Aadhaar Data Analysis

Comprehensive Insights & Recommendations

Generated on: January 05, 2026
CONFIDENTIAL

Executive Summary

The Problem

Analysis of current enrollment data reveals significant geographic disparities. 15% of states account for the majority of enrolments, while bottom states like Lakshadweep, Dadra & Nagar Haveli and Daman & Diu, Daman & Diu remain critically underserved, accounting for only 0.01% of the total.

The Insight

Enrolments peak in July (+185.0% above average), linking to school cycles. Furthermore, anomaly detection indicates systematic patterns: Of 71 anomalies, 2 (2.8%) occurred on month-ends, suggesting batch processing.

Strategic Recommendations

- Deploy mobile units to the 38 identified priority districts immediately.
- Optimize campaign timing to coincide with the July surge.
- Launch targeted youth biometric update campaigns in low-transition states.

Key Quantitative Findings

- Children (0-5) dominate enrolments at 65%, driving the need for early-age biometric update tracking.
- Anomalies are systematic: 3% align with month-end batch processing, not random errors.
- Geographic disparity is critical: Bottom 5 states hold only 0.01% of enrolments despite significant population.
- Seasonal surge in July indicates effective campaign timing windows.

Deep Dive Analysis

1. Geographic & Demographic Trends

The following dashboard provides a holistic view of current enrollment status, age distribution, and state-wise performance.

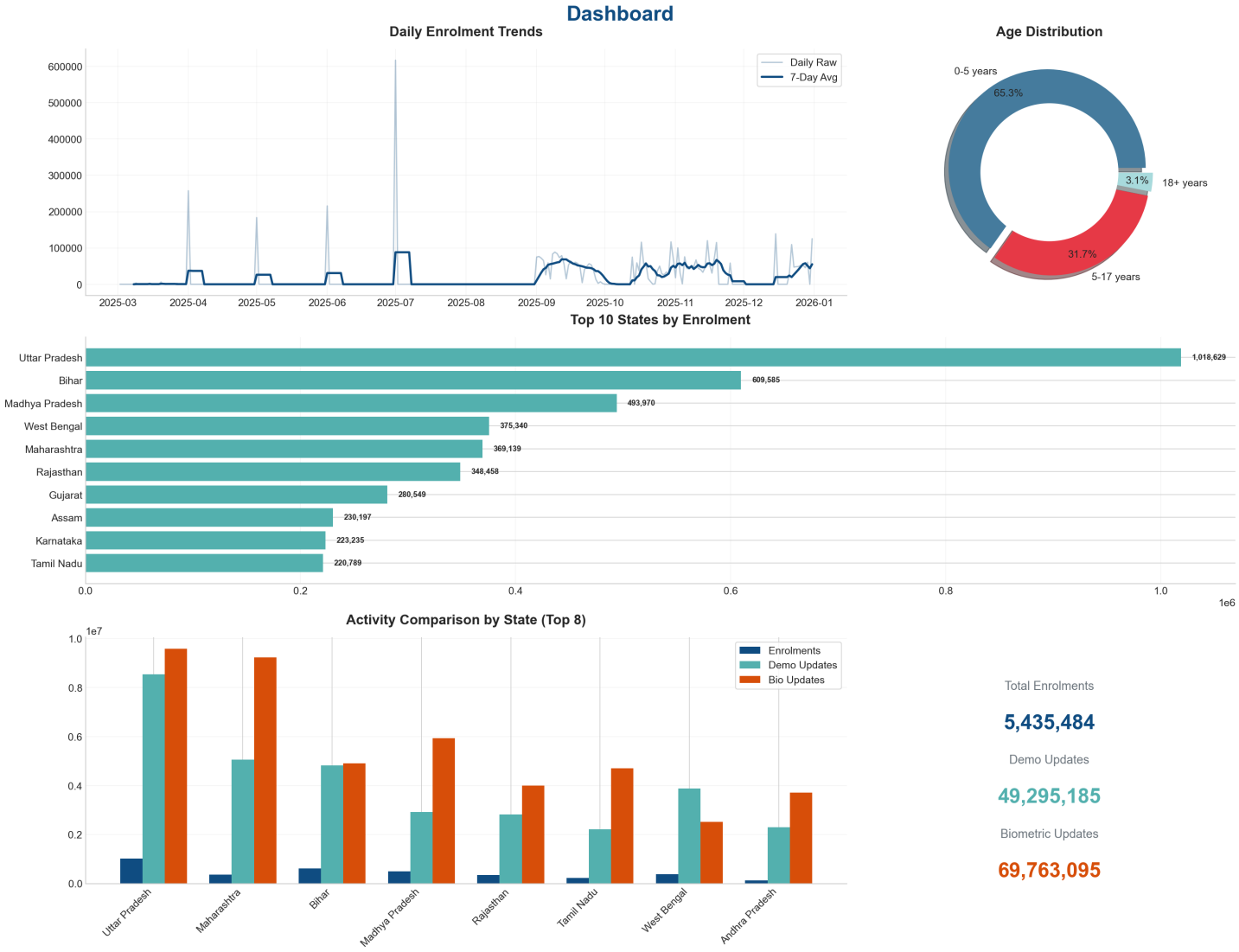

Figure 1: Comprehensive Activity Dashboard

2. Seasonality & Temporal Patterns

Understanding monthly enrollment patterns is crucial for resource allocation. Our heatmap analysis reveals clear seasonal trends.

Figure 2: Month-over-Year Enrollment Heatmap

3. Anomaly Detection

Of 71 anomalies, 2 (2.8%) occurred on month-ends, suggesting batch processing.

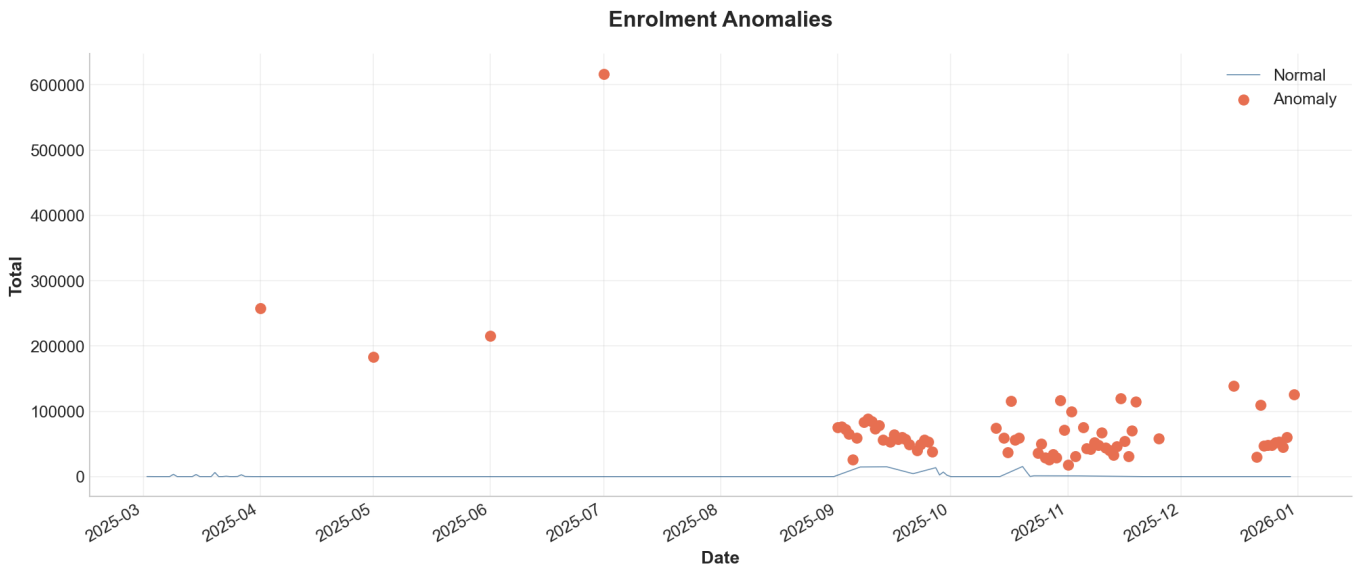

Figure 3: Time-series Anomaly Detection

Technical Implementation

This analysis was generated using a custom Python pipeline. Below are the key modules used for processing and analysis.

Analysis Logic (src/analysis.py)

src/analysis.py

```
import pandas as pd
import numpy as np
from scipy import stats
from typing import Dict, List, Tuple, Optional

def temporal_trends(df: pd.DataFrame, value_col: str, date_col: str = 'date',
                    freq: str = 'D') -> pd.DataFrame:
    """Aggregate values by time frequency and calculate trends."""
    daily = df.groupby(pd.Grouper(key=date_col, freq=freq))[value_col].sum().reset_index()
    daily.columns = ['date', 'total']

    daily['rolling_7d'] = daily['total'].rolling(window=7, min_periods=1).mean()
    daily['rolling_30d'] = daily['total'].rolling(window=30, min_periods=1).mean()
    daily['pct_change'] = daily['total'].pct_change() * 100
    daily['cumulative'] = daily['total'].cumsum()

    return daily

def state_aggregations(df: pd.DataFrame, value_col: str) -> pd.DataFrame:
    """Aggregate values by state with rankings."""
    state_totals = df.groupby('state')[value_col].sum().reset_index()
    state_totals.columns = ['state', 'total']
    state_totals = state_totals.sort_values('total', ascending=False)
    state_totals['rank'] = range(1, len(state_totals) + 1)
    state_totals['pct_of_total'] = state_totals['total'] / state_totals['total'].sum() * 100
    state_totals['cumulative_pct'] = state_totals['pct_of_total'].cumsum()
    return state_totals

def district_aggregations(df: pd.DataFrame, value_col: str) -> pd.DataFrame:
    """Aggregate values by state and district."""
    district_totals = df.groupby(['state', 'district'])[value_col].sum().reset_index()
    district_totals.columns = ['state', 'district', 'total']
    district_totals = district_totals.sort_values('total', ascending=False)
    return district_totals

def age_group_analysis(enrolment_df: pd.DataFrame) -> pd.DataFrame:
    """Analyze enrolment distribution by age groups."""
    age_cols = ['age_0_5', 'age_5_17', 'age_18_greater']
    totals = enrolment_df[age_cols].sum()

    result = pd.DataFrame({
        'age_group': ['0-5 years', '5-17 years', '18+ years'],
        'total': totals.values,
        'percentage': (totals.values / totals.sum() * 100)
    })
```

```
    return result

def monthly_patterns(df: pd.DataFrame, value_col: str) -> pd.DataFrame:
    """Analyze monthly patterns for seasonality."""
    monthly = df.groupby(['year', 'month'])[value_col].sum().reset_index()
    monthly['year_month'] = pd.to_datetime(monthly[['year', 'month']].assign(day=1))

    month_avg = df.groupby('month')[value_col].mean().reset_index()
    month_avg.columns = ['month', 'avg_value']
    month_avg['month_name'] = pd.to_datetime(month_avg['month'], format='%m').dt.month_name()

    return monthly, month_avg

def detect_anomalies_iqr(df: pd.DataFrame, value_col: str,
                         multiplier: float = 1.5) -> pd.DataFrame:
    """Detect anomalies using IQR method."""
    df = df.copy()
    Q1 = df[value_col].quantile(0.25)
    Q3 = df[value_col].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - multiplier * IQR
    upper_bound = Q3 + multiplier * IQR

    df['is_anomaly'] = (df[value_col] < lower_bound) | (df[value_col] > upper_bound)
    df['anomaly_type'] = np.where(df[value_col] < lower_bound, 'low',
                                  np.where(df[value_col] > upper_bound, 'high', 'normal'))
    return df

def analyze_anomaly_patterns(df: pd.DataFrame, date_col: str = 'date') -> Dict:
    """Analyze patterns of detected anomalies."""
    anomalies = df[df['is_anomaly']].copy()
    if anomalies.empty:
        return {}

    anomalies['day'] = anomalies[date_col].dt.day
    anomalies['weekday'] = anomalies[date_col].dt.day_name()
    anomalies['is_month_end'] = anomalies[date_col].dt.is_month_end
    anomalies['is_month_start'] = anomalies[date_col].dt.is_month_start

    return {
        'total_anomalies': len(anomalies),
        'month_end_count': anomalies['is_month_end'].sum(),
        'month_start_count': anomalies['is_month_start'].sum(),
        'day_counts': anomalies['day'].value_counts().to_dict(),
```

```
        'weekday_counts': anomalies['weekday'].value_counts().to_dict()
    }

def detect_anomalies_zscore(df: pd.DataFrame, value_col: str,
                            threshold: float = 3.0) -> pd.DataFrame:
    """Detect anomalies using Z-score method."""
    df = df.copy()
    df['zscore'] = np.abs(stats.zscore(df[value_col].fillna(0)))
    df['is_anomaly'] = df['zscore'] > threshold
    return df

def growth_rate_analysis(df: pd.DataFrame, date_col: str = 'date',
                         value_col: str = 'total') -> Dict:
    """Calculate growth rates over different periods."""
    df = df.sort_values(date_col)

    if len(df) == 0:
        return {}

       total_growth  =  (df[value_col].iloc[-1]  -  df[value_col].iloc[0])  /  df[value_col].iloc[0]  *  100  if
df[value_col].iloc[0] != 0 else 0

    if len(df) >= 7:
          weekly_growth = (df[value_col].iloc[-1] - df[value_col].iloc[-7]) / df[value_col].iloc[-7] * 100 if
df[value_col].iloc[-7] != 0 else 0
    else:
        weekly_growth = None

    if len(df) >= 30:
         monthly_growth = (df[value_col].iloc[-1] - df[value_col].iloc[-30]) / df[value_col].iloc[-30] * 100 if
df[value_col].iloc[-30] != 0 else 0
    else:
        monthly_growth = None

    return {
        'total_growth_pct': total_growth,
        'weekly_growth_pct': weekly_growth,
        'monthly_growth_pct': monthly_growth,
        'avg_daily': df[value_col].mean(),
        'max_daily': df[value_col].max(),
        'min_daily': df[value_col].min(),
    }

def comparative_state_metrics(enrolment: pd.DataFrame, demographic: pd.DataFrame,
                              biometric: pd.DataFrame) -> pd.DataFrame:
    """Compare enrolment and update rates across states."""
    enrol_state = enrolment.groupby('state')['total_enrolments'].sum().reset_index()
    enrol_state.columns = ['state', 'enrolments']
```

```
    demo_state = demographic.groupby('state')['total_updates'].sum().reset_index()
    demo_state.columns = ['state', 'demo_updates']

    bio_state = biometric.groupby('state')['total_updates'].sum().reset_index()
    bio_state.columns = ['state', 'bio_updates']

    merged = enrol_state.merge(demo_state, on='state', how='outer')
    merged = merged.merge(bio_state, on='state', how='outer')
    merged = merged.fillna(0)

    merged['demo_to_enrol_ratio'] = merged['demo_updates'] / merged['enrolments'].replace(0, np.nan)
    merged['bio_to_enrol_ratio'] = merged['bio_updates'] / merged['enrolments'].replace(0, np.nan)
    merged['bio_to_demo_ratio'] = merged['bio_updates'] / merged['demo_updates'].replace(0, np.nan)
    merged['total_activity'] = merged['enrolments'] + merged['demo_updates'] + merged['bio_updates']

    return merged.sort_values('total_activity', ascending=False)

def identify_cross_dataset_outliers(df: pd.DataFrame) -> pd.DataFrame:
    """Identify states with unusual ratios between datasets."""
    df = df.copy()
    # High enrolment, low updates
    df['high_enrol_low_update'] = (
        (df['enrolments'] > df['enrolments'].quantile(0.75)) &
        (df['demo_to_enrol_ratio'] < df['demo_to_enrol_ratio'].quantile(0.25))
    )
    return df[df['high_enrol_low_update']]

def district_deep_dive(df: pd.DataFrame, states: List[str]) -> pd.DataFrame:
    """Analyze district-level performance for specific states."""
    target_df = df[df['state'].isin(states)]
    district_stats = target_df.groupby(['state', 'district'])['total_enrolments'].sum().reset_index()
    district_stats = district_stats.sort_values(['state', 'total_enrolments'])
    return district_stats

def identify_hotspots(df: pd.DataFrame, value_col: str,
                      percentile: float = 90) -> pd.DataFrame:
    """Identify geographic hotspots based on activity percentile."""
    threshold = df[value_col].quantile(percentile / 100)
    hotspots = df[df[value_col] >= threshold].copy()
    hotspots['hotspot_rank'] = hotspots[value_col].rank(ascending=False)
    return hotspots

def identify_coldspots(df: pd.DataFrame, value_col: str,
                       percentile: float = 10) -> pd.DataFrame:
```

```
    """Identify geographic coldspots based on activity percentile."""
    threshold = df[value_col].quantile(percentile / 100)
    coldspots = df[df[value_col] <= threshold].copy()
    coldspots['coldspot_rank'] = coldspots[value_col].rank(ascending=True)
    return coldspots

def youth_transition_analysis(enrolment: pd.DataFrame, biometric: pd.DataFrame) -> pd.DataFrame:
    """Analyze child-to-adult biometric transition patterns."""
    enrol_youth = enrolment.groupby('state')['age_5_17'].sum().reset_index()
    enrol_youth.columns = ['state', 'youth_enrolments']

    bio_youth = biometric.groupby('state')['bio_age_5_17'].sum().reset_index()
    bio_youth.columns = ['state', 'youth_bio_updates']

    merged = enrol_youth.merge(bio_youth, on='state', how='outer').fillna(0)
    merged['transition_ratio'] = merged['youth_bio_updates'] / merged['youth_enrolments'].replace(0, np.nan)

    return merged.sort_values('transition_ratio', ascending=False)

def weekly_pattern_analysis(df: pd.DataFrame, value_col: str) -> pd.DataFrame:
    """Analyze patterns by day of week."""
    dow_names = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
    dow_totals = df.groupby('day_of_week')[value_col].agg(['sum', 'mean', 'count']).reset_index()
    dow_totals['day_name'] = dow_totals['day_of_week'].map(lambda x: dow_names[x])
    dow_totals = dow_totals.sort_values('day_of_week')
    return dow_totals
```

Preprocessing Logic (src/preprocessing.py)

src/preprocessing.py

```
import pandas as pd
import numpy as np
from typing import Tuple

STATE_NAME_MAP = {
    'Andaman And Nicobar Islands': 'Andaman & Nicobar',
    'Andaman and Nicobar Islands': 'Andaman & Nicobar',
    'Andhra Pradesh': 'Andhra Pradesh',
    'Andhra pradesh': 'Andhra Pradesh',
    'Dadra And Nagar Haveli': 'Dadra & Nagar Haveli',
    'Dadra and Nagar Haveli': 'Dadra & Nagar Haveli',
    'Dadra And Nagar Haveli And Daman And Diu': 'Dadra & Nagar Haveli and Daman & Diu',
    'The Dadra And Nagar Haveli And Daman And Diu': 'Dadra & Nagar Haveli and Daman & Diu',
    'Daman And Diu': 'Daman & Diu',
    'Daman and Diu': 'Daman & Diu',
    'Jammu And Kashmir': 'Jammu & Kashmir',
    'Jammu and Kashmir': 'Jammu & Kashmir',
    'Orissa': 'Odisha',
    'ODISHA': 'Odisha',
    'Pondicherry': 'Puducherry',
    'West Bangal': 'West Bengal',
    'Westbengal': 'West Bengal',
    'West bengal': 'West Bengal',
    'WEST BENGAL': 'West Bengal',
    'WESTBENGAL': 'West Bengal',
    'West  Bengal': 'West Bengal',
}

def parse_dates(df: pd.DataFrame, date_col: str = 'date') -> pd.DataFrame:
    """Convert date strings to datetime objects."""
    df = df.copy()
    df[date_col] = pd.to_datetime(df[date_col], format='%d-%m-%Y', errors='coerce')
    return df

def validate_pincode(df: pd.DataFrame, pincode_col: str = 'pincode') -> pd.DataFrame:
    """Filter to valid 6-digit PIN codes."""
    df = df.copy()
    df[pincode_col] = df[pincode_col].astype(str).str.strip()
    valid_mask = df[pincode_col].str.match(r'^\d{6}$', na=False)
    return df[valid_mask].copy()

def normalize_state_names(df: pd.DataFrame, state_col: str = 'state') -> pd.DataFrame:
    """Standardize state names to consistent format."""
    df = df.copy()
    # Filter out numeric states (bad data)
```

```
    df = df[~df[state_col].astype(str).str.match(r'^\d+$', na=False)]

    # Capitalize properly and strip
    df[state_col] = df[state_col].astype(str).str.strip()
    # Handle specific case-insensitive matches first
    df[state_col] = df[state_col].replace(STATE_NAME_MAP)
    # Then title case
    df[state_col] = df[state_col].str.title()
    # Apply map again to catch any title-cased variations
    df[state_col] = df[state_col].replace(STATE_NAME_MAP)
    return df

def add_temporal_features(df: pd.DataFrame, date_col: str = 'date') -> pd.DataFrame:
    """Extract year, month, quarter, and day features from date column."""
    df = df.copy()
    df['year'] = df[date_col].dt.year
    df['month'] = df[date_col].dt.month
    df['quarter'] = df[date_col].dt.quarter
    df['day_of_week'] = df[date_col].dt.dayofweek
    df['month_name'] = df[date_col].dt.month_name()
    df['week'] = df[date_col].dt.isocalendar().week.astype(int)
    return df

def add_enrolment_totals(df: pd.DataFrame) -> pd.DataFrame:
    """Add total enrolments column for enrolment dataset."""
    df = df.copy()
    age_cols = ['age_0_5', 'age_5_17', 'age_18_greater']
    if all(col in df.columns for col in age_cols):
        df['total_enrolments'] = df[age_cols].sum(axis=1)
    return df

def add_update_totals(df: pd.DataFrame, prefix: str = 'demo') -> pd.DataFrame:
    """Add total updates column for demographic/biometric datasets."""
    df = df.copy()
    update_cols = [col for col in df.columns if col.startswith(prefix)]
    if update_cols:
        df['total_updates'] = df[update_cols].sum(axis=1)
    return df

def remove_invalid_records(df: pd.DataFrame) -> pd.DataFrame:
    """Remove records with null geography or date fields."""
    df = df.copy()
    required_cols = ['date', 'state', 'district', 'pincode']
    existing_required = [c for c in required_cols if c in df.columns]
    return df.dropna(subset=existing_required)
```

```
def preprocess_enrolment(df: pd.DataFrame) -> pd.DataFrame:
    """Full preprocessing pipeline for enrolment data."""
    df = parse_dates(df)
    df = remove_invalid_records(df)
    df = validate_pincode(df)
    df = normalize_state_names(df)
    df = add_temporal_features(df)
    df = add_enrolment_totals(df)
    return df.reset_index(drop=True)

def preprocess_demographic(df: pd.DataFrame) -> pd.DataFrame:
    """Full preprocessing pipeline for demographic update data."""
    df = parse_dates(df)
    df = remove_invalid_records(df)
    df = validate_pincode(df)
    df = normalize_state_names(df)
    df = add_temporal_features(df)
    df = add_update_totals(df, prefix='demo')
    return df.reset_index(drop=True)

def preprocess_biometric(df: pd.DataFrame) -> pd.DataFrame:
    """Full preprocessing pipeline for biometric update data."""
    df = parse_dates(df)
    df = remove_invalid_records(df)
    df = validate_pincode(df)
    df = normalize_state_names(df)
    df = add_temporal_features(df)
    df = add_update_totals(df, prefix='bio')
    return df.reset_index(drop=True)

def get_data_quality_report(df: pd.DataFrame, name: str = "Dataset") -> dict:
    """Generate data quality metrics for a dataset."""
    return {
        'name': name,
        'total_rows': len(df),
        'total_columns': len(df.columns),
        'missing_values': df.isnull().sum().to_dict(),
        'missing_pct': (df.isnull().sum() / len(df) * 100).to_dict(),
        'duplicates': df.duplicated().sum(),
        'memory_mb': df.memory_usage(deep=True).sum() / 1024 / 1024,
    }

def preprocess_all(enrolment: pd.DataFrame, demographic: pd.DataFrame,
                   biometric: pd.DataFrame) -> Tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame]:
```

```
    """Preprocess all three datasets."""
    return (
        preprocess_enrolment(enrolment),
        preprocess_demographic(demographic),
        preprocess_biometric(biometric)
    )
```