titanic_survival

September 18, 2025

[1]:
```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter

# sklearn imports
from sklearn.model_selection import train_test_split, cross_val_score,␣
 ↪GridSearchCV
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪classification_report, roc_auc_score, roc_curve
import joblib

# plotting style
sns.set_theme(style="whitegrid")
%matplotlib inline
```

1 Load the data

[3]:
```
TRAIN_PATH = "../data/train.csv"
TEST_PATH = "../data/test.csv"

# load
train = pd.read_csv(TRAIN_PATH)
test = pd.read_csv(TEST_PATH)

print("Train shape:", train.shape)
print("Test shape:", test.shape)
```

```
Train shape: (891, 12)
```

```
Test shape: (418, 11)
```

2 Quick look at the data (head, info, missing values)

[4]:
```python
display(train.head())
print("\n--- Info ---")
print(train.info())
print("\n--- Missing values (train) ---")
print(train.isnull().sum().sort_values(ascending=False).head(10))

print("\n--- Missing values (test) ---")
print(test.isnull().sum().sort_values(ascending=False).head(10))
```

```
   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


--- 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
```

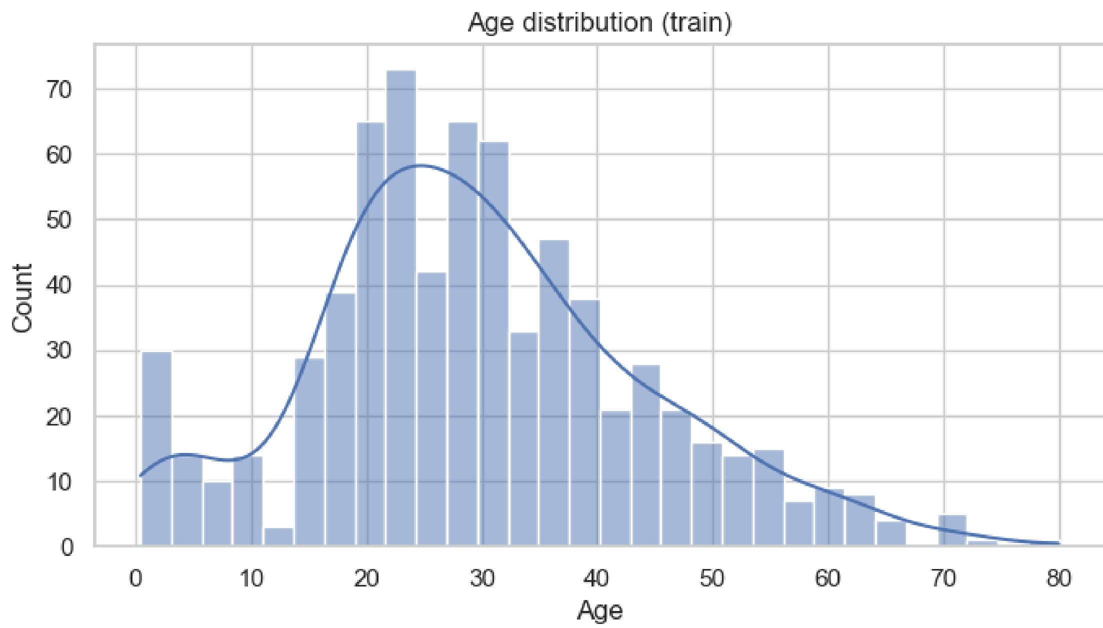
 7   Parch   891 non-null   int64
 8   Ticket   891 non-null   object
 9   Fare   891 non-null   float64
 10 Cabin   204 non-null   object
 11 Embarked   889 non-null   object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
None

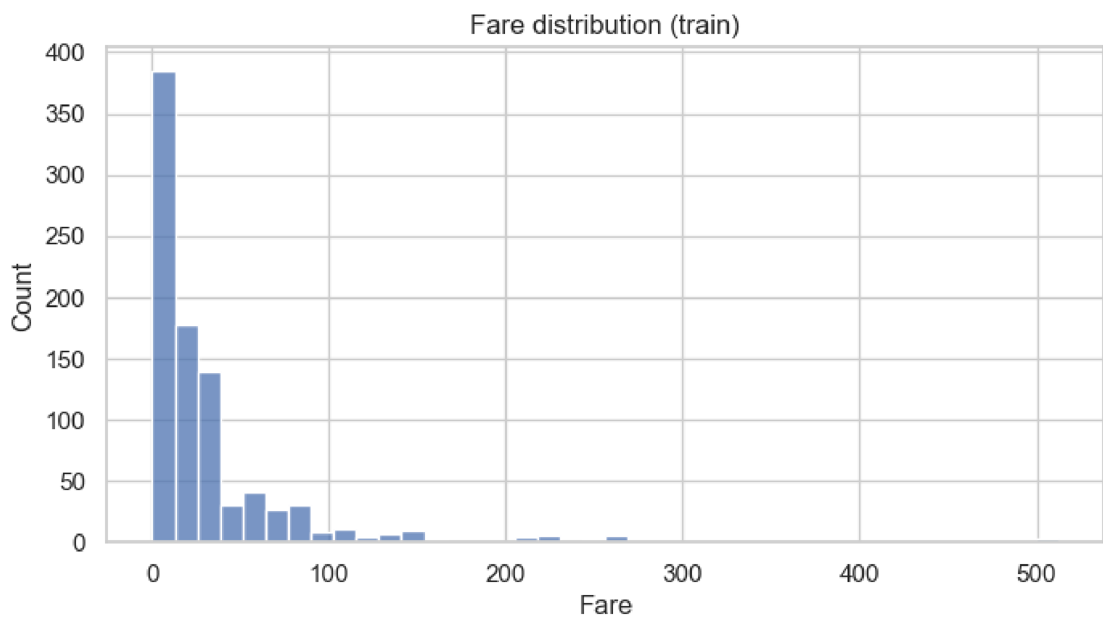
--- Missing values (train) ---
Cabin   687
Age   177
Embarked   2
PassengerId   0
Name   0
Pclass   0
Survived   0
Sex   0
Parch   0
SibSp   0
dtype: int64

--- Missing values (test) ---
Cabin   327
Age   86
Fare   1
Name   0
Pclass   0
PassengerId   0
Sex   0
Parch   0
SibSp   0
Ticket   0
dtype: int64
```

# 3 4) Target distribution & basic EDA

Let's inspect survival balance and some univariate distributions.

[5]:
```
# Target distribution
plt.figure(figsize=(6,4))
sns.countplot(x='Survived', data=train)
plt.title("Survived distribution (train)")
plt.xticks([0,1], ["Died (0)", "Survived (1)"])
plt.show()

Age distribution
```
```

```
plt.figure(figsize=(8,4))
sns.histplot(train['Age'].dropna(), kde=True, bins=30)
plt.title("Age distribution (train)")
plt.show()

# Fare distribution (log-scale for visualization)
plt.figure(figsize=(8,4))
sns.histplot(train['Fare'].dropna(), bins=40)
plt.title("Fare distribution (train)")
plt.show()
```

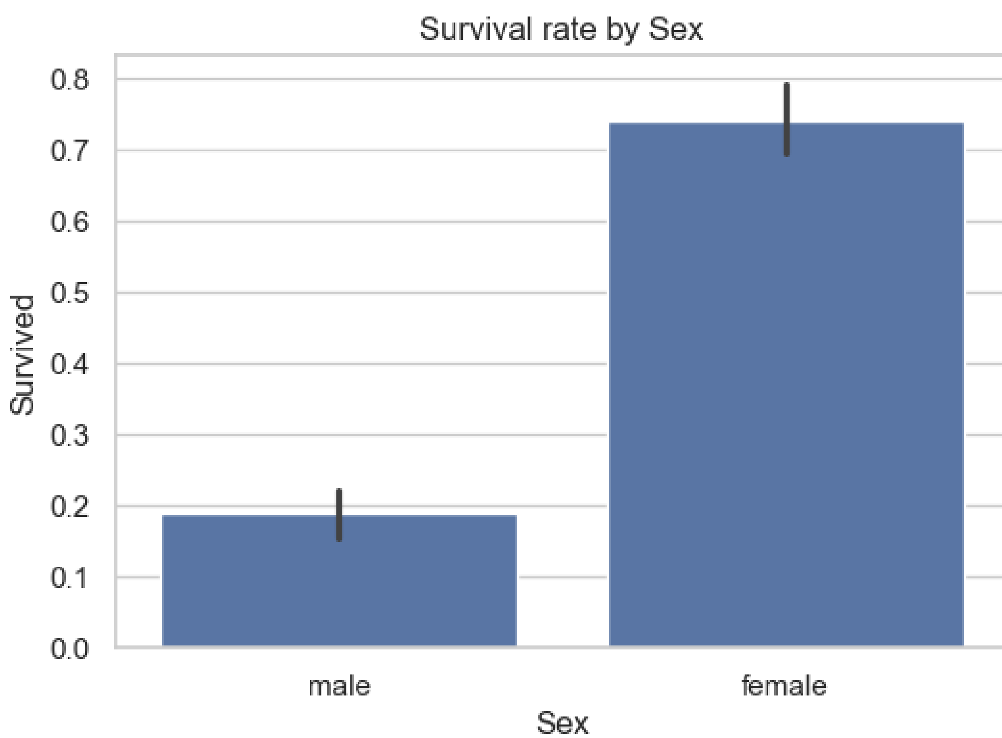


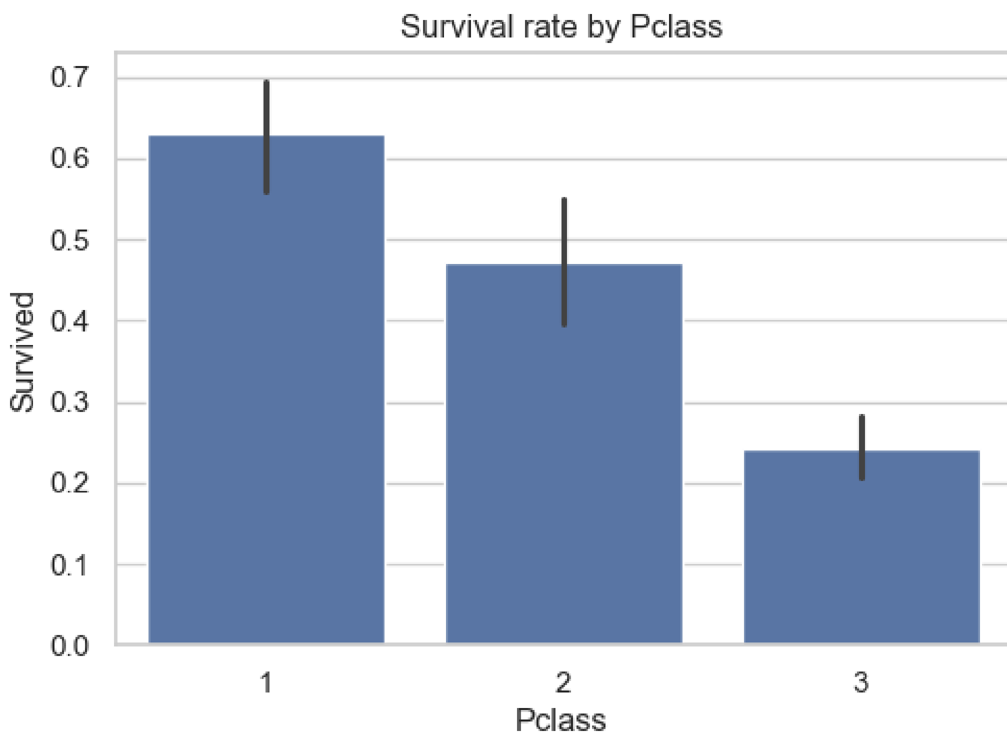


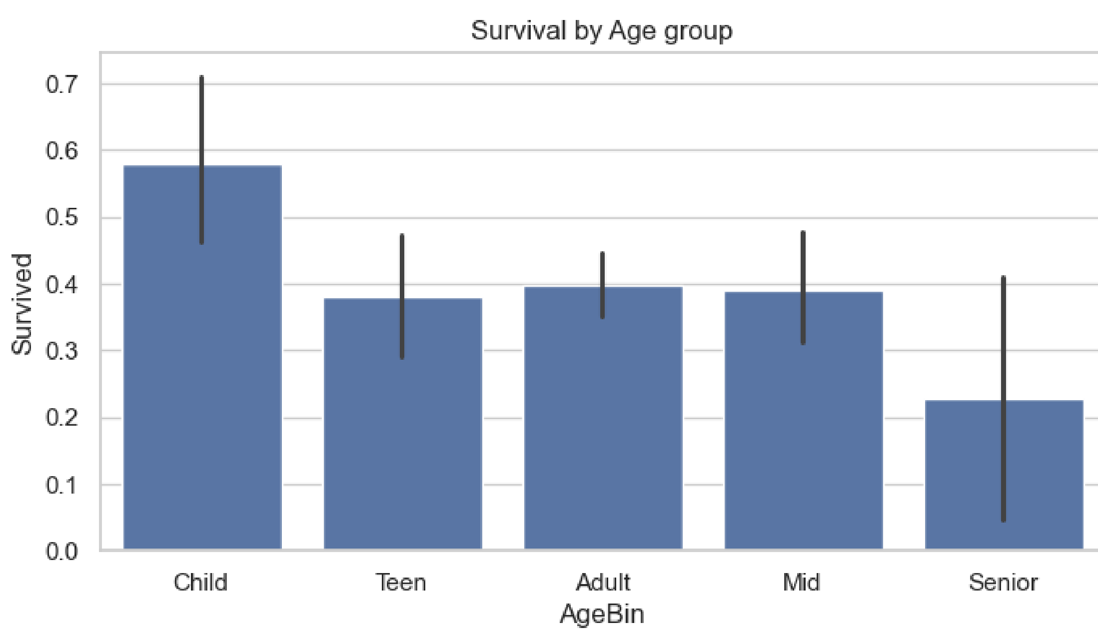
4 5) Bivariate analysis: features vs target

See how sex, class, embarcation relate to survival.

[6]:
```python
# Survival by Sex
plt.figure(figsize=(6,4))
sns.barplot(x='Sex', y='Survived', data=train)
plt.title("Survival rate by Sex")
plt.show()

# Survival by Pclass
plt.figure(figsize=(6,4))
sns.barplot(x='Pclass', y='Survived', data=train)
plt.title("Survival rate by Pclass")
plt.show()

# Survival vs Age (age bins)
train['AgeBin'] = pd.cut(train['Age'], bins=[0,12,20,40,60,80],␣
 ↪labels=['Child','Teen','Adult','Mid','Senior'])
plt.figure(figsize=(8,4))
sns.barplot(x='AgeBin', y='Survived', data=train,␣
 ↪order=['Child','Teen','Adult','Mid','Senior'])
plt.title("Survival by Age group")
plt.show()
train.drop(columns=['AgeBin'], inplace=True)
```




Survival rate by Pclass
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Survived
1
2
3
Pclass


Survival by Age group
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Survived
Child
Teen
Adult
Mid
Senior
AgeBin

5 6) Feature engineering: create helpful features

We'll create: - `Title` from Name (Mr, Mrs, Miss, Master, Rare) - `FamilySize` = SibSp + Parch + 1 - `IsAlone` from FamilySize - `Deck` from Cabin (optional; many NA)

[7]:
```python
def extract_title(name):
    if pd.isna(name):
        return "None"
    # common format: "Last, Title. First"
    title = name.split(',')[1].split('.')[0].strip()
    return title

def simplify_title(title):
    # Map many rare titles to 'Rare' and standardize
    title = title.lower()
    if title in ['mr', 'mrs', 'miss', 'master']:
        return title.title()
    if title in ['ms']:
        return 'Miss'
    if title in ['mme', 'mademoiselle']:
        return 'Mrs'
    # everything else rare
    return 'Rare'

def add_features(df):
    df = df.copy()
    # Title
    df['Title'] = df['Name'].apply(extract_title).apply(simplify_title)
    # Family size
    df['FamilySize'] = df['SibSp'] + df['Parch'] + 1
    df['IsAlone'] = (df['FamilySize'] == 1).astype(int)
    # Deck (first letter of Cabin). Will be NaN if Cabin missing.
    df['Deck'] = df['Cabin'].apply(lambda x: str(x)[0] if pd.notna(x) else np.
↪nan)
    return df

train_fe = add_features(train)
test_fe = add_features(test)

print("Titles in train:", train_fe['Title'].value_counts().to_dict())
display(train_fe[['Title','FamilySize','IsAlone','Deck']].head())
```

```
Titles in train: {'Mr': 517, 'Miss': 183, 'Mrs': 126, 'Master': 40, 'Rare': 25}
```

	Title	FamilySize	IsAlone	Deck
0	Mr	2	0	NaN
1	Mrs	2	0	C
2	Miss	1	1	NaN
3	Mrs	2	0	C

```
4     Mr           1         1  NaN
```

6 7) Handle missing values with domain-specific logic

Strategy: - Fill `Embarked` with mode - Fill `Fare` missing in test with median - Impute `Age` by median of `Title` groups when available, else overall median - Drop `Cabin` (too many missing) — we keep `Deck` if desired but it has lots of missing

[9]:
```
# Copy dataframes to avoid modifying originals
train_clean = train_fe.copy()
test_clean = test_fe.copy()

# Embarked -> fill with mode
for df in [train_clean, test_clean]:
    if 'Embarked' in df.columns:
        df['Embarked']=df['Embarked'].fillna(df['Embarked'].mode()[0])

# Fare -> fill with median (test may have missing Fare)
test_clean['Fare']=test_clean['Fare'].fillna(test_clean['Fare'].median())

# Age -> fill by Title median
title_age_median = train_clean.groupby('Title')['Age'].median()
# fallback median
overall_age_median = train_clean['Age'].median()

def fill_age_by_title(row):
    if pd.notna(row['Age']):
        return row['Age']
    title = row['Title']
    if pd.notna(title) and title in title_age_median.index and pd.
 ↪notna(title_age_median.loc[title]):
        return title_age_median.loc[title]
    return overall_age_median

train_clean['Age'] = train_clean.apply(fill_age_by_title, axis=1)
test_clean['Age']  = test_clean.apply(fill_age_by_title, axis=1)

# Drop Cabin column (we have Deck but it's sparse)
for df in [train_clean, test_clean]:
    if 'Cabin' in df.columns:
        df.drop(columns=['Cabin'], inplace=True)

# Quick check
print("Missing values after cleaning (train):")
print(train_clean.isnull().sum().sort_values(ascending=False).head(10))
print("\nMissing values after cleaning (test):")
print(test_clean.isnull().sum().sort_values(ascending=False).head(10))
```

```
Missing values after cleaning (train):
Deck           687
Survived         0
PassengerId      0
Name             0
Sex              0
Age              0
Pclass           0
SibSp            0
Parch            0
Fare             0
dtype: int64

Missing values after cleaning (test):
Deck           327
PassengerId      0
Name             0
Sex              0
Age              0
Pclass           0
SibSp            0
Parch            0
Fare             0
Ticket           0
dtype: int64
```

7 8) Prepare final feature list & split target

We'll select a set of features (mix of numeric & categorical) and prepare X/y.

[10]:
```python
# Feature selection
features = ['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked',␣
↪'Title', 'FamilySize', 'IsAlone', 'Deck']
# Note: Deck will have many NaNs; it will be handled by the imputer/encoder

X = train_clean[features].copy()
y = train_clean['Survived'].copy()

X_test_final = test_clean[features].copy()  # for final predictions

print("X shape:", X.shape)
display(X.head())
```

```
X shape: (891, 11)

   Pclass     Sex   Age  SibSp  Parch     Fare Embarked Title  FamilySize  \
0       3    male  22.0      1      0   7.2500        S    Mr           2
1       1  female  38.0      1      0  71.2833        C   Mrs           2
```

```
2       3  female  26.0      0      0   7.9250        S  Miss            1
3       1  female  35.0      1      0  53.1000        S   Mrs            2
4       3    male  35.0      0      0   8.0500        S    Mr            1

   IsAlone Deck
0        0  NaN
1        0    C
2        1  NaN
3        0    C
4        1  NaN
```

8 9) Split into training and validation sets

We'll use an 80/20 split for local evaluation (stratify by `y`).

[11]:
```
X_train, X_val, y_train, y_val = train_test_split(
    X, y, test_size=0.20, random_state=42, stratify=y
)
print("Train:", X_train.shape, "Validation:", X_val.shape)
```

```
Train: (712, 11) Validation: (179, 11)
```

9 10) Build preprocessing pipeline

We will: - Impute numeric columns by median and scale them - Impute categorical columns by most frequent and one-hot encode them - Combine using `ColumnTransformer` so the pipeline can be used on train & test equally

[13]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer

numeric_features = ['Age', 'SibSp', 'Parch', 'Fare', 'FamilySize']
categorical_features = ['Pclass', 'Sex', 'Embarked', 'Title', 'IsAlone', 'Deck']

numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
])

preprocessor = ColumnTransformer(transformers=[
```

```
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)
], remainder='drop')

# Test that preprocessor can fit-transform training data
preprocessor.fit(X_train)
print("Preprocessor fitted.")
```

```
Preprocessor fitted.
```

10 11) Baseline model pipelines

We'll create two pipelines: - Logistic Regression (simple, interpretable) - Random Forest (powerful tree-based)

[15]:

```
# Logistic Regression pipeline
log_pipe = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('clf', LogisticRegression(max_iter=1000, random_state=42))
])

# Random Forest pipeline
rf_pipe = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('clf', RandomForestClassifier(n_estimators=100, random_state=42))
])

# Quick cross-validation (5-fold) to get baseline scores
from sklearn.model_selection import cross_val_score
print("Logistic CV:", np.mean(cross_val_score(log_pipe, X_train, y_train, cv=5,␣
 ↪scoring='accuracy')))
print("RandomForest CV:", np.mean(cross_val_score(rf_pipe, X_train, y_train,␣
 ↪cv=5, scoring='accuracy')))
```

```
Logistic CV: 0.8188811188811188
RandomForest CV: 0.7922387471683245
```

11 12) Train on training set and evaluate on validation set

Fit both pipelines and compute accuracy, classification report, confusion matrix, and ROC AUC.

[16]:

```
# Fit Logistic Regression
log_pipe.fit(X_train, y_train)
y_pred_log = log_pipe.predict(X_val)
y_prob_log = log_pipe.predict_proba(X_val)[:,1]

print("Logistic Regression")
```

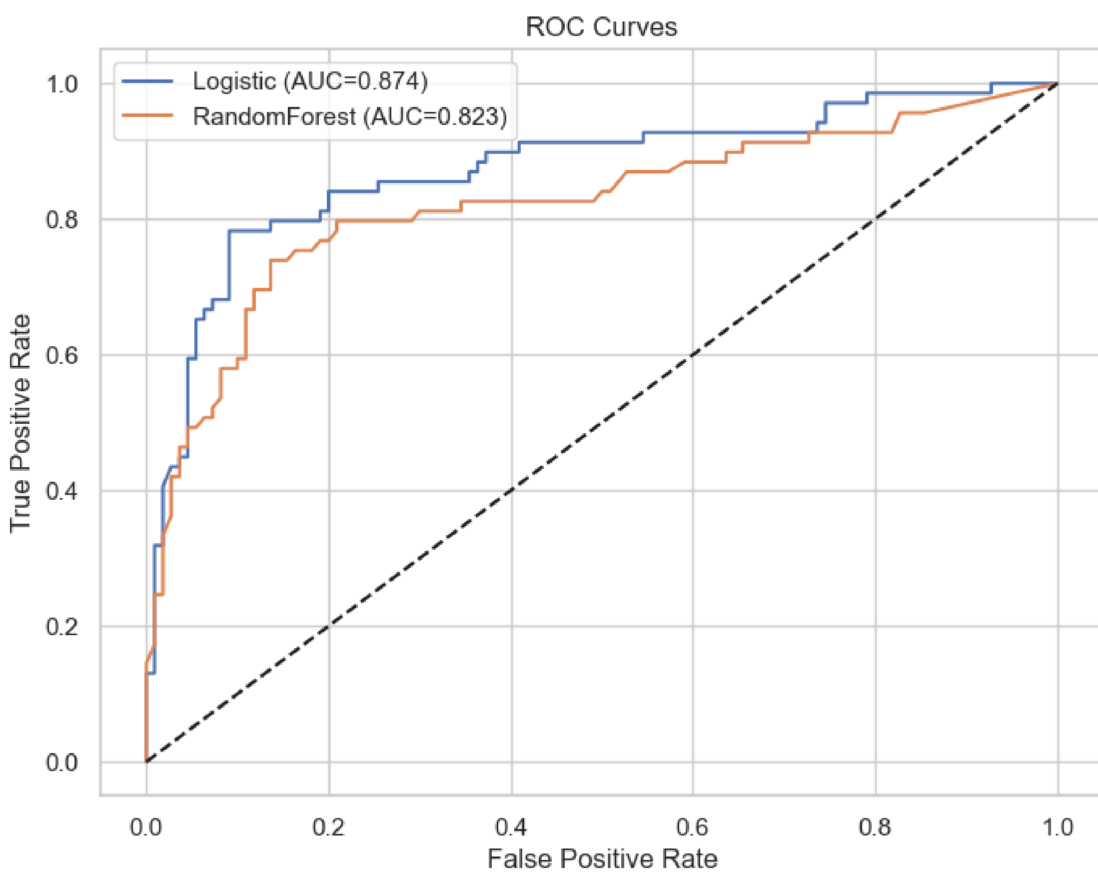
```python
print("Accuracy:", accuracy_score(y_val, y_pred_log))
print(classification_report(y_val, y_pred_log))
print("ROC AUC:", roc_auc_score(y_val, y_prob_log))
print("Confusion Matrix:\n", confusion_matrix(y_val, y_pred_log))

# Fit Random Forest
rf_pipe.fit(X_train, y_train)
y_pred_rf = rf_pipe.predict(X_val)
y_prob_rf = rf_pipe.predict_proba(X_val)[:,1]

print("\nRandom Forest")
print("Accuracy:", accuracy_score(y_val, y_pred_rf))
print(classification_report(y_val, y_pred_rf))
print("ROC AUC:", roc_auc_score(y_val, y_prob_rf))
print("Confusion Matrix:\n", confusion_matrix(y_val, y_pred_rf))
```

```
Logistic Regression
Accuracy: 0.8603351955307262
              precision    recall  f1-score   support

           0       0.87      0.91      0.89       110
           1       0.84      0.78      0.81        69

    accuracy                           0.86       179
   macro avg       0.86      0.85      0.85       179
weighted avg       0.86      0.86      0.86       179

ROC AUC: 0.8735177865612647
Confusion Matrix:
 [[100  10]
 [ 15  54]]

Random Forest
Accuracy: 0.7988826815642458
              precision    recall  f1-score   support

           0       0.82      0.86      0.84       110
           1       0.76      0.70      0.73        69

    accuracy                           0.80       179
   macro avg       0.79      0.78      0.78       179
weighted avg       0.80      0.80      0.80       179

ROC AUC: 0.8228590250329382
Confusion Matrix:
 [[95 15]
 [21 48]]
```

12 13) ROC curve comparison

Plot ROC curves for both models on the validation set.

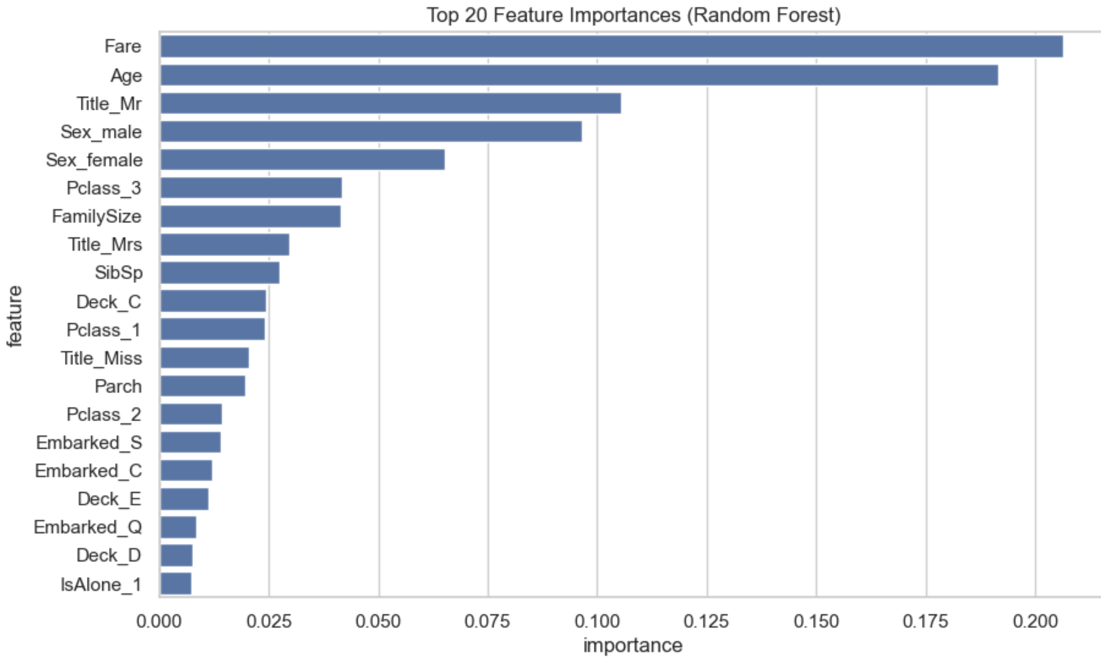
[17]:
```
fpr_log, tpr_log, _ = roc_curve(y_val, y_prob_log)
fpr_rf, tpr_rf, _ = roc_curve(y_val, y_prob_rf)

plt.figure(figsize=(8,6))
plt.plot(fpr_log, tpr_log, label=f'Logistic␣
  ↪(AUC={roc_auc_score(y_val,y_prob_log):.3f})')
plt.plot(fpr_rf, tpr_rf, label=f'RandomForest␣
  ↪(AUC={roc_auc_score(y_val,y_prob_rf):.3f})')
plt.plot([0,1],[0,1],'k--')
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curves")
plt.legend()
plt.show()
```



13 14) Feature importance (Random Forest)

To show feature importances we need the post-preprocessor feature names and the classifier's feature_importances_.

[18]:
```
# Extract numeric feature names (they remain as is)
num_feats = numeric_features

# Extract categorical feature names after OneHot encoding
# We need to get fitted OneHotEncoder to obtain categories
ohe = rf_pipe.named_steps['preprocessor'].named_transformers_['cat'].
 ↪named_steps['onehot']
try:
    cat_feature_names = list(ohe.get_feature_names_out(categorical_features))
except:
    # fallback for older sklearn versions
    cat_feature_names = []
    for i, cat in enumerate(categorical_features):
        cats = ohe.categories_[i]
        cat_feature_names += [f"{cat}_{c}" for c in cats]

feature_names = num_feats + cat_feature_names

# Get importances
importances = rf_pipe.named_steps['clf'].feature_importances_

# Build DataFrame and plot
feat_imp_df = pd.DataFrame({'feature': feature_names, 'importance':␣
 ↪importances})
feat_imp_df = feat_imp_df.sort_values(by='importance', ascending=False).
 ↪reset_index(drop=True)
plt.figure(figsize=(10,6))
sns.barplot(x='importance', y='feature', data=feat_imp_df.head(20))
plt.title("Top 20 Feature Importances (Random Forest)")
plt.show()

display(feat_imp_df.head(30))
```



	feature	importance
0	Fare	0.206273
1	Age	0.191500
2	Title_Mr	0.105510
3	Sex_male	0.096514
4	Sex_female	0.065112
5	Pclass_3	0.041688
6	FamilySize	0.041341
7	Title_Mrs	0.029868
8	SibSp	0.027396
9	Deck_C	0.024355
10	Pclass_1	0.024215
11	Title_Miss	0.020396
12	Parch	0.019639
13	Pclass_2	0.014462
14	Embarked_S	0.014108
15	Embarked_C	0.012118
16	Deck_E	0.011274
17	Embarked_Q	0.008531
18	Deck_D	0.007618
19	IsAlone_1	0.007444
20	Title_Master	0.006981
21	IsAlone_0	0.006648
22	Title_Rare	0.004793
23	Deck_B	0.004516
24	Deck_A	0.003098

```
25         Deck_F     0.002203
26         Deck_G     0.002032
27         Deck_T     0.000368
```

14 15) Hyperparameter tuning (Random Forest) — small GridSearchCV

This is a small grid; you can expand later if you want. We'll tune `n_estimators`, `max_depth`, and `min_samples_split`.

[19]:
```
param_grid = {
    'clf__n_estimators': [100, 200],
    'clf__max_depth': [None, 6, 10],
    'clf__min_samples_split': [2, 5]
}

# Use GridSearchCV with a pipeline; scoring by accuracy
grid_search = GridSearchCV(rf_pipe, param_grid, cv=5, scoring='accuracy',␣
  ↪n_jobs=-1, verbose=1)
grid_search.fit(X_train, y_train)

print("Best params:", grid_search.best_params_)
print("Best CV score:", grid_search.best_score_)

best_rf = grid_search.best_estimator_
# evaluate on validation set
y_pred_best = best_rf.predict(X_val)
y_prob_best = best_rf.predict_proba(X_val)[:,1]
print("Validation accuracy (best RF):", accuracy_score(y_val, y_pred_best))
print("Validation ROC AUC (best RF):", roc_auc_score(y_val, y_prob_best))
```

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
Best params: {'clf__max_depth': 6, 'clf__min_samples_split': 5,
'clf__n_estimators': 200}
Best CV score: 0.8245149216980202
Validation accuracy (best RF): 0.8100558659217877
Validation ROC AUC (best RF): 0.8538208168642951
```

15 16) Train final model on the *entire* training dataset

Retrain the best model on full `train` data (not the local validation split) for final predictions on `test.csv`.

[20]:
```
# Prepare full training data (we already have X, y from earlier)
# If we used grid_search.best_estimator_, we can fit it on full data
final_model = grid_search.best_estimator_
```

```
# Fit on full training set
final_model.fit(X, y)
print("Final model trained on full training data.")
```

```
Final model trained on full training data.
```

16 17) Create predictions for `test.csv` and build submission file

Kaggle expects a CSV with `PassengerId` and `Survived` columns.

[25]:
```
# Ensure test has PassengerId
if 'PassengerId' not in test.columns:
    raise ValueError("test.csv must contain 'PassengerId' column for creating␣
 ↪submission file.")

# Predict
test_preds = final_model.predict(X_test_final)

submission = pd.DataFrame({
    "PassengerId": test['PassengerId'],
    "Survived": test_preds.astype(int)
})

submission_filename = "../data/titanic_submission.csv"
submission.to_csv(submission_filename, index=False)
print(f"Submission saved to {submission_filename}")
display(submission.head())
```

```
Submission saved to ../data/titanic_submission.csv
```

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

17 18) Save the final model to disk

We save the whole pipeline so that preprocessing is included.

[24]:
```
model_filename = "../model/titanic_final_model.joblib"
joblib.dump(final_model, model_filename)
print(f"Saved model pipeline to {model_filename}")
```

```
Saved model pipeline to ../model/titanic_final_model.joblib
```