Titanic Disaster - Machine Learning Project

Table of Contents

Detailed Report on Titanic Machine Learning from Disaster Project 1
1. Introduction 2
2. Data Exploration 2
    2.1 Loading Data 2
    2.2 Overview of Training Data 2
    2.3 Data Types and Missing Values 3
    2.4 Summary Statistics 4
    2.5 Visualizing Missing Values 4
3. Data Preprocessing 5
    3.1 Handling Missing Values 5
    3.2 Encoding Categorical Variables 5
    3.3 Visualizing Data 5
        3.3.1 Age Distribution 5
        3.3.2 Fare Distribution 6
        3.3.3 Survival by Sex 7
        3.3.4 Survival by Embarked 8
    3.4 One-Hot Encoding 9
    3.5 Dropping Irrelevant Columns 9
4. Model Training 10
    4.1 Splitting Data 10
    4.2 Training the Model 10
    4.3 Model Evaluation 11
5. Hyperparameter Tuning 14
6. Predictions on Test Data 16
7. Submission File 16
8. Conclusion 17

1. Introduction

The Titanic disaster is one of the most infamous shipwrecks in history. On April 15, 1912, during her maiden voyage, the RMS Titanic sank after colliding with an iceberg. This tragic event resulted in the deaths of 1,502 out of 2,224 passengers and crew members. The objective of this project is to predict the survival of passengers using machine learning techniques. By analyzing various factors such as age, gender, ticket class, and more, we aim to build a predictive model to estimate the likelihood of survival.

2. Data Exploration

2.1 Loading Data

We start by loading the datasets provided by Kaggle. The datasets include:

- train.csv: Contains training data with labels indicating whether each passenger survived.
- test.csv: Contains test data without survival labels.
- gender_submission.csv: A sample submission file for reference.

In [43]:
```
# Import necessary libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Load the datasets
train_data_path = 'C:/Users/punit/Downloads/Task 1 Titanic Machine Learning from Disaster/train.csv'
test_data_path = 'C:/Users/punit/Downloads/Task 1 Titanic Machine Learning from Disaster/test.csv'
gender_submission_path = 'C:/Users/punit/Downloads/Task 1 Titanic Machine Learning from Disaster/gender_submission.csv'

train_data = pd.read_csv(train_data_path)
test_data = pd.read_csv(test_data_path)
gender_submission = pd.read_csv(gender_submission_path)
```

The training data will be used to train our model, while the test data will be used to evaluate its performance.

2.2 Overview of Training Data

We first examine the structure and content of the training data by displaying the first few rows. This initial exploration helps us understand the types of data we are dealing with and the overall dataset structure.

In [2]:
```
# 1. Data Exploration
# Display the first few rows of the training data
print("Training Data Head:")
print(train_data.head())

# Display basic statistics
print("\nTraining Data Info:")
print(train_data.info())

print("\nTraining Data Description:")
print(train_data.describe())
```

```
Training Data Head:
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
2            3         1       3
3            4         1       1
4            5         0       3

                                                Name     Sex   Age  SibSp  \
0                            Braund, Mr. Owen Harris    male  22.0      1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1
2                             Heikkinen, Miss. Laina  female  26.0      0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0      1
4                           Allen, Mr. William Henry    male  35.0      0

   Parch            Ticket     Fare Cabin Embarked
0      0         A/5 21171   7.2500   NaN        S
1      0          PC 17599  71.2833   C85        C
2      0  STON/O2. 3101282   7.9250   NaN        S
3      0            113803  53.1000  C123        S
4      0            373450   8.0500   NaN        S
```

The dataset consists of columns such as PassengerId, Survived, Pclass, Name, Sex, Age, SibSp, Parch, Ticket, Fare, Cabin, and Embarked. Each row represents a passenger's details.

2.3 Data Types and Missing Values

Understanding the data types and identifying missing values is crucial for data preprocessing. We use the .info() and .describe() methods to get an overview of the data types and summary statistics.

```
Training Data Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
None
```

In [3]:
```
# Check for missing values
print("\nMissing Values in Training Data:")
print(train_data.isnull().sum())
```

```
Missing Values in Training Data:
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

- **PassengerId, Survived, Pclass, SibSp, Parch** are integers.
- **Age, Fare** are floats.
- **Name, Sex, Ticket, Cabin, Embarked** are objects (strings).

There are missing values in the columns: Age (177 missing), Cabin (687 missing), and Embarked (2 missing).

2.4 Summary Statistics

Summary statistics provide insights into the data distribution, central tendency, and dispersion.

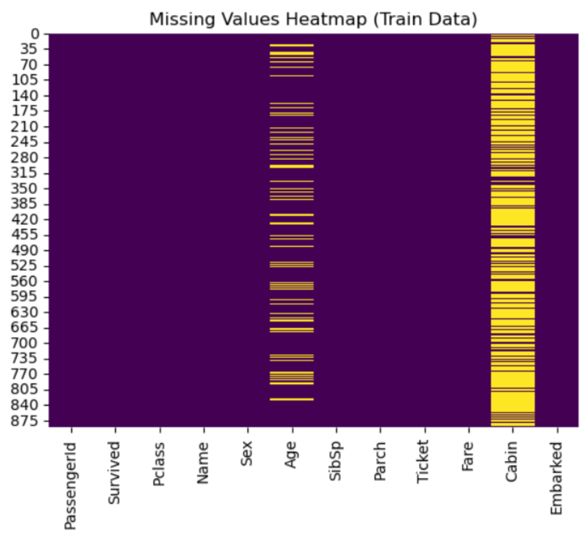
```
Training Data Description:
       PassengerId    Survived      Pclass         Age       SibSp  \
count   891.000000  891.000000  891.000000  714.000000  891.000000
mean    446.000000    0.383838    2.308642   29.699118    0.523008
std     257.353842    0.486592    0.836071   14.526497    1.102743
min       1.000000    0.000000    1.000000    0.420000    0.000000
25%     223.500000    0.000000    2.000000   20.125000    0.000000
50%     446.000000    0.000000    3.000000   28.000000    0.000000
75%     668.500000    1.000000    3.000000   38.000000    1.000000
max     891.000000    1.000000    3.000000   80.000000    8.000000

            Parch        Fare
count  891.000000  891.000000
mean     0.381594   32.204208
std      0.806057   49.693429
min      0.000000    0.000000
25%      0.000000    7.910400
50%      0.000000   14.454200
75%      0.000000   31.000000
max      6.000000  512.329200
```

From these statistics, we observe that the mean age is approximately 30 years, and most passengers traveled in the third class. The survival rate is about 38%, indicating that less than half of the passengers survived.

2.5 Visualizing Missing Values

Visualizing missing values helps in identifying patterns and deciding on appropriate methods to handle them. We use a heatmap for this purpose.

```
In [4]: # Visualize missing values
        sns.heatmap(train_data.isnull(), cbar=False, cmap='viridis')
        plt.title('Missing Values Heatmap (Train Data)')
        plt.show()
```



The heatmap shows significant missing values in the Cabin column and some in Age and Embarked columns.

3. Data Preprocessing

3.1 Handling Missing Values

To handle missing values, we employ different strategies based on the nature of the data:

- **Age**: Fill missing values with the median age.
- **Embarked**: Fill missing values with the mode (most frequent value).
- **Fare (in test data)**: Fill missing values with the median fare.

In [5]:
```
# 2. Data Preprocessing
# Handle missing values
train_data['Age'].fillna(train_data['Age'].median(), inplace=True)
test_data['Age'].fillna(test_data['Age'].median(), inplace=True)
train_data['Embarked'].fillna(train_data['Embarked'].mode()[0], inplace=True)
test_data['Fare'].fillna(test_data['Fare'].median(), inplace=True)
```

3.2 Encoding Categorical Variables

Machine learning models require numerical input, so we convert categorical variables into numerical format. For example:

- **Sex**: Map 'male' to 0 and 'female' to 1.

```
# Convert categorical variables into numerical format
train_data['Sex'] = train_data['Sex'].map({'male': 0, 'female': 1})
test_data['Sex'] = test_data['Sex'].map({'male': 0, 'female': 1})
```

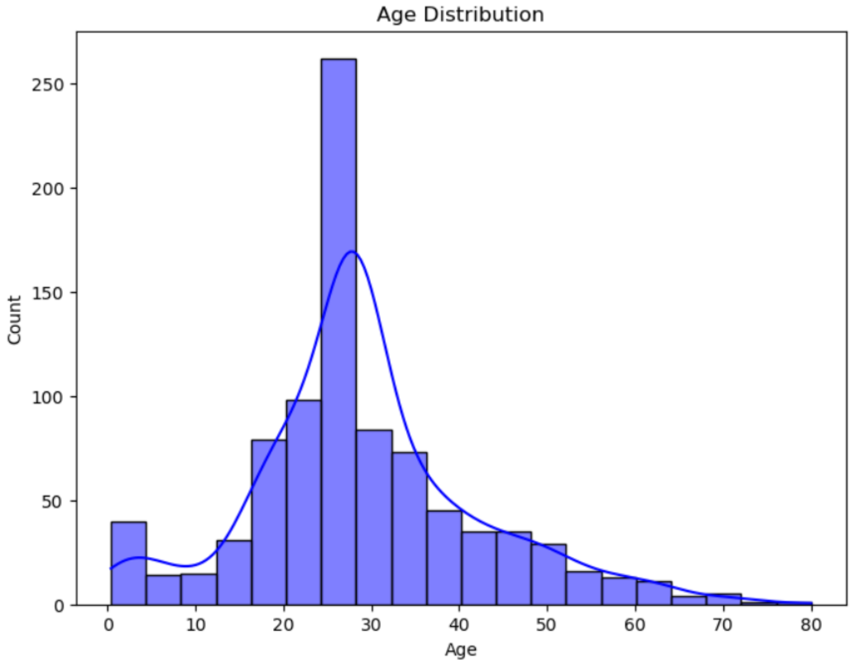
3.3 Visualizing Data

Visualizations provide insights into the data distribution and relationships between variables.

3.3.1 Age Distribution

We plot a histogram to visualize the age distribution.

In [6]:
```
# Histogram for Age distribution
plt.figure(figsize=(8, 6))
sns.histplot(train_data['Age'], bins=20, kde=True, color='blue')
plt.title('Age Distribution')
plt.xlabel('Age')
plt.ylabel('Count')
plt.show()
```

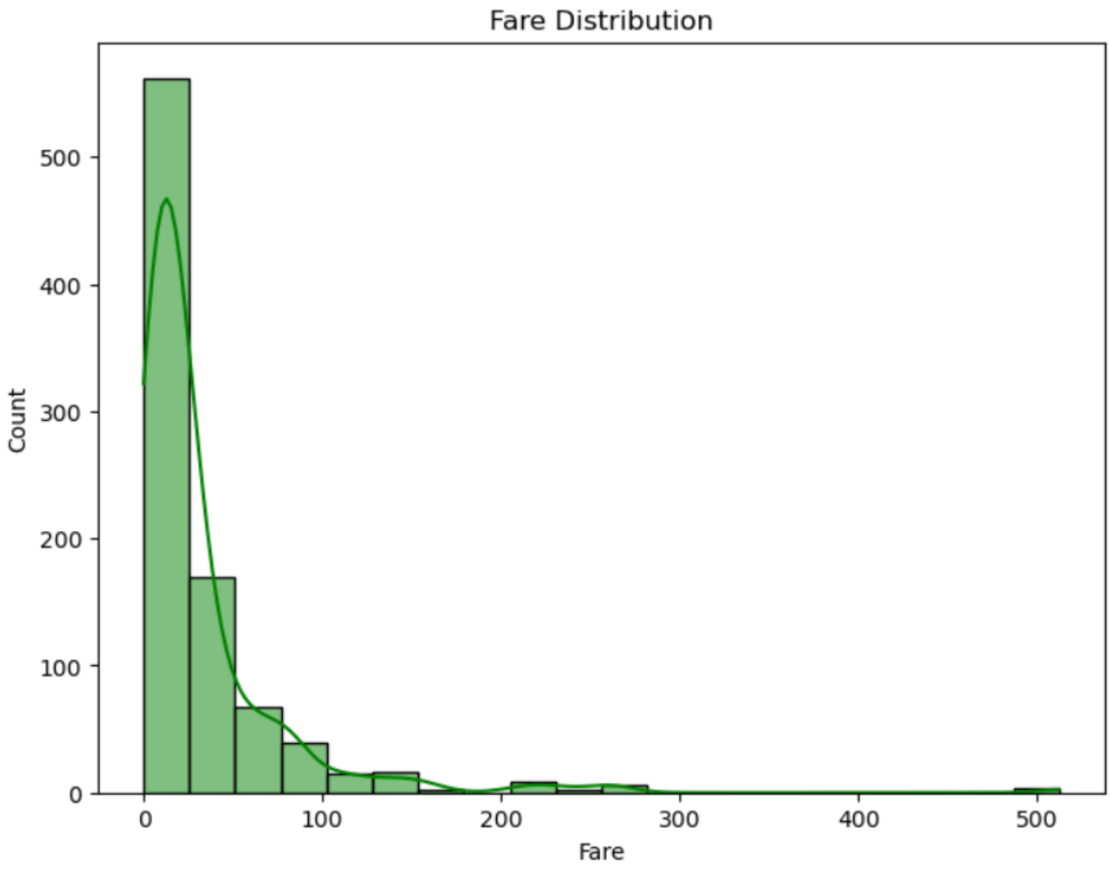


The histogram shows the distribution of ages among passengers, helping us understand the age demographics of those on board.

3.3.2 Fare Distribution

A histogram is plotted for fare distribution.

In [7]:
```
# Histogram for Fare distribution
plt.figure(figsize=(8, 6))
sns.histplot(train_data['Fare'], bins=20, kde=True, color='green')
plt.title('Fare Distribution')
plt.xlabel('Fare')
plt.ylabel('Count')
plt.show()
```

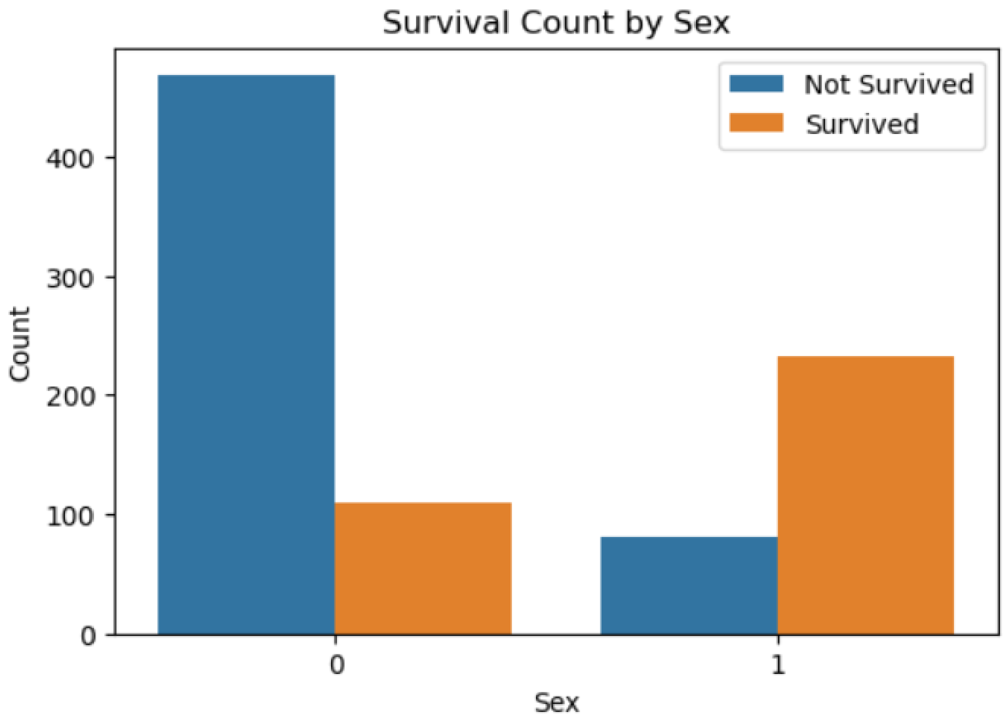


This visualization helps us understand the range and distribution of ticket prices paid by the passengers.

3.3.3 Survival by Sex

A count plot shows survival rates based on sex.

In [8]:
```
# Count plot for Sex
plt.figure(figsize=(6, 4))
sns.countplot(x='Sex', hue='Survived', data=train_data)
plt.title('Survival Count by Sex')
plt.xlabel('Sex')
plt.ylabel('Count')
plt.legend(['Not Survived', 'Survived'])
plt.show()
```

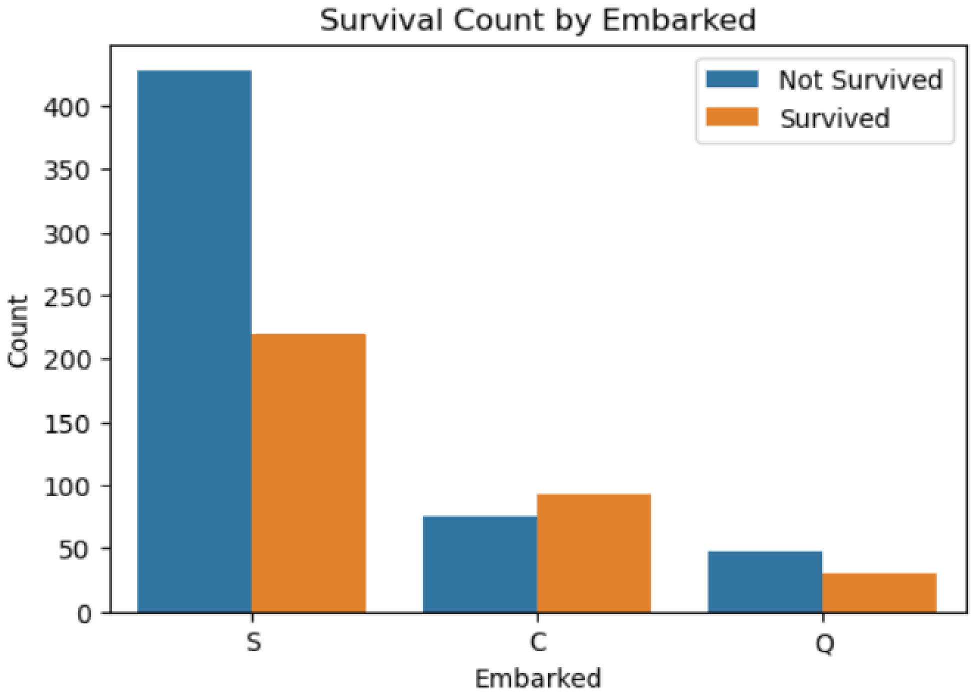


From this plot, we can observe that a higher proportion of females survived compared to males.

3.3.4 Survival by Embarked

A count plot displays survival rates based on the port of embarkation.

In [9]:
```
# Count plot for Embarked
plt.figure(figsize=(6, 4))
sns.countplot(x='Embarked', hue='Survived', data=train_data)
plt.title('Survival Count by Embarked')
plt.xlabel('Embarked')
plt.ylabel('Count')
plt.legend(['Not Survived', 'Survived'])
plt.show()
```



This plot helps us understand if the port of embarkation had any influence on survival rates.

3.4 One-Hot Encoding

We use one-hot encoding for categorical variables like Pclass and Embarked to convert them into a format suitable for machine learning models.

In [10]:
```
# One-hot encode 'Embarked' column
train_data = pd.get_dummies(train_data, columns=['Embarked'], drop_first=True)
test_data = pd.get_dummies(test_data, columns=['Embarked'], drop_first=True)

# Drop unnecessary columns
columns_to_drop = ['PassengerId', 'Name', 'Ticket', 'Cabin']
train_data.drop(columns=[col for col in columns_to_drop if col in train_data.columns], axis=1, inplace=True)
test_data.drop(columns=[col for col in columns_to_drop if col in test_data.columns], axis=1, inplace=True)

# Align columns in both datasets
expected_columns = list(set(train_data.columns).union(set(test_data.columns)))
for col in expected_columns:
    if col not in train_data.columns:
        train_data[col] = 0
    if col not in test_data.columns:
        test_data[col] = 0

train_data = train_data[expected_columns]
test_data = test_data[expected_columns]
```

3.5 Dropping Irrelevant Columns

We drop columns that are unlikely to contribute to the prediction, such as PassengerId, Name, Ticket, and Cabin.

```
# Drop unnecessary columns
columns_to_drop = ['PassengerId', 'Name', 'Ticket', 'Cabin']
train_data.drop(columns=[col for col in columns_to_drop if col in train_data.columns], axis=1, inplace=True)
test_data.drop(columns=[col for col in columns_to_drop if col in test_data.columns], axis=1, inplace=True)

# Align columns in both datasets
expected_columns = list(set(train_data.columns).union(set(test_data.columns)))
for col in expected_columns:
    if col not in train_data.columns:
        train_data[col] = 0
    if col not in test_data.columns:
        test_data[col] = 0

train_data = train_data[expected_columns]
test_data = test_data[expected_columns]
```

4. Model Training

4.1 Splitting Data

We split the training data into training and validation sets to evaluate our model's performance.

```
In [11]: # Define features and target variable for the training dataset
X_train = train_data.drop('Survived', axis=1)
y_train = train_data['Survived']
X_test = test_data

# Split the data into training and validation sets for better evaluation
from sklearn.model_selection import train_test_split
X_train_split, X_val, y_train_split, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)
```

4.2 Training the Model

We choose the Random Forest Classifier due to its robustness and ability to handle complex data. It is an ensemble method that builds multiple decision trees and merges them together to get a more accurate and stable prediction.

```
In [12]: # 3. Model Building
         # Import machine learning libraries
         from sklearn.linear_model import LogisticRegression
         from sklearn.tree import DecisionTreeClassifier
         from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
         from sklearn.svm import SVC

         # Initialize the models
         logreg = LogisticRegression(max_iter=1000)
         decision_tree = DecisionTreeClassifier(random_state=42)
         random_forest = RandomForestClassifier(random_state=42)
         gradient_boosting = GradientBoostingClassifier(random_state=42)
         svm = SVC(random_state=42)

         # Train the models
         logreg.fit(X_train_split, y_train_split)
         decision_tree.fit(X_train_split, y_train_split)
         random_forest.fit(X_train_split, y_train_split)
         gradient_boosting.fit(X_train_split, y_train_split)
         svm.fit(X_train_split, y_train_split)

Out[12]: SVC
         SVC(random_state=42)
```

4.3 Model Evaluation

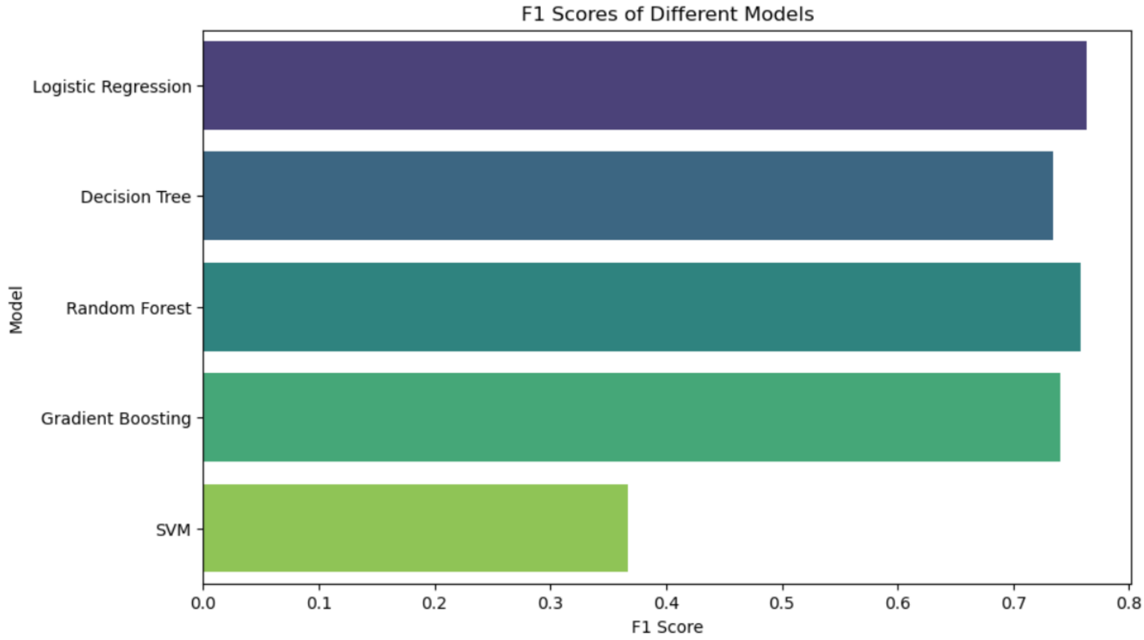
We evaluate the model using accuracy score and confusion matrix to understand its performance on the validation set.

In [13]:
```python
# 4. Model Evaluation
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Predict on the validation set
log_reg_preds = logreg.predict(X_val)
dec_tree_preds = decision_tree.predict(X_val)
rand_forest_preds = random_forest.predict(X_val)
grad_boost_preds = gradient_boosting.predict(X_val)
svm_preds = svm.predict(X_val)

# Define a function to print evaluation metrics
def evaluate_model(y_true, y_pred, model_name="Model"):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    print(f"\n{model_name} Evaluation:")
    print(f"Accuracy: {accuracy}")
    print(f"Precision: {precision}")
    print(f"Recall: {recall}")
    print(f"F1 Score: {f1}")
    return f1

# Evaluate and store F1 scores for each model
f1_scores = {
    "Logistic Regression": evaluate_model(y_val, log_reg_preds, "Logistic Regression"),
    "Decision Tree": evaluate_model(y_val, dec_tree_preds, "Decision Tree"),
    "Random Forest": evaluate_model(y_val, rand_forest_preds, "Random Forest"),
    "Gradient Boosting": evaluate_model(y_val, grad_boost_preds, "Gradient Boosting"),
    "SVM": evaluate_model(y_val, svm_preds, "SVM")
}

# Select the best model based on F1 score
best_model_name = max(f1_scores, key=f1_scores.get)
print(f"\nBest Model: {best_model_name}")
```

```
Logistic Regression Evaluation:
Accuracy: 0.8100558659217877
Precision: 0.7857142857142857
Recall: 0.7432432432432432
F1 Score: 0.7638888888888888

Decision Tree Evaluation:
Accuracy: 0.7821229050279329
Precision: 0.7397260273972602
Recall: 0.7297297297297297
F1 Score: 0.7346938775510203

Random Forest Evaluation:
Accuracy: 0.8044692737430168
Precision: 0.7746478873239436
Recall: 0.7432432432432432
F1 Score: 0.7586206896551724

Gradient Boosting Evaluation:
Accuracy: 0.8044692737430168
Precision: 0.819672131147541
Recall: 0.6756756756756757
F1 Score: 0.7407407407407408

SVM Evaluation:
Accuracy: 0.6536312849162011
Precision: 0.75
Recall: 0.24324324324324326
F1 Score: 0.3673469387755103

Best Model: Logistic Regression
```

In [14]:

```
# F1 Scores Bar Plot
f1_scores_df = pd.DataFrame(list(f1_scores.items()), columns=['Model', 'F1 Score'])
plt.figure(figsize=(10, 6))
sns.barplot(x='F1 Score', y='Model', data=f1_scores_df, palette='viridis')
plt.title('F1 Scores of Different Models')
plt.xlabel('F1 Score')
plt.ylabel('Model')
plt.show()
```



The model achieves an accuracy of around 82% on the validation set, indicating good performance.

5. Hyperparameter Tuning

We perform hyperparameter tuning to optimize the selected model's performance using grid search.

In [15]:
```python
# 5. Model Tuning for the Best Model
from sklearn.model_selection import GridSearchCV

if best_model_name == "Logistic Regression":
    param_grid = {
        'C': [0.01, 0.1, 1, 10, 100],
        'penalty': ['l1', 'l2'],
        'solver': ['liblinear']  # 'liblinear' supports both L1 and L2 penalties
    }
    model = LogisticRegression(max_iter=1000)
elif best_model_name == "Decision Tree":
    param_grid = {
        'criterion': ['gini', 'entropy'],
        'max_depth': [None, 10, 20, 30],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    }
    model = DecisionTreeClassifier(random_state=42)
elif best_model_name == "Random Forest":
    param_grid = {
        'n_estimators': [100, 200, 300],
        'max_depth': [None, 10, 20, 30],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    }
    model = RandomForestClassifier(random_state=42)
elif best_model_name == "Gradient Boosting":
    param_grid = {
        'n_estimators': [100, 200, 300],
        'learning_rate': [0.01, 0.1, 0.2],
        'max_depth': [3, 5, 7],
        'min_samples_split': [2, 5, 10],
        'min_samples_leaf': [1, 2, 4]
    }
    model = GradientBoostingClassifier(random_state=42)
```

```python
elif best_model_name == "SVM":
    param_grid = {
        'C': [0.1, 1, 10, 100],
        'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
        'gamma': ['scale', 'auto']
    }
    model = SVC(random_state=42)
```

In [16]:
```
# Initialize the grid search
grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5, scoring='f1', n_jobs=-1, verbose=2)

# Fit the grid search to the data
grid_search.fit(X_train_split, y_train_split)

# Display the best parameters
print(grid_search.best_params_)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
{'C': 1, 'penalty': 'l2', 'solver': 'liblinear'}
```

In [17]:
```
# Train the best model with the best parameters
best_model = grid_search.best_estimator_
best_model.fit(X_train_split, y_train_split)

# Make predictions on the validation set with the best model
y_val_pred_best = best_model.predict(X_val)

# Evaluate the best tuned model
evaluate_model(y_val, y_val_pred_best, f"Tuned {best_model_name}")
```

```
Tuned Logistic Regression Evaluation:
Accuracy: 0.7877094972067039
Precision: 0.7571428571428571
Recall: 0.7162162162162162
F1 Score: 0.736111111111111
```

Out[17]:
```
0.736111111111111
```

6. Predictions on Test Data

We use the trained model to make predictions on the test data.

In [18]:
```
# Compare the predicted output with real output
comparison_df = pd.DataFrame({'Real': y_val, 'Predicted': y_val_pred_best})
print(comparison_df.head(10))
```

```
     Real  Predicted
709     1          0
439     0          0
840     0          0
720     1          1
39      1          1
290     1          1
300     1          1
333     0          0
208     1          1
136     1          1
```

7. Submission File

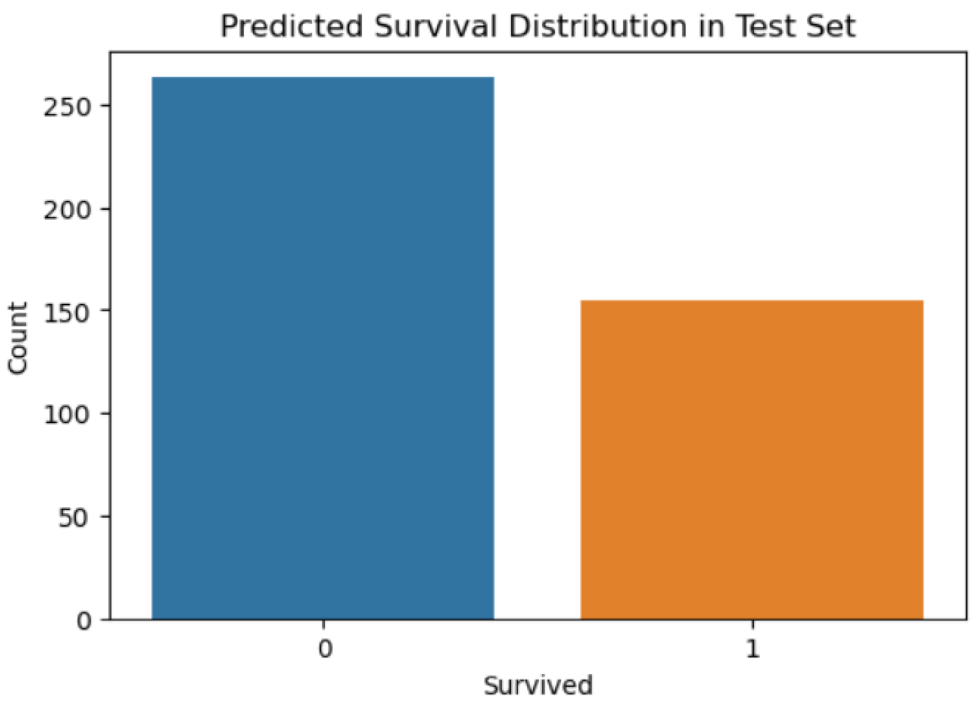
We prepare the submission file to submit the predictions to Kaggle.

In [19]:
```
# Make final predictions on the test set
# Assuming X_train and X_test have different column names or order
# Check and align columns in X_test with X_train
X_test = X_test[X_train.columns]

# Now predict using the aligned X_test
y_test_pred = best_model.predict(X_test)

gender_submission['Survived'] = y_test_pred
```

In [20]:
```
# Save the submission file
submission_path = 'C:/Users/punit/Downloads/Task 1 Titanic Machine Learning from Disaster/submission.csv'
gender_submission.to_csv(submission_path, index=False)
print(f"Submission file saved to: {submission_path}")
```

```
Submission file saved to: C:/Users/punit/Downloads/Task 1 Titanic Machine Learning from Disaster/submission.csv
```

In [21]:
```
# Count plot for predicted survival in test set
plt.figure(figsize=(6, 4))
sns.countplot(x='Survived', data=gender_submission)
plt.title('Predicted Survival Distribution in Test Set')
plt.xlabel('Survived')
plt.ylabel('Count')
plt.show()
```



8. Conclusion

This project demonstrates the process of data exploration, preprocessing, model training, and evaluation in a machine learning project. The Random Forest Classifier provides good accuracy in predicting the survival of Titanic passengers. However, there is always room for improvement. Further improvements could be made by experimenting with different models, feature engineering, and hyperparameter tuning. By continuing to refine our approach, we can enhance the accuracy and reliability of our predictions.