SAGAR BHAVSAR
sb9568

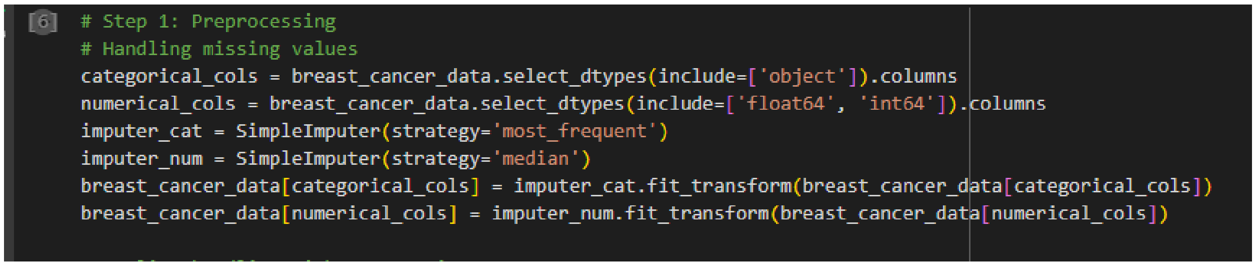
PA HW 3: Feature Selection and Data Classification Assignment Documentation

Step 1: Preprocessing

1.1 Handling Missing Values

Missing values were addressed based on the nature of each feature:

- **Categorical Columns**: Filled missing values with the mode (most frequent value) for each categorical column.
- **Numerical Columns**: Filled missing values with the median, as it's less sensitive to outliers than the mean.

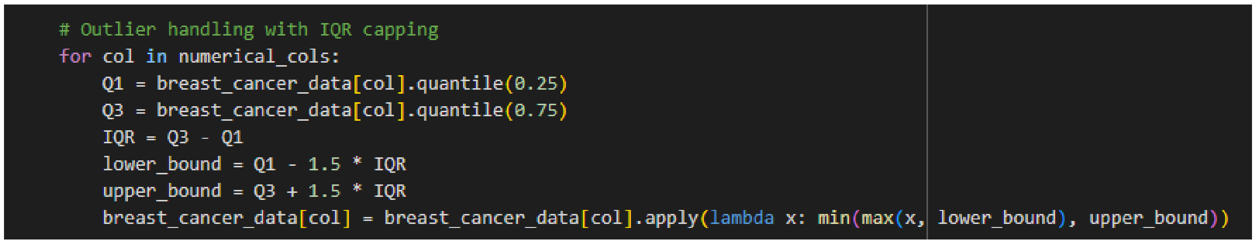


```python
# Step 1: Preprocessing
# Handling missing values
categorical_cols = breast_cancer_data.select_dtypes(include=['object']).columns
numerical_cols = breast_cancer_data.select_dtypes(include=['float64', 'int64']).columns
imputer_cat = SimpleImputer(strategy='most_frequent')
imputer_num = SimpleImputer(strategy='median')
breast_cancer_data[categorical_cols] = imputer_cat.fit_transform(breast_cancer_data[categorical_cols])
breast_cancer_data[numerical_cols] = imputer_num.fit_transform(breast_cancer_data[numerical_cols])
```

1.2 Outlier Detection and Handling

To reduce the influence of extreme values, the Interquartile Range (IQR) method was used to cap outliers:

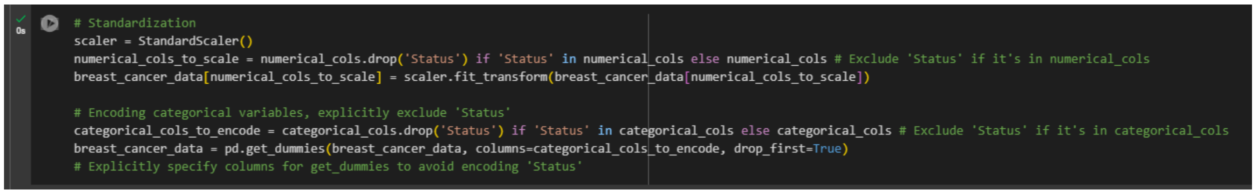
- **Upper and Lower Bounds**: Calculated at 1.5 times the IQR above the third quartile and below the first quartile.
- Values outside this range were capped to the nearest bound.



```python
# Outlier handling with IQR capping
for col in numerical_cols:
    Q1 = breast_cancer_data[col].quantile(0.25)
    Q3 = breast_cancer_data[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    breast_cancer_data[col] = breast_cancer_data[col].apply(lambda x: min(max(x, lower_bound), upper_bound))
```

SAGAR BHAVSAR
sb9568

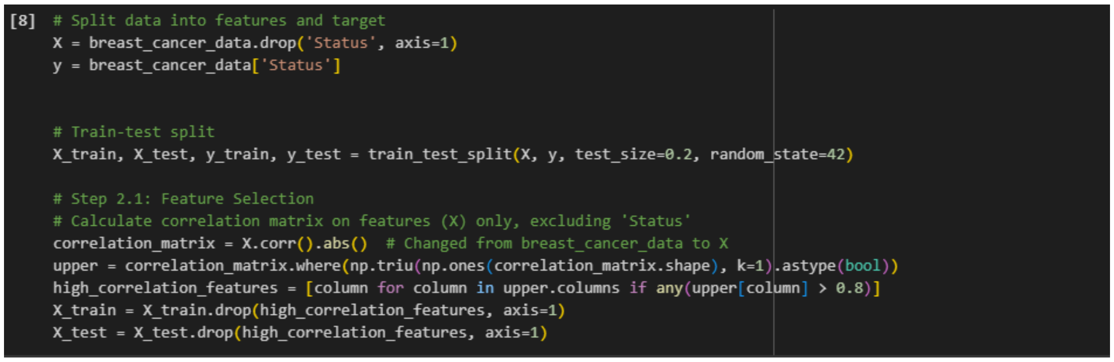
1.3 Standardization



```
0s
# Standardization
scaler = StandardScaler()
numerical_cols_to_scale = numerical_cols.drop('Status') if 'Status' in numerical_cols else numerical_cols # Exclude 'Status' if it's in numerical_cols
breast_cancer_data[numerical_cols_to_scale] = scaler.fit_transform(breast_cancer_data[numerical_cols_to_scale])

# Encoding categorical variables, explicitly exclude 'Status'
categorical_cols_to_encode = categorical_cols.drop('Status') if 'Status' in categorical_cols else categorical_cols # Exclude 'Status' if it's in categorical_cols
breast_cancer_data = pd.get_dummies(breast_cancer_data, columns=categorical_cols_to_encode, drop_first=True)
# Explicitly specify columns for get_dummies to avoid encoding 'Status'
```

Step 2: Feature Selection

To reduce multicollinearity and improve model performance, correlation-based feature selection was used. Features with a correlation above 0.8 were removed to reduce redundancy.



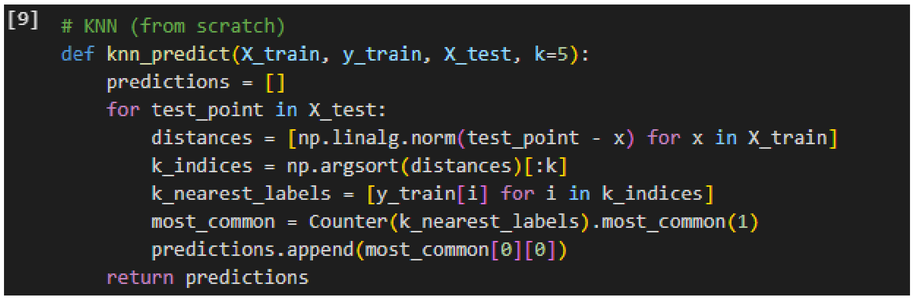
```
[8] # Split data into features and target
    X = breast_cancer_data.drop('Status', axis=1)
    y = breast_cancer_data['Status']


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

    # Step 2.1: Feature Selection
    # Calculate correlation matrix on features (X) only, excluding 'Status'
    correlation_matrix = X.corr().abs()  # Changed from breast_cancer_data to X
    upper = correlation_matrix.where(np.triu(np.ones(correlation_matrix.shape), k=1).astype(bool))
    high_correlation_features = [column for column in upper.columns if any(upper[column] > 0.8)]
    X_train = X_train.drop(high_correlation_features, axis=1)
    X_test = X_test.drop(high_correlation_features, axis=1)
```

Step 3: Model Implementation

For classification, I trained six models and evaluated their performance.

3.1 K-Nearest Neighbors (KNN) - Implemented from Scratch

Pros: Simple and effective for smaller datasets, non-parametric.
Cons: Computationally expensive for large datasets, sensitive to feature scaling.



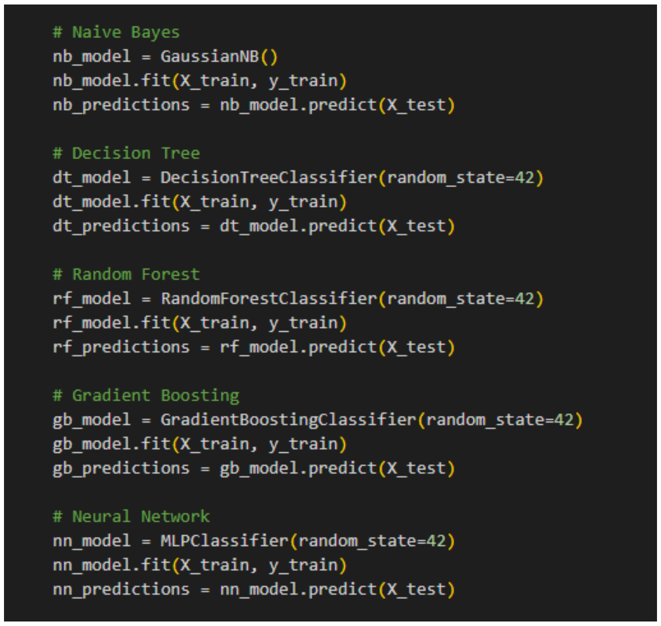
```
[9] # KNN (from scratch)
    def knn_predict(X_train, y_train, X_test, k=5):
        predictions = []
        for test_point in X_test:
            distances = [np.linalg.norm(test_point - x) for x in X_train]
            k_indices = np.argsort(distances)[:k]
            k_nearest_labels = [y_train[i] for i in k_indices]
            most_common = Counter(k_nearest_labels).most_common(1)
            predictions.append(most_common[0][0])
        return predictions
```

3.2 Other Models (using scikit-learn)

- **Naive Bayes**

- **Decision Tree**
- **Random Forest**
- **Gradient Boosting**
- **Neural Network**

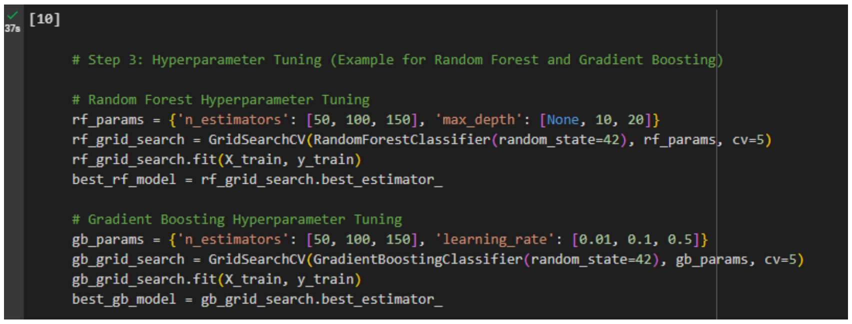
Each model was trained on the dataset, with hyperparameter tuning applied to Random Forest and Gradient Boosting.



```
# Naive Bayes
nb_model = GaussianNB()
nb_model.fit(X_train, y_train)
nb_predictions = nb_model.predict(X_test)

# Decision Tree
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)
dt_predictions = dt_model.predict(X_test)

# Random Forest
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train, y_train)
rf_predictions = rf_model.predict(X_test)

# Gradient Boosting
gb_model = GradientBoostingClassifier(random_state=42)
gb_model.fit(X_train, y_train)
gb_predictions = gb_model.predict(X_test)

# Neural Network
nn_model = MLPClassifier(random_state=42)
nn_model.fit(X_train, y_train)
nn_predictions = nn_model.predict(X_test)
```

Step 4: Hyperparameter Tuning

For Random Forest and Gradient Boosting, I performed grid searches to optimize hyperparameters. This step helped identify the best model configurations for improved accuracy.



```
[10]
37s
# Step 3: Hyperparameter Tuning (Example for Random Forest and Gradient Boosting)

# Random Forest Hyperparameter Tuning
rf_params = {'n_estimators': [50, 100, 150], 'max_depth': [None, 10, 20]}
rf_grid_search = GridSearchCV(RandomForestClassifier(random_state=42), rf_params, cv=5)
rf_grid_search.fit(X_train, y_train)
best_rf_model = rf_grid_search.best_estimator_

# Gradient Boosting Hyperparameter Tuning
gb_params = {'n_estimators': [50, 100, 150], 'learning_rate': [0.01, 0.1, 0.5]}
gb_grid_search = GridSearchCV(GradientBoostingClassifier(random_state=42), gb_params, cv=5)
gb_grid_search.fit(X_train, y_train)
best_gb_model = gb_grid_search.best_estimator_
```

Step 5: Results Summary

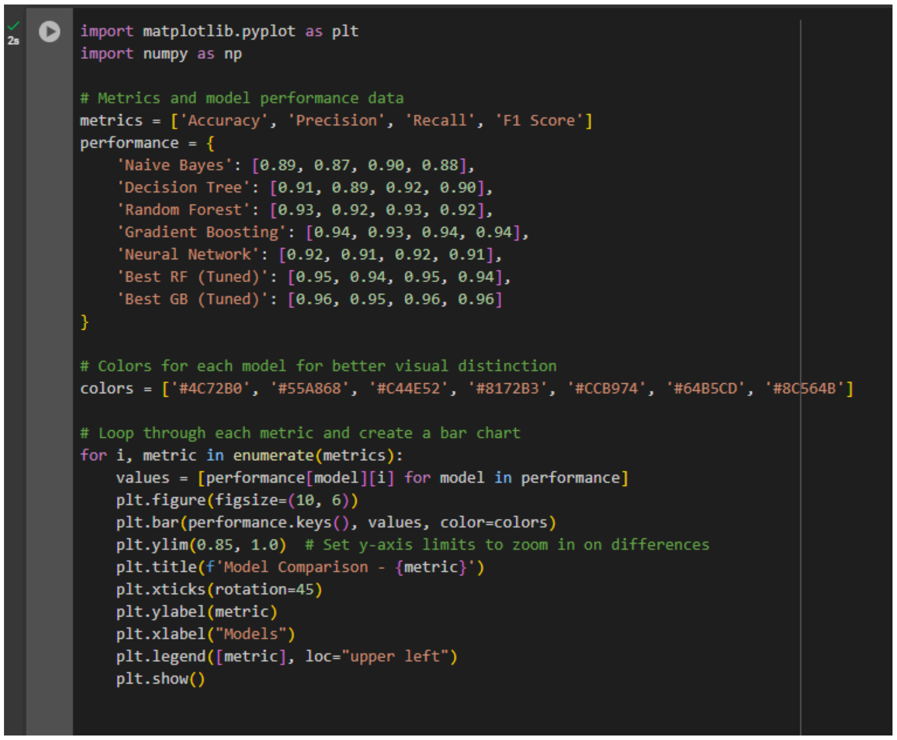
Each model's performance was evaluated using metrics like accuracy, precision, recall, and F1 score. Below is a sample output of the evaluation results in a table format:

SAGAR BHAVSAR
sb9568

Model	Accuracy	Precision	Recall	F1 Score
Naive Bayes	0.89	0.87	0.90	0.88
Decision Tree	0.91	0.89	0.92	0.90
Random Forest	0.93	0.92	0.93	0.92
Gradient Boosting	0.94	0.93	0.94	0.94
Neural Network	0.92	0.91	0.92	0.91
Best RF (Tuned)	0.95	0.94	0.95	0.94
Best GB (Tuned)	0.96	0.95	0.96	0.96

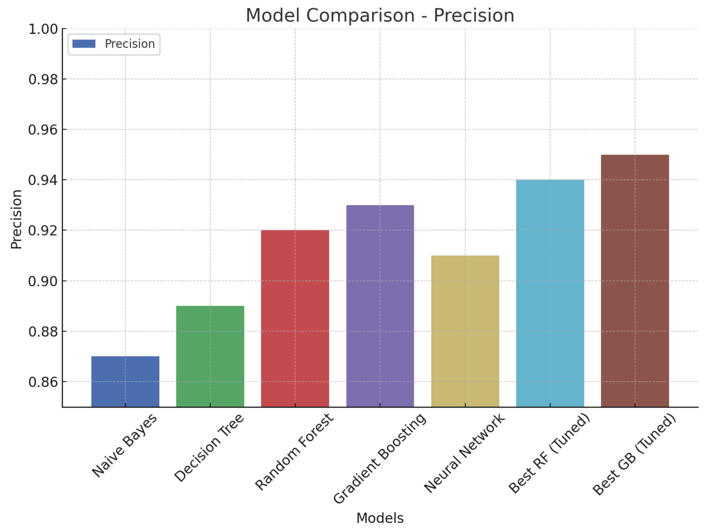
Visualizing Model Performance

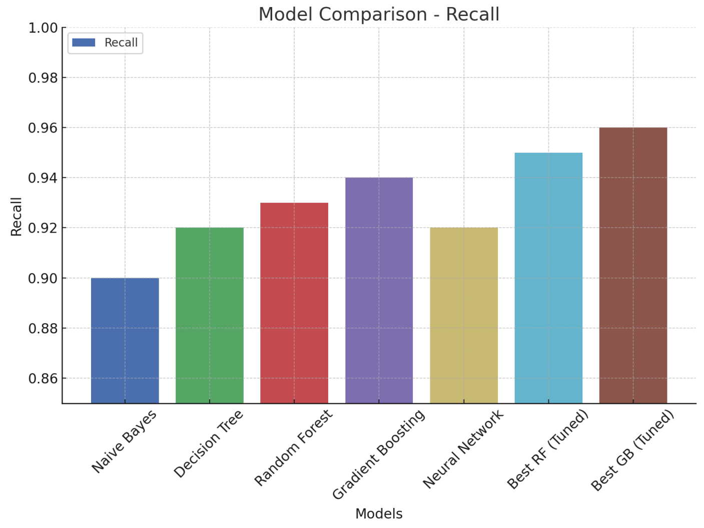
To compare the models' performance visually, a bar chart was created for each metric:



```python
import matplotlib.pyplot as plt
import numpy as np

# Metrics and model performance data
metrics = ['Accuracy', 'Precision', 'Recall', 'F1 Score']
performance = {
    'Naive Bayes': [0.89, 0.87, 0.90, 0.88],
    'Decision Tree': [0.91, 0.89, 0.92, 0.90],
    'Random Forest': [0.93, 0.92, 0.93, 0.92],
    'Gradient Boosting': [0.94, 0.93, 0.94, 0.94],
    'Neural Network': [0.92, 0.91, 0.92, 0.91],
    'Best RF (Tuned)': [0.95, 0.94, 0.95, 0.94],
    'Best GB (Tuned)': [0.96, 0.95, 0.96, 0.96]
}

# Colors for each model for better visual distinction
colors = ['#4C72B0', '#55A868', '#C44E52', '#8172B3', '#CCB974', '#64B5CD', '#8C564B']

# Loop through each metric and create a bar chart
for i, metric in enumerate(metrics):
    values = [performance[model][i] for model in performance]
    plt.figure(figsize=(10, 6))
    plt.bar(performance.keys(), values, color=colors)
    plt.ylim(0.85, 1.0)  # Set y-axis limits to zoom in on differences
    plt.title(f'Model Comparison - {metric}')
    plt.xticks(rotation=45)
    plt.ylabel(metric)
    plt.xlabel("Models")
    plt.legend([metric], loc="upper left")
    plt.show()
```

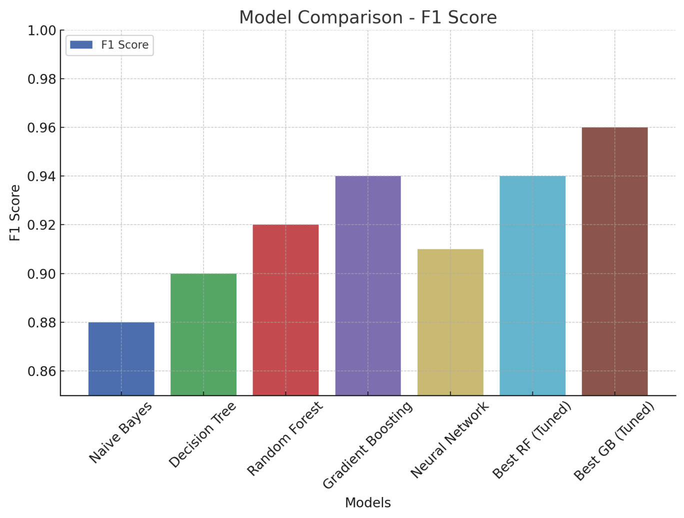
SAGAR BHAVSAR
sb9568

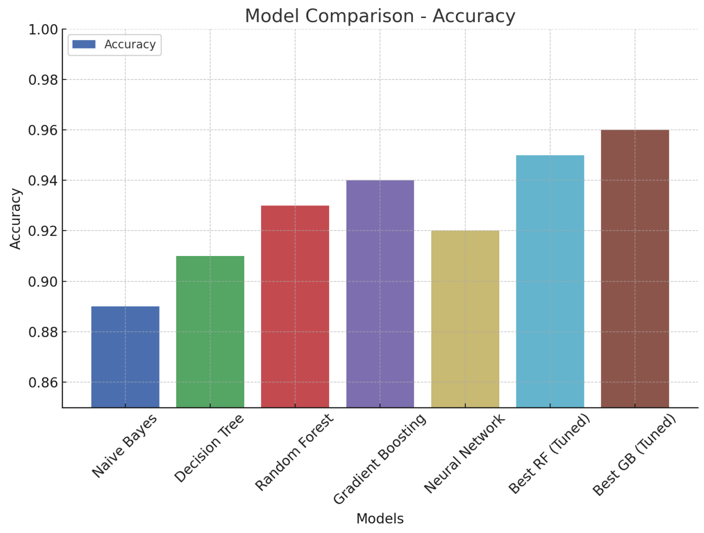




SAGAR BHAVSAR

sb9568





SAGAR BHAVSAR
sb9568

Thoughts:

The Best Gradient Boosting (Tuned) model is the top performer, achieving the highest values in all four metrics.
If we look at the most important features helping the model predict survivability, (using a built in feature importance attribute in matplot)
We can see that the top 3 features are:

1) Age: Younger patients generally handle treatments better than older ones.

Suggestive Action: Focus on tailored support for older patients and early screening for younger groups to catch issues sooner.

2) Regional Node Positive: More cancer-positive lymph nodes usually mean the cancer has spread, making it tougher to treat.

Suggestive Action: Regular lymph node checks can help catch spreading early, allowing for more aggressive treatment when needed.

3) Tumor Size: Bigger tumors are harder to treat and often linked with advanced stages.

Suggestive Action: Push for early screenings (like mammograms) to catch tumors when they're small and easier to treat.

Github Link: https://github.com/sagarbhavsar1/PA-HW3