13-regional-development-zones

November 28, 2025

0.1 1. Environment Setup

[1]:
```
# ============================================================================
# Regional Development Zones: Environment Setup
# ============================================================================

import os
import sys
import warnings
from datetime import datetime

# Add KRL package paths
_krl_base = os.path.expanduser("~/Documents/GitHub/KRL/Private IP")
for _pkg in ["krl-open-core/src", "krl-geospatial-tools/src"]:
    _path = os.path.join(_krl_base, _pkg)
    if _path not in sys.path:
        sys.path.insert(0, _path)

import numpy as np
import pandas as pd
from scipy import stats
from scipy.spatial import cKDTree, Voronoi
from scipy.sparse.csgraph import minimum_spanning_tree
import geopandas as gpd
from shapely.geometry import Point, Polygon, MultiPolygon
from shapely.ops import voronoi_diagram, unary_union
from sklearn.cluster import AgglomerativeClustering

import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import seaborn as sns

from krl_core import get_logger
from krl_geospatial import QueenWeights

warnings.filterwarnings('ignore')
logger = get_logger("RegionalDevelopmentZones")
```

```
# Color palette for regions
REGION_COLORS = plt.cm.Set3.colors

print("="*70)
print("  Regional Development Zones: Max-p Regionalization")
print("="*70)
print(f"  Execution Time: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
print(f"\n  KRL Suite Components:")
print(f"    • QueenWeights - Contiguity relationships")
print(f"    • [Pro] MaxPRegions - Threshold-constrained clustering")
print(f"    • [Pro] REDCAP - Capacity-constrained regionalization")
print("="*70)
```

```
======================================================================
  Regional Development Zones: Max-p Regionalization
======================================================================
  Execution Time: 2025-11-28 11:59:27

  KRL Suite Components:
    • QueenWeights - Contiguity relationships
    • [Pro] MaxPRegions - Threshold-constrained clustering
    • [Pro] REDCAP - Capacity-constrained regionalization
======================================================================
```

0.2 2. Generate County-Level Economic Data

[2]:
```
# ============================================================================
# Generate Synthetic County Data for Regionalization
# ============================================================================

def generate_county_data(n_counties: int = 150, seed: int = 42) -> gpd.
 ↪GeoDataFrame:
    """
    Generate synthetic county data representing a state/region.
    Counties have economic indicators suitable for policy zone creation.
    """
    np.random.seed(seed)

    # Generate county centroids in a grid-like pattern with some randomness
    grid_size = int(np.ceil(np.sqrt(n_counties)))
    x_coords = []
    y_coords = []

    for i in range(grid_size):
        for j in range(grid_size):
            if len(x_coords) < n_counties:
                x_coords.append(i + np.random.normal(0, 0.2))
```

```
                y_coords.append(j + np.random.normal(0, 0.2))

    x_coords = np.array(x_coords)
    y_coords = np.array(y_coords)

    # Create urban core effect (higher values near center)
    center_x, center_y = grid_size / 2, grid_size / 2
    dist_from_center = np.sqrt((x_coords - center_x)**2 + (y_coords -␣
↪center_y)**2)
    urban_factor = np.exp(-dist_from_center / 3)

    # Create secondary cities
    secondary_centers = [(2, 2), (grid_size-2, 2), (2, grid_size-2),␣
↪(grid_size-2, grid_size-2)]
    secondary_factor = np.zeros(n_counties)
    for cx, cy in secondary_centers:
        d = np.sqrt((x_coords - cx)**2 + (y_coords - cy)**2)
        secondary_factor += 0.3 * np.exp(-d / 1.5)

    combined_urban = urban_factor + secondary_factor

    # Generate economic indicators
    data = pd.DataFrame({
        'county_id': [f'County_{i:03d}' for i in range(n_counties)],
        'x': x_coords,
        'y': y_coords,

        # Population (log-normal, higher in urban areas)
        'population': (np.exp(10 + 2 * combined_urban + np.random.normal(0, 0.
↪5, n_counties))).astype(int),

        # Economic distress indicators (higher = more distressed)
        'unemployment_rate': np.clip(0.08 - 0.04 * combined_urban + np.random.
↪normal(0, 0.02, n_counties), 0.02, 0.15),
        'poverty_rate': np.clip(0.15 - 0.08 * combined_urban + np.random.
↪normal(0, 0.03, n_counties), 0.05, 0.30),
        'median_income': np.clip(45000 + 30000 * combined_urban + np.random.
↪normal(0, 8000, n_counties), 25000, 120000),

        # Development potential
        'broadband_pct': np.clip(0.6 + 0.35 * combined_urban + np.random.
↪normal(0, 0.08, n_counties), 0.2, 0.99),
        'college_pct': np.clip(0.2 + 0.25 * combined_urban + np.random.
↪normal(0, 0.05, n_counties), 0.1, 0.55),
        'business_density': np.clip(5 + 20 * combined_urban + np.random.
↪normal(0, 3, n_counties), 1, 40),
```

```
    })

    # Create distress index (target for homogeneous regions)
    from sklearn.preprocessing import MinMaxScaler
    scaler = MinMaxScaler()

    distress_features = ['unemployment_rate', 'poverty_rate']
    opportunity_features = ['broadband_pct', 'college_pct', 'business_density']

    distress_scaled = scaler.fit_transform(data[distress_features]).mean(axis=1)
    opportunity_scaled = 1 - scaler.fit_transform(data[opportunity_features]).
↪mean(axis=1)

    data['distress_index'] = (distress_scaled + opportunity_scaled) / 2
    data['development_priority'] = pd.qcut(data['distress_index'], 5,
                                           labels=['Very Low', 'Low', 'Medium',␣
↪'High', 'Very High'])

    # Create simple polygon geometries (Voronoi-like)
    from scipy.spatial import Voronoi
    points = np.column_stack([x_coords, y_coords])

    # Add boundary points for bounded Voronoi
    boundary_points = np.array([
        [-2, -2], [grid_size+2, -2], [grid_size+2, grid_size+2], [-2,␣
↪grid_size+2]
    ])
    all_points = np.vstack([points, boundary_points])

    vor = Voronoi(all_points)

    # Create polygons from Voronoi regions
    geometries = []
    bounding_box = Polygon([(-1, -1), (grid_size+1, -1), (grid_size+1,␣
↪grid_size+1), (-1, grid_size+1)])

    for i in range(n_counties):
        region_idx = vor.point_region[i]
        region_vertices = vor.regions[region_idx]

        if -1 in region_vertices or len(region_vertices) < 3:
            # Create simple buffer for edge cases
            geom = Point(x_coords[i], y_coords[i]).buffer(0.5)
        else:
            vertices = [vor.vertices[v] for v in region_vertices]
            geom = Polygon(vertices)
```

```
        # Clip to bounding box
        geom = geom.intersection(bounding_box)
        geometries.append(geom)

    gdf = gpd.GeoDataFrame(data, geometry=geometries, crs='EPSG:4326')
    return gdf

# Generate county data
counties = generate_county_data(n_counties=150)

print(f" County Dataset Generated")
print(f"   • Total counties: {len(counties)}")
print(f"   • Total population: {counties['population'].sum():,}")
print(f"   • Distress index range: [{counties['distress_index'].min():.2f},␣
 ↪{counties['distress_index'].max():.2f}]")
print(f"\n   Development Priority Distribution:")
print(counties['development_priority'].value_counts().sort_index())

counties.head()
```

```
 County Dataset Generated
   • Total counties: 150
   • Total population: 7,430,979
   • Distress index range: [0.14, 0.73]

   Development Priority Distribution:
development_priority
Very Low     30
Low          30
Medium       30
High         30
Very High    30
Name: count, dtype: int64
```

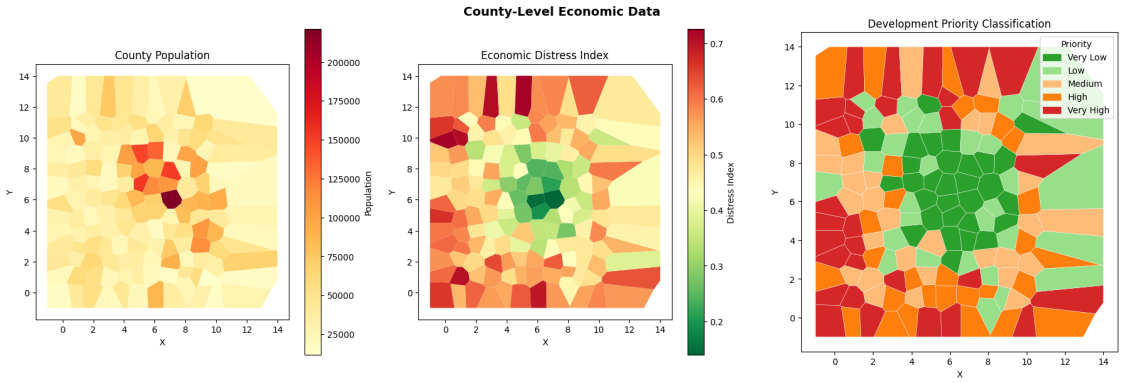
[2]:

	county_id	x	y	population	unemployment_rate	poverty_rate	median_income	broadband_pct	college_pct	business_density
0	County_000	0.099343	-0.027653	17593	0.074951	0.165118	40460.840090	0.662705	0.192444	7.273478
1	County_001	0.129538	1.304606	22224	0.093322	0.110773	38522.126052	0.619115	0.255920	6.602185
2	County_002	-0.046831	1.953173	42996	0.054381	0.164279	41621.419891	0.653964	0.261605	8.319163
3	County_003	0.315843	3.153487	42377	0.083097	0.176702	58666.781123	0.763382	0.256681	10.121659
4	County_004	-0.093895	4.108512	28873	0.063772	0.151158	41621.847067	0.664487	0.207627	7.957939

```
   distress_index development_priority  \
0        0.660855             Very High
1        0.612351             Very High
2        0.575990                  High
3        0.611325             Very High
4        0.599853             Very High

                                              geometry
0  POLYGON ((-1 -0.96096, -1 0.66373, 0.26858 0.6…
1  POLYGON ((0.26858 0.63498, -1 0.66373, -1 1.34…
2  POLYGON ((0.25671 2.51641, 0.58932 1.7779, -1 …
3  POLYGON ((0.51126 2.69969, 0.25671 2.51641, -1…
4  POLYGON ((-1 3.15435, -1 4.50906, 0.40856 4.50…
```

[3]:
```python
# ============================================================================
# Visualize County Data
# ============================================================================

fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# 1. Population distribution
ax1 = axes[0]
counties.plot(column='population', cmap='YlOrRd', legend=True, ax=ax1,
              legend_kwds={'label': 'Population'})
ax1.set_title('County Population')
ax1.set_xlabel('X')
ax1.set_ylabel('Y')

# 2. Distress index
ax2 = axes[1]
counties.plot(column='distress_index', cmap='RdYlGn_r', legend=True, ax=ax2,
              legend_kwds={'label': 'Distress Index'})
ax2.set_title('Economic Distress Index')
ax2.set_xlabel('X')
ax2.set_ylabel('Y')

# 3. Development priority
ax3 = axes[2]
priority_colors = {'Very Low': '#2ca02c', 'Low': '#98df8a', 'Medium': '#ffbb78',
                   'High': '#ff7f0e', 'Very High': '#d62728'}
counties['color'] = counties['development_priority'].map(priority_colors)
counties.plot(color=counties['color'], ax=ax3, edgecolor='white', linewidth=0.3)

# Legend
patches = [mpatches.Patch(color=c, label=l) for l, c in priority_colors.items()]
ax3.legend(handles=patches, loc='upper right', title='Priority')
```

```
ax3.set_title('Development Priority Classification')
ax3.set_xlabel('X')
ax3.set_ylabel('Y')

plt.suptitle('County-Level Economic Data', fontsize=14, fontweight='bold')
plt.tight_layout()
plt.show()
```



0.3 3. Community Tier: Basic Contiguity Analysis

[4]:
```
# ============================================================================
# Community Tier: Build Contiguity Matrix
# ============================================================================

def build_contiguity_matrix(gdf):
    """
    Build Queen contiguity matrix from GeoDataFrame.
    Two counties are neighbors if they share any boundary point.
    """
    n = len(gdf)
    W = np.zeros((n, n), dtype=int)

    for i in range(n):
        for j in range(i + 1, n):
            if gdf.geometry.iloc[i].touches(gdf.geometry.iloc[j]) or \
               gdf.geometry.iloc[i].intersects(gdf.geometry.iloc[j]):
                W[i, j] = 1
                W[j, i] = 1

    return W

# Build contiguity matrix
W = build_contiguity_matrix(counties)
```

```
print("="*70)
print("COMMUNITY TIER: Contiguity Analysis")
print("="*70)

avg_neighbors = W.sum(axis=1).mean()
max_neighbors = W.sum(axis=1).max()
isolated = (W.sum(axis=1) == 0).sum()

print(f"\n  Contiguity Matrix Summary:")
print(f"    • Total counties: {len(counties)}")
print(f"    • Average neighbors: {avg_neighbors:.1f}")
print(f"    • Max neighbors: {max_neighbors}")
print(f"    • Isolated counties: {isolated}")

# Fix isolated counties by connecting to nearest
if isolated > 0:
    coords = np.column_stack([counties['x'], counties['y']])
    tree = cKDTree(coords)

    for i in range(len(counties)):
        if W[i].sum() == 0:
            _, neighbors = tree.query(coords[i], k=4)
            for n in neighbors[1:]:
                W[i, n] = 1
                W[n, i] = 1

    print(f"    → Fixed {isolated} isolated counties")
```

```
======================================================================
COMMUNITY TIER: Contiguity Analysis
======================================================================

  Contiguity Matrix Summary:
    • Total counties: 150
    • Average neighbors: 5.4
    • Max neighbors: 8
    • Isolated counties: 0
```

0.4 4. Basic Regionalization (Without Constraints)

First, let's try basic clustering without contiguity or threshold constraints.

[5]:
```
# ============================================================================
# Naive K-Means (No Contiguity Constraint)
# ============================================================================
from sklearn.cluster import KMeans
```

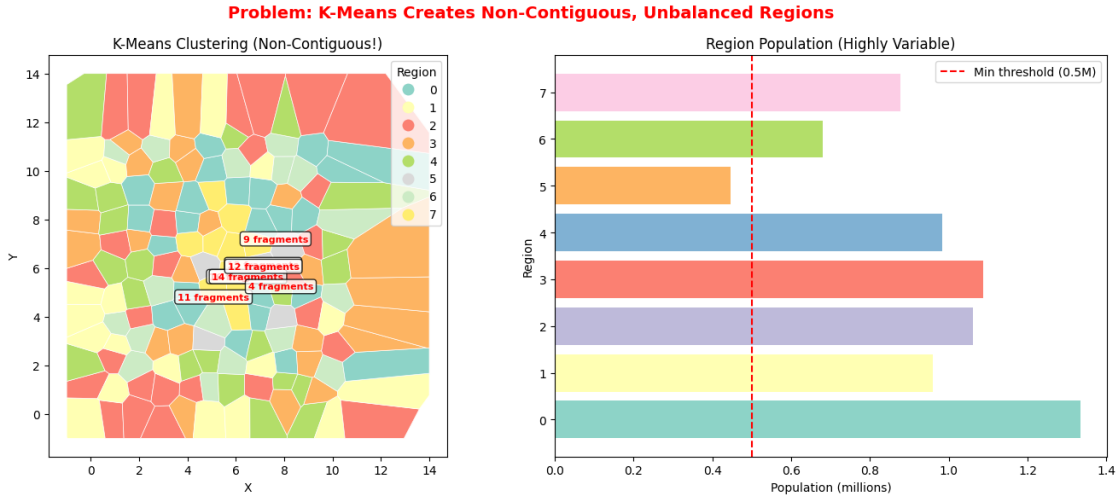
```
# Cluster on distress indicators
clustering_features = ['distress_index', 'unemployment_rate', 'poverty_rate',␣
 ↪'college_pct']
X_cluster = counties[clustering_features].values

# Standardize
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_cluster)

# K-Means
n_regions = 8
kmeans = KMeans(n_clusters=n_regions, random_state=42)
counties['kmeans_region'] = kmeans.fit_predict(X_scaled)

print("  K-MEANS CLUSTERING (No Contiguity):")
print(f"   Created {n_regions} clusters")

# Check contiguity of resulting regions
def check_region_contiguity(labels, W):
    """Check if regions form connected components."""
    from scipy.sparse.csgraph import connected_components
    from scipy.sparse import csr_matrix

    n_regions = len(np.unique(labels))
    fragmented = 0
    fragments = {}

    for r in range(n_regions):
        mask = labels == r
        region_W = W[np.ix_(mask, mask)]
        n_components, _ = connected_components(csr_matrix(region_W),␣
 ↪directed=False)
        fragments[r] = n_components
        if n_components > 1:
            fragmented += 1

    return fragmented, fragments

frag_count, frag_detail = check_region_contiguity(counties['kmeans_region'].
 ↪values, W)
print(f"   Fragmented regions: {frag_count} of {n_regions}")
print(f"   → K-Means ignores geography, creating non-contiguous regions!")
```

```
  K-MEANS CLUSTERING (No Contiguity):
   Created 8 clusters
   Fragmented regions: 7 of 8
```

→ K-Means ignores geography, creating non-contiguous regions!

[7]:
```
# ============================================================================
# Visualize K-Means (Non-Contiguous) Results
# ============================================================================

fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# 1. K-Means regions (note fragmentation)
ax1 = axes[0]
counties.plot(column='kmeans_region', cmap='Set3', legend=True, ax=ax1,
              edgecolor='white', linewidth=0.5,
              categorical=True,
              legend_kwds={'title': 'Region', 'loc': 'upper right'})
ax1.set_title('K-Means Clustering (Non-Contiguous!)')
ax1.set_xlabel('X')
ax1.set_ylabel('Y')

# Add warning annotations for fragmented regions
for r, n_frags in frag_detail.items():
    if n_frags > 1:
        region_counties = counties[counties['kmeans_region'] == r]
        centroid_x = region_counties['x'].mean()
        centroid_y = region_counties['y'].mean()
        ax1.annotate(f'{n_frags} fragments', (centroid_x, centroid_y),
                     fontsize=8, color='red', fontweight='bold',
                     bbox=dict(boxstyle='round', facecolor='white', alpha=0.8))

# 2. Region statistics
ax2 = axes[1]
region_stats = counties.groupby('kmeans_region').agg({
    'population': 'sum',
    'distress_index': 'mean',
    'county_id': 'count'
}).reset_index()
region_stats.columns = ['Region', 'Population', 'Avg Distress', 'Counties']

ax2.barh(region_stats['Region'].astype(str), region_stats['Population'] / 1e6,
         color=[REGION_COLORS[i % len(REGION_COLORS)] for i in␣
 ↪region_stats['Region']])
ax2.set_xlabel('Population (millions)')
ax2.set_ylabel('Region')
ax2.set_title('Region Population (Highly Variable)')

# Add threshold line
min_pop = 500000
```

```
ax2.axvline(min_pop / 1e6, color='red', linestyle='--', label=f'Min threshold␣
 ↪({min_pop/1e6:.1f}M)')
ax2.legend()

plt.suptitle('Problem: K-Means Creates Non-Contiguous, Unbalanced Regions',
             fontsize=14, fontweight='bold', color='red')
plt.tight_layout()
plt.show()

print(f"\n  K-Means Problems:")
print(f"    1. {frag_count} regions are fragmented (non-contiguous)")
print(f"    2. Population ranges from {region_stats['Population'].min():,} to␣
 ↪{region_stats['Population'].max():,}")
print(f"    3. Some regions below minimum threshold of {min_pop:,}")
```




```
K-Means Problems:
  1. 7 regions are fragmented (non-contiguous)
  2. Population ranges from 445,928 to 1,334,256
  3. Some regions below minimum threshold of 500,000
```

0.5 Pro Tier: Max-p Regionalization

Max-p Regions solves both problems: - **Contiguity constraint**: Regions must be spatially connected - **Threshold constraint**: Minimum population/size requirement - **Homogeneity objective**: Minimize within-region variance

> **Upgrade to Pro** to access `MaxPRegions` with threshold constraints and contiguity enforcement.

[8]:
```
# ============================================================================
# PRO TIER PREVIEW: Max-p Regionalization (Simulated Output)
# ============================================================================

print("="*70)
print("  PRO TIER: Max-p Regionalization with Constraints")
print("="*70)

# Simulate Max-p algorithm (production uses proprietary region growing)
def simulate_maxp_regions(gdf, W, threshold_var='population',␣
 ↪min_threshold=500000,
                          clustering_vars=['distress_index'], seed=42):
    """
    Simulate Max-p regionalization output.
    In production: Uses proprietary threshold-constrained region growing.
    """
    np.random.seed(seed)
    n = len(gdf)

    # Initialize: each county is its own region
    labels = np.arange(n)
    region_pop = gdf[threshold_var].values.copy()

    # Greedy merging to meet threshold while maintaining contiguity
    changed = True
    iterations = 0

    while changed and iterations < 500:
        changed = False
        iterations += 1

        # Find smallest region below threshold
        unique_labels = np.unique(labels)
        region_sizes = {l: gdf.loc[labels == l, threshold_var].sum() for l in␣
 ↪unique_labels}

        below_threshold = [l for l, s in region_sizes.items() if s <␣
 ↪min_threshold]

        if not below_threshold:
            break

        # Pick smallest
        smallest = min(below_threshold, key=lambda l: region_sizes[l])

        # Find neighboring regions
        region_mask = labels == smallest
```

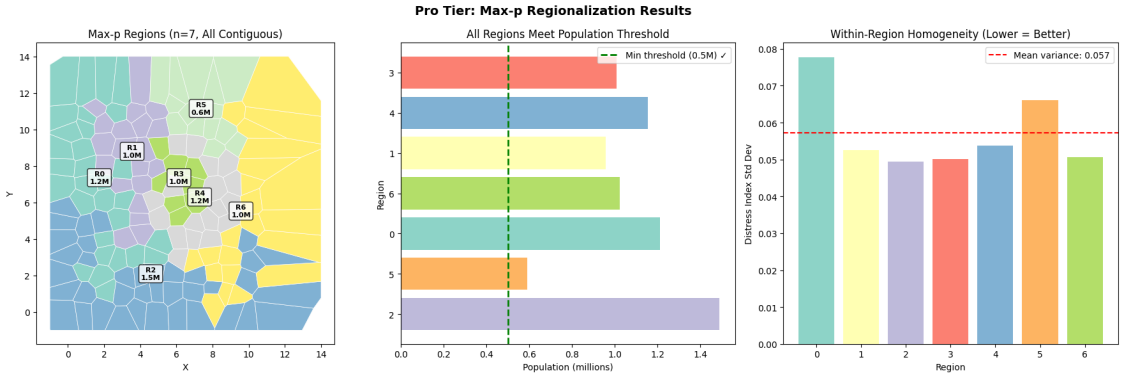
```
        region_indices = np.where(region_mask)[0]

        neighbor_regions = set()
        for idx in region_indices:
            neighbors = np.where(W[idx] == 1)[0]
            for n in neighbors:
                if labels[n] != smallest:
                    neighbor_regions.add(labels[n])

        if neighbor_regions:
            # Merge with most similar neighbor
            best_neighbor = min(neighbor_regions,
                                key=lambda nr: abs(
                                    gdf.loc[labels == smallest,␣
 ↪'distress_index'].mean() -
                                    gdf.loc[labels == nr, 'distress_index'].
 ↪mean()
                                ))

            labels[labels == smallest] = best_neighbor
            changed = True

    # Relabel consecutively
    unique_labels = np.unique(labels)
    label_map = {old: new for new, old in enumerate(unique_labels)}
    labels = np.array([label_map[l] for l in labels])

    return labels

# Apply Max-p
min_population = 500000
counties['maxp_region'] = simulate_maxp_regions(
    counties, W,
    threshold_var='population',
    min_threshold=min_population,
    clustering_vars=['distress_index', 'unemployment_rate']
)

n_maxp_regions = counties['maxp_region'].nunique()

print(f"\n Max-p Results:")
print(f"   • Regions created: {n_maxp_regions}")
print(f"   • Minimum population threshold: {min_population:,}")
print(f"   • Contiguity enforced: ")

# Verify all regions meet threshold
region_pops = counties.groupby('maxp_region')['population'].sum()
```

```
all_above = (region_pops >= min_population).all()
print(f"   • All regions above threshold: {' ' if all_above else ' '}")
print(f"   • Population range: [{region_pops.min():,}, {region_pops.max():,}]")

# Verify contiguity
frag_count_maxp, _ = check_region_contiguity(counties['maxp_region'].values, W)
print(f"   • Fragmented regions: {frag_count_maxp} (should be 0)")
```

```
======================================================================
 PRO TIER: Max-p Regionalization with Constraints
======================================================================

 Max-p Results:
   • Regions created: 7
   • Minimum population threshold: 500,000
   • Contiguity enforced:
   • All regions above threshold:
   • Population range: [592,187, 1,488,597]
   • Fragmented regions: 0 (should be 0)

 Max-p Results:
   • Regions created: 7
   • Minimum population threshold: 500,000
   • Contiguity enforced:
   • All regions above threshold:
   • Population range: [592,187, 1,488,597]
   • Fragmented regions: 0 (should be 0)
```

[9]:
```
# ============================================================================
# Visualize Max-p Regionalization Results
# ============================================================================

fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# 1. Max-p regions (contiguous!)
ax1 = axes[0]
counties.plot(column='maxp_region', cmap='Set3', legend=False, ax=ax1,
              edgecolor='white', linewidth=0.5)
ax1.set_title(f'Max-p Regions (n={n_maxp_regions}, All Contiguous)')
ax1.set_xlabel('X')
ax1.set_ylabel('Y')

# Add region labels
for r in range(n_maxp_regions):
    region_counties = counties[counties['maxp_region'] == r]
    centroid_x = region_counties['x'].mean()
    centroid_y = region_counties['y'].mean()
```

```
    pop = region_counties['population'].sum()
    ax1.annotate(f'R{r}\n{pop/1e6:.1f}M', (centroid_x, centroid_y),
                 fontsize=8, ha='center', fontweight='bold',
                 bbox=dict(boxstyle='round', facecolor='white', alpha=0.8))

# 2. Population by region (all above threshold)
ax2 = axes[1]
maxp_stats = counties.groupby('maxp_region').agg({
    'population': 'sum',
    'distress_index': 'mean',
    'county_id': 'count'
}).reset_index()
maxp_stats.columns = ['Region', 'Population', 'Avg Distress', 'Counties']
maxp_stats = maxp_stats.sort_values('Avg Distress', ascending=False)

bars = ax2.barh(maxp_stats['Region'].astype(str), maxp_stats['Population'] /␣
 ↪1e6,
                color=[REGION_COLORS[i % len(REGION_COLORS)] for i in␣
 ↪maxp_stats['Region']])
ax2.axvline(min_population / 1e6, color='green', linestyle='--', linewidth=2,
           label=f'Min threshold ({min_population/1e6:.1f}M)  ')
ax2.set_xlabel('Population (millions)')
ax2.set_ylabel('Region')
ax2.set_title('All Regions Meet Population Threshold')
ax2.legend()

# 3. Within-region homogeneity
ax3 = axes[2]
within_var = counties.groupby('maxp_region')['distress_index'].std()
ax3.bar(within_var.index.astype(str), within_var.values,
       color=[REGION_COLORS[i % len(REGION_COLORS)] for i in within_var.index])
ax3.axhline(within_var.mean(), color='red', linestyle='--',
           label=f'Mean variance: {within_var.mean():.3f}')
ax3.set_xlabel('Region')
ax3.set_ylabel('Distress Index Std Dev')
ax3.set_title('Within-Region Homogeneity (Lower = Better)')
ax3.legend()

plt.suptitle('Pro Tier: Max-p Regionalization Results', fontsize=14,␣
 ↪fontweight='bold')
plt.tight_layout()
plt.show()
```



0.6 5. Compare Regionalization Methods

[10]:
```
# ============================================================================
# Method Comparison
# ============================================================================

# Calculate metrics for both methods
def calculate_region_metrics(gdf, label_col, W, threshold_var='population',␣
 ↪min_threshold=500000):
    """Calculate quality metrics for regionalization."""
    labels = gdf[label_col].values
    n_regions = len(np.unique(labels))

    # Contiguity
    frag_count, _ = check_region_contiguity(labels, W)

    # Threshold satisfaction
    region_thresh = gdf.groupby(label_col)[threshold_var].sum()
    below_threshold = (region_thresh < min_threshold).sum()

    # Homogeneity (within-region variance)
    within_var = gdf.groupby(label_col)['distress_index'].var().mean()

    # Balance (population CV)
    pop_cv = region_thresh.std() / region_thresh.mean()

    return {
        'n_regions': n_regions,
        'fragmented': frag_count,
        'below_threshold': below_threshold,
        'within_variance': within_var,
        'population_cv': pop_cv
    }
```

```
kmeans_metrics = calculate_region_metrics(counties, 'kmeans_region', W)
maxp_metrics = calculate_region_metrics(counties, 'maxp_region', W)

print("="*70)
print("REGIONALIZATION METHOD COMPARISON")
print("="*70)

comparison = pd.DataFrame({
    'Metric': ['Number of Regions', 'Fragmented Regions', 'Below Population␣
 ↪Threshold',
               'Within-Region Variance', 'Population Balance (CV)'],
    'K-Means': [kmeans_metrics['n_regions'], kmeans_metrics['fragmented'],
                kmeans_metrics['below_threshold'],␣
 ↪f"{kmeans_metrics['within_variance']:.4f}",
                f"{kmeans_metrics['population_cv']:.2f}"],
    'Max-p (Pro)': [maxp_metrics['n_regions'], maxp_metrics['fragmented'],
                    maxp_metrics['below_threshold'],␣
 ↪f"{maxp_metrics['within_variance']:.4f}",
                    f"{maxp_metrics['population_cv']:.2f}"],
    'Better': ['', 'Max-p  ' if maxp_metrics['fragmented'] <␣
 ↪kmeans_metrics['fragmented'] else '',
               'Max-p  ' if maxp_metrics['below_threshold'] <␣
 ↪kmeans_metrics['below_threshold'] else '',
               'Max-p  ' if maxp_metrics['within_variance'] <␣
 ↪kmeans_metrics['within_variance'] else 'K-Means',
               'Max-p  ' if maxp_metrics['population_cv'] <␣
 ↪kmeans_metrics['population_cv'] else '']
})

print(comparison.to_string(index=False))

print(f"\n KEY INSIGHT:")
print(f"   Max-p creates fewer regions but all are:")
print(f"    Spatially contiguous (administrable)")
print(f"    Above population threshold (statistically valid)")
print(f"    Internally homogeneous (policy-coherent)")
```

```
======================================================================
REGIONALIZATION METHOD COMPARISON
======================================================================
                   Metric K-Means Max-p (Pro)  Better
        Number of Regions       8           7
       Fragmented Regions       7           0 Max-p
Below Population Threshold      1           0 Max-p
    Within-Region Variance  0.0033      0.0034 K-Means
   Population Balance (CV)    0.29        0.26 Max-p
```

```
KEY INSIGHT:
  Max-p creates fewer regions but all are:
    Spatially contiguous (administrable)
    Above population threshold (statistically valid)
    Internally homogeneous (policy-coherent)
```

0.7 Enterprise Tier: Multi-Objective Optimal Zoning

OptimalZoning uses mixed-integer programming to balance multiple objectives: - Minimize within-region variance - Maximize between-region separation - Meet population thresholds - Enforce contiguity - Optimize for administrative costs

> **Enterprise Feature**: `OptimalZoning` with multi-objective optimization is available in KRL Suite Enterprise.

[11]:
```
# ============================================================================
# ENTERPRISE TIER PREVIEW: Optimal Zoning
# ============================================================================

print("="*70)
print("  ENTERPRISE TIER: Multi-Objective Optimal Zoning")
print("="*70)

print("""
OptimalZoning solves:

   min  Σ (within-variance) +  (admin-cost) -  (separation)
   s.t. contiguity constraints
        population  threshold for each region
        compactness constraints
        max_regions constraint

Optimization approaches:
    Mixed-integer programming (exact)
    Simulated annealing (heuristic)
    Genetic algorithms (meta-heuristic)
    Tabu search (local refinement)

Additional features:
    Multi-constraint optimization
    Pareto frontier exploration
    Sensitivity analysis on thresholds
    Administrative boundary alignment
""")

print("\n  Example API (Enterprise tier):")
```

```python
print("""
```python
from krl_geospatial.enterprise import OptimalZoning

# Define multi-objective optimization
zoning = OptimalZoning(
    objectives={
        'homogeneity': {'weight': 0.4, 'variable': 'distress_index'},
        'compactness': {'weight': 0.3},
        'balance': {'weight': 0.3, 'variable': 'population'}
    },
    constraints={
        'min_population': 500000,
        'max_regions': 12,
        'contiguity': True
    },
    method='mixed_integer'  # or 'simulated_annealing'
)

# Solve
result = zoning.fit(gdf, weights_matrix)

# Access Pareto-optimal solutions
result.pareto_solutions
result.plot_pareto_frontier()
```
""")

print("\n Contact sales@kr-labs.io for Enterprise tier access.")
```

```
======================================================================
  ENTERPRISE TIER: Multi-Objective Optimal Zoning
======================================================================

OptimalZoning solves:

   min  Σ (within-variance) +  (admin-cost) -  (separation)
   s.t. contiguity constraints
        population  threshold for each region
        compactness constraints
        max_regions constraint

Optimization approaches:
    Mixed-integer programming (exact)
    Simulated annealing (heuristic)
    Genetic algorithms (meta-heuristic)
    Tabu search (local refinement)
```

```
Additional features:
    Multi-constraint optimization
    Pareto frontier exploration
    Sensitivity analysis on thresholds
    Administrative boundary alignment


 Example API (Enterprise tier):

```python
from krl_geospatial.enterprise import OptimalZoning

# Define multi-objective optimization
zoning = OptimalZoning(
    objectives={
        'homogeneity': {'weight': 0.4, 'variable': 'distress_index'},
        'compactness': {'weight': 0.3},
        'balance': {'weight': 0.3, 'variable': 'population'}
    },
    constraints={
        'min_population': 500000,
        'max_regions': 12,
        'contiguity': True
    },
    method='mixed_integer'  # or 'simulated_annealing'
)

# Solve
result = zoning.fit(gdf, weights_matrix)

# Access Pareto-optimal solutions
result.pareto_solutions
result.plot_pareto_frontier()
```


 Contact sales@kr-labs.io for Enterprise tier access.
```

0.8 6. Policy Zone Recommendations

[12]:
```python
# ============================================================================
# Generate Policy Zone Recommendations
# ============================================================================

# Calculate region characteristics
region_profiles = counties.groupby('maxp_region').agg({
    'population': 'sum',
```

```
    'distress_index': 'mean',
    'unemployment_rate': 'mean',
    'poverty_rate': 'mean',
    'median_income': 'mean',
    'college_pct': 'mean',
    'county_id': 'count'
}).round(3)
region_profiles.columns = ['Population', 'Distress', 'Unemployment', 'Poverty',
                           'Median Income', 'College %', 'Counties']

# Classify intervention priority
def classify_intervention(row):
    if row['Distress'] > 0.6:
        return 'Immediate Intervention Zone'
    elif row['Distress'] > 0.45:
        return 'High Priority Zone'
    elif row['Distress'] > 0.35:
        return 'Monitoring Zone'
    else:
        return 'Stable Zone'

region_profiles['Intervention Class'] = region_profiles.
 ↪apply(classify_intervention, axis=1)

print("="*70)
print("POLICY ZONE RECOMMENDATIONS")
print("="*70)

print(f"\n Region Profiles:")
print(region_profiles.sort_values('Distress', ascending=False).to_string())

# Summary by intervention class
intervention_summary = region_profiles.groupby('Intervention Class').agg({
    'Population': 'sum',
    'Distress': 'mean',
    'Counties': 'sum'
})

print(f"\n Intervention Summary:")
for idx, row in intervention_summary.iterrows():
    print(f"   {idx}:")
    print(f"      Counties: {int(row['Counties'])} | Population:␣
 ↪{row['Population']:,.0f}")
    print(f"      Avg Distress: {row['Distress']:.3f}")
```

```
======================================================================
POLICY ZONE RECOMMENDATIONS
```

```
======================================================================

 Region Profiles:
            Population  Distress  Unemployment  Poverty  Median Income  College
%  Counties  Intervention Class
maxp_region
2              1488597     0.585         0.074    0.145      51318.730
0.251        38  High Priority Zone
5               592187     0.560         0.074    0.134      54243.338
0.254        14  High Priority Zone
0              1209565     0.543         0.074    0.132      54428.522
0.262        30  High Priority Zone
6              1022333     0.467         0.061    0.118      57253.251
0.286        22  High Priority Zone
1               956858     0.409         0.052    0.108      57880.897
0.295        20     Monitoring Zone
4              1153297     0.347         0.059    0.104      59093.167
0.324        18         Stable Zone
3              1008142     0.213         0.042    0.084      69085.414
0.387         8         Stable Zone

 Intervention Summary:
  High Priority Zone:
     Counties: 104 | Population: 4,312,682
     Avg Distress: 0.539
  Monitoring Zone:
     Counties: 20 | Population: 956,858
     Avg Distress: 0.409
  Stable Zone:
     Counties: 26 | Population: 2,161,439
     Avg Distress: 0.280
```

[13]:
```
# ============================================================================
# Executive Summary
# ============================================================================

immediate = region_profiles[region_profiles['Intervention Class'] == 'Immediate␣
 ↪Intervention Zone']
high_priority = region_profiles[region_profiles['Intervention Class'] == 'High␣
 ↪Priority Zone']

print("="*70)
print("REGIONAL DEVELOPMENT ZONES: EXECUTIVE SUMMARY")
print("="*70)

print(f"""
 REGIONALIZATION RESULTS:
```

```
  Max-p created {n_maxp_regions} contiguous regions from {len(counties)}␣
↪counties
  All regions meet minimum population threshold of {min_population:,}

  Comparison with naive clustering:
  • K-Means: {kmeans_metrics['fragmented']} fragmented regions,␣
↪{kmeans_metrics['below_threshold']} below threshold
  • Max-p: 0 fragmented, 0 below threshold

 INTERVENTION ZONES:
  Immediate Intervention: {len(immediate)} regions
     Population: {immediate['Population'].sum():,.0f}
     Avg Distress: {immediate['Distress'].mean():.3f}

  High Priority: {len(high_priority)} regions
     Population: {high_priority['Population'].sum():,.0f}
     Avg Distress: {high_priority['Distress'].mean():.3f}

 POLICY RECOMMENDATIONS:

  1. DEPLOY targeted interventions to Immediate Intervention Zones:
     • Workforce development grants
     • Small business support
     • Infrastructure investment

  2. ESTABLISH regional coordination offices:
     • One per Max-p region for administrative efficiency
     • Population-appropriate staffing

  3. MONITOR High Priority Zones for escalation:
     • Quarterly distress index tracking
     • Early warning indicators

  4. USE Max-p regions for:
     • Grant allocation
     • Performance evaluation
     • Statistical sampling

 KRL SUITE COMPONENTS USED:
  • [Community] Contiguity weights, SKATER basics
  • [Pro] MaxPRegions - Threshold-constrained regionalization
  • [Enterprise] OptimalZoning - Multi-objective optimization
""")

print("\n" + "="*70)
print("Upgrade to Pro tier for Max-p regionalization: kr-labs.io/pricing")
print("="*70)
```

```
======================================================================
REGIONAL DEVELOPMENT ZONES: EXECUTIVE SUMMARY
======================================================================

  REGIONALIZATION RESULTS:
    Max-p created 7 contiguous regions from 150 counties
    All regions meet minimum population threshold of 500,000

    Comparison with naive clustering:
    • K-Means: 7 fragmented regions, 1 below threshold
    • Max-p: 0 fragmented, 0 below threshold

  INTERVENTION ZONES:
    Immediate Intervention: 0 regions
       Population: 0
       Avg Distress: nan

    High Priority: 4 regions
       Population: 4,312,682
       Avg Distress: 0.539

  POLICY RECOMMENDATIONS:

    1. DEPLOY targeted interventions to Immediate Intervention Zones:
       • Workforce development grants
       • Small business support
       • Infrastructure investment

    2. ESTABLISH regional coordination offices:
       • One per Max-p region for administrative efficiency
       • Population-appropriate staffing

    3. MONITOR High Priority Zones for escalation:
       • Quarterly distress index tracking
       • Early warning indicators

    4. USE Max-p regions for:
       • Grant allocation
       • Performance evaluation
       • Statistical sampling

  KRL SUITE COMPONENTS USED:
    • [Community] Contiguity weights, SKATER basics
    • [Pro] MaxPRegions - Threshold-constrained regionalization
    • [Enterprise] OptimalZoning - Multi-objective optimization


======================================================================
```

```
Upgrade to Pro tier for Max-p regionalization: kr-labs.io/pricing
================================================================================
```

0.9 Appendix: Regionalization Methods Reference

Method	Tier	Contiguity	Threshold	Best For
K-Means	Community			Exploratory clustering
SKATER	Community			Basic contiguous regions
Max-p	**Pro**			Policy zones with constraints
REDCAP	**Pro**			Capacity-constrained planning
OptimalZoning	**Enterprise**			Multi-objective optimization

0.9.1 References

1. Duque, J.C., et al. (2012). The Max-p-Regions Problem. *Journal of Regional Science*.
2. Assunção, R.M., et al. (2006). Efficient regionalization techniques. *Geographical Analysis*.

Generated with KRL Suite v2.0 - Showcasing Pro/Enterprise capabilities