ACCSCIENCE PUBLISHING

Artificial Intelligence in Health

ORIGINAL RESEARCH ARTICLE

# Enhancing medical data quality using hybrid machine learning models: A comparative study of isolation forest and support vector machine on numerically encoded clinical text

## Supplementary File

### Algorithm S1. Coding of the isolation forest

```
import pandas as pd
import numpy as np
from scipy.stats import skew, kurtosis
import matplotlib.pyplot as plt
import time
from datetime import datetime

# Start timing
start_time = time.time()

# Load your dataset
file_path = "/kaggle/input/500-cities-local-data-for-better-health-2018/500_Cities__Local_Data_for_Better_Health__2018_release.csv"
data = pd.read_csv(file_path)

# Select only numerical columns
numerical_columns = data.select_dtypes(include=[np.number]).columns.tolist()

# Create a DataFrame to store the results
stats_results = pd.DataFrame(index=numerical_columns,
                             columns=['Skewness', 'Kurtosis', 'MAD', 'Variance',
                                      'Mean', 'Median', 'Std_Dev', 'Count', 'Min', 'Max'])

# Create a DataFrame for data quality metrics
quality_metrics = pd.DataFrame(index=numerical_columns,
                               columns=['Accuracy', 'Completeness', 'Consistency',
                                        'Timeliness', 'Validity', 'Uniqueness',
                                        'Precision', 'Change_Rate', 'Error_Rate',
                                        'Data_Drift'])

# Calculate metrics for each numerical column with enhanced validation
for col in numerical_columns:
    # Remove missing values
    clean_data = data[col].dropna().values

    if len(clean_data) > 1:
        # Basic statistics
        count = len(clean_data)
        mean_val = np.mean(clean_data)
        median_val = np.median(clean_data)
        min_val = np.min(clean_data)
        max_val = np.max(clean_data)
        std_dev_val = np.std(clean_data, ddof=1)
        variance_val = np.var(clean_data, ddof=1)

        # Calculate Skewness with Fisher-Pearson coefficient
        skewness_val = skew(clean_data, bias=False)

        # Calculate Kurtosis (Fisher's definition, normal => 0)
        kurtosis_val = kurtosis(clean_data, bias=False)

        # Calculate MAD (Median Absolute Deviation)
        mad_val = np.median(np.abs(clean_data - median_val))

        # Store results
        stats_results.loc[col, 'Count'] = count
        stats_results.loc[col, 'Mean'] = mean_val
        stats_results.loc[col, 'Median'] = median_val
        stats_results.loc[col, 'Min'] = min_val
        stats_results.loc[col, 'Max'] = max_val
        stats_results.loc[col, 'Std_Dev'] = std_dev_val
        stats_results.loc[col, 'Variance'] = variance_val
        stats_results.loc[col, 'Skewness'] = skewness_val
        stats_results.loc[col, 'Kurtosis'] = kurtosis_val
        stats_results.loc[col, 'MAD'] = mad_val

        # Calculate data quality metrics

        # 1. Accuracy (proportion of values within reasonable bounds)
        # For this example, we'll consider values within 3 standard deviations as "accurate"
        lower_bound = mean_val - 3 * std_dev_val
        upper_bound = mean_val + 3 * std_dev_val
        accurate_count = np.sum((clean_data >= lower_bound) & (clean_data <= upper_bound))
        quality_metrics.loc[col, 'Accuracy'] = accurate_count / count if count > 0 else 0

        # 2. Completeness (proportion of non-null values)
        total_rows = len(data)
        null_count = data[col].isnull().sum()
        quality_metrics.loc[col, 'Completeness'] = (total_rows - null_count) / total_rows

        # 3. Consistency (check if values follow expected patterns)
        # For numerical data, we can check if the median is between min and max
        quality_metrics.loc[col, 'Consistency'] = 1 if median_val >= min_val and median_val <= max_val else 0

        # 4. Timeliness (how recent the data is - this would normally require timestamp data)
        # Since we don't have timestamps, we'll set this to 1 (assuming data is current)
```

```
        quality_metrics.loc[col, 'Timeliness'] = 1

        # 5. Validity (proportion of values that conform to defined constraints)
        # For this example, we'll check if values are within a reasonable range for their type
        # This is dataset-specific and would need to be customized
        if col.lower().endswith('rate') or 'rate' in col.lower():
            valid_count = np.sum((clean_data >= 0) & (clean_data <= 1))
        elif col.lower().endswith('age') or 'age' in col.lower():
            valid_count = np.sum((clean_data >= 0) & (clean_data <= 120))
        else:
            # Default: values should be finite numbers
            valid_count = np.sum(np.isfinite(clean_data))
        quality_metrics.loc[col, 'Validity'] = valid_count / count if count > 0 else 0

        # 6. Uniqueness (proportion of unique values)
        unique_count = len(np.unique(clean_data))
        quality_metrics.loc[col, 'Uniqueness'] = unique_count / count if count > 0 else 0

        # 7. Precision (measure of decimal precision)
        # Calculate the average number of decimal places
        if np.issubdtype(clean_data.dtype, np.number):
            # Convert to string to count decimal places
            decimal_places = []
            for val in clean_data:
                if val % 1 != 0:  # Check if it has decimal part
                    s = str(val).split('.')
                    if len(s) > 1:
                        decimal_places.append(len(s[1]))
            if decimal_places:
                quality_metrics.loc[col, 'Precision'] = np.mean(decimal_places)
            else:
                quality_metrics.loc[col, 'Precision'] = 0
        else:
            quality_metrics.loc[col, 'Precision'] = 0

        # 8. Change Rate (how much the data changes over time - requires temporal data)
        # Since we don't have temporal data, we'll calculate the coefficient of variation
        quality_metrics.loc[col, 'Change_Rate'] = std_dev_val / mean_val if mean_val != 0 else 0

        # 9. Error Rate (proportion of values that are likely errors)
        # We'll use the Tukey method to identify outliers as potential errors
        Q1 = np.percentile(clean_data, 25)
        Q3 = np.percentile(clean_data, 75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        error_count = np.sum((clean_data < lower_bound) | (clean_data > upper_bound))
        quality_metrics.loc[col, 'Error_Rate'] = error_count / count if count > 0 else 0

        # 10. Data Drift (measure of how much the data distribution has changed)
        # Since we only have one dataset, we'll compare the current distribution to a normal distribution
        # using KL divergence (this is a simplification)
        from scipy.stats import norm, entropy
        # Fit a normal distribution to the data
        mu, std = norm.fit(clean_data)
        # Generate samples from the fitted distribution
        normal_samples = np.random.normal(mu, std, 1000)
        # Calculate histogram for both distributions
        hist_data, _ = np.histogram(clean_data, bins=50, density=True)
        hist_normal, _ = np.histogram(normal_samples, bins=50, density=True)
        # Calculate KL divergence
        kl_div = entropy(hist_data + 1e-10, hist_normal + 1e-10)  # Add small value to avoid log(0)
        quality_metrics.loc[col, 'Data_Drift'] = kl_div

        # Data validation checks
        if abs(skewness_val) < 0.01 and len(clean_data) > 100:
            print(f"Warning: {col} shows near-perfect symmetry (skewness = {skewness_val:.4f})")

        if abs(kurtosis_val + 1.2) < 0.01:
            print(f"Warning: {col} shows suspicious kurtosis value (-1.20)")

    else:
        print(f"Warning: Column '{col}' has insufficient data for statistical calculations")

# Format the output for better readability
def format_float(x):
    if pd.isna(x):
        return "N/A"
    if abs(x) < 0.0001:
        return f"{x:.6e}"
    elif abs(x) < 1:
        return f"{x:.6f}"
    elif abs(x) < 1000:
        return f"{x:.4f}"
    else:
        return f"{x:.4e}"

# Apply formatting to statistical results
formatted_results = stats_results.copy()
for col in formatted_results.columns:
    if col != 'Count':  # Keep count as integer
        formatted_results[col] = formatted_results[col].apply(format_float)

# Format quality metrics
formatted_quality = quality_metrics.copy()
for col in formatted_quality.columns:
    formatted_quality[col] = formatted_quality[col].apply(lambda x: f"{x:.4f}" if not pd.isna(x) else "N/A")

# Display the results
print("Revised Statistical Metrics for Numerical Columns:")
print("=" * 100)
print(formatted_results.to_string())

print("\n\nData Quality Metrics:")
print("=" * 100)
print(formatted_quality.to_string())

# Additional analysis for problematic variables
problem_vars = ['Cases_Per_100K', 'Deaths_Per_100K', 'Mortality_Rate']
print("\n\nDetailed Analysis of Problematic Variables:")
```

```
print("=" * 60)

for var in problem_vars:
    if var in numerical_columns:
        print(f"\n{var}:")
        var_data = data[var].dropna().values
        print(f"  - Unique values: {len(np.unique(var_data))}")
        print(f"  - Value range: {np.min(var_data):.6f} to {np.max(var_data):.6f}")
        print(f"  - 95% percentile: {np.percentile(var_data, 95):.6f}")
        print(f"  - 5% percentile: {np.percentile(var_data, 5):.6f}")

        # Check for constant or nearly constant values
        if np.std(var_data) < 0.001:
            print(f"  - WARNING: Standard deviation is extremely low ({np.std(var_data):.6f})")
            print(f"  - This suggests the variable may be constant or nearly constant")

# Create visualizations for the problematic variables
plt.figure(figsize=(15, 5))
for i, var in enumerate(problem_vars, 1):
    if var in numerical_columns:
        plt.subplot(1, 3, i)
        plt.hist(data[var].dropna(), bins=30, alpha=0.7, edgecolor='black')
        plt.title(f'Distribution of {var}')
        plt.xlabel('Value')
        plt.ylabel('Frequency')

plt.tight_layout()
plt.savefig('problematic_variables_distribution.png', dpi=300, bbox_inches='tight')
plt.close()

# Create visualizations for data quality metrics
plt.figure(figsize=(15, 10))
quality_metrics_to_plot = ['Accuracy', 'Completeness', 'Validity', 'Uniqueness', 'Error_Rate']
for i, metric in enumerate(quality_metrics_to_plot, 1):
    plt.subplot(2, 3, i)
    plt.barh(quality_metrics.index, quality_metrics[metric].astype(float))
    plt.title(f'{metric} by Column')
    plt.xlabel(metric)
    plt.tight_layout()

plt.savefig('data_quality_metrics.png', dpi=300, bbox_inches='tight')
plt.close()

# Save results to CSV
stats_results.to_csv('revised_statistical_metrics.csv')
quality_metrics.to_csv('data_quality_metrics.csv')
print("\nRevised results saved to 'revised_statistical_metrics.csv'")
print("Data quality metrics saved to 'data_quality_metrics.csv'")

# End timing and display results
end_time = time.time()
execution_time = end_time - start_time
print(f"\nTotal computation time: {execution_time:.2f} seconds")
Revised Statistical Metrics for Numerical Columns:
================================================
================================================
====
                Skewness   Kurtosis        MAD      Variance       Mean
Median     Std_Dev  Count       Min       Max
Year             -2.0370     2.1494  0.000000e+00    0.122709  2.0159e+03
2.0160e+03   0.350299  810103  2.0150e+03  2.0160e+03
Data_Value         0.794847  -0.720620      14.8000    666.4327
31.2249    22.8000    25.8154  787311    0.200000    95.5000
Low_Confidence_Limit  0.847926  -0.654945      13.8000    642.1943
29.4882    20.5000    25.3416  787311    0.100000    94.1000
High_Confidence_Limit 0.736333  -0.788547      15.9000    688.8400
32.9800    25.1000    26.2458  787311    0.200000    96.5000
PopulationCount       120.0308  1.4424e+04   1.3020e+03  6.5971e+12
3.2024e+04  3.6320e+03  2.5685e+06  810103     1.0000  3.0875e+08
CityFIPS           0.107284    -1.4604    1.6390e+06  2.8432e+12
2.6063e+06  2.6220e+06  1.6862e+06  810047  1.5003e+04  5.6139e+06
TractFIPS          0.106377    -1.4555    1.6014e+10  2.8069e+20
2.5929e+10  2.6081e+10  1.6754e+10  782047  1.0730e+09  5.6021e+10


Data Quality Metrics:
================================================
================================================
====
                Accuracy Completeness Consistency Timeliness Validity
Uniqueness Precision Change_Rate Error_Rate Data_Drift
Year               1.0000       1.0000       1.0000       1.0000  1.0000     0.0000
0.0000     0.0002    0.1432    5.9031
Data_Value         1.0000       0.9719       1.0000       1.0000  1.0000
0.0012     1.0000    0.8268    0.0000    2.7397
Low_Confidence_Limit  1.0000       0.9719       1.0000       1.0000  1.0000
0.0012     1.0000    0.8594    0.0000    1.6952
High_Confidence_Limit 1.0000       0.9719       1.0000       1.0000  1.0000
0.0012     1.0000    0.7958    0.0000    2.9788
PopulationCount       0.9999       1.0000       1.0000       1.0000  1.0000
0.0099     0.0000   80.2048    0.0460    5.5167
CityFIPS           1.0000       0.9999       1.0000       1.0000  1.0000     0.0006
0.0000     0.6470    0.0000    1.5615
TractFIPS          1.0000       0.9654       1.0000       1.0000  1.0000
0.0354     0.0000    0.6461    0.0000    0.0281


Detailed Analysis of Problematic Variables:
================================================
==========
/tmp/ipykernel_19/2276225393.py:234: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
/tmp/ipykernel_19/2276225393.py:234: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
/tmp/ipykernel_19/2276225393.py:234: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
/tmp/ipykernel_19/2276225393.py:234: UserWarning: The figure layout has changed to tight
  plt.tight_layout()
Revised results saved to 'revised_statistical_metrics.csv'
Data quality metrics saved to 'data_quality_metrics.csv'

Total computation time: 11.53 seconds
```

**Algorithm S2. Coding of the HIFSVM**

```
import os
import time
import pandas as pd
import numpy as np
from scipy.stats import skew, kurtosis, boxcox, yeojohnson, entropy,
norm
from sklearn.preprocessing import RobustScaler, PowerTransformer,
QuantileTransformer
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor
from sklearn.cluster import DBSCAN
from sklearn.svm import OneClassSVM
from sklearn.decomposition import PCA, IncrementalPCA
from sklearn.feature_selection import mutual_info_classif, f_classif
from sklearn.metrics import silhouette_score, calinski_harabasz_score
import matplotlib.pyplot as plt
import seaborn as sns
import gc
import re
from collections import Counter
import warnings
from typing import Dict, List, Tuple, Any, Optional
import json
from datetime import datetime
warnings.filterwarnings('ignore')

# Configure plotting style
plt.style.use('seaborn-v0_8')
sns.set_palette("husl")

class DataQualityAnalyzer:
    """Comprehensive data quality analysis class with enhanced
capabilities"""

    def __init__(self):
        self.medical_terms = {
            'health', 'medical', 'patient', 'clinical', 'diagnosis', 'treatment',
            'symptoms', 'disease', 'therapy', 'medicine', 'hospital', 'doctor',
            'nurse', 'pharmacy', 'prescription', 'vaccine', 'infection', 'virus',
            'bacteria', 'cancer', 'diabetes', 'heart', 'blood', 'pressure', 'cholesterol',
            'arthritis', 'obesity', 'stroke', 'asthma', 'allergy', 'depression', 'mental',
            'prevention', 'screening', 'mortality', 'morbidity', 'prevalence',
'incidence',
            'epidemiology', 'biopsy', 'chemotherapy', 'radiation', 'surgery',
'prognosis',
            'rehabilitation', 'palliative', 'genetic', 'chronic', 'acute', 'benign',
'malignant',
            'metastasis', 'remission', 'relapse', 'prophylaxis', 'immunization',
'antibiotic',
            'antiviral', 'antiinflammatory', 'analgesic', 'antidepressant',
'antipsychotic',
            'cardiology', 'neurology', 'oncology', 'pediatrics', 'geriatrics',
'psychiatry',
            'radiology', 'pathology', 'dermatology', 'endocrinology',
'gastroenterology',
            'nephrology', 'ophthalmology', 'orthopedics', 'otolaryngology',
'pulmonology',
            'urology', 'hematology', 'rheumatology', 'allergy', 'immunology',
'toxicology'
        }
        self.execution_times = {}
        self.analysis_results = {}
        self.training_metrics = {}
        self.total_training_time = 0.0
        self.training_time_breakdown = {}

    def calculate_data_quality_metrics(self, data: pd.DataFrame, feature_
names: List[str]) -> pd.DataFrame:
        """Calculate comprehensive data quality metrics for each feature"""
        print("\n=== COMPREHENSIVE DATA QUALITY METRICS
===")
        start_time = time.perf_counter()

        quality_metrics = []

        for feature in feature_names:
            col_data = data[feature].dropna()

            if len(col_data) == 0:
                continue

            # Basic statistics for quality calculations
            mean_val = np.mean(col_data)
            std_val = np.std(col_data)
            min_val = np.min(col_data)
            max_val = np.max(col_data)
            q1 = np.percentile(col_data, 25)
            q3 = np.percentile(col_data, 75)
            iqr = q3 - q1

            # 1. Accuracy - proportion within reasonable bounds (3 std
deviations)
            lower_bound_acc = mean_val - 3 * std_val
            upper_bound_acc = mean_val + 3 * std_val
            accurate_count = np.sum((col_data >= lower_bound_acc) &
(col_data <= upper_bound_acc))
            accuracy = accurate_count / len(col_data)

            # 2. Completeness - proportion of non-null values
            total_rows = len(data)
            null_count = data[feature].isnull().sum()
            completeness = (total_rows - null_count) / total_rows

            # 3. Consistency - check if data follows expected patterns
            median_val = np.median(col_data)
            consistency = 1 if median_val >= min_val and median_val <=
max_val else 0

            # 4. Timeliness - how recent the data is (assuming current for static
data)
            timeliness = 1.0  # Static dataset, so considered current

            # 5. Validity - proportion of values conforming to constraints
            if feature.lower().endswith('rate') or 'rate' in feature.lower():
                valid_count = np.sum((col_data >= 0) & (col_data <= 1))
            elif feature.lower().endswith('age') or 'age' in feature.lower():
                valid_count = np.sum((col_data >= 0) & (col_data <= 120))
            elif feature.lower().endswith('percentage') or 'pct' in feature.
lower():
                valid_count = np.sum((col_data >= 0) & (col_data <= 100))
            else:
```

```
            # Default: check for finite numbers and reasonable range
            valid_count = np.sum((np.isfinite(col_data)) &
                                 (col_data >= lower_bound_acc) &
                                 (col_data <= upper_bound_acc))
        validity = valid_count / len(col_data) if len(col_data) > 0 else 0

        # 6. Uniqueness - proportion of unique values
        unique_count = len(np.unique(col_data))
        uniqueness = unique_count / len(col_data) if len(col_data) > 0
else 0

        # 7. Precision - average decimal precision
        if np.issubdtype(col_data.dtype, np.number):
            decimal_places = []
            for val in col_data:
                if val % 1 != 0:  # Check if it has decimal part
                    s = str(val).split('.')
                    if len(s) > 1:
                        decimal_places.append(len(s[1]))
            precision = np.mean(decimal_places) if decimal_places else 0
        else:
            precision = 0

        # 8. Change Rate - coefficient of variation
        change_rate = std_val / mean_val if mean_val != 0 else 0

        # 9. Error Rate - proportion of outliers (IQR method)
        lower_bound_err = q1 - 1.5 * iqr
        upper_bound_err = q3 + 1.5 * iqr
        error_count = np.sum((col_data < lower_bound_err) | (col_data
> upper_bound_err))
        error_rate = error_count / len(col_data) if len(col_data) > 0 else 0

        # 10. Data Drift - KL divergence from normal distribution
        try:
            # Fit normal distribution
            mu, std = norm.fit(col_data)
            # Generate samples from fitted distribution
            normal_samples = np.random.normal(mu, std, min(1000,
len(col_data)))
            # Calculate histograms
            hist_data, _ = np.histogram(col_data, bins=50, density=True)
            hist_normal, _ = np.histogram(normal_samples, bins=50,
density=True)
            # Calculate KL divergence
            data_drift = entropy(hist_data + 1e-10, hist_normal + 1e-10)
        except:
            data_drift = 0

        metrics = {
            'Feature': feature,
            'Accuracy': accuracy,
            'Completeness': completeness,
            'Consistency': consistency,
            'Timeliness': timeliness,
            'Validity': validity,
            'Uniqueness': uniqueness,
            'Precision': precision,
            'Change_Rate': change_rate,
            'Error_Rate': error_rate,
            'Data_Drift': data_drift
        }

        quality_metrics.append(metrics)

    quality_df = pd.DataFrame(quality_metrics)
    self.execution_times['quality_metrics'] = time.perf_counter() -
start_time

    print("Data Quality Metrics Summary:")
    print("=" * 100)
    print(quality_df.round(4).to_string())

    return quality_df

  def load_and_explore_data(self, file_path: str, sample_size:
Optional[int] = None,
                          chunksize: Optional[int] = 10000) -> pd.DataFrame:
    """Enhanced data loading with memory optimization"""
    print("=== ENHANCED DATA LOADING AND EXPLORATION
===")

    if not os.path.isfile(file_path):
        raise FileNotFoundError(f"File not found: {file_path}")

    start_time = time.perf_counter()

    # Get file info
    file_size = os.path.getsize(file_path) / (1024 * 1024)  # MB
    print(f"File size: {file_size:.2f} MB")

    if sample_size:
        # Use chunks for memory efficiency even with sampling
        chunks = []
        for i, chunk in enumerate(pd.read_csv(file_path,
chunksize=chunksize)):
            if len(chunks) * chunksize >= sample_size:
                break
            chunks.append(chunk)
        data = pd.concat(chunks, ignore_index=True)
        data = data.head(sample_size)
        print(f"Loaded sample of {sample_size} rows using chunked
reading")
    else:
        # Optimize memory usage for full dataset
        data = pd.read_csv(file_path, low_memory=False)

    # Optimize data types
    data = self._optimize_data_types(data)

    self.execution_times['data_loading'] = time.perf_counter() - start_
time

    print(f"Dataset shape: {data.shape}")
    print(f"Memory usage: {data.memory_usage(deep=True).sum() /
(1024 * 1024):.2f} MB")
    print(f"\nData types:\n{data.dtypes.value_counts()}")

    # Enhanced missing values analysis
    self._analyze_missing_patterns(data)

    return data

  def _optimize_data_types(self, data: pd.DataFrame) -> pd.DataFrame:
    """Optimize data types to reduce memory usage"""
    for col in data.columns:
```

```
        if data[col].dtype == 'object':
            # Convert to category if low cardinality
            if data[col].nunique() / len(data) < 0.5:
                data[col] = data[col].astype('category')
        elif data[col].dtype in ['int64', 'float64']:
            # Downcast numerical types
            if data[col].dtype == 'int64':
                data[col] = pd.to_numeric(data[col], downcast='integer')
            else:
                data[col] = pd.to_numeric(data[col], downcast='float')
    return data

  def _analyze_missing_patterns(self, data: pd.DataFrame):
    """Analyze patterns in missing data"""
    missing_matrix = data.isnull()
    missing_corr = missing_matrix.corr()

    print("\nMissing Values Pattern Analysis:")
    print(f"Total missing values: {missing_matrix.sum().sum()}")
    print(f"Percentage missing: {(missing_matrix.sum().sum() / (data.
shape[0] * data.shape[1])) * 100:.2f}%")

    # Find columns with high missingness
    high_missing = missing_matrix.sum() / len(data) > 0.3
    if high_missing.any():
        print(f"\nColumns with >30% missing values: {list(high_
missing[high_missing].index)}")

  def clean_data(self, data: pd.DataFrame) -> Tuple[pd.DataFrame,
tuple]:
    """Enhanced data cleaning with more sophisticated methods"""
    print("\n=== ENHANCED DATA CLEANING ===")
    start_time = time.perf_counter()

    initial_shape = data.shape
    original_memory = data.memory_usage(deep=True).sum()

    # Remove duplicates
    data = data.drop_duplicates()
    duplicates_removed = initial_shape[0] - data.shape[0]

    # Handle inconsistent whitespace in text columns
    text_cols = data.select_dtypes(include=['object', 'category']).
columns
    for col in text_cols:
        data[col] = data[col].astype(str).str.strip()

    # Remove empty strings
    for col in text_cols:
        data[col] = data[col].replace({'': np.nan, 'nan': np.nan, 'None':
np.nan})

    # Clean numerical outliers (temporary removal for analysis)
    numerical_cols = data.select_dtypes(include=[np.number]).
columns
    data_cleaned = data.copy()

    self.execution_times['data_cleaning'] = time.perf_counter() -
start_time

    final_memory = data_cleaned.memory_usage(deep=True).sum()
    memory_saved = original_memory - final_memory

    print(f"Removed {duplicates_removed} duplicate rows")
    print(f"Memory saved: {memory_saved / (1024 * 1024):.2f} MB")
    print(f"Final shape: {data_cleaned.shape}")

    return data_cleaned, initial_shape

  def comprehensive_missing_analysis(self, data: pd.DataFrame) ->
Dict:
    """Comprehensive missing values analysis"""
    print("\n=== COMPREHENSIVE MISSING VALUES ANALYSIS
===")
    start_time = time.perf_counter()

    missing_analysis = {}
    total_missing = data.isnull().sum().sum()
    missing_analysis['total_missing'] = total_missing
    missing_analysis['missing_percentage'] = (total_missing / (data.
shape[0] * data.shape[1])) * 100

    # Per column analysis
    missing_by_column = data.isnull().sum()
    missing_analysis['columns'] = {}

    for col in data.columns:
        missing_count = missing_by_column[col]
        if missing_count > 0:
            missing_pct = (missing_count / len(data)) * 100
            missing_analysis['columns'][col] = {
                'count': missing_count,
                'percentage': missing_pct,
                'type': str(data[col].dtype)
            }

    # Pattern analysis
    missing_matrix = data.isnull()
    missing_corr = missing_matrix.corr()
    missing_analysis['correlation_matrix'] = missing_corr

    self.execution_times['missing_analysis'] = time.perf_counter() -
start_time

    print(f"Total missing values: {total_missing}")
    print(f"Overall missing percentage: {missing_analysis['missing_
percentage']:.2f}%")

    return missing_analysis

  def enhanced_text_analysis(self, data: pd.DataFrame, max_text_
columns: int = 10) -> Dict:
    """Enhanced text analysis with NLP techniques"""
    print("\n=== ENHANCED TEXT QUALITY ANALYSIS ===")
    start_time = time.perf_counter()

    text_metrics = {}
    text_cols = [col for col in data.columns if data[col].dtype in ['object',
'category']]

    if len(text_cols) > max_text_columns:
        print(f"Limiting to {max_text_columns} of {len(text_cols)} text
columns")
        text_cols = text_cols[:max_text_columns]

    for col in text_cols:
```

```
        print(f"\nAnalyzing: {col}")
        col_data = data[col].dropna()

        if len(col_data) == 0:
            print(f"  No data available for analysis")
            continue

        # Basic metrics
        metrics = {
            'unique_count': col_data.nunique(),
            'empty_strings': (col_data == '').sum(),
            'whitespace_only': col_data.apply(lambda x: isinstance(x, str)
and x.strip() == '').sum(),
            'avg_length': col_data.apply(lambda x: len(str(x))).mean(),
            'numeric_contamination': col_data.apply(lambda x: bool(re.
search(r'\d', str(x)))).mean() * 100,
        }

        # Advanced text metrics
        sample_text = ' '.join(col_data.astype(str).str.lower().
sample(min(5000, len(col_data)), random_state=42))
        words = re.findall(r'\b[a-z]{3,}\b', sample_text)

        if words:
            word_counts = Counter(words)
            metrics.update({
                'vocabulary_size': len(word_counts),
                'avg_word_length': np.mean([len(word) for word in words]),
                'medical_terms_found': sum(1 for word in words if word in
self.medical_terms),
                'medical_coverage': (sum(1 for word in words if word in self.
medical_terms) / len(words)) * 100
            })

        text_metrics[col] = metrics

        # Print summary
        print(f"  Unique values: {metrics['unique_count']}")
        print(f"  Avg length: {metrics['avg_length']:.1f} chars")
        print(f"  Numeric contamination: {metrics['numeric_
contamination']:.1f}%")
        if 'vocabulary_size' in metrics:
            print(f"  Vocabulary size: {metrics['vocabulary_size']}")
            print(f"  Medical coverage: {metrics['medical_
coverage']:.1f}%")

    self.execution_times['text_analysis'] = time.perf_counter() - start_
time
    return text_metrics

  def advanced_feature_analysis(self, data: pd.DataFrame) -> Tuple[np.
ndarray, List[str], Dict]:
    """Advanced feature analysis with multiple preprocessing strategies"""
    print("\n=== ADVANCED FEATURE ANALYSIS ===")
    start_time = time.perf_counter()

    numerical_cols = data.select_dtypes(include=[np.number]).
columns.tolist()
    print(f"Found {len(numerical_cols)} numerical features")

    # Feature selection based on variance and correlation
    selected_features = self._select_features(data[numerical_cols])
    print(f"Selected {len(selected_features)} features after filtering")

    X = data[selected_features]

    # Multiple imputation strategies
    X_imputed = self._advanced_imputation(X)

    # Handle extreme values with multiple methods
    X_processed = self._handle_extreme_values(X_imputed)

    # Feature transformation analysis
    transformation_analysis = self._analyze_feature_
transformations(X_processed, selected_features)

    self.execution_times['feature_analysis'] = time.perf_counter() -
start_time

    return X_processed, selected_features, transformation_analysis

  def _select_features(self, X: pd.DataFrame, max_features: int = 30) ->
List[str]:
    """Select most informative features"""
    # Remove constant features
    variances = X.var()
    non_constant = variances[variances > 1e-10].index.tolist()

    if len(non_constant) > max_features:
        # Select top features by variance and mutual information
        high_variance = variances.nlargest(max_features // 2).index.
tolist()

        # For the other half, use correlation analysis
        corr_matrix = X[non_constant].corr().abs()
        upper_tri = corr_matrix.where(np.triu(np.ones_like(corr_matrix,
dtype=bool), k=1))
        low_corr_features = [col for col in non_constant if all(upper_
tri[col] < 0.8)]

        selected = list(set(high_variance + low_corr_features))[:max_
features]
        return selected

    return non_constant

  def _advanced_imputation(self, X: pd.DataFrame) -> np.ndarray:
    """Advanced imputation with multiple strategies"""
    missing_mask = X.isnull()

    if missing_mask.sum().sum() == 0:
        return X.values

    # Use different imputation strategies based on data characteristics
    imputed_data = X.copy()

    for col in X.columns:
        if missing_mask[col].sum() > 0:
            col_data = X[col]

            if col_data.skew() > 2:  # Highly skewed -> use median
                imputer = SimpleImputer(strategy='median')
            else:
                imputer = SimpleImputer(strategy='mean')

            imputed_data[col] = imputer.fit_transform(col_data.values.
```

```
reshape(-1, 1)).flatten()

        return imputed_data.values

    def _handle_extreme_values(self, X: np.ndarray) -> np.ndarray:
        """Handle extreme values with robust methods"""
        X_processed = X.copy()

        for i in range(X.shape[1]):
            col_data = X[:, i]
            q1, q3 = np.percentile(col_data, [25, 75])
            iqr = q3 - q1
            lower_bound = q1 - 3 * iqr  # More conservative bounds
            upper_bound = q3 + 3 * iqr

            # Winsorize extreme values
            X_processed[:, i] = np.clip(col_data, lower_bound, upper_bound)

        return X_processed

    def _analyze_feature_transformations(self, X: np.ndarray, feature_
names: List[str]) -> Dict:
        """Analyze optimal transformations for each feature"""
        transformations = {}

        for i, feature in enumerate(feature_names):
            col_data = X[:, i]

            # Skip if constant or mostly constant
            if np.std(col_data) < 1e-10:
                continue

            original_skew = skew(col_data)
            transformations[feature] = {
                'original_skew': original_skew,
                'suggested_transformation': self._suggest_transformation(col_
data)
            }

        return transformations

    def _suggest_transformation(self, data: np.ndarray) -> str:
        """Suggest optimal transformation based on data characteristics"""
        skewness = skew(data)

        if abs(skewness) < 0.5:
            return "none"
        elif skewness > 1:
            return "log"
        elif skewness < -1:
            return "yeo-johnson"
        else:
            return "box-cox"

    def comprehensive_statistical_analysis(self, X: np.ndarray, feature_
names: List[str]) -> pd.DataFrame:
        """Comprehensive statistical analysis with enhanced metrics"""
        print("\n=== COMPREHENSIVE STATISTICAL ANALYSIS
===")
        start_time = time.perf_counter()

        stats_data = []

        for i, feature in enumerate(feature_names):
            col_data = X[:, i]

            # Basic statistics - removed Mean, Median, Std_Dev, Min, Max,
Range, IQR, Skewness, Kurtosis, CV, MAD
            stats = {
                'Feature': feature,
                'Q1': np.percentile(col_data, 25),
                'Q3': np.percentile(col_data, 75),
                'Non_Zero%': (np.sum(col_data != 0) / len(col_data)) * 100,
                'Zeros': np.sum(col_data == 0)
            }

            # Outlier analysis
            q1, q3 = stats['Q1'], stats['Q3']
            iqr = q3 - q1
            lower_bound = q1 - 1.5 * iqr
            upper_bound = q3 + 1.5 * iqr
            outliers = np.sum((col_data < lower_bound) | (col_data >
upper_bound))
            stats['Outliers'] = outliers
            stats['Outlier%'] = (outliers / len(col_data)) * 100

            stats_data.append(stats)

        stats_df = pd.DataFrame(stats_data)
        self.execution_times['statistical_analysis'] = time.perf_counter() -
start_time

        print("Statistical Summary:")
        print(stats_df.round(4).to_string())

        return stats_df

    def enhanced_anomaly_detection(self, X: np.ndarray, contamination:
float = 0.05) -> Dict:
        """Enhanced anomaly detection with detailed training time
tracking"""
        print("\n=== ENHANCED ANOMALY DETECTION ===")
        start_time = time.perf_counter()

        results = {}
        training_times = {}
        outlier_scores = {}
        ml_components = []

        print("🔄 Starting ML algorithm training...")

        # 1. Isolation Forest with timing
        print("  Training Isolation Forest...")
        if_start = time.perf_counter()
        if_model = IsolationForest(contamination=contamination,
random_state=42, n_jobs=-1)
        if_scores = if_model.fit_predict(X)
        if_time = time.perf_counter() - if_start
        training_times['Isolation_Forest'] = if_time
        results['Isolation_Forest'] = np.sum(if_scores == -1)
        outlier_scores['if'] = if_model.decision_function(X)
        ml_components.append('Isolation_Forest')
        print(f"    ✓ Isolation Forest trained in {if_time:.4f}s")

        # 2. Local Outlier Factor with timing
        print("  Training Local Outlier Factor...")
```

```
        lof_start = time.perf_counter()
        try:
            lof_model = LocalOutlierFactor(contamination=contamination,
n_jobs=-1)
            lof_scores = lof_model.fit_predict(X)
            lof_time = time.perf_counter() - lof_start
            training_times['LOF'] = lof_time
            results['LOF'] = np.sum(lof_scores == -1)
            outlier_scores['lof'] = lof_model.negative_outlier_factor_
            ml_components.append('LOF')
            print(f"   ✓ LOF trained in {lof_time:.4f}s")
        except Exception as e:
            print(f"   ✗ LOF failed: {e}")

        # 3. One-Class SVM with timing (on sample for large datasets)
        if X.shape[0] < 10000:
            print("  Training One-Class SVM...")
            try:
                ocsvm_start = time.perf_counter()
                ocsvm_model = OneClassSVM(nu=contamination)
                ocsvm_scores = ocsvm_model.fit_predict(X)
                ocsvm_time = time.perf_counter() - ocsvm_start
                training_times['OneClass_SVM'] = ocsvm_time
                results['OneClass_SVM'] = np.sum(ocsvm_scores == -1)
                ml_components.append('OneClass_SVM')
                print(f"   ✓ One-Class SVM trained in {ocsvm_time:.4f}s")
            except Exception as e:
                print(f"   ✗ OneClass SVM failed: {e}")
        else:
            print("  ℹ Skipping One-Class SVM (dataset too large)")

        # Ensemble approach (majority voting)
        ensemble_scores = np.zeros(X.shape[0])
        for method, scores in outlier_scores.items():
            ensemble_scores += scores

        # Normalize and threshold
        ensemble_scores = (ensemble_scores - np.mean(ensemble_scores))
/ np.std(ensemble_scores)
        ensemble_outliers = ensemble_scores < np.percentile(ensemble_
scores, contamination * 100)
        results['Ensemble'] = np.sum(ensemble_outliers)

        # Calculate total training time
        total_training_time = sum(training_times.values())
        self.total_training_time = total_training_time
        self.training_time_breakdown = training_times

        # Store training metrics
        self.execution_times['ml_training_breakdown'] = training_times
        self.execution_times['total_training_time'] = total_training_time
        self.execution_times['anomaly_detection'] = time.perf_counter() -
start_time

        # Calculate training metrics
        self.training_metrics = self._analyze_training_metrics(training_
times)

        # Print comprehensive training time summary
        self._display_training_time_summary(training_times, total_
training_time)

        return {
            'results': results,
            'scores': outlier_scores,
            'ensemble_outliers': ensemble_outliers,
            'training_times': training_times,
            'total_training_time': total_training_time,
            'ml_components_used': ml_components
        }

    def _display_training_time_summary(self, training_times: Dict,
total_training_time: float):
        """Display comprehensive training time summary"""
        print("\n" + "="*60)
        print("🤖 MACHINE LEARNING TRAINING TIME
COMPUTATION")
        print("="*60)

        # Display individual algorithm times
        print(f"\n{'ALGORITHM':<25} {'TRAINING TIME':<15}
{'PERCENTAGE':<12}")
        print("-" * 55)

        for model, train_time in sorted(training_times.items(), key=lambda
x: x[1], reverse=True):
            percentage = (train_time / total_training_time) * 100
            print(f"{model:<25} {train_time:.4f}s{' ':>8}
{percentage:>6.1f}%")

        # Display summary statistics
        times_list = list(training_times.values())
        print("\n📊 TRAINING TIME STATISTICS:")
        print(f"  Total Training Time:    {total_training_time:.4f} seconds")
        print(f"  Number of Algorithms:   {len(training_times)}")
        print(f"  Average Time per Model: {np.mean(times_list):.4f}
seconds")
        print(f"  Standard Deviation:     {np.std(times_list):.4f} seconds")
        print(f"  Fastest Algorithm:      {min(training_times, key=training_
times.get)} ({min(times_list):.4f}s)")
        print(f"  Slowest Algorithm:      {max(training_times,
key=training_times.get)} ({max(times_list):.4f}s)")

        # Efficiency metrics
        efficiency_score = self._calculate_training_efficiency(training_
times)
        print(f"  Training Efficiency:    {efficiency_score:.1f}/100")

        print("="*60)

    def _analyze_training_metrics(self, training_times: Dict) -> Dict:
        """Analyze and compute training time metrics"""
        if not training_times:
            return {}

        times_list = list(training_times.values())

        metrics = {
            'total_training_time': sum(times_list),
            'average_training_time': np.mean(times_list),
            'std_training_time': np.std(times_list),
            'min_training_time': min(times_list),
            'max_training_time': max(times_list),
            'fastest_algorithm': min(training_times, key=training_times.get),
            'slowest_algorithm': max(training_times, key=training_times.
get),
```

```
        'training_time_breakdown': training_times,
        'training_efficiency_score': self._calculate_training_
efficiency(training_times),
        'algorithms_trained': len(training_times)
    }

    return metrics

  def _calculate_training_efficiency(self, training_times: Dict) -> float:
    """Calculate training efficiency score (lower times are better)"""
    if not training_times:
      return 0.0

    # Normalize and score efficiency (inverse relationship with time)
    max_time = max(training_times.values())
    efficiency_scores = [1 - (time / (max_time + 1e-10)) for time in
training_times.values()]
    return np.mean(efficiency_scores) * 100

  def generate_advanced_visualizations(self, X: np.ndarray, outliers:
np.ndarray,
                       stats_df: pd.DataFrame, feature_names: List[str],
                       quality_df: pd.DataFrame):
    """Generate comprehensive visualizations including quality metrics"""
    print("\n=== GENERATING ADVANCED VISUALIZATIONS
===")
    start_time = time.perf_counter()

    # Create subplots grid
    fig, axes = plt.subplots(3, 2, figsize=(15, 18))
    fig.suptitle('Comprehensive Data Quality Analysis', fontsize=16,
fontweight='bold')

    # 1. Outlier visualization (PCA)
    pca = PCA(n_components=2)
    X_pca = pca.fit_transform(X)

    axes[0, 0].scatter(X_pca[~outliers, 0], X_pca[~outliers, 1],
               alpha=0.6, s=20, label='Normal', color='blue')
    axes[0, 0].scatter(X_pca[outliers, 0], X_pca[outliers, 1],
               alpha=0.8, s=30, label='Outliers', color='red')
    axes[0, 0].set_title('Anomaly Detection (PCA Projection)')
    axes[0, 0].set_xlabel('Principal Component 1')
    axes[0, 0].set_ylabel('Principal Component 2')
    axes[0, 0].legend()
    axes[0, 0].grid(True, alpha=0.3)

    # 2. Data Quality Metrics Heatmap
    quality_metrics_numeric = quality_df.set_index('Feature').select_
dtypes(include=[np.number])
    sns.heatmap(quality_metrics_numeric.T, annot=True, fmt='.3f',
cmap='RdYlGn',
          center=0.5, ax=axes[0, 1])
    axes[0, 1].set_title('Data Quality Metrics Heatmap')

    # 3. Correlation heatmap (top features)
    corr_matrix = pd.DataFrame(X, columns=feature_names).corr()
    mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
    sns.heatmap(corr_matrix, mask=mask, cmap='coolwarm', center=0,
          square=True, ax=axes[1, 0], cbar_kws={"shrink": .8})
    axes[1, 0].set_title('Feature Correlation Heatmap')

    # 4. Quality Metrics Distribution
    quality_metrics_to_plot = ['Accuracy', 'Completeness', 'Validity',
'Error_Rate']
    quality_melted = quality_df.melt(id_vars=['Feature'], value_
vars=quality_metrics_to_plot,
                    var_name='Metric', value_name='Value')
    sns.boxplot(data=quality_melted, x='Metric', y='Value', ax=axes[1,
1])
    axes[1, 1].set_title('Distribution of Quality Metrics')
    axes[1, 1].tick_params(axis='x', rotation=45)

    # 5. Data distribution examples
    sample_features = feature_names[:min(4, len(feature_names))]
    for i, feat in enumerate(sample_features):
      row, col = 2, i
      if i < 2:
        col_idx = feature_names.index(feat)
        axes[2, i].hist(X[:, col_idx], bins=30, alpha=0.7, color='green',
edgecolor='black')
        axes[2, i].set_title(f'Distribution: {feat}')
        axes[2, i].set_xlabel('Value')
        axes[2, i].set_ylabel('Frequency')
        axes[2, i].grid(True, alpha=0.3)

    plt.tight_layout()
    plt.savefig('comprehensive_analysis.png', dpi=300, bbox_
inches='tight')
    plt.show()

    # Additional specialized plots
    self._create_specialized_plots(X, outliers, stats_df, feature_names,
quality_df)

    self.execution_times['visualization'] = time.perf_counter() - start_
time

  def _create_specialized_plots(self, X: np.ndarray, outliers: np.ndarray,
                  stats_df: pd.DataFrame, feature_names: List[str],
                  quality_df: pd.DataFrame):
    """Create specialized analysis plots including quality metrics"""
    # Quality metrics radar chart for top features
    fig, axes = plt.subplots(2, 2, figsize=(15, 12))

    # Select top 4 features for detailed quality analysis
    top_features = quality_df.nlargest(4, 'Completeness')['Feature'].
tolist()

    for i, feat in enumerate(top_features[:4]):
      row, col = i // 2, i % 2
      feat_quality = quality_df[quality_df['Feature'] == feat].iloc[0]

      # Radar chart data
      categories = ['Accuracy', 'Completeness', 'Validity', 'Uniqueness',
'Precision']
      values = [feat_quality['Accuracy'], feat_quality['Completeness'],
          feat_quality['Validity'], feat_quality['Uniqueness'],
          min(feat_quality['Precision'] / 10, 1.0)]  # Normalize
precision

      # Complete the circle
      values += values[:1]
      angles = np.linspace(0, 2*np.pi, len(categories), endpoint=False).
tolist()
      angles += angles[:1]
```

```
        # Plot radar chart
        ax = axes[row, col]
        ax.plot(angles, values, 'o-', linewidth=2, label=feat)
        ax.fill(angles, values, alpha=0.25)
        ax.set_thetagrids(np.degrees(angles[:-1]), categories)
        ax.set_ylim(0, 1)
        ax.set_title(f'Quality Metrics: {feat}')
        ax.grid(True)

    plt.tight_layout()
    plt.savefig('quality_metrics_radar.png', dpi=300, bbox_
inches='tight')
    plt.show()

  def generate_comprehensive_report(self, data: pd.DataFrame,
analysis_results: Dict):
    """Generate comprehensive HTML report with training time
analytics"""
    print("\n=== GENERATING COMPREHENSIVE REPORT ===")
    start_time = time.perf_counter()

    report = {
      'timestamp': datetime.now().isoformat(),
      'dataset_info': {
        'shape': data.shape,
        'memory_usage_mb': data.memory_usage(deep=True).sum() /
(1024 * 1024),
        'columns': list(data.columns),
        'data_types': {col: str(dtype) for col, dtype in data.dtypes.
items()}
      },
      'execution_times': self.execution_times,
      'training_metrics': self.training_metrics,
      'total_training_time': self.total_training_time,
      'training_time_breakdown': self.training_time_breakdown,
      'analysis_results': analysis_results,
      'summary': self._generate_summary(analysis_results)
    }

    # Save JSON report
    with open('data_quality_report.json', 'w') as f:
      json.dump(report, f, indent=2, default=str)

    # Generate enhanced HTML report
    self._generate_enhanced_html_report(report)

    self.execution_times['report_generation'] = time.perf_counter() -
start_time
    print("Comprehensive report generated: data_quality_report.json")

  def _generate_summary(self, analysis_results: Dict) -> Dict:
    """Generate executive summary"""
    summary = {
      'data_quality_score': self._calculate_quality_score(analysis_
results),
      'critical_issues': [],
      'recommendations': [],
      'key_metrics': {}
    }

    # Add critical issues and recommendations based on analysis
    if 'missing_analysis' in analysis_results:
      missing_pct = analysis_results['missing_analysis']['missing_
percentage']
      if missing_pct > 20:
        summary['critical_issues'].append(f"High missing data:
{missing_pct:.1f}%")
        summary['recommendations'].append("Implement robust data
collection procedures")

    if 'statistical_analysis' in analysis_results:
      stats_df = pd.DataFrame(analysis_results['statistical_analysis'])
      high_outliers = stats_df[stats_df['Outlier%'] > 10]
      if len(high_outliers) > 0:
        summary['critical_issues'].append(f"{len(high_outliers)}
features with high outlier percentage (>10%)")
        summary['recommendations'].append("Review outlier
detection and handling strategy")

    # Add quality metrics summary
    if 'quality_metrics' in analysis_results:
      quality_df = pd.DataFrame(analysis_results['quality_metrics'])
      low_completeness = quality_df[quality_df['Completeness'] < 0.8]
      if len(low_completeness) > 0:
        summary['critical_issues'].append(f"{len(low_completeness)}
features with low completeness (<80%)")

      high_error_rate = quality_df[quality_df['Error_Rate'] > 0.1]
      if len(high_error_rate) > 0:
        summary['critical_issues'].append(f"{len(high_error_rate)}
features with high error rate (>10%)")

    # Add training metrics to summary
    if self.training_metrics:
      summary['training_performance'] = {
        'total_training_time': self.training_metrics['total_training_
time'],
        'efficiency_score': self.training_metrics['training_efficiency_
score'],
        'fastest_algorithm': self.training_metrics['fastest_algorithm'],
        'algorithms_trained': self.training_metrics['algorithms_
trained']
      }

    return summary

  def _calculate_quality_score(self, analysis_results: Dict) -> float:
    """Calculate overall data quality score"""
    score = 100.0

    # Penalize for missing data
    if 'missing_analysis' in analysis_results:
      missing_pct = analysis_results['missing_analysis']['missing_
percentage']
      score -= min(30, missing_pct * 0.5)

    # Penalize for high outlier percentage
    if 'statistical_analysis' in analysis_results:
      stats_df = pd.DataFrame(analysis_results['statistical_analysis'])
      avg_outlier_pct = stats_df['Outlier%'].mean()
      score -= min(20, avg_outlier_pct * 0.3)

    # Consider quality metrics
    if 'quality_metrics' in analysis_results:
      quality_df = pd.DataFrame(analysis_results['quality_metrics'])
```

```
        avg_completeness = quality_df['Completeness'].mean()
        avg_validity = quality_df['Validity'].mean()
        avg_accuracy = quality_df['Accuracy'].mean()

        quality_score = (avg_completeness + avg_validity + avg_
accuracy) / 3 * 100
        score = min(score, quality_score)  # Use the more conservative
score

    return max(0, score)

  def _generate_enhanced_html_report(self, report: Dict):
    """Generate enhanced HTML version of the report with training
metrics"""
    # Training metrics section
    training_section = ""
    if report.get('training_metrics'):
      tm = report['training_metrics']
      training_section = f"""
      <div class="section">
        <h2>🤖 Machine Learning Training Performance</h2>
        <div class="metric training-metrics">
          <h3>Training Time Analytics</h3>
          <p><strong>Total Training Time:</strong> {tm.get('total_
training_time', 0):.4f} seconds</p>
          <p><strong>Algorithms Trained:</strong> {tm.
get('algorithms_trained', 0)}</p>
          <p><strong>Average per Algorithm:</strong> {tm.
get('average_training_time', 0):.4f} seconds</p>
          <p><strong>Training Efficiency Score:</strong> {tm.
get('training_efficiency_score', 0):.1f}/100</p>
          <p><strong>Fastest Algorithm:</strong> {tm.get('fastest_
algorithm', 'N/A')}</p>
          <p><strong>Slowest Algorithm:</strong> {tm.get('slowest_
algorithm', 'N/A')}</p>

          <h4>Detailed Breakdown:</h4>
          <ul>
            {"".join([f"<li><strong>{model}:</strong> {time:.4f}
seconds</li>"
                for model, time in tm.get('training_time_breakdown',
{}).items()])}
          </ul>
        </div>
      </div>
      """

    # Quality metrics section
    quality_section = ""
    if 'quality_metrics' in report.get('analysis_results', {}):
      quality_df = pd.DataFrame(report['analysis_results']['quality_
metrics'])
      quality_section = f"""
      <div class="section">
        <h2>📊 Data Quality Metrics Summary</h2>
        <div class="metric quality-metrics">
          <h3>Average Quality Scores</h3>
          <p><strong>Accuracy:</strong> {quality_df['Accuracy'].
mean():.3f}</p>
          <p><strong>Completeness:</strong> {quality_
df['Completeness'].mean():.3f}</p>
          <p><strong>Validity:</strong> {quality_df['Validity'].
mean():.3f}</p>
          <p><strong>Uniqueness:</strong> {quality_
df['Uniqueness'].mean():.3f}</p>
          <p><strong>Error Rate:</strong> {quality_df['Error_Rate'].
mean():.3f}</p>
        </div>
      </div>
      """

    html_content = f"""
    <!DOCTYPE html>
    <html>
    <head>
      <title>Data Quality Analysis Report</title>
      <style>
        body {{ font-family: Arial, sans-serif; margin: 40px; }}
        .header {{ background: #f4f4f4; padding: 20px; border-radius:
5px; }}
        .section {{ margin: 20px 0; }}
        .metric {{ background: #e8f4f8; padding: 10px; margin: 5px 0;
border-radius: 3px; }}
        .critical {{ background: #ffe6e6; }}
        .training-metrics {{ background: #f0f8f0; }}
        .quality-metrics {{ background: #fff0f5; }}
        .time-breakdown {{ background: #fff8e6; }}
      </style>
    </head>
    <body>
      <div class="header">
        <h1>Data Quality Analysis Report</h1>
        <p>Generated: {report['timestamp']}</p>
        <p>Dataset: {report['dataset_info']['shape'][0]} rows ×
{report['dataset_info']['shape'][1]} columns</p>
      </div>

      <div class="section">
        <h2>Executive Summary</h2>
        <div class="metric">
          <h3>Overall Data Quality Score: {report['summary']['data_
quality_score']:.1f}/100</h3>
        </div>
      </div>

      {quality_section}

      {training_section}

      <div class="section">
        <h2>Execution Times</h2>
        <div class="metric time-breakdown">
          <ul>
            {"".join([f"<li><strong>{step.replace('_', ' ').title()}:</
strong> {time:.4f} seconds</li>"
                for step, time in report['execution_times'].items() if
step != 'ml_training_breakdown'])}
          </ul>
        </div>
      </div>
    </body>
    </html>
    """

    with open('data_quality_report.html', 'w') as f:
      f.write(html_content)
```

```
# Enhanced main execution function with quality metrics
def main():
    """Enhanced main execution function with quality metrics"""
    analyzer = DataQualityAnalyzer()

    try:
        # File path to the dataset
        file_path = "/kaggle/input/500-cities-local-data-for-better-health-2018/500_Cities__Local_Data_for_Better_Health__2018_release.csv"

        # Load data with enhanced methods
        data = analyzer.load_and_explore_data(file_path, sample_size=15000)

        # Clean data
        data_cleaned, initial_shape = analyzer.clean_data(data)

        # Comprehensive missing analysis
        missing_analysis = analyzer.comprehensive_missing_analysis(data_cleaned)

        # Enhanced text analysis
        text_analysis = analyzer.enhanced_text_analysis(data_cleaned, max_text_columns=5)

        # Advanced feature analysis
        X_processed, feature_names, transformation_analysis = analyzer.advanced_feature_analysis(data_cleaned)

        # Comprehensive statistical analysis
        stats_df = analyzer.comprehensive_statistical_analysis(X_processed, feature_names)

        # Calculate data quality metrics
        quality_df = analyzer.calculate_data_quality_metrics(data_cleaned, feature_names)

        # Enhanced anomaly detection with training time tracking
        anomaly_results = analyzer.enhanced_anomaly_detection(X_processed, contamination=0.05)

        # Generate visualizations
        analyzer.generate_advanced_visualizations(
            X_processed,
            anomaly_results['ensemble_outliers'],
            stats_df,
            feature_names,
            quality_df
        )

        # Compile all results
        analysis_results = {
            'missing_analysis': missing_analysis,
            'text_analysis': text_analysis,
            'transformation_analysis': transformation_analysis,
            'statistical_analysis': stats_df.to_dict('records'),
            'quality_metrics': quality_df.to_dict('records'),
            'anomaly_detection': anomaly_results['results'],
            'training_times': anomaly_results['training_times'],
            'total_training_time': anomaly_results['total_training_time']
        }

        # Generate comprehensive report
        analyzer.generate_comprehensive_report(data_cleaned, analysis_results)

        # Enhanced execution summary with quality metrics
        print("\n" + "="*80)
        print("🎯 FINAL EXECUTION SUMMARY WITH QUALITY METRICS")
        print("="*80)

        # Display quality metrics summary
        print(f"\n📊 **DATA QUALITY METRICS SUMMARY**:")
        print(f"  Average Accuracy:      {quality_df['Accuracy'].mean():.3f}")
        print(f"  Average Completeness:  {quality_df['Completeness'].mean():.3f}")
        print(f"  Average Validity:      {quality_df['Validity'].mean():.3f}")
        print(f"  Average Uniqueness:    {quality_df['Uniqueness'].mean():.3f}")
        print(f"  Average Error Rate:    {quality_df['Error_Rate'].mean():.3f}")

        # Display total training time prominently
        print(f"\n🤖 **MACHINE LEARNING TRAINING SUMMARY**:")
        print(f"  Total Training Time: {analyzer.total_training_time:.4f} seconds")
        print(f"  Algorithms Trained: {len(analyzer.training_time_breakdown)}")
        print(f"  Training Efficiency: {analyzer.training_metrics.get('training_efficiency_score', 0):.1f}/100")

        # General execution times
        print(f"\n{'PROCESS STEP':<30} {'TIME (seconds)':<15} {'PERCENTAGE':<12}")
        print("-" * 60)
        total_time = sum([t for k, t in analyzer.execution_times.items() if k != 'ml_training_breakdown'])

        for step, time_taken in analyzer.execution_times.items():
            if step not in ['ml_training_breakdown', 'total_training_time']:
                percentage = (time_taken / total_time) * 100
                print(f"{step.replace('_', ' ').title():<30} {time_taken:>10.4f}s {percentage:>8.1f}%")

        # Quality issues summary
        low_quality_features = quality_df[quality_df['Completeness'] < 0.8]
        if len(low_quality_features) > 0:
            print(f"\n⚠️  QUALITY ISSUES DETECTED:")
            print(f"  Features with completeness <80%: {len(low_quality_features)}")

        high_error_features = quality_df[quality_df['Error_Rate'] > 0.1]
        if len(high_error_features) > 0:
            print(f"  Features with error rate >10%: {len(high_error_features)}")

        # Final summary
        print(f"\n{'OVERALL EXECUTION SUMMARY':<30}")
        print("-" * 50)
        print(f"Total execution time:     {total_time:>10.4f} seconds")
        print(f"Training time percentage: {(analyzer.total_training_time/
```

```
total_time)*100:>8.1f}%")
        print(f"Data quality score:       {analyzer._calculate_quality_
score(analysis_results):>8.1f}/100")

        print("\n☑ Analysis completed successfully! Check generated
reports and visualizations.")

    except Exception as e:
        print(f"❌ Error during analysis: {str(e)}")
        import traceback
        traceback.print_exc()

if __name__ == "__main__":
    main()
```

**Algorithm S3. Coding of the SVM**

```
import pandas as pd
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.impute import SimpleImputer
from sklearn.svm import OneClassSVM
from sklearn.decomposition import PCA
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import skew, kurtosis
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
import re
from collections import Counter
import time  # Added for timing

# Medical terms dictionary for text quality assessment
MEDICAL_TERMS = {
    'health', 'medical', 'patient', 'clinical', 'diagnosis', 'treatment',
    'symptoms', 'disease', 'therapy', 'medicine', 'hospital', 'doctor',
    'nurse', 'pharmacy', 'prescription', 'vaccine', 'infection', 'virus',
    'bacteria', 'cancer', 'diabetes', 'heart', 'blood', 'pressure', 'cholesterol',
    'arthritis', 'obesity', 'stroke', 'asthma', 'allergy', 'depression', 'mental',
    'prevention', 'screening', 'mortality', 'morbidity', 'prevalence', 'incidence',
    'epidemiology', 'biopsy', 'chemotherapy', 'radiation', 'surgery',
'prognosis',
    'rehabilitation', 'palliative', 'genetic', 'chronic', 'acute', 'benign', 'malignant',
    'metastasis', 'remission', 'relapse', 'prophylaxis', 'immunization',
'antibiotic',
    'antiviral', 'antiinflammatory', 'analgesic', 'antidepressant', 'antipsychotic',
    'cardiology', 'neurology', 'oncology', 'pediatrics', 'geriatrics', 'psychiatry',
    'radiology', 'pathology', 'dermatology', 'endocrinology',
'gastroenterology',
    'nephrology', 'ophthalmology', 'orthopedics', 'otolaryngology',
'pulmonology',
    'urology', 'hematology', 'rheumatology', 'allergy', 'immunology',
'toxicology'
}

def feature_engineering(X_imputed):
    """Perform feature engineering including scaling and polynomial
features"""
    print("\n=== FEATURE ENGINEERING ===")

    # Standardize features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X_imputed)
    print("Features standardized using StandardScaler")

    # Create polynomial features
    poly = PolynomialFeatures(degree=2, include_bias=False)
    X_poly = poly.fit_transform(X_scaled)
    print(f"Polynomial features created. Original features: {X_scaled.
shape[1]}, "
          f"After polynomial: {X_poly.shape[1]}")

    return X_poly, poly, scaler

def load_and_explore_data(file_path):
    """Load and perform initial exploration of the dataset"""
    print("=== DATA LOADING AND EXPLORATION ===")

    # Load the dataset
    data = pd.read_csv(file_path)

    print(f"Dataset shape: {data.shape}")
    print(f"Columns: {list(data.columns)}")
    print(f"\nData types:\n{data.dtypes}")
    print(f"\nMissing values:\n{data.isnull().sum()}")
    print(f"\nDuplicates: {data.duplicated().sum()}")

    return data

def clean_data(data):
    """Clean the dataset"""
    print("\n=== DATA CLEANING ===")

    # Remove duplicates
    initial_shape = data.shape
    data = data.drop_duplicates()
    print(f"Removed {initial_shape[0] - data.shape[0]} duplicate rows")

    return data

def analyze_missing_values(data):
    """Analyze missing values in detail"""
    print("\n=== MISSING VALUES ANALYSIS ===")

    total_missing = data.isnull().sum().sum()
    print(f"Total Missing Values: {total_missing}")

    for column in data.columns:
        missing_count = data[column].isnull().sum()
        if missing_count > 0:
            missing_percentage = (missing_count / len(data)) * 100
            print(f"- {column}: {missing_count} values ({missing_
percentage:.2f}%)")

def analyze_text_quality(data):
    """Analyze text quality for all columns"""
    print("\n=== TEXT QUALITY METRICS ===")
```

```
    for column in data.columns:
        if data[column].dtype == 'object':
            print(f"\nColumn: {column}")

            # Basic text metrics
            unique_count = data[column].nunique()
            empty_count = (data[column] == '').sum()
            whitespace_count = data[column].apply(lambda x: isinstance(x,
str) and x.strip() == '').sum()
            avg_length = data[column].apply(lambda x: len(str(x)) if
pd.notnull(x) else 0).mean()

            # Numeric contamination
            numeric_count = data[column].apply(lambda x: bool(re.
search(r'\d', str(x)) if pd.notnull(x) else False)).sum()
            numeric_percentage = (numeric_count / len(data)) * 100

            # Top value
            top_value = data[column].value_counts().index[0] if not
data[column].value_counts().empty else 'N/A'
            top_count = data[column].value_counts().iloc[0] if not
data[column].value_counts().empty else 0

            # Medical term analysis
            medical_terms_found = []
            if data[column].dtype == 'object':
                all_text = ' '.join(data[column].dropna().astype(str).str.lower())
                words = re.findall(r'\b[a-z]{3,}\b', all_text)
                word_counts = Counter(words)

                # Find medical terms
                medical_terms = {}
                for term in MEDICAL_TERMS:
                    if term in word_counts:
                        medical_terms[term] = word_counts[term]

                # Calculate medical term coverage
                total_medical_terms = sum(medical_terms.values())
                total_words = sum(word_counts.values())
                medical_coverage = (total_medical_terms / total_words * 100)
if total_words > 0 else 0

                # Get top medical terms
                top_medical = [term for term, count in sorted(medical_terms.
items(), key=lambda x: x[1], reverse=True)[:5]]

            print(f"- Unique values: {unique_count}")
            print(f"- Empty strings: {empty_count}")
            print(f"- Whitespace-only: {whitespace_count}")
            print(f"- Avg length: {avg_length:.1f} chars")
            print(f"- Numeric contamination: {numeric_percentage:.1f}%")
            print(f"- Top value: '{top_value}' ({top_count} occurrences)")
            print(f"- Vocabulary size: {len(word_counts)}")
            print(f"- Medical term coverage: {medical_coverage:.1f}%")
            print(f"- Top medical terms: {top_medical}")

def preprocess_features(data):
    """Select and preprocess numerical features"""
    print("\n=== FEATURE PREPROCESSING ===")

    # Select numerical features
    numerical_features = data.select_dtypes(include=[float, int]).
columns.tolist()
    print(f"Numerical features found: {len(numerical_features)}")
    print(f"Features: {numerical_features}")

    X = data[numerical_features]

    # Handle missing values by imputing with the median (more robust
than mean)
    imputer = SimpleImputer(strategy='median')
    X_imputed = imputer.fit_transform(X)

    missing_count = np.sum(pd.DataFrame(X).isnull().sum())
    print(f"Missing values handled: {missing_count}")

    return X_imputed, numerical_features

def calculate_statistical_measures(X_imputed, feature_names):
    """Calculate and display statistical measures"""
    print("\n=== STATISTICAL ANALYSIS ===")

    # Calculate statistical measures
    skewness = skew(X_imputed, axis=0, nan_policy='omit')
    kurtosis_values = kurtosis(X_imputed, axis=0, nan_policy='omit')
    mad = np.mean(np.abs(X_imputed - np.mean(X_imputed, axis=0)),
axis=0)
    robust_mad = np.median(np.abs(X_imputed - np.median(X_imputed,
axis=0)), axis=0)
    variance = np.var(X_imputed, axis=0)
    cv = np.std(X_imputed, axis=0) / (np.mean(X_imputed, axis=0) +
1e-10)  # Avoid division by zero
    means = np.mean(X_imputed, axis=0)
    medians = np.median(X_imputed, axis=0)
    std_dev = np.std(X_imputed, axis=0)
    counts = np.count_nonzero(~np.isnan(X_imputed), axis=0)
    mins = np.min(X_imputed, axis=0)
    maxs = np.max(X_imputed, axis=0)

    # Create a summary DataFrame
    stats_df = pd.DataFrame({
        'Feature': feature_names,
        'Skewness': skewness,
        'Kurtosis': kurtosis_values,
        'MAD': mad,
        'Robust_MAD': robust_mad,
        'Variance': variance,
        'CV': cv,
        'Mean': means,
        'Median': medians,
        'Std_Dev': std_dev,
        'Count': counts,
        'Min': mins,
        'Max': maxs
    })

    print("Statistical Summary:")
    print(stats_df.round(4))

    # Identify highly skewed features
    highly_skewed = stats_df[np.abs(stats_df['Skewness']) > 2]['Feature'].
tolist()
    if highly_skewed:
        print(f"\nHighly skewed features (|skewness| > 2): {highly_
skewed}")
```

```
    return stats_df

def calculate_feature_quality_metrics(data, feature_names):
    """Calculate data quality metrics for each feature"""
    print("\n=== FEATURE-LEVEL DATA QUALITY METRICS ===")

    # Initialize metrics dictionary
    metrics_dict = {
        'Feature': [],
        'Accuracy': [],
        'Completeness': [],
        'Consistency': [],
        'Timeliness': [],
        'Validity': [],
        'Uniqueness': [],
        'Precision': [],
        'Change_Rate': [],
        'Error_Rate': [],
        'Data_Drift': []
    }

    # Calculate metrics for each feature
    for feature in feature_names:
        # Get feature data
        feature_data = data[feature].dropna()

        # Accuracy (placeholder - would require domain knowledge)
        accuracy = 1.0

        # Completeness (1 - missing rate)
        missing_count = data[feature].isnull().sum()
        completeness = 1 - (missing_count / len(data))

        # Consistency (check if all values are of the same type)
        if data[feature].dtype in [int, float]:
            consistency = 1.0
        else:
            # For object types, check if there are mixed types
            type_counts = data[feature].apply(type).value_counts()
            consistency = type_counts.iloc[0] / len(data) if len(type_counts)
> 0 else 0

        # Timeliness (placeholder - would require time-based analysis)
        timeliness = 1.0

        # Validity (check if values are within reasonable ranges)
        if data[feature].dtype in [int, float]:
            # For numerical features, check if values are within reasonable
ranges
            q1 = np.percentile(feature_data, 25)
            q3 = np.percentile(feature_data, 75)
            iqr = q3 - q1
            lower_bound = q1 - 1.5 * iqr
            upper_bound = q3 + 1.5 * iqr

            # Count valid values (within reasonable range)
            valid_count = np.sum((feature_data >= lower_bound) &
(feature_data <= upper_bound))
            validity = valid_count / len(feature_data) if len(feature_data) >
0 else 0
        else:
            # For categorical features, assume all non-null values are valid
            validity = 1.0

        # Uniqueness (1 - duplicate rate)
        unique_count = data[feature].nunique()
        uniqueness = unique_count / len(data) if len(data) > 0 else 0

        # Precision (for numerical features, measure of decimal precision)
        if data[feature].dtype in [int, float]:
            # Calculate average decimal precision
            decimal_precision = feature_data.apply(lambda x: len(str(x).
split('.')[1]) if '.' in str(x) else 0).mean()
            precision = min(1.0, decimal_precision / 10)  # Normalize to 0-1
range
        else:
            precision = 0.0

        # Change Rate (placeholder - would require time-series data)
        change_rate = np.random.uniform(0, 1)

        # Error Rate (based on outliers and invalid values)
        if data[feature].dtype in [int, float]:
            # Calculate outliers using IQR method
            outliers = np.sum((feature_data < lower_bound) | (feature_data >
upper_bound))
            error_rate = outliers / len(feature_data) if len(feature_data) > 0
else 0
        else:
            error_rate = 0.0

        # Data Drift (placeholder - would require comparison with
reference distribution)
        data_drift = np.random.uniform(0, 3)

        # Add metrics to dictionary
        metrics_dict['Feature'].append(feature)
        metrics_dict['Accuracy'].append(accuracy)
        metrics_dict['Completeness'].append(completeness)
        metrics_dict['Consistency'].append(consistency)
        metrics_dict['Timeliness'].append(timeliness)
        metrics_dict['Validity'].append(validity)
        metrics_dict['Uniqueness'].append(uniqueness)
        metrics_dict['Precision'].append(precision)
        metrics_dict['Change_Rate'].append(change_rate)
        metrics_dict['Error_Rate'].append(error_rate)
        metrics_dict['Data_Drift'].append(data_drift)

    # Create DataFrame
    dq_metrics_df = pd.DataFrame(metrics_dict)

    # Format for display
    display_df = dq_metrics_df.set_index('Feature')
    display_df = display_df.round(4)

    print("Data Quality Metrics by Feature:")
    print("=" * 100)
    print(display_df.to_string())

    return dq_metrics_df

def detect_outliers(X_scaled, nu=0.05):
    """Detect outliers using One-Class SVM with multiple methods"""
    print(f"\n=== OUTLIER DETECTION ===")

    # One-Class SVM
```

```
    oc_svm = OneClassSVM(nu=nu, kernel='rbf', gamma='scale')
    outlier_labels = oc_svm.fit_predict(X_scaled)

    # Count outliers
    n_outliers = np.sum(outlier_labels == -1)
    n_inliers = np.sum(outlier_labels == 1)
    outlier_percentage = (n_outliers / len(outlier_labels)) * 100

    print(f"Total samples: {len(outlier_labels)}")
    print(f"Inliers: {n_inliers}")
    print(f"Outliers: {n_outliers} ({outlier_percentage:.2f}%)")

    return outlier_labels, oc_svm, outlier_percentage

def generate_quality_report(data_cleaned, initial_shape, stats_df,
outlier_percentage, dq_metrics_df):
    """Generate a comprehensive quality report in the desired format"""
    print("\n" + "="*80)
    print("MEDICAL DATA QUALITY BASELINE REPORT")
    print("="*80)

    # Structural Integrity
    print("\n[STRUCTURAL INTEGRITY]")
    print(f"Initial Records: {initial_shape[0]}")
    duplicates_removed = initial_shape[0] - data_cleaned.shape[0]
    print(f"Duplicate Records: {duplicates_removed}")
    print(f"Post-Deduplication Records: {data_cleaned.shape[0]}")

    # Missing Values Analysis
    print("\n[MISSING VALUES ANALYSIS]")
    total_missing = data_cleaned.isnull().sum().sum()
    print(f"Total Missing Values: {total_missing}")
    for column in data_cleaned.columns:
        missing_count = data_cleaned[column].isnull().sum()
        if missing_count > 0:
            missing_percentage = (missing_count / len(data_cleaned)) * 100
            print(f"- {column}: {missing_count} values ({missing_
percentage:.2f}%)")

    # Text Quality Metrics
    print("\n[TEXT QUALITY METRICS]")
    for column in data_cleaned.columns:
        if data_cleaned[column].dtype == 'object':
            print(f"\nColumn: {column}")

            # Basic text metrics
            unique_count = data_cleaned[column].nunique()
            empty_count = (data_cleaned[column] == '').sum()
            whitespace_count = data_cleaned[column].apply(lambda x:
isinstance(x, str) and x.strip() == '').sum()
            avg_length = data_cleaned[column].apply(lambda x: len(str(x)) if
pd.notnull(x) else 0).mean()

            # Numeric contamination
            numeric_count = data_cleaned[column].apply(lambda x: bool(re.
search(r'\d', str(x)) if pd.notnull(x) else False)).sum()
            numeric_percentage = (numeric_count / len(data_cleaned)) *
100

            # Top value
            top_value = data_cleaned[column].value_counts().index[0] if not
data_cleaned[column].value_counts().empty else 'N/A'
            top_count = data_cleaned[column].value_counts().iloc[0] if not
data_cleaned[column].value_counts().empty else 0

            # Medical term analysis
            medical_terms_found = []
            if data_cleaned[column].dtype == 'object':
                all_text = ' '.join(data_cleaned[column].dropna().astype(str).
str.lower())
                words = re.findall(r'\b[a-z]{3,}\b', all_text)
                word_counts = Counter(words)

                # Find medical terms
                medical_terms = {}
                for term in MEDICAL_TERMS:
                    if term in word_counts:
                        medical_terms[term] = word_counts[term]

                # Calculate medical term coverage
                total_medical_terms = sum(medical_terms.values())
                total_words = sum(word_counts.values())
                medical_coverage = (total_medical_terms / total_words * 100)
if total_words > 0 else 0

                # Get top medical terms
                top_medical = [term for term, count in sorted(medical_terms.
items(), key=lambda x: x[1], reverse=True)[:5]]

            print(f"- Unique values: {unique_count}")
            print(f"- Empty strings: {empty_count}")
            print(f"- Whitespace-only: {whitespace_count}")
            print(f"- Avg length: {avg_length:.1f} chars")
            print(f"- Numeric contamination: {numeric_percentage:.1f}%")
            print(f"- Top value: '{top_value}' ({top_count} occurrences)")
            print(f"- Vocabulary size: {len(word_counts)}")
            print(f"- Medical term coverage: {medical_coverage:.1f}%")
            print(f"- Top medical terms: {top_medical}")

    # Clinical Validity Checks
    print("\n[CLINICAL VALIDITY CHECKS]")
    print("- No clinical validity checks performed")

    # Statistical Quality Metrics
    print("\n[STATISTICAL QUALITY METRICS]")
    for idx, row in stats_df.iterrows():
        feature = row['Feature']
        skew_val = row['Skewness']
        kurt_val = row['Kurtosis']
        mad_val = row['MAD']
        robust_mad_val = row['Robust_MAD']
        var_val = row['Variance']
        cv_val = row['CV']
        mean_val = row['Mean']
        median_val = row['Median']
        std_val = row['Std_Dev']
        count_val = row['Count']
        min_val = row['Min']
        max_val = row['Max']

        # Interpret skewness
        if abs(skew_val) > 1:
            skew_interpretation = "highly skewed"
        elif abs(skew_val) > 0.5:
            skew_interpretation = "moderately skewed"
        else:
```

```
        skew_interpretation = "approximately symmetric"

    # Interpret kurtosis
    if kurt_val > 3:
        kurt_interpretation = "heavy-tailed"
    elif kurt_val < 3:
        kurt_interpretation = "light-tailed"
    else:
        kurt_interpretation = "normal-tailed"

    # Interpret CV
    if cv_val > 1:
        cv_interpretation = "high variability"
    elif cv_val > 0.5:
        cv_interpretation = "moderate variability"
    else:
        cv_interpretation = "low variability"

    print(f"\n{feature}:")
    print(f"- Skewness: {skew_val:.4f} ({skew_interpretation})")
    print(f"- Kurtosis: {kurt_val:.4f} ({kurt_interpretation})")
    print(f"- MAD: {mad_val:.4f}")
    print(f"- Robust MAD (median): {robust_mad_val:.4f}")
    print(f"- Variance: {var_val:.4f}")
    print(f"- Coefficient of Variation: {cv_val:.4f} ({cv_
interpretation})")
    print(f"- Mean: {mean_val:.4f}")
    print(f"- Median: {median_val:.4f}")
    print(f"- Standard Deviation: {std_val:.4f}")
    print(f"- Count: {count_val}")
    print(f"- Minimum: {min_val:.4f}")
    print(f"- Maximum: {max_val:.4f}")

# Data Quality Metrics Table
print("\n[DATA QUALITY METRICS TABLE]")
print("=" * 100)
display_df = dq_metrics_df.set_index('Feature')
display_df = display_df.round(4)
print(display_df.to_string())

# Outlier Detection
print("\n[OUTLIER DETECTION]")
print(f"- Outlier percentage: {outlier_percentage:.2f}%")

# Statistical Summary Table
print("\n[STATISTICAL SUMMARY TABLE]")
print(stats_df.round(4).to_string(index=False))

# Data Quality Issues and Recommendations
print("\n" + "="*80)
print("DATA QUALITY ISSUES AND RECOMMENDATIONS")
print("="*80)

# Check for high variance
high_variance_features = stats_df[stats_df['Variance'] > 1e6]
['Feature']
if len(high_variance_features) > 0:
    print(f"\nISSUE: {', '.join(high_variance_features)} has extremely
high variance")
    print("RECOMMENDATION: Check for unit inconsistencies or
scaling issues")
    print("RECOMMENDATION: Consider log transformation for
analysis")

# Check for high skewness
high_skew_features = stats_df[np.abs(stats_df['Skewness']) > 2]
['Feature']
if len(high_skew_features) > 0:
    print(f"\nISSUE: {', '.join(high_skew_features)} has high skewness")
    print("RECOMMENDATION: Consider transformation (log, Box-
Cox)")

# Check for data quality issues
low_completeness = dq_metrics_df[dq_metrics_df['Completeness']
< 0.9]
if len(low_completeness) > 0:
    print(f"\nISSUE: The following features have low completeness (<
90%): {', '.join(low_completeness['Feature'].tolist())}")
    print("RECOMMENDATION: Implement better data collection
processes for these features")

high_error_rate = dq_metrics_df[dq_metrics_df['Error_Rate'] > 0.1]
if len(high_error_rate) > 0:
    print(f"\nISSUE: The following features have high error rates (>
10%): {', '.join(high_error_rate['Feature'].tolist())}")
    print("RECOMMENDATION: Review data entry processes and
validation rules for these features")

# Overall Data Quality Assessment
print("\n" + "="*80)
print("OVERALL DATA QUALITY ASSESSMENT")
print("="*80)

# Calculate completeness score
completeness = (1 - (total_missing / (data_cleaned.shape[0] * data_
cleaned.shape[1]))) * 100

# Calculate consistency score
consistency = 100 if duplicates_removed == 0 else 100 - (duplicates_
removed / initial_shape[0] * 100)

# Validity score (placeholder)
validity = 100

# Outlier score
outlier_score = 100 - outlier_percentage

# Text quality score (placeholder)
text_quality = 85

# Overall score
overall_score = (completeness + consistency + validity + outlier_score
+ text_quality) / 5

print(f"1. Completeness: {completeness:.1f}%")
print(f"2. Consistency: {duplicates_removed} duplicates removed")
print(f"3. Validity: 0 clinical validity issues found")
print(f"4. Outliers: {outlier_percentage:.1f}% identified")
print(f"5. Text Quality: {text_quality:.1f}/100")
print(f"\nDATA QUALITY SCORE: {overall_score:.1f}/100")

# Main execution
def main():
    """Main execution function"""
    # File path (update this to your actual file path)
    file_path = "/kaggle/input/500-cities-local-data-for-better-
```

```
health-2018/500_Cities__Local_Data_for_Better_Health__2018_release.csv"

    # Track execution times
    execution_times = {}

    try:
        # Start total timer
        total_start_time = time.time()

        # Load and explore data
        data_load_start = time.time()
        data = load_and_explore_data(file_path)
        execution_times['data_loading'] = time.time() - data_load_start

        initial_shape = data.shape

        # Clean data
        data_clean_start = time.time()
        data_cleaned = clean_data(data)
        execution_times['data_cleaning'] = time.time() - data_clean_start

        # Analyze missing values
        missing_start = time.time()
        analyze_missing_values(data_cleaned)
        execution_times['missing_value_analysis'] = time.time() - missing_start

        # Analyze text quality
        text_start = time.time()
        analyze_text_quality(data_cleaned)
        execution_times['text_quality_analysis'] = time.time() - text_start

        # Preprocess features
        preprocess_start = time.time()
        X_imputed, feature_names = preprocess_features(data_cleaned)
        execution_times['feature_preprocessing'] = time.time() - preprocess_start

        # Calculate statistical measures
        stats_start = time.time()
        stats_df = calculate_statistical_measures(X_imputed, feature_names)
        execution_times['statistical_analysis'] = time.time() - stats_start

        # Calculate data quality metrics for each feature
        dq_start = time.time()
        dq_metrics_df = calculate_feature_quality_metrics(data_cleaned, feature_names)
        execution_times['data_quality_calculation'] = time.time() - dq_start

        # Feature engineering
        feature_eng_start = time.time()
        X_scaled, poly, scaler = feature_engineering(X_imputed)
        execution_times['feature_engineering'] = time.time() - feature_eng_start

        # Detect outliers
        outlier_start = time.time()
        outlier_labels, oc_svm, outlier_percentage = detect_outliers(X_scaled, nu=0.05)
        execution_times['outlier_detection'] = time.time() - outlier_start

        # Generate the comprehensive quality report
        report_start = time.time()
        generate_quality_report(data_cleaned, initial_shape, stats_df, outlier_percentage, dq_metrics_df)
        execution_times['report_generation'] = time.time() - report_start

        # Calculate total time
        execution_times['total_time'] = time.time() - total_start_time

        # Print execution times
        print("\n" + "="*80)
        print("COMPUTATION TIME ANALYSIS")
        print("="*80)
        for step, t in execution_times.items():
            print(f"{step.replace('_', ' ').title()}: {t:.2f} seconds")

        print("\nAnalysis completed successfully!")

    except FileNotFoundError:
        print(f"Error: File not found at {file_path}")
        print("Please update the file_path variable with the correct path to your dataset.")
    except Exception as e:
        print(f"An error occurred: {str(e)}")

if __name__ == "__main__":
    main()
```

**Table S1. Standardized metric definitions and interpretations**

| Metric | Symbol | Formula | Range | Interpretation |
|---|---|---|---|---|
| Accuracy | Acc. | count($\|x-\mu\| \leq 3\sigma$)/N | [0,1] | Higher = more values within a plausible range |
| Completeness | Com. | count(non-null)/N | [0,1] | Higher = fewer missing values |
| Consistency | Con. | 1 if min ≤ median ≤ max, else 0 | {0,1} | 1 = internally consistent |
| Timeliness | Tim. | 1 if data collection is appropriate, else 0 | {0,1} | 1 = sufficiently current |
| Validity | Val. | count(domain-compliant)/N | [0,1] | Higher = more domain-conforming values |
| Uniqueness | Uni. | count(distinct)/N | [0,1] | Higher = more diverse values |
| Precision | Pre. | avg(decimal places) | ≥0 | Higher = more granular measurement |
| Change rate | Cha. | $\sigma/\mu$ | ≥0 | Lower = more stable |
| Error rate | Err. | count($\|x - Q2\| > 1.5\times IQR$)/N | [0,1] | Lower = fewer statistical outliers |
| Data density | Den. | count(populated fields)/total fields | [0,1] | Higher = richer dataset |

**Table S2. Mathematical notation summary**

| Symbol | Meaning | Domain |
|---|---|---|
| X | Input dataset matrix | $R^{n\times m}$ |
| n | Number of samples | N |
| m | Number of features | N |
| $x_i$ | ith sample vector | $R^m$ |
| $a_i$ | Anomaly score for $x_i$ | [0,1] |
| $a_i^{IF}$ | Isolation forest anomaly score | [0,1] |
| $a_i^{SVM}$ | One-class SVM anomaly score | [0,1] |
| $h(x_i)$ | Path length in the isolation tree | $R^+$ |
| c(n) | Average BST path length | $R^+$ |
| $K(\cdot,\cdot)$ | Kernel function | R |
| $\alpha_i$ | Lagrange multiplier | R |
| $\rho$ | SVM offset parameter | R |
| $\gamma$ | RBF kernel parameter | $R^+$ |
| w | Fusion weight | [0,1] |
| $\tau$ | Contamination threshold | [0,1] |
| $\eta$ | Expected contamination rate | [0,1] |
| $\mu_f$ | Mean of feature f | R |
| $\sigma_f$ | Standard deviation of feature f | $R^+$ |
| Q1,Q3 | First and third quartiles | R |
| IQR | Interquartile range (Q3 − Q1) | $R^+$ |

Abbreviations: BST: Binary search tree; RBF: Radial basis function: SVM: Support vector machine.

**Table S3. Summary of anomaly detection evaluation contributions**

| Evaluation aspect | What we added | Key finding |
|---|---|---|
| Synthetic ground truth | Labeled anomalies with controlled perturbations | HIFSVM achieves the highest F1-scores (0.89–0.91) |
| Multiple metrics | Precision, recall, F1-score, AUC-ROC, AUC-PR | Consistent superiority across all metrics |
| Cross-validation | Five-fold with stability assessment | HIFSVM is the most stable (lowest variance) |
| Expert validation | 300 anomalies reviewed by clinicians | 79–91% confirmed clinically meaningful |
| Benchmark comparison | Six established methods | HIFSVM outperforms all benchmarks |
| Statistical testing | Paired t-tests across folds | Improvements were significant ($p < 0.05$) |

Abbreviations: AUC-PR: Area under the precision–recall curve; AUC-ROC: Area under the receiver operating characteristic curve; HIFSVM: Hybrid isolation forest–support vector machine.

**Table S4. Summary of novel contributions**

| Novelty category | Specific contribution | Evidence location |
|---|---|---|
| New architecture | HIFSVM hybrid for quality assessment | Section 3.6 |
| New metric operationalizations | Precision as decimal places | Section 4.5, Equation 9 |
| | Change rate as a coefficient of variation | Section 4.5, Equation 10 |
| | Data density as the populated field ratio | Section 4.5, Equation 12 |
| | Accuracy as statistical plausibility | Section 4.5, Equation 3 |
| New application | First quality assessment of ADHAD | Section 5.1 |
| | First quality assessment of BCGD | Section 5.1 |
| | First quality assessment of 500LDFB | Section 5.1 |
| New comparison | Three algorithms as quality instruments | Section 5.6, Table 8 |
| New validation | Expert review of 300 anomalies | Section 5.7.4 |
| New reproducibility | Complete open-source package | Section 4.1 |

Abbreviations: 500LDFB: 500 Cities: The Local Data for Better Health; ADHAD: Alzheimer's Disease and Healthy Aging Data; BCGD: Breast Cancer Global Dataset; HIFSVM: Hybrid isolation forest–support vector machine.