03_modelling

July 21, 2025

1 London Modelling

[1]:
```
import pandas as pd
import geopandas as gpd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split, RandomizedSearchCV,␣
 ↪GridSearchCV
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
from sklearn.ensemble import RandomForestRegressor
import seaborn as sns
from scipy.stats import gaussian_kde
import contextily as ctx
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.colors import ListedColormap
import matplotlib.font_manager as fm
from matplotlib.patches import Patch
%config InlineBackend.figure_format = 'retina'
from mgwr.gwr import GWR
from mgwr.sel_bw import Sel_BW
import matplotlib.font_manager as fm
import xgboost as xgb
import shap
from scipy.spatial import cKDTree

# Load Cambria font
cambria_path = "/Library/Fonts/Microsoft/Cambria.ttf"  # Adjust path if needed
cambria_prop = fm.FontProperties(fname=cambria_path)

# Load modelling functions
import sys
sys.path.append('../../scripts')  # go up one folder, into scripts/

from modelling import (
    prepare_data,
    run_modelling_pipeline,
```

```
    map_gwr_coefficients,
    calculate_t_values,
    categorise_significance,
    plot_significance,
    geographically_weighted_rf
)
```

1.1 1. Import Data

[2]:
```
# Import data
london_airbnb = gpd.read_file('../../data/london/airbnb_per_london_glx.gpkg')
london_tourism_fs = gpd.read_file('../../data/london/london_glx_tourism_pois.
 ↪gpkg')
```

[3]:
```
# Merge the two datasets on glx_id keeping only one geometry column
london_airbnb_tourism_fs = london_tourism_fs.merge(london_airbnb, on='glx_id',␣
 ↪how='left')

# Drop geometry_x and rename geometry_y to geometry
london_airbnb_tourism_fs.drop(columns=['geometry_x'], inplace=True)
london_airbnb_tourism_fs.rename(columns={'geometry_y': 'geometry'},␣
 ↪inplace=True)
```

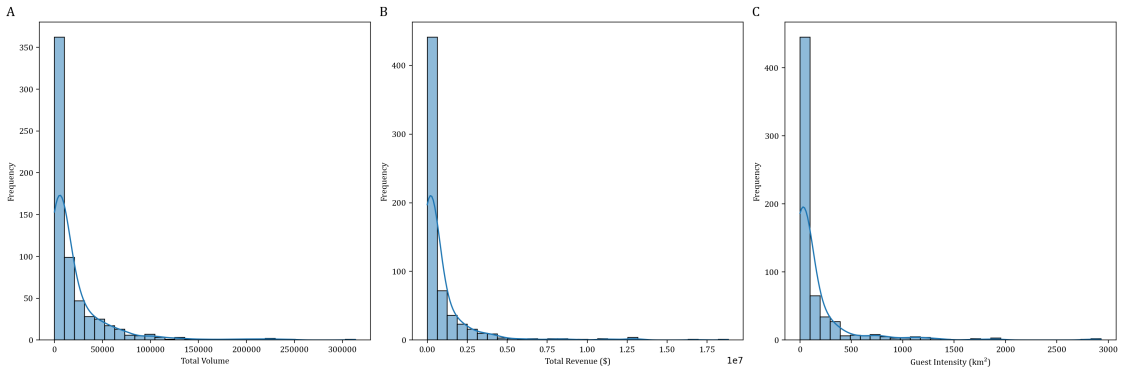
[4]:
```
# Create total_pois_log column
london_airbnb_tourism_fs['total_pois_log'] = np.
 ↪log1p(london_airbnb_tourism_fs['total_pois'])
```

[5]:
```
# Import London GLX shapefile
london_glx = gpd.read_file("../../data/london/glondon_seamless.gpkg")
london_glx = london_glx.to_crs(epsg=27700)
```

1.2 2. Data Exploration

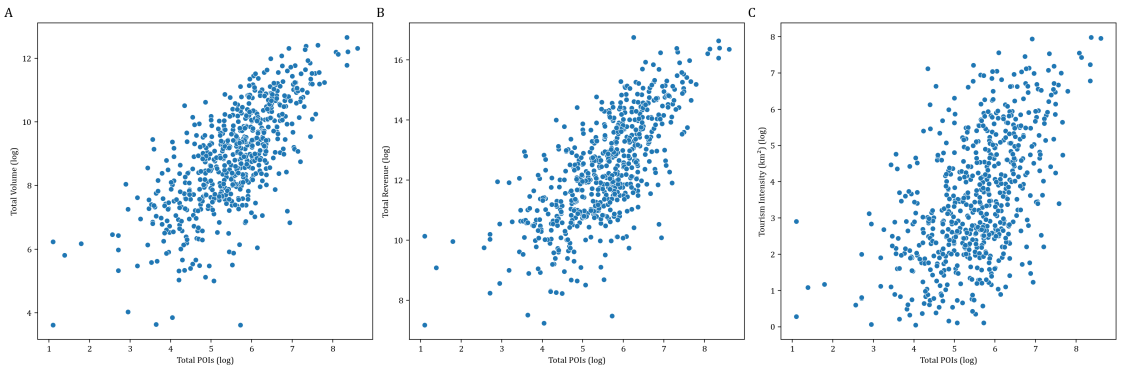
[6]:
```
# Create plot
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Plot 1: Total Volume
sns.histplot(london_airbnb_tourism_fs['total_volume'], bins=30, kde=True,␣
 ↪ax=axes[0])
axes[0].set_xlabel('Total Volume', fontproperties=cambria_prop)
axes[0].set_ylabel('Frequency', fontproperties=cambria_prop)

# Plot 2: Total Revenue
sns.histplot(london_airbnb_tourism_fs['total_revenue'], bins=30, kde=True,␣
 ↪ax=axes[1])
axes[1].set_xlabel('Total Revenue ($)', fontproperties=cambria_prop)
axes[1].set_ylabel('Frequency', fontproperties=cambria_prop)
```

```
# Plot 3: Tourism Intensity
sns.histplot(london_airbnb_tourism_fs['total_tourism_intensity_km2'], bins=30,␣
 ↪kde=True, ax=axes[2])
axes[2].set_xlabel('Guest Intensity (km²)', fontproperties=cambria_prop)
axes[2].set_ylabel('Frequency', fontproperties=cambria_prop)

# Add A-C panel tags
panel_labels = ['A', 'B', 'C']
for i, ax in enumerate(axes):
    ax.text(
        -0.1, 1.05,
        panel_labels[i],
        transform=ax.transAxes,
        fontsize=16,
        fontproperties=cambria_prop,
        fontweight='bold',
        va='top',
        ha='left'
    ),
    ax.grid(False)
    for label in ax.get_xticklabels() + ax.get_yticklabels():
        label.set_fontproperties(cambria_prop)

plt.tight_layout()
plt.show()
```



[7]:
```
# Create three scatter plots showing total_pois vs, total_volume,␣
 ↪total_revenue, and total_tourism_intensity_km2
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

sns.scatterplot(data=london_airbnb_tourism_fs, x='total_pois_log',␣
 ↪y='total_volume_log', ax=axes[0])
```

```
axes[0].set_xlabel('Total POIs (log)', fontproperties=cambria_prop)
axes[0].set_ylabel('Total Volume (log)', fontproperties=cambria_prop)

sns.scatterplot(data=london_airbnb_tourism_fs, x='total_pois_log',␣
 ↪y='total_revenue_log', ax=axes[1])
axes[1].set_xlabel('Total POIs (log)', fontproperties=cambria_prop)
axes[1].set_ylabel('Total Revenue (log)', fontproperties=cambria_prop)

sns.scatterplot(data=london_airbnb_tourism_fs, x='total_pois_log',␣
 ↪y='total_tourism_intensity_km2_log', ax=axes[2])
axes[2].set_xlabel('Total POIs (log)', fontproperties=cambria_prop)
axes[2].set_ylabel('Tourism Intensity (km²) (log)', fontproperties=cambria_prop)

# Add A-C panel tags
panel_labels = ['A', 'B', 'C']
for i, ax in enumerate(axes):
    ax.text(
        -0.1, 1.05,
        panel_labels[i],
        transform=ax.transAxes,
        fontsize=16,
        fontproperties=cambria_prop,
        fontweight='bold',
        va='top',
        ha='left'
    ),
    ax.grid(False)
    for label in ax.get_xticklabels() + ax.get_yticklabels():
        label.set_fontproperties(cambria_prop)

plt.tight_layout()
plt.show()
```



[8]:
```
# Define the non-logged features
features = [
    'air_travel_density',
    'arts_other_density',
    'bars_nightlife_density',
    'cafes_bakeries_density',
    'cultural_institution_density',
    'dining_other_density',
    'entertainment_nightlife_density',
    'event_space_density',
    'fast_food_density',
    'food_drink_production_density',
    'full_service_restaurant_density',
    'high_end_dining_density',
    'historic_landmark_density',
    'leisure_activity_centre_density',
    'local_public_transit_density',
    'lodging_density',
    'major_attraction_density',
    'major_transit_hub_density',
    'mobility_services_density',
    'music_cinema_venue_density',
    'nature_trail_density',
    'outdoors_other_density',
    'park_garden_density',
    'public_art_density',
    'public_plaza_square_density',
    'retail_books_hobbies_density',
    'retail_fashion_apparel_density',
    'retail_food_beverage_density',
    'retail_gifts_misc_density',
    'retail_health_beauty_density',
    'retail_home_electronics_density',
    'retail_luxury_density',
    'retail_markets_plazas_density',
    'retail_other_density',
    'rural_agricultural_density',
    'tourism_services_density',
    'transport_infrastructure_density',
    'travel_other_density',
    'waterfront_river_density'
    ]

# Define the logged features
features_logged = [
    'air_travel_density_log',
    'arts_other_density_log',
```

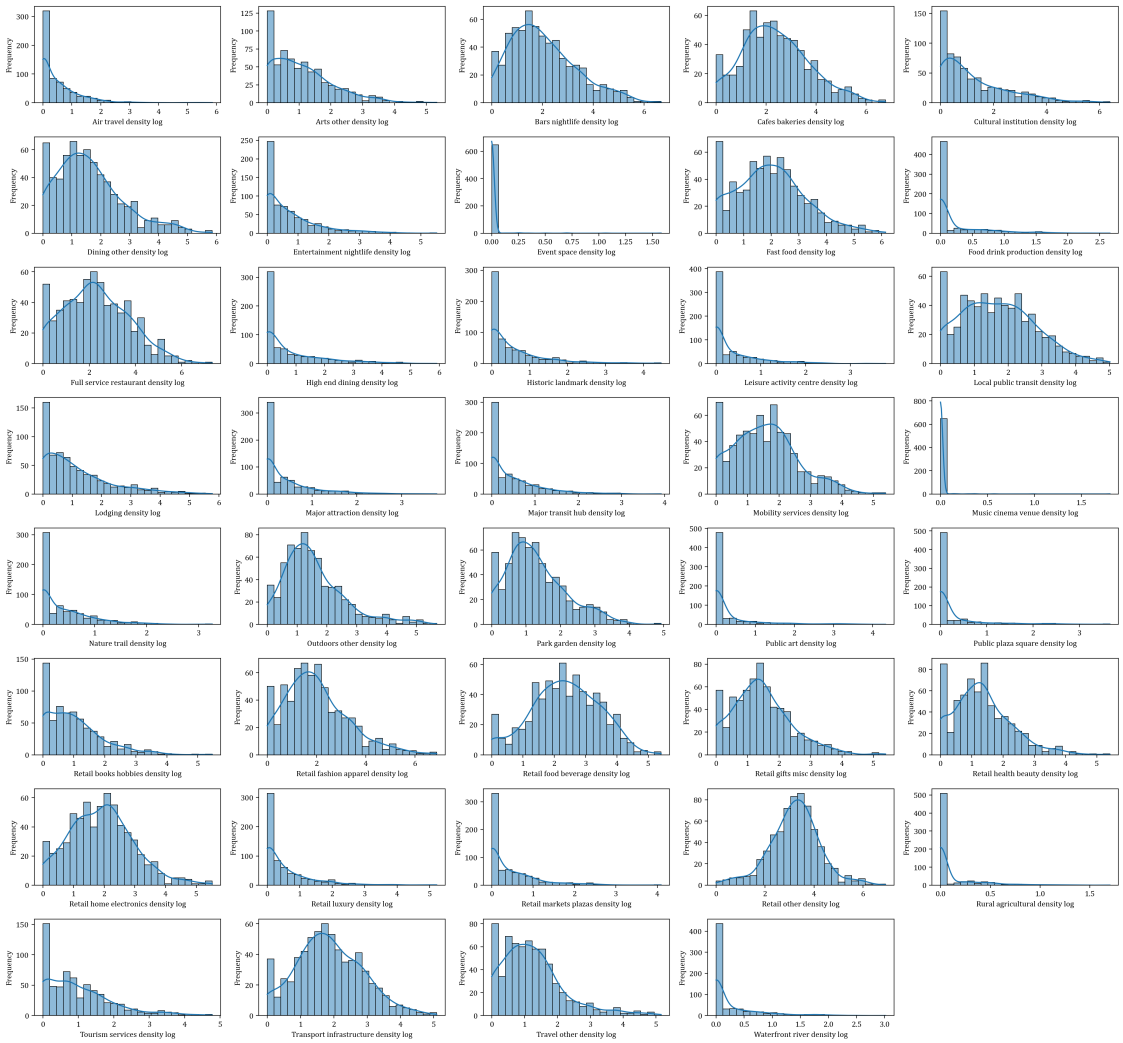
```
    'bars_nightlife_density_log',
    'cafes_bakeries_density_log',
    'cultural_institution_density_log',
    'dining_other_density_log',
    'entertainment_nightlife_density_log',
    'event_space_density_log',
    'fast_food_density_log',
    'food_drink_production_density_log',
    'full_service_restaurant_density_log',
    'high_end_dining_density_log',
    'historic_landmark_density_log',
    'leisure_activity_centre_density_log',
    'local_public_transit_density_log',
    'lodging_density_log',
    'major_attraction_density_log',
    'major_transit_hub_density_log',
    'mobility_services_density_log',
    'music_cinema_venue_density_log',
    'nature_trail_density_log',
    'outdoors_other_density_log',
    'park_garden_density_log',
    'public_art_density_log',
    'public_plaza_square_density_log',
    'retail_books_hobbies_density_log',
    'retail_fashion_apparel_density_log',
    'retail_food_beverage_density_log',
    'retail_gifts_misc_density_log',
    'retail_health_beauty_density_log',
    'retail_home_electronics_density_log',
    'retail_luxury_density_log',
    'retail_markets_plazas_density_log',
    'retail_other_density_log',
    'rural_agricultural_density_log',
    'tourism_services_density_log',
    'transport_infrastructure_density_log',
    'travel_other_density_log',
    'waterfront_river_density_log'
]
```

[9]:
```
# For all features, create histograms to check for normality and put into a␣
 ↪single figure and hide empty subplots
fig, axes = plt.subplots(14, 5, figsize=(20, 32))
axes = axes.flatten()
for i, feature in enumerate(features):
    sns.histplot(london_airbnb_tourism_fs[feature], bins=25, kde=True,␣
 ↪ax=axes[i])
```

```
    axes[i].set_xlabel(axes[i].get_xlabel().replace('_', ' ').
 ↪replace('density', 'Density (per km²)').capitalize(),␣
 ↪fontproperties=cambria_prop)
    axes[i].set_ylabel('Frequency', fontproperties=cambria_prop)
    axes[i].grid(False)
    for label in axes[i].get_xticklabels() + axes[i].get_yticklabels():
        label.set_fontproperties(cambria_prop)
# Hide any unused subplots
for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])
plt.tight_layout()
plt.show();
```

All right skewed, therefore, logged transformation needed.

[10]:
```
# For all features, create histograms to check for normality and put into a␣
 ↪single figure and hide empty subplots
fig, axes = plt.subplots(14, 5, figsize=(20, 32))
axes = axes.flatten()
for i, feature in enumerate(features_logged):
    sns.histplot(london_airbnb_tourism_fs[feature], bins=25, kde=True,␣
 ↪ax=axes[i])
    axes[i].set_xlabel(axes[i].get_xlabel().replace('_', ' ').
 ↪replace('density_log', 'Density (per km²) (log)').capitalize(),␣
 ↪fontproperties=cambria_prop)
    axes[i].set_ylabel('Frequency', fontproperties=cambria_prop)
    axes[i].grid(False)
    for label in axes[i].get_xticklabels() + axes[i].get_yticklabels():
        label.set_fontproperties(cambria_prop)
# Hide any unused subplots
for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])
plt.tight_layout()
plt.show();
```



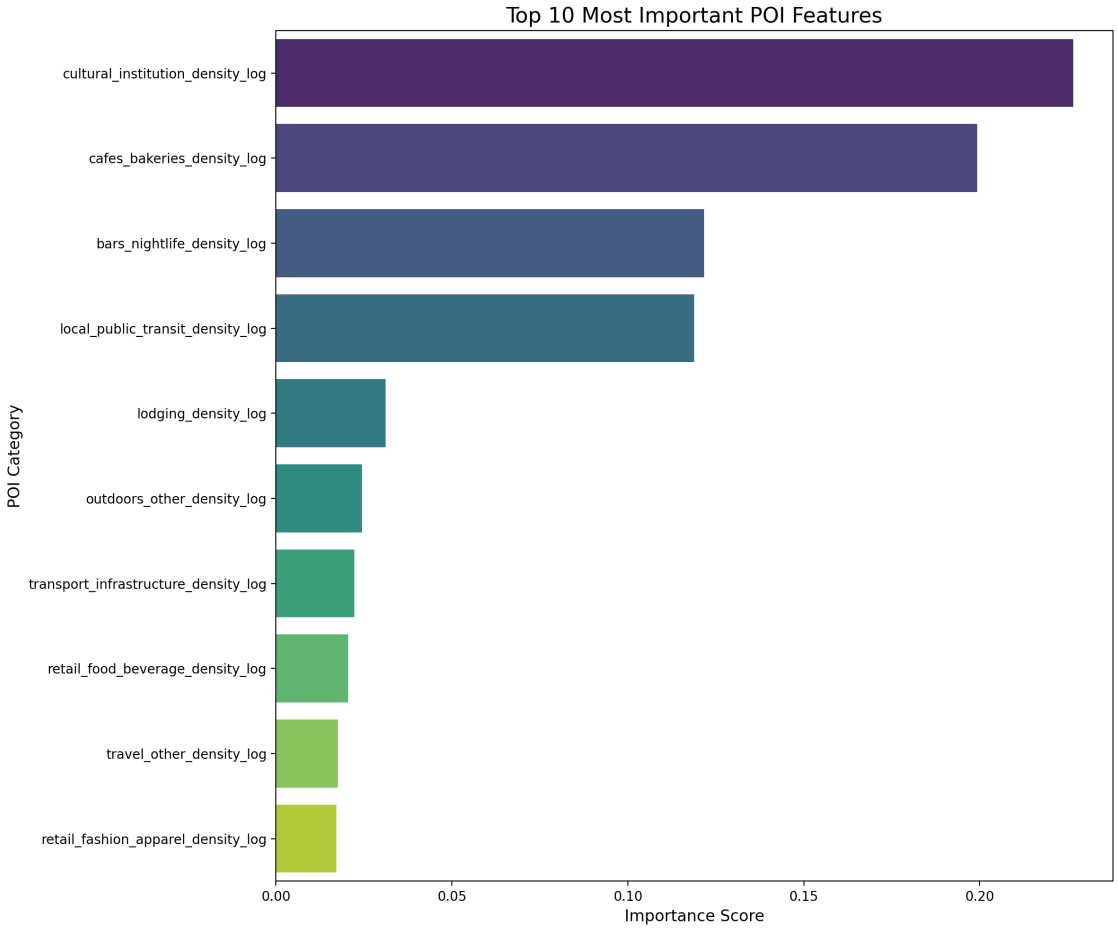
1.3 3. Random Forest Model

1.3.1 Total Revenue

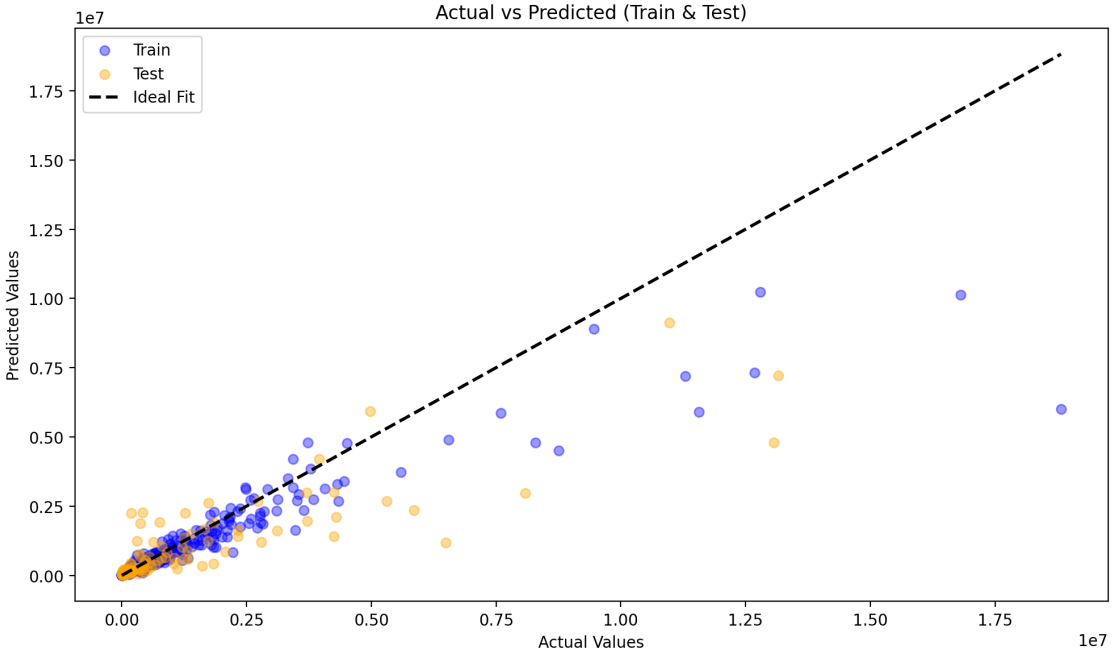
[11]:
```
print("=" * 50)
print("Analysing: Total Revenue ($)")
print("=" * 50)
# Prepare the data
X_rev, y_rev = prepare_data(london_airbnb_tourism_fs, features_logged,␣
 ↪'total_revenue_log')
# Run the modelling pipeline
revenue_model, revenue_importances = run_modelling_pipeline(X_rev, y_rev,␣
 ↪london_airbnb_tourism_fs['total_revenue'])
```

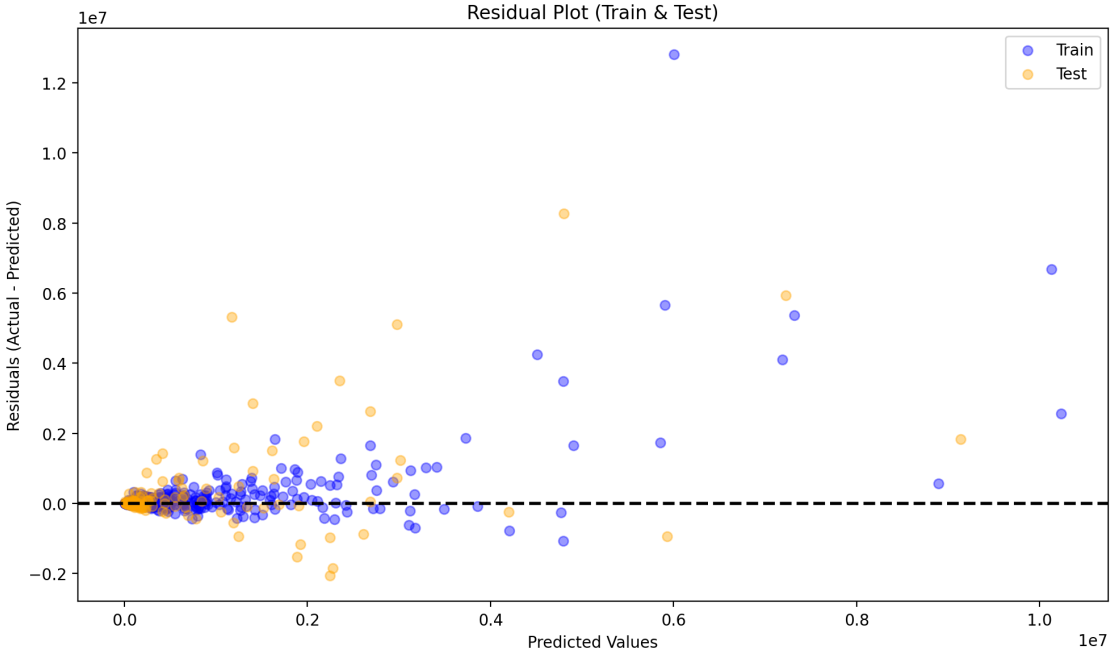
```
==================================================
Analysing: Total Revenue ($)
```

```
==================================================
Data prepared for target: 'total_revenue_log'
Number of features: 39
Number of rows after cleaning: 629

--- Training Baseline Model ---
Baseline R-squared (R²): 0.77
Baseline MAE (log): 0.678
Baseline RMSE (log): 0.888
Baseline MAE (original): 582,204.65
Baseline RMSE (original): 1,337,005.25

--- Starting Hyperparameter Tuning ---
Tuning complete.
Best parameters found: {'n_estimators': 100, 'min_samples_leaf': 2,
'max_features': 1.0, 'max_depth': 10}
Tuned R-squared (R²): 0.77
Tuned MAE (log): 0.675
Tuned RMSE (log): 0.882
Tuned MAE (original): 583,861.75
Tuned RMSE (original): 1,349,399.07

/Users/elisdavies/Documents/University_of_Liverpool/MSc Geographic Data
Science/ENVS492 - Dissertation/POI-and-STR-Guest-
Modelling/notebooks/london/../../scripts/modelling.py:136: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same
effect.

  sns.barplot(x=sorted_importances.values, y=sorted_importances.index,
palette="viridis")
```




```
--- Top 10 Most Important Features ---
cultural_institution_density_log        0.226536
cafes_bakeries_density_log              0.199258
bars_nightlife_density_log              0.121711
local_public_transit_density_log        0.118878
lodging_density_log                     0.031159
outdoors_other_density_log              0.024387
transport_infrastructure_density_log    0.022264
retail_food_beverage_density_log        0.020516
travel_other_density_log                0.017613
retail_fashion_apparel_density_log      0.017209
dtype: float64
```


Actual vs Predicted (Train & Test)
1e7
Train
Test
Ideal Fit
Predicted Values
Actual Values
1e7
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75

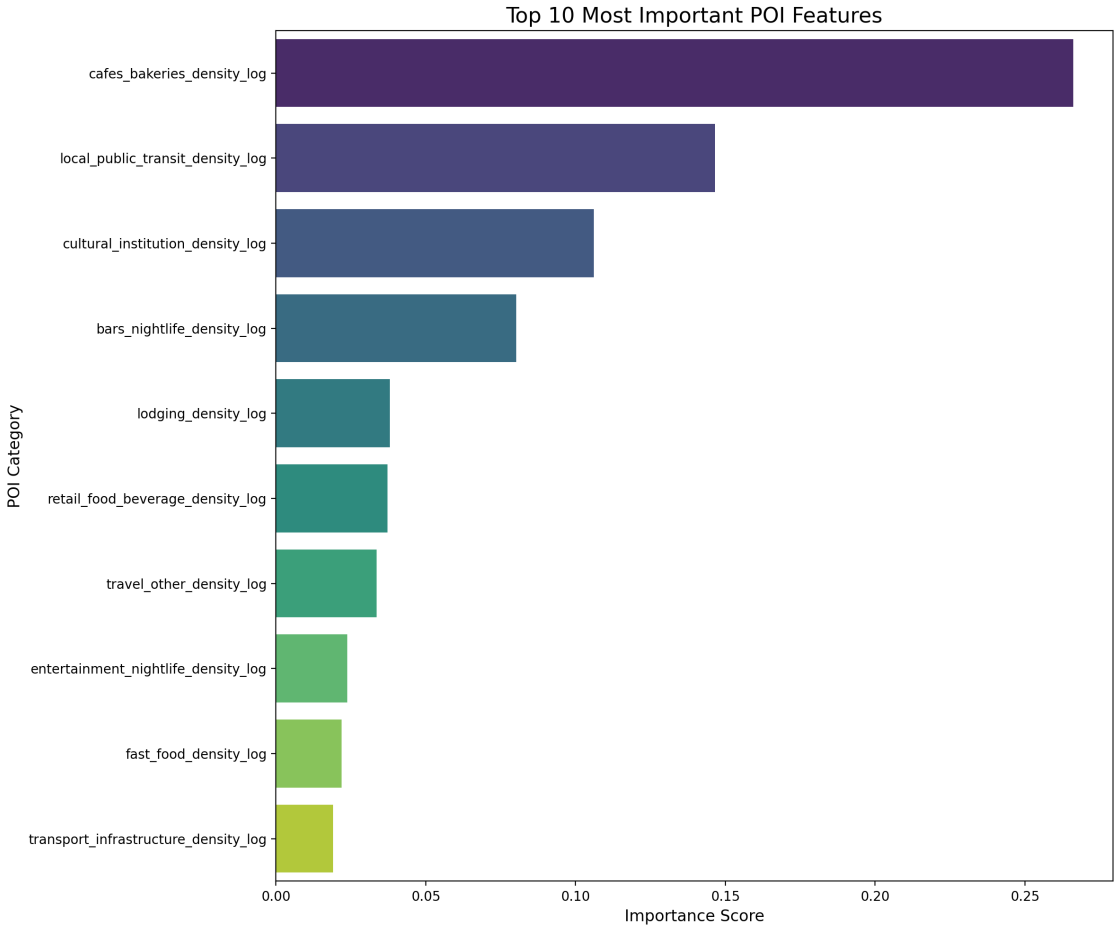

Residual Plot (Train & Test)
1e7
Train
Test
Residuals (Actual - Predicted)
Predicted Values
1e7
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2

1.3.2 Total Volume

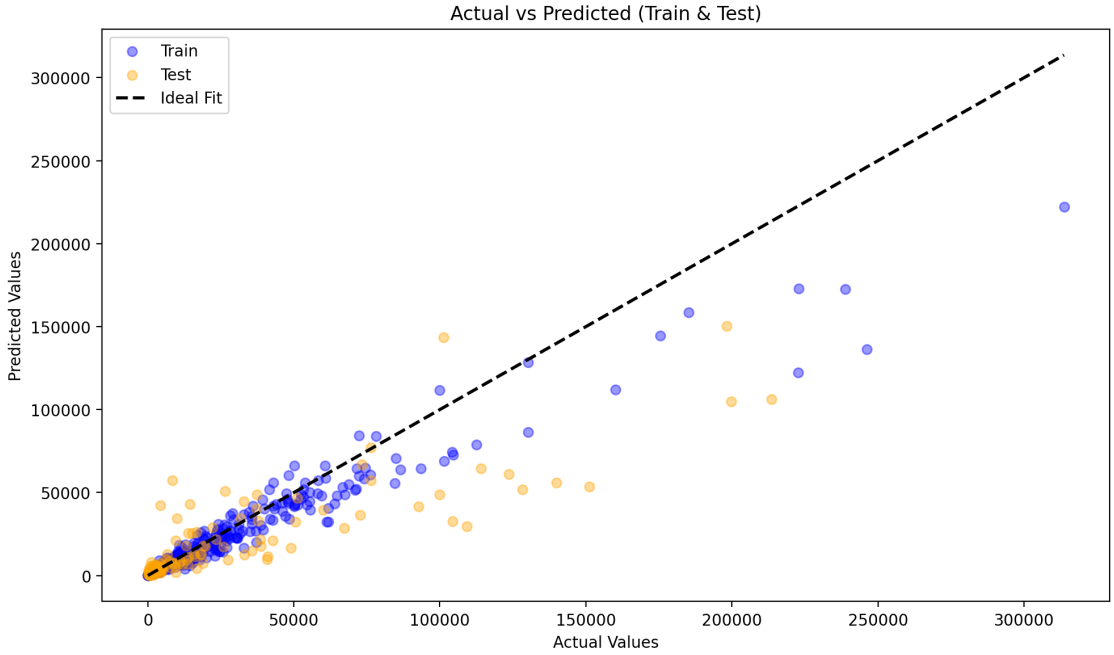
[12]:
```
print("=" * 50)
print("Analysing: Total Volume (number of guests)")
print("=" * 50)
# Prepare the data
X_rev, y_rev = prepare_data(london_airbnb_tourism_fs, features_logged,␣
 ↪'total_volume_log')
# Run the modelling pipeline
revenue_model, volume_importances = run_modelling_pipeline(X_rev, y_rev,␣
 ↪london_airbnb_tourism_fs['total_volume'])
```

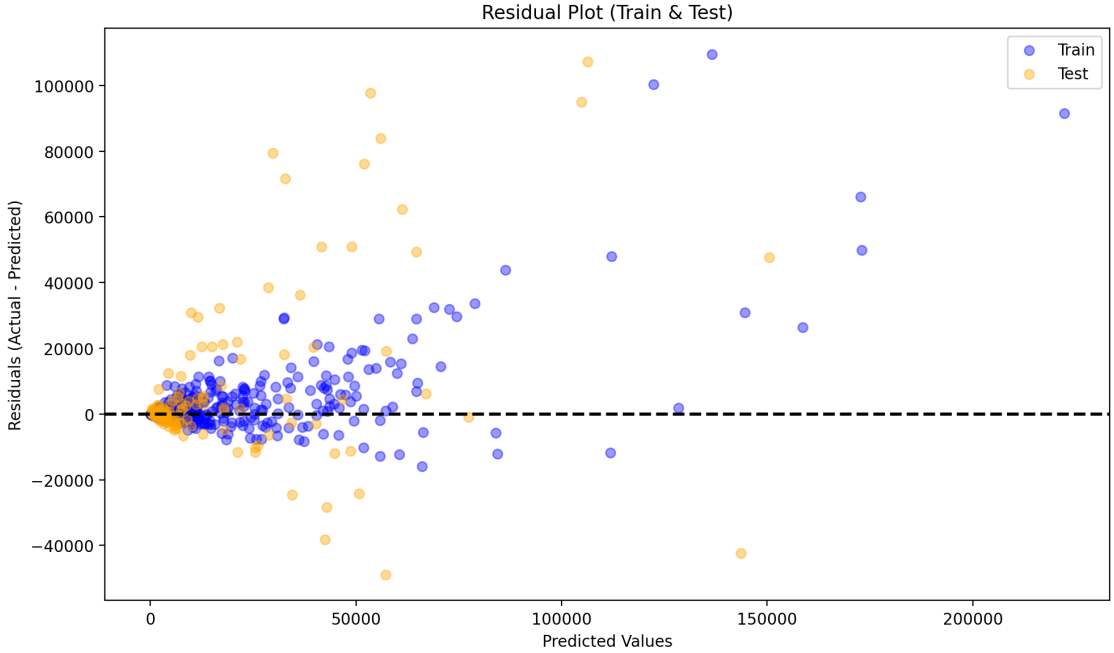
```
==================================================
Analysing: Total Volume (number of guests)
==================================================
Data prepared for target: 'total_volume_log'
Number of features: 39
Number of rows after cleaning: 629

--- Training Baseline Model ---
Baseline R-squared (R²): 0.75
Baseline MAE (log): 0.655
Baseline RMSE (log): 0.839
Baseline MAE (original): 13,305.60
Baseline RMSE (original): 25,705.06

--- Starting Hyperparameter Tuning ---
Tuning complete.
Best parameters found: {'n_estimators': 500, 'min_samples_leaf': 1,
'max_features': 1.0, 'max_depth': 10}
Tuned R-squared (R²): 0.75
Tuned MAE (log): 0.656
Tuned RMSE (log): 0.839
Tuned MAE (original): 13,641.04
Tuned RMSE (original): 26,244.34
```

```
/Users/elisdavies/Documents/University_of_Liverpool/MSc Geographic Data
Science/ENVS492 - Dissertation/POI-and-STR-Guest-
Modelling/notebooks/london/../../scripts/modelling.py:136: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same
effect.

  sns.barplot(x=sorted_importances.values, y=sorted_importances.index,
palette="viridis")
```




```
--- Top 10 Most Important Features ---
cafes_bakeries_density_log              0.266033
local_public_transit_density_log        0.146501
cultural_institution_density_log        0.106084
bars_nightlife_density_log              0.080225
lodging_density_log                     0.037952
retail_food_beverage_density_log        0.037165
travel_other_density_log                0.033572
entertainment_nightlife_density_log     0.023842
fast_food_density_log                   0.021855
transport_infrastructure_density_log    0.019012
dtype: float64
```


Actual vs Predicted (Train & Test)
Train
Test
Ideal Fit
Predicted Values
Actual Values
0
50000
100000
150000
200000
250000
300000

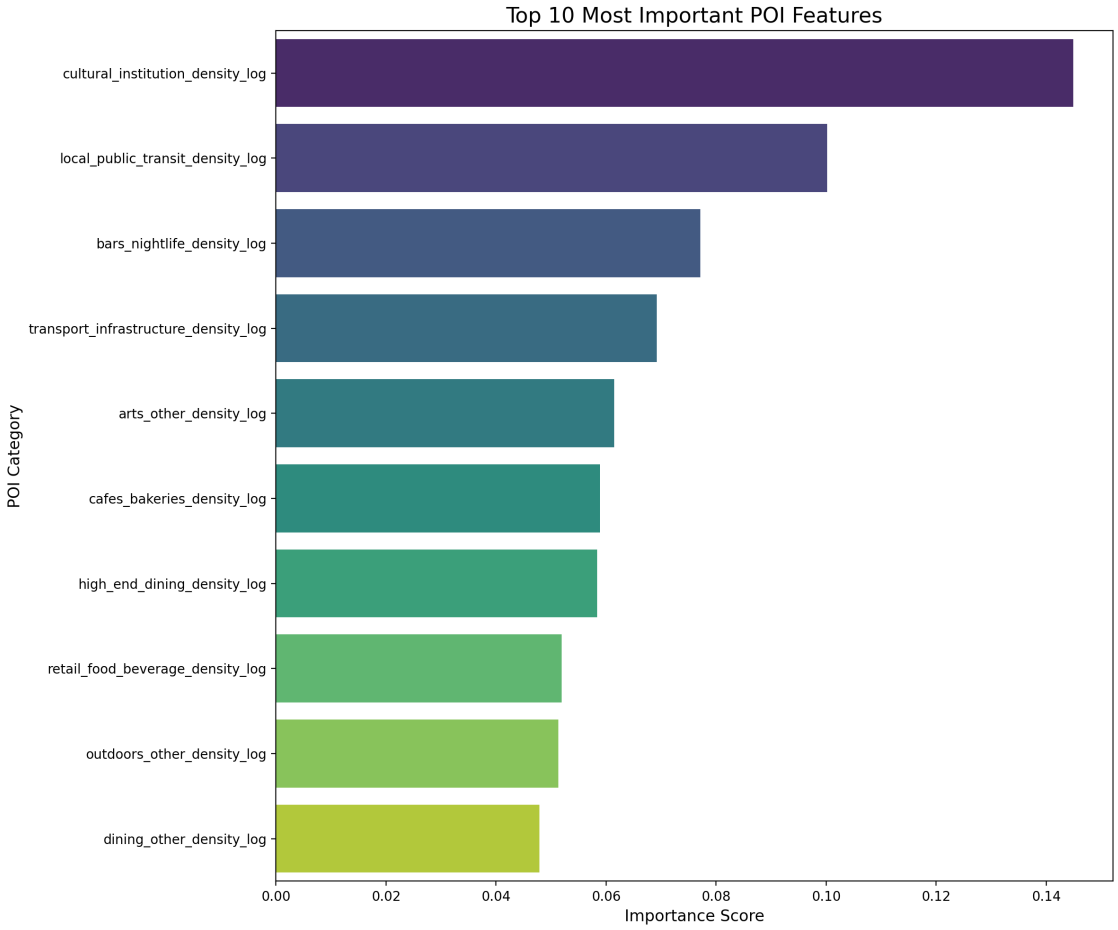

Residual Plot (Train & Test)
Train
Test
Residuals (Actual - Predicted)
Predicted Values
100000
80000
60000
40000
20000
0
−20000
−40000
0
50000
100000
150000
200000

1.3.3 Guest Density

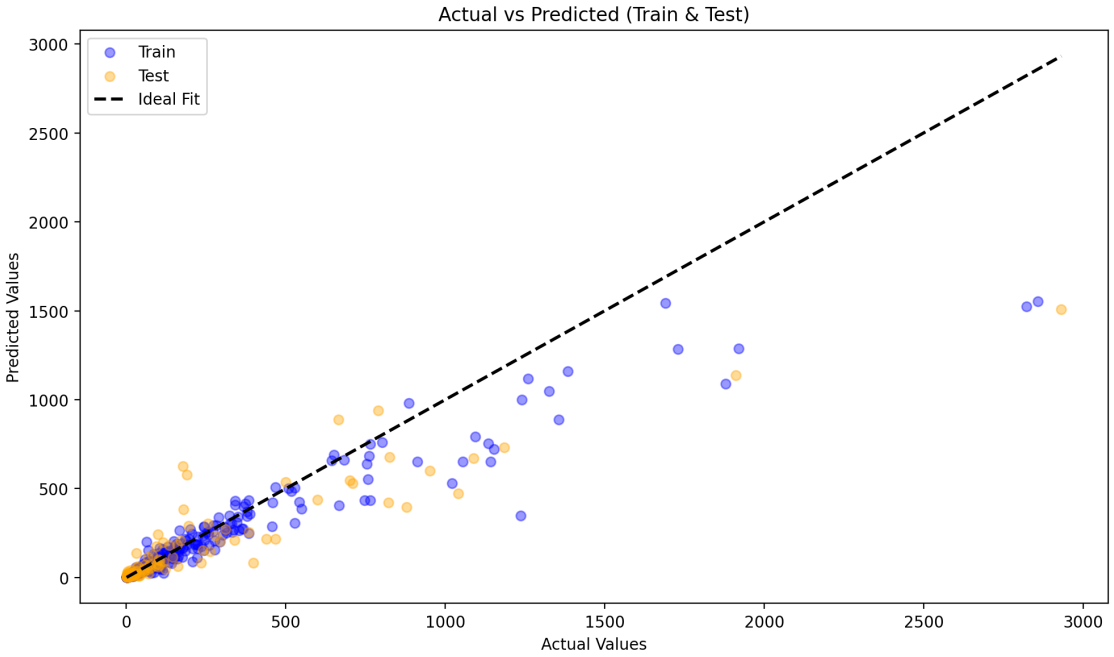
[13]:
```
print("=" * 50)
print("Analysing: Total Guest Density (Number of guestes per km2)")
print("=" * 50)
# Prepare the data
X_rev, y_rev = prepare_data(london_airbnb_tourism_fs, features_logged,␣
 ↪'total_tourism_intensity_km2_log')
# Run the modelling pipeline
revenue_model, density_importances = run_modelling_pipeline(X_rev, y_rev,␣
 ↪london_airbnb_tourism_fs['total_tourism_intensity_km2'])
```

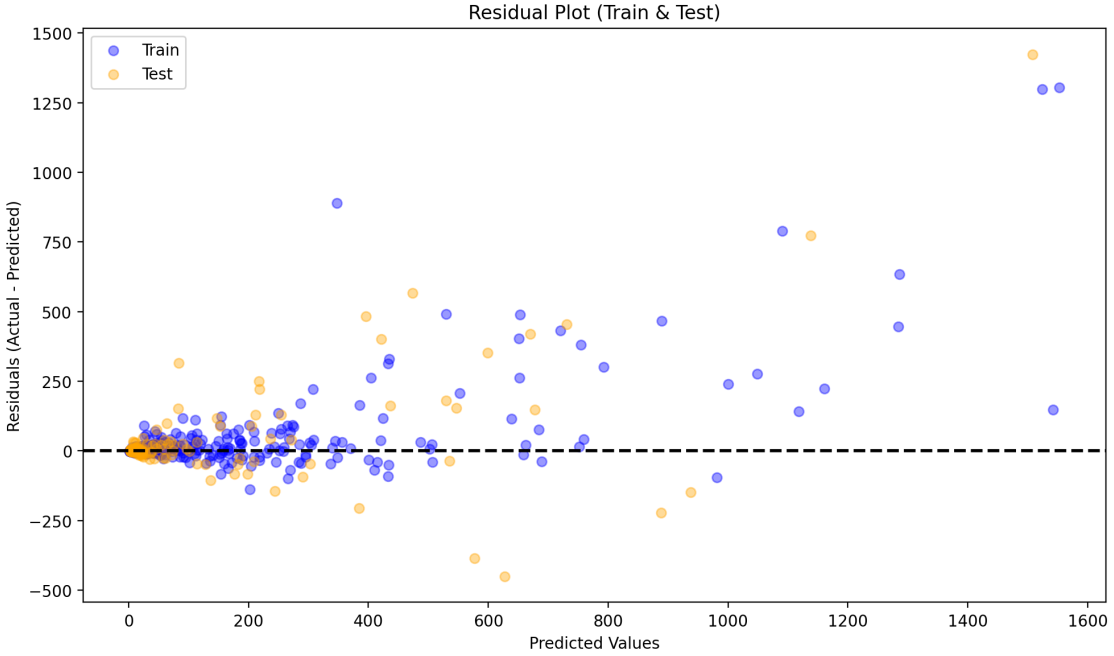
```
==================================================
Analysing: Total Guest Density (Number of guestes per km2)
==================================================
Data prepared for target: 'total_tourism_intensity_km2_log'
Number of features: 39
Number of rows after cleaning: 629

--- Training Baseline Model ---
Baseline R-squared (R²): 0.85
Baseline MAE (log): 0.526
Baseline RMSE (log): 0.701
Baseline MAE (original): 80.80
Baseline RMSE (original): 195.43

--- Starting Hyperparameter Tuning ---
Tuning complete.
Best parameters found: {'n_estimators': 300, 'min_samples_leaf': 2,
'max_features': 'sqrt', 'max_depth': None}
Tuned R-squared (R²): 0.85
Tuned MAE (log): 0.555
Tuned RMSE (log): 0.714
Tuned MAE (original): 82.16
Tuned RMSE (original): 195.51
```

```
/Users/elisdavies/Documents/University_of_Liverpool/MSc Geographic Data
Science/ENVS492 - Dissertation/POI-and-STR-Guest-
Modelling/notebooks/london/../../scripts/modelling.py:136: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in
v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same
effect.

  sns.barplot(x=sorted_importances.values, y=sorted_importances.index,
palette="viridis")
```




```
--- Top 10 Most Important Features ---
cultural_institution_density_log        0.144895
local_public_transit_density_log        0.100195
bars_nightlife_density_log              0.077109
transport_infrastructure_density_log    0.069269
arts_other_density_log                  0.061523
cafes_bakeries_density_log              0.058925
high_end_dining_density_log             0.058357
retail_food_beverage_density_log        0.051903
outdoors_other_density_log              0.051349
dining_other_density_log                0.047928
dtype: float64
```


Actual vs Predicted (Train & Test)
Train
Test
Ideal Fit
Predicted Values
Actual Values
0
500
1000
1500
2000
2500
3000


Residual Plot (Train & Test)
Train
Test
Residuals (Actual - Predicted)
Predicted Values
−500
−250
0
250
500
750
1000
1250
1500
0
200
400
600
800
1000
1200
1400
1600

1.4 4. Geographically Weighted Regression

[14]:
```
# Drop rows with NaN values in the target column - this applies to all targets
london_airbnb_tourism_fs_clean = london_airbnb_tourism_fs.
 ↪dropna(subset=['total_revenue'])

# Define the features and target variable
gwr_feature_columns = [
    'Park/Garden_density_log',
    'Lodging (Hotel/Hostel etc.)_density_log',
    'Retail (Grocery/Convenience)_density_log'
    ]

gwr_feature_columns = [
    'cultural_institution_density_log',
    'cafes_bakeries_density_log',
    'bars_nightlife_density_log',
    'local_public_transit_density_log',
    'retail_food_beverage_density_log',
    'transport_infrastructure_density_log'
]
```

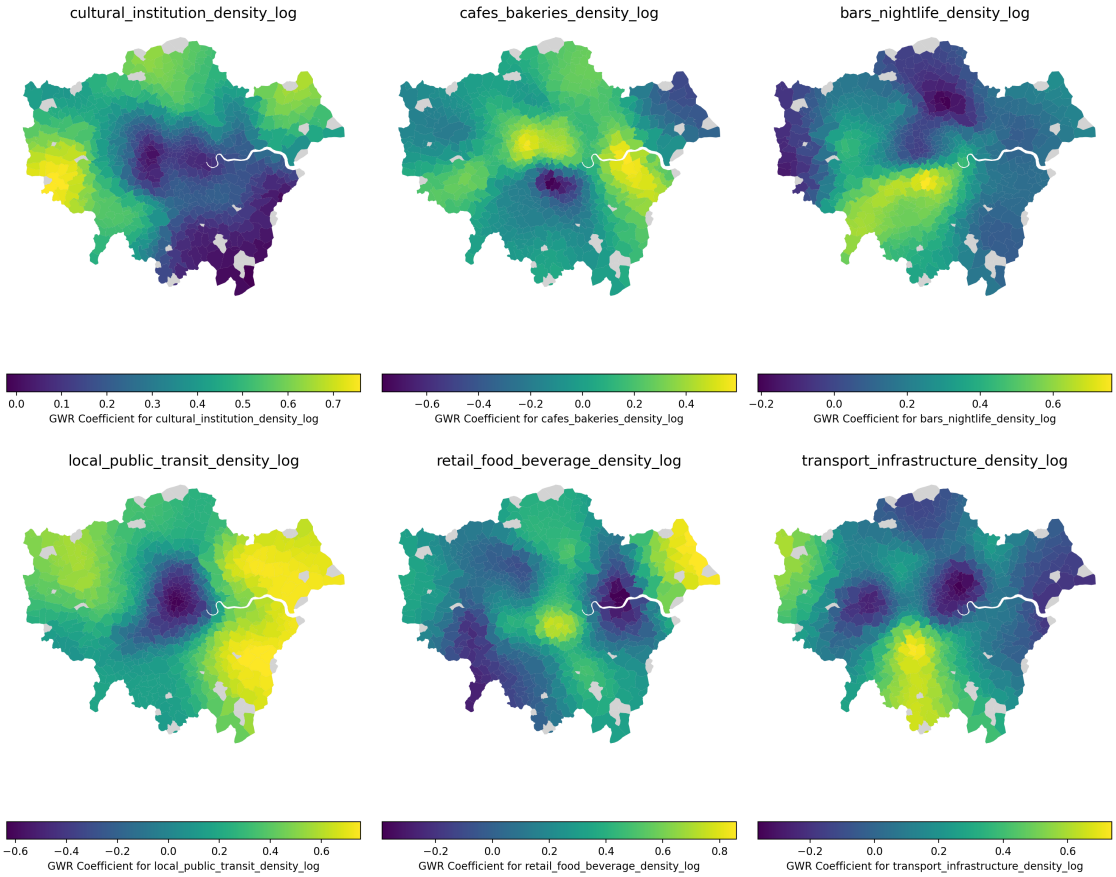
1.4.1 Total Revenue

[15]:
```
import warnings
warnings.filterwarnings("ignore")

# Define the target variable
target_column_log = 'total_revenue_log'

df = london_airbnb_tourism_fs_clean

# Prepare X, y, and coordinates for GWR
y = df[target_column_log].values.reshape((-1, 1))
X = df[gwr_feature_columns].values
coords_df = df.to_crs('EPSG:27700').geometry.centroid
coords = list(zip(coords_df.x, coords_df.y))

# 3. Find the Optimal Bandwidth
print(f"--- Running GWR for: {target_column_log} ---")
print("Searching for optimal GWR bandwidth...")
selector = Sel_BW(coords, y, X)
best_bandwidth = selector.search()
print(f"Optimal bandwidth found: {best_bandwidth}")

# 4. Run the GWR Model
print("\nRunning GWR model...")
```

```
gwr_model = GWR(coords, y, X, bw=best_bandwidth)
gwr_results = gwr_model.fit()
print(gwr_results.summary())

# 5. Plot GWR Coefficients
map_gwr_coefficients(df, gwr_results, gwr_feature_columns,␣
 ↪london_airbnb_tourism_fs)

# 6. Calculate t-values
calculate_t_values(df, gwr_results, gwr_feature_columns)

# 7. Categorize significance
categorise_significance(df, gwr_feature_columns)

# 8. Plot significance maps
plot_significance(df, gwr_feature_columns, london_airbnb_tourism_fs)
```

```
--- Running GWR for: total_revenue_log ---
Searching for optimal GWR bandwidth…
Optimal bandwidth found: 170.0

Running GWR model…
===========================================================================
Model type                                                       Gaussian
Number of observations:                                               629
Number of covariates:                                                   7

Global Regression Results
---------------------------------------------------------------------------
Residual sum of squares:                                         635.131
Log-likelihood:                                                 -895.563
AIC:                                                            1805.126
AICc:                                                           1807.358
BIC:                                                           -3373.119
R2:                                                                0.666
Adj. R2:                                                           0.663

Variable                        Est.         SE  t(Est/SE)    p-value
------------------------------ ---------- ---------- ---------- ----------
X0                              9.761      0.115      84.794      0.000
X1                              0.331      0.072      4.621       0.000
X2                              0.035      0.114      0.304       0.761
X3                              0.128      0.095      1.350       0.177
X4                              0.322      0.082      3.920       0.000
X5                              0.358      0.092      3.901       0.000
X6                              0.193      0.092      2.092       0.036

Geographically Weighted Regression (GWR) Results
```

```
-------------------------------------------------------------------------
Spatial kernel:                                        Adaptive bisquare
Bandwidth used:                                                  170.000

Diagnostic information
-------------------------------------------------------------------------
Residual sum of squares:                                         432.848
Effective number of parameters (trace(S)):                        65.040
Degree of freedom (n - trace(S)):                                563.960
Sigma estimate:                                                    0.876
Log-likelihood:                                                 -774.970
AIC:                                                            1682.020
AICc:                                                           1697.777
BIC:                                                            1975.512
R2:                                                                0.772
Adjusted R2:                                                       0.746
Adj. alpha (95%):                                                  0.005
Adj. critical t value (95%):                                       2.793

Summary Statistics For GWR Parameter Estimates
-------------------------------------------------------------------------
Variable                   Mean        STD        Min     Median        Max
-------------------- ---------- ---------- ---------- ---------- ----------
X0                       10.474      0.980      9.068     10.309     12.969
X1                        0.317      0.194     -0.021      0.287      0.761
X2                        0.087      0.255     -0.772      0.114      0.593
X3                        0.225      0.210     -0.209      0.180      0.761
X4                        0.197      0.364     -0.634      0.242      0.754
X5                        0.220      0.237     -0.388      0.231      0.859
X6                        0.126      0.244     -0.362      0.088      0.737
=========================================================================

None
```


cultural_institution_density_log
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7
GWR Coefficient for cultural_institution_density_log
cafes_bakeries_density_log
−0.6 −0.4 −0.2 0.0 0.2 0.4
GWR Coefficient for cafes_bakeries_density_log
bars_nightlife_density_log
−0.2 0.0 0.2 0.4 0.6
GWR Coefficient for bars_nightlife_density_log
local_public_transit_density_log
−0.6 −0.4 −0.2 0.0 0.2 0.4 0.6
GWR Coefficient for local_public_transit_density_log
retail_food_beverage_density_log
−0.2 0.0 0.2 0.4 0.6 0.8
GWR Coefficient for retail_food_beverage_density_log
transport_infrastructure_density_log
−0.2 0.0 0.2 0.4 0.6
GWR Coefficient for transport_infrastructure_density_log

1.4.2 Total Volume

[16]:
```
import warnings
warnings.filterwarnings("ignore")

# Define the target variable
target_column_log = 'total_volume_log'

df = london_airbnb_tourism_fs_clean

# Prepare X, y, and coordinates for GWR
y = df[target_column_log].values.reshape((-1, 1))
X = df[gwr_feature_columns].values
coords_df = df.to_crs('EPSG:27700').geometry.centroid
coords = list(zip(coords_df.x, coords_df.y))

# 3. Find the Optimal Bandwidth
print(f"--- Running GWR for: {target_column_log} ---")
print("Searching for optimal GWR bandwidth...")
selector = Sel_BW(coords, y, X)
best_bandwidth = selector.search()
print(f"Optimal bandwidth found: {best_bandwidth}")
```

```python
# 4. Run the GWR Model
print("\nRunning GWR model...")
gwr_model = GWR(coords, y, X, bw=best_bandwidth)
gwr_results = gwr_model.fit()
print(gwr_results.summary())

# 5. Plot GWR Coefficients
map_gwr_coefficients(df, gwr_results, gwr_feature_columns,␣
↪london_airbnb_tourism_fs)

# 6. Calculate t-values
calculate_t_values(df, gwr_results, gwr_feature_columns)

# 7. Categorize significance
categorise_significance(df, gwr_feature_columns)

# 8. Plot significance maps
plot_significance(df, gwr_feature_columns, london_airbnb_tourism_fs)
```

```
--- Running GWR for: total_volume_log ---
Searching for optimal GWR bandwidth…
Optimal bandwidth found: 170.0

Running GWR model…
===========================================================================
Model type                                                       Gaussian
Number of observations:                                               629
Number of covariates:                                                   7

Global Regression Results
---------------------------------------------------------------------------
Residual sum of squares:                                         612.371
Log-likelihood:                                                 -884.086
AIC:                                                            1782.172
AICc:                                                           1784.404
BIC:                                                           -3395.879
R2:                                                                0.618
Adj. R2:                                                           0.614

Variable                       Est.         SE  t(Est/SE)    p-value
------------------------------ ---------- ---------- ---------- ----------
X0                            6.426      0.113     56.852      0.000
X1                            0.219      0.070      3.113      0.002
X2                            0.055      0.112      0.485      0.628
X3                            0.060      0.093      0.650      0.516
X4                            0.297      0.081      3.687      0.000
X5                            0.474      0.090      5.267      0.000
```

X6	0.123	0.091	1.358	0.174

Geographically Weighted Regression (GWR) Results

Spatial kernel:	Adaptive bisquare
Bandwidth used:	170.000

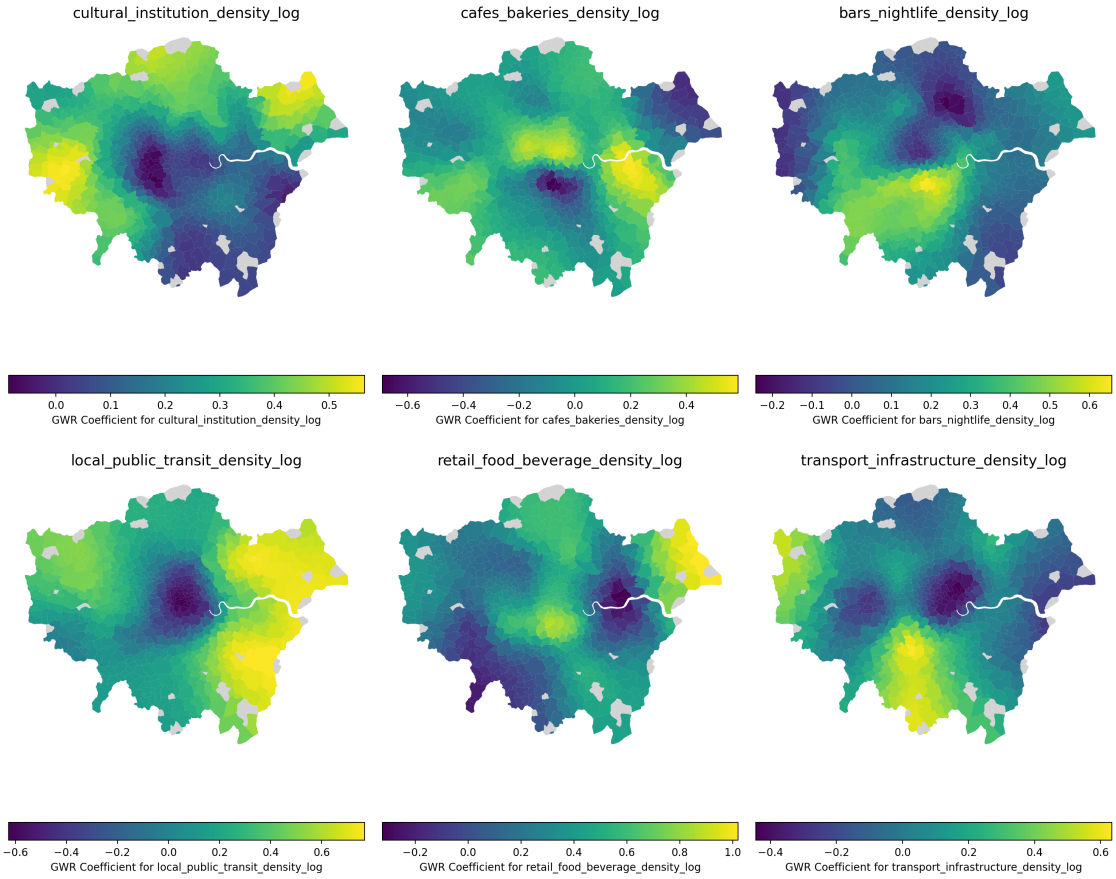
Diagnostic information

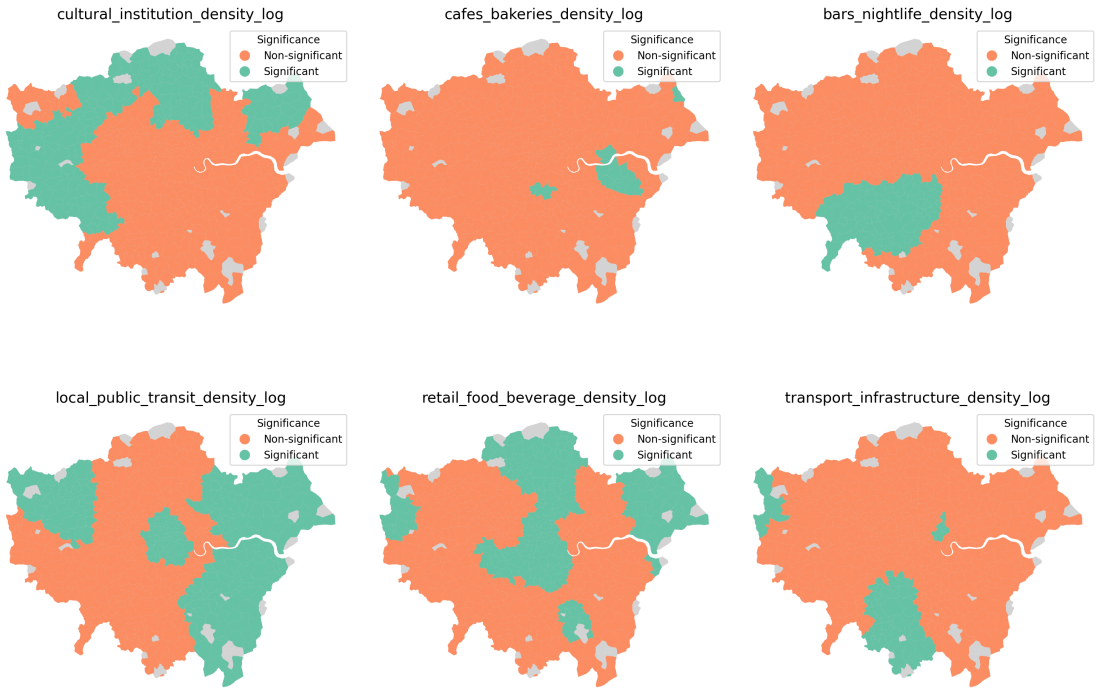
Residual sum of squares:	422.999
Effective number of parameters (trace(S)):	65.040
Degree of freedom (n - trace(S)):	563.960
Sigma estimate:	0.866
Log-likelihood:	-767.731
AIC:	1667.543
AICc:	1683.300
BIC:	1961.035
R2:	0.736
Adjusted R2:	0.706
Adj. alpha (95%):	0.005
Adj. critical t value (95%):	2.793

Summary Statistics For GWR Parameter Estimates

Variable	Mean	STD	Min	Median	Max
X0	7.113	0.951	5.579	6.951	9.438
X1	0.228	0.163	-0.087	0.200	0.565
X2	0.089	0.233	-0.691	0.101	0.585
X3	0.167	0.192	-0.242	0.132	0.656
X4	0.181	0.358	-0.626	0.199	0.768
X5	0.332	0.256	-0.324	0.335	1.020
X6	0.063	0.240	-0.444	0.034	0.633

None


cultural_institution_density_log
0.0 0.1 0.2 0.3 0.4 0.5
GWR Coefficient for cultural_institution_density_log
cafes_bakeries_density_log
−0.6 −0.4 −0.2 0.0 0.2 0.4
GWR Coefficient for cafes_bakeries_density_log
bars_nightlife_density_log
−0.2 −0.1 0.0 0.1 0.2 0.3 0.4 0.5 0.6
GWR Coefficient for bars_nightlife_density_log
local_public_transit_density_log
−0.6 −0.4 −0.2 0.0 0.2 0.4 0.6
GWR Coefficient for local_public_transit_density_log
retail_food_beverage_density_log
−0.2 0.0 0.2 0.4 0.6 0.8 1.0
GWR Coefficient for retail_food_beverage_density_log
transport_infrastructure_density_log
−0.4 −0.2 0.0 0.2 0.4 0.6
GWR Coefficient for transport_infrastructure_density_log



1.4.3 Guest Density

[17]:
```
import warnings
warnings.filterwarnings("ignore")

# Define the target variable
target_column_log = 'total_tourism_intensity_km2_log'

df = london_airbnb_tourism_fs_clean

# Prepare X, y, and coordinates for GWR
y = df[target_column_log].values.reshape((-1, 1))
X = df[gwr_feature_columns].values
coords_df = df.to_crs('EPSG:27700').geometry.centroid
coords = list(zip(coords_df.x, coords_df.y))

# 3. Find the Optimal Bandwidth
print(f"--- Running GWR for: {target_column_log} ---")
print("Searching for optimal GWR bandwidth...")
selector = Sel_BW(coords, y, X)
best_bandwidth = selector.search()
```

```
print(f"Optimal bandwidth found: {best_bandwidth}")

# 4. Run the GWR Model
print("\nRunning GWR model...")
gwr_model = GWR(coords, y, X, bw=best_bandwidth)
gwr_results = gwr_model.fit()
print(gwr_results.summary())

# 5. Plot GWR Coefficients
map_gwr_coefficients(df, gwr_results, gwr_feature_columns,␣
 ↪london_airbnb_tourism_fs)

# 6. Calculate t-values
calculate_t_values(df, gwr_results, gwr_feature_columns)

# 7. Categorize significance
categorise_significance(df, gwr_feature_columns)

# 8. Plot significance maps
plot_significance(df, gwr_feature_columns, london_airbnb_tourism_fs)
```

```
--- Running GWR for: total_tourism_intensity_km2_log ---
Searching for optimal GWR bandwidth…
Optimal bandwidth found: 91.0

Running GWR model…
===========================================================================
Model type                                                       Gaussian
Number of observations:                                               629
Number of covariates:                                                   7

Global Regression Results
---------------------------------------------------------------------------
Residual sum of squares:                                         385.825
Log-likelihood:                                                 -738.801
AIC:                                                            1491.602
AICc:                                                           1493.834
BIC:                                                           -3622.425
R2:                                                                0.800
Adj. R2:                                                           0.798

Variable                        Est.         SE  t(Est/SE)    p-value
------------------------------ ---------- ---------- ---------- ----------
X0                              0.760      0.090      8.474      0.000
X1                              0.500      0.056      8.968      0.000
X2                             -0.274      0.089     -3.073      0.002
X3                              0.121      0.074      1.637      0.102
X4                              0.399      0.064      6.236      0.000
```

X5	0.533	0.071	7.457	0.000
X6	0.292	0.072	4.065	0.000

Geographically Weighted Regression (GWR) Results

Spatial kernel:	Adaptive bisquare
Bandwidth used:	91.000

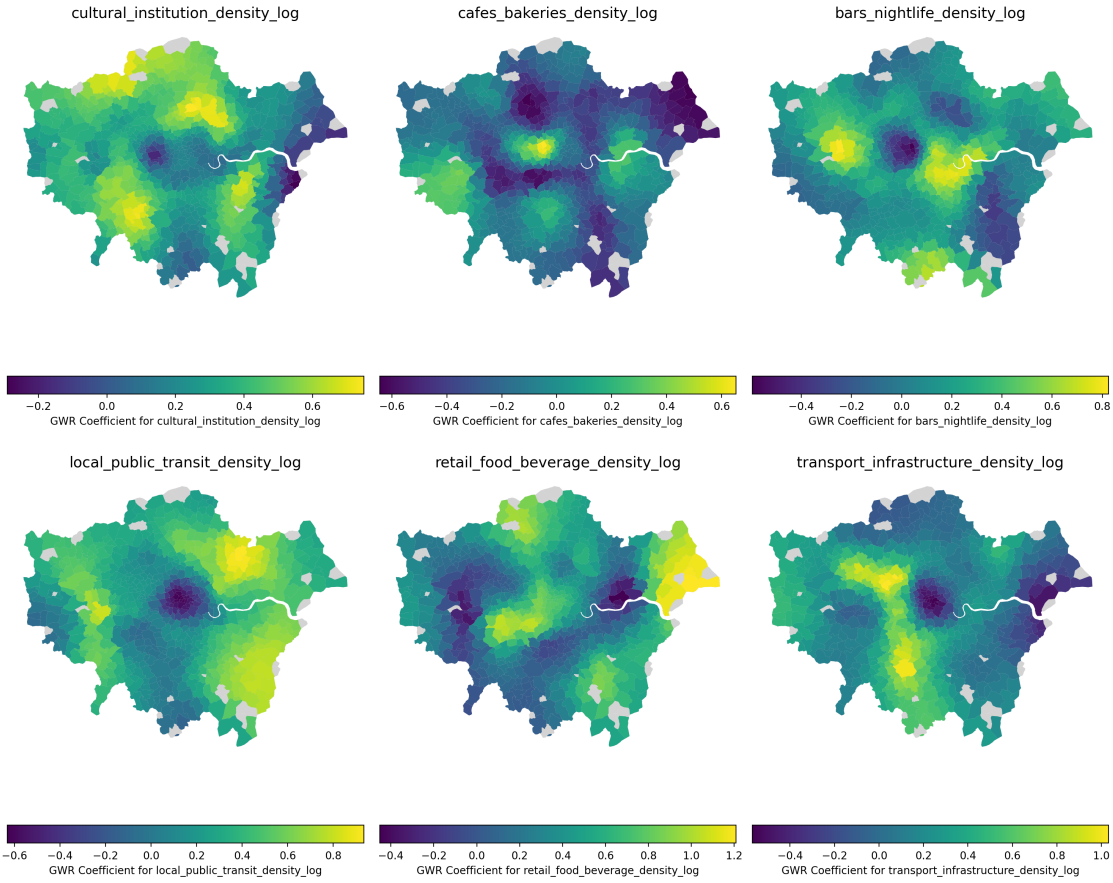
Diagnostic information

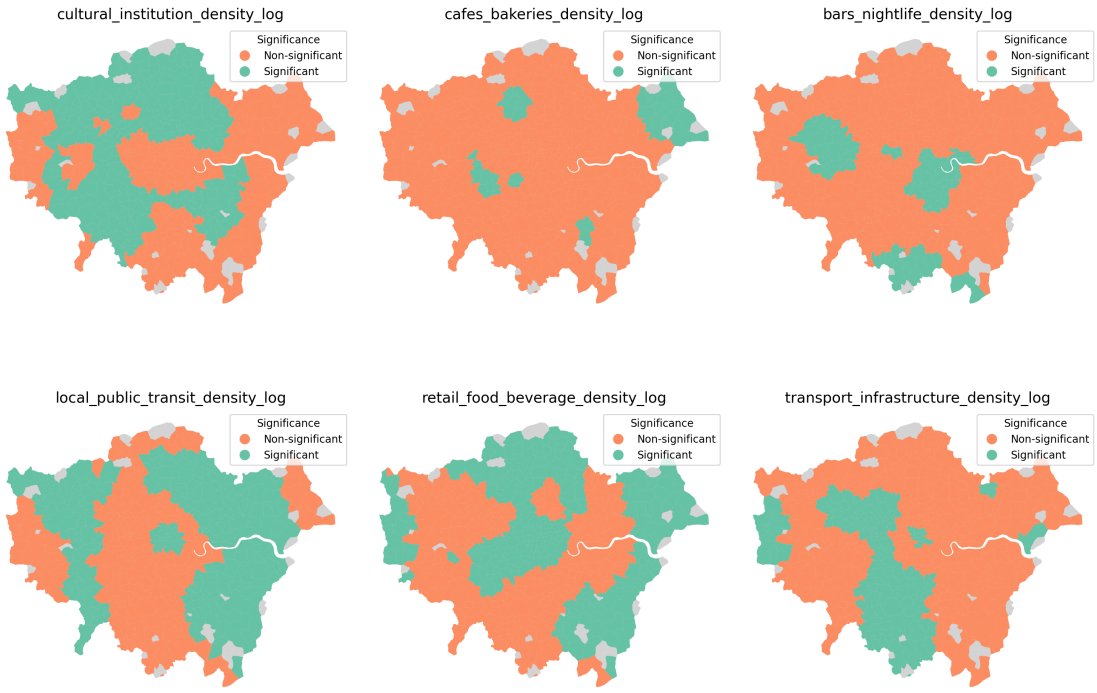
Residual sum of squares:	167.304
Effective number of parameters (trace(S)):	117.840
Degree of freedom (n - trace(S)):	511.160
Sigma estimate:	0.572
Log-likelihood:	-476.013
AIC:	1189.706
AICc:	1245.649
BIC:	1717.847
R2:	0.913
Adjusted R2:	0.893
Adj. alpha (95%):	0.003
Adj. critical t value (95%):	2.982

Summary Statistics For GWR Parameter Estimates

Variable	Mean	STD	Min	Median	Max
X0	1.509	0.932	0.207	1.271	4.172
X1	0.334	0.190	-0.291	0.334	0.750
X2	-0.151	0.252	-0.644	-0.166	0.653
X3	0.156	0.261	-0.596	0.148	0.825
X4	0.262	0.320	-0.631	0.284	0.934
X5	0.396	0.331	-0.452	0.382	1.209
X6	0.234	0.334	-0.570	0.200	1.033

None


cultural_institution_density_log
−0.2 0.0 0.2 0.4 0.6
GWR Coefficient for cultural_institution_density_log
cafes_bakeries_density_log
−0.6 −0.4 −0.2 0.0 0.2 0.4 0.6
GWR Coefficient for cafes_bakeries_density_log
bars_nightlife_density_log
−0.4 −0.2 0.0 0.2 0.4 0.6 0.8
GWR Coefficient for bars_nightlife_density_log
local_public_transit_density_log
−0.6 −0.4 −0.2 0.0 0.2 0.4 0.6 0.8
GWR Coefficient for local_public_transit_density_log
retail_food_beverage_density_log
−0.4 −0.2 0.0 0.2 0.4 0.6 0.8 1.0 1.2
GWR Coefficient for retail_food_beverage_density_log
transport_infrastructure_density_log
−0.4 −0.2 0.0 0.2 0.4 0.6 0.8 1.0
GWR Coefficient for transport_infrastructure_density_log



1.5 5. Geographically Weighted Random Forest

1.5.1 Total Revenue

[18]:
```
# Set CRS
gdf = london_airbnb_tourism_fs.to_crs(epsg=27700)

# Get coordinates matrix
coords = np.array([[geom.centroid.x, geom.centroid.y] for geom in gdf.geometry])

# Target variable and predictors
target = "total_revenue_log"
X_cols = features_logged
```

[19]:
```
# --- Parameters ---
k_neighbors = 100           # Step 1: How many neighbors to use for each local␣
 ↪model
min_local_data = 10         # Step 2: Minimum required valid observations for␣
 ↪local fitting

# --- Build spatial index ---
# Step 3: Extract centroid coordinates for each spatial unit
```

```
coords = np.array([[geom.centroid.x, geom.centroid.y] for geom in gdf.geometry])

# Step 4: Build a KD-Tree for fast nearest-neighbor queries
tree = cKDTree(coords)

# --- Outputs ---
# Step 5: Prepare lists to store model results
predictions = []                # Store predicted values at each location
local_r2s = []                  # Store local R² values
feature_importance_list = []    # Store feature importance vectors

# --- Loop through each location ---
# Step 6: Loop over every spatial unit (polygon)
for i, point in enumerate(coords):
    # Step 7: Find k nearest neighbors around the focal point
    distances, indices = tree.query(point, k=k_neighbors)
    raw_local_data = gdf.iloc[indices]

    # Step 8: Remove neighbors that have missing values in predictors or target
    local_data = raw_local_data.dropna(subset=[target] + X_cols)

    # Step 9: If not enough valid data, skip this location
    if len(local_data) < min_local_data:
        predictions.append(np.nan)
        local_r2s.append(np.nan)
        feature_importance_list.append([np.nan] * len(X_cols))
        continue

    # Step 10: Get distances only for valid neighbors
    valid_idx_mask = raw_local_data.index.isin(local_data.index)
    valid_distances = distances[valid_idx_mask]

    # Step 11: Extract predictor and target matrices
    X_local = local_data[X_cols].values
    y_local = local_data[target].values

    # Step 12: Compute bisquare weights for valid neighbors
    D = valid_distances.max()
    weights = (1 - (valid_distances / D) ** 2) ** 2
    weights[valid_distances >= D] = 0

    # Step 13: Normalize weights into probabilities for sampling
    prob = weights / weights.sum()

    # Step 14: Sample with replacement using weights - spatial bootstrap
    sample_idx = np.random.choice(len(X_local), size=len(X_local), p=prob,
                                  replace=True)
```

```
    X_weighted = X_local[sample_idx]
    y_weighted = y_local[sample_idx]

    # Step 15: Train local random forest on spatially weighted data
    rf = RandomForestRegressor(n_estimators=100, random_state=42)
    rf.fit(X_weighted, y_weighted)

    # Step 16: Evaluate model R² on *original* local data (not weighted)
    y_local_pred = rf.predict(X_local)
    r2_local = r2_score(y_local, y_local_pred)
    local_r2s.append(r2_local)

    # Step 17: Predict the target value at the focal location itself
    X_pred = gdf.iloc[[i]][X_cols].values
    pred = rf.predict(X_pred)[0]
    predictions.append(pred)

    # Step 18: Save feature importances for this local model
    feature_importance_list.append(rf.feature_importances_)

    # Step 19: Print progress
    if i % 50 == 0:
        print(f"Processed {i}/{len(gdf)} points")

# --- Store predictions, R², and feature importances ---
# Step 20: Add prediction and R² columns to the GeoDataFrame
gdf["grf_prediction"] = predictions
gdf["grf_local_r2"] = local_r2s

# Step 21: Convert list of feature importance arrays into a DataFrame and merge
importances_df = pd.DataFrame(feature_importance_list,
                              columns=[f"{col}_importance" for col in X_cols])
gdf = gdf.join(importances_df)

# --- Identify top feature per location ---
# Step 22: Find the feature with the highest importance at each location
importance_cols = [f"{col}_importance" for col in X_cols]
importance_only = gdf[importance_cols]

# Step 23: Get the name of the most important POI for each polygon
gdf["top_poi"] = importance_only.idxmax(axis=1)

# Step 24: Clean up the POI column names for readability
gdf["top_poi_clean"] = gdf["top_poi"].str.replace("_density_log_importance", "",
                                                  regex=False)
gdf["top_poi_clean"] = gdf["top_poi"].str.replace("_log_importance", "",
                                                  regex=False)
```

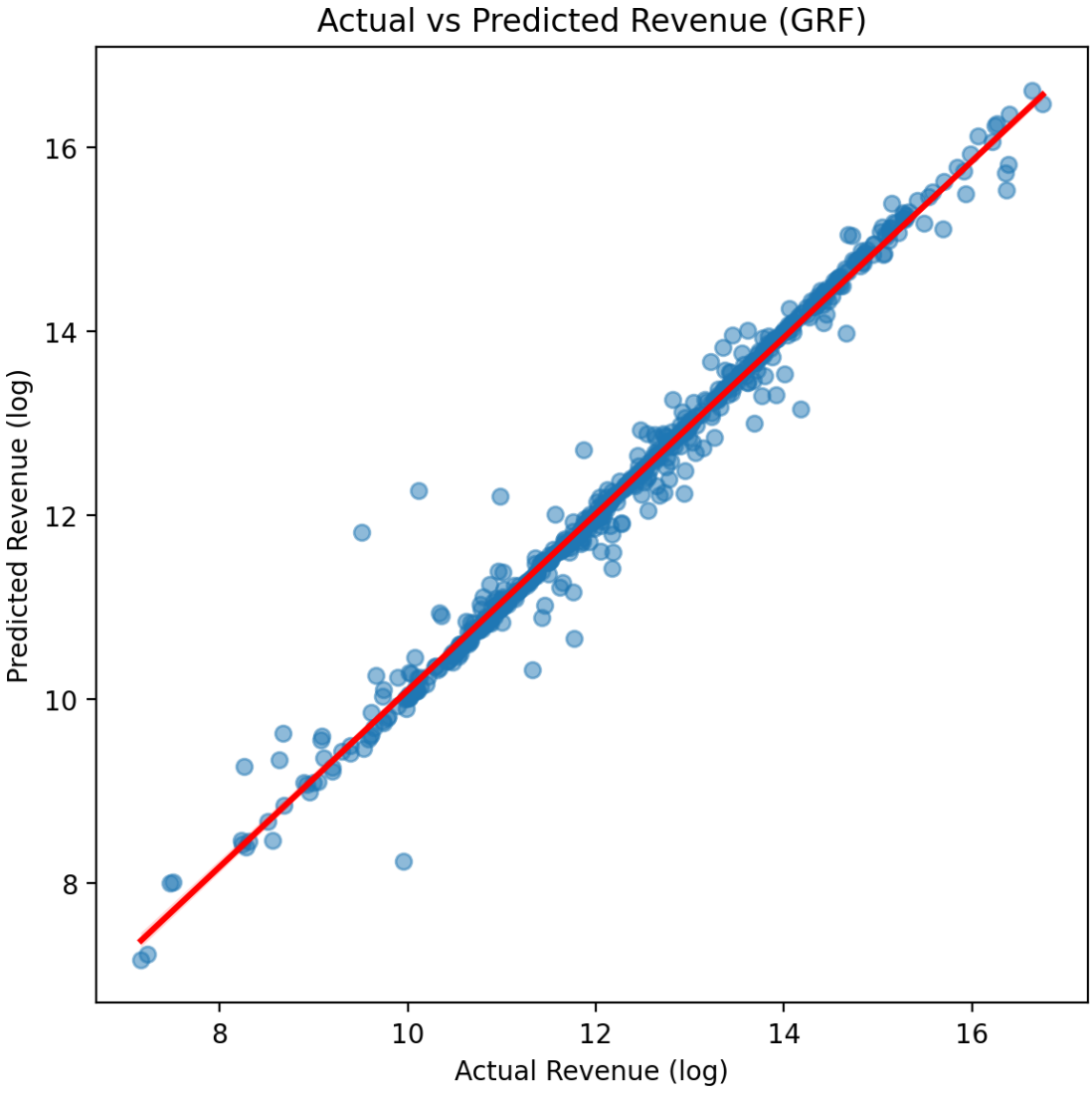
```
Processed 0/657 points
Processed 50/657 points
Processed 100/657 points
Processed 150/657 points
Processed 200/657 points
Processed 250/657 points
Processed 300/657 points
Processed 350/657 points
Processed 400/657 points
Processed 450/657 points
Processed 500/657 points
Processed 550/657 points
Processed 600/657 points
Processed 650/657 points
```

[20]:
```python
# Mask for valid (non-missing) rows
valid_mask = gdf["grf_prediction"].notna() & gdf[target].notna()

# Extract valid predictions and true values (log scale)
y_true_log = gdf.loc[valid_mask, target]
y_pred_log = gdf.loc[valid_mask, "grf_prediction"]

# ---- Log-scale metrics ----
rmse_log = np.sqrt(mean_squared_error(y_true_log, y_pred_log))
mae_log = mean_absolute_error(y_true_log, y_pred_log)
r2 = r2_score(y_true_log, y_pred_log)

# ---- Original-scale metrics ----
y_true_orig = np.exp(y_true_log)
y_pred_orig = np.exp(y_pred_log)

rmse_orig = np.sqrt(mean_squared_error(y_true_orig, y_pred_orig))
mae_orig = mean_absolute_error(y_true_orig, y_pred_orig)

# ---- Print all metrics ----
print("GRF Evaluation Metrics")
print("----------------------")
print(f"R²: {r2:.3f}")
print(f"RMSE (log scale): {rmse_log:.3f}")
print(f"MAE  (log scale): {mae_log:.3f}")
print(f"RMSE (original scale): {rmse_orig:,.2f}")
print(f"MAE  (original scale): {mae_orig:,.2f}")
```

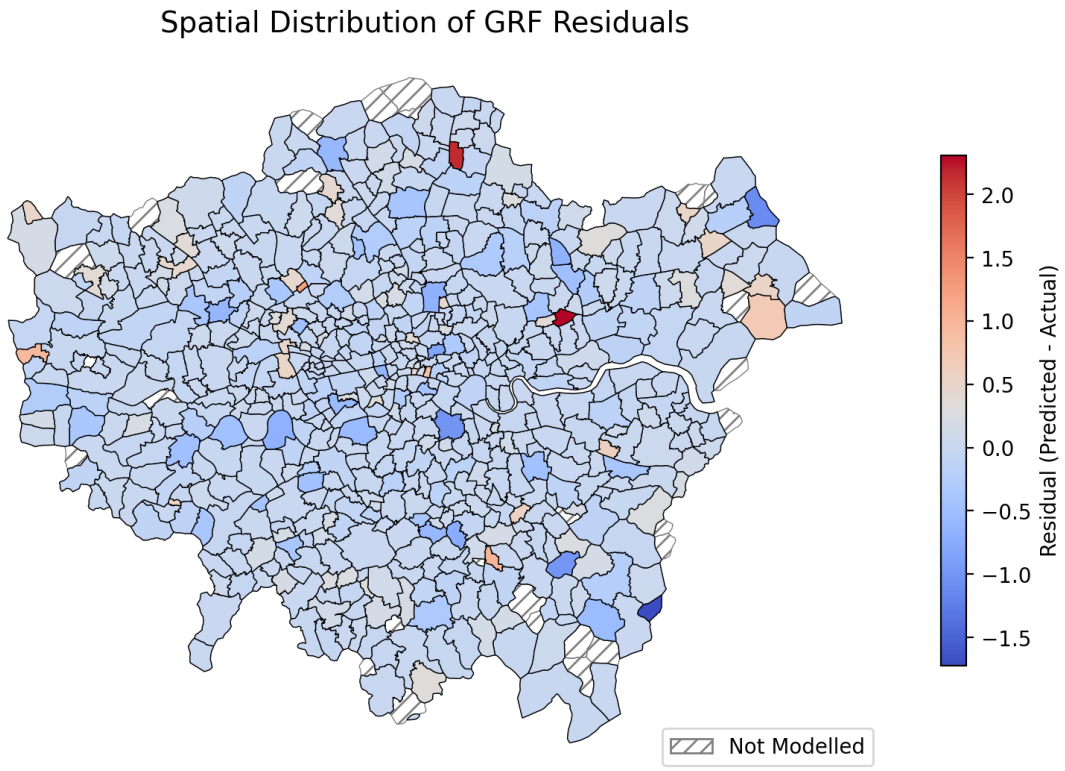
```
GRF Evaluation Metrics
----------------------
R²: 0.980
RMSE (log scale): 0.245
MAE  (log scale): 0.109
```

```
RMSE (original scale): 524,700.91
MAE  (original scale): 103,070.71
```

[21]:
```
plt.figure(figsize=(6, 6))
sns.regplot(x=gdf.loc[valid_mask, target],
            y=gdf.loc[valid_mask, "grf_prediction"],
            line_kws={'color': 'red'},
            scatter_kws={'alpha': 0.5})

plt.xlabel("Actual Revenue (log)")
plt.ylabel("Predicted Revenue (log)")
plt.title("Actual vs Predicted Revenue (GRF)")
plt.grid(False)
plt.tight_layout()
plt.show()
```

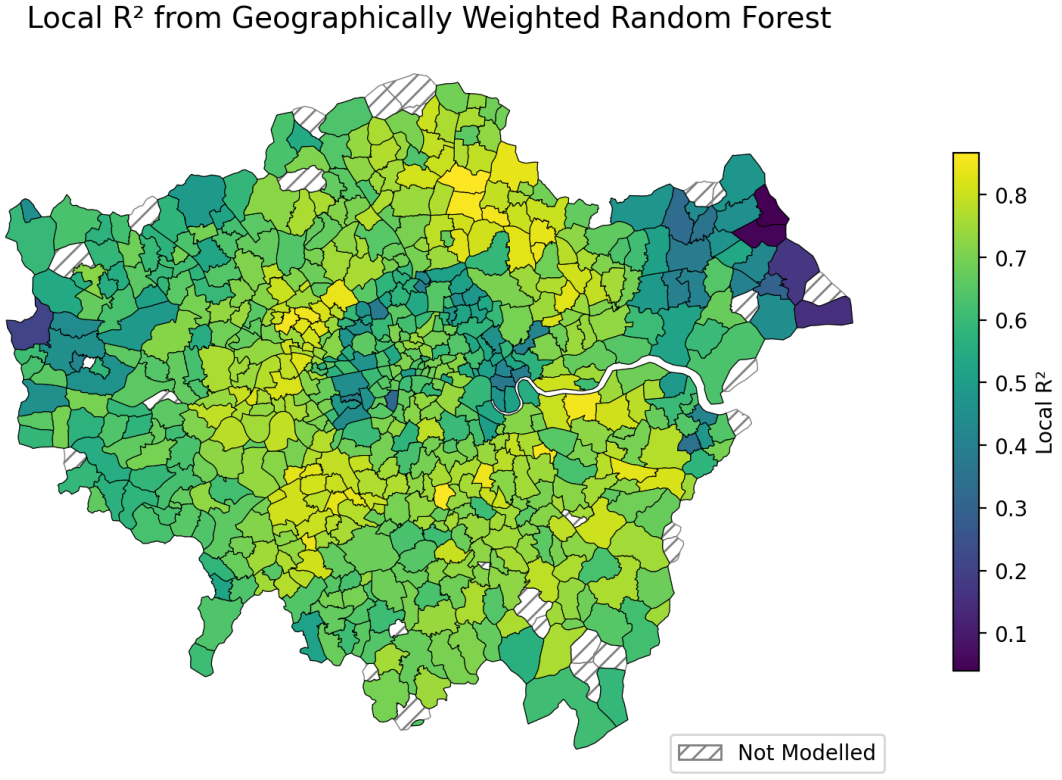


[22]:
```
gdf["grf_residual"] = gdf["grf_prediction"] - gdf[target]

# Filter valid predictions
gdf_valid = gdf[valid_mask]

# Create shared figure and axis
fig, ax = plt.subplots(figsize=(8, 6))

# --- Baselayer: all boundaries with hatch for unmodelled areas ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
    hatch='///',
    linewidth=0.5,
    zorder=0
)

# --- Residuals map ---
gdf_valid.plot(
    ax=ax,
    column="grf_residual",
    cmap="coolwarm",
    edgecolor='black',
    linewidth=0.5,
    legend=True,
    legend_kwds={
        'label': "Residual (Predicted - Actual)",
        'shrink': 0.6
    },
    zorder=1
)

# --- Final styling ---
hatch_patch = Patch(facecolor='white', edgecolor='gray', hatch='///',␣
 ↪label='Not Modelled')
ax.legend(handles=[hatch_patch], loc='lower right')

ax.set_title("Spatial Distribution of GRF Residuals", fontsize=14)
ax.axis('off')
plt.tight_layout()
plt.show()
```



[23]:
```
# Filter to valid R² values
gdf_valid_r2 = gdf_valid[gdf_valid["grf_local_r2"].notna()]

# Plot
fig, ax = plt.subplots(figsize=(8, 6))

# --- Baselayer ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
    hatch='///',
    linewidth=0.5,
    zorder=0
)

# --- R² choropleth ---
gdf_valid_r2.plot(
    ax=ax,
    column="grf_local_r2",
    cmap="viridis",
    edgecolor="black",
```

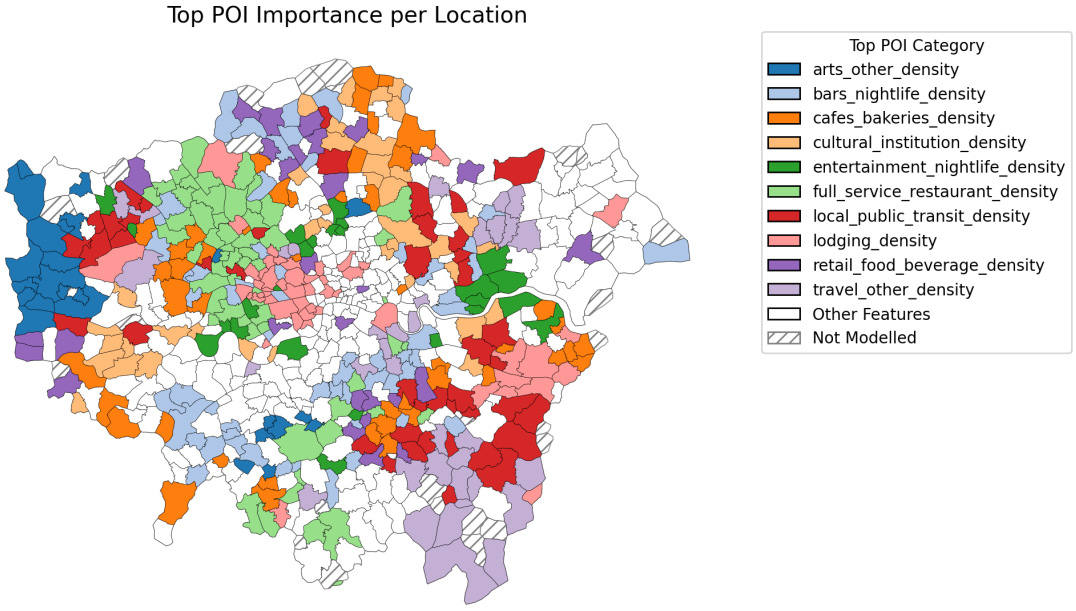
```
    linewidth=0.4,
    legend=True,
    legend_kwds={
        'label': "Local R²",
        'shrink': 0.6,
        'orientation': "vertical"
    },
    zorder=1  # On top
)

hatch_patch = Patch(facecolor='white', edgecolor='gray', hatch='///',␣
 ↪label='Not Modelled')
ax.legend(handles=[hatch_patch], loc='lower right')
# --- Styling ---
ax.set_title("Local R² from Geographically Weighted Random Forest", fontsize=14)
ax.axis("off")
plt.tight_layout()
plt.show()
```



[24]:
```
# --- Step 0: Prepare GeoDataFrame ---
gdf_valid = gdf[valid_mask].copy()  # Ensure gdf_valid is up to date
```

```
# --- Step 1: Define top N POI features to highlight ---
top_n = 10
top_pois = (
    gdf_valid["top_poi_clean"]
    .value_counts()
    .nlargest(top_n)
    .index.tolist()
)

# --- Step 2: Assign "Other Features" to less frequent ones ---
gdf_valid["top_poi_plot"] = gdf_valid["top_poi_clean"].where(
    gdf_valid["top_poi_clean"].isin(top_pois),
    "Other Features"
)

# --- Step 3: Build color map ---
unique_pois = sorted(set(top_pois))
unique_pois.append("Other Features")

colors = list(plt.cm.tab20.colors)
while len(colors) < len(unique_pois) - 1:
    colors += colors  # Ensure enough colors

poi_color_map = dict(zip(unique_pois[:-1], colors[:len(unique_pois) - 1]))
poi_color_map["Other Features"] = "white"

# --- Step 4: Create figure and axis ---
fig, ax = plt.subplots(figsize=(10, 8))

# --- Step 5: Plot full Edinburgh shapefile as base (hatch for non-modeled) ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
    hatch="///",
    linewidth=0.5,
    zorder=0
)

# --- Step 6: Plot colored GRF results ---
for poi, color in poi_color_map.items():
    subset = gdf_valid[gdf_valid["top_poi_plot"] == poi]
    subset.plot(ax=ax, color=color, edgecolor="black", linewidth=0.2, zorder=1)

# --- Step 7: Add legend ---
legend_elements = [
```

```
    Patch(facecolor=poi_color_map[poi], edgecolor='black', label=poi)
    for poi in unique_pois
]

# Add legend entry for hatch
legend_elements.append(
    Patch(facecolor='white', edgecolor='gray', hatch='///', label='Not␣
 ↪Modelled')
)

ax.legend(
    handles=legend_elements,
    title="Top POI Category",
    bbox_to_anchor=(1.05, 1), loc="upper left", borderaxespad=0.
)

# --- Step 8: Finalize map ---
plt.title("Top POI Importance per Location", fontsize=14)
plt.axis("off")
plt.tight_layout()
plt.show()
```



1.5.2 Total Volume

[25]:
```
# Set CRS
gdf = london_airbnb_tourism_fs.to_crs(epsg=27700)

# Get coordinates matrix
coords = np.array([[geom.centroid.x, geom.centroid.y] for geom in gdf.geometry])

# Target variable and predictors
target = "total_volume_log"
X_cols = features_logged
```

[26]:
```
# --- Parameters ---
k_neighbors = 100           # Step 1: How many neighbors to use for each local␣
 ↪model
min_local_data = 10         # Step 2: Minimum required valid observations for␣
 ↪local fitting

# --- Build spatial index ---
# Step 3: Extract centroid coordinates for each spatial unit
coords = np.array([[geom.centroid.x, geom.centroid.y] for geom in gdf.geometry])

# Step 4: Build a KD-Tree for fast nearest-neighbor queries
tree = cKDTree(coords)

# --- Outputs ---
# Step 5: Prepare lists to store model results
predictions = []                # Store predicted values at each location
local_r2s = []                  # Store local R² values
feature_importance_list = []    # Store feature importance vectors

# --- Loop through each location ---
# Step 6: Loop over every spatial unit (polygon)
for i, point in enumerate(coords):
    # Step 7: Find k nearest neighbors around the focal point
    distances, indices = tree.query(point, k=k_neighbors)
    raw_local_data = gdf.iloc[indices]

    # Step 8: Remove neighbors that have missing values in predictors or target
    local_data = raw_local_data.dropna(subset=[target] + X_cols)

    # Step 9: If not enough valid data, skip this location
    if len(local_data) < min_local_data:
        predictions.append(np.nan)
        local_r2s.append(np.nan)
        feature_importance_list.append([np.nan] * len(X_cols))
        continue
```

```
    # Step 10: Get distances only for valid neighbors
    valid_idx_mask = raw_local_data.index.isin(local_data.index)
    valid_distances = distances[valid_idx_mask]

    # Step 11: Extract predictor and target matrices
    X_local = local_data[X_cols].values
    y_local = local_data[target].values

    # Step 12: Compute bisquare weights for valid neighbors
    D = valid_distances.max()
    weights = (1 - (valid_distances / D) ** 2) ** 2
    weights[valid_distances >= D] = 0

    # Step 13: Normalize weights into probabilities for sampling
    prob = weights / weights.sum()

    # Step 14: Sample with replacement using weights - spatial bootstrap
    sample_idx = np.random.choice(len(X_local), size=len(X_local), p=prob,
                                  replace=True)
    X_weighted = X_local[sample_idx]
    y_weighted = y_local[sample_idx]

    # Step 15: Train local random forest on spatially weighted data
    rf = RandomForestRegressor(n_estimators=100, random_state=42)
    rf.fit(X_weighted, y_weighted)

    # Step 16: Evaluate model R² on *original* local data (not weighted)
    y_local_pred = rf.predict(X_local)
    r2_local = r2_score(y_local, y_local_pred)
    local_r2s.append(r2_local)

    # Step 17: Predict the target value at the focal location itself
    X_pred = gdf.iloc[[i]][X_cols].values
    pred = rf.predict(X_pred)[0]
    predictions.append(pred)

    # Step 18: Save feature importances for this local model
    feature_importance_list.append(rf.feature_importances_)

    # Step 19: Print progress
    if i % 50 == 0:
        print(f"Processed {i}/{len(gdf)} points")

# --- Store predictions, R², and feature importances ---
# Step 20: Add prediction and R² columns to the GeoDataFrame
gdf["grf_prediction"] = predictions
```

```
gdf["grf_local_r2"] = local_r2s

# Step 21: Convert list of feature importance arrays into a DataFrame and merge
importances_df = pd.DataFrame(feature_importance_list,
                              columns=[f"{col}_importance" for col in X_cols])
gdf = gdf.join(importances_df)

# --- Identify top feature per location ---
# Step 22: Find the feature with the highest importance at each location
importance_cols = [f"{col}_importance" for col in X_cols]
importance_only = gdf[importance_cols]

# Step 23: Get the name of the most important POI for each polygon
gdf["top_poi"] = importance_only.idxmax(axis=1)

# Step 24: Clean up the POI column names for readability
gdf["top_poi_clean"] = gdf["top_poi"].str.replace("_density_log_importance", "",
                                                  regex=False)
gdf["top_poi_clean"] = gdf["top_poi"].str.replace("_log_importance", "",
                                                  regex=False)
```

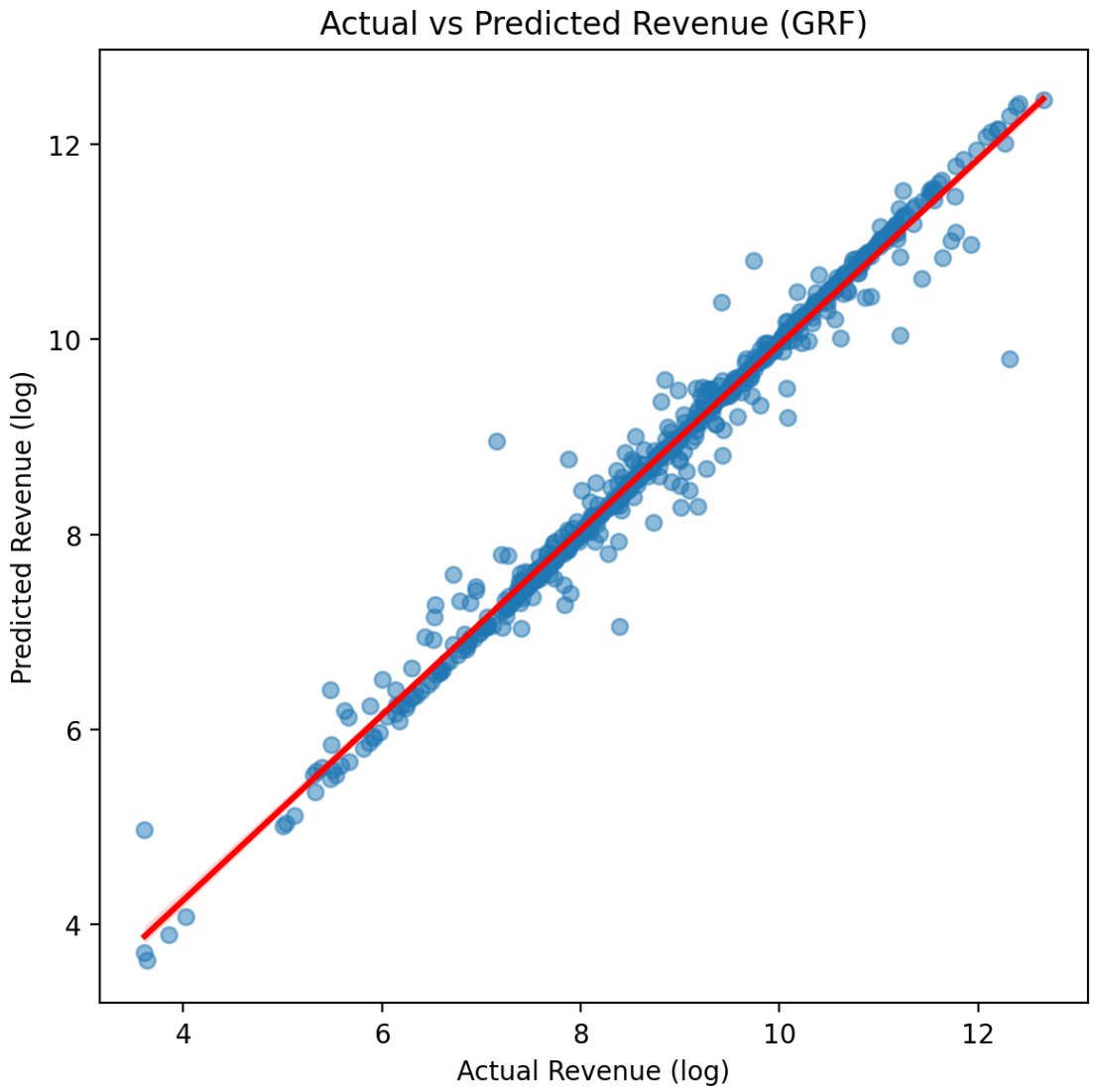
```
Processed 0/657 points
Processed 50/657 points
Processed 100/657 points
Processed 150/657 points
Processed 200/657 points
Processed 250/657 points
Processed 300/657 points
Processed 350/657 points
Processed 400/657 points
Processed 450/657 points
Processed 500/657 points
Processed 550/657 points
Processed 600/657 points
Processed 650/657 points
```

[27]:
```
# Evaluate log and original scale
valid_mask = gdf["grf_prediction"].notna() & gdf[target].notna()
y_true_log = gdf.loc[valid_mask, target]
y_pred_log = gdf.loc[valid_mask, "grf_prediction"]

# Metrics on log scale
rmse_log = np.sqrt(mean_squared_error(y_true_log, y_pred_log))
mae_log = mean_absolute_error(y_true_log, y_pred_log)
r2 = r2_score(y_true_log, y_pred_log)

# Metrics on original scale
```

```
y_true_orig = np.exp(y_true_log)
y_pred_orig = np.exp(y_pred_log)
rmse_orig = np.sqrt(mean_squared_error(y_true_orig, y_pred_orig))
mae_orig = mean_absolute_error(y_true_orig, y_pred_orig)

print("GRF Evaluation Metrics")
print("--------------------------------")
print(f"R² {r2:.3f}")
print(f"RMSE (log scale): {rmse_log:.3f}")
print(f"MAE  (log scale): {mae_log:.3f}")
print(f"RMSE (original scale): {rmse_orig:,.2f}")
print(f"MAE  (original scale): {mae_orig:,.2f}")
```

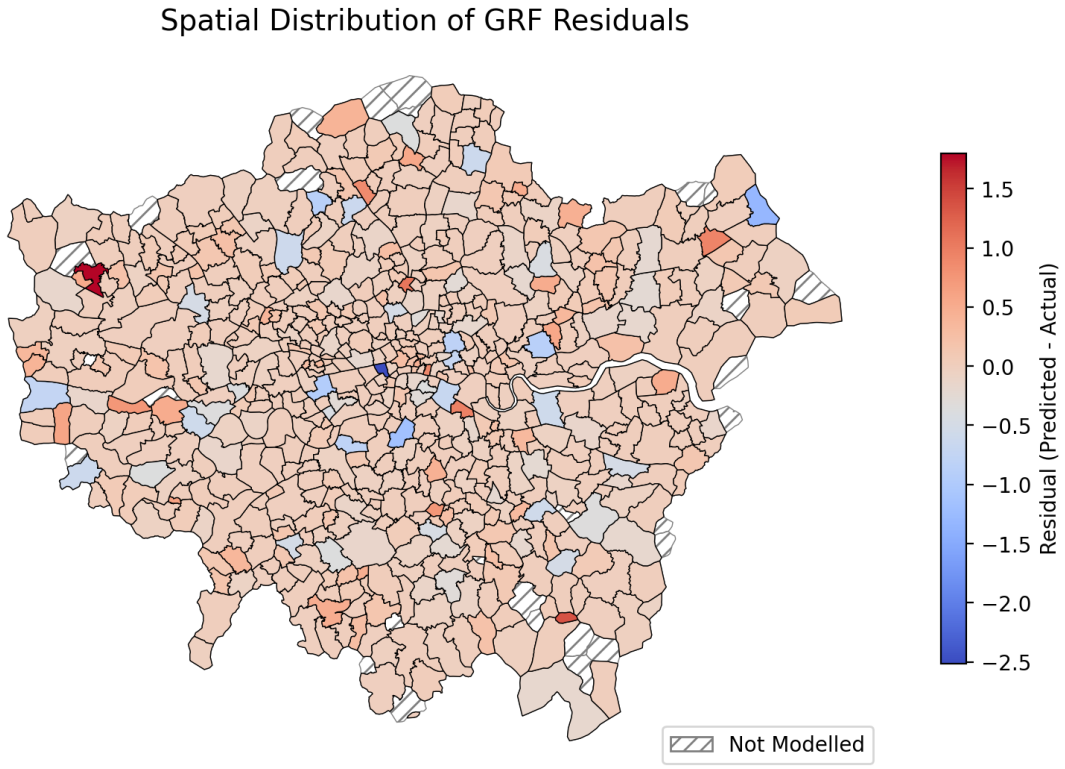
```
GRF Evaluation Metrics
--------------------------------
R² 0.975
RMSE (log scale): 0.251
MAE  (log scale): 0.111
RMSE (original scale): 11,347.28
MAE  (original scale): 2,341.68
```

[28]:
```
plt.figure(figsize=(6, 6))
sns.regplot(x=gdf.loc[valid_mask, target],
            y=gdf.loc[valid_mask, "grf_prediction"],
            line_kws={'color': 'red'},
            scatter_kws={'alpha': 0.5})

plt.xlabel("Actual Revenue (log)")
plt.ylabel("Predicted Revenue (log)")
plt.title("Actual vs Predicted Revenue (GRF)")
plt.grid(False)
plt.tight_layout()
plt.show()
```



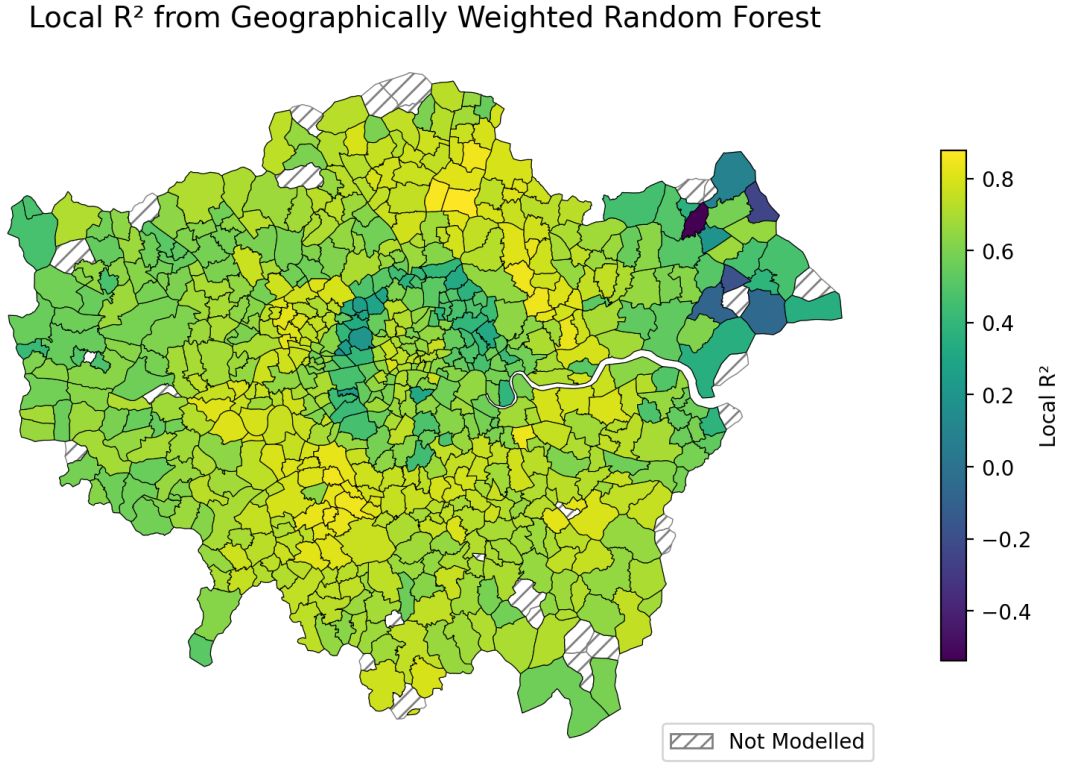
[29]:
```
gdf["grf_residual"] = gdf["grf_prediction"] - gdf[target]

# Filter valid predictions
gdf_valid = gdf[valid_mask]

# Create shared figure and axis
fig, ax = plt.subplots(figsize=(8, 6))

# --- Baselayer: all boundaries with hatch for unmodelled areas ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
```

```
    hatch='///',
    linewidth=0.5,
    zorder=0
)

# --- Residuals map ---
gdf_valid.plot(
    ax=ax,
    column="grf_residual",
    cmap="coolwarm",
    edgecolor='black',
    linewidth=0.5,
    legend=True,
    legend_kwds={
        'label': "Residual (Predicted - Actual)",
        'shrink': 0.6
    },
    zorder=1
)

# --- Final styling ---
hatch_patch = Patch(facecolor='white', edgecolor='gray', hatch='///',␣
 ↪label='Not Modelled')
ax.legend(handles=[hatch_patch], loc='lower right')

ax.set_title("Spatial Distribution of GRF Residuals", fontsize=14)
ax.axis('off')
plt.tight_layout()
plt.show()
```



[30]:
```
# Filter to valid R² values
gdf_valid_r2 = gdf_valid[gdf_valid["grf_local_r2"].notna()]

# Plot
fig, ax = plt.subplots(figsize=(8, 6))

# --- Baselayer ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
    hatch='///',
    linewidth=0.5,
    zorder=0
)

# --- R² choropleth ---
gdf_valid_r2.plot(
    ax=ax,
    column="grf_local_r2",
    cmap="viridis",
    edgecolor="black",
```

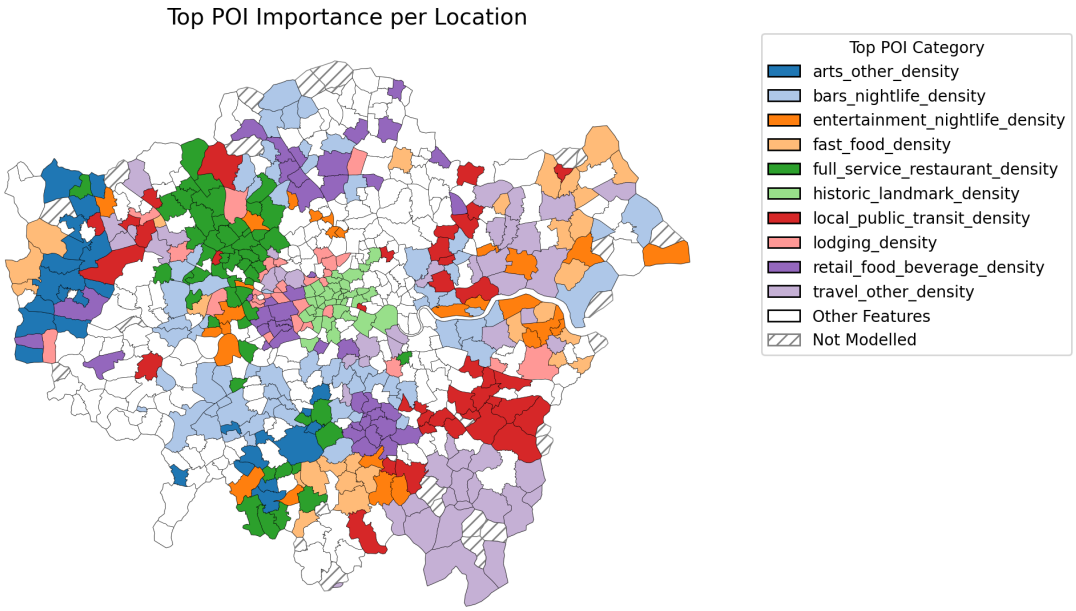
```
    linewidth=0.4,
    legend=True,
    legend_kwds={
        'label': "Local R²",
        'shrink': 0.6,
        'orientation': "vertical"
    },
    zorder=1  # On top
)

hatch_patch = Patch(facecolor='white', edgecolor='gray', hatch='///',␣
↪label='Not Modelled')
ax.legend(handles=[hatch_patch], loc='lower right')
# --- Styling ---
ax.set_title("Local R² from Geographically Weighted Random Forest", fontsize=14)
ax.axis("off")
plt.tight_layout()
plt.show()
```



[31]:
```
# --- Step 0: Prepare GeoDataFrame ---
gdf_valid = gdf[valid_mask].copy()  # Ensure gdf_valid is up to date
```

```
# --- Step 1: Define top N POI features to highlight ---
top_n = 10
top_pois = (
    gdf_valid["top_poi_clean"]
    .value_counts()
    .nlargest(top_n)
    .index.tolist()
)

# --- Step 2: Assign "Other Features" to less frequent ones ---
gdf_valid["top_poi_plot"] = gdf_valid["top_poi_clean"].where(
    gdf_valid["top_poi_clean"].isin(top_pois),
    "Other Features"
)

# --- Step 3: Build color map ---
unique_pois = sorted(set(top_pois))
unique_pois.append("Other Features")

colors = list(plt.cm.tab20.colors)
while len(colors) < len(unique_pois) - 1:
    colors += colors  # Ensure enough colors

poi_color_map = dict(zip(unique_pois[:-1], colors[:len(unique_pois) - 1]))
poi_color_map["Other Features"] = "white"

# --- Step 4: Create figure and axis ---
fig, ax = plt.subplots(figsize=(10, 8))

# --- Step 5: Plot full Edinburgh shapefile as base (hatch for non-modeled) ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
    hatch="///",
    linewidth=0.5,
    zorder=0
)

# --- Step 6: Plot colored GRF results ---
for poi, color in poi_color_map.items():
    subset = gdf_valid[gdf_valid["top_poi_plot"] == poi]
    subset.plot(ax=ax, color=color, edgecolor="black", linewidth=0.2, zorder=1)

# --- Step 7: Add legend ---
legend_elements = [
    Patch(facecolor=poi_color_map[poi], edgecolor='black', label=poi)
```

```
    for poi in unique_pois
]

# Add legend entry for hatch
legend_elements.append(
    Patch(facecolor='white', edgecolor='gray', hatch='///', label='Not␣
 ↪Modelled')
)

ax.legend(
    handles=legend_elements,
    title="Top POI Category",
    bbox_to_anchor=(1.05, 1), loc="upper left", borderaxespad=0.
)

# --- Step 8: Finalize map ---
plt.title("Top POI Importance per Location", fontsize=14)
plt.axis("off")
plt.tight_layout()
plt.show()
```



1.5.3 Guest Density

[32]:
```
# Set CRS
gdf = london_airbnb_tourism_fs.to_crs(epsg=27700)

# Get coordinates matrix
coords = np.array([[geom.centroid.x, geom.centroid.y] for geom in gdf.geometry])

# Target variable and predictors
target = "total_tourism_intensity_km2_log"
X_cols = features_logged
```

[ ]:
```
# --- Parameters ---
k_neighbors = 100           # Step 1: How many neighbors to use for each local␣
 ↪model
min_local_data = 10         # Step 2: Minimum required valid observations for␣
 ↪local fitting

# --- Build spatial index ---
# Step 3: Extract centroid coordinates for each spatial unit
coords = np.array([[geom.centroid.x, geom.centroid.y] for geom in gdf.geometry])

# Step 4: Build a KD-Tree for fast nearest-neighbor queries
tree = cKDTree(coords)

# --- Outputs ---
# Step 5: Prepare lists to store model results
predictions = []                # Store predicted values at each location
local_r2s = []                  # Store local R² values
feature_importance_list = []    # Store feature importance vectors

# --- Loop through each location ---
# Step 6: Loop over every spatial unit (polygon)
for i, point in enumerate(coords):
    # Step 7: Find k nearest neighbors around the focal point
    distances, indices = tree.query(point, k=k_neighbors)
    raw_local_data = gdf.iloc[indices]

    # Step 8: Remove neighbors that have missing values in predictors or target
    local_data = raw_local_data.dropna(subset=[target] + X_cols)

    # Step 9: If not enough valid data, skip this location
    if len(local_data) < min_local_data:
        predictions.append(np.nan)
        local_r2s.append(np.nan)
        feature_importance_list.append([np.nan] * len(X_cols))
        continue
```

```
    # Step 10: Get distances only for valid neighbors
    valid_idx_mask = raw_local_data.index.isin(local_data.index)
    valid_distances = distances[valid_idx_mask]

    # Step 11: Extract predictor and target matrices
    X_local = local_data[X_cols].values
    y_local = local_data[target].values

    # Step 12: Compute bisquare weights for valid neighbors
    D = valid_distances.max()
    weights = (1 - (valid_distances / D) ** 2) ** 2
    weights[valid_distances >= D] = 0

    # Step 13: Normalize weights into probabilities for sampling
    prob = weights / weights.sum()

    # Step 14: Sample with replacement using weights - spatial bootstrap
    sample_idx = np.random.choice(len(X_local), size=len(X_local), p=prob,
                                  replace=True)
    X_weighted = X_local[sample_idx]
    y_weighted = y_local[sample_idx]

    # Step 15: Train local random forest on spatially weighted data
    rf = RandomForestRegressor(n_estimators=100, random_state=42)
    rf.fit(X_weighted, y_weighted)

    # Step 16: Evaluate model R² on *original* local data (not weighted)
    y_local_pred = rf.predict(X_local)
    r2_local = r2_score(y_local, y_local_pred)
    local_r2s.append(r2_local)

    # Step 17: Predict the target value at the focal location itself
    X_pred = gdf.iloc[[i]][X_cols].values
    pred = rf.predict(X_pred)[0]
    predictions.append(pred)

    # Step 18: Save feature importances for this local model
    feature_importance_list.append(rf.feature_importances_)

    # Step 19: Print progress
    if i % 50 == 0:
        print(f"Processed {i}/{len(gdf)} points")

# --- Store predictions, R², and feature importances ---
# Step 20: Add prediction and R² columns to the GeoDataFrame
gdf["grf_prediction"] = predictions
```

```
gdf["grf_local_r2"] = local_r2s

# Step 21: Convert list of feature importance arrays into a DataFrame and merge
importances_df = pd.DataFrame(feature_importance_list,
                              columns=[f"{col}_importance" for col in X_cols])
gdf = gdf.join(importances_df)

# --- Identify top feature per location ---
# Step 22: Find the feature with the highest importance at each location
importance_cols = [f"{col}_importance" for col in X_cols]
importance_only = gdf[importance_cols]

# Step 23: Get the name of the most important POI for each polygon
gdf["top_poi"] = importance_only.idxmax(axis=1)

# Step 24: Clean up the POI column names for readability
gdf["top_poi_clean"] = gdf["top_poi"].str.replace("_density_log_importance", "",
                                                  regex=False)
gdf["top_poi_clean"] = gdf["top_poi"].str.replace("_log_importance", "",
                                                  regex=False)
```

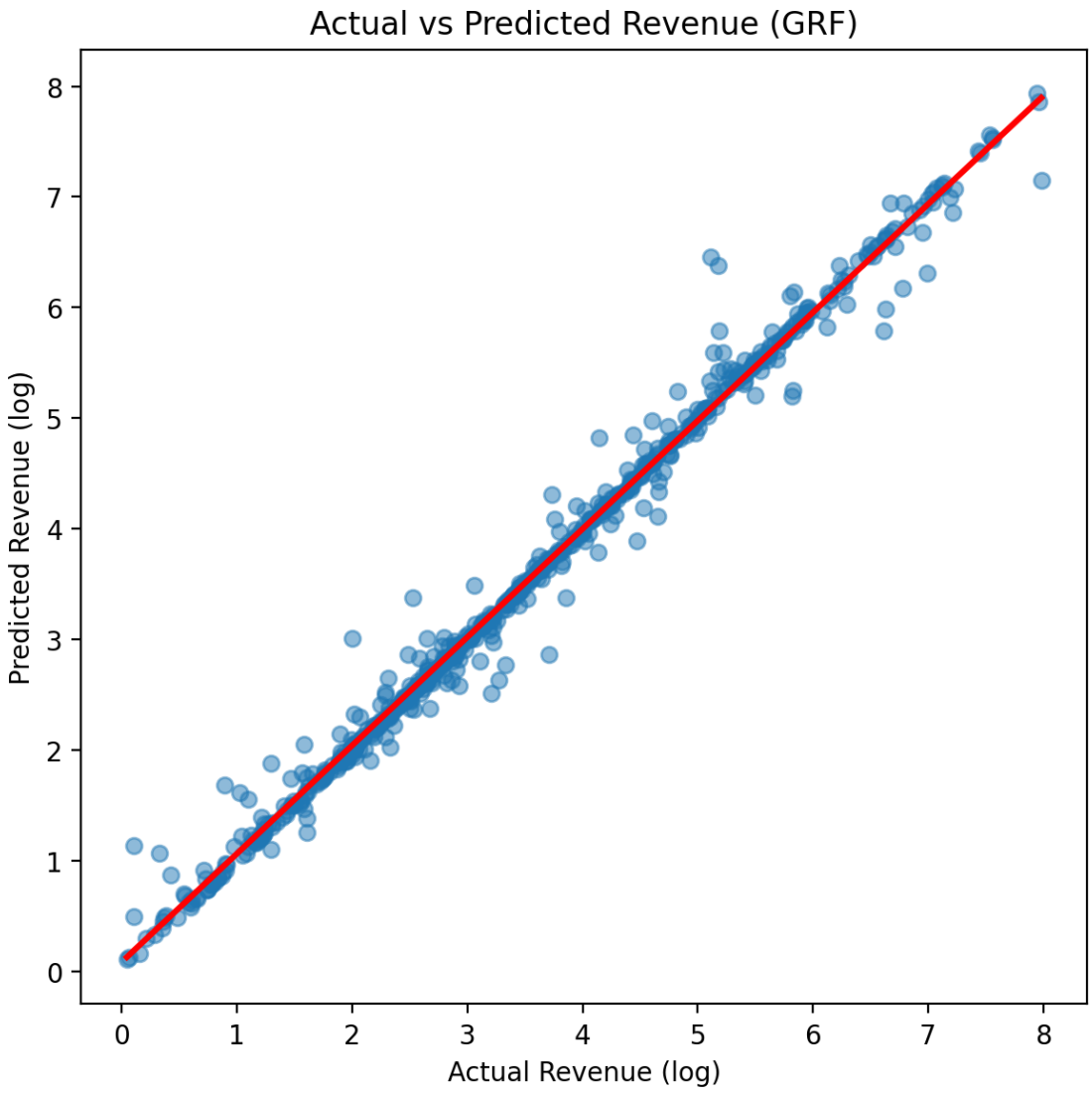
```
Processed 0/657 points
Processed 50/657 points
Processed 100/657 points
Processed 150/657 points
Processed 200/657 points
Processed 250/657 points
Processed 300/657 points
Processed 350/657 points
Processed 400/657 points
Processed 450/657 points
Processed 500/657 points
Processed 550/657 points
Processed 600/657 points
Processed 650/657 points
```

[34]:
```
# Evaluate log and original scale
valid_mask = gdf["grf_prediction"].notna() & gdf[target].notna()
y_true_log = gdf.loc[valid_mask, target]
y_pred_log = gdf.loc[valid_mask, "grf_prediction"]

# Metrics on log scale
rmse_log = np.sqrt(mean_squared_error(y_true_log, y_pred_log))
mae_log = mean_absolute_error(y_true_log, y_pred_log)
r2 = r2_score(y_true_log, y_pred_log)

# Metrics on original scale
```

```
y_true_orig = np.exp(y_true_log)
y_pred_orig = np.exp(y_pred_log)
rmse_orig = np.sqrt(mean_squared_error(y_true_orig, y_pred_orig))
mae_orig = mean_absolute_error(y_true_orig, y_pred_orig)

print("GRF Evaluation Metrics")
print("--------------------------------")
print(f"R² {r2:.3f}")
print(f"RMSE (log scale): {rmse_log:.3f}")
print(f"MAE  (log scale): {mae_log:.3f}")
print(f"RMSE (original scale): {rmse_orig:,.2f}")
print(f"MAE  (original scale): {mae_orig:,.2f}")
```

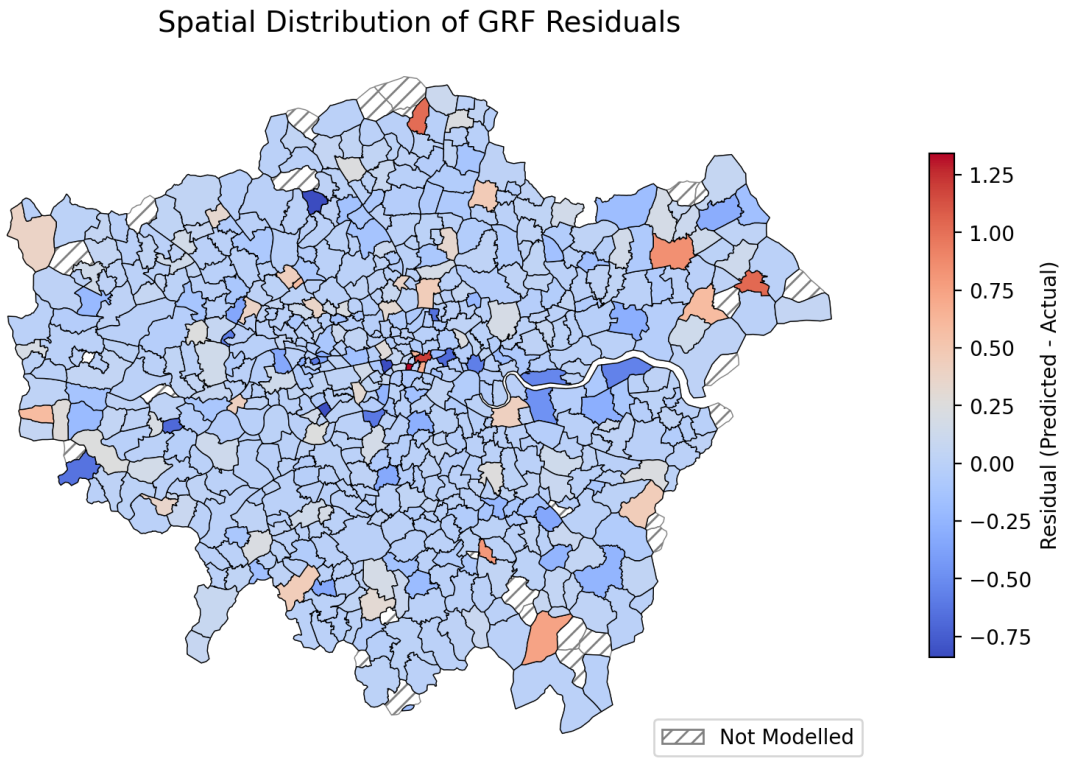
```
GRF Evaluation Metrics
--------------------------------
R² 0.989
RMSE (log scale): 0.187
MAE  (log scale): 0.087
RMSE (original scale): 85.91
MAE  (original scale): 16.61
```

[35]:
```
plt.figure(figsize=(6, 6))
sns.regplot(x=gdf.loc[valid_mask, target],
            y=gdf.loc[valid_mask, "grf_prediction"],
            line_kws={'color': 'red'},
            scatter_kws={'alpha': 0.5})

plt.xlabel("Actual Revenue (log)")
plt.ylabel("Predicted Revenue (log)")
plt.title("Actual vs Predicted Revenue (GRF)")
plt.grid(False)
plt.tight_layout()
plt.show()
```



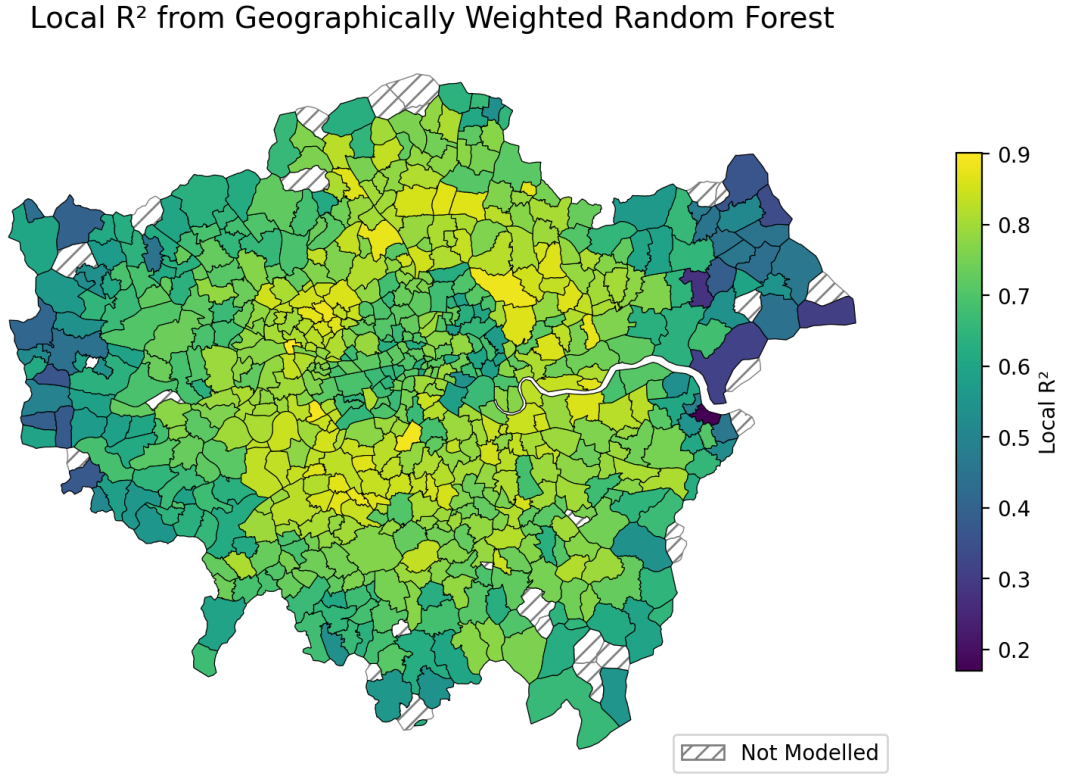
[36]:
```
gdf["grf_residual"] = gdf["grf_prediction"] - gdf[target]

# Filter valid predictions
gdf_valid = gdf[valid_mask]

# Create shared figure and axis
fig, ax = plt.subplots(figsize=(8, 6))

# --- Baselayer: all boundaries with hatch for unmodelled areas ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
```

```
    hatch='///',
    linewidth=0.5,
    zorder=0
)

# --- Residuals map ---
gdf_valid.plot(
    ax=ax,
    column="grf_residual",
    cmap="coolwarm",
    edgecolor='black',
    linewidth=0.5,
    legend=True,
    legend_kwds={
        'label': "Residual (Predicted - Actual)",
        'shrink': 0.6
    },
    zorder=1
)

# --- Final styling ---
hatch_patch = Patch(facecolor='white', edgecolor='gray', hatch='///',␣
 ↪label='Not Modelled')
ax.legend(handles=[hatch_patch], loc='lower right')

ax.set_title("Spatial Distribution of GRF Residuals", fontsize=14)
ax.axis('off')
plt.tight_layout()
plt.show()
```



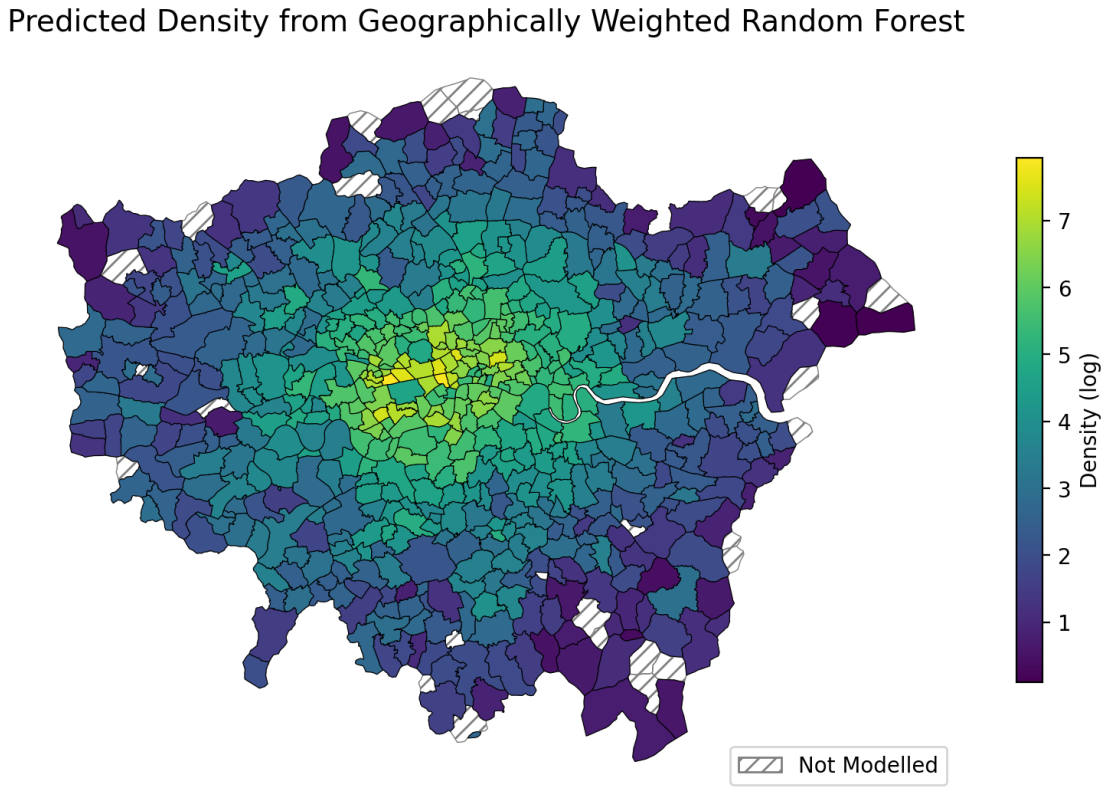
[37]:
```
# Filter to valid R² values
gdf_valid_r2 = gdf_valid[gdf_valid["grf_local_r2"].notna()]

# Plot
fig, ax = plt.subplots(figsize=(8, 6))

# --- Baselayer ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
    hatch='///',
    linewidth=0.5,
    zorder=0
)

# --- R² choropleth ---
gdf_valid_r2.plot(
    ax=ax,
    column="grf_local_r2",
    cmap="viridis",
    edgecolor="black",
```

```
    linewidth=0.4,
    legend=True,
    legend_kwds={
        'label': "Local R²",
        'shrink': 0.6,
        'orientation': "vertical"
    },
    zorder=1  # On top
)

hatch_patch = Patch(facecolor='white', edgecolor='gray', hatch='///',␣
↪label='Not Modelled')
ax.legend(handles=[hatch_patch], loc='lower right')
# --- Styling ---
ax.set_title("Local R² from Geographically Weighted Random Forest", fontsize=14)
ax.axis("off")
plt.tight_layout()
plt.show()
```



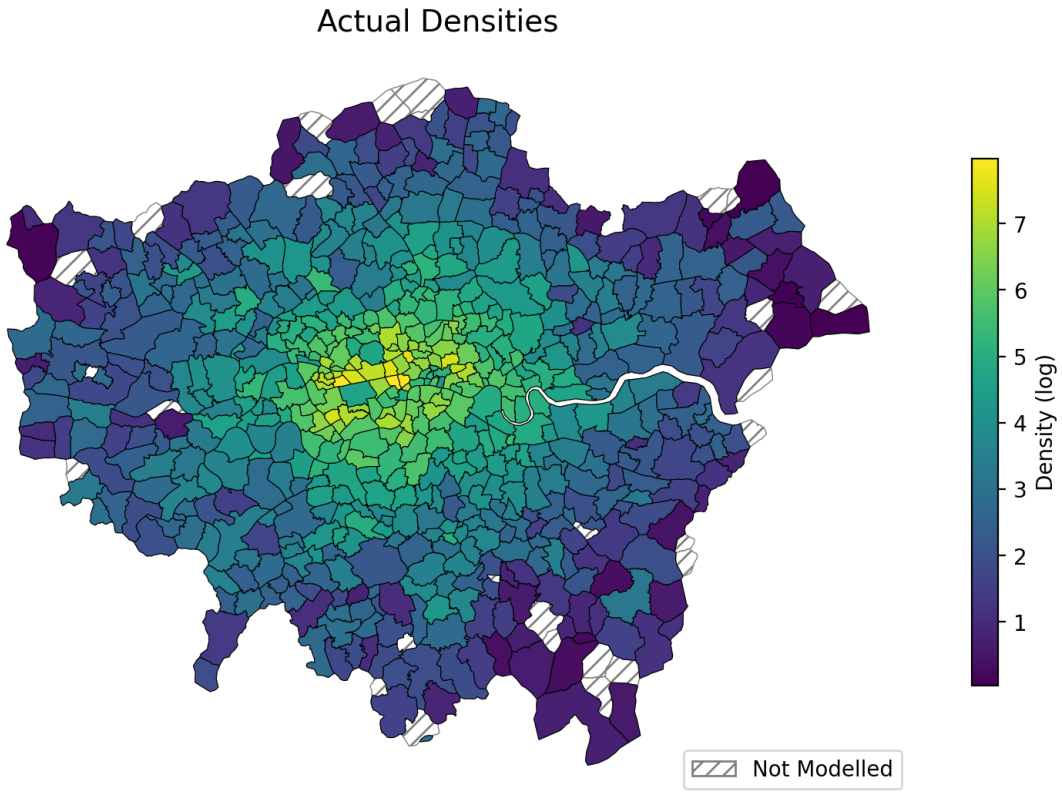
[44]:
```
# Plot
fig, ax = plt.subplots(figsize=(8, 6))
```

```
# --- Baselayer ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
    hatch='///',
    linewidth=0.5,
    zorder=0
)

# --- R² choropleth ---
gdf_valid.plot(
    ax=ax,
    column="grf_prediction",
    cmap="viridis",
    edgecolor="black",
    linewidth=0.4,
    legend=True,
    legend_kwds={
        'label': "Density (log)",
        'shrink': 0.6,
        'orientation': "vertical"
    },
    zorder=1  # On top
)

hatch_patch = Patch(facecolor='white', edgecolor='gray', hatch='///',␣
 ↪label='Not Modelled')
ax.legend(handles=[hatch_patch], loc='lower right')
# --- Styling ---
ax.set_title("Predicted Density from Geographically Weighted Random Forest",␣
 ↪fontsize=14)
ax.axis("off")
plt.tight_layout()
plt.show()
```



[46]:
```
# Set crs to match the baselayer
london_airbnb_tourism_fs = london_airbnb_tourism_fs.to_crs(epsg=27700)

# Plot
fig, ax = plt.subplots(figsize=(8, 6))

# --- Baselayer ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
    hatch='///',
    linewidth=0.5,
    zorder=0
)

# --- R² choropleth ---
london_airbnb_tourism_fs.plot(
    ax=ax,
    column="total_tourism_intensity_km2_log",
    cmap="viridis",
```

```
    edgecolor="black",
    linewidth=0.4,
    legend=True,
    legend_kwds={
        'label': "Density (log)",
        'shrink': 0.6,
        'orientation': "vertical"
    },
    zorder=1  # On top
)

hatch_patch = Patch(facecolor='white', edgecolor='gray', hatch='///',␣
↪label='Not Modelled')
ax.legend(handles=[hatch_patch], loc='lower right')
# --- Styling ---
ax.set_title("Actual Densities", fontsize=14)
ax.axis("off")
plt.tight_layout()
plt.show()
```



[ ]:
```
# --- Step 0: Prepare GeoDataFrame ---
gdf_valid = gdf[valid_mask].copy()  # Ensure gdf_valid is up to date

# --- Step 1: Define top N POI features to highlight ---
top_n = 10
top_pois = (
    gdf_valid["top_poi_clean"]
    .value_counts()
    .nlargest(top_n)
    .index.tolist()
)

# --- Step 2: Assign "Other Features" to less frequent ones ---
gdf_valid["top_poi_plot"] = gdf_valid["top_poi_clean"].where(
    gdf_valid["top_poi_clean"].isin(top_pois),
    "Other Features"
)

# --- Step 3: Build color map ---
unique_pois = sorted(set(top_pois))
unique_pois.append("Other Features")

colors = list(plt.cm.tab20.colors)
while len(colors) < len(unique_pois) - 1:
    colors += colors  # Ensure enough colors

poi_color_map = dict(zip(unique_pois[:-1], colors[:len(unique_pois) - 1]))
poi_color_map["Other Features"] = "white"

# --- Step 4: Create figure and axis ---
fig, ax = plt.subplots(figsize=(10, 8))

# --- Step 5: Plot full Edinburgh shapefile as base (hatch for non-modeled) ---
london_glx.plot(
    ax=ax,
    facecolor="none",
    edgecolor="grey",
    hatch="///",
    linewidth=0.5,
    zorder=0
)

# --- Step 6: Plot colored GRF results ---
for poi, color in poi_color_map.items():
    subset = gdf_valid[gdf_valid["top_poi_plot"] == poi]
    subset.plot(ax=ax, color=color, edgecolor="black", linewidth=0.2, zorder=1)
```

```
# --- Step 7: Add legend ---
legend_elements = [
    Patch(facecolor=poi_color_map[poi], edgecolor='black', label=poi)
    for poi in unique_pois
]

# Add legend entry for hatch
legend_elements.append(
    Patch(facecolor='white', edgecolor='gray', hatch='///', label='Not␣
 ↪Modelled')
)

ax.legend(
    handles=legend_elements,
    title="Top POI Category",
    bbox_to_anchor=(1.05, 1), loc="upper left", borderaxespad=0.
)

# --- Step 8: Finalize map ---
plt.title("Top POI Importance per Location", fontsize=14)
plt.axis("off")
plt.tight_layout()
plt.show()
```