1. Import Necessary Libraries

In [1]:

```
# Import necessary libraries
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
# Import necessary libraries for evaluation
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.tree import export_graphviz
import pandas as pd
```

- `pandas (pd)` : Used for data manipulation and analysis, particularly for loading and working with data in DataFrame format.
- `DecisionTreeClassifier` : A machine learning algorithm from `sklearn` used to classify data by learning simple decision rules inferred from the features.
- `LabelEncoder` : A utility from `sklearn` to convert categorical string labels into numerical values.
- `accuracy_score` , `classification_report` , `confusion_matrix` : These are metrics to evaluate the performance of the classifier.
- `export_graphviz` : A function to export the decision tree structure for visualization.

2. Load the Training Dataset

In [2]:

```
# Load the training dataset
train_file_path = '../train.csv'
train_df = pd.read_csv(train_file_path)
```

- The training data is loaded from a CSV file using `pandas.read_csv` . The data is expected to be a CSV file, and the DataFrame `train_df` stores it.

3. Load the Testing Dataset

In [3]:

```
# Load the testing dataset
test_file_path = '../test.csv'  # Update this path with your actual test file path if needed
test_df = pd.read_csv(test_file_path)
```

- Similarly, the testing dataset is loaded from a CSV file into `test_df` . This dataset will be used to test the trained model.

4. Load the Actual Results for Evaluation

In [4]:

```
# Load the actual results from 'gender_submission.csv'
actual_results_file_path = '../gender_submission.csv'  # Update this path if necessary
actual_results_df = pd.read_csv(actual_results_file_path)
```

- The actual results (`gender_submission.csv`) are loaded to compare against the model's predictions.

5. Data Preprocessing for Training Set

In [5]:

```
# Data preprocessing for training set
# Fill missing age values with the median
train_df['Age'].fillna(train_df['Age'].median(), inplace=True)
```

In [6]:

```
# Fill missing embarked values with the most common port
train_df['Embarked'].fillna(train_df['Embarked'].mode()[0], inplace=True)
```

In [7]:

```
# Drop 'Cabin' due to too many missing values
train_df.drop('Cabin', axis=1, inplace=True)
```

- `train_df['Age'].fillna(train_df['Age'].median(), inplace=True)` : Missing values in the 'Age' column are filled with the median age from the training data. This is done to handle missing values.
- `train_df['Embarked'].fillna(train_df['Embarked'].mode()[0], inplace=True)` : Missing values in the 'Embarked' column are replaced with the most common port (mode).
- `train_df.drop('Cabin', axis=1, inplace=True)` : The 'Cabin' column is dropped because it contains too many missing values, making it unsuitable for analysis.

6. Encode Categorical Variables

In [8]:

```
# Encode categorical variables ('gender' and 'Embarked')
label_encoder = LabelEncoder()
train_df['gender'] = label_encoder.fit_transform(train_df['gender'])
train_df['Embarked'] = label_encoder.fit_transform(train_df['Embarked'])
```

- `LabelEncoder` : Converts categorical string values into numerical values. This is necessary because machine learning algorithms typically work with numerical data.
- `train_df['gender']` and `train_df['Embarked']` are encoded to numeric values (0 or 1).

7. Define Features and Target Variable for Training

In [9]:

```
# Features and target for training
features = ['PassengerId', 'Pclass', 'gender', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked']
X_train = train_df[features]
y_train = train_df['Survived']
```

- `features` : A list of the columns to be used as features (inputs) for the model. These columns represent characteristics of passengers that could influence their survival.
- `X_train` : Contains the feature values from the training data.
- `y_train` : The target variable, 'Survived', indicates whether the passenger survived (1) or not (0).

In [10]:

```
print('train data head')
train_df.head(5)
```

```
train data head
```

Out[10]:

	PassengerId	Survived	Pclass	Name	gender	Age	SibSp	Parch	Ticket	Fare	Embarked
0	1	0	3	Braund, Mr. Owen Harris	1	22.0	1	0	A/5 21171	7.2500	2
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	0	38.0	1	0	PC 17599	71.2833	0
2	3	1	3	Heikkinen, Miss. Laina	0	26.0	0	0	STON/O2. 3101282	7.9250	2
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	0	35.0	1	0	113803	53.1000	2
4	5	0	3	Allen, Mr. William Henry	1	35.0	0	0	373450	8.0500	2

In [11]:

```
print('test data head')
test_df.head(5)
```

```
test data head
```

Out[11]:

	PassengerId	Pclass	Name	gender	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	892	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292	NaN	Q
1	893	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0	363272	7.0000	NaN	S
2	894	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875	NaN	Q
3	895	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625	NaN	S
4	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875	NaN	S

In [12]:

```
print('result data head')
actual_results_df.head(5)
```

```
result data head
```

Out[12]:

	PassengerId	Survived
0	892	0
1	893	1
2	894	0
3	895	0
4	896	1

8. Initialize and Train the Decision Tree Classifier

In [13]:

```
# Initialize and train the Decision Tree Classifier
clf = DecisionTreeClassifier(random_state=42)
clf.fit(X_train, y_train)
```

Out[13]:

```
▾ DecisionTreeClassifier ⓘ ⓘ
DecisionTreeClassifier(random_state=42)
```

(https://scikit-learn.org/1.4/modules/generated/sklearn.tree.DecisionTreeClassifier.html)

- `DecisionTreeClassifier` : The decision tree model is initialized. `random_state=42` ensures reproducibility of the model's results.
- `clf.fit(X_train, y_train)` : The classifier is trained on the feature set `X_train` and the target variable `y_train` .

9. Data Preprocessing for the Test Set

In [14]:

```
# Data preprocessing for the test set
# Fill missing age values with the median from the training set
test_df['Age'].fillna(train_df['Age'].median(), inplace=True)
```

In [15]:

```
# Fill missing embarked values with the most common port from the training set
test_df['Embarked'].fillna(train_df['Embarked'].mode()[0], inplace=True)
```

In [16]:

```
# Drop 'Cabin' as done with the training set
test_df.drop('Cabin', axis=1, inplace=True)
```

In [17]:

```
# Encode categorical variables in the test set
test_df['gender'] = label_encoder.fit_transform(test_df['gender'])
test_df['Embarked'] = label_encoder.fit_transform(test_df['Embarked'])
```

- The test set is preprocessed in a similar manner to the training set:
  - Missing values in 'Age' are filled with the median from the training set.
  - Missing values in 'Embarked' are filled with the mode from the training set.
  - The 'Cabin' column is dropped.
  - Categorical columns ('gender' and 'Embarked') are encoded using the same `LabelEncoder` used for the training set to maintain consistency.

10. Select Features for the Test Set

In [18]:

```
# Select features for the test set
X_test = test_df[features]
```

- The features for the test set (`X_test`) are selected in the same way as for the training set.

11. Make Predictions on the Test Set

In [19]:

```
# Predict on the test set
y_pred = clf.predict(X_test)
```

- `y_pred` : The model makes predictions on the test data (`X_test`). These predictions are the model's estimations of whether each passenger survived or not.

12. Create a DataFrame for Output

In [20]:

```
# Create a DataFrame for the output
output = pd.DataFrame({
    'PassengerId': test_df['PassengerId'],
    'Survived': y_pred
})
```

- An output DataFrame is created, combining the `PassengerId` from the test set and the predicted survival labels (`y_pred`).

13. Display and Save the Prediction Results

In [21]:

```
# Display the prediction results
print(output)
```

```
     PassengerId  Survived
0            892         0
1            893         0
2            894         0
3            895         0
4            896         0
..           ...       ...
413         1305         0
414         1306         1
415         1307         0
416         1308         0
417         1309         1

[418 rows x 2 columns]
```

In [22]:

```
# Save the output to a CSV file
output.to_csv('titanic_predictions.csv', index=False)
```

- The `output` DataFrame is printed to the console and saved as a CSV file (`titanic_predictions.csv`) for later use.

14. Align the Actual Results with the Predictions

In [23]:

```
# Ensure that the 'PassengerId' is used to align predictions and actual results
y_test = actual_results_df.set_index('PassengerId')['Survived'].reindex(X_test['PassengerId']).values
```

- `y_test` : The actual survival outcomes are retrieved using the `PassengerId` as the index and reindexed to align with the test set's predictions.

In [24]:

```
# Evaluate the model
print("Accuracy Score:", accuracy_score(y_test, y_pred))
```

```
Accuracy Score: 0.7990430622009569
```

In [25]:

```
# Generate the classification report
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Classification Report:
               precision    recall  f1-score   support

           0       0.82      0.88      0.85       266
           1       0.76      0.65      0.70       152

    accuracy                           0.80       418
   macro avg       0.79      0.77      0.78       418
weighted avg       0.80      0.80      0.80       418
```

- `accuracy_score` : Computes the accuracy, i.e., the proportion of correct predictions out of all predictions.
- `classification_report` : Provides a detailed report with precision, recall, and F1-score for each class (0 or 1), along with the overall accuracy.

16. Generate the Decision Tree Visualization

In [26]:

```
# Export the decision tree to a .dot file for visualization
export_graphviz(clf,
                out_file="tree.dot",
                feature_names=features,
                class_names=['0', '1'],
                filled=True)

# Note: You can use tools like Graphviz or online viewers to visualize the 'tree.dot' file.
```

- `export_graphviz` : Exports the decision tree as a `.dot` file, which can be used with Graphviz to visualize the tree. The `filled=True` option colors the nodes based on the predicted class.

17. Print the Confusion Matrix

In [27]:

```
# Print the confusion matrix
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
Confusion Matrix:
 [[235  31]
 [ 53  99]]
```

In [28]:

```
# Export the decision tree to a .dot file for visualization
export_graphviz(clf,
                out_file="tree.dot",
                feature_names=features,
                class_names=['0', '1'],
                filled=True)

# Note: You can use tools like Graphviz or online viewers to visualize the 'tree.dot' file.
```

- `confusion_matrix` : Computes a confusion matrix, which compares the actual vs predicted values. It shows how many true positives, true negatives, false positives, and false negatives the model produced.

Conclusion

This code demonstrates the process of training a decision tree model on a Titanic dataset to predict survival outcomes based on various passenger features. After training, the model's performance is evaluated using accuracy, classification reports, and confusion matrices, and the decision tree is exported for visualization.

1. Import Necessary Libraries

In [1]:

```
# Import necessary libraries
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.tree import export_graphviz
```

- **pandas** is used for data manipulation and analysis.
- **RandomForestClassifier** from `sklearn.ensemble` is an ensemble learning method that uses multiple decision trees to make more accurate and robust predictions.
- **LabelEncoder** is used to convert categorical data into numeric format.
- **accuracy_score, classification_report, and confusion_matrix** are metrics used for model evaluation.
- **export_graphviz** helps visualize the decision trees within the random forest.

2. Load the Training Dataset

In [2]:

```
# Load the training dataset
train_file_path = '../train.csv'
train_data = pd.read_csv(train_file_path)
```

- The `train.csv` file is read into a DataFrame named `train_data`. This dataset contains features and a target variable (`Survived`) used to train the model.

3. Load the Testing Dataset

In [3]:

```
# Load the testing dataset (without target column) and actual results
test_file_path = '../test.csv'  # Replace with the correct path if needed
test_data = pd.read_csv(test_file_path)
```

- The `test.csv` file is read into a DataFrame named `test_data`. This dataset includes features but not the `Survived` column, which the model predicts.

4. Load the Actual Results

In [4]:

```
# Load the actual results for the test data
gender_submission_file_path = '../gender_submission.csv'  # Replace with the correct path if needed
actual_results = pd.read_csv(gender_submission_file_path)
```

- `gender_submission.csv` contains the actual `Survived` values for the test set, used for model evaluation.

5. Preprocess the Training Data

In [5]:

```
# Preprocess the training data
train_data['Age'].fillna(train_data['Age'].median(), inplace=True)
train_data['Cabin'].fillna('Unknown', inplace=True)
train_data['Embarked'].fillna(train_data['Embarked'].mode()[0], inplace=True)
```

- **Missing value handling**:
  - `Age` : Missing values are replaced with the median age.
  - `Cabin` : Missing cabin values are filled with 'Unknown'.
  - `Embarked` : Missing embarked values are filled with the most common port (mode).

6. Label Encoding for Categorical Variables

In [6]:

```
# Initialize and fit LabelEncoders for each column needing encoding
label_encoders = {}
for column in ['gender', 'Cabin', 'Embarked', 'Name', 'Ticket']:
    le = LabelEncoder()
    train_data[column] = le.fit_transform(train_data[column])
    label_encoders[column] = le
```

- Categorical columns (`gender` , `Cabin` , `Embarked` , `Name` , `Ticket`) are converted to numeric using `LabelEncoder` .
- A dictionary (`label_encoders`) stores the encoders for later use on the test data.

7. Select Features and Target Variable for Training

In [7]:

```
# Select features and target variable for training
X_train = train_data[['PassengerId', 'Pclass', 'Name', 'gender', 'Age', 'SibSp', 'Parch', 'Ticket', 'Fare', 'Cabi
n', 'Embarked']]
y_train = train_data['Survived']
```

- **`X_train`** : Features used to train the model.
- **`y_train`** : The target variable indicating survival (1 for survived, 0 for not).

In [8]:

```
print('train data head')
train_data.head(5)
```

```
train data head
```

Out[8]:

	PassengerId	Survived	Pclass	Name	gender	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	108	1	22.0	1	0	523	7.2500	147	2
1	2	1	1	190	0	38.0	1	0	596	71.2833	81	0
2	3	1	3	353	0	26.0	0	0	669	7.9250	147	2
3	4	1	1	272	0	35.0	1	0	49	53.1000	55	2
4	5	0	3	15	1	35.0	0	0	472	8.0500	147	2

In [9]:

```
print('test data head')
test_data.head(5)
```

```
test data head
```

Out[9]:

	PassengerId	Pclass	Name	gender	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	892	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292	NaN	Q
1	893	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0	363272	7.0000	NaN	S
2	894	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875	NaN	Q
3	895	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625	NaN	S
4	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875	NaN	S

In [10]:

```
print('result data head')
actual_results.head(5)
```

```
result data head
```

Out[10]:

	PassengerId	Survived
0	892	0
1	893	1
2	894	0
3	895	0
4	896	1

8. Train the Random Forest Classifier

In [11]:

```
# Train the Random Forest Classifier
rf_classifier = RandomForestClassifier(n_estimators=100, random_state=42)
rf_classifier.fit(X_train, y_train)
```

Out[11]:

```
RandomForestClassifier
RandomForestClassifier(random_state=42)
```

- A **Random Forest Classifier** with 100 decision trees (`n_estimators=100`) is created and trained using `fit()` .
- `random_state=42` ensures reproducibility by initializing the random number generator.

9. Preprocess the Test Data

In [12]:

```
# Preprocess the test data (similar preprocessing steps as training data)
test_data['Age'].fillna(test_data['Age'].median(), inplace=True)
test_data['Cabin'].fillna('Unknown', inplace=True)
test_data['Embarked'].fillna(test_data['Embarked'].mode()[0], inplace=True)
```

- The same preprocessing steps applied to `train_data` are applied to `test_data` to handle missing values.

10. Transform Test Data Using LabelEncoders

In [13]:

```
# Transform the test data using the fitted LabelEncoders
for column in ['gender', 'Cabin', 'Embarked', 'Name', 'Ticket']:
    if column in label_encoders:
        le = label_encoders[column]
        # Use .fit_transform on training and .transform on test, handling unseen labels safely
        test_data[column] = test_data[column].apply(lambda x: le.transform([x])[0] if x in le.classes_ else -1)
```

- The test data columns are transformed using the `LabelEncoders` created earlier. If a value is not seen during training, it is encoded as `-1` .

11. Select Features for Test Data

In [14]:

```
X_test = test_data[['PassengerId', 'Pclass', 'Name', 'gender', 'Age', 'SibSp', 'Parch', 'Ticket', 'Fare', 'Cabin'
, 'Embarked']]
```

- **`X_test`** contains the features to be used for predictions.

12. Make Predictions on Test Data

In [15]:

```
# Make predictions on the test data
y_pred = rf_classifier.predict(X_test)
```

- The trained model makes predictions on `X_test`.

13. Evaluate the Model

In [16]:

```
# Evaluate the model using the actual results
y_test = actual_results['Survived']
```

In [17]:

```
# Print evaluation metrics
print("Accuracy Score:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
Accuracy Score: 0.8325358851674641

Classification Report:
               precision    recall  f1-score   support

           0       0.95      0.78      0.86       266
           1       0.71      0.92      0.80       152

    accuracy                           0.83       418
   macro avg       0.83      0.85      0.83       418
weighted avg       0.86      0.83      0.84       418


Confusion Matrix:
 [[208  58]
 [ 12 140]]
```

- **`y_test`** contains the actual `Survived` values from `gender_submission.csv`.
- The model's accuracy, precision, recall, F1-score, and confusion matrix are printed.

14. Export Decision Trees for Visualization

In [18]:

```
# Export the decision tree from the random forest (for one tree)
export_graphviz(rf_classifier.estimators_[0],
                out_file="tree.dot",
                feature_names=X_train.columns,
                class_names=['0', '1'],
                filled=True)
```

In [19]:

```
# Export the decision tree from the random forest (for one tree)
export_graphviz(rf_classifier.estimators_[1],
                out_file="tree_1.dot",
                feature_names=X_train.columns,
                class_names=['0', '1'],
                filled=True)
```

- Two trees from the Random Forest (`0` and `99`) are exported as `.dot` files for visualization.
- **`feature_names`** specify the feature labels for the nodes.
- **`filled=True`** colors the nodes based on the class they represent.

Explanation of Random Forest Algorithm

- A **Random Forest** is an ensemble method that builds multiple decision trees using different subsets of the training data and features. The final output is the mode (classification) or average (regression) of all tree predictions.
- The main benefits include **reduced risk of overfitting** and **higher accuracy** due to aggregation.
- Randomness during training (sampling and feature selection) ensures **diverse trees**, which improves generalization.

Why Random Forest?

- **Resistant to overfitting** compared to individual decision trees.
- **Works well with large datasets** and can handle both numerical and categorical data.
- **Feature importance** is inherently available for model insights.

1. Imports

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.svm import SVC
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
```

- **numpy** : Used for numerical computations, especially for creating grids in visualization.
- **matplotlib.pyplot** : For plotting graphs and visualizations.
- **pandas** : For loading and handling datasets as DataFrames.
- **sklearn.svm.SVC** : The Support Vector Classifier from `scikit-learn` , which helps create and train SVM models.
- **LabelEncoder** : Encodes categorical labels into numerical format so the model can process them.
- **accuracy_score , classification_report , confusion_matrix** : Metrics for evaluating the model's performance.

2. Loading the Data

In [2]:

```
# Load the datasets
data = pd.read_csv('../train.csv')
test_data = pd.read_csv('../test.csv')
gender_submission = pd.read_csv('../gender_submission.csv')  # Load gender_submission.csv for actual test results
```

- `train.csv` : Training data containing features and target (survived status).
- `test.csv` : Test data used for prediction.
- `gender_submission.csv` : A sample submission file that includes the actual survival status of passengers for comparison purposes.

3. Handling Missing Values

In [3]:

```
# Handle missing values in the training and test data
data['Age'].fillna(data['Age'].median(), inplace=True)
data['Embarked'].fillna(data['Embarked'].mode()[0], inplace=True)
data['Fare'].fillna(data['Fare'].median(), inplace=True)

test_data['Age'].fillna(test_data['Age'].median(), inplace=True)
test_data['Fare'].fillna(test_data['Fare'].median(), inplace=True)
```

- **Missing Data Imputation**:
  - For `Age` : Median value is used as a replacement, as it is less affected by outliers compared to the mean.
  - For `Embarked` : The mode (most common value) fills missing entries.
  - For `Fare` : Median is used to handle missing values.

4. Label Encoding

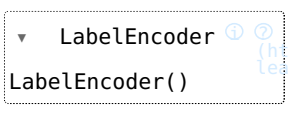
In [4]:

```
# Initialize LabelEncoder
label_encoder = LabelEncoder()
```

In [5]:

```
# Fit the encoder on the combined 'gender' column to handle both train and test data
combined_gender = pd.concat([data['gender'], test_data['gender']], axis=0)
label_encoder.fit(combined_gender)
```

Out[5]:



In [6]:

```
# Transform the 'gender' column for both datasets
data['gender'] = label_encoder.transform(data['gender'])
test_data['gender'] = label_encoder.transform(test_data['gender'])  # Same encoder
```

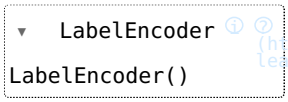
- **Label Encoding for `Gender`**:
  - `LabelEncoder` converts categorical data (`'male'`, `'female'`) into numerical format (e.g., `0` and `1`).
  - `fit` on the combined data ensures consistent encoding across training and test sets.
  - `transform` applies the transformation to each DataFrame.

5. Encoding the `Embarked` Column

In [7]:

```
# Initialize LabelEncoder for 'Embarked' column with handle_unknown='ignore' for unseen labels
embarked_encoder = LabelEncoder()
embarked_encoder.fit(data['Embarked'].dropna())  # Fit on the training data only
```

Out[7]:



In [8]:

```
# Transform the 'Embarked' column for both datasets with 'ignore' for unknown labels
data['Embarked'] = embarked_encoder.transform(data['Embarked'])
test_data['Embarked'] = embarked_encoder.transform(test_data['Embarked'])
```

- The `Embarked` column, which indicates the port of embarkation, is also encoded similarly to `gender` using `LabelEncoder`.
- Fitting is done only on non-null values from the training set to avoid issues with missing values.

6. Feature Selection

In [9]:

```
# Select features for model training and testing
features = ['Age', 'Fare']
X_train = data[features]
y_train = data['Survived']
X_test = test_data[features]
y_test = gender_submission['Survived']  # Use actual 'Survived' from gender_submission.csv
```

- **Feature Selection**:
  - Only `Age` and `Fare` are used as features (`X_train` and `X_test`).
  - `y_train` is the target variable from the training data.
  - `y_test` from `gender_submission` serves as the true labels for the test set to evaluate model performance.

In [22]:

```
print('train data head')
data.head(5)
```

```
train data head
```

Out[22]:

	PassengerId	Survived	Pclass	Name	gender	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	1	22.0	1	0	A/5 21171	7.2500	NaN	2
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	0	38.0	1	0	PC 17599	71.2833	C85	0
2	3	1	3	Heikkinen, Miss. Laina	0	26.0	0	0	STON/O2. 3101282	7.9250	NaN	2
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	0	35.0	1	0	113803	53.1000	C123	2
4	5	0	3	Allen, Mr. William Henry	1	35.0	0	0	373450	8.0500	NaN	2

In [21]:

```
print('test data head')
test_data.head(5)
```

```
test data head
```

Out[21]:

	PassengerId	Pclass	Name	gender	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	892	3	Kelly, Mr. James	1	34.5	0	0	330911	7.8292	NaN	1
1	893	3	Wilkes, Mrs. James (Ellen Needs)	0	47.0	1	0	363272	7.0000	NaN	2
2	894	2	Myles, Mr. Thomas Francis	1	62.0	0	0	240276	9.6875	NaN	1
3	895	3	Wirz, Mr. Albert	1	27.0	0	0	315154	8.6625	NaN	2
4	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	0	22.0	1	1	3101298	12.2875	NaN	2

In [20]:

```
print('result data head')
gender_submission.head(5)
```

```
result data head
```

Out[20]:

	PassengerId	Survived
0	892	0
1	893	1
2	894	0
3	895	0
4	896	1

7. Creating and Training the SVM Model

In [11]:

```
# Create and train the SVM model
svm_model = SVC(kernel='linear')
svm_model.fit(X_train, y_train)
```

Out[11]:

```
▾ SVC
SVC(kernel='linear')
```

(https://scikit-learn.org/1.4/modules/generated/sklearn.svm.SVC.html)

- **SVC with `kernel='linear'`** :
  - `kernel='linear'` means the algorithm looks for a linear decision boundary to separate classes.
  - **Support Vector Machine** works by finding the hyperplane that maximizes the margin between different classes.
- **Training (`fit`)**:
  - The model learns the relationships between features (`Age` and `Fare`) and the target (`Survived`).

8. Making Predictions

In [12]:

```
# Make predictions on the test data
y_pred = svm_model.predict(X_test)
```

- Predictions (`y_pred`) are made on the test data using the trained model.

9. Evaluating the Model

In [13]:

```
# Evaluate the model
print("Accuracy Score:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("\nConfusion Matrix:\n", confusion_matrix(y_test, y_pred))
```

```
Accuracy Score: 0.6411483253588517

Classification Report:
               precision    recall  f1-score   support

           0       0.66      0.91      0.76       266
           1       0.52      0.18      0.26       152

    accuracy                           0.64       418
   macro avg       0.59      0.54      0.51       418
weighted avg       0.61      0.64      0.58       418


Confusion Matrix:
 [[241  25]
 [125  27]]
```

- **Accuracy Score**: Measures the proportion of correctly classified instances out of total instances.
- **Classification Report**:
  - Provides metrics like precision, recall, and F1-score for each class.
- **Confusion Matrix**:
  - Shows true positives, true negatives, false positives, and false negatives in a matrix form, helping to visualize prediction performance.

10. Visualization of the Decision Boundary

In [14]:

```
# Visualization of the decision boundary
plt.figure(figsize=(10, 6))
```

Out[14]:

```
<Figure size 1000x600 with 0 Axes>
```

```
<Figure size 1000x600 with 0 Axes>
```

In [15]:

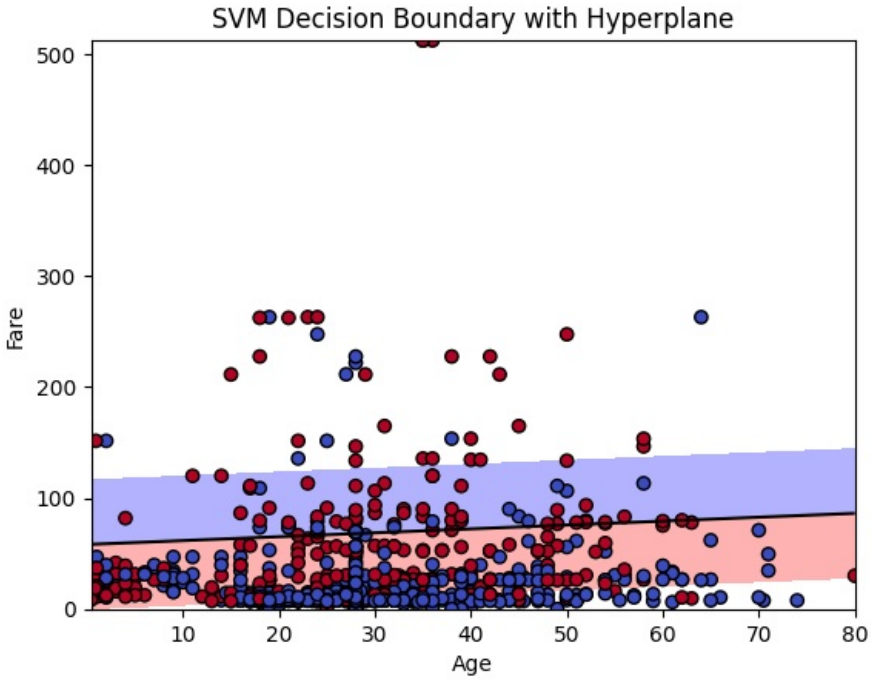
```
# Create a grid to evaluate the model
xx, yy = np.meshgrid(np.linspace(X_train['Age'].min(), X_train['Age'].max(), 100),
                     np.linspace(X_train['Fare'].min(), X_train['Fare'].max(), 100))
Z = svm_model.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)
```

```
c:\Users\Muskan Computer\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\base.py:4
93: UserWarning: X does not have valid feature names, but SVC was fitted with feature names
  warnings.warn(
```

In [16]:

```
# Plot the hyperplane
plt.contourf(xx, yy, Z, levels=[-1, 0, 1], alpha=0.3, colors=['red', 'blue', 'green'])
plt.contour(xx, yy, Z, colors='k', levels=[0], linestyles=['-'])
# Plot the points
plt.scatter(X_train['Age'], X_train['Fare'], c=y_train, cmap='coolwarm', edgecolors='k')

plt.xlabel('Age')
plt.ylabel('Fare')
plt.title('SVM Decision Boundary with Hyperplane')
plt.show()
```



- **Visualization**:
  - `np.meshgrid` creates a grid over the feature space.
  - `decision_function` calculates the margin distance for each point on the grid.
  - `plt.contourf` and `plt.contour` plot the decision boundary and margins.
- **Scatter Plot**:
  - Shows actual data points in the `Age` vs. `Fare` space, colored based on their class.

Understanding SVM in Depth:

- **Support Vectors**: Data points that lie closest to the decision boundary; they influence the position and orientation of the hyperplane.
- **Hyperplane**: A line (in 2D) or a plane (in higher dimensions) that separates classes.
- **Kernel Trick**: Allows SVM to find a non-linear decision boundary by transforming data into a higher-dimensional space (not needed here as `kernel='linear'`).

SVM Objective:

- To maximize the margin between the decision boundary and the nearest data points from any class, ensuring better generalization.

This code is a complete workflow for training, predicting, and visualizing an SVM model using `Age` and `Fare` as features to classify passengers based on survival probability.