Forecast HSR score

In [54]:
```
import os, warnings
os.environ["LOKY_MAX_CPU_COUNT"] = str(os.cpu_count())
warnings.filterwarnings('ignore')
os.environ["OMP_NUM_THREADS"] = "1"        # also helps LightGBM avoid thread-overh
```

Import packages

In [51]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# scikit-learn import
from sklearn.model_selection import train_test_split, StratifiedKFold, cross_valida
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.feature_selection import SelectFromModel, RFE
from sklearn.ensemble import RandomForestClassifier, StackingClassifier, VotingClas
from xgboost import XGBClassifier
from sklearn.svm import SVC
from imblearn.pipeline import Pipeline as ImbPipeline
from imblearn.over_sampling import SMOTE
from sklearn.metrics import classification_report, roc_auc_score, f1_score, accurac
from sklearn.calibration import CalibratedClassifierCV
from imblearn.combine import SMOTEENN
from lightgbm import LGBMClassifier
```

1. Data Preparation

In [2]:
```
# Load dataset
df = pd.read_csv('dataset/dataframe.csv')

# peak at the first few row
df.head()
```

Out[2]:

	line	country_id	length	max_speed	cost	success	country	income_level	re
0	Shanghai maglev train	CHN	30.5	431	1200.0	0.0	China	Upper middle income	As Pa
1	Beiging-Shanghai	CHN	1318.0	350	34700.0	1.0	China	Upper middle income	As Pa
2	Beijing–Guangzhou	CHN	2230.0	350	42350.0	1.0	China	Upper middle income	As Pa
3	Hangzhou–Fuzhou–Shenzhen	CHN	1495.0	350	13312.0	1.0	China	Upper middle income	As Pa
4	Huhanrong PDL	CHN	2078.0	350	30400.0	1.0	China	Upper middle income	As Pa

In [3]:

```
# data types and non-null counts
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 160 entries, 0 to 159
Data columns (total 16 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   line           160 non-null    object 
 1   country_id     160 non-null    object 
 2   length         160 non-null    float64
 3   max_speed      160 non-null    int64  
 4   cost           160 non-null    float64
 5   success        160 non-null    float64
 6   country        160 non-null    object 
 7   income_level   160 non-null    object 
 8   region         160 non-null    object 
 9   gdp_growth     160 non-null    float64
 10  gdp_total      160 non-null    float64
 11  gdp_pc         160 non-null    float64
 12  rail_km        160 non-null    float64
 13  pop_thousands  160 non-null    float64
 14  pop_density    160 non-null    float64
 15  urban_rate     160 non-null    float64
dtypes: float64(10), int64(1), object(5)
memory usage: 20.1+ KB
```

In [4]:

```
# basic stats (numeric and categorical)
df.describe(include='all')
```

Out[4]:

	line	country_id	length	max_speed	cost	success	country
count	160	160	160.000000	160.000000	160.00000	160.000000	160
unique	160	28	NaN	NaN	NaN	NaN	28
top	Shanghai maglev train	DEU	NaN	NaN	NaN	NaN	Germany
freq	1	29	NaN	NaN	NaN	NaN	29
mean	NaN	NaN	302.169881	250.081250	5869.42125	0.537500	NaN
std	NaN	NaN	469.154510	52.376956	8866.44196	0.500157	NaN
min	NaN	NaN	7.700000	160.000000	55.00000	0.000000	NaN
25%	NaN	NaN	65.600000	200.000000	968.25000	0.000000	NaN
50%	NaN	NaN	153.750000	245.000000	2603.00000	1.000000	NaN
75%	NaN	NaN	332.250000	300.000000	6935.50000	1.000000	NaN
max	NaN	NaN	3422.000000	431.000000	57000.00000	1.000000	NaN

2. Exploratory Data Analysis

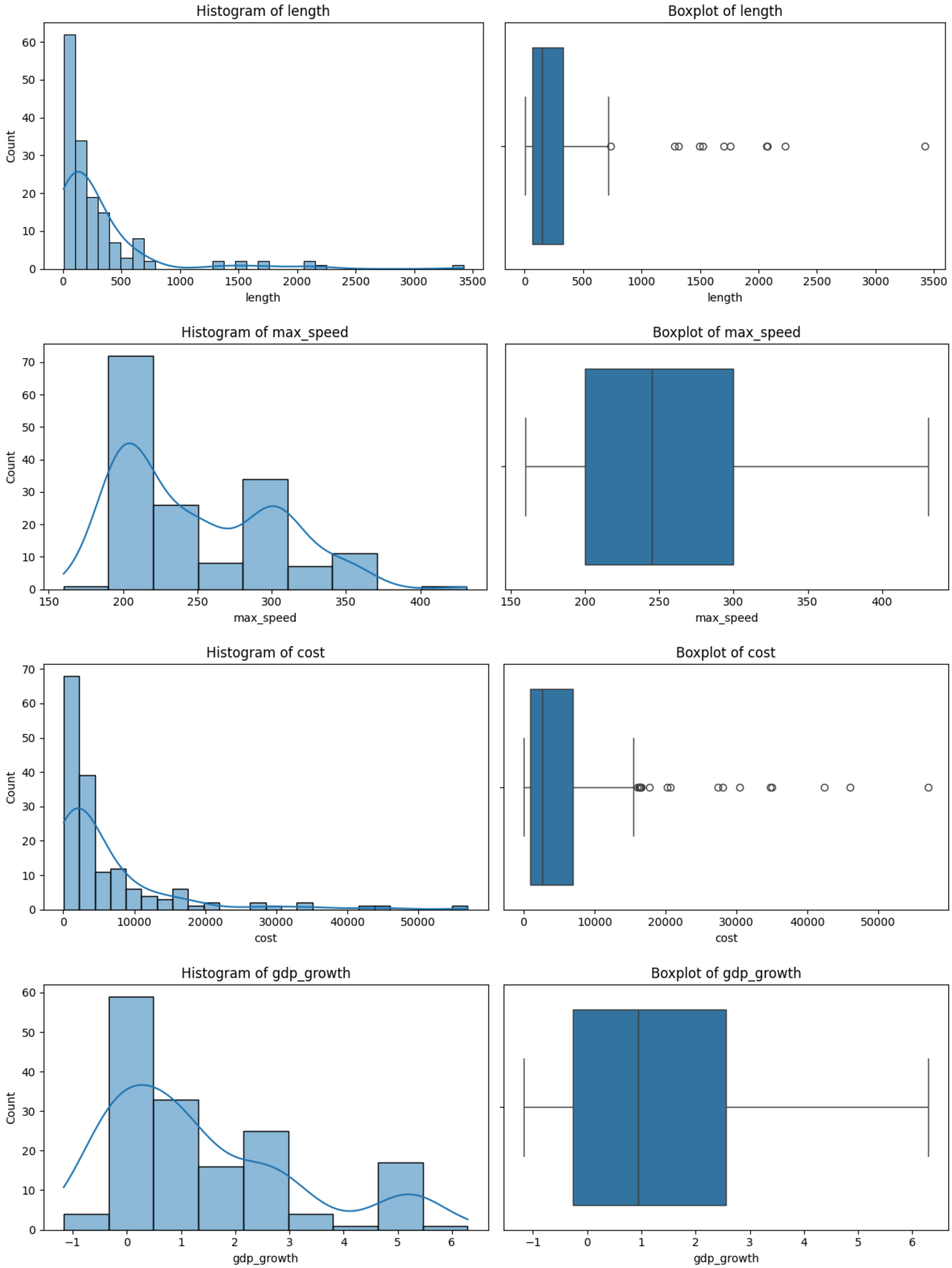
In [5]:

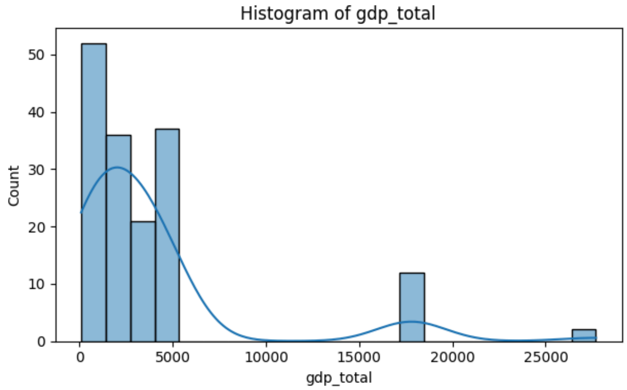
```
numeric_features = [
    'length','max_speed','cost',
    'gdp_growth','gdp_total','gdp_pc',
    'rail_km','pop_thousands','pop_density','urban_rate'
]
categorical_features = ['country_id','country','income_level','region']
```

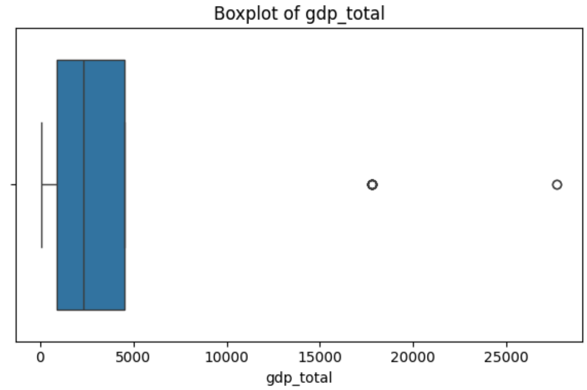
a. Univariate analysis

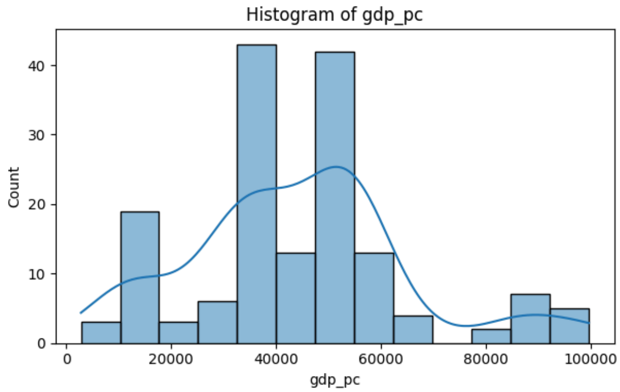
In [6]:

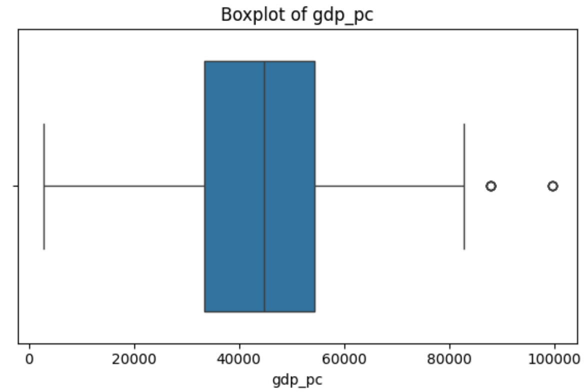
```
# Numeric features: histograms + boxplots
for col in numeric_features:
    fig, axes = plt.subplots(1, 2, figsize=(12, 4))
    sns.histplot(df[col].dropna(), kde=True, ax=axes[0])
    axes[0].set_title(f'Histogram of {col}')
    sns.boxplot(x=df[col], ax=axes[1])
    axes[1].set_title(f'Boxplot of {col}')
    plt.tight_layout()
    plt.show()
```

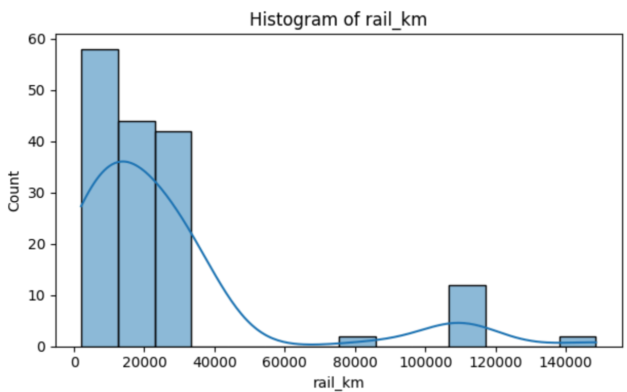

Histogram of length
Count
length
Boxplot of length
length
Histogram of max_speed
Count
max_speed
Boxplot of max_speed
max_speed
Histogram of cost
Count
cost
Boxplot of cost
cost
Histogram of gdp_growth
Count
gdp_growth
Boxplot of gdp_growth
gdp_growth

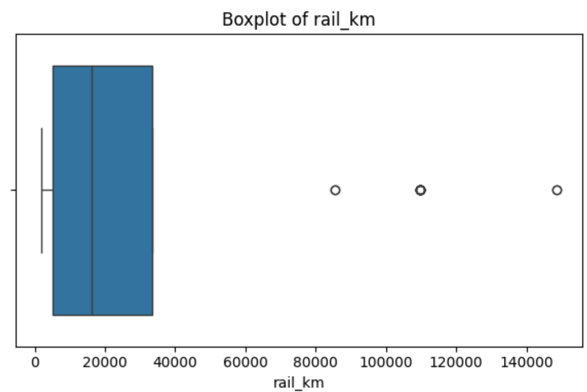

Histogram of gdp_total
Count
gdp_total

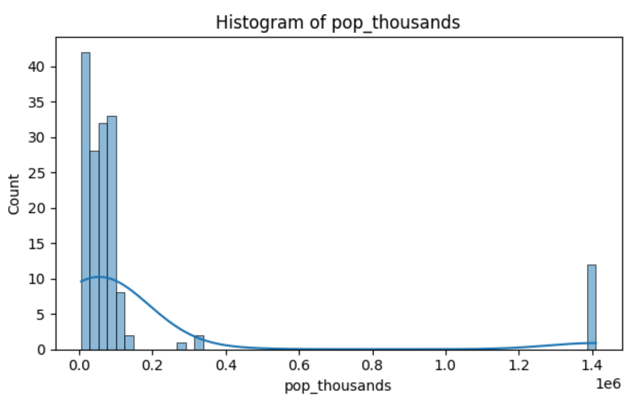

Boxplot of gdp_total
gdp_total

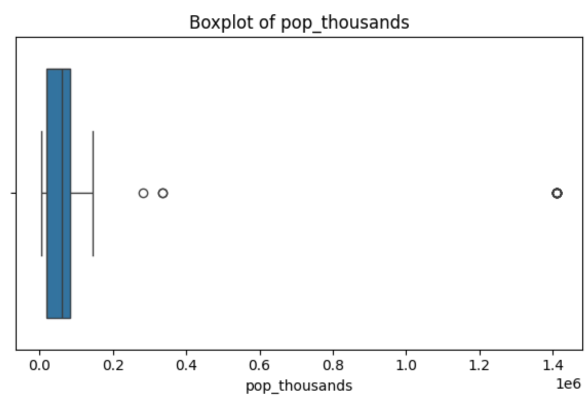

Histogram of gdp_pc
Count
gdp_pc

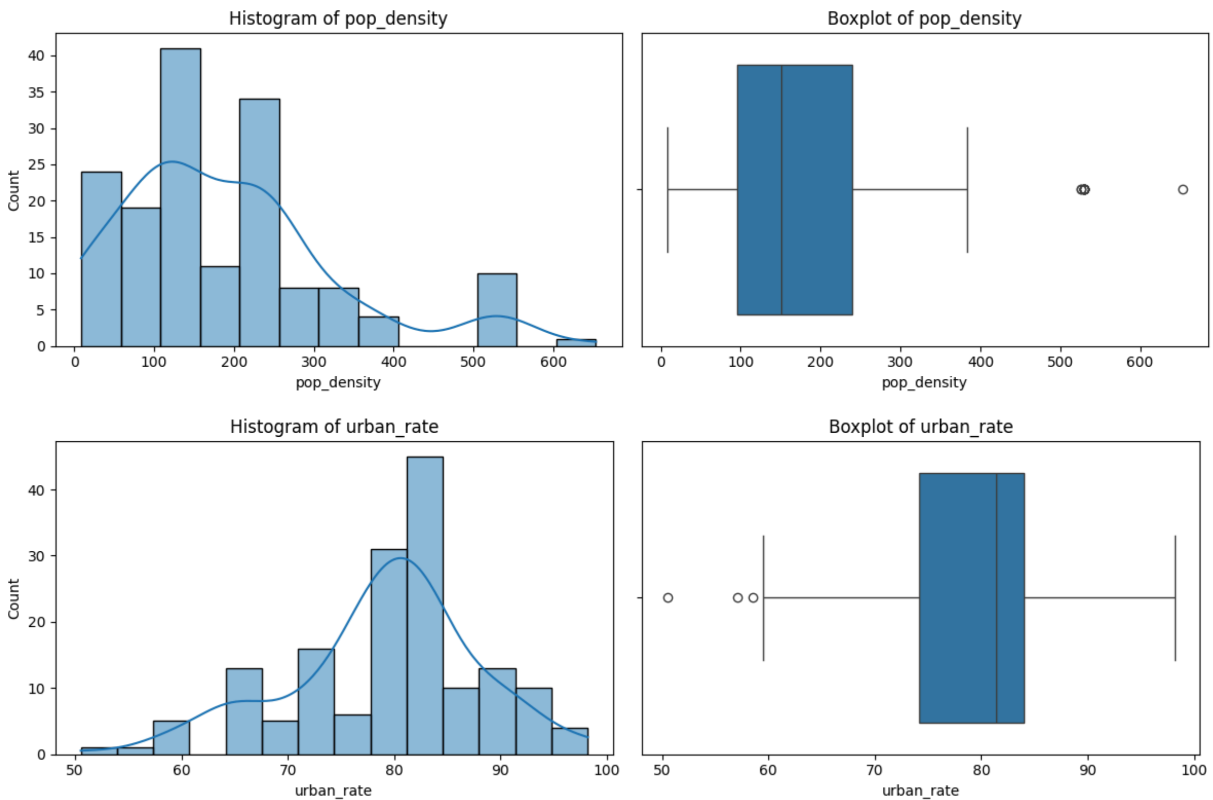

Boxplot of gdp_pc
gdp_pc


Histogram of rail_km
Count
rail_km

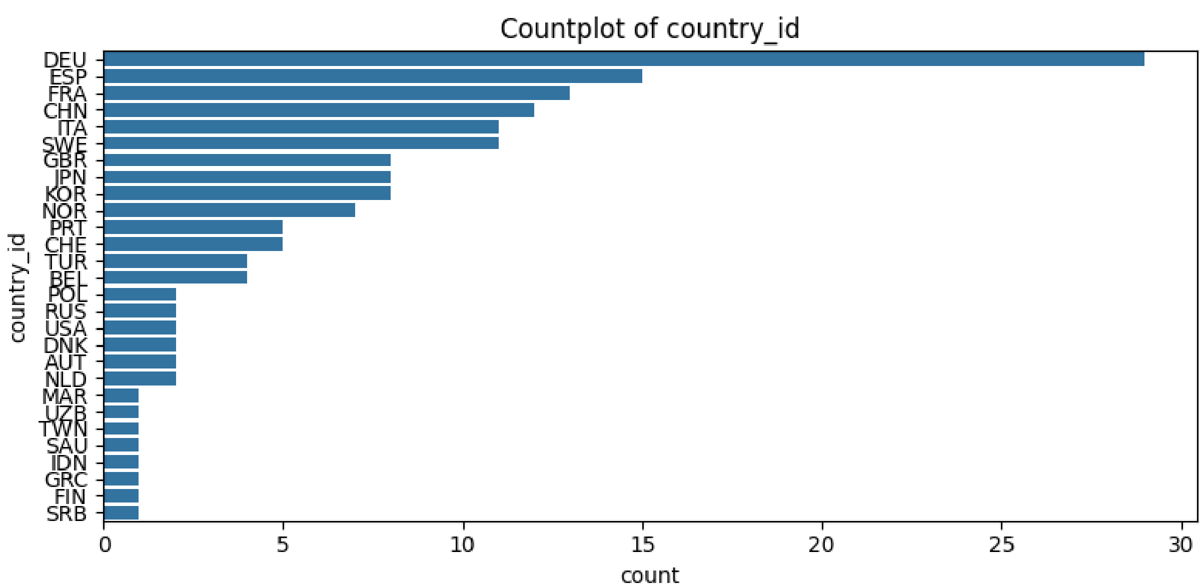

Boxplot of rail_km
rail_km

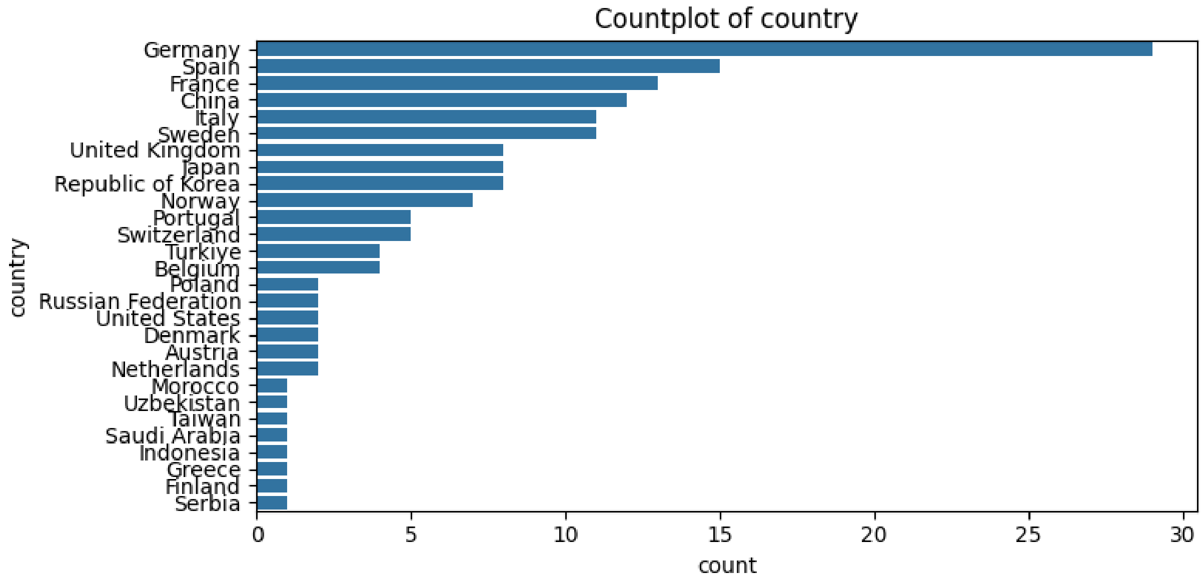

Histogram of pop_thousands
Count
pop_thousands
1e6

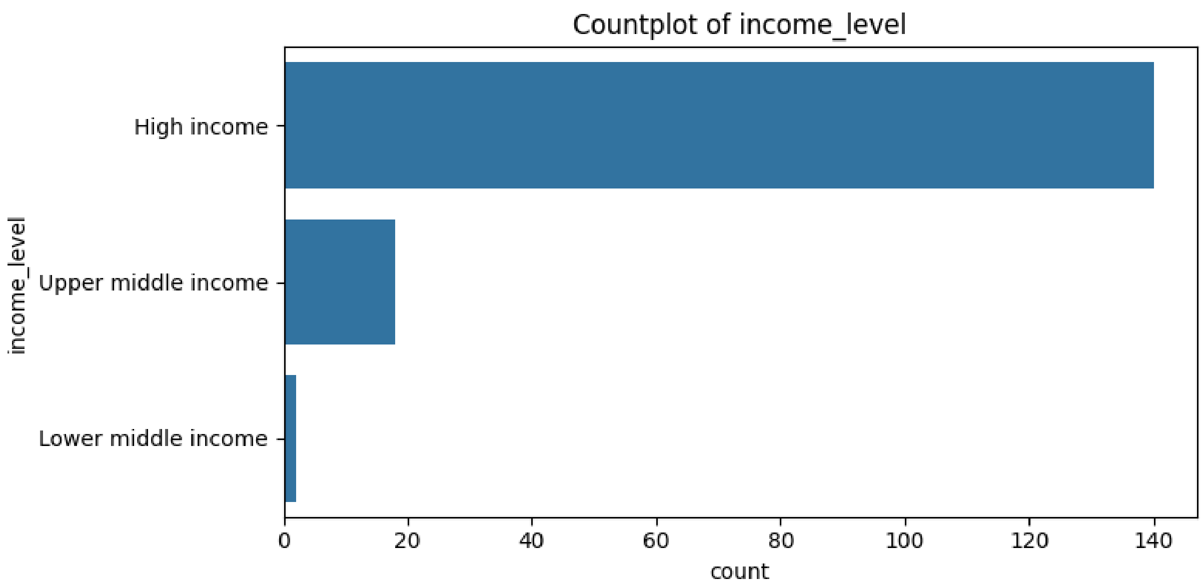

Boxplot of pop_thousands
pop_thousands
1e6

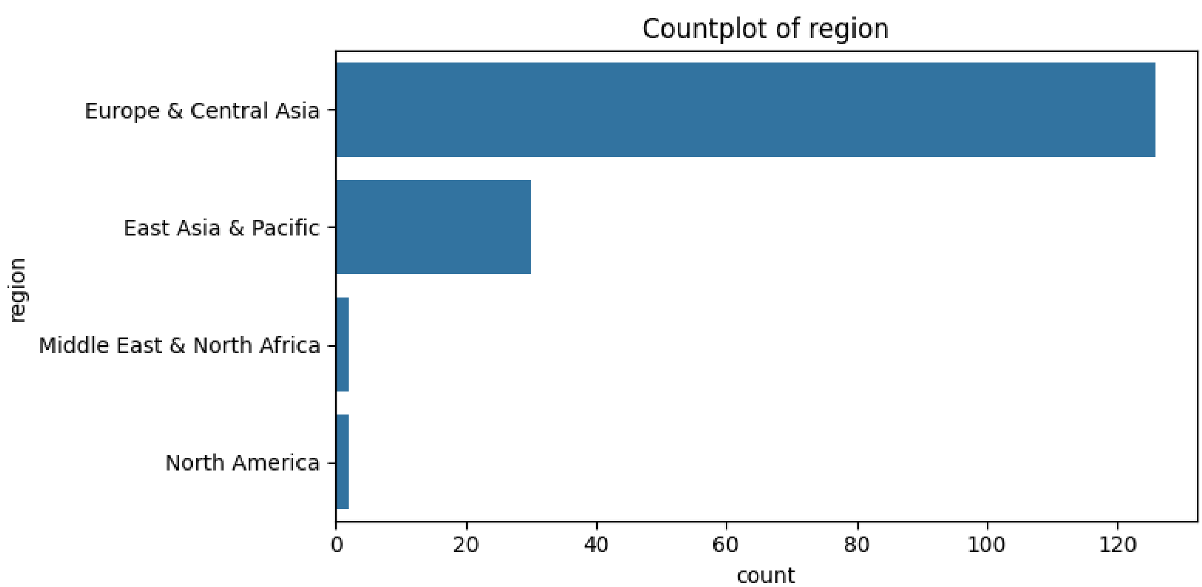


In [7]:
```
# Categorical features: countplots
for col in categorical_features:
    plt.figure(figsize=(8, 4))
    sns.countplot(y=col, data=df, order=df[col].value_counts().index)
    plt.title(f'Countplot of {col}')
    plt.tight_layout()
    plt.show()
```



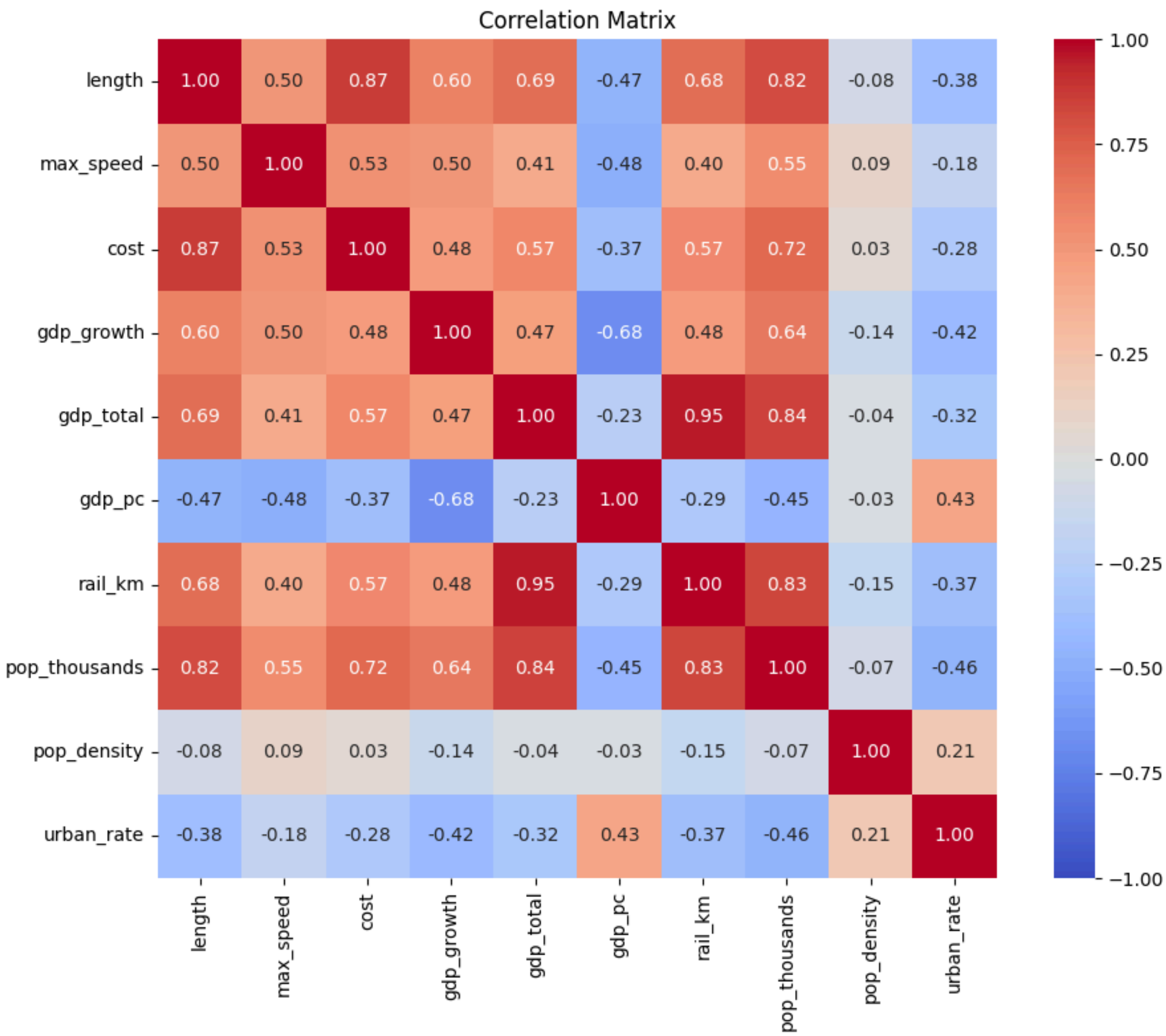

Countplot of country
country
Germany
Spain
France
China
Italy
Sweden
United Kingdom
Japan
Republic of Korea
Norway
Portugal
Switzerland
Turkiye
Belgium
Poland
Russian Federation
United States
Denmark
Austria
Netherlands
Morocco
Uzbekistan
Taiwan
Saudi Arabia
Indonesia
Greece
Finland
Serbia
0
5
10
15
20
25
30
count


Countplot of income_level
income_level
High income
Upper middle income
Lower middle income
0
20
40
60
80
100
120
140
count

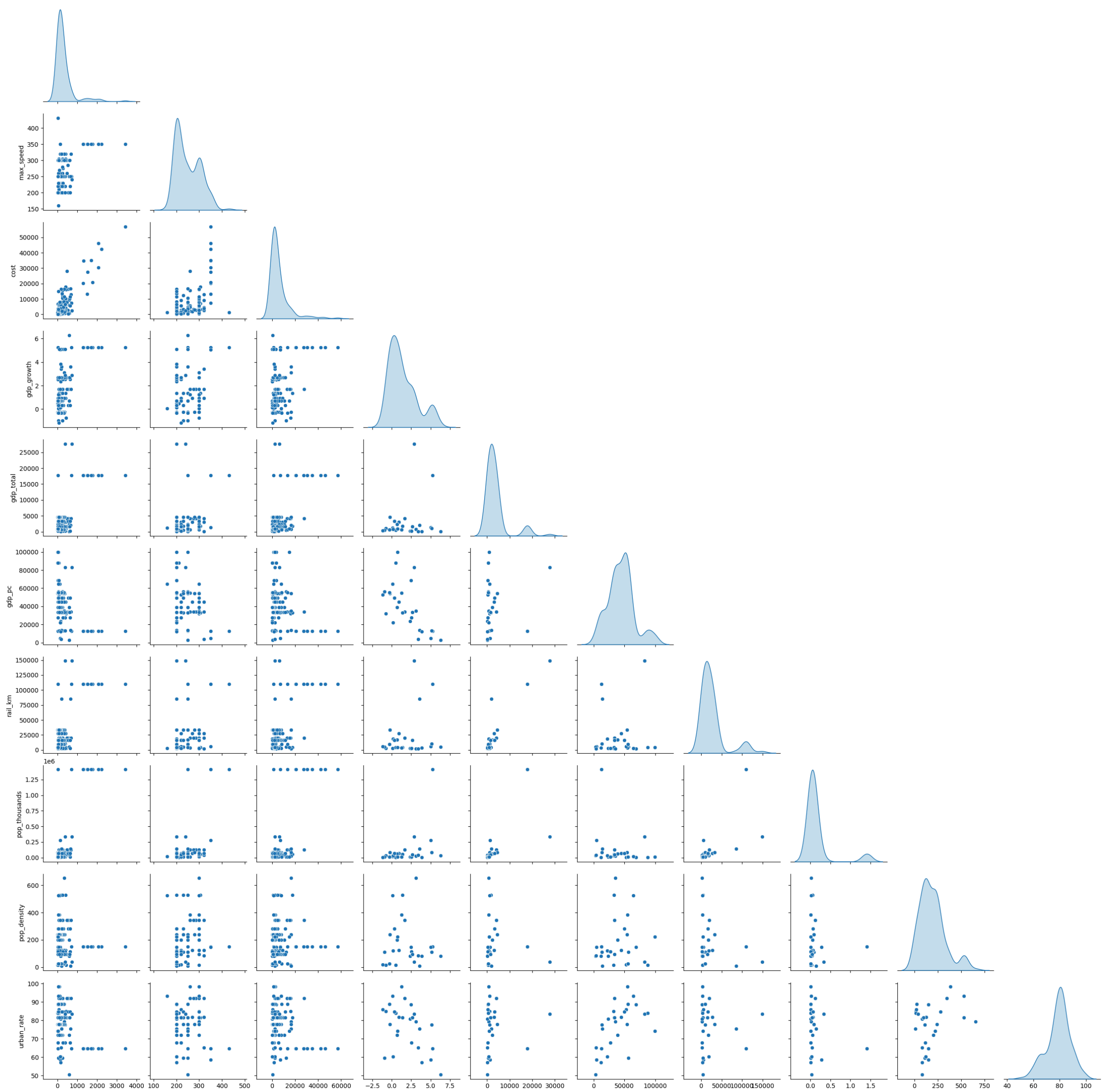

Countplot of region
region
Europe & Central Asia
East Asia & Pacific
Middle East & North Africa
North America
0
20
40
60
80
100
120
count

b. Bivariate relationships

In [8]:
```
# Correlation matrix for numeric features
plt.figure(figsize=(10, 8))
corr = df[numeric_features].corr()
sns.heatmap(corr, annot=True, fmt=".2f", cmap='coolwarm', square=True, vmax=1, vmin
plt.title('Correlation Matrix')
plt.tight_layout()
plt.show()
```

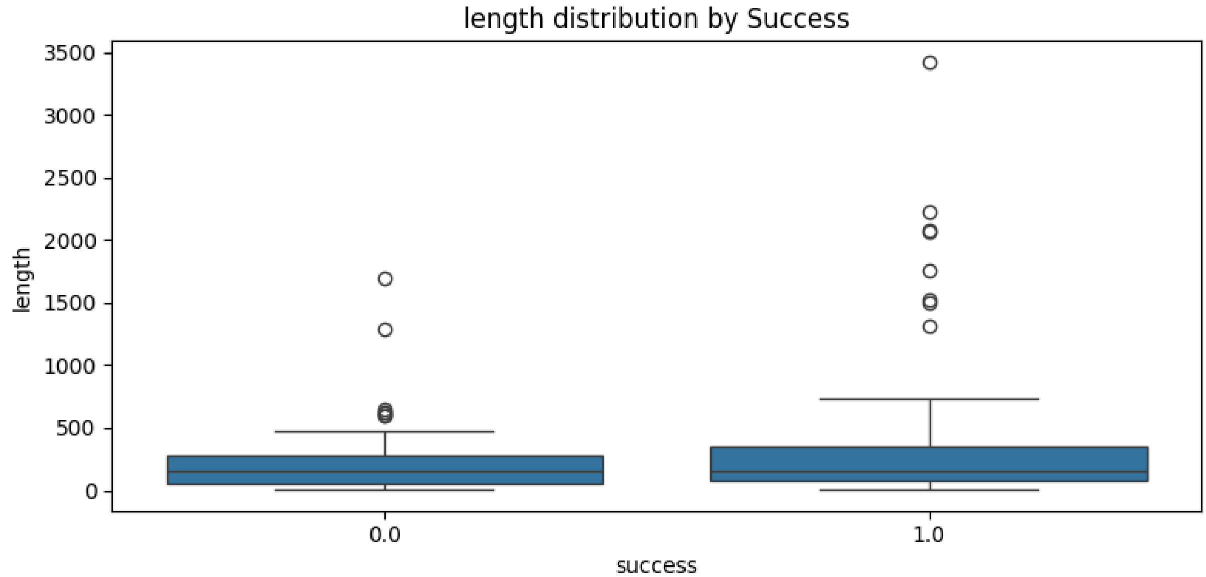


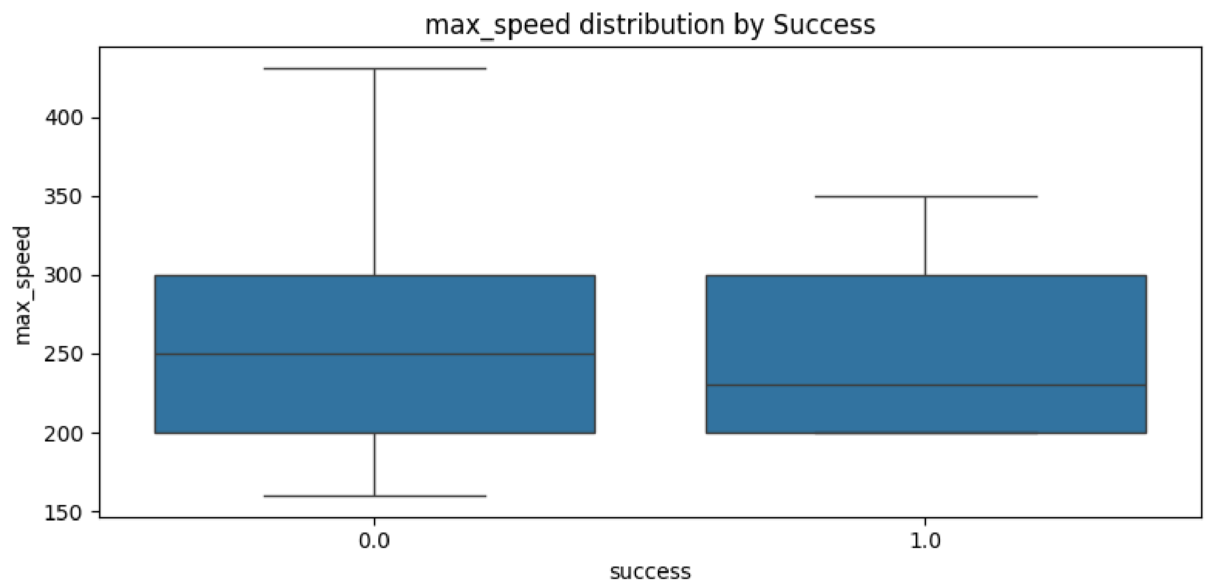
In [9]:
```
# Pairwise scatter + KDE on the diagonal
sns.pairplot(df[numeric_features], diag_kind='kde', corner=True)
plt.show()
```

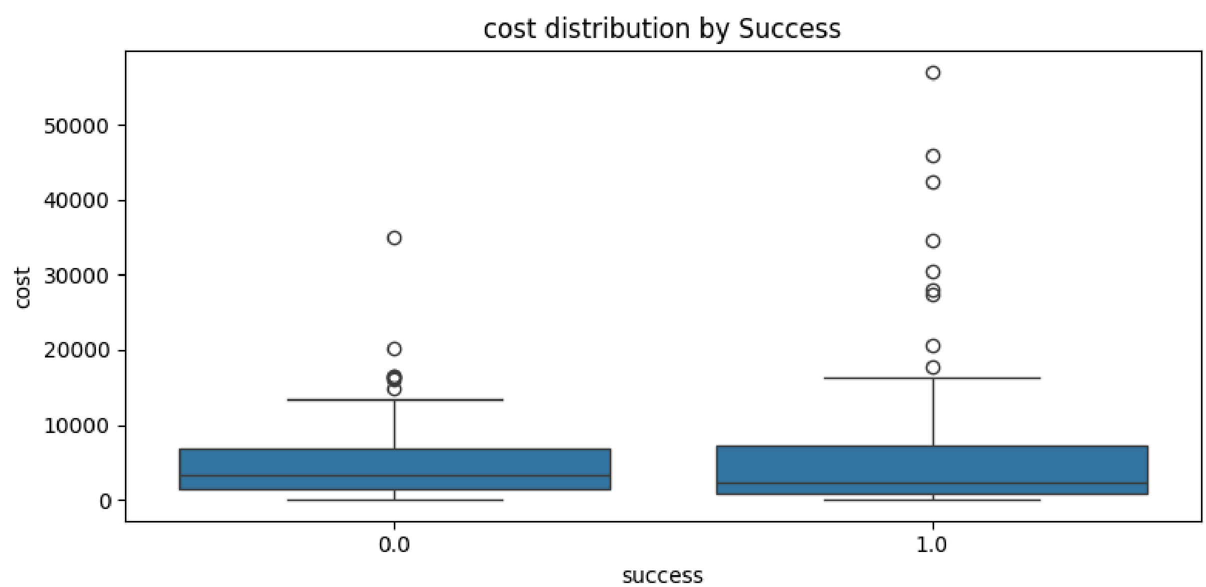


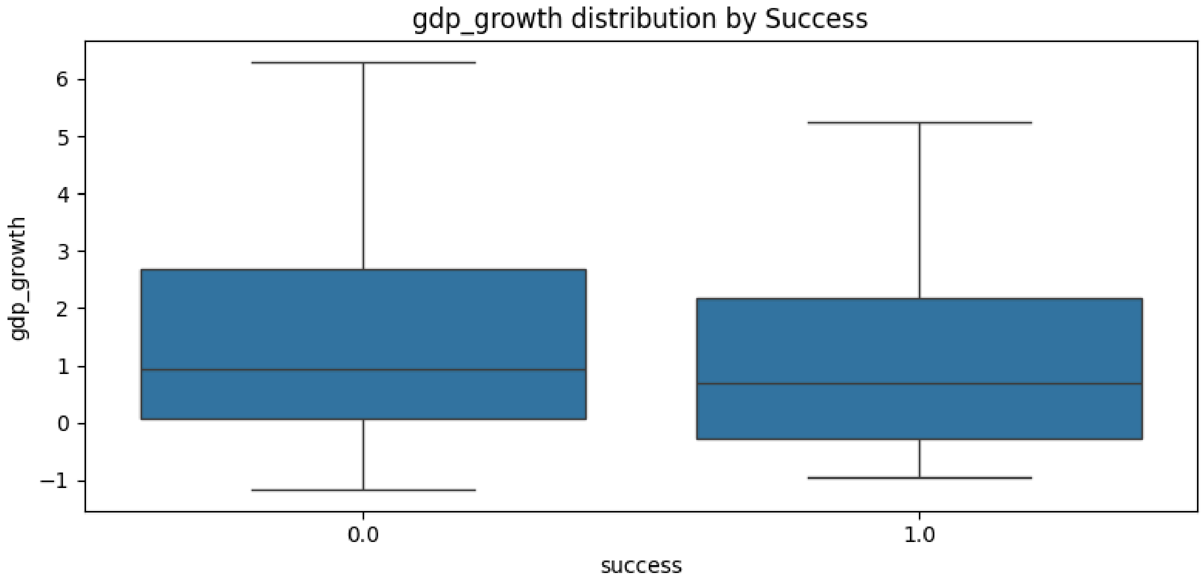
c. Target vs. features

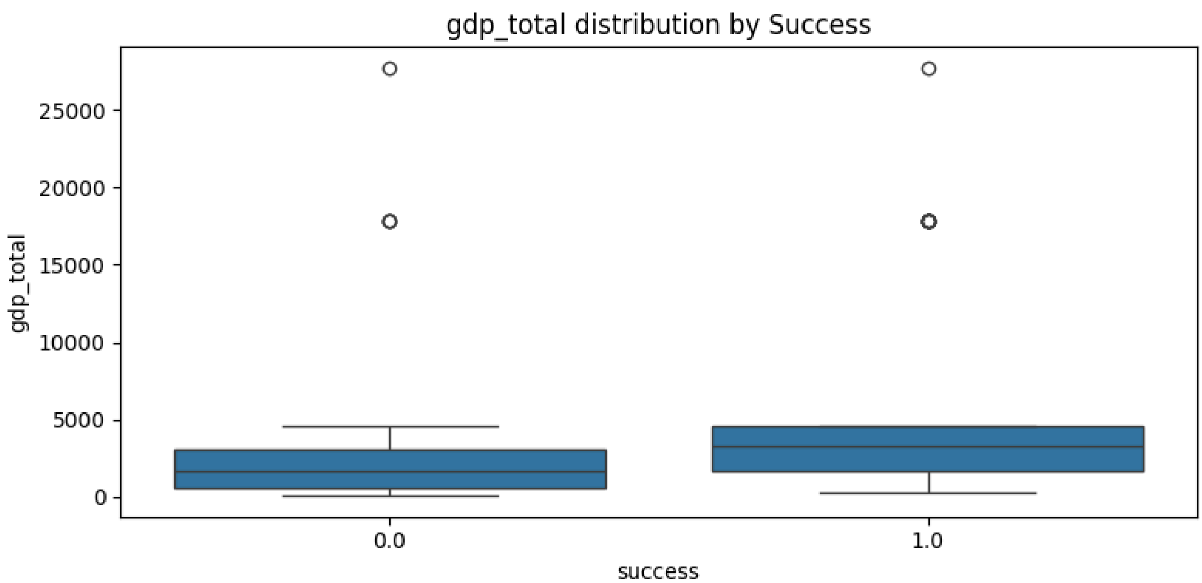
In [10]:
```
# Numeric features by target
for col in numeric_features:
    plt.figure(figsize=(8, 4))
    sns.boxplot(x='success', y=col, data=df)
    plt.title(f'{col} distribution by Success')
    plt.tight_layout()
    plt.show()
```

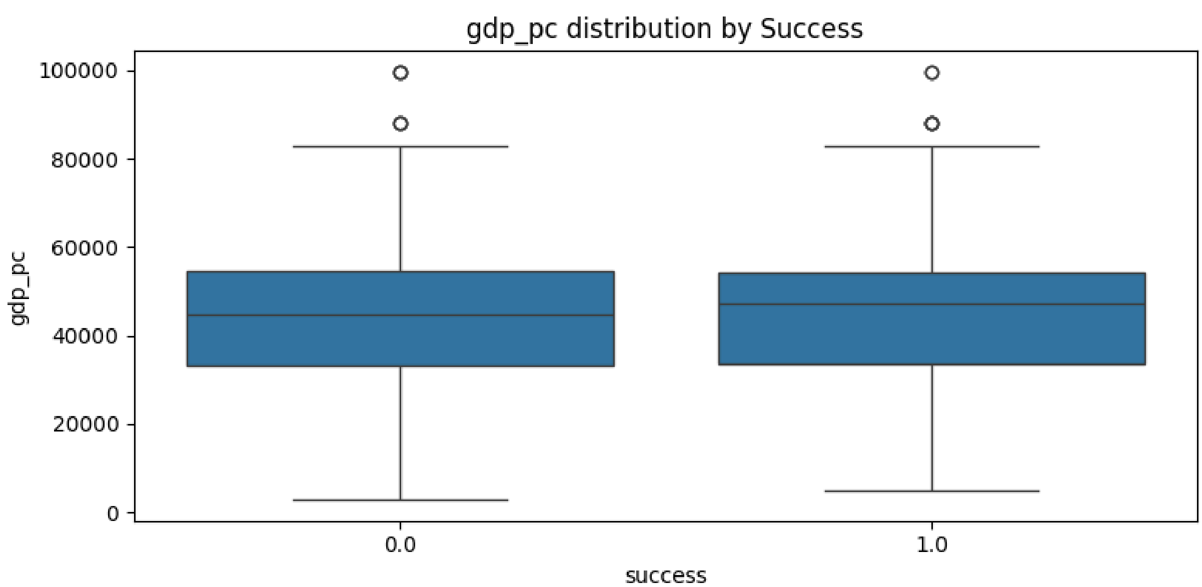

length distribution by Success
length
3500
3000
2500
2000
1500
1000
500
0
0.0
1.0
success

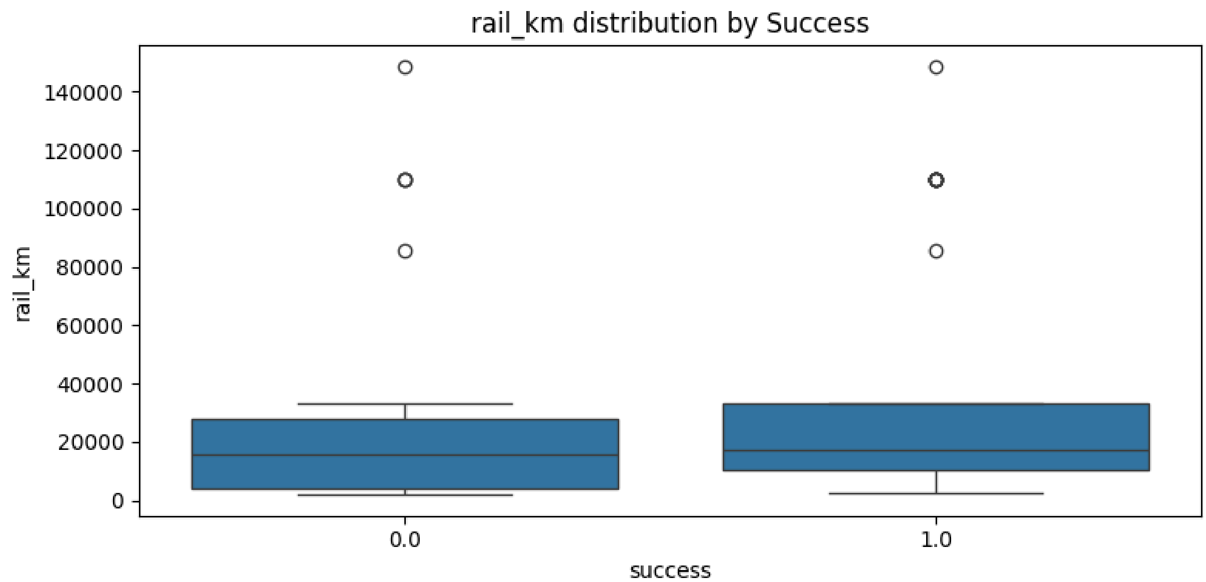

max_speed distribution by Success
max_speed
400
350
300
250
200
150
0.0
1.0
success

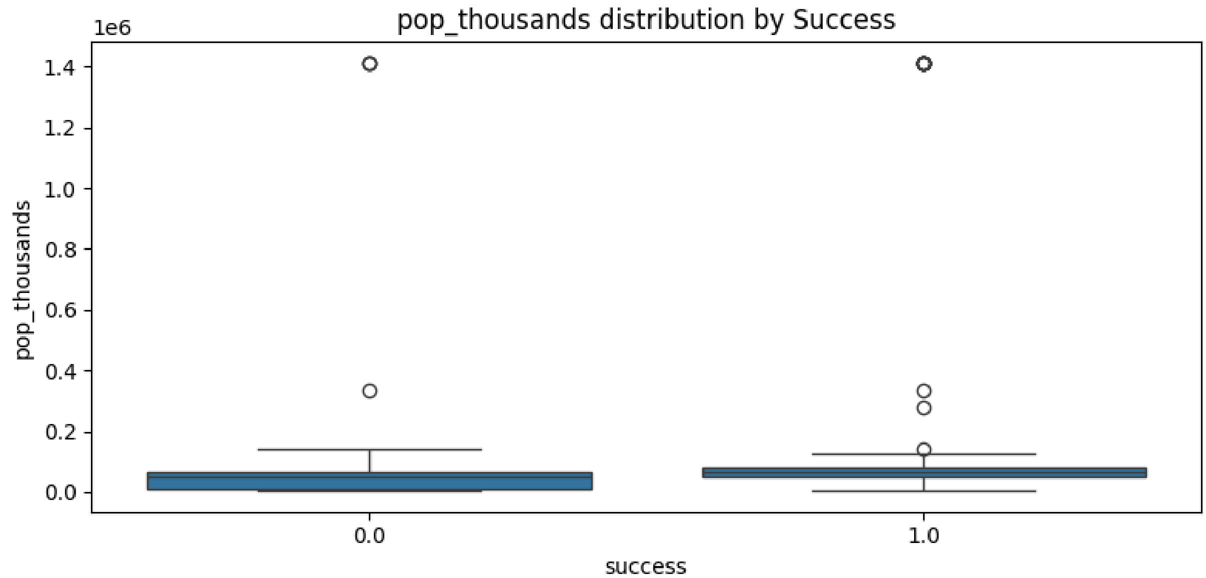

cost distribution by Success
cost
50000
40000
30000
20000
10000
0
0.0
1.0
success

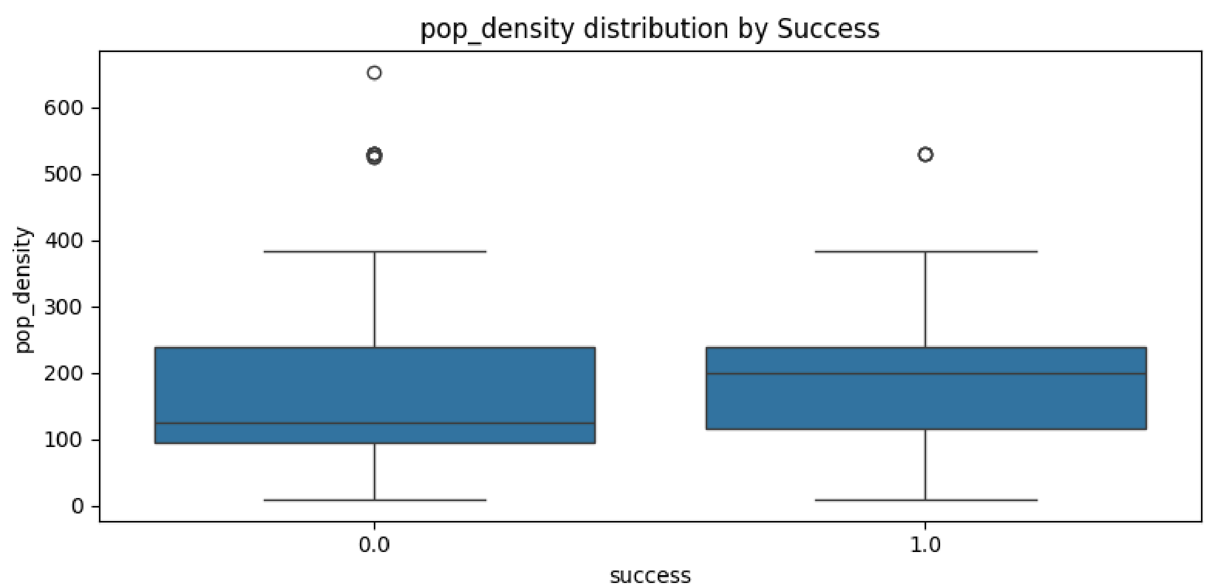

gdp_growth distribution by Success
gdp_growth
6
5
4
3
2
1
0
−1
0.0
1.0
success

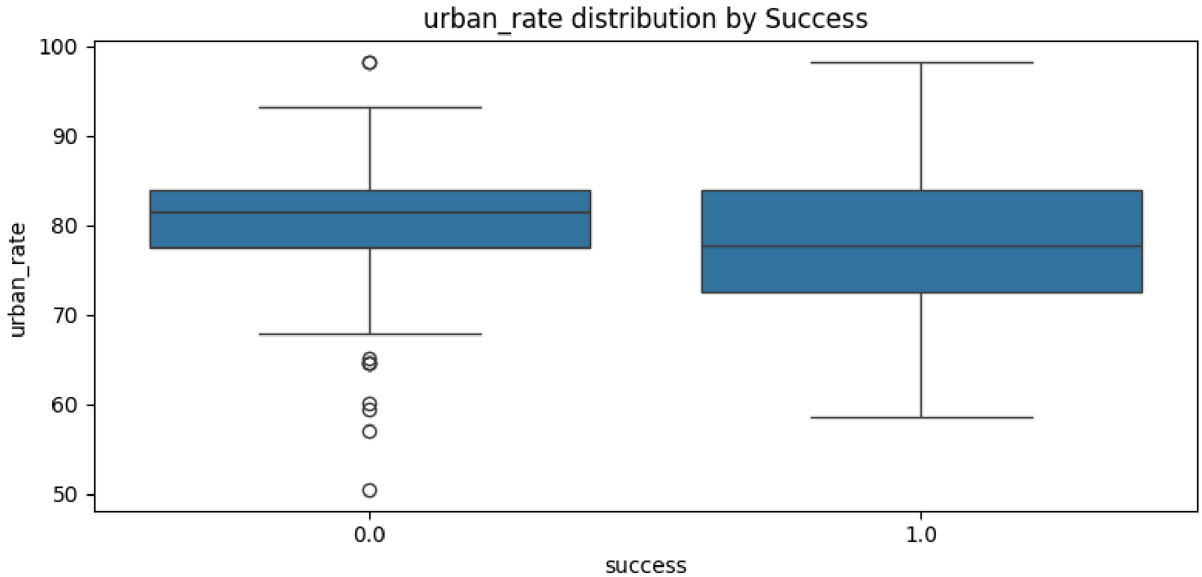

gdp_total distribution by Success
gdp_total
25000
20000
15000
10000
5000
0
0.0
1.0
success


gdp_pc distribution by Success
gdp_pc
100000
80000
60000
40000
20000
0
0.0
1.0
success


rail_km distribution by Success
rail_km
140000
120000
100000
80000
60000
40000
20000
0
0.0
1.0
success

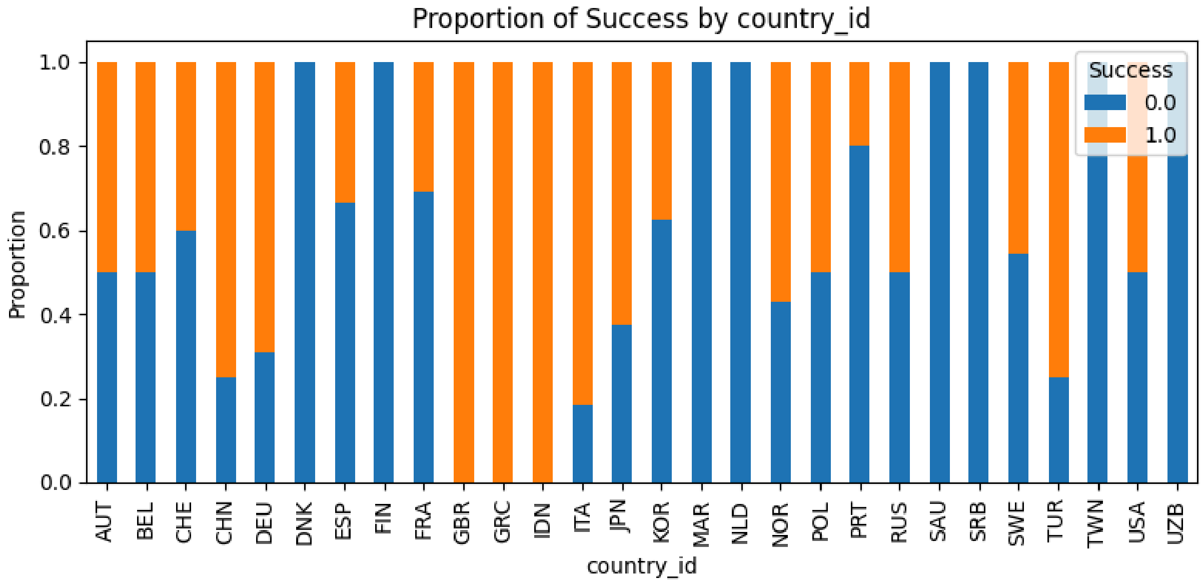

pop_thousands distribution by Success
1e6
pop_thousands
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
1.0
success

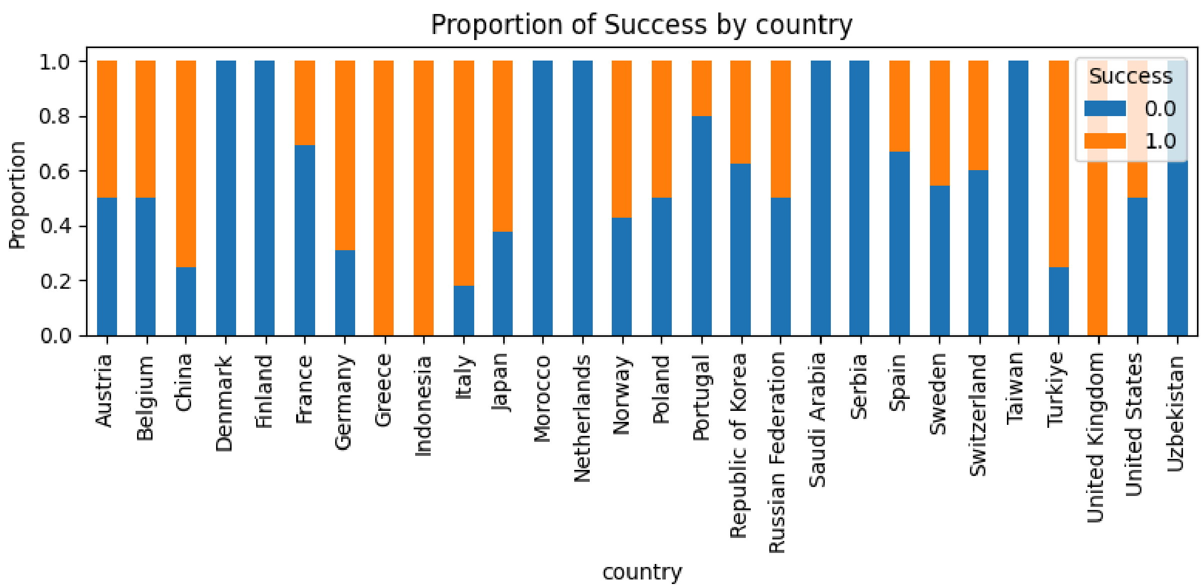

pop_density distribution by Success
pop_density
600
500
400
300
200
100
0
0.0
1.0
success

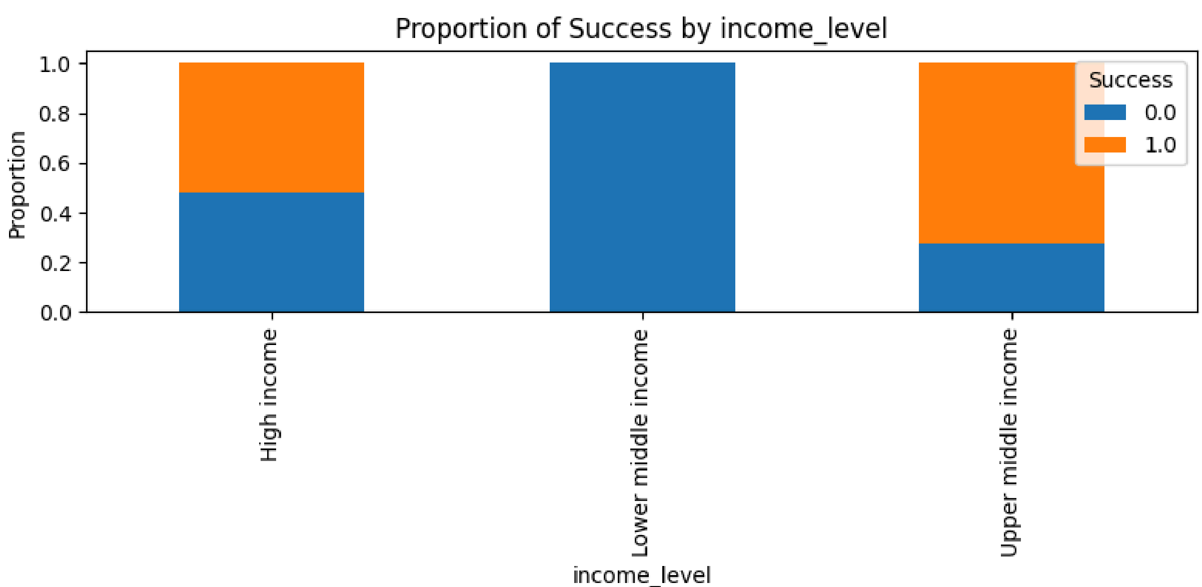


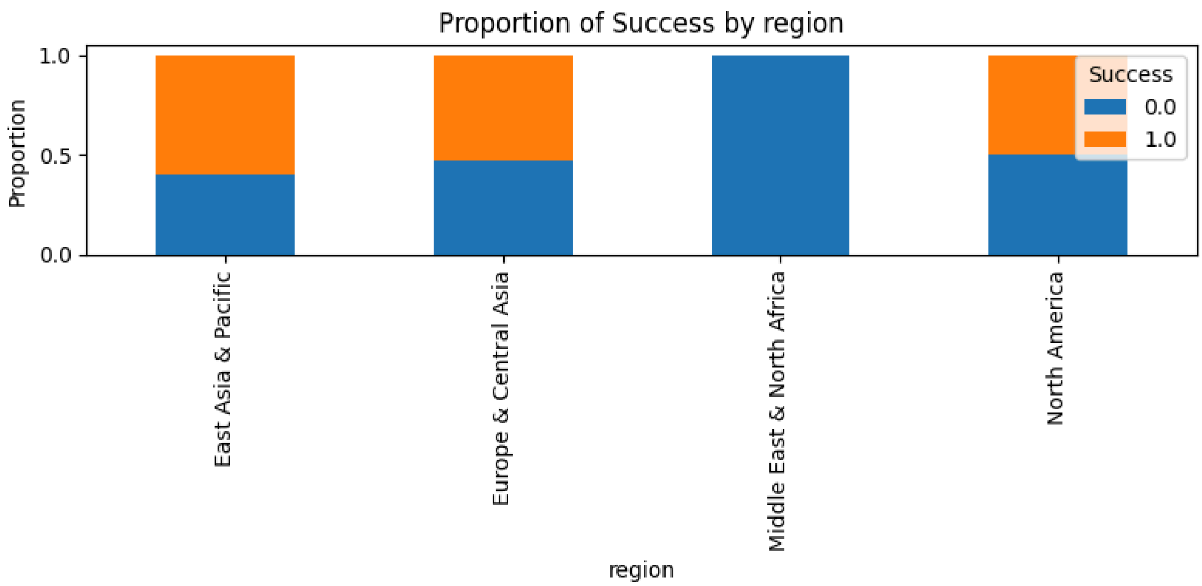
In [11]:

```
# Categorical features vs. target: stacked bar (proportions)
for col in categorical_features:
    prop = (
        df
        .groupby([col, 'success'])
        .size()
        .unstack(fill_value=0)
        .pipe(lambda d: d.div(d.sum(axis=1), axis=0))
    )
    prop.plot(
        kind='bar',
        stacked=True,
        figsize=(8, 4),
        legend=True
    )
    plt.ylabel('Proportion')
    plt.title(f'Proportion of Success by {col}')
    plt.legend(title='Success', loc='upper right')
    plt.tight_layout()
    plt.show()
```


Proportion of Success by country_id
Proportion
country_id
Success
0.0
1.0
AUT BEL CHE CHN DEU DNK ESP FIN FRA GBR GRC IDN ITA JPN KOR MAR NLD NOR POL PRT RUS SAU SRB SWE TUR TWN USA UZB


Proportion of Success by country
Proportion
country
Success
0.0
1.0
Austria Belgium China Denmark Finland France Germany Greece Indonesia Italy Japan Morocco Netherlands Norway Poland Portugal Republic of Korea Russian Federation Saudi Arabia Serbia Spain Sweden Switzerland Taiwan Turkiye United Kingdom United States Uzbekistan


Proportion of Success by income_level
Proportion
income_level
Success
0.0
1.0
High income
Lower middle income
Upper middle income



Insight

1. Univariate distribution

- Length and cost are highly right-skewed, with a long tail of very long/very expensive lines ⇒ may be need to log-transform these to tame outliers.
- Max speed clusters around a few discrete values (200, 250, 300, 350, 431 km/h) ⇒ it may act more like a categorical or ordinal variable than continuous.
- GDP nad population metrics also show heavy skew ⇒ log scaling before modeling.

2. Categorial breakdowns

- We may drop the countries and regions columns, since it doesn't match our model's purpose of prediction.

3. Pairwise and target relationships

- Cost vs length and gdp_total are extremely highly correlated ⇒ may drop or combine one of each pair to reduct multicollinearity.
- In the success vs feature boxplots, succesful lines tend to have higher median cost per km and higher urban_rate, while very short or very low-cost lines rarely succeed.

3. Feature engineering and selection

a. Feature engineering

In [12]:
```
# log-transform skewed numeric columns
for col in ['length','cost','gdp_total','pop_thousands','rail_km']:
    df[f'log_{col}'] = np.log1p(df[col])
```

In [13]:
```
# ratio / interaction features
df['cost_per_km']          = df['cost'] / df['length']
```

```
df['speed_to_cost']       = df['max_speed'] / df['cost']
df['density_interaction'] = df['gdp_pc'] * df['pop_density']
```

In [14]:
```
# length bins
df['length_bin'] = pd.cut(
    df['length'],
    bins=[0, 100, 500, np.inf],
    labels=['short','medium','long']
)
```

In [15]:
```
# remove country/region columns entirely
# (nothing to do here since we simply won't include them downstream)
```

In [16]:
```
# map income_level to ordinal if it exists
if 'income_level' in df.columns:
    ord_map = {'Low income':0,'Lower middle income':1,'Upper middle income':2,'High
    df['income_level'] = df['income_level'].map(ord_map)
```

b. Define features and target

In [17]:
```
numeric_features = [
    'max_speed','gdp_growth','gdp_pc','pop_density','urban_rate',
    'log_length','log_cost','log_gdp_total','log_pop_thousands','log_rail_km',
    'cost_per_km','speed_to_cost','density_interaction'
]
if 'income_level' in df.columns:
    numeric_features.append('income_level')

categorical_features = ['length_bin']

X = df[numeric_features + categorical_features]
y = df['success']
```

c. Train/test split

In [18]:
```
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,
    random_state=42
)
```

d. Prepocessing pipeline

In [19]:
```
num_pipe = Pipeline([
    ('impute', SimpleImputer(strategy='median')),
    ('scale',   StandardScaler())
])
cat_pipe = Pipeline([
    ('impute', SimpleImputer(strategy='constant', fill_value='missing')),
    ('ohe',     OneHotEncoder(handle_unknown='ignore'))
])
preprocessor = ColumnTransformer([
```

```
    ('num', num_pipe, numeric_features),
    ('cat', cat_pipe, categorical_features)
])

X_train_prep = preprocessor.fit_transform(X_train)
X_test_prep  = preprocessor.transform(X_test)
```

In [20]:
```
# retrieve feature names
ohe_names     = preprocessor \
    .named_transformers_['cat'] \
    .named_steps['ohe'] \
    .get_feature_names_out(categorical_features)
all_features = numeric_features + list(ohe_names)
```

e. Feature selection

In [21]:
```
#L1-Logistic
l1_selector = SelectFromModel(
    LogisticRegression(
        penalty='l1', solver='saga', C=1.0,
        max_iter=5000, random_state=42
    ),
    threshold='mean'
)
X_l1 = l1_selector.fit_transform(X_train_prep, y_train)
l1_feats = [
    f for f, keep in zip(all_features, l1_selector.get_support()) if keep
]
print("L1-selected features:\n", l1_feats)
```

```
L1-selected features:
 ['gdp_pc', 'urban_rate', 'log_length', 'log_gdp_total', 'cost_per_km', 'speed_to_co
st', 'length_bin_medium']
```

In [22]:
```
# Random Forest importances
rf = RandomForestClassifier(random_state=42)
rf.fit(X_train_prep, y_train)
importances = rf.feature_importances_
top_idx     = np.argsort(importances)[::-1][:10]
print("\nTop 10 RF-important features:")
for i in top_idx:
    print(f"  {all_features[i]}: {importances[i]:.4f}")
```

```
Top 10 RF-important features:
  cost_per_km: 0.1559
  speed_to_cost: 0.1340
  log_cost: 0.1330
  log_length: 0.1294
  log_pop_thousands: 0.0600
  max_speed: 0.0486
  density_interaction: 0.0483
  pop_density: 0.0475
  log_rail_km: 0.0471
  gdp_growth: 0.0454
```

In [23]:
```
# RFE (10 features)
rfe = RFE(
    estimator=LogisticRegression(
        penalty='l2', solver='liblinear', random_state=42
    ),
    n_features_to_select=10
)
rfe.fit(X_train_prep, y_train)
rfe_feats = [
    f for f, keep in zip(all_features, rfe.get_support()) if keep
]
print("\nRFE-selected features:\n", rfe_feats)
```

```
RFE-selected features:
 ['gdp_pc', 'urban_rate', 'log_length', 'log_gdp_total', 'log_pop_thousands', 'log_r
ail_km', 'cost_per_km', 'speed_to_cost', 'length_bin_medium', 'length_bin_short']
```

Insight

1. Core predictors

Across L1-Logistic, Random Forest, and RFE, a small set of features consistently emerges:

- Economic strength: gdp_pc, log_gdp_total
- Cost intensity: cost_per_km, log_cost
- Corridor scale: log_length
- Speed efficiency: speed_to_cost
- Urbanization: urban_rate
- Length category: length_bin_medium (and length_bin_short in RFE)

These align perfectly with the intuition that wealth, relative cost, physical size, and how fast a line is (per unit cost) drive "success."

2. Secondary signals

Raw demand proxies like log_pop_thousands and log_rail_km show up in RF and RFE but not in L1, suggesting they carry useful—but somewhat redundant—information.

Growth momentum (gdp_growth) and pure speed (max_speed) rank lower, indicating marginal gains once you account for cost and scale.

3. Categorical effect of length

Medium-length corridors (length_bin_medium) consistently predict success, while very short corridors (length_bin_short) appear only in RFE—hinting at a sweet-spot around 100–500 km.

4. Supervised modeling

a. Define classifiers

In [26]:
```
models = {
    'LogisticRegression': LogisticRegression(
        class_weight='balanced',
        solver='saga',
        penalty='l2',
        max_iter=5000,
        random_state=42
    ),
    'RandomForest': RandomForestClassifier(
        n_estimators=200,
        class_weight='balanced',
        random_state=42
    ),
    'XGBoost': XGBClassifier(
        use_label_encoder=False,
        eval_metric='logloss',
        scale_pos_weight=(y_train==0).sum()/(y_train==1).sum(),
        random_state=42
    ),
    'SVM': SVC(
        kernel='rbf',
        probability=True,
        class_weight='balanced',
        random_state=42
    )
}
```

b. Cross validation setup

In [27]:
```
cv      = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scoring = ['accuracy','precision','recall','f1','roc_auc']
results = {}
```

In [28]:
```
for name, clf in models.items():
    pipe = ImbPipeline([
        ('preproc', preprocessor),
        ('smote',   SMOTE(random_state=42)),
        ('clf',     clf)
    ])
    scores = cross_validate(
        pipe,
        X_train, y_train,
        cv=cv,
        scoring=scoring,
        return_train_score=False,
        n_jobs=-1
    )
    # store mean test-scores
    results[name] = {m: scores[f'test_{m}'].mean() for m in scoring}
```

In [29]:
```
# Tabulate CV results
cv_df = pd.DataFrame(results).T
```

```python
print("\nCross-Validated Performance (means):")
print(cv_df)
```

```
Cross-Validated Performance (means):
                    accuracy  precision    recall        f1   roc_auc
LogisticRegression  0.600615   0.627457  0.575824  0.591295  0.647311
RandomForest        0.640923   0.664411  0.650549  0.651505  0.704795
XGBoost             0.623692   0.640833  0.693407  0.661288  0.704113
SVM                 0.585538   0.604084  0.531868  0.558314  0.615534
```

In [ ]:

```python
#Pick best model (by ROC-AUC here) and evaluate on the hold-out set
best_name = cv_df['roc_auc'].idxmax()
print(f"\nBest model by ROC-AUC: {best_name}")

best_pipe = ImbPipeline([
    ('preproc', preprocessor),
    ('smote',   SMOTE(random_state=42)),
    ('clf',     models[best_name])
])

best_pipe.fit(X_train, y_train)
y_pred  = best_pipe.predict(X_test)
y_proba = best_pipe.predict_proba(X_test)[:, 1]

print(f"\nTest-set classification report for {best_name}:")
print(classification_report(y_test, y_pred))

print(f"Test-set ROC-AUC: {roc_auc_score(y_test, y_proba):.3f}")
```

```
Best model by ROC-AUC: RandomForest

Test-set classification report for RandomForest:
              precision    recall  f1-score   support

         0.0       0.62      0.53      0.57        15
         1.0       0.63      0.71      0.67        17

    accuracy                           0.62        32
   macro avg       0.62      0.62      0.62        32
weighted avg       0.62      0.62      0.62        32

Test-set ROC-AUC: 0.653
```


```
Test-set classification report for RandomForest:
              precision    recall  f1-score   support

         0.0       0.62      0.53      0.57        15
         1.0       0.63      0.71      0.67        17

    accuracy                           0.62        32
   macro avg       0.62      0.62      0.62        32
weighted avg       0.62      0.62      0.62        32

Test-set ROC-AUC: 0.653
```


c. Hyperparameter tuning for the RandomForest Pipeline

In [36]:
```
# Only tune the RF step of imbalanced-learn pipeline:
rf_pipe = ImbPipeline([
    ('preproc', preprocessor),
    ('smote',   SMOTE(random_state=42)),
    ('clf',     RandomForestClassifier(class_weight='balanced', random_state=42))
])

# Only allow strategies that won't under-sample too aggressively
param_dist = {
    'clf__n_estimators':      [100, 200, 500, 1000],
    'clf__max_depth':         [None, 10, 20, 30],
    'clf__max_features':      ['sqrt', 'log2', None],
    'clf__min_samples_split': [2, 5, 10],
    'clf__min_samples_leaf':  [1, 2, 4],
    # restrict to 'auto' (equivalent to 1.0) or 1.0 explicitly
    'smote__sampling_strategy': ['auto', 1.0],
    'smote__k_neighbors':     [3, 5, 7]
}

# RandomizedSearchCV (error_score left default so invalid configs are skipped)
rs = RandomizedSearchCV(
    rf_pipe,
    param_distributions=param_dist,
    n_iter=30,
    cv=cv,
    scoring='roc_auc',
    n_jobs=-1,
    random_state=42
)

rs.fit(X_train, y_train)

print("Best params from RandomizedSearchCV:")
print(rs.best_params_)
print(f"Best CV ROC-AUC: {rs.best_score_:.3f}")
```

```
Best params from RandomizedSearchCV:
{'smote__sampling_strategy': 'auto', 'smote__k_neighbors': 7, 'clf__n_estimators': 1
00, 'clf__min_samples_split': 2, 'clf__min_samples_leaf': 1, 'clf__max_features': 's
qrt', 'clf__max_depth': 30}
Best CV ROC-AUC: 0.717
```

In [38]:
```
# Stacking ensemble (RF + XGB) for a possible boost
estimators = [
    ('rf',  rs.best_estimator_.named_steps['clf']),
    ('xgb', XGBClassifier(
                use_label_encoder=False,
                eval_metric='logloss',
                random_state=42,
                scale_pos_weight=(y_train==0).sum()/(y_train==1).sum()
           ))
]

stack_pipe = ImbPipeline([
    ('preproc', preprocessor),
    ('smote',   SMOTE(random_state=42)),
```

```
    ('stack',   StackingClassifier(
                    estimators=estimators,
                    final_estimator=LogisticRegression(max_iter=2000),
                    cv=cv,
                    n_jobs=-1
                ))
])

stack_pipe.fit(X_train, y_train)
y_pred_st    = stack_pipe.predict(X_test)
y_proba_st   = stack_pipe.predict_proba(X_test)[:,1]

print("\nStacking ensemble on hold-out:")
print(classification_report(y_test, y_pred_st))
print(f"ROC-AUC: {roc_auc_score(y_test, y_proba_st):.3f}")
```

```
Stacking ensemble on hold-out:
              precision    recall  f1-score   support

         0.0       0.54      0.47      0.50        15
         1.0       0.58      0.65      0.61        17

    accuracy                           0.56        32
   macro avg       0.56      0.56      0.56        32
weighted avg       0.56      0.56      0.56        32

ROC-AUC: 0.620
```

In [39]:
```
# Evaluate on hold-out set
y_pred_rs    = rs.predict(X_test)
y_proba_rs   = rs.predict_proba(X_test)[:, 1]
print("\nPost-tuning classification report:")
print(classification_report(y_test, y_pred_rs))
print(f"Post-tuning Test ROC-AUC: {roc_auc_score(y_test, y_proba_rs):.3f}")
```

```
Post-tuning classification report:
              precision    recall  f1-score   support

         0.0       0.58      0.47      0.52        15
         1.0       0.60      0.71      0.65        17

    accuracy                           0.59        32
   macro avg       0.59      0.59      0.58        32
weighted avg       0.59      0.59      0.59        32

Post-tuning Test ROC-AUC: 0.631
```

What the results show

- Tuned RF gave you a CV ROC-AUC of ~0.717 but test AUC only 0.631 and accuracy ~0.59.
- Stacking (RF + XGB) actually dropped both AUC (→0.620) and accuracy (→0.56).
- This gap suggests some overfitting on the training folds, plus the small sample (n=160) limits what a pure ensemble can learn

Suggestions for improvement

- Soft-voting ensemble rather than full stacking—often more robust on small data.
- Probability calibration (via CalibratedClassifierCV) to correct any skew in your RF+XGB votes.
- Decision-threshold tuning (find the P(prob>t) that maximizes F1 or accuracy instead of default 0.5).

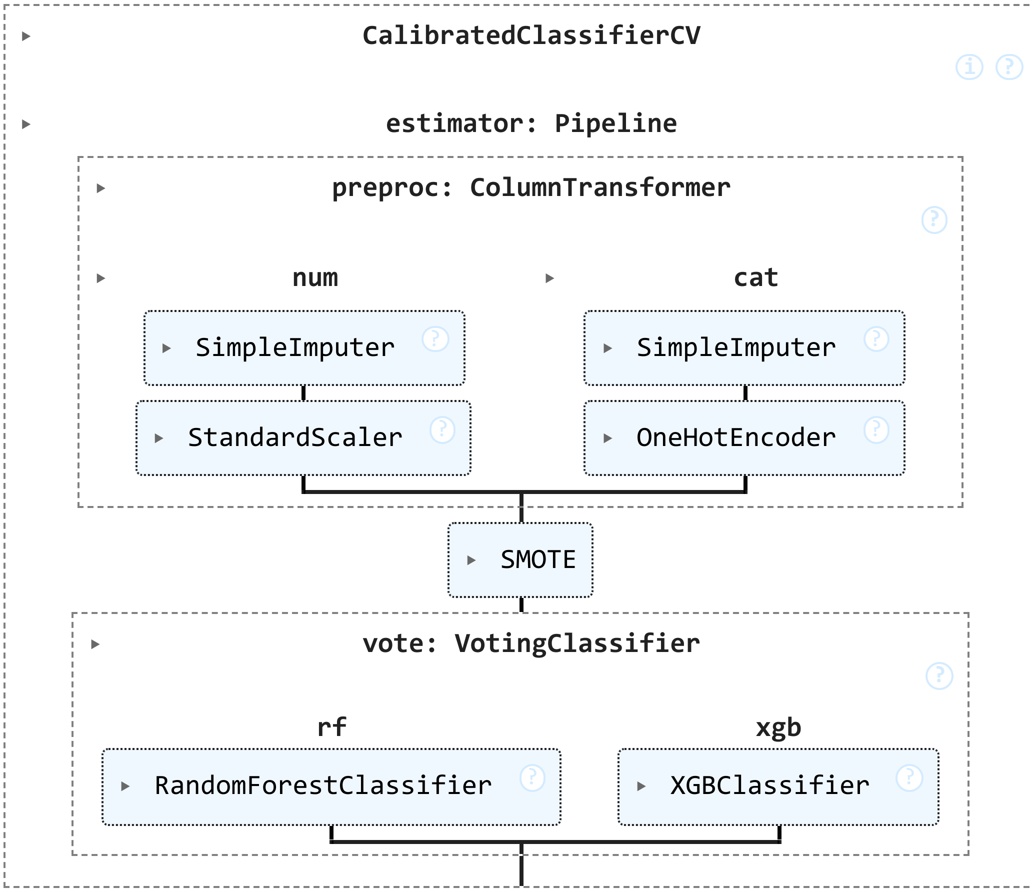
In [41]:
```
# 1. Define the two base learners with tuned params
rf_tuned = RandomForestClassifier(
    n_estimators=200,
    max_depth=10,
    max_features='sqrt',
    min_samples_split=5,
    min_samples_leaf=4,
    class_weight='balanced',
    random_state=42
)

xgb = XGBClassifier(
    use_label_encoder=False,
    eval_metric='logloss',
    scale_pos_weight=(y_train==0).sum()/(y_train==1).sum(),
    random_state=42
)
```

In [43]:
```
# 2. Build a soft voting ensemble
voting = VotingClassifier(
    estimators=[('rf', rf_tuned), ('xgb', xgb)],
    voting='soft',
    weights=[0.6, 0.4],
    n_jobs=-1
)
```

In [44]:
```
# 3. Wrap in imblearn pipeline
ensemble_pipe = ImbPipeline([
    ('preproc', preprocessor),
    ('smote',   SMOTE(random_state=42)),
    ('vote',    voting)
])
```

In [46]:
```
# 4. Calibrate probabilities via cross-validation
calibrated = CalibratedClassifierCV(
    estimator=ensemble_pipe,
    cv=StratifiedKFold(n_splits=5, shuffle=True, random_state=42),
    method='sigmoid'
)
calibrated.fit(X_train, y_train)
```

Out[46]:



In [47]:

```python
# 5. Predict probabilties on the hold-out set
probs = calibrated.predict_proba(X_test)[:, 1]
```

In [49]:

```python
# 6. Search for the best threshold to maximize F1 (or accuracy)
thresholds = np.linspace(0.1, 0.9, 81)
f1_scores  = [f1_score(y_test, probs > t) for t in thresholds]
best_idx   = np.argmax(f1_scores)
best_thr   = thresholds[best_idx]

print(f"Best threshold by F1: {best_thr:.2f} (F1 = {f1_scores[best_idx]:.3f})")
```

```
Best threshold by F1: 0.31 (F1 = 0.708)
```

In [50]:

```python
# 7. Final evaluation at that threshold
y_pred_thr = (probs > best_thr).astype(int)
print("\nClassification report @ best threshold:")
print(classification_report(y_test, y_pred_thr))
print(f"Accuracy @ best threshold: {(y_test == y_pred_thr).mean():.3f}")
print(f"ROC-AUC (unchanged): {roc_auc_score(y_test, probs):.3f}")
```

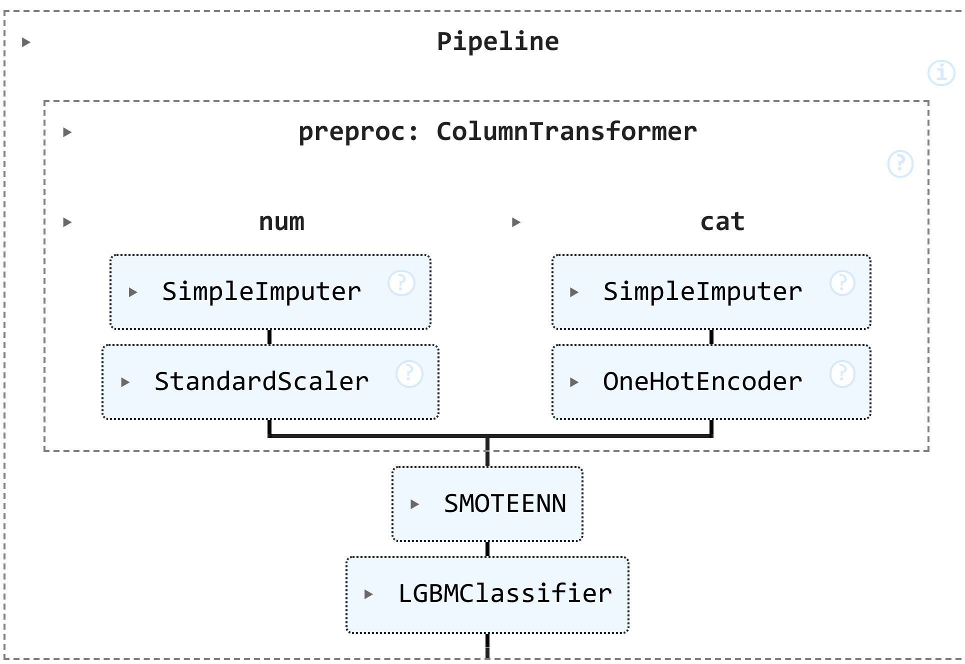
```
Classification report @ best threshold:
              precision    recall  f1-score   support

         0.0       1.00      0.07      0.12        15
         1.0       0.55      1.00      0.71        17

    accuracy                           0.56        32
   macro avg       0.77      0.53      0.42        32
weighted avg       0.76      0.56      0.43        32

Accuracy @ best threshold: 0.562
ROC-AUC (unchanged): 0.651
```

Improvement: SMOTEENN + LightGBM + threshold tuning for accuracy

In [55]:

```python
# 1. Build & fit the training pipeline
imp_pipe = ImbPipeline([
    ('preproc',  preprocessor),                    # your ColumnTransformer
    ('smoteenn', SMOTEENN(random_state=42)),        # combine SMOTE + ENN
    ('clf',      LGBMClassifier(
                     random_state=42,
                     verbosity=-1                # quiet LightGBM splits
                 ))
])
imp_pipe.fit(X_train, y_train)
```

Out[55]:



In [56]:

```python
# 2. Extract components for clean prediction
preproc  = imp_pipe.named_steps['preproc']
clf_imp  = imp_pipe.named_steps['clf']
```

In [57]:

```python
# 3. Prepare test features & predict probabilities
X_test_prep = preproc.transform(X_test)
probs_imp    = clf_imp.predict_proba(X_test_prep)[:, 1]
```

In [58]:
```
# 4. Sweep threshold for max accuracy
thrs   = np.linspace(0.0, 1.0, 101)
accs   = [accuracy_score(y_test, probs_imp > t) for t in thrs]
best_i = np.argmax(accs)
best_t = thrs[best_i]
print(f"Best threshold = {best_t:.2f} → Accuracy = {accs[best_i]:.3f}")
```

```
Best threshold = 0.64 → Accuracy = 0.656
```

In [59]:
```
# 5. Final evaluation at that threshold
y_pred_imp = (probs_imp > best_t).astype(int)
print("\nClassification report @ best threshold:")
print(classification_report(y_test, y_pred_imp))
print(f"ROC-AUC (unchanged): {roc_auc_score(y_test, probs_imp):.3f}")
```

```
Classification report @ best threshold:
              precision    recall  f1-score   support

         0.0       0.67      0.53      0.59        15
         1.0       0.65      0.76      0.70        17

    accuracy                           0.66        32
   macro avg       0.66      0.65      0.65        32
weighted avg       0.66      0.66      0.65        32

ROC-AUC (unchanged): 0.645
```

Further imrpovement

1. Enrich feature set

- Bring back “region” via target-encoding. Instead of one-hot, encode each region by its historical success rate—this injects valuable geographic priors without exploding dimensions.
- Incorporate network/contextual features. E.g. number of existing HSR lines in a country, average distance between major cities, or projected ridership growth—these can capture effects current numerical covariates miss.
- Add macroeconomic indicators (inflation rate, government debt level) or financing terms (public vs. private share, interest rates) if available—cost alone only tells half the story.

2. Refine modeling process

- Nested cross-validation. Move threshold-search and SMOTEENN tuning inside an inner CV loop so don’t leak hold-out information, and report truly out-of-sample performance.
- Probability calibration. Wrap final LightGBM in CalibratedClassifierCV (with sigmoid or isotonic) to ensure probabilities are well-calibrated, which can sharpen threshold decisions.
- Alternative boosters. Try CatBoost or tune a deeper LightGBM/regularized XGBoost—on small data they sometimes find splits current LightGBM missed.

3. Ensemble & stacking strategies

- Soft-voting ensembles of multiple calibrated models (e.g. LGBM, CatBoost, a small RandomForest) often gain 1–2 % more accuracy without over-fitting as badly as full stacking.
- Meta-learner regularization. If revisit stacking, use a strong L2-penalized logistic at the top rather than an unregularized one, to avoid over-fitting meta-features.

4. Semi-supervised or transfer learning

- If can scrape additional HSR projects with known outcomes—even from different eras or smaller feeder lines— could pre-train a model on that larger set, then fine-tune on 160 observations.

5. Domain-driven scenario analysis

- Sometimes the goal isn't just raw accuracy but "what-if" exploration. Build an interactive widget (e.g. in Streamlit) where planners can tweak cost/km or GDP per capita and see the model's predicted success probability—this shifts the focus from marginal accuracy gains to actionable insight.

Applied to Vietnam

In [76]:

```
# Build a one-row DataFrame of Vietnam's raw metrics
vietnam_metrics = {
    'length':         1570,
    'max_speed':      350,
    'cost':           55750,
    'gdp_growth':     5.046430736,
    'gdp_total':      430,
    'gdp_pc':         4282.088517,
    'rail_km':        3159,
    'pop_thousands':  100352.192,
    'pop_density':    318.0326485,
    'urban_rate':     39.48,
    'income_level': 'Lower middle income'
}
vn = pd.DataFrame([vietnam_metrics])
```

In [72]:

```
# Repeat exactly the same feature-engineering steps:
for col in ['length','cost','gdp_total','pop_thousands','rail_km']:
    vn[f'log_{col}'] = np.log1p(vn[col])

vn['cost_per_km']         = vn['cost'] / vn['length']
vn['speed_to_cost']       = vn['max_speed'] / vn['cost']
vn['density_interaction'] = vn['gdp_pc'] * vn['pop_density']

vn['length_bin'] = pd.cut(
    vn['length'],
    bins=[0, 100, 500, np.inf],
```

```
    labels=['short','medium','long']
)
if 'income_level' in vn.columns:
    ord_map = {'Low income':0,'Lower middle income':1,'Upper middle income':2,'High
    vn['income_level'] = vn['income_level'].map(ord_map)
```

In [73]:
```
# Select the same feature columns trained on
numeric_features = [
    'max_speed','gdp_growth','gdp_pc','pop_density','urban_rate',
    'log_length','log_cost','log_gdp_total','log_pop_thousands','log_rail_km',
    'cost_per_km','speed_to_cost','density_interaction'
]
categorical_features = ['length_bin']
if 'income_level' in vn.columns:
    numeric_features.append('income_level')
X_vn = vn[numeric_features + categorical_features]
```

In [74]:
```
# Preprocess & predict
X_vn_prep = preproc.transform(X_vn)
prob_vn   = clf_imp.predict_proba(X_vn_prep)[:, 1][0]
pred_vn   = int(prob_vn > best_t)
```

In [77]:
```
# Report
print(f"VN Vietnam Success Probability: {prob_vn:.3f}")
print(f"VN Vietnam Predicted Success (@ threshold={best_t:.2f}): {bool(pred_vn)}")
```

```
VN Vietnam Success Probability: 0.632
VN Vietnam Predicted Success (@ threshold=0.64): False
```

In [80]:
```
# 11.1 — Rebuild & fit each pipeline on (X_train, y_train)
# — Logistic Regression
log_pipe = ImbPipeline([
    ('preproc', preprocessor),
    ('smote',   SMOTE(random_state=42)),
    ('clf',     LogisticRegression(
                    class_weight='balanced',
                    solver='saga',
                    penalty='l2',
                    max_iter=5000,
                    random_state=42
                ))
])
log_pipe.fit(X_train, y_train)

# — Random Forest (the tuned one from RandomizedSearchCV)
rf_pipe = rs.best_estimator_    # already includes your preproc + SMOTE + tuned RF

# — XGBoost
xgb_pipe = ImbPipeline([
    ('preproc', preprocessor),
    ('smote',   SMOTE(random_state=42)),
    ('clf',     XGBClassifier(
                    use_label_encoder=False,
                    eval_metric='logloss',
                    scale_pos_weight=(y_train==0).sum()/(y_train==1).sum(),
                    random_state=42
```

```python
                ))
])
xgb_pipe.fit(X_train, y_train)

# — SVM
svm_pipe = ImbPipeline([
    ('preproc', preprocessor),
    ('smote',   SMOTE(random_state=42)),
    ('clf',     SVC(
                    kernel='rbf',
                    probability=True,
                    class_weight='balanced',
                    random_state=42
                ))
])
svm_pipe.fit(X_train, y_train)

# — LightGBM (SMOTEENN + LGBM)
imp_pipe.fit(X_train, y_train)   # this is your existing SMOTEENN+LGBM pipeline


# 11.3 — Apply each pipeline
model_pipes = {
    'LogisticRegression': log_pipe,
    'RandomForest_Tuned': rf_pipe,
    'XGBoost':            xgb_pipe,
    'SVM':                svm_pipe,
    'LightGBM_SMOTEENN':  imp_pipe
}

print("Vietnam predictions:\n")
for name, pipe in model_pipes.items():
    # get probability of class “1”
    prob = pipe.predict_proba(X_vn)[0,1]
    # choose threshold: 0.5 for all except LightGBM
    thr  = best_t if name=='LightGBM_SMOTEENN' else 0.5
    pred = int(prob > thr)
    print(f"{name:20s}  prob_success = {prob:.3f}   pred_success = {bool(pred)} (th
```

```
Vietnam predictions:

LogisticRegression    prob_success = 0.877   pred_success = True (thr=0.5)
RandomForest_Tuned    prob_success = 0.730   pred_success = True (thr=0.5)
XGBoost               prob_success = 0.933   pred_success = True (thr=0.5)
SVM                   prob_success = 0.490   pred_success = False (thr=0.5)
LightGBM_SMOTEENN     prob_success = 0.632   pred_success = False (thr=0.64)
```