데이터 전처리

In [48]:
```
import pandas as pd
import glob
import numpy as np
import seaborn as sns

sns.set(style="whitegrid", color_codes=True)

# 데이터 불러오기
DATA_DIR = 'titanic'
data_files = sorted(glob.glob(DATA_DIR + "/*.csv"), reverse=True)
print("불러온 파일:", data_files)

# train, test 각각 읽기
train_df = pd.read_csv(data_files[0])
test_df = pd.read_csv(data_files[1])

n_train = len(train_df)
n_test = len(test_df)

# 두 데이터 합치기
df = pd.concat([train_df, test_df], sort=False).reset_index(drop=True)
print("통합 후 데이터 크기:", df.shape)
```

```
불러온 파일: ['titanic/train.csv', 'titanic/test.csv']
통합 후 데이터 크기: (1309, 13)
```

In [49]:
```
pd.options.display.float_format = '{:.2f}'.format
print(df.isnull().sum() / len(df) * 100)
```

```
Unnamed: 0     0.00
PassengerId    0.00
Survived       0.00
Pclass         0.00
Name           0.00
Sex            0.00
Age           20.09
SibSp          0.00
Parch          0.00
Ticket         0.00
Fare           0.08
Cabin         77.46
Embarked       0.15
dtype: float64
```

In [50]:
```
df[df["Embarked"].isnull()]
```

Out[50]:

	Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
61	61	62	1	1	Icard, Miss. Amelie	female	38.00	0	0	113572	80.00	B28	NaN
829	829	830	1	1	Stone, Mrs. George Nelson (Martha Evelyn)	female	62.00	0	0	113572	80.00	B28	NaN

In [51]:
```
df[df["Fare"].isnull()]
```

Out[51]:

	Unnamed: 0	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1043	152	1044	0	3	Storey, Mr. Thomas	male	60.50	0	0	3701	NaN	NaN	S

In [52]:
```
# Age: Pclass 기준 평균으로 채움
df["Age"] = df["Age"].fillna(df.groupby("Pclass")["Age"].transform("mean"))

# Embarked: 결측을 'S'로 채움
df["Embarked"] = df["Embarked"].fillna("S")

# Fare: 결측을 3등급의 평균으로 채움
mean_fare_pclass3 = df.loc[df["Pclass"] == 3, "Fare"].mean()
df["Fare"] = df["Fare"].fillna(mean_fare_pclass3)
```

In [53]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1309 entries, 0 to 1308
Data columns (total 13 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Unnamed: 0   1309 non-null   int64
 1   PassengerId  1309 non-null   int64
 2   Survived     1309 non-null   int64
 3   Pclass       1309 non-null   int64
 4   Name         1309 non-null   object
 5   Sex          1309 non-null   object
 6   Age          1309 non-null   float64
 7   SibSp        1309 non-null   int64
 8   Parch        1309 non-null   int64
 9   Ticket       1309 non-null   object
 10  Fare         1309 non-null   float64
 11  Cabin        295 non-null    object
 12  Embarked     1309 non-null   object
dtypes: float64(2), int64(6), object(5)
memory usage: 133.1+ KB
```

In [54]:
```python
object_columns = ["PassengerId", "Pclass", "Name", "Sex",
                  "Ticket", "Cabin", "Embarked"]
numeric_columns = ["Age", "SibSp", "Parch", "Fare"]

for col in object_columns:
    df[col] = df[col].astype(object)
for col in numeric_columns:
    df[col] = df[col].astype(float)

df["SibSp"] = df["SibSp"].astype(int)
df["Parch"] = df["Parch"].astype(int)
```

In [55]:
```python
def merge_and_get(ldf, rdf, on=None, how="inner", index=None):
    if index is True:
        return pd.merge(ldf, rdf, how=how, left_index=True, right_index=True)
    else:
        return pd.merge(ldf, rdf, how=how, on=on)

one_hot_df = merge_and_get(df, pd.get_dummies(df["Sex"], prefix="Sex"), index=True)
one_hot_df = merge_and_get(one_hot_df, pd.get_dummies(df["Pclass"], prefix="Pclass"), index=True)
one_hot_df = merge_and_get(one_hot_df, pd.get_dummies(df["Embarked"], prefix="Embarked"), index=True)

# 원본 범주형 제거
one_hot_df = one_hot_df.drop(columns=["Sex", "Pclass", "Embarked"])

# 학습에 의미 없는 텍스트 컬럼 제거
drop_cols = ["PassengerId", "Name", "Ticket", "Cabin"]
one_hot_df = one_hot_df.drop(columns=drop_cols, errors="ignore")
```

In [56]:
```python
# 위에서 기록한 n_train, n_test 사용
x_train = one_hot_df.iloc[:n_train, :].copy()
x_test = one_hot_df.iloc[n_train:, :].copy()

y_train = train_df["Survived"].reset_index(drop=True)
y_test = test_df["Survived"].reset_index(drop=True)

# 입력에서는 Survived 제거
x_train = x_train.drop(columns=['Survived'], errors='ignore')
x_test = x_test.drop(columns=['Survived'], errors='ignore')

print("Train 크기:", x_train.shape)
print("Test 크기:", x_test.shape)
```

```
Train 크기: (891, 13)
Test 크기: (418, 13)
```

Linear Regression

In [57]:
```python
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# 모델 학습
lr = LinearRegression()
lr.fit(x_train, y_train)

# train 데이터로 cutoff 탐색
y_train_pred = lr.predict(x_train)

thresholds = np.arange(0.1, 0.9, 0.01)
```

```
train_results = []

for t in thresholds:
    y_train_bin = (y_train_pred >= t).astype(int)
    acc = accuracy_score(y_train, y_train_bin)
    prec = precision_score(y_train, y_train_bin)
    rec = recall_score(y_train, y_train_bin)
    f1 = f1_score(y_train, y_train_bin)
    train_results.append([t, acc, prec, rec, f1])

# DataFrame 정리
train_result_df = pd.DataFrame(train_results, columns=["Threshold", "Accuracy", "Precision", "Recall", "F1"])

# train 기준으로 best cutoff 선택
best_row = train_result_df.loc[train_result_df["F1"].idxmax()]
best_t = best_row["Threshold"]

print("Best Cutoff from Train")
print(best_row)

# 선택된 cutoff로 test 데이터 예측
y_test_pred = lr.predict(x_test)
y_test_bin = (y_test_pred >= best_t).astype(int)

# test 성능 평가
test_acc = accuracy_score(y_test, y_test_bin)
test_prec = precision_score(y_test, y_test_bin)
test_rec = recall_score(y_test, y_test_bin)
test_f1 = f1_score(y_test, y_test_bin)

print(f"\nLinear Regression 성능 (using best cutoff)")
print(f"Cutoff: {best_t:.2f}")
print(f"Accuracy : {test_acc:.3f}")
print(f"Precision: {test_prec:.3f}")
print(f"Recall   : {test_rec:.3f}")
print(f"F1 Score : {test_f1:.3f}")
```

```
Best Cutoff from Train
Threshold   0.35
Accuracy    0.78
Precision   0.68
Recall      0.81
F1          0.74
Name: 25, dtype: float64

Linear Regression 성능 (using best cutoff)
Cutoff: 0.35
Accuracy : 0.730
Precision: 0.613
Recall   : 0.772
F1 Score : 0.683
```

Logistic Regression

In [58]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report
import numpy as np

# 로지스틱 회귀 모델 학습
log_reg = LogisticRegression(max_iter=1000)
log_reg.fit(x_train, y_train)

# 예측
y_pred_proba = log_reg.predict_proba(x_test)[:, 1]  # 생존(1)일 확률
y_pred = (y_pred_proba >= 0.5).astype(int)

# 성능 평가
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)

print(f"Logistic Regression 성능")
print(f"Accuracy : {acc:.3f}")
print(f"Precision: {prec:.3f}")
print(f"Recall   : {rec:.3f}")
print(f"F1 Score : {f1:.3f}")
```

```
Logistic Regression 성능
Accuracy : 0.770
Precision: 0.704
Recall   : 0.677
F1 Score : 0.690
```

두 모델을 비교해 봅시다. 어느 모델이 성능이 나은가요?

두 모델은 전반적으로 비슷한 수준의 분류 성능을 보이지만 세부적인 성능 지표와 안정성 측면에서 차이를 보인다. 선형 회귀는 F1 Score가 0.68로 로지스틱 회귀(0.69)와 거의 유사한 수준이었으며 Recall(재현율)이 0.77로 높게 나타났다. 이는 실제 생존자를 놓치지 않고 잘 예측하는 경향을 보이며 민감도가 중요한 상황에서 유리하게 작용할 수 있다. 반면 Accuracy(정확도)와 Precision(정밀도)은 로지스틱 회귀가 더 높은 수치를 기록했다. 로지스틱 회귀는 모델이 생존이라고 예측한 경우 실제로 생존일 확률이 더 높다는 점에서 신뢰도가 높다. 또한 F1 Score가 비슷한 수준을 유지하면서 전체적인 지표 간의 균형이 보다 안정적으로 나타났다. 종합적으로 볼 때 선형 회귀는 재현율이 높아 민감도가 중요한 상황에서 적합하지만 로지스틱 회귀는 정확도와 정밀도가 높고 모델의 안정성이 우수하여 일반적인 분류 문제에서는 로지스틱 회귀가 더 적합한 모델이라고 볼 수 있다. 따라서 로지스틱 회귀가 전반적으로 더 일관적이고 신뢰할 수 있는 성능을 보이는 모델로 볼 수 있다.

왜 성능 차이가 날까요?

두 모델의 성능 차이는 학습 원리와 손실 함수의 구조적 차이에서 비롯된다. 선형 회귀는 평균제곱오차를 최소화하여 연속적인 값을 예측하도록 설계된 모델이다. 따라서 분류 문제에 적용할 경우 예측 결과가 확률이 아닌 단순 실수 값으로 출력되며 이를 0과 1로 구분하기 위해 임의적인 cutoff설정이 필요하다. 이러한 특성 때문에 선형 회귀는 예측값의 범위가 제한되지 않고 분류 기준에 따라 결과가 크게 달라질 수 있다. 반면, 로지스틱 회귀는 시그모이드 함수를 사용해 예측값을 자연스럽게 0과 1 사이의 확률로 변환한다. 또한 회귀 문제가 아닌 분류 문제에 특화된 로그 손실을 최소화하여 클래스 간의 경계를 보다 안정적이고 일관되게 학습할 수 있다. 결국 두 모델의 성능 차이는 선형 회귀가 분류 문제에 부적합한 손실 함수를 사용한다는 점과 로지스틱 회귀가 확률 기반의 최적화 구조를 통해 분류에 최적화되어있다는 점에서 발생한다. 즉, 로지스틱 회귀는 분류 문제에서 해석하기 쉽고 안정적인 결과를 제공하는 반면 선형 회귀는 연속 예측에 적합하지만 분류 문제에서는 근본적인 한계를 갖는다.

왜 선형회귀를 분류기로 쓰면 안 될까요?

선형회귀는 연속적인 값을 예측하기 위한 회귀 분석 기법으로 수치 예측 문제에는 적합하지만 생존 여부 예측과 같은 이진 분류 문제에 적용할 경우 근본적인 한계가 존재한다. 첫째, 선형 회귀의 예측값은 $-\infty$ 에서 $+\infty$ 까지의 연속적인 실수 범위를 가지므로 확률로 해석할 수 없다. 예측값이 -0.3이나 1.5와 같이 비현실적인 값으로 출력될 수 있으며 이는 확률적인 의미를 갖지는 못한다. 둘째, 선형 회귀는 출력이 정규화되어 있지 않기 때문에 분류를 위해 임의의 기준값을 설정해야 한다. 그러나 이 기준값의 변화에 따라 모델의 성능이 크게 달라질 수 있어 결정 경계가 불안정하고 일관성이 부족하다. 셋째, 선형 회귀의 손실 함수인 평균제곱오차는 연속형 예측 오차를 최소화하는데 초점을 맞추고 있어 클래스 확률을 최적화해야하는 분류 문제에는 부적합하다. 마지막으로, 선형 회귀는 MSE손실을 사용하기 때문에 이상치에 매우 민감하다. 극단적인 값이 존재할 경우 전체 회귀선이 크게 왜곡되어 예측 결과가 불안정해질 수 있다.