In [1]:
```
# Colab-friendly installs
%pip install -q tensorflow  # or latest TF2.x
%pip install -q nodevectors # for Node2Vec-like walks
%pip install -q scipy
%pip install -q networkx pandas numpy scikit-learn matplotlib seaborn scipy
```

```
Note: you may need to restart the kernel to use updated packages.
Note: you may need to restart the kernel to use updated packages.
Note: you may need to restart the kernel to use updated packages.
Note: you may need to restart the kernel to use updated packages.
```

In [40]:
```
import networkx as nx
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_auc_score, accuracy_score, classification_repo
import itertools

def load_pajek_graph(filepath):
    """
    Loads a Pajek .net file into a NetworkX DiGraph.
    Handles potential parsing quirks by forcing a Directed Graph.
    """
    try:
        # networkx read_pajek returns a MultiGraph or MultiDiGraph usually
        G_multi = nx.read_pajek(filepath)
        # Convert to a simple Directed Graph (summing weights if duplicates ex
        G = nx.DiGraph()
        for u, v, data in G_multi.edges(data=True):
            weight = data.get('weight', 1.0)
            if G.has_edge(u, v):
                G[u][v]['weight'] += weight
            else:
                G.add_edge(u, v, weight=weight)
        return G
    except Exception as e:
        print(f"Error loading {filepath}: {e}")
        return None

def extract_features(G_current, target_edges_set):
    """
    Generates a dataset for link prediction.
    Rows: All possible pairs of nodes.
    Columns: Topological features + Previous Weight.
    Target: 1 if link exists in the 'future' (target_edges_set), else 0.
    """

    # Get all unique nodes (countries)
    nodes = sorted(list(G_current.nodes()))

    # Create an undirected view for calculation of specific metrics
    # (Jaccard and Adamic-Adar are typically defined for undirected neighbors)
    G_undir = G_current.to_undirected()
```

```
    data = []

    # Iterate through all possible pairs (u, v)
    # We use permutations because direction matters (Arg -> Bra is different f
    for u, v in itertools.permutations(nodes, 2):

        # 1. Feature: Weight in the current graph (Temporal persistence)
        # If they are already connected, how strong is it?
        prev_weight = G_current[u][v]['weight'] if G_current.has_edge(u, v) el

        # 2. Feature: Common Neighbors (Undirected count is usually robust for
        common_neigh = len(list(nx.common_neighbors(G_undir, u, v)))

        # 3. Feature: Jaccard Coefficient
        preds_jaccard = list(nx.jaccard_coefficient(G_undir, [(u, v)]))
        jaccard_score = preds_jaccard[0][2] if preds_jaccard else 0

        # 4. Feature: Adamic-Adar Index (Resource allocation)
        preds_aa = list(nx.adamic_adar_index(G_undir, [(u, v)]))
        adamic_adar = preds_aa[0][2] if preds_aa else 0

        # 5. Feature: Preferential Attachment (Degree product)
        pref_attach = G_undir.degree(u) * G_undir.degree(v)

        # --- TARGET VARIABLE ---
        # Does this link exist in the FUTURE graph?
        label = 1 if (u, v) in target_edges_set else 0

        data.append([u, v, prev_weight, common_neigh, jaccard_score, adamic_ad

    df = pd.DataFrame(data, columns=['u', 'v', 'prev_weight', 'common_neighbor
                                     'jaccard', 'adamic_adar', 'pref_attach',
    return df

# --- MAIN EXECUTION ---

# 1. Load the graphs
print("Loading graphs...")
G_2010 = load_pajek_graph('/Users/aishwarysinghrathour/Desk/TRimester_5/Graph_
G_2015 = load_pajek_graph('/Users/aishwarysinghrathour/Desk/TRimester_5/Graph_
G_2018 = load_pajek_graph('/Users/aishwarysinghrathour/Desk/TRimester_5/Graph_

# Ensure we loaded correctly
if G_2010 and G_2015 and G_2018:
    print(f"Nodes in 2010: {len(G_2010.nodes())}, Edges: {len(G_2010.edges())}

    # 2. Prepare Training Data (Learn from 2010 -> 2015 evolution)
    print("Preparing Training Data (2010 -> 2015)...")
    # Set of edges in 2015 for fast lookup
    edges_2015 = set(G_2015.edges())
    train_df = extract_features(G_2010, edges_2015)

    # 3. Prepare Test Data (Predict 2015 -> 2018 evolution)
```

```
    print("Preparing Test Data (2015 -> 2018)...")
    edges_2018 = set(G_2018.edges())
    test_df = extract_features(G_2015, edges_2018)

    # 4. Train Model
    features = ['prev_weight', 'common_neighbors', 'jaccard', 'adamic_adar', '

    X_train = train_df[features]
    y_train = train_df['label']

    X_test = test_df[features]
    y_test = test_df['label']

    print("Training Random Forest Classifier...")
    clf = RandomForestClassifier(n_estimators=100, random_state=42, class_weig
    clf.fit(X_train, y_train)

    # 5. Evaluate
    print("Evaluating on 2018 predictions...")
    y_pred_proba = clf.predict_proba(X_test)[:, 1] # Probability of link exist
    y_pred = clf.predict(X_test)

    # Metrics
    auc = roc_auc_score(y_test, y_pred_proba)
    print(f"\n--- Results ---")
    print(f"ROC-AUC Score: {auc:.4f} (Higher is better, 0.5 is random guessing
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

    # 6. Feature Importance
    importances = clf.feature_importances_
    print("\nFeature Importance:")
    for name, imp in zip(features, importances):
        print(f"{name}: {imp:.4f}")

else:
    print("Failed to load one or more graphs. Please check file paths.")
```

```
Loading graphs...
Nodes in 2010: 161, Edges: 18579
Preparing Training Data (2010 -> 2015)...
Preparing Test Data (2015 -> 2018)...
Training Random Forest Classifier...
Evaluating on 2018 predictions...

--- Results ---
ROC-AUC Score: 0.8007 (Higher is better, 0.5 is random guessing)

Classification Report:
              precision    recall  f1-score   support

           0       0.72      0.57      0.64      8730
           1       0.80      0.88      0.84     17030

    accuracy                           0.78     25760
   macro avg       0.76      0.73      0.74     25760
weighted avg       0.77      0.78      0.77     25760


Feature Importance:
prev_weight: 0.3655
common_neighbors: 0.0604
jaccard: 0.0990
adamic_adar: 0.2493
pref_attach: 0.2258
```

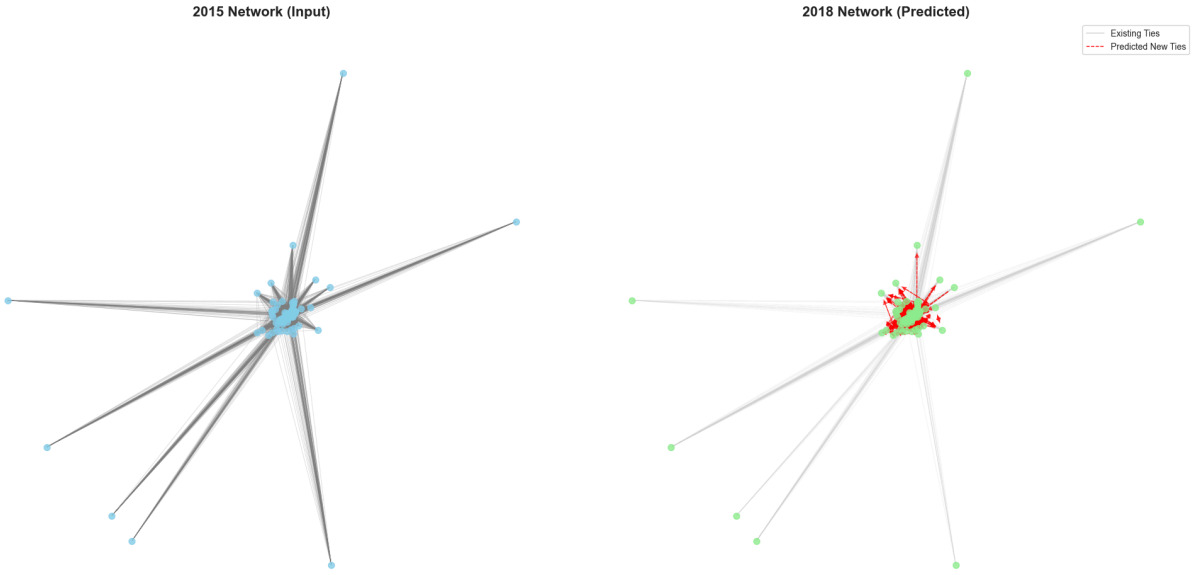
In [42]:

```python
import networkx as nx
import matplotlib.pyplot as plt

def visualize_prediction(G_past, G_future_predicted, title_before, title_after
    """
    Visualizes the evolution of the network.

    Args:
    - G_past: The graph at time T (e.g., 2015)
    - G_future_predicted: The graph with predicted edges for T+1
    - title_before: Title for the first plot
    - title_after: Title for the second plot
    """

    # 1. Setup the Layout (Keep node positions constant for comparison)
    # We use the union of both graphs to ensure all nodes have a fixed place
    G_combined = nx.compose(G_past, G_future_predicted)
    # k controls the spacing (higher = more spread out)
    pos = nx.spring_layout(G_combined, seed=42, k=0.15)

    # --- PLOT 1: BEFORE (The Past Graph) ---
    plt.figure(figsize=(20, 10))

    # Left Subplot: Before
    plt.subplot(1, 2, 1)
    plt.title(title_before, fontsize=16, fontweight='bold')
```

```
    # Draw nodes
    nx.draw_networkx_nodes(G_past, pos, node_size=50, node_color='skyblue', al

    # Draw edges (Past links are solid gray)
    nx.draw_networkx_edges(G_past, pos, edge_color='gray', alpha=0.3, width=0.

    # Draw labels (optional, can be cluttered for 161 nodes)
    # nx.draw_networkx_labels(G_past, pos, font_size=8)
    plt.axis('off')

    # --- PLOT 2: AFTER (The Predicted Graph) ---
    # We want to distinguish between "Old Links" (persisted) and "New Links" (
    plt.subplot(1, 2, 2)
    plt.title(title_after, fontsize=16, fontweight='bold')

    # Identify New Links: Edges in Predicted that were NOT in Past
    past_edges = set(G_past.edges())
    predicted_edges = set(G_future_predicted.edges())
    new_edges = list(predicted_edges - past_edges)
    persisted_edges = list(predicted_edges & past_edges)

    # Draw nodes
    nx.draw_networkx_nodes(G_future_predicted, pos, node_size=50, node_color='

    # Draw OLD edges (Background, Gray)
    nx.draw_networkx_edges(G_future_predicted, pos, edgelist=persisted_edges,
                           edge_color='lightgray', alpha=0.2, width=0.5, arrow

    # Draw NEW PREDICTED edges (Foreground, Red)
    if new_edges:
        nx.draw_networkx_edges(G_future_predicted, pos, edgelist=new_edges,
                               edge_color='red', alpha=0.6, width=1.0, style='

    plt.axis('off')

    # Add a legend for the second plot
    from matplotlib.lines import Line2D
    custom_lines = [Line2D([0], [0], color='lightgray', lw=1),
                    Line2D([0], [0], color='red', lw=1, linestyle='--')]
    plt.legend(custom_lines, ['Existing Ties', 'Predicted New Ties'], loc='upp

    plt.tight_layout()
    plt.show()

# --- HOW TO RUN THIS WITH YOUR PREVIOUS CODE ---

# 1. Reconstruct the "Predicted Graph" from the model output
# (Assuming 'clf' is your trained model and 'test_df' is your prepared data)
print("Reconstructing Predicted Graph...")

# Get predictions (0 or 1)
predictions = clf.predict(X_test)
```

```
# Create a new graph for 2018 (Predicted)
G_2018_Predicted = nx.DiGraph()
G_2018_Predicted.add_nodes_from(G_2015.nodes()) # Start with same nodes

# Add edges where prediction == 1
# 'test_df' has columns ['u', 'v', ...] and we align it with 'predictions'
for index, row in test_df.iterrows():
    u, v = row['u'], row['v']
    # The dataframe index corresponds to the predictions array index
    if predictions[index] == 1:
        G_2018_Predicted.add_edge(u, v)

# 2. Run the Visualization
# This will show 2015 on the left and Predicted 2018 on the right
visualize_prediction(G_2015, G_2018_Predicted,
                     title_before="2015 Network (Input)",
                     title_after="2018 Network (Predicted)")
```

```
Reconstructing Predicted Graph...
```



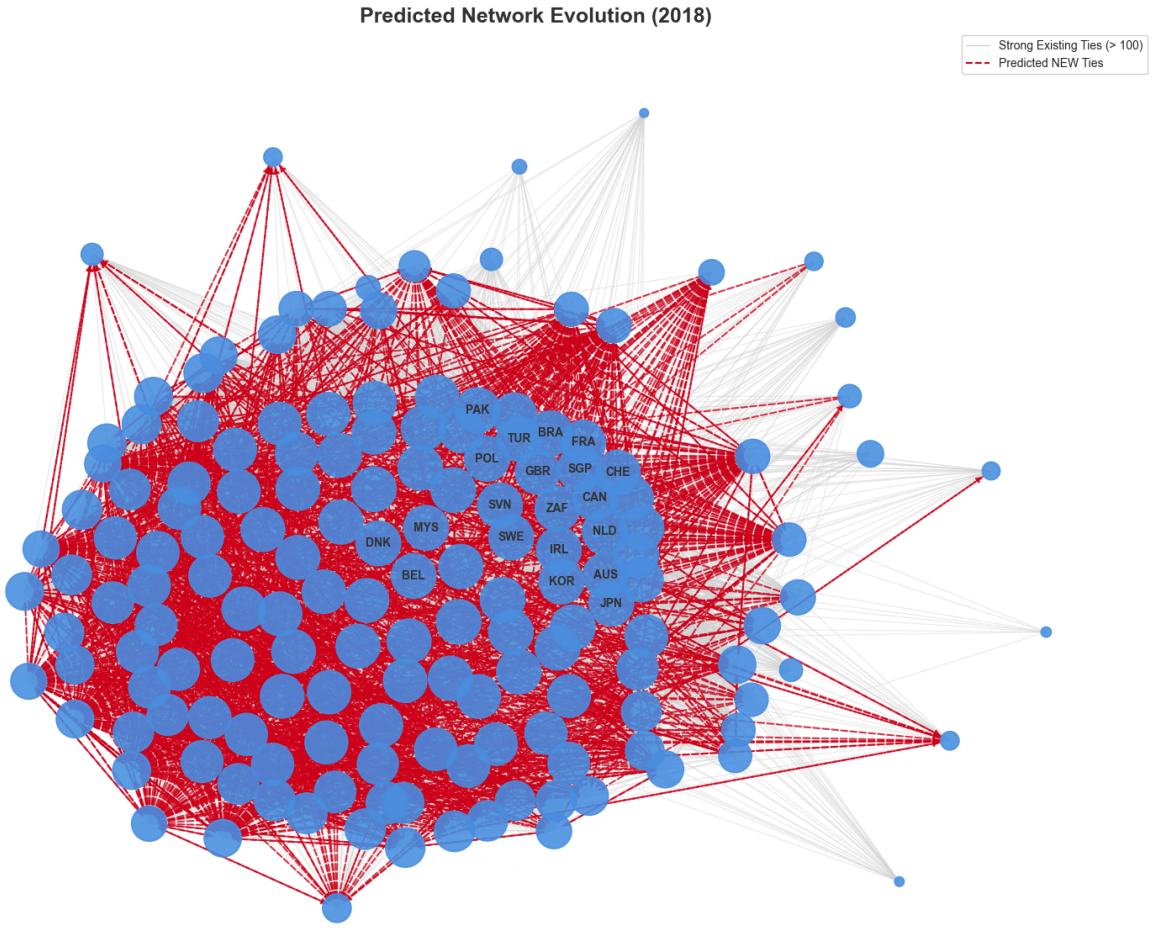
In [43]:

```
import networkx as nx
import matplotlib.pyplot as plt

def visualize_clearer_prediction(G_past, G_future_predicted, title, min_weight
    """
    Visualizes the network with filters to reduce clutter.

    Args:
    - min_weight_threshold: Only show PAST edges if weight > this value.
                            (Adjust this based on your data! Try 10, 100, or 1
    """
    plt.figure(figsize=(15, 12))

    # 1. layout: Kamada-Kawai is often better for separating clusters than Spr
```

```
    # We calculate layout on the full predicted graph
    print("Computing layout...")
    pos = nx.kamada_kawai_layout(G_future_predicted)

    # 2. Separate Edges for Plotting
    past_edges = G_past.edges(data=True)
    predicted_edges = G_future_predicted.edges()

    # FILTER 1: existing_strong_edges (Background)
    # Only draw existing edges if they are "strong" (weight > threshold)
    existing_strong_edges = [
        (u, v) for u, v, d in past_edges
        if d.get('weight', 0) > min_weight_threshold
    ]

    # FILTER 2: new_predicted_edges (Foreground)
    # Edges that are in Future but NOT in Past
    past_edge_set = set(G_past.edges())
    new_predicted_edges = [
        (u, v) for u, v in predicted_edges
        if (u, v) not in past_edge_set
    ]

    # 3. Draw the Graph
    plt.title(title, fontsize=18, fontweight='bold', color='#333333')

    # Draw Nodes (Size based on degree/importance)
    d = dict(G_future_predicted.degree)
    node_sizes = [v * 5 for v in d.values()] # Scale node size by connections
    nx.draw_networkx_nodes(G_future_predicted, pos, node_size=node_sizes, node

    # Draw Background Edges (Stable, Strong Ties) - Thin and Gray
    nx.draw_networkx_edges(G_future_predicted, pos, edgelist=existing_strong_e
                           edge_color='lightgray', alpha=0.4, width=0.8, arrow

    # Draw New Predicted Edges (The Insight) - Red and Prominent
    nx.draw_networkx_edges(G_future_predicted, pos, edgelist=new_predicted_edg
                           edge_color='#D0021B', alpha=0.8, width=1.5, style='

    # 4. Smart Labeling (Only label top 20 hubs to avoid text mess)
    # Sort nodes by degree
    sorted_nodes = sorted(G_future_predicted.degree, key=lambda x: x[1], rever
    top_hubs = [n for n, deg in sorted_nodes[:20]]

    # Create a label dictionary just for top hubs
    labels = {n: n for n in top_hubs}
    nx.draw_networkx_labels(G_future_predicted, pos, labels, font_size=10, for

    # Legend
    from matplotlib.lines import Line2D
    legend_elements = [
        Line2D([0], [0], color='lightgray', lw=1, label=f'Strong Existing Ties
        Line2D([0], [0], color='#D0021B', lw=1.5, linestyle='--', label='Predi
```

```
    ]
    plt.legend(handles=legend_elements, loc='upper right')

    plt.axis('off')
    plt.tight_layout()
    plt.show()

# --- RUN THIS ---
# You might need to experiment with 'min_weight_threshold'.
# Since your data has weights like 25875 and 0.345, try 100 or 1000 first.
visualize_clearer_prediction(G_2015, G_2018_Predicted,
                             "Predicted Network Evolution (2018)",
                             min_weight_threshold=100)
```

```
Computing layout...
```



In [45]:
```
%pip install -q pyvis
```

```
Note: you may need to restart the kernel to use updated packages.
```

In [47]:
```
from pyvis.network import Network

def visualize_interactive(G_past, G_future_predicted):
    nt = Network(height='750px', width='100%', bgcolor='#222222', font_color='
```

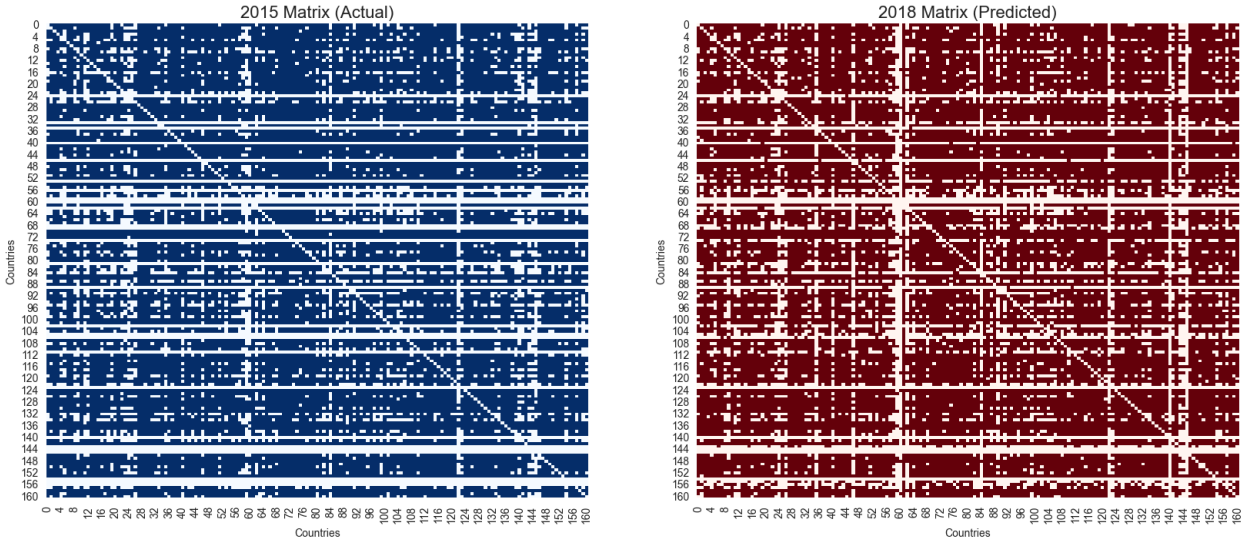
```
    # Calculate new edges
    past_edges = set(G_past.edges())

    # Add nodes and edges to PyVis
    for node in G_future_predicted.nodes():
        nt.add_node(node, title=node)

    for u, v in G_future_predicted.edges():
        if (u, v) not in past_edges:
            # NEW PREDICTED LINK (Red)
            nt.add_edge(u, v, color='red', width=2, title="Predicted New")
        elif G_past.edges[u, v].get('weight', 0) > 100:
            # STRONG EXISTING LINK (Gray)
            nt.add_edge(u, v, color='#444444', width=1)

    nt.show('prediction_graph.html')
    print("Graph saved to prediction_graph.html")

visualize_interactive(G_2015, G_2018_Predicted)
```

```
Warning: When  cdn_resources is 'local' jupyter notebook has issues displaying
graphics on chrome/safari. Use cdn_resources='in_line' or cdn_resources='remot
e' if you have issues viewing graphics in a notebook.
prediction_graph.html
Graph saved to prediction_graph.html
```

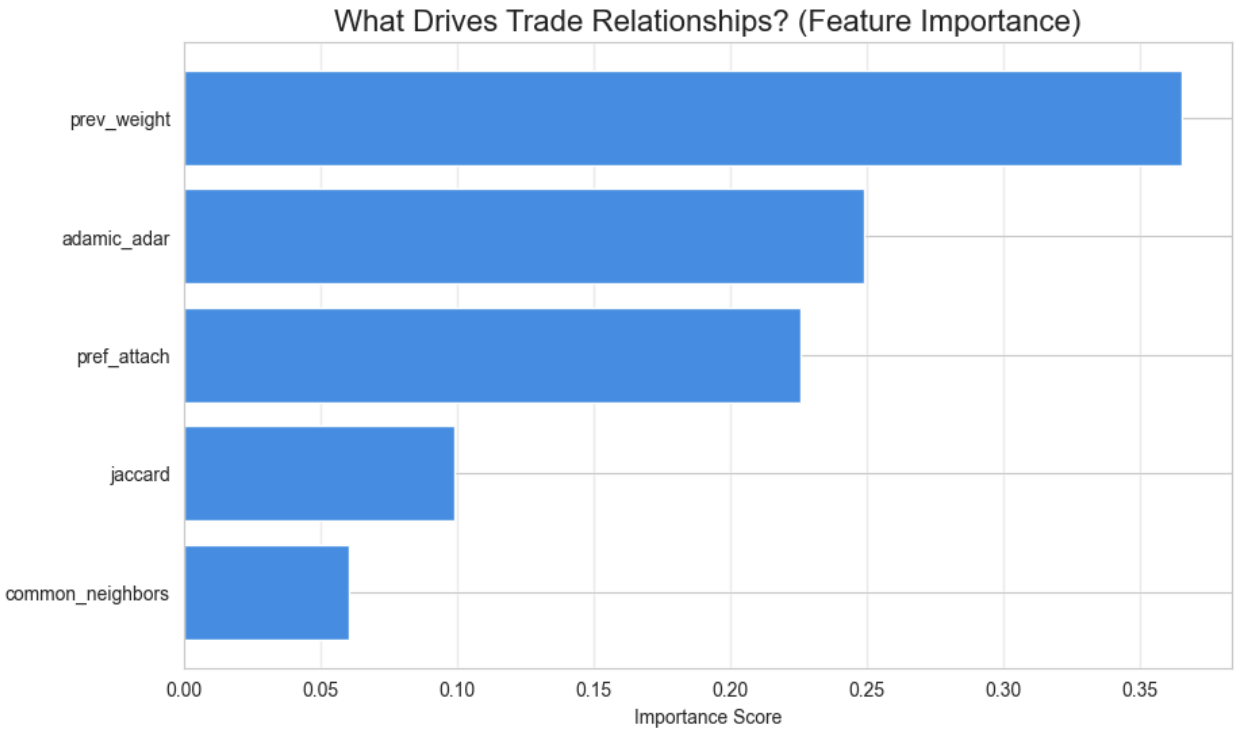
In [48]:

```
import seaborn as sns
import matplotlib.pyplot as plt
import networkx as nx

def plot_adjacency_heatmap(G1, G2, title1, title2):
    # Get nodes in same order for both
    nodes = sorted(G1.nodes())

    # Convert to matrices
    adj1 = nx.to_numpy_array(G1, nodelist=nodes, weight=None) # Binary (0/1)
    adj2 = nx.to_numpy_array(G2, nodelist=nodes, weight=None)

    fig, axes = plt.subplots(1, 2, figsize=(20, 8))

    # Plot 1
    sns.heatmap(adj1, ax=axes[0], cmap="Blues", cbar=False)
    axes[0].set_title(title1, fontsize=16)
    axes[0].set_xlabel("Countries")
    axes[0].set_ylabel("Countries")

    # Plot 2
    sns.heatmap(adj2, ax=axes[1], cmap="Reds", cbar=False)
    axes[1].set_title(title2, fontsize=16)
    axes[1].set_xlabel("Countries")
    axes[1].set_ylabel("Countries")

    plt.show()
```

```
# Run it
plot_adjacency_heatmap(G_2015, G_2018_Predicted, "2015 Matrix (Actual)", "2018
```

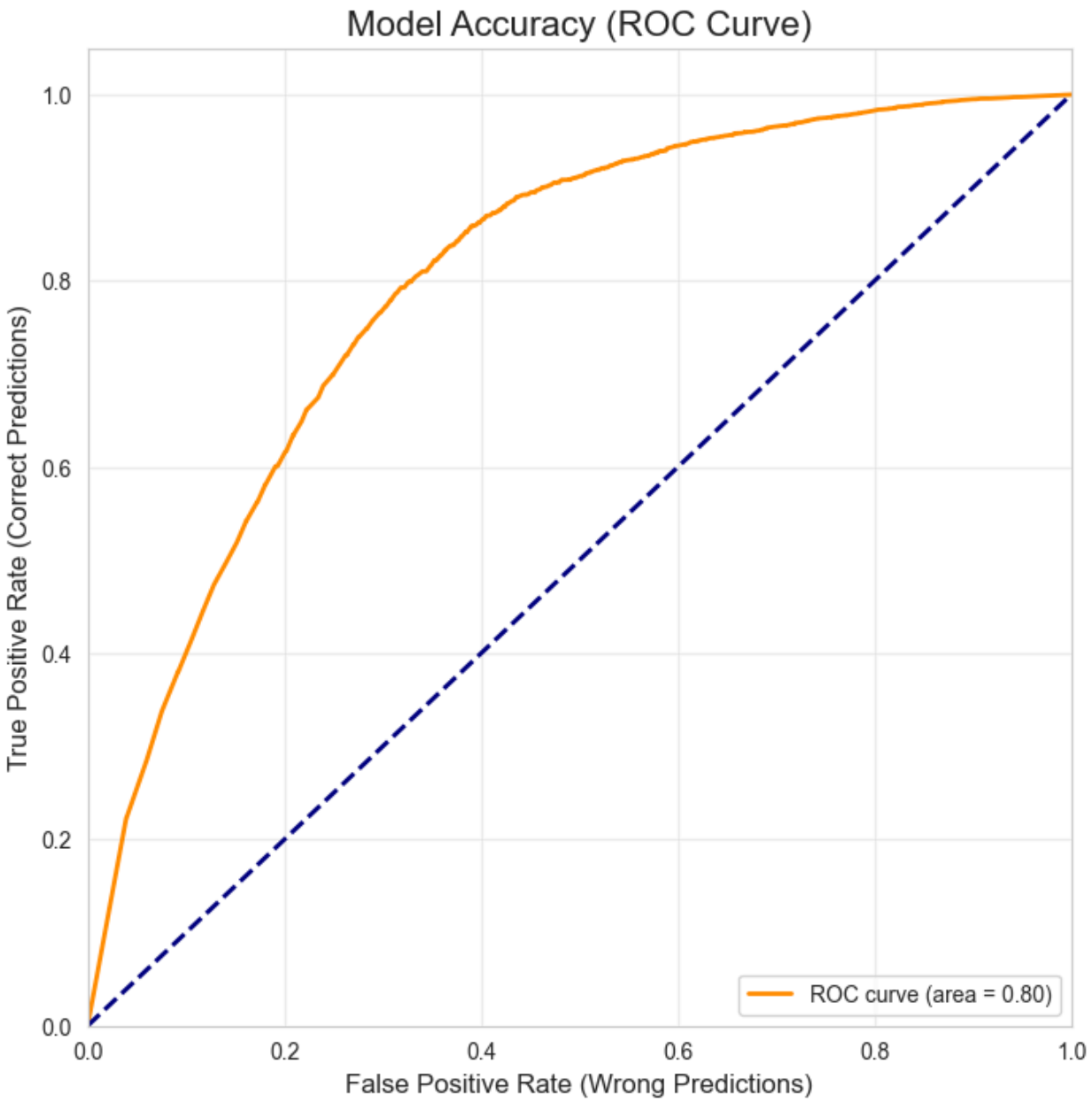


In [49]:
```
import pandas as pd
import matplotlib.pyplot as plt

def plot_feature_importance(clf, feature_names):
    # Get importance from the trained model
    importances = clf.feature_importances_

    # Create a DataFrame for easy plotting
    df_imp = pd.DataFrame({'Feature': feature_names, 'Importance': importances
    df_imp = df_imp.sort_values('Importance', ascending=True)

    # Plot
    plt.figure(figsize=(10, 6))
    plt.barh(df_imp['Feature'], df_imp['Importance'], color='#4A90E2')
    plt.title('What Drives Trade Relationships? (Feature Importance)', fontsiz
    plt.xlabel('Importance Score')
    plt.grid(axis='x', alpha=0.3)
    plt.show()

# Run it (using variables from your previous training step)
features = ['prev_weight', 'common_neighbors', 'jaccard', 'adamic_adar', 'pref
plot_feature_importance(clf, features)
```



In [50]:

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

def plot_roc_curve(y_test, y_pred_proba):
    fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
    roc_auc = auc(fpr, tpr)

    plt.figure(figsize=(8, 8))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {rc
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate (Wrong Predictions)', fontsize=12)
    plt.ylabel('True Positive Rate (Correct Predictions)', fontsize=12)
    plt.title('Model Accuracy (ROC Curve)', fontsize=16)
    plt.legend(loc="lower right")
    plt.grid(alpha=0.3)
    plt.show()

# Run it
# y_pred_proba comes from the earlier step: clf.predict_proba(X_test)[:, 1]
plot_roc_curve(y_test, y_pred_proba)
```



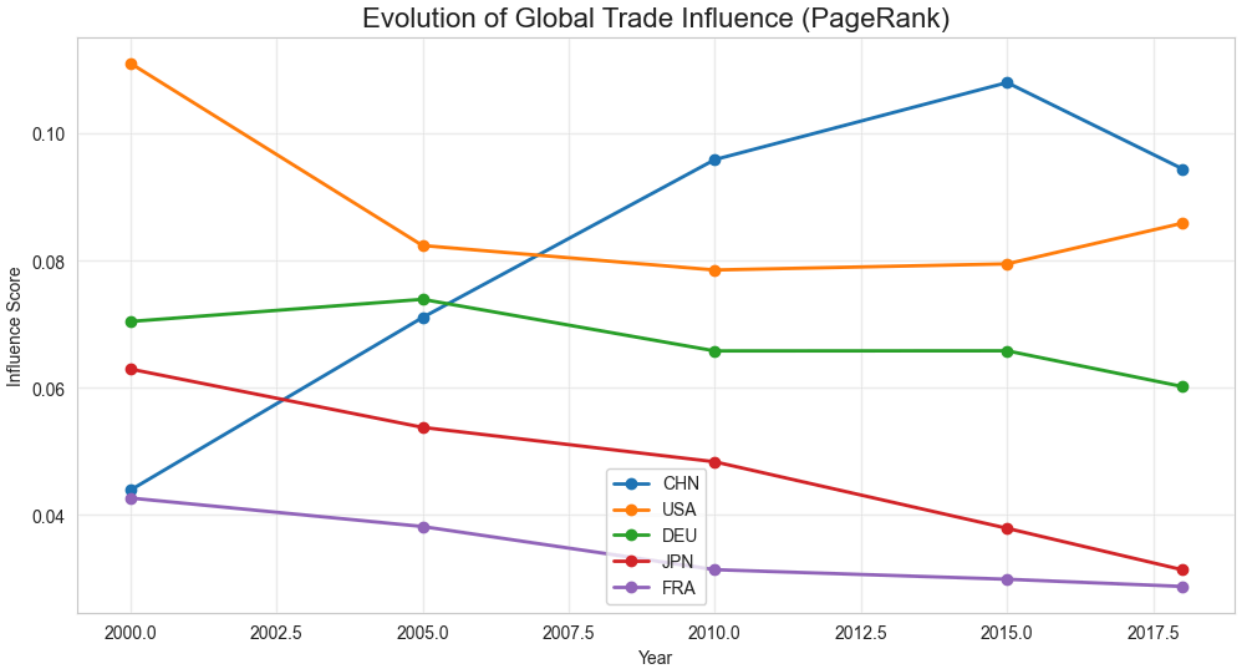
In [56]:
```python
import networkx as nx
import pandas as pd
import matplotlib.pyplot as plt
from networkx.algorithms.community import greedy_modularity_communities

# --- 1. LOAD THE DATA ---
print("1. Loading Data...")
files = ['/Users/aishwarysinghrathour/Desk/TRimester_5/Graph_Analytics/Project
years = [2000, 2005, 2010, 2015, 2018]
graphs = {}

def load_pajek_graph(filepath):
    try:
        # Read file
        G_multi = nx.read_pajek(filepath)
```

```
        G = nx.DiGraph()
        # Fix potential multi-edge issues
        for u, v, data in G_multi.edges(data=True):
            weight = data.get('weight', 1.0)
            if G.has_edge(u, v):
                G[u][v]['weight'] += weight
            else:
                G.add_edge(u, v, weight=weight)
        return G
    except Exception as e:
        print(f"Failed to load {filepath}: {e}")
        return None

# Load all files into a dictionary
for file, year in zip(files, years):
    g = load_pajek_graph(file)
    if g:
        graphs[year] = g
        print(f"   - Loaded {year}: {len(g.nodes())} countries")

# --- 2. RUN TREND ANALYSIS (The Rise & Fall) ---
print("\n2. Generating Trend Analysis Graph...")

def analyze_dominance_trends(graph_dict):
    data = []
    for year, G in graph_dict.items():
        # Calculate PageRank (Influence)
        pr = nx.pagerank(G, weight='weight')
        for country, score in pr.items():
            data.append({'Year': year, 'Country': country, 'Score': score})

    df = pd.DataFrame(data)

    # Find the top 5 countries in 2018
    final_year_data = df[df['Year'] == 2018]
    if final_year_data.empty:
        print("   Error: No data for 2018 found.")
        return

    top_countries = final_year_data.nlargest(5, 'Score')['Country'].tolist()
    print(f"   - Top 5 Influential Countries in 2018: {top_countries}")

    # Plot
    plt.figure(figsize=(12, 6))
    for country in top_countries:
        trajectory = df[df['Country'] == country]
        plt.plot(trajectory['Year'], trajectory['Score'], marker='o', label=cc

    plt.title("Evolution of Global Trade Influence (PageRank)", fontsize=16)
    plt.xlabel("Year")
    plt.ylabel("Influence Score")
    plt.legend()
    plt.grid(True, alpha=0.3)
```

```
    plt.show()

# CALL THE FUNCTION
if graphs:
    analyze_dominance_trends(graphs)


# --- 3. RUN COMMUNITY DETECTION (Trade Blocs) ---
print("\n3. Detecting Trade Communities in 2018...")

def find_trade_blocs(G, year):
    if not G: return

    # Convert to undirected for community detection
    G_undir = G.to_undirected()

    # Greedy Modularity method
    communities = list(greedy_modularity_communities(G_undir, weight='weight')

    print(f"   --- Major Trade Blocs in {year} ---")
    count = 1
    for c in communities:
        # Only print communities with more than 5 countries
        if len(c) > 5:
            # Convert to list and sort to make it readable
            members = sorted(list(c))
            # Print first 10 members
            preview = ", ".join(members[:10])
            print(f"   Bloc {count} ({len(members)} countries): {preview}...")
            count += 1

# CALL THE FUNCTION
if 2018 in graphs:
    find_trade_blocs(graphs[2018], 2018)


# --- 4. RUN SHOCK SIMULATION ---
print("\n4. Simulating Economic Shock (Removing 'USA')...")

def simulate_shock(G, target_country):
    if target_country not in G:
        print(f"   Error: {target_country} not found in the graph.")
        return

    G_shocked = G.copy()
    G_shocked.remove_node(target_country)

    # We use 'density' as a proxy for efficiency here because it's faster to c
    # (Global efficiency can be slow on large graphs)
    density_before = nx.density(G)
    density_after = nx.density(G_shocked)

    drop = (density_before - density_after) / density_before * 100
```

```
    print(f"   - Network Density BEFORE removing {target_country}: {density_be
    print(f"   - Network Density AFTER removing {target_country}:  {density_af
    print(f"   - Impact: Connectivity dropped by {drop:.2f}%")

# CALL THE FUNCTION
if 2018 in graphs:
    simulate_shock(graphs[2018], "USA") # You can change "USA" to "CHN" or "DE
```

```
1. Loading Data...
   - Loaded 2000: 161 countries
   - Loaded 2005: 161 countries
   - Loaded 2010: 161 countries
   - Loaded 2015: 161 countries
   - Loaded 2018: 161 countries

2. Generating Trend Analysis Graph...
   - Top 5 Influential Countries in 2018: ['CHN', 'USA', 'DEU', 'JPN', 'FRA']
```




```
3. Detecting Trade Communities in 2018...
   --- Major Trade Blocs in 2018 ---
   Bloc 1 (71 countries): AGO, ARE, AUS, BDI, BEN, BGD, BHR, BRN, BTN, BWA...
   Bloc 2 (57 countries): ALB, ARM, AUT, AZE, BEL, BFA, BGR, BIH, BLR, CAF...
   Bloc 3 (33 countries): ARG, ATG, BHS, BMU, BOL, BRA, BRB, CAN, CHL, COL...

4. Simulating Economic Shock (Removing 'USA')...
   - Network Density BEFORE removing USA: 0.6611
   - Network Density AFTER removing USA:  0.6586
   - Impact: Connectivity dropped by 0.38%
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: