Sri Sivasubramaniya Nadar College of Engineering, Chennai
(An autonomous Institution affiliated to Anna University)

Degree & Branch	B.E. Computer Science & Engineering	Semester	V
Subject Code & Name	ICS1512 & Machine Learning Algorithms Laboratory		
Academic year	2025-2026 (Odd)	Batch:2023-2028	**Due date: 25/08/2025**

Experiment 3: Ensemble Prediction and Decision Tree Model Evaluation

Aim and Objective:

To build classifiers such as Decision Tree, AdaBoost, Gradient Boosting, XGBoost, Random Forest, and Stacked Models (using SVM, Na¨ıve Bayes, Decision Tree) and evaluate their performance through 5-Fold Cross-Validation and hyperparameter tuning.

Libraries used:

- Numpy
- Pandas
- Matplotlib
- Seaborn
- sklearn
- xgboost

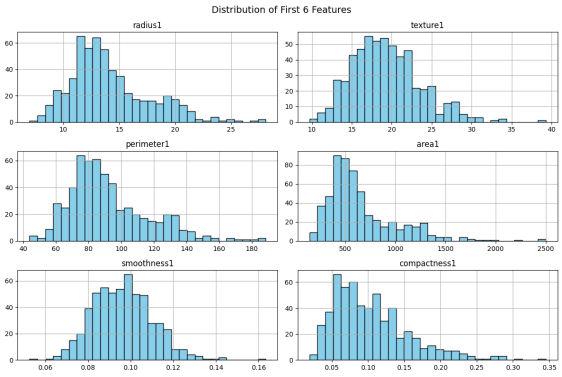
IMPORTING DATASET + EDA + FEATURE CORRELATION AND OTHER PLOTS:

```
# Importing dataset and required libraries
from ucimlrepo import fetch_ucirepo
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split, GridSearchCV, StratifiedKFold, cross_
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.pipeline import Pipeline
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    roc_auc_score, roc_curve, precision_recall_curve, average_precision_score,
```

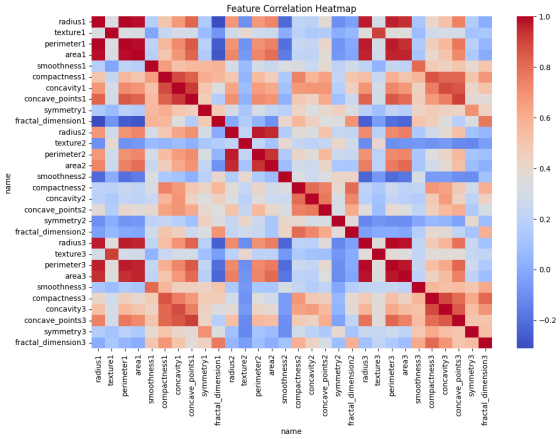
```
    confusion_matrix, classification_report
)
from sklearn.utils import parallel_backend

# fetch dataset
breast_cancer_wisconsin_diagnostic = fetch_ucirepo(id=17)

# data (as pandas dataframes)
X = breast_cancer_wisconsin_diagnostic.data.features
y = breast_cancer_wisconsin_diagnostic.data.targets

# variable information
print(breast_cancer_wisconsin_diagnostic.variables)

# Adding columns headers
df = pd.DataFrame(X, columns=breast_cancer_wisconsin_diagnostic.variables.name)

# Removing unnecessary features
df.drop(columns=['ID','Diagnosis'], inplace=True)

# Malignant -> 1, Benign -> 0
df['Diagnosis'] = y
df['Diagnosis'] = df['Diagnosis'].map({'M': 1, 'B': 0})

# Splitting features and target
X = df.drop(columns=['Diagnosis'])
y = df['Diagnosis']
columns = df.columns.tolist()
print(columns)

# Check for missing values
y.isnull().sum()

''' Since there is no missing missing values in the dataset, we don't apply unwanted operat

# Displaying first 5 records
print(df.head())

# 4. EDA
print("\nClass distribution:")
print(y.value_counts())

# Plot class distribution
sns.countplot(x='Diagnosis', data=df, palette = 'coolwarm')
plt.title("Class Distribution (Diagnosis)")
plt.show()

# Histograms of first 6 features
```

```
X.iloc[:, :6].hist(bins=30, figsize=(12, 8), color='skyblue', edgecolor='black')
plt.suptitle("Distribution of First 6 Features", fontsize=14)
plt.tight_layout()
plt.show()

# Feature correlation heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(X.corr(), cmap='coolwarm')
plt.title("Feature Correlation Heatmap")
plt.show()

# Train test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)
```

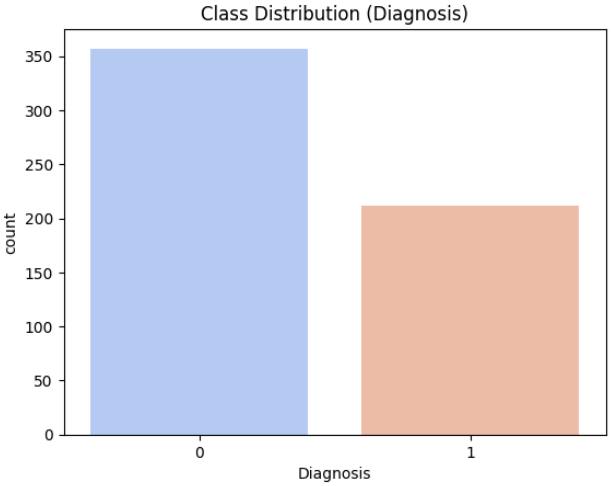
Note: This is common for all models



(a) Feature distribution



(b) Correlation Heatmap



(c) Class Distribution

Figure 1: Exploratory Data Analysis

FUNCTIONS FOR MODEL TEST + PERFORMANCE METRICS + K - FOLD CV:

```
# Predict & Performance Analysis
def test_model(best_estimator, X_test,y_test):
    best_estimator = grid.best_estimator_
    y_pred = best_estimator.predict(X_test)
    y_proba = best_estimator.predict_proba(X_test)[:, 1]
    return performance_metrics(y_test, y_pred, y_proba)

# Performance metrics
def performance_metrics(y_true, y_pred, y_proba):
    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred)
```

```
    rec = recall_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    roc_auc = roc_auc_score(y_test, y_proba)
    avg_precision = average_precision_score(y_test, y_proba)

    print("\nTest metrics")
    print(f"Accuracy: {acc:.4f}")
    print(f"Precision: {prec:.4f}")
    print(f"Recall: {rec:.4f}")
    print(f"F1: {f1:.4f}")
    print(f"ROC AUC: {roc_auc:.4f}")
    print(f"Avg Precision (PR AUC-ish): {avg_precision:.4f}")
    print("\nClassification report:\n", classification_report(y_test, y_pred))

    # Confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.title("Confusion Matrix")
    plt.show()

    # 7. Curves: ROC & Precision-Recall
    fpr, tpr, _ = roc_curve(y_test, y_proba)
    precision, recall, _ = precision_recall_curve(y_test, y_proba)

    plt.figure(figsize=(6,4))
    plt.plot(fpr, tpr, label=f"ROC AUC = {roc_auc:.3f}")
    plt.plot([0,1],[0,1],'--', alpha=0.5)
    plt.xlabel("FPR"); plt.ylabel("TPR")
    plt.title("ROC Curve")
    plt.legend(); plt.show()

# Define multiple metrics
def kfoldCV(best_estimator, X, y, cv):
    scoring = {
        'precision': 'precision',
        'recall': 'recall',
        'f1': 'f1'
    }

    # Run cross-validation
    with parallel_backend('threading'):
        cv_results = cross_validate(
            best_estimator, X, y, cv=cv,
            scoring=scoring,
            return_train_score=False,
            n_jobs=-1
```

```
        )

    n_splits = cv.get_n_splits()
    # Display per-fold results
    for i in range(n_splits):
        print(f"\nFold {i+1}: Precision: {cv_results['test_precision'][i]:.4f}   Recall: {

    # Display averages
    print("\n=== Average Metrics ===")
    print(f"Mean Precision: {cv_results['test_precision'].mean():.4f} ± {cv_results['test_
    print(f"Mean Recall:    {cv_results['test_recall'].mean():.4f} ± {cv_results['test_reca
    print(f"Mean F1 Score:  {cv_results['test_f1'].mean():.4f} ± {cv_results['test_f1'].std

def feature_selection(estimator, X_train, y_train, method="model", top_k=None):
    feature_names = X_train.columns

    # ---- Model-based importance ----
    if method == "model":
        if hasattr(estimator, "feature_importances_"):   # Trees, RF, XGB, GB
            importances = estimator.feature_importances_
        elif hasattr(estimator, "coef_"):                # Linear models
            importances = estimator.coef_.ravel()
        else:
            raise ValueError("Estimator does not support feature_importances_ or coef_. Use

    # ---- Permutation importance (model-agnostic) ----
    elif method == "permutation":
        perm = permutation_importance(estimator, X_train, y_train, n_repeats=20, random_sta
        importances = perm.importances_mean

    else:
        raise ValueError("method must be either 'model' or 'permutation'.")

    # Build importance DataFrame
    feat_imp = pd.DataFrame({
        "Feature": feature_names,
        "Importance": importances
    }).sort_values(by="Importance", ascending=False).reset_index(drop=True)

    # If top_k specified
    if top_k:
        feat_imp = feat_imp.head(top_k)

    # Plot
    plt.figure(figsize=(8, 6))
    plt.barh(feat_imp["Feature"][::-1], feat_imp["Importance"][::-1])
    plt.title(f"Top {top_k if top_k else len(feat_imp)} Features ({method} importance)")
    plt.xlabel("Importance")
```

```
    plt.ylabel("Feature")
    plt.show()

    return feat_imp
```

Note: These functions are common for all models

PIPELINE FOR DECISION TREES WITH GRIDSEARCHCV:

```
# Decision Tree Classifier with Hyperparameter Tuning
dtree = DecisionTreeClassifier(random_state=42)

param_grid = {
    'criterion': ['gini', 'entropy', 'log_loss'],
    'max_depth': [None, 5, 10, 15, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': [None, 'sqrt', 'log2']
}

# K-Fold Cross Validation
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

grid = GridSearchCV(
    estimator=dtree,
    param_grid=param_grid,
    cv=cv,
    scoring='accuracy',
    n_jobs=-1,
    verbose=2
)


with parallel_backend('threading'):
    grid.fit(X_train, y_train)

print("\nBest Parameters:", grid.best_params_)
print(f"Best CV Accuracy: {grid.best_score_ :.4f}")

# Test the decision tree model
best_estimator = grid.best_estimator_
test_model(best_estimator, X_test, y_test)
kfoldCV(best_estimator, X, y, cv)
```

PIPELINE FOR ADABOOST CLASSIFIER WITH GRIDSEARCHCV:

```
# AdaBoost Classifier with Hyperparameter Tuning
from sklearn.ensemble import AdaBoostClassifier
base_tree = DecisionTreeClassifier(random_state=42, max_depth=1)  # stump
adaboost = AdaBoostClassifier(estimator=base_tree, random_state=42)


param_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.001, 0.01, 0.1, 1],
    'estimator__max_depth': [1, 2, 3],  # tuning the base tree depth
    'estimator__criterion': ['gini', 'entropy']
}


cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

grid = GridSearchCV(
    estimator=adaboost,
    param_grid=param_grid,
    cv=cv,
    scoring='accuracy',
    n_jobs=-1,
    verbose=2
)

with parallel_backend('threading'):
    grid.fit(X_train, y_train)

print("\nBest Parameters:", grid.best_params_)
print("Best CV Accuracy:", grid.best_score_)

# Test the decision tree model
best_estimator = grid.best_estimator_
test_model(best_estimator, X_test, y_test)
kfoldCV(best_estimator, X, y, cv)
```

PIPELINE FOR GRADIENT BOOSTING CLASSIFIER WITH GRIDSEARCHCV:

```
# Gradient Boosting Classifier with Hyperparameter Tuning
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier(random_state=42)

param_grid = {
    'n_estimators': [100, 200, 300],     # number of boosting stages
    'learning_rate': [0.01, 0.1, 0.2],  # step size shrinkage
```

```
    'max_depth': [3, 4, 5],              # depth of individual trees
    'min_samples_split': [2, 5, 10],     # minimum samples to split a node
    'min_samples_leaf': [1, 3, 5],       # minimum samples per leaf
    'subsample': [1.0, 0.8]              # fraction of samples for each tree (stochastic bo
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

grid = GridSearchCV(
    estimator=gb,
    param_grid=param_grid,
    cv=cv,
    scoring='accuracy',
    n_jobs=-1,
    verbose=2
)

with parallel_backend('threading'):
    grid.fit(X_train, y_train)

print("\nBest Parameters:", grid.best_params_)
print("Best CV Accuracy:", grid.best_score_)

# Test the decision tree model
best_estimator = grid.best_estimator_
test_model(best_estimator, X_test, y_test)
kfoldCV(best_estimator, X, y, cv)
```

PIPELINE FOR XGBOOST CLASSIFIER WITH GRIDSEARCHCV:

```
# XGBoost Classifier with Hyperparameter Tuning
xgb = XGBClassifier(
    eval_metric='logloss',
    use_label_encoder=False,
    random_state=42
)

param_grid = {
    'n_estimators': [100, 200],
    'learning_rate': [0.05, 0.1],
    'max_depth': [3, 4],
    'subsample': [1.0, 0.8],
    'colsample_bytree': [1.0, 0.8],
    'gamma': [0, 0.1]
}
```

```
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

grid = GridSearchCV(
    estimator=xgb,
    param_grid=param_grid,
    cv=cv,
    scoring='accuracy',
    n_jobs=-1,
    verbose=2
)


grid.fit(X_train, y_train)

print("\nBest Parameters:", grid.best_params_)
print("Best CV Accuracy:", grid.best_score_)

# Test the decision tree model
best_estimator = grid.best_estimator_
test_model(best_estimator, X_test, y_test)
kfoldCV(best_estimator, X, y, cv)
```

PIPELINE FOR RANDOM FOREST CLASSIFIER WITH GRIDSEARCHCV:

```
# Random Forest Classifier with Hyperparameter Tuning
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(random_state=42)

param_grid = {
    'n_estimators': [100, 200],       # number of trees
    'max_depth': [None, 10, 20],      # depth of trees
    'min_samples_split': [2, 5],      # min samples to split a node
    'min_samples_leaf': [1, 2],       # min samples at a leaf
    'max_features': ['sqrt', 'log2']  # feature selection per split
}

cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

grid = GridSearchCV(
    estimator=rf,
    param_grid=param_grid,
    cv=cv,
    scoring='accuracy',
    n_jobs=-1,
    verbose=2
```

```
)

with parallel_backend('threading'):
    grid.fit(X_train, y_train)

print("\nBest Parameters:", grid.best_params_)
print("Best CV Accuracy:", grid.best_score_)

# Test the decision tree model
best_estimator = grid.best_estimator_
test_model(best_estimator, X_test, y_test)
kfoldCV(best_estimator, X, y, cv)
```

PIPELINES FOR STACKED ENSEMBLE MODELS:

SVM + NAIVE BAYES + DECISION TREES (BASE MODELS):
LOGISTIC REGRESSION (Final estimator):

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC

# Define base learners
svm_base = Pipeline([
    ("scaler", StandardScaler()),
    ("svc", SVC(probability=True, random_state=42))
])
nb_base = GaussianNB()
dt_base = DecisionTreeClassifier(random_state=42)

# Define stack (example: Logistic Regression as final estimator)
stack_clf = StackingClassifier(
    estimators=[
        ("svm", svm_base),
        ("nb", nb_base),
        ("dt", dt_base)
    ],
    final_estimator=LogisticRegression(max_iter=1000, random_state=42),
    n_jobs=-1
)

# Parameter grid (for meta-learner)
param_grid = {
    'final_estimator__C': [0.1, 1, 10],        # Regularization strength
```

```
    'final_estimator__penalty': ['l2'],        # Only l2 is supported with liblinear/saga
    'final_estimator__solver': ['lbfgs', 'saga']
}

# Cross-validation strategy
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Grid search
grid = GridSearchCV(
    estimator=stack_clf,
    param_grid=param_grid,
    cv=cv,
    scoring='accuracy',
    n_jobs=-1,
    verbose=2
)

# Fit with parallel backend
with parallel_backend('threading'):
    grid.fit(X_train, y_train)

print("\nBest Parameters:", grid.best_params_)
print("Best CV Accuracy:", grid.best_score_)

# Test performance
best_estimator = grid.best_estimator_
test_model(best_estimator, X_test, y_test)

# K-fold CV evaluation
kfoldCV(best_estimator, X, y, cv)

# Feature importance (only works if final estimator supports it)
with parallel_backend('threading'):
    feat_imp = feature_selection(best_estimator, X_train, y_train, method="permutation", t
    print(feat_imp)
```

RANDOM FOREST (Final estimator):

```
from sklearn.ensemble import RandomForestClassifier
stack_clf_rf = StackingClassifier(
    estimators=[
        ("svm", svm_base),
        ("nb", nb_base),
        ("dt", dt_base)
    ],
```

```
    final_estimator=RandomForestClassifier(random_state=42, n_jobs=-1),
    n_jobs=-1
)

param_grid_rf = {
    'final_estimator__n_estimators': [100, 200],
    'final_estimator__max_depth': [None, 10, 20],
    'final_estimator__min_samples_split': [2, 5],
    'final_estimator__min_samples_leaf': [1, 2],
    'final_estimator__max_features': ['sqrt', 'log2']
}

# Grid search
grid_rf = GridSearchCV(
    estimator=stack_clf_rf,
    param_grid=param_grid_rf,
    cv=cv,
    scoring='accuracy',
    n_jobs=-1,
    verbose=2
)

with parallel_backend('threading'):
    grid_rf.fit(X_train, y_train)

print("\nBest Parameters:", grid_rf.best_params_)
print("Best CV Accuracy:", grid_rf.best_score_)

best_estimator_rf = grid_rf.best_estimator_
test_model(best_estimator_rf, X_test, y_test)
kfoldCV(best_estimator_rf, X, y, cv)

with parallel_backend('threading'):
    feat_imp_rf = feature_selection(best_estimator_rf, X_train, y_train, method="permutati
print(feat_imp_rf)
```

SVM + KNN + DECISION TREES (BASE MODELS):
LOGISTIC REGRESSION (Final estimator):

```
from sklearn.neighbors import KNeighborsClassifier
# Define KNN base
knn_base = Pipeline([
    ("scaler", StandardScaler()),
    ("knn", KNeighborsClassifier())
])
```

```
# Define stack
stack_clf_knn = StackingClassifier(
    estimators=[
        ("svm", svm_base),
        ("dt", dt_base),
        ("knn", knn_base)
    ],
    final_estimator=LogisticRegression(max_iter=1000, random_state=42),
    n_jobs=-1
)

# Parameter grid (for meta-learner Logistic Regression)
param_grid_knn = {
    'final_estimator__C': [0.1, 1, 10],
    'final_estimator__penalty': ['l2'],
    'final_estimator__solver': ['lbfgs', 'saga']
}

# Grid search
grid_knn = GridSearchCV(
    estimator=stack_clf_knn,
    param_grid=param_grid_knn,
    cv=cv,
    scoring='accuracy',
    n_jobs=-1,
    verbose=2
)

with parallel_backend('threading'):
    grid_knn.fit(X_train, y_train)

print("\nBest Parameters:", grid_knn.best_params_)
print("Best CV Accuracy:", grid_knn.best_score_)

best_estimator_knn = grid_knn.best_estimator_
test_model(best_estimator_knn, X_test, y_test)
kfoldCV(best_estimator_knn, X, y, cv)

# Logistic regression doesn't provide feature_importances_ directly
with parallel_backend('threading'):
    feat_imp_knn = feature_selection(best_estimator_knn, X_train, y_train, method="permutat
    print(feat_imp_knn)
```

CV TABLE FOR ALL MODELS:

MODEL	FOLD 1	FOLD 2	FOLD 3	FOLD 4	FOLD 5	AVG ACC
DECISION TREE	0.9386	0.8772	0.9298	0.9035	0.9649	0.9192
ADABOOST	0.9912	0.9035	0.9474	0.9561	0.9735	0.9543
GRADIENT BOOSTING	0.9825	0.9298	0.9561	0.9912	0.9823	0.9684
XGBOOST	0.9649	0.9474	0.9561	0.9649	0.9558	0.9578
RANDOM FOREST	0.9649	0.9386	0.9649	0.9649	0.9646	0.9596
STACKED MODEL	0.9912	0.9386	0.9737	0.9737	0.9823	0.9719

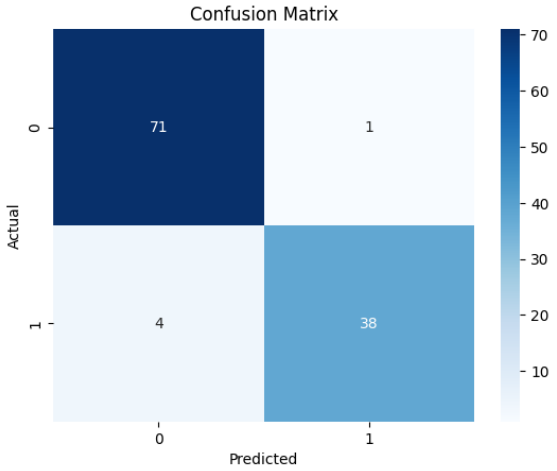
Table 1: Model performance across folds with average accuracy

STACKED ENSEMBLE MODEL TRIALS:

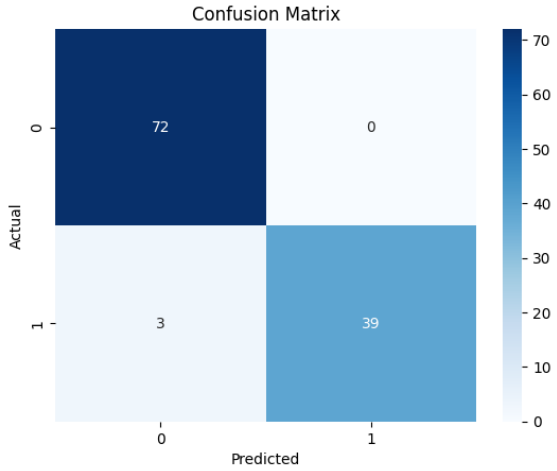
BASE MODELS	FINAL ESTIMATOR	ACC/F1
SVM + NB + DT	Logistic Regression	0.9825/0.9756
SVM + NB + DT	Random Forest	0.9649/0.9500
SVM + KNN + DT	Logistic Regression	0.9649/0.9500

Table 2: Stack Ensemble Hyperparamter tuning

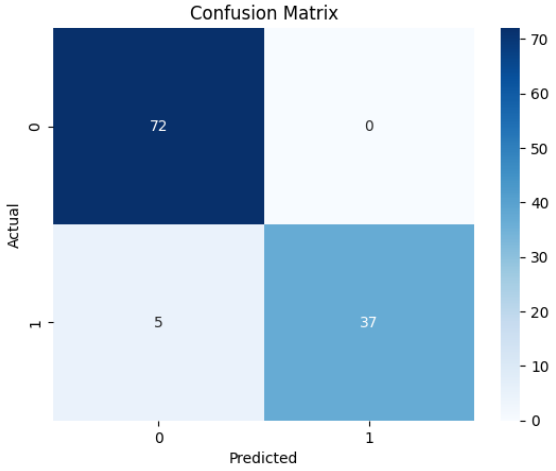
CONFUSION MATRICES:



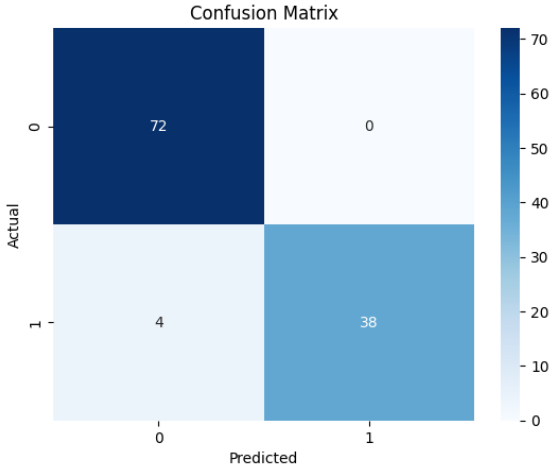
(a) Decision Tree Classifier



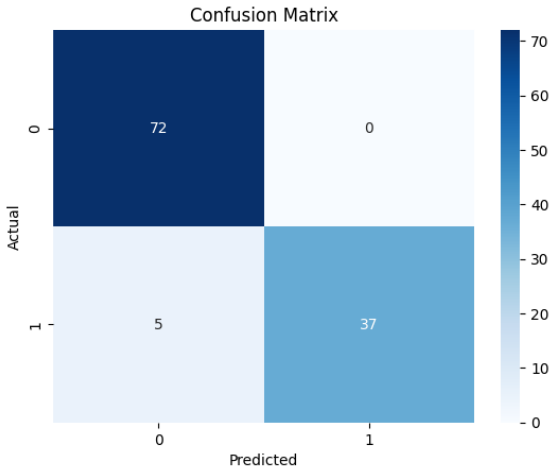
(b) AdaBoost Classifier



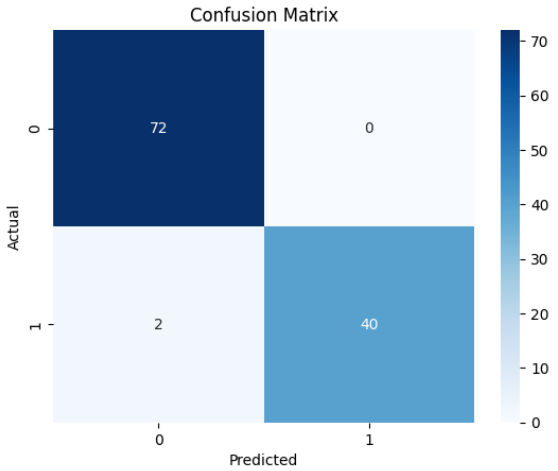
(a) Gradient Boosting Classifier



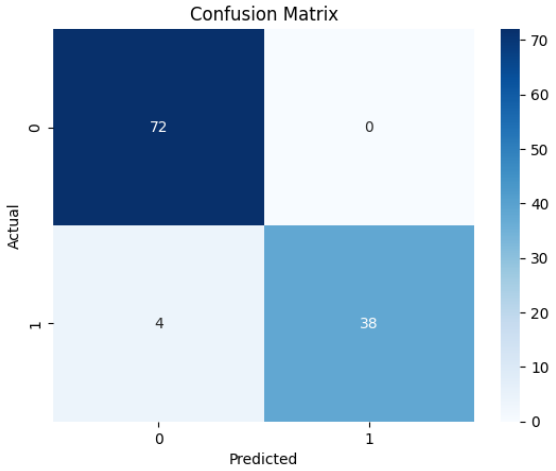
(b) XGBoost Classifier



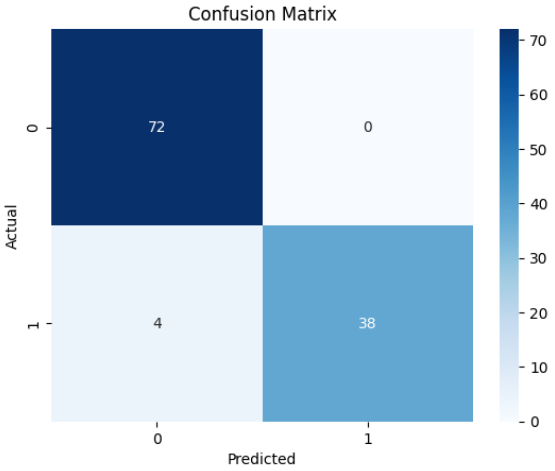
(c) Random Forest Classifier



(d) SEM(SVM + NB + DT + Logistic Regression)

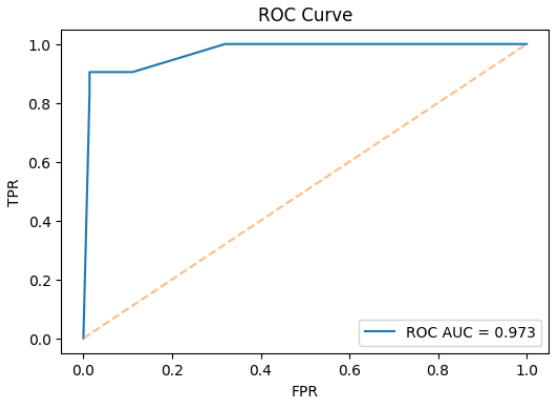


(e) SEM(SVM + NB + DT + Random Forest)

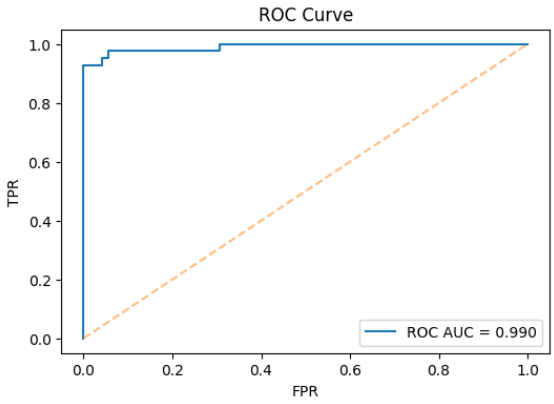


(f) SEM(SVM + KNN + DT + Logistic Regression)

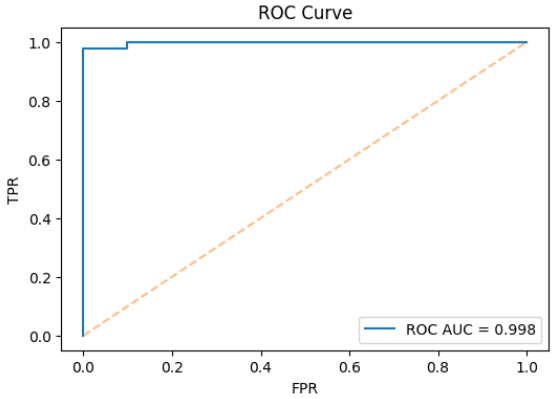
ROC CURVES:



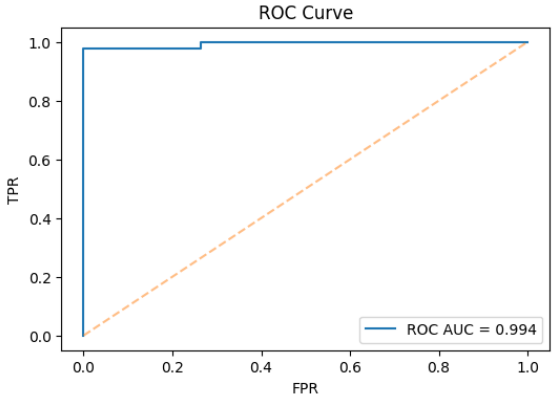
(a) Decision Tree Classifier



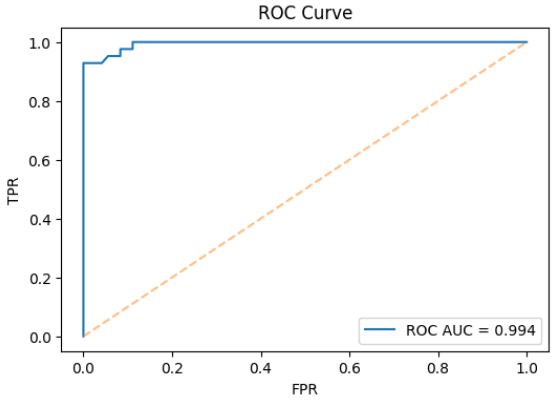
(b) AdaBoost Classifier



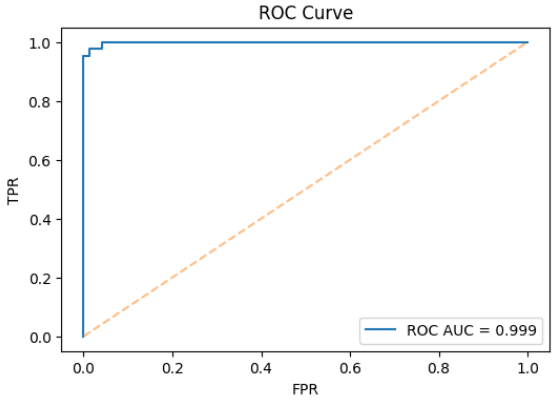
(c) Gradient Boosting Classifier



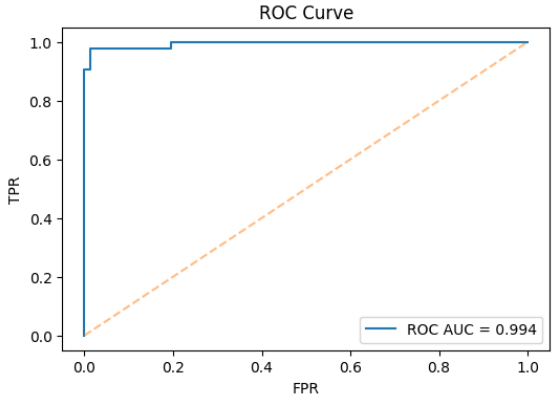
(d) XGBoost Classifier



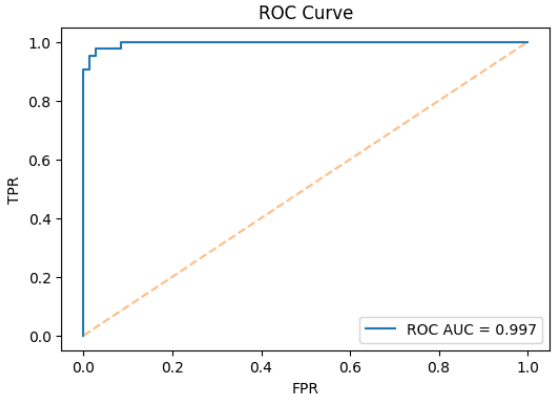
(e) Random Forest Classifier



(f) SEM(SVM + NB + DT + Logistic)



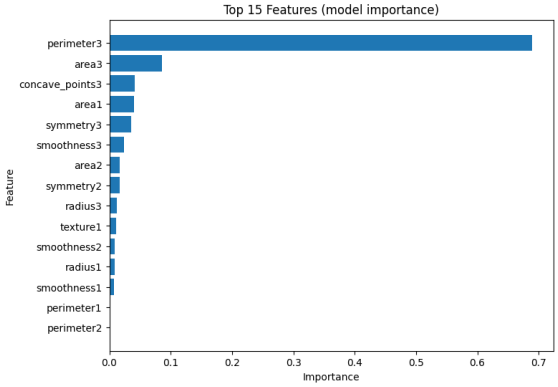
(a) SEM(SVM + NB + DT + Random Forest)



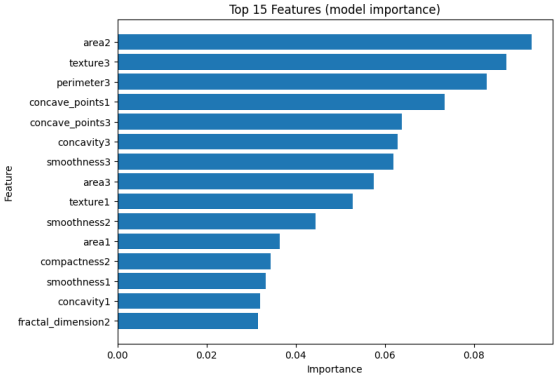
(b) SEM(SVM + KNN + DT + Logistic)

Figure 5: Comparison of different classifiers

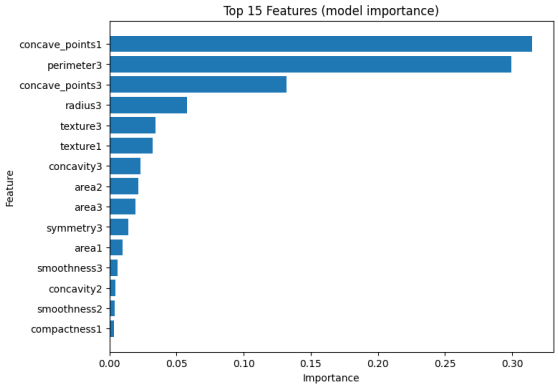
FEATURE SELECTION:



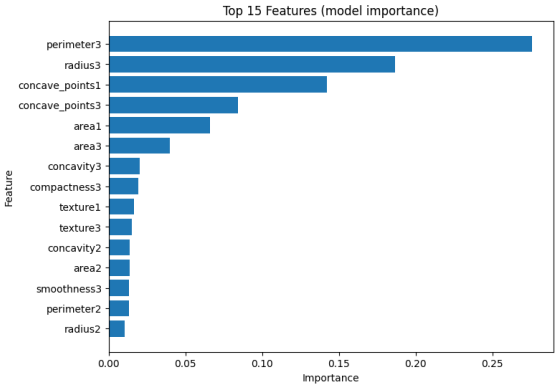
(a) Decision Tree Classifier



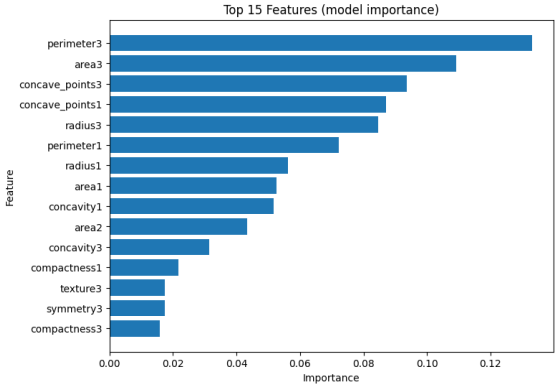
(b) AdaBoost Classifier



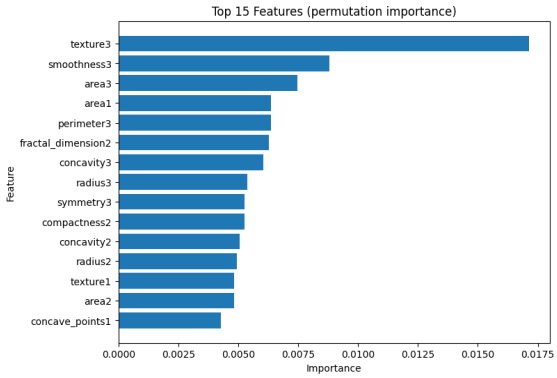
(c) Gradient Boosting Classifier



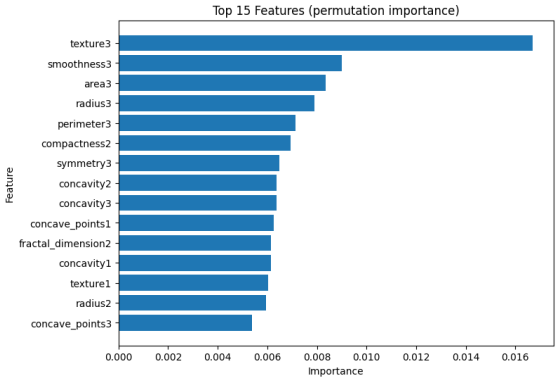
(d) XGBoost Classifier



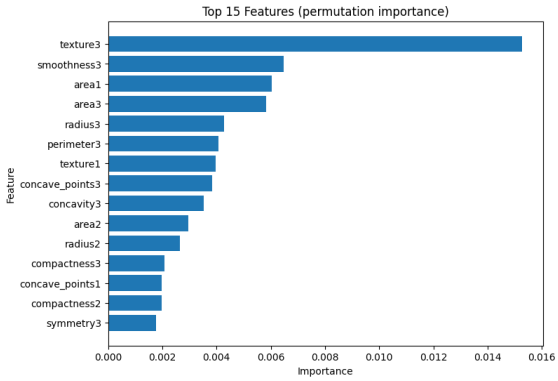
(a) Random Forest Classifier



(b) SEM(SVC + NB + DT + Logistic)



(c) SEM(SVC + NB + DT + Random Forest)



(d) SEM(SVC + KNN + DT + Logistic)

Figure 7: Feature Selection of different classifiers

Observations and Conclusion:

1. From the CV Table, we can see that **Stacked Ensemble Model with SVM, Naive Bayes and Decision Tree model as base and Logistic Regression as final estimator** achieved the best validation accuracy of **0.9719**.

2. Decision Tree performs underperformed compared to the ensemble models with **average CV accuracy of 0.91** while others achieve an average of **0.95 accuracy**

3. Random Forest achieved the best validation accuracy of **0.967** at **max depth of 10** and **n-estimator of 200**.

4. **Stacked Ensemble Model with SVM, Naive Bayes and Decision Tree as base models and Logistic Regression as final estimator** showed better generalization with **best test accuracy of 0.9825** and **roc auc of 0.999**

5. From the above stacked ensembled models, it is evident that s**tacking improves the performance** over base models.