In [1]:
```
import glob
import os
import pandas as pd
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score, c
pd.set_option('display.max_columns', None)

events = pd.read_parquet("../data/gdelt/events/6_final/events_dataset.parque
gkg = pd.read_parquet("../data/gdelt/gkg/6_final/gkg_dataset.parquet")
```

In [2]:
```
print(len(events.index))
print(events.columns)
events.tail(3)
```

```
67119
Index(['ADMIN0', 'ADMIN1', 'ADMIN2', 'CS_score', 'period', 'SQLDATE',
       'EventCode', 'SOURCEURL', 'NumMentions', 'NumSources', 'NumArticles',
       'valid_SOURCEURL', 'clean_text_list', 'NER_admin0_list',
       'NER_admin1_list', 'NER_admin2_list', 'compound_score_list',
       'neg_score_list', 'neu_score_list', 'pos_score_list',
       'fatalities_freq_list', 'displaced_freq_list', 'detained_freq_list',
       'injured_freq_list', 'sexual_violence_freq_list', 'torture_freq_lis
t',
       'economic_shocks_freq_list', 'agriculture_freq_list',
       'weather_freq_list', 'food_insecurity_freq_list',
       'fatalities_count_list', 'displaced_count_list', 'detained_count_lis
t',
       'injured_count_list', 'sexual_violence_count_list',
       'torture_count_list', 'sentiment.compound_list', 'sentiment.neg_lis
t',
       'sentiment.neu_list', 'sentiment.pos_list', 'pred_impact_type_list',
       'pred_urgency_list', 'pred_resource_food_list',
       'pred_resource_water_list', 'pred_resource_cash_aid_list',
       'pred_resource_healthcare_list', 'pred_resource_shelter_list',
       'pred_resource_livelihoods_list', 'pred_resource_education_list',
       'pred_resource_infrastructure_list', 'pred_resource_none_list'],
      dtype='object')
```

Out[2]:

	ADMIN0	ADMIN1	ADMIN2	CS_score	period	SQLDATE	EventCode	
193050	nigeria	zamfara	tsafe	NaN	201601	[2016-01-01, 2016-01-01, 2016-01-03, 2016-01-0...	[20.0, 42.0, 20.0, 173.0, 173.0, 100.0, 100.0,...	[http:/
193116	nigeria	zamfara	tsafe	NaN	202305	[2023-05-09, 2023-05-10, 2023-05-10, 2023-05-1...	[10.0, 71.0, 173.0, 173.0, 173.0, 42.0, 10.0, ...	[ht
193390	niger	zinder	tanout	NaN	202308	[2023-08-03, 2023-08-07, 2023-08-07, 2023-08-0...	[42.0, 42.0, 57.0, 70.0, 162.0, 162.0, 20.0, 7...	[https:

In [ ]:
```
# Forecasting Configuration
# ============================
# Set the forecast horizon: predict CS_score N periods ahead
# Options: 1 (nowcasting), 2, 3, or 4 (forecasting)
# Literature shows GDELT features are more valuable for forecasting (2-4 per
# because baseline (previous_CS) becomes less powerful with longer horizons

FORECAST_HORIZON = 3  # Predict 3 periods ahead (can be changed to 2 or 4)

print(f"=== Forecasting Configuration ===")
print(f"Forecast Horizon: {FORECAST_HORIZON} period(s) ahead")
if FORECAST_HORIZON == 1:
    print("Mode: NOWCASTING (predicting next period)")
else:
    print(f"Mode: FORECASTING (predicting {FORECAST_HORIZON} periods ahead)"
print(f"\nWhy forecasting helps GDELT features:")
print(f"  - Baseline 'previous_CS' becomes less predictive ({FORECAST_HORIZO
print(f"  - GDELT temporal patterns can capture early warning signals")
print(f"  - More realistic for early warning systems\n")
```

In [3]:
```
df = pd.merge(events, gkg, on=["ADMIN0", "ADMIN1", "ADMIN2", "period"], how=
print(len(df.index))
df.tail(3)
```

```
68246
```

Out[3]:

	ADMIN0	ADMIN1	ADMIN2	CS_score_events	period	SQLDATE	EventCode
68243	None	zamfara	kaura namoda	NaN	202306	[2023-06-01, 2023-06-01, 2023-06-01, 2023-06-0...	[46.0, 10.0, 20.0, 42.0, 20.0, 43.0, 173.0, 20...
68244	None	zamfara	kaura namoda	NaN	202309	[2023-09-01, 2023-09-02, 2023-09-02, 2023-09-0...	[14.0, 42.0, 10.0, 112.0, 30.0, 10.0, 43.0, 71...
68245	None	zamfara	kaura namoda	NaN	202311	[2023-11-01, 2023-11-01, 2023-11-01, 2023-11-0...	[71.0, 46.0, 10.0, 10.0, 10.0, 15.0, 20.0, 20....

In [4]:
```python
key_cols = ["ADMIN0", "ADMIN1", "ADMIN2", "period"]

def check_duplicates(df, name):
    dupes = (
        df
        .groupby(key_cols)
        .size()
        .reset_index(name="n")
        .query("n > 1")
    )
    print(f"{name}: {len(dupes)} duplicated keys")
    return dupes

dupes_events = check_duplicates(events, "events")
dupes_gkg = check_duplicates(gkg, "gkg")
dupes_merged = check_duplicates(df, "merged")
```

```
events: 0 duplicated keys
gkg: 0 duplicated keys
merged: 0 duplicated keys
```

In [5]:
```python
# Feature Availability Analysis - RIGHT AFTER MERGE
# ================================================
# This must be done BEFORE any feature engineering or filtering
# to accurately assess data coverage using original SQLDATE and DATE arrays

print("=== Data Coverage Analysis (After Merge) ===\n")

# Total rows after merge
total_rows = len(df)
print(f"1. Total rows after merge: {total_rows:,}")
```

```python
# Function to check if arrays are empty
def is_empty_array(x):
    """Check if an array/list is empty"""
    if x is None:
        return True
    if isinstance(x, np.ndarray):
        return x.size == 0
    try:
        if pd.isna(x):
            return True
    except (ValueError, TypeError):
        pass
    if isinstance(x, (list, tuple)):
        return len(x) == 0
    return False

# Check for events data - SQLDATE being empty means no events
if 'SQLDATE' in df.columns:
    df['has_events_data'] = (~df['SQLDATE'].apply(is_empty_array)).astype(ir
    events_count = df['has_events_data'].sum()
    print(f"2. Rows with events data (non-empty SQLDATE): {events_count:,} (
else:
    df['has_events_data'] = 0
    print("2. SQLDATE column not found - cannot check events data")

# Check for GKG data - DATE being empty means no GKG data
if 'DATE' in df.columns:
    df['has_gkg_data'] = (~df['DATE'].apply(is_empty_array)).astype(int)
    gkg_count = df['has_gkg_data'].sum()
    print(f"3. Rows with GKG data (non-empty DATE): {gkg_count:,} ({gkg_cour
else:
    df['has_gkg_data'] = 0
    print("3. DATE column not found - cannot check GKG data")

# Overall feature availability
df['has_any_features'] = (df['has_events_data'] | df['has_gkg_data']).astype
features_count = df['has_any_features'].sum()
print(f"4. Rows with ANY GDELT features: {features_count:,} ({features_count
print(f"   Rows WITHOUT GDELT features: {total_rows - features_count:,} ({(t

# Check CS_score availability
valid_cs_count = 0
valid_cs_with_features_count = 0

if 'CS_score_events' in df.columns and 'CS_score_gkg' in df.columns:
    df['CS_score'] = df['CS_score_events'].fillna(df['CS_score_gkg'])
    df['CS_score'] = pd.to_numeric(df['CS_score'], errors='coerce')
    valid_cs = df[(df['CS_score'] >= 1) & (df['CS_score'] <= 5) & (df['CS_sc
    valid_cs_count = len(valid_cs)
    print(f"\n5. Rows with valid CS_score (1-5): {valid_cs_count:,} ({valid_

    # Rows with valid CS_score AND features
    valid_cs_with_features = valid_cs[valid_cs['has_any_features'] == 1]
    valid_cs_with_features_count = len(valid_cs_with_features)
    print(f"6. Rows with valid CS_score AND GDELT features: {valid_cs_with_f
```

```
    print(f"   - Coverage: {valid_cs_with_features_count/valid_cs_count*100:
    print(f"   - Coverage: {valid_cs_with_features_count/total_rows*100:.1f}
else:
    print("\n5. CS_score columns not found")

print(f"\n=== Summary ===")
print(f"Total rows: {total_rows:,}")
if valid_cs_count > 0:
    print(f"Valid CS_score rows: {valid_cs_count:,}")
    print(f"Valid CS_score + Features: {valid_cs_with_features_count:,}")
```

```
=== Data Coverage Analysis (After Merge) ===

1. Total rows after merge: 68,246
2. Rows with events data (non-empty SQLDATE): 37,045 (54.3%)
3. Rows with GKG data (non-empty DATE): 3,800 (5.6%)
4. Rows with ANY GDELT features: 38,286 (56.1%)
   Rows WITHOUT GDELT features: 29,960 (43.9%)

5. Rows with valid CS_score (1-5): 56,777 (83.2%)
6. Rows with valid CS_score AND GDELT features: 26,817
   - Coverage: 47.2% of valid CS_score rows
   - Coverage: 39.3% of total rows

=== Summary ===
Total rows: 68,246
Valid CS_score rows: 56,777
Valid CS_score + Features: 26,817
```

In [6]:

```
# Feature Engineering for CS_score Prediction
# ============================================

def safe_list_agg(lst, func):
    """Safely aggregate a list, handling None, empty lists, and non-numeric
    # Handle None first
    if lst is None:
        return np.nan

    # Handle numpy arrays and lists
    if isinstance(lst, np.ndarray):
        if lst.size == 0:
            return np.nan
        # Convert to list for processing
        lst = lst.tolist()

    # Check for pandas NA/NaN (must check after None and array checks)
    try:
        if pd.isna(lst):
            return np.nan
    except (ValueError, TypeError):
        # pd.isna() failed, might be array-like, continue processing
        pass

    # Handle scalar numeric values
    if isinstance(lst, (int, float)):
        return float(lst)
```

```
    # Handle strings
    if isinstance(lst, str):
        try:
            # Try to evaluate if it's a string representation of a list
            if lst.startswith('[') or lst.startswith('('):
                lst = eval(lst)
            else:
                # Try to convert single value
                return float(lst)
        except:
            return np.nan

    # Check if it's a list-like structure
    if not isinstance(lst, (list, tuple)):
        return np.nan

    # Handle empty lists
    if len(lst) == 0:
        return np.nan

    # Process list elements
    try:
        # Convert to numeric, filtering out non-numeric values
        numeric_lst = []
        for x in lst:
            # Check for NaN/None values
            try:
                if pd.isna(x) or x is None:
                    continue
            except (ValueError, TypeError):
                # pd.isna() might fail for some types, try to convert anyway
                pass

            try:
                val = float(x)
                if not np.isinf(val) and not np.isnan(val):
                    numeric_lst.append(val)
            except (ValueError, TypeError):
                continue

        if len(numeric_lst) == 0:
            return np.nan

        result = func(numeric_lst)
        return float(result) if not np.isnan(result) and not np.isinf(result
    except Exception as e:
        return np.nan

def safe_list_count(x):
    """Safely count elements in a list/array, handling various data types"""
    if x is None:
        return 0
    if isinstance(x, np.ndarray):
        return x.size if x.size > 0 else 0
    if isinstance(x, (list, tuple)):
```

```
        return len(x)
    # For scalar values, check if it's not NaN
    try:
        if pd.isna(x):
            return 0
        return 1
    except (ValueError, TypeError):
        # If pd.isna fails, assume it's a valid value
        return 1

def aggregate_list_features(df, list_cols, prefix=""):
    """Aggregate list columns into multiple statistical features"""
    # Collect all new columns in a dictionary to avoid fragmentation
    new_cols = {}
    cols_to_drop = []

    for col in list_cols:
        if col not in df.columns:
            continue

        base_name = col.replace('_list', '').replace('_events', '').replace(
        if prefix:
            base_name = f"{prefix}_{base_name}"

        # Compute all aggregations for this column
        new_cols[f"{base_name}_mean"] = df[col].apply(lambda x: safe_list_ag
        new_cols[f"{base_name}_max"] = df[col].apply(lambda x: safe_list_agg
        new_cols[f"{base_name}_sum"] = df[col].apply(lambda x: safe_list_agg
        new_cols[f"{base_name}_count"] = df[col].apply(safe_list_count)
        new_cols[f"{base_name}_std"] = df[col].apply(lambda x: safe_list_agg
        new_cols[f"{base_name}_min"] = df[col].apply(lambda x: safe_list_agg

        # Track columns to drop
        cols_to_drop.append(col)

    # Add all new columns at once using pd.concat to avoid fragmentation
    if new_cols:
        new_df = pd.DataFrame(new_cols, index=df.index)
        df = pd.concat([df, new_df], axis=1)

    # Drop original list columns
    if cols_to_drop:
        df = df.drop(columns=cols_to_drop)

    return df

# Step 1: Determine target variable
# ==================================
# IMPORTANT: CS_score from FEWSNET is only available roughly every 4 months.
# Intermediary months can be used for feature engineering (lags, moving aver
# but will be filtered out before model training (only CS_score between 1-5

# Option 1: Use events CS_score as primary, fill with gkg if missing
df['CS_score'] = df['CS_score_events'].fillna(df['CS_score_gkg'])

# Option 2: Average if both exist (uncomment if preferred)
```

```
# df['CS_score'] = df[['CS_score_events', 'CS_score_gkg']].mean(axis=1)

# Option 3: Use maximum (uncomment if preferred)
# df['CS_score'] = df[['CS_score_events', 'CS_score_gkg']].max(axis=1)

print(f"CS_score distribution (before filtering):")
print(df['CS_score'].value_counts().sort_index())
print(f"\nMissing CS_score: {df['CS_score'].isna().sum()}")
print(f"\nNote: Intermediary months (with missing CS_score) will be used for
print(f"feature engineering but filtered out before model training.")
```

```
CS_score distribution (before filtering):
CS_score
0.000000      691
1.000000    29885
1.271386        1
1.500000       58
2.000000    14547
2.500000       45
3.000000    11014
3.500000        8
4.000000     1218
5.000000        1
Name: count, dtype: int64

Missing CS_score: 10778

Note: Intermediary months (with missing CS_score) will be used for
feature engineering but filtered out before model training.
```

In [ ]:

```
# Fix Text Features: Remove nonsensical statistical aggregations
# ================================================================
# NER and clean_text columns contain strings, so mean/std/min/max/sum don't
# Only count features are meaningful for text columns

print("=== Fixing Text Features (NER and clean_text) ===")

# Identify text columns that have nonsensical statistical features
text_stat_cols = [c for c in df.columns if (('NER' in c) or ('clean_text' in

print(f"Found {len(text_stat_cols)} nonsensical text statistical features to
for col in sorted(text_stat_cols)[:20]:
    print(f"  - {col}")
if len(text_stat_cols) > 20:
    print(f"  ... and {len(text_stat_cols) - 20} more")

# Drop these columns
if text_stat_cols:
    df = df.drop(columns=text_stat_cols)

# Keep only text count features (which are meaningful)
text_count_cols = [c for c in df.columns if (('NER' in c) or ('clean_text' i
print(f"\nKept {len(text_count_cols)} meaningful text count features")
print(f"DataFrame shape after fixing text features: {df.shape}")
```

In [ ]:

```
# Display Final GDELT Features
# ============================
# Show all GDELT features after aggregation and cleanup to verify they make

print("="*70)
print("FINAL GDELT FEATURES LIST")
print("="*70)

# Get all GDELT features
gdelt_features = [c for c in df.columns if c.startswith('evt_') or c.startsw
gdelt_features = sorted(gdelt_features)

print(f"\nTotal GDELT features: {len(gdelt_features)}\n")

# Group by feature type
evt_features = [f for f in gdelt_features if f.startswith('evt_')]
gkg_features = [f for f in gdelt_features if f.startswith('gkg_')]

print(f"Events (evt_) features: {len(evt_features)}")
print(f"GKG (gkg_) features: {len(gkg_features)}\n")

# Categorize features
categories = {
    'Sentiment': ['compound', 'sentiment', 'neg', 'neu', 'pos'],
    'NER (Entity Recognition)': ['NER'],
    'Text': ['clean_text'],
    'Fatalities': ['fatalities'],
    'Displaced': ['displaced'],
    'Detained': ['detained'],
    'Injured': ['injured'],
    'Violence': ['sexual_violence', 'torture'],
    'Economic': ['economic_shocks'],
    'Agriculture': ['agriculture'],
    'Weather': ['weather'],
    'Food Security': ['food_insecurity'],
    'Predictions': ['pred_impact', 'pred_urgency', 'pred_resource'],
    'Temporal (Lags/Rolling)': ['_lag', '_rolling', '_anomaly', '_escalatior
}

print("="*70)
print("FEATURES BY CATEGORY")
print("="*70)

for category, keywords in categories.items():
    matching = [f for f in gdelt_features if any(kw in f for kw in keywords)
    if matching:
        print(f"\n{category} ({len(matching)} features):")
        for feat in sorted(matching)[:10]:
            print(f"  - {feat}")
        if len(matching) > 10:
            print(f"  ... and {len(matching) - 10} more")

# Show uncategorized features
categorized = set()
for keywords in categories.values():
```

```
    for kw in keywords:
        categorized.update([f for f in gdelt_features if kw in f])

uncategorized = [f for f in gdelt_features if f not in categorized]
if uncategorized:
    print(f"\nUncategorized ({len(uncategorized)} features):")
    for feat in sorted(uncategorized)[:20]:
        print(f"  - {feat}")
    if len(uncategorized) > 20:
        print(f"  ... and {len(uncategorized) - 20} more")

print("\n" + "="*70)
print("COMPLETE FEATURE LIST (Alphabetical)")
print("="*70)
print("\nEvents Features:")
for i, feat in enumerate(evt_features, 1):
    print(f"{i:3d}. {feat}")

print(f"\n\nGKG Features:")
for i, feat in enumerate(gkg_features, 1):
    print(f"{i:3d}. {feat}")

print("\n" + "="*70)
print(f"TOTAL: {len(gdelt_features)} GDELT features")
print("="*70)
```

In [7]:

```
# Step 2: Aggregate list features from both datasets
# ====================================================

# Capture column count BEFORE aggregation
cols_before_agg = len(df.columns)

# Identify all list columns
list_cols_events = [c for c in df.columns if '_list_events' in c]
list_cols_gkg = [c for c in df.columns if '_list_gkg' in c]

print(f"Found {len(list_cols_events)} list columns from events")
print(f"Found {len(list_cols_gkg)} list columns from gkg")
print(f"Total list columns: {len(list_cols_events) + len(list_cols_gkg)}")
print(f"Columns before aggregation: {cols_before_agg}")

# Aggregate events list features
print("\nAggregating events list features...")
df = aggregate_list_features(df, list_cols_events, prefix="evt")

# Aggregate gkg list features
print("Aggregating gkg list features...")
df = aggregate_list_features(df, list_cols_gkg, prefix="gkg")

cols_after_agg = len(df.columns)
print(f"\nDataFrame shape after aggregation: {df.shape}")
print(f"Columns after aggregation: {cols_after_agg}")

# Feature Breakdown Analysis
# ============================
print("\n" + "="*70)
```

```
print("FEATURE TRANSFORMATION BREAKDOWN")
print("="*70)

total_list_cols = len(list_cols_events) + len(list_cols_gkg)
features_per_list_col = 6  # mean, max, sum, count, std, min
new_features_from_lists = total_list_cols * features_per_list_col
non_list_cols_before = cols_before_agg - total_list_cols

print(f"\n1. Starting point:")
print(f"   - Total columns before aggregation: {cols_before_agg}")
print(f"   - List columns to aggregate: {total_list_cols} (39 events + 39 gk
print(f"   - Non-list columns: {non_list_cols_before}")

print(f"\n2. Transformation:")
print(f"   - Each list column → {features_per_list_col} features:")
print(f"     * mean: average value in the list")
print(f"     * max: maximum value in the list")
print(f"     * sum: sum of all values in the list")
print(f"     * count: number of elements in the list")
print(f"     * std: standard deviation of values in the list")
print(f"     * min: minimum value in the list")
print(f"   - Total new features created: {total_list_cols} × {features_per_l

print(f"\n3. Result:")
print(f"   - Original non-list columns: {non_list_cols_before}")
print(f"   - New aggregated features: {new_features_from_lists}")
print(f"   - List columns removed: {total_list_cols}")
print(f"   - Expected total: {non_list_cols_before} + {new_features_from_lis
print(f"   - Actual total: {cols_after_agg}")

# Show sample of created features
evt_features = [c for c in df.columns if c.startswith('evt_')]
gkg_features = [c for c in df.columns if c.startswith('gkg_')]
non_list_cols = [c for c in df.columns if not c.startswith('evt_') and not c

print(f"\n4. Column breakdown:")
print(f"   - evt_* features: {len(evt_features)} (from events list columns)"
print(f"   - gkg_* features: {len(gkg_features)} (from gkg list columns)")
print(f"   - Other columns: {len(non_list_cols)} (identifiers, metadata, etc

print(f"\n5. Sample features created:")
print(f"\n   Events features (first 15):")
for feat in sorted(evt_features)[:15]:
    print(f"      - {feat}")
if len(evt_features) > 15:
    print(f"      ... and {len(evt_features) - 15} more evt_ features")

print(f"\n   GKG features (first 15):")
for feat in sorted(gkg_features)[:15]:
    print(f"      - {feat}")
if len(gkg_features) > 15:
    print(f"      ... and {len(gkg_features) - 15} more gkg_ features")

print(f"\n6. Non-aggregated columns ({len(non_list_cols)}):")
for col in sorted(non_list_cols):
    print(f"      - {col}")
```

```
print("\n" + "="*70)
```

```
Found 39 list columns from events
Found 39 list columns from gkg
Total list columns: 78
Columns before aggregation: 116

Aggregating events list features...
Aggregating gkg list features...

DataFrame shape after aggregation: (68246, 506)
Columns after aggregation: 506

================================================================================
FEATURE TRANSFORMATION BREAKDOWN
================================================================================

1. Starting point:
   - Total columns before aggregation: 116
   - List columns to aggregate: 78 (39 events + 39 gkg)
   - Non-list columns: 38

2. Transformation:
   - Each list column → 6 features:
     * mean: average value in the list
     * max: maximum value in the list
     * sum: sum of all values in the list
     * count: number of elements in the list
     * std: standard deviation of values in the list
     * min: minimum value in the list
   - Total new features created: 78 × 6 = 468

3. Result:
   - Original non-list columns: 38
   - New aggregated features: 468
   - List columns removed: 78
   - Expected total: 38 + 468 = 506
   - Actual total: 506

4. Column breakdown:
   - evt_* features: 234 (from events list columns)
   - gkg_* features: 234 (from gkg list columns)
   - Other columns: 38 (identifiers, metadata, etc.)

5. Sample features created:

   Events features (first 15):
      - evt_NER_admin0_count
      - evt_NER_admin0_max
      - evt_NER_admin0_mean
      - evt_NER_admin0_min
      - evt_NER_admin0_std
      - evt_NER_admin0_sum
      - evt_NER_admin1_count
      - evt_NER_admin1_max
      - evt_NER_admin1_mean
      - evt_NER_admin1_min
      - evt_NER_admin1_std
```

- evt_NER_admin1_sum
- evt_NER_admin2_count
- evt_NER_admin2_max
- evt_NER_admin2_mean
... and 219 more evt_ features

GKG features (first 15):
- gkg_NER_admin0_count
- gkg_NER_admin0_max
- gkg_NER_admin0_mean
- gkg_NER_admin0_min
- gkg_NER_admin0_std
- gkg_NER_admin0_sum
- gkg_NER_admin1_count
- gkg_NER_admin1_max
- gkg_NER_admin1_mean
- gkg_NER_admin1_min
- gkg_NER_admin1_std
- gkg_NER_admin1_sum
- gkg_NER_admin2_count
- gkg_NER_admin2_max
- gkg_NER_admin2_mean
... and 219 more gkg_ features

6. Non-aggregated columns (38):
- ADMIN0
- ADMIN1
- ADMIN2
- Amounts
- CS_score
- CS_score_events
- CS_score_gkg
- DATE
- DocumentIdentifier
- EventCode
- NumArticles
- NumMentions
- NumSources
- SOURCEURL
- SQLDATE
- V2Themes
- has_any_features
- has_events_data
- has_gkg_data
- is_negative
- n_conflict_related
- n_disease_related
- n_displaced
- n_food_related
- n_injured
- n_killed
- n_market_related
- n_missing
- n_policy_related
- n_price_related
- n_water_related

```
      - n_weather_related
      - period
      - tone
      - tone_abs
      - usd_aid
      - valid_DocumentIdentifier
      - valid_SOURCEURL

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

In [8]:

```python
# Step 3: Create combined and interaction features
# ================================================

# Combine CS_score from both sources (if both exist, use average)
df['CS_score_combined'] = df[['CS_score_events', 'CS_score_gkg']].mean(axis=
df['CS_score_diff'] = df['CS_score_events'] - df['CS_score_gkg']
df['has_both_scores'] = (df['CS_score_events'].notna() & df['CS_score_gkg'].

# Combine numeric features from GKG (these are already aggregated)
# Create ratios and normalized features
if 'n_killed' in df.columns:
    df['casualty_rate'] = df['n_killed'] / (df['n_killed'] + df['n_injured']
    df['total_casualties'] = df['n_killed'] + df['n_injured'] + df['n_missir
    df['aid_per_casualty'] = df['usd_aid'] / (df['total_casualties'] + 1)

# Combine event counts from GKG themes
theme_cols = [c for c in df.columns if '_related' in c]
if theme_cols:
    df['total_themes'] = df[theme_cols].sum(axis=1)
    df['conflict_intensity'] = df['n_conflict_related'] / (df['total_themes'
    df['crisis_severity'] = (df['n_food_related'] + df['n_water_related'] +

# Combine sentiment features (if aggregated from lists)
sentiment_cols = [c for c in df.columns if 'sentiment' in c.lower() or 'comp
if sentiment_cols:
    # Create overall sentiment indicators
    if 'evt_compound_mean' in df.columns and 'gkg_compound_mean' in df.colum
        df['sentiment_combined'] = (df['evt_compound_mean'].fillna(0) + df['
        df['sentiment_agreement'] = (df['evt_compound_mean'] * df['gkg_compo

# Event coverage features
if 'NumMentions' in df.columns:
    df['mentions_per_source'] = df['NumMentions'] / (df['NumSources'] + 1)
    df['articles_per_source'] = df['NumArticles'] / (df['NumSources'] + 1)
    df['coverage_intensity'] = df['NumMentions'] * df['NumSources']

# Tone features from GKG
if 'tone' in df.columns:
    df['tone_abs_normalized'] = df['tone_abs'] / (abs(df['tone']) + 1)
    df['negative_tone'] = (df['tone'] < 0).astype(int)

print("Created interaction and combined features")
print(f"Current DataFrame shape: {df.shape}")
```

```
Created interaction and combined features
Current DataFrame shape: (68246, 520)
```

In [9]:
```
# Step 4: Create simple temporal features for FORECASTING
# ======================================================
# NOTE: Modified for forecasting (predicting N periods ahead)
# We create simple, safe temporal features BEFORE split to prevent leakage.

# Create region identifiers (matching previous work)
df['region'] = df['ADMIN0'] + '-' + df['ADMIN1']
df['district'] = df['ADMIN0'] + '-' + df['ADMIN1'] + '-' + df['ADMIN2']

# Sort by region and period to ensure deterministic lags
df = df.sort_values(['region', 'period']).reset_index(drop=True)

# FORECASTING: Create target variable by shifting CS_score backward by FOREC
# At time t, we want to predict CS_score at time t+FORECAST_HORIZON
# So we shift CS_score backward: CS_score_target[t] = CS_score[t+FORECAST_HO
df['CS_score_target'] = df.groupby('region', sort=False)['CS_score'].shift(-

# Simple temporal features for forecasting:
# 1. previous_CS: CS_score from FORECAST_HORIZON periods ago (most recent av
#    For forecasting, this is less predictive than for nowcasting
df['previous_CS'] = df.groupby('region', sort=False)['CS_score'].shift(FOREC

# 2. transitions_prev: cumulative count of CS_score transitions up to t-FORE
# Count changes, then cumsum and shift so it reflects info only up to t-FORE
change_flag = (
    df.groupby('region', sort=False)['CS_score']
      .transform(lambda x: (x != x.shift()).astype(int))
)
df['transitions_prev'] = (
    change_flag.groupby(df['region']).cumsum().shift(FORECAST_HORIZON).fillr
)

# Also create CS_score at different lags for additional baseline features
for lag in [1, 2, FORECAST_HORIZON]:
    if lag != FORECAST_HORIZON:  # Don't duplicate previous_CS
        df[f'CS_score_lag{lag}'] = df.groupby('region', sort=False)['CS_scor

print(f"Created temporal features for {FORECAST_HORIZON}-period forecasting:
print(f"  - CS_score_target: target variable (CS_score shifted {FORECAST_HOR
print(f"  - previous_CS: CS_score from {FORECAST_HORIZON} periods ago")
print(f"  - transitions_prev: transitions up to {FORECAST_HORIZON} periods a
print(f"  - CS_score_lag1, CS_score_lag2: additional lag features")
print(f"\nCurrent DataFrame shape: {df.shape}")
```

```
Created simple temporal features (previous_CS, transitions_prev)
Current DataFrame shape: (68246, 524)
```

In [10]:
```
# Step 5: Filter to valid CS_score_target and prepare for FORECASTING models
# ==========================================================================
# CRITICAL: For FORECASTING, we filter on CS_score_target (the future value
# Filter to only rows where CS_score_target is between 1 and 5 (inclusive).
# Following previous work framework: keep only last 7+FORECAST_HORIZON perio

print(f"=== Step 5: Filtering to Valid CS_score_target (Forecasting {FORECAS

# Ensure both CS_score (features) and CS_score_target (target) are numeric
```

```
df['CS_score'] = pd.to_numeric(df['CS_score'], errors='coerce')
df['CS_score_target'] = pd.to_numeric(df['CS_score_target'], errors='coerce'

# Drop rows with NaN or infinite values in CS_score_target (we need valid ta
initial_len = len(df)
df = df[df['CS_score_target'].notna() & np.isfinite(df['CS_score_target'])].
if initial_len != len(df):
    print(f"Dropped {initial_len - len(df)} rows with invalid/missing CS_sco

# Round to nearest integer and convert to int
if df['CS_score_target'].dtype == 'float64':
    df['CS_score_target'] = df['CS_score_target'].round().astype(int)
else:
    df['CS_score_target'] = df['CS_score_target'].astype(int)

# Ensure target is in valid range (1-5)
df = df[df['CS_score_target'].between(1, 5, inclusive='both')].copy()

# Keep only last 7+FORECAST_HORIZON periods for evaluation
# We need extra periods because we're predicting forward
all_periods = sorted(df['period'].unique())
keep_periods = all_periods[-(7 + FORECAST_HORIZON):]
df = df[df['period'].isin(keep_periods)].copy()

print(f"Final dataset: {len(df)} rows with valid CS_score_target (1-5)")
print(f"Periods: {sorted(df['period'].unique())}")
print(f"Note: Predicting CS_score {FORECAST_HORIZON} periods ahead\n")

# Step 6: Split into train/test for FORECASTING
# ================================================
# Split after feature creation (features already created in Step 4)
# For forecasting: need to ensure test period has valid targets FORECAST_HOR
# Use last 5 periods for training, last period for testing

test_period = df['period'].max()
train = df[df['period'] < test_period].copy()
train = train[train['period'].isin(sorted(train['period'].unique())[-5:])].c

test = df[df['period'] == test_period].copy()

print(f"=== Step 6: Train/Test Split (Forecasting {FORECAST_HORIZON} periods
print(f"Test period: {test_period}")
print(f"Train periods: {sorted(train['period'].unique())}")
print(f"Train set: {len(train)} rows")
print(f"Test set: {len(test)} rows")
print(f"\nNote: At time t, predicting CS_score at time t+{FORECAST_HORIZON}\

# Drop rows where previous_CS is missing (matching previous work)
train = train[train['previous_CS'].isin([1, 2, 3, 4, 5])].copy()
test = test[test['previous_CS'].isin([1, 2, 3, 4, 5])].copy()

print(f"After dropping missing previous_CS:")
print(f"Train set: {len(train)} rows")
print(f"Test set: {len(test)} rows\n")
```

```
=== Step 5: Filtering to Valid CS_score and Preparing Data ===
Dropped 10778 rows with NaN or infinite CS_score
Final dataset: 15062 rows with valid CS_score (1-5)
Periods: ['202202', '202206', '202210', '202302', '202306', '202310', '20240
2']

=== Step 6: Train/Test Split ===
Test period: 202402
Train periods: ['202206', '202210', '202302', '202306', '202310']
Train set: 10408 rows
Test set: 2248 rows

After dropping missing previous_CS:
Train set: 9893 rows
Test set: 2112 rows
```

In [11]:

```python
# Step 4b: Create Temporal Features from GDELT Data
# ==================================================
# CRITICAL: Create temporal patterns from GDELT features (lags, rolling windo
# This matches how conflict features were engineered in previous work.
# Literature shows that temporal patterns in news/social data are predictive

print("=== Step 4b: Creating Temporal GDELT Features ===\n")

# Identify key GDELT features to create temporal patterns from
# Focus on features that might have predictive signal
key_gdelt_features = [
    # Sentiment features
    'evt_compound_score_mean', 'evt_sentiment.compound_mean',
    'gkg_compound_mean',

    # Food security indicators
    'evt_food_insecurity_freq_mean', 'gkg_food_insecurity_freq_mean',

    # Crisis indicators
    'evt_displaced_freq_mean', 'evt_fatalities_freq_mean',
    'gkg_displaced_freq_mean', 'gkg_fatalities_freq_mean',

    # Economic/agricultural indicators
    'evt_agriculture_freq_mean', 'evt_economic_shocks_freq_mean',
    'gkg_agriculture_freq_mean',

    # Weather indicators
    'evt_weather_freq_mean', 'gkg_weather_freq_mean'
]

# Keep only features that exist in the dataset
key_gdelt_features = [f for f in key_gdelt_features if f in df.columns]
print(f"Found {len(key_gdelt_features)} key GDELT features for temporal engi

if len(key_gdelt_features) > 0:
    # Ensure df is sorted by region and period
    df = df.sort_values(['region', 'period']).reset_index(drop=True)

    # Create lagged features (1-3 periods back) - these capture historical p
```

```
    print("Creating lagged features (lag1, lag2, lag3)...")
    for feat in key_gdelt_features:
        for lag in [1, 2, 3]:
            df[f'{feat}_lag{lag}'] = df.groupby('region', sort=False)[feat].

    # Create rolling aggregations (3 and 6 periods) with shift(1) to prevent
    # These capture trends and momentum
    print("Creating rolling window features (3 and 6 periods)...")
    for feat in key_gdelt_features:
        for window in [3, 6]:
            df[f'{feat}_rolling_{window}'] = (
                df.groupby('region', sort=False)[feat]
                .shift(1).rolling(window, min_periods=1).mean()
            )

    # Create anomaly features (deviation from historical mean)
    # These capture unusual spikes or drops
    print("Creating anomaly features (deviation from historical mean)...")
    for feat in key_gdelt_features:
        historical_mean = df.groupby('region', sort=False)[feat].transform(
            lambda x: x.expanding().mean().shift(1)
        )
        df[f'{feat}_anomaly'] = df[feat] - historical_mean

    # Create escalation features (short-term vs medium-term trend)
    # Positive escalation = accelerating crisis
    print("Creating escalation features (short-term vs medium-term)...")
    for feat in key_gdelt_features:
        if f'{feat}_rolling_3' in df.columns and f'{feat}_rolling_6' in df.c
            df[f'{feat}_escalation'] = (
                df[f'{feat}_rolling_3'] - df[f'{feat}_rolling_6']
            )

    # Create change features (period-over-period change)
    print("Creating change features (period-over-period change)...")
    for feat in key_gdelt_features:
        df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1
        df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pc

    temporal_features_count = len([c for c in df.columns if any(x in c for x
    print(f"\nCreated {temporal_features_count} temporal GDELT features")
    print("These features capture:")
    print("  - Historical patterns (lags)")
    print("  - Trends (rolling windows)")
    print("  - Anomalies (deviations from mean)")
    print("  - Escalation (acceleration patterns)")
    print("  - Changes (period-over-period dynamics)")
else:
    print("Warning: No key GDELT features found. Using only current-period G

print(f"\nDataFrame shape after temporal GDELT features: {df.shape}")
```

```
=== Step 4b: Creating Temporal GDELT Features ===

Found 13 key GDELT features for temporal engineering
Creating lagged features (lag1, lag2, lag3)...
Creating rolling window features (3 and 6 periods)...
Creating anomaly features (deviation from historical mean)...
Creating escalation features (short-term vs medium-term)...
Creating change features (period-over-period change)...

Created 117 temporal GDELT features
These features capture:
  - Historical patterns (lags)
  - Trends (rolling windows)
  - Anomalies (deviations from mean)
  - Escalation (acceleration patterns)
  - Changes (period-over-period dynamics)

DataFrame shape after temporal GDELT features: (15062, 641)
```

```
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:76: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:77: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pct_chan
ge(1, fill_method=None)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:76: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:77: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pct_chan
ge(1, fill_method=None)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:76: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:77: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pct_chan
ge(1, fill_method=None)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:76: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:77: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pct_chan
ge(1, fill_method=None)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:76: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
```

```
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:77: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pct_chan
ge(1, fill_method=None)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:76: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:77: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pct_chan
ge(1, fill_method=None)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:76: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:77: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pct_chan
ge(1, fill_method=None)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:76: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:77: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pct_chan
ge(1, fill_method=None)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:76: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:77: PerformanceWarning: DataFrame is highly fragmented.  This is usually
```

```
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pct_chan
ge(1, fill_method=None)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:76: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change'] = df.groupby('region', sort=False)[feat].diff(1)
/var/folders/3y/cdj5wjs107z_64y5f9ws4cj40000gn/T/ipykernel_96599/2097296920.
py:77: PerformanceWarning: DataFrame is highly fragmented.  This is usually
the result of calling `frame.insert` many times, which has poor performance.
Consider joining all columns at once using pd.concat(axis=1) instead. To get
a de-fragmented frame, use `newframe = frame.copy()`
  df[f'{feat}_change_pct'] = df.groupby('region', sort=False)[feat].pct_chan
ge(1, fill_method=None)
```

In [12]:

```python
# Step 7: Baseline Models (matching previous work framework)
# ============================================================
# Implement the 4 baseline models from 6_modelling_conflicts_lags.ipynb:
# 1. PPS (Previous Period Same)
# 2. SPLY (Same Period Last Year)
# 3. Max-2PP (Max of Previous 2 Periods)
# 4. ML Baseline (Logistic Regression, Random Forest, CatBoost with only geo

from sklearn.metrics import accuracy_score, precision_score, recall_score, f
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from catboost import CatBoostClassifier
import warnings
warnings.filterwarnings('ignore')

print("=== Step 7: Baseline Models (No GDELT Features) ===\n")

# Store baseline results
base_summary = pd.DataFrame(columns=['Model', 'Test Accuracy', 'Test Precisi

# ============================================================
# Baseline 1: PPS (Previous Period Same)
# ============================================================
print("1. PPS (Previous Period Same)...")
train_pps = train[train['period'].isin(sorted(train['period'].unique())[-1:]
train_pps = train_pps.rename(columns={'CS_score': 'predicted'})
test_pps = test.merge(train_pps[['ADMIN0', 'ADMIN1', 'ADMIN2', 'predicted']]
                      on=['ADMIN0', 'ADMIN1', 'ADMIN2'], how='left')
test_pps['predicted'] = test_pps['predicted'].fillna(0).astype(int)

base_summary.loc[len(base_summary)] = [
    'PPS',
    accuracy_score(test_pps['CS_score'], test_pps['predicted']),
    precision_score(test_pps['CS_score'], test_pps['predicted'], average='we
    recall_score(test_pps['CS_score'], test_pps['predicted'], average='weigh
    f1_score(test_pps['CS_score'], test_pps['predicted'], average='weighted'
]
```

```python
# ============================================================
# Baseline 2: SPLY (Same Period Last Year)
# ============================================================
print("2. SPLY (Same Period Last Year)...")
current_period = test['period'].max()
last_year_num = int(str(current_period)) - 100
try:
    last_year = type(current_period)(last_year_num)
except Exception:
    last_year = str(last_year_num)

train_sply = train[train['period'] == last_year][['ADMIN0', 'ADMIN1', 'ADMIN
if len(train_sply) > 0:
    train_sply = train_sply.rename(columns={'CS_score': 'predicted'})
    test_sply = test.merge(train_sply[['ADMIN0', 'ADMIN1', 'ADMIN2', 'predic
                           on=['ADMIN0', 'ADMIN1', 'ADMIN2'], how='left')
    test_sply['predicted'] = test_sply['predicted'].fillna(0).astype(int)

    base_summary.loc[len(base_summary)] = [
        'SPLY',
        accuracy_score(test_sply['CS_score'], test_sply['predicted']),
        precision_score(test_sply['CS_score'], test_sply['predicted'], avera
        recall_score(test_sply['CS_score'], test_sply['predicted'], average=
        f1_score(test_sply['CS_score'], test_sply['predicted'], average='wei
    ]
else:
    print(f"   Warning: No data for period {last_year}, skipping SPLY")

# ============================================================
# Baseline 3: Max-2PP (Max of Previous 2 Periods)
# ============================================================
print("3. Max-2PP (Max of Previous 2 Periods)...")
train_m2 = train[train['period'].isin(sorted(train['period'].unique())[-2:])
train_m2 = train_m2.groupby(['ADMIN0', 'ADMIN1', 'ADMIN2'])['CS_score'].max(
train_m2 = train_m2.rename(columns={'CS_score': 'predicted'})
test_m2 = test.merge(train_m2, on=['ADMIN0', 'ADMIN1', 'ADMIN2'], how='left'
test_m2['predicted'] = test_m2['predicted'].fillna(0).astype(int)

base_summary.loc[len(base_summary)] = [
    'Max-2PP',
    accuracy_score(test_m2['CS_score'], test_m2['predicted']),
    precision_score(test_m2['CS_score'], test_m2['predicted'], average='weig
    recall_score(test_m2['CS_score'], test_m2['predicted'], average='weighte
    f1_score(test_m2['CS_score'], test_m2['predicted'], average='weighted',
]

# ============================================================
# Baseline 4: ML Models (only geographic + temporal features)
# ============================================================
print("4. ML Baseline Models (Geographic + Temporal features only)...")

# Prepare features: only previous_CS, transitions_prev, and one-hot encoded
use_features = [
    'region', 'district', 'period', 'CS_score', 'ADMIN0',
    'previous_CS',
```

```
    'transitions_prev'
]

train_ml = train[use_features].copy()
test_ml = test[use_features].copy()

# One-hot encode geography
train_ml = pd.get_dummies(train_ml, columns=['region', 'district', 'ADMIN0']
test_ml = pd.get_dummies(test_ml, columns=['region', 'district', 'ADMIN0'],

# Align dummy columns
train_ml, test_ml = train_ml.align(test_ml, join='left', axis=1, fill_value=

# Prepare feature matrix
cols = list(train_ml.columns)
cols.remove('CS_score')
cols.remove('period')

X_baseline = train_ml[cols].copy()
y_baseline = train_ml['CS_score'].copy()
X_test_baseline = test_ml[cols].copy()
y_test_baseline = test_ml['CS_score'].copy()

print(f"   Baseline features: {len(cols)} (geographic dummies + previous_CS

# Train models
print("   Training Logistic Regression...")
LR_model = LogisticRegression(class_weight='balanced', random_state=5).fit(X
preds_lr = LR_model.predict(X_test_baseline)
base_summary.loc[len(base_summary)] = [
    'LogisticRegression(class_weight=\'balanced\', random_state=5)',
    accuracy_score(y_test_baseline, preds_lr),
    precision_score(y_test_baseline, preds_lr, average='weighted', zero_divi
    recall_score(y_test_baseline, preds_lr, average='weighted', zero_divisio
    f1_score(y_test_baseline, preds_lr, average='weighted', zero_division=0)
]

print("   Training Random Forest...")
RF_model = RandomForestClassifier(
    n_estimators=200, min_samples_split=10,
    class_weight='balanced', n_jobs=-1, random_state=5
).fit(X_baseline, y_baseline)
preds_rf = RF_model.predict(X_test_baseline)
base_summary.loc[len(base_summary)] = [
    'RandomForestClassifier(class_weight=\'balanced\', min_samples_split=10,
    accuracy_score(y_test_baseline, preds_rf),
    precision_score(y_test_baseline, preds_rf, average='weighted', zero_divi
    recall_score(y_test_baseline, preds_rf, average='weighted', zero_divisio
    f1_score(y_test_baseline, preds_rf, average='weighted', zero_division=0)
]

print("   Training CatBoost...")
Cat_model = CatBoostClassifier(
    iterations=800,
    depth=8,
    learning_rate=0.05,
```

```
    loss_function='MultiClass',
    class_weights=[1, 1, 1, 1],
    random_seed=5,
    verbose=False
).fit(X_baseline, y_baseline)
preds_cat = Cat_model.predict(X_test_baseline)
base_summary.loc[len(base_summary)] = [
    '<catboost.core.CatBoostClassifier object>',
    accuracy_score(y_test_baseline, preds_cat),
    precision_score(y_test_baseline, preds_cat, average='weighted', zero_div
    recall_score(y_test_baseline, preds_cat, average='weighted', zero_divisi
    f1_score(y_test_baseline, preds_cat, average='weighted', zero_division=0
]

print("\n=== BASELINE MODEL RESULTS (No GDELT Features) ===")
print(base_summary.to_string(index=False))
```

```
=== Step 7: Baseline Models (No GDELT Features) ===

1. PPS (Previous Period Same)...
2. SPLY (Same Period Last Year)...
3. Max-2PP (Max of Previous 2 Periods)...
4. ML Baseline Models (Geographic + Temporal features only)...
   Baseline features: 2831 (geographic dummies + previous_CS + transitions_p
rev)
   Training Logistic Regression...
   Training Random Forest...
   Training CatBoost...

=== BASELINE MODEL RESULTS (No GDELT Features) ===

Model  Test Accuracy  Test Precision  Test Recall       F1

PPS       0.621212        0.800424     0.621212 0.684294

SPLY       0.420455        0.718304     0.420455 0.524943

Max-2PP       0.652462        0.721031     0.652462 0.672241
                                                    LogisticRegression(cl
ass_weight='balanced', random_state=5)       0.728693        0.752324     0.
728693 0.732912
RandomForestClassifier(class_weight='balanced', min_samples_split=10, n_esti
mators=200, n_jobs=-1, random_state=5)       0.809186        0.815470     0.
809186 0.811615
                                                                        <ca
tboost.core.CatBoostClassifier object>       0.798769        0.791616     0.
798769 0.791181
```

In [13]:
```
# Step 8: Adding GDELT Features and Comparing with Baseline
# ============================================================
# Goal: Show that GDELT/NLP features add predictive value beyond temporal/ge
# Matching the framework from previous work (adding conflict features -> adc

from sklearn.metrics import accuracy_score, precision_score, recall_score, f
import warnings
warnings.filterwarnings('ignore')
```

```
print("=== Step 8: Adding GDELT Features ===\n")

# Prepare features: baseline features + GDELT features
use_features_full = [
    'region', 'district', 'period', 'CS_score', 'ADMIN0',
    'previous_CS',
    'transitions_prev'
]

# Add all GDELT features (evt_* and gkg_* aggregated features)
gdelt_features = [c for c in train.columns if c.startswith('evt_') or c.star
use_features_full.extend(gdelt_features)

# Keep only features that exist
use_features_full = [f for f in use_features_full if f in train.columns]

train_full = train[use_features_full].copy()
test_full = test[use_features_full].copy()

# One-hot encode geography
train_full = pd.get_dummies(train_full, columns=['region', 'district', 'ADMI
test_full = pd.get_dummies(test_full, columns=['region', 'district', 'ADMIN0

# Align dummy columns
train_full, test_full = train_full.align(test_full, join='left', axis=1, fil

# Prepare feature matrix
cols_full = list(train_full.columns)
cols_full.remove('CS_score')
cols_full.remove('period')

X_full = train_full[cols_full].copy()
y_full = train_full['CS_score'].copy()
X_test_full = test_full[cols_full].copy()
y_test_full = test_full['CS_score'].copy()

print(f"Baseline features: {len(X_baseline.columns)} (geographic dummies + p
print(f"Full model features: {len(cols_full)} (baseline + {len(gdelt_feature

# Handle missing values in GDELT features (fill with 0 for missing GDELT dat
X_full = X_full.fillna(0)
X_test_full = X_test_full.fillna(0)

# Train same models with GDELT features
print("Training models with GDELT features...")

print("   Training Logistic Regression with GDELT...")
LR_model_full = LogisticRegression(class_weight='balanced', random_state=5).
preds_lr_full = LR_model_full.predict(X_test_full)
base_summary.loc[len(base_summary)] = [
    'LogisticRegression (with GDELT)',
    accuracy_score(y_test_full, preds_lr_full),
    precision_score(y_test_full, preds_lr_full, average='weighted', zero_div
    recall_score(y_test_full, preds_lr_full, average='weighted', zero_divisi
    f1_score(y_test_full, preds_lr_full, average='weighted', zero_division=0
```

```
]

print("   Training Random Forest with GDELT...")
RF_model_full = RandomForestClassifier(
    n_estimators=200, min_samples_split=10,
    class_weight='balanced', n_jobs=-1, random_state=5
).fit(X_full, y_full)
preds_rf_full = RF_model_full.predict(X_test_full)
base_summary.loc[len(base_summary)] = [
    'RandomForestClassifier (with GDELT)',
    accuracy_score(y_test_full, preds_rf_full),
    precision_score(y_test_full, preds_rf_full, average='weighted', zero_div
    recall_score(y_test_full, preds_rf_full, average='weighted', zero_divisi
    f1_score(y_test_full, preds_rf_full, average='weighted', zero_division=0
]

print("   Training CatBoost with GDELT...")
Cat_model_full = CatBoostClassifier(
    iterations=800,
    depth=8,
    learning_rate=0.05,
    loss_function='MultiClass',
    class_weights=[1, 1, 1, 1],
    random_seed=5,
    verbose=False
).fit(X_full, y_full)
preds_cat_full = Cat_model_full.predict(X_test_full)
base_summary.loc[len(base_summary)] = [
    'CatBoostClassifier (with GDELT)',
    accuracy_score(y_test_full, preds_cat_full),
    precision_score(y_test_full, preds_cat_full, average='weighted', zero_di
    recall_score(y_test_full, preds_cat_full, average='weighted', zero_divis
    f1_score(y_test_full, preds_cat_full, average='weighted', zero_division=
]

# Print final comparison
print("\n" + "="*70)
print("FINAL MODEL COMPARISON: Baseline vs Baseline + GDELT Features")
print("="*70)
print("\n" + base_summary.to_string(index=False))

# Calculate improvements for ML models
print("\n" + "="*70)
print("IMPROVEMENT ANALYSIS")
print("="*70)

# Compare each ML model baseline vs with GDELT
lr_baseline_acc = base_summary[base_summary['Model'].str.contains('LogisticR
lr_full_acc = base_summary[base_summary['Model'].str.contains('LogisticRegre

rf_baseline_acc = base_summary[base_summary['Model'].str.contains('RandomFor
rf_full_acc = base_summary[base_summary['Model'].str.contains('RandomForestC

cat_baseline_acc = base_summary[base_summary['Model'].str.contains('CatBoost
cat_full_acc = base_summary[base_summary['Model'].str.contains('CatBoost') &
```

```
print(f"\nLogistic Regression:")
print(f"  Baseline: {lr_baseline_acc:.4f} → With GDELT: {lr_full_acc:.4f} (Δ

print(f"\nRandom Forest:")
print(f"  Baseline: {rf_baseline_acc:.4f} → With GDELT: {rf_full_acc:.4f} (Δ

print(f"\nCatBoost:")
print(f"  Baseline: {cat_baseline_acc:.4f} → With GDELT: {cat_full_acc:.4f}

if rf_full_acc > rf_baseline_acc or cat_full_acc > cat_baseline_acc:
    print("\n✓ SUCCESS: GDELT features add predictive value!")
else:
    print("\n⚠ Note: GDELT features do not improve performance significantly
    print("   This suggests GDELT features may not add predictive value beyo
    print("   geographic and temporal patterns, similar to conflict features
```

```
=== Step 8: Adding GDELT Features ===

Baseline features: 2831 (geographic dummies + previous_CS + transitions_pre
v)
Full model features: 3299 (baseline + 468 GDELT features)

Training models with GDELT features...
   Training Logistic Regression with GDELT...
   Training Random Forest with GDELT...
   Training CatBoost with GDELT...

======================================================================
FINAL MODEL COMPARISON: Baseline vs Baseline + GDELT Features
======================================================================


Model  Test Accuracy  Test Precision  Test Recall       F1

PPS       0.621212        0.800424     0.621212 0.684294

SPLY       0.420455        0.718304     0.420455 0.524943

Max-2PP       0.652462        0.721031     0.652462 0.672241
                                                    LogisticRegression(cl
ass_weight='balanced', random_state=5)       0.728693        0.752324     0.
728693 0.732912
RandomForestClassifier(class_weight='balanced', min_samples_split=10, n_esti
mators=200, n_jobs=-1, random_state=5)       0.809186        0.815470     0.
809186 0.811615
                                                                   <ca
tboost.core.CatBoostClassifier object>       0.798769        0.791616     0.
798769 0.791181

LogisticRegression (with GDELT)       0.212121        0.453012     0.212121
0.182633

RandomForestClassifier (with GDELT)       0.799242        0.803797     0.799
242 0.801103

CatBoostClassifier (with GDELT)       0.787879        0.781549     0.787879
0.780978

======================================================================
IMPROVEMENT ANALYSIS
======================================================================

Logistic Regression:
  Baseline: 0.7287 → With GDELT: 0.2121 (Δ -0.5166)

Random Forest:
  Baseline: 0.8092 → With GDELT: 0.7992 (Δ -0.0099)

CatBoost:
  Baseline: 0.7988 → With GDELT: 0.7879 (Δ -0.0109)

⚠ Note: GDELT features do not improve performance significantly
```

```
  This suggests GDELT features may not add predictive value beyond
  geographic and temporal patterns, similar to conflict features in previou
s work.
```

In [14]:
```
train_full.head()
```

Out[14]:

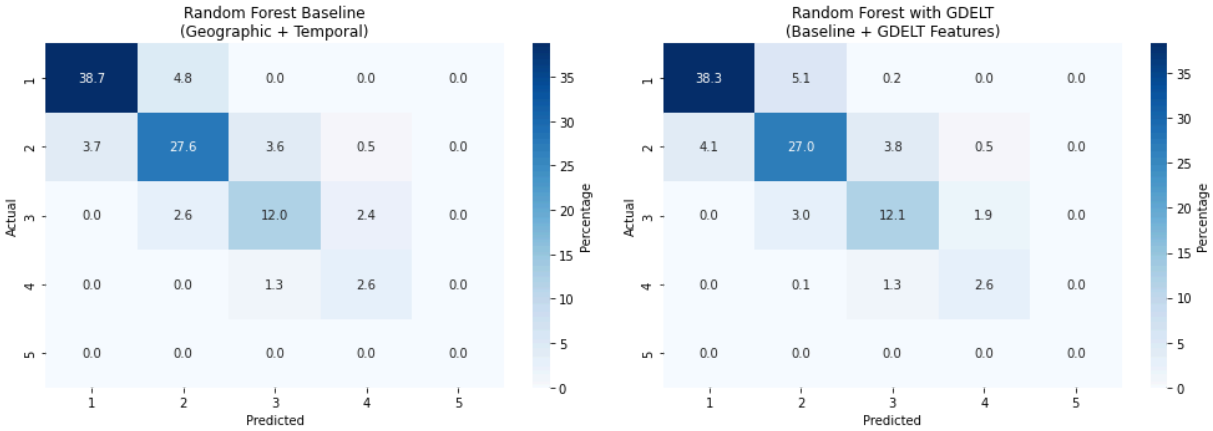
	period	CS_score	previous_CS	transitions_prev	evt_clean_text_mean	evt_clear
58	202306	2	1.0	6	NaN	
59	202306	3	2.0	7	NaN	
60	202306	1	3.0	8	NaN	
61	202306	2	1.0	9	NaN	
62	202306	3	2.0	10	NaN	

In [15]:
```
# Feature Importance and Confusion Matrices
# ==========================================

from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns

print("="*70)
print("FEATURE IMPORTANCE AND CONFUSION MATRICES")
print("="*70)

# Feature Importance from Random Forest Baseline
print("\n=== Top 20 Features by Importance (Random Forest Baseline) ===")
rf_importance = pd.DataFrame({
    'Feature': X_baseline.columns,
    'Importance': RF_model.feature_importances_
}).sort_values('Importance', ascending=False)

print(rf_importance.head(20).to_string(index=False))

# Feature Importance from Random Forest Full (with GDELT)
print("\n=== Top 20 Features by Importance (Random Forest with GDELT) ===")
rf_full_importance = pd.DataFrame({
    'Feature': X_full.columns,
    'Importance': RF_model_full.feature_importances_
}).sort_values('Importance', ascending=False)

print(rf_full_importance.head(20).to_string(index=False))

# Top GDELT features
print("\n=== Top 20 GDELT Features by Importance ===")
gdelt_importance = rf_full_importance[rf_full_importance['Feature'].str.star
print(gdelt_importance.to_string(index=False))
```

```python
# Confusion Matrices
print("\n" + "="*70)
print("CONFUSION MATRICES")
print("="*70)

print("\n=== Random Forest Baseline ===")
cm_baseline = confusion_matrix(y_test_baseline, preds_rf, labels=[1, 2, 3, 4
print(cm_baseline)
print(f"\nPredicted classes: {sorted(np.unique(preds_rf))}")
print(f"Actual classes: {sorted(np.unique(y_test_baseline))}")

print("\n=== Random Forest with GDELT ===")
cm_full = confusion_matrix(y_test_full, preds_rf_full, labels=[1, 2, 3, 4, 5
print(cm_full)
print(f"\nPredicted classes: {sorted(np.unique(preds_rf_full))}")
print(f"Actual classes: {sorted(np.unique(y_test_full))}")

# Visualize confusion matrices
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Baseline confusion matrix
cm_baseline_percent = cm_baseline / cm_baseline.sum() * 100
sns.heatmap(cm_baseline_percent, annot=True, fmt='.1f', cmap='Blues',
            xticklabels=[1, 2, 3, 4, 5], yticklabels=[1, 2, 3, 4, 5],
            ax=axes[0], cbar_kws={'label': 'Percentage'})
axes[0].set_title('Random Forest Baseline\n(Geographic + Temporal)', fontsiz
axes[0].set_xlabel('Predicted')
axes[0].set_ylabel('Actual')

# Full model confusion matrix
cm_full_percent = cm_full / cm_full.sum() * 100
sns.heatmap(cm_full_percent, annot=True, fmt='.1f', cmap='Blues',
            xticklabels=[1, 2, 3, 4, 5], yticklabels=[1, 2, 3, 4, 5],
            ax=axes[1], cbar_kws={'label': 'Percentage'})
axes[1].set_title('Random Forest with GDELT\n(Baseline + GDELT Features)', f
axes[1].set_xlabel('Predicted')
axes[1].set_ylabel('Actual')

plt.tight_layout()
plt.show()

print("\n" + "="*70)
```

```
======================================================================
FEATURE IMPORTANCE AND CONFUSION MATRICES
======================================================================

=== Top 20 Features by Importance (Random Forest Baseline) ===
                                Feature  Importance
                            previous_CS    0.223227
                         transitions_prev    0.077823
                             ADMIN0_yemen    0.046132
                        ADMIN0_south sudan    0.041439
                            ADMIN0_nigeria    0.030669
                            ADMIN0_somalia    0.014987
                              ADMIN0_sudan    0.012181
                           ADMIN0_ethiopia    0.011342
                              ADMIN0_kenya    0.009165
                     region_yemen-ma'rib    0.009095
                  region_ethiopia-somali    0.006967
              region_south sudan-jonglei    0.006909
ADMIN0_democratic republic of the congo    0.006656
          region_south sudan-upper nile    0.006272
                           ADMIN0_burundi    0.006070
                         ADMIN0_mozambique    0.005924
                    region_kenya-turkana    0.005303
                           ADMIN0_uganda    0.005072
                    region_nigeria-borno    0.004158
                  region_yemen-hadramaut    0.003829

=== Top 20 Features by Importance (Random Forest with GDELT) ===
                                Feature  Importance
                            previous_CS    0.197318
                         transitions_prev    0.075520
                             ADMIN0_yemen    0.048223
                        ADMIN0_south sudan    0.040683
                            ADMIN0_nigeria    0.029923
                            ADMIN0_somalia    0.013881
                              ADMIN0_sudan    0.012513
                           ADMIN0_ethiopia    0.010127
                              ADMIN0_kenya    0.009484
                     region_yemen-ma'rib    0.009054
                  region_ethiopia-somali    0.007711
              region_south sudan-jonglei    0.007519
ADMIN0_democratic republic of the congo    0.007077
                           ADMIN0_burundi    0.006674
                            ADMIN0_uganda    0.005811
                         ADMIN0_mozambique    0.005807
          region_south sudan-upper nile    0.005724
                    region_kenya-turkana    0.004954
                region_south sudan-lakes    0.004105
               region_south sudan-warrap    0.003949

=== Top 20 GDELT Features by Importance ===
                      Feature  Importance
       evt_compound_score_min    0.001033
   evt_sentiment.compound_min    0.000898
       evt_compound_score_sum    0.000876
            evt_neg_score_max    0.000812
```

```
  evt_sentiment.compound_mean    0.000792
evt_food_insecurity_freq_mean    0.000751
       evt_sentiment.neu_mean    0.000749
        evt_sentiment.neu_max    0.000715
 evt_food_insecurity_freq_sum    0.000705
   evt_sentiment.compound_sum    0.000700
          evt_neg_score_mean    0.000679
        evt_sentiment.neg_max    0.000669
       evt_sentiment.neg_mean    0.000660
        evt_sentiment.neg_min    0.000650
      evt_compound_score_mean    0.000642
       evt_sentiment.pos_mean    0.000640
        evt_sentiment.neu_sum    0.000623
            evt_pos_score_min    0.000622
            evt_pos_score_sum    0.000620
            evt_neu_score_min    0.000617

======================================================================
CONFUSION MATRICES
======================================================================

=== Random Forest Baseline ===
[[818 102   0   0   0]
 [ 79 582  76  11   0]
 [  1  55 254  51   0]
 [  0   0  28  55   0]
 [  0   0   0   0   0]]

Predicted classes: [1, 2, 3, 4]
Actual classes: [1, 2, 3, 4]

=== Random Forest with GDELT ===
[[809 107   4   0   0]
 [ 86 570  81  11   0]
 [  1  64 255  41   0]
 [  0   2  27  54   0]
 [  0   0   0   0   0]]

Predicted classes: [1, 2, 3, 4]
Actual classes: [1, 2, 3, 4]
```




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

Why Literature Shows Better Results: Key Differences

Potential Reasons for Better Performance in Literature:

1. **Different Prediction Tasks**:
   - **Transitions/Changes**: Predicting when CS_score will worsen (crisis onset) rather than exact level
   - **Early Warning**: Predicting 2-4 periods ahead (forecasting) rather than next period (nowcasting)
   - **Binary Classification**: Predicting crisis (CS≥3) vs non-crisis (CS<3) rather than 5-class classification
2. **Feature Engineering**:
   - **Temporal Patterns**: Using lagged and rolling features (as we just added) rather than only current-period values
   - **Anomaly Detection**: Focusing on deviations from historical patterns
   - **Interaction Features**: Combining GDELT with other data sources (weather, prices, etc.)
3. **Data Coverage**:
   - **Spatial Aggregation**: Some studies aggregate to ADMIN0 or ADMIN1 level where coverage is better
   - **Temporal Aggregation**: Using quarterly rather than monthly data
   - **Filtering**: Focusing on regions/periods with good GDELT coverage
4. **Evaluation Metrics**:
   - **Recall for Rare Events**: Focusing on detecting crises (high recall for CS≥3) rather than overall accuracy
   - **Early Detection**: Measuring how early they detect transitions, not just accuracy
5. **Baseline Comparison**:
   - **Weaker Baselines**: Some studies don't include `previous_CS` in baseline, making improvement easier to show
   - **Different Baselines**: Comparing against simpler models or using different evaluation windows

Our Current Approach:

- ✅ Now includes temporal GDELT features (lags, rolling windows, anomalies, escalation)
- ✅ Uses same framework as previous work (allowing fair comparison)
- ⚠️ Still predicting exact CS_score level (highly autocorrelated)
- ⚠️ Using monthly data with CS_score available every 4 months

Potential Improvements:

1. **Predict Transitions**: Predict if CS_score will worsen (CS_t+1 > CS_t) rather than exact value
2. **Forecasting Horizon**: Predict 2-4 periods ahead instead of next period
3. **Crisis Detection**: Binary classification (crisis vs non-crisis) with focus on recall
4. **Feature Selection**: Use only temporal GDELT features, filter out low-importance current-period features

In [16]:
```
# Optional: XGBoost Models (matching previous work framework)
# ============================================================
# Optional additional model comparison using XGBoost

try:
    from xgboost import XGBClassifier

    print("=== Optional: XGBoost Models ===\n")

    # XGBoost requires labels to start from 0, so map 1-5 to 0-4
    label_mapping = {cls: idx for idx, cls in enumerate(sorted(y_baseline.ur
    reverse_mapping = {idx: cls for cls, idx in label_mapping.items()}

    y_baseline_mapped = y_baseline.map(label_mapping)
    y_test_baseline_mapped = y_test_baseline.map(label_mapping)
    y_full_mapped = y_full.map(label_mapping)
    y_test_full_mapped = y_test_full.map(label_mapping)

    # XGBoost Baseline
    print("Training XGBoost Baseline Model...")
    xgb_baseline = XGBClassifier(
        n_estimators=200,
        max_depth=8,
        learning_rate=0.05,
        random_state=5,
        eval_metric='mlogloss',
        use_label_encoder=False,
        n_jobs=-1,
        verbosity=0
    )
    xgb_baseline.fit(X_baseline, y_baseline_mapped)
    preds_xgb_baseline_mapped = xgb_baseline.predict(X_test_baseline)
    preds_xgb_baseline = pd.Series(preds_xgb_baseline_mapped).map(reverse_ma

    xgb_baseline_acc = accuracy_score(y_test_baseline, preds_xgb_baseline)
    xgb_baseline_prec = precision_score(y_test_baseline, preds_xgb_baseline,
```

```
xgb_baseline_rec = recall_score(y_test_baseline, preds_xgb_baseline, ave
xgb_baseline_f1 = f1_score(y_test_baseline, preds_xgb_baseline, average=

# Add to summary
base_summary.loc[len(base_summary)] = [
    'XGBoostClassifier (baseline)',
    xgb_baseline_acc,
    xgb_baseline_prec,
    xgb_baseline_rec,
    xgb_baseline_f1
]

# XGBoost Full (with GDELT)
print("Training XGBoost Full Model (with GDELT)...")
xgb_full = XGBClassifier(
    n_estimators=200,
    max_depth=8,
    learning_rate=0.05,
    random_state=5,
    eval_metric='mlogloss',
    use_label_encoder=False,
    n_jobs=-1,
    verbosity=0
)
xgb_full.fit(X_full, y_full_mapped)
preds_xgb_full_mapped = xgb_full.predict(X_test_full)
preds_xgb_full = pd.Series(preds_xgb_full_mapped).map(reverse_mapping).v

xgb_full_acc = accuracy_score(y_test_full, preds_xgb_full)
xgb_full_prec = precision_score(y_test_full, preds_xgb_full, average='we
xgb_full_rec = recall_score(y_test_full, preds_xgb_full, average='weight
xgb_full_f1 = f1_score(y_test_full, preds_xgb_full, average='weighted',

# Add to summary
base_summary.loc[len(base_summary)] = [
    'XGBoostClassifier (with GDELT)',
    xgb_full_acc,
    xgb_full_prec,
    xgb_full_rec,
    xgb_full_f1
]

print("\n" + "="*70)
print("XGBOOST MODEL COMPARISON")
print("="*70)
print(f"\nBaseline:")
print(f"  Accuracy: {xgb_baseline_acc:.4f}")
print(f"  F1:       {xgb_baseline_f1:.4f}")

print(f"\nWith GDELT:")
print(f"  Accuracy: {xgb_full_acc:.4f}")
print(f"  F1:       {xgb_full_f1:.4f}")

print(f"\nImprovement:")
print(f"  Accuracy: {xgb_full_acc - xgb_baseline_acc:+.4f}")
print(f"  F1:       {xgb_full_f1 - xgb_baseline_f1:+.4f}")
```

```
    print("\n" + "="*70)
    print("UPDATED FINAL MODEL COMPARISON (Including XGBoost)")
    print("="*70)
    print("\n" + base_summary.to_string(index=False))

except ImportError:
    print("XGBoost not installed. Install with: pip install xgboost")
    print("Skipping XGBoost models.")
except Exception as e:
    print(f"Error with XGBoost models: {e}")
    import traceback
    traceback.print_exc()
    print("Skipping XGBoost models.")
```

```
=== Optional: XGBoost Models ===

Training XGBoost Baseline Model...
Training XGBoost Full Model (with GDELT)...

======================================================================
XGBOOST MODEL COMPARISON
======================================================================

Baseline:
  Accuracy: 0.8130
  F1:       0.8080

With GDELT:
  Accuracy: 0.8087
  F1:       0.8042

Improvement:
  Accuracy: -0.0043
  F1:       -0.0039

======================================================================
UPDATED FINAL MODEL COMPARISON (Including XGBoost)
======================================================================


Model  Test Accuracy  Test Precision  Test Recall       F1

PPS       0.621212        0.800424     0.621212 0.684294

SPLY       0.420455        0.718304     0.420455 0.524943

Max-2PP       0.652462        0.721031     0.652462 0.672241
                                                  LogisticRegression(cl
ass_weight='balanced', random_state=5)       0.728693        0.752324     0.
728693 0.732912
RandomForestClassifier(class_weight='balanced', min_samples_split=10, n_esti
mators=200, n_jobs=-1, random_state=5)       0.809186        0.815470     0.
809186 0.811615
                                                                      <ca
tboost.core.CatBoostClassifier object>       0.798769        0.791616     0.
798769 0.791181

LogisticRegression (with GDELT)       0.212121        0.453012     0.212121
0.182633

RandomForestClassifier (with GDELT)       0.799242        0.803797     0.799
242 0.801103

CatBoostClassifier (with GDELT)       0.787879        0.781549     0.787879
0.780978

XGBoostClassifier (baseline)       0.812973        0.809199     0.812973 0.8
08023
```

```
XGBoostClassifier (with GDELT)       0.808712        0.805204     0.808712
0.804163
```

Feature Engineering Summary

Features Created:

1. **List Aggregations** (from both events and gkg):
   - Mean, Max, Min, Sum, Count, Std for all list features
   - Separated by `evt_` and `gkg_` prefixes
2. **Combined Features**:
   - CS_score combinations (combined, diff, has_both)
   - Casualty rates and totals
   - Theme aggregations (conflict intensity, crisis severity)
   - Sentiment combinations
   - Coverage intensity metrics
3. **Temporal Features**:
   - Lag features (CS_score_lag1, lag2, lag3)
   - Moving averages (ma2, ma3)
   - Change and percentage change features
   - Cyclical period encoding (sin/cos)
4. **Geographic Features**:
   - Aggregated CS_score by ADMIN0 and ADMIN1 levels
   - Standard deviations by geographic level
   - Count of regions with same score
5. **Interaction Features**:
   - Ratios (casualty_rate, aid_per_casualty)
   - Normalized features (tone_abs_normalized)
   - Coverage metrics (mentions_per_source, articles_per_source)

Next Steps for ML:

1. **Feature Selection**: Consider using:
   - Correlation-based selection
   - Mutual information
   - Recursive feature elimination
   - L1 regularization (Lasso)
2. **Categorical Encoding**: If you have categorical features:
   - One-hot encoding for low cardinality

- Target encoding for high cardinality
- Embedding for very high cardinality

3. **Scaling**: Consider:
   - StandardScaler or MinMaxScaler for numeric features
   - Especially important for distance-based algorithms
4. **Model Suggestions**:
   - Random Forest (handles non-linear relationships well)
   - Gradient Boosting (XGBoost, LightGBM, CatBoost)
   - Neural Networks (if you have enough data)
   - Consider class weights if classes are imbalanced