe the following key results:

- **Naive Bayes**, **SVM**, **KNN**, **Gradient Boosting**, and **AdaBoost** are compared based on accuracy, precision, recall, and F1 score.
- **Gradient Boosting** and **AdaBoost** perform better in terms of accuracy and F1 score across the cross-validation folds.
- The results give insights into which model is most suitable for predicting heart disease.

Final Visualizations: You generate bar charts showing the precision, recall, F1 score, and accuracy for each model using different values of `k` for cross-validation.

This project helps identify the best-performing machine learning models for classifying heart disease, which could be crucial in a real-world medical prediction scenario.

What is Validation techniques ?

Validation techniques in machine learning are methods used to evaluate how well a model will generalize to unseen data, ensuring that it performs well not just on the training data but also on new, previously unseen data. The goal of validation is to prevent overfitting (fitting too closely to the training data) and to ensure that the model performs robustly in different scenarios.

Here are the most common validation techniques:

1. Holdout Validation (Train-Test Split)

This is the simplest and most commonly used technique.

- **Process**: The dataset is split into two parts: a training set and a test set (commonly 70-30% or 80-20% split).
    - **Training set**: Used to train the model.
    - **Test set**: Used to evaluate the model's performance.

  The model is trained on the training set and then tested on the test set to check its accuracy and other performance metrics.

- **Advantages**: Simple and quick.
- **Disadvantages**: Performance evaluation might be biased or unreliable depending on how the data was split. A single test set might not be representative of the entire dataset.

2. K-Fold Cross-Validation

This is a more robust and commonly used technique in machine learning.

- **Process**: The dataset is split into `k` equal parts (folds). The model is trained `k` times, each time using `k-1` folds as the training set and the remaining fold as the test set. The final performance is the average of all `k` tests.

  For example, in **5-Fold Cross-Validation**, the dataset is divided into 5 parts. The model is trained 5 times, each time leaving out one fold as the test set and using the remaining 4 folds as the training set. The performance metrics are averaged over the 5 runs.

- **Advantages**:
    - Reduces bias in the performance estimate.
    - More reliable because every data point is used both for training and testing.
    - Suitable for smaller datasets.
- **Disadvantages**: More computationally expensive compared to simple train-test split because the model is trained multiple times.

3. Stratified K-Fold Cross-Validation

This is a variant of K-Fold Cross-Validation where the folds are stratified to ensure that each fold has approximately the same proportion of target classes as the original dataset.

- **Process**: Similar to K-Fold Cross-Validation, but with the added condition that the distribution of classes is preserved in each fold.
- **Advantages**: Especially useful when the dataset is imbalanced (i.e., when one class occurs much more frequently than others).

4. Leave-One-Out Cross-Validation (LOOCV)

This is an extreme case of K-Fold Cross-Validation where `k` is equal to the number of samples in the dataset.

- **Process**: For each iteration, a single sample is left out as the test set, and the remaining `n-1` samples are used as the training set. This is repeated for all `n` samples.
- **Advantages**: Very thorough because it uses every data point for testing.
- **Disadvantages**: Computationally expensive for large datasets, as the model needs to be trained `n` times.

5. Time Series Cross-Validation

This is a special case of cross-validation designed for time series data.

- **Process**: Time series data is not randomly split because of the temporal nature. Instead, a rolling or expanding window approach is used. For example:
    - **Expanding Window**: You start with a small training set and a test set just ahead of it in time. With each iteration, the training set grows by including more historical data, and the test set moves forward in time.
    - **Rolling Window**: The training window size is fixed, and the window slides forward in time for each iteration, keeping the training set the same size.
- **Advantages**: Preserves the temporal ordering of the data, which is crucial for time series forecasting.
- **Disadvantages**: Depending on the window size, it can lead to some data being left out of the training process.

6. Bootstrap Validation

This technique involves sampling the dataset with replacement to create multiple different training datasets.

- **Process**: Random samples are taken from the original dataset with replacement to create a "bootstrap sample" of the same size. The model is trained on this bootstrap sample, and the remaining data points (those not included in the sample) are used as the test set. This process is repeated several times.
- **Advantages**: Useful when the dataset is small and you want to create multiple variations of the dataset for training and testing.
- **Disadvantages**: Some samples are repeated in the training set, which can introduce some bias.

7. Nested Cross-Validation

This method is used when hyperparameter tuning is part of the model-building process. It ensures that model selection (including hyperparameter tuning) does not overfit the data.

- **Process**: It involves two loops of cross-validation:
  - **Outer loop**: Used for model performance evaluation.
  - **Inner loop**: Used for hyperparameter tuning within each fold of the outer loop.
- **Advantages**: Prevents overfitting during hyperparameter tuning.
- **Disadvantages**: Computationally expensive because it involves multiple layers of cross-validation.

Conclusion

The choice of validation technique depends on the size and nature of the dataset:

- For smaller datasets, **K-Fold Cross-Validation** or **Stratified K-Fold Cross-Validation** is preferred.
- For imbalanced datasets, **Stratified K-Fold Cross-Validation** works best.
- For time-series data, **Time Series Cross-Validation** ensures that the temporal structure is maintained.
- For complex models that require hyperparameter tuning, **Nested Cross-Validation** can be effective.

Validation is crucial in machine learning to ensure the model performs well on new, unseen data and avoids overfitting.