Project Files Content

FILE: pipeline.log

FILE: ted_ml_pipeline.py

```
#!/usr/bin/env python3
"""
EU Procurement Monitoring System - TED ML Pipeline
This script provides the main entry point for the TED procurement data processing and outlier detection.
It offers a simplified architecture focusing on core functionality.
"""
import os
import sys
import argparse
import pandas as pd
import json
from datetime import datetime, timedelta
import logging
# Import pipeline components
from components.ted_data_retriever import TEDDataRetriever
from components.ted_data_preprocessor import TEDDataPreprocessor
from components.ted_data_storage import TEDDataStorage
from transforming.isolation_forest_model import IsolationForestModel
from visualization.visualizer import TEDVisualizer
# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("pipeline.log"),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)
class TEDMLPipeline:
    """Main class for the EU procurement data processing and outlier detection pipeline"""

    def __init__(self, base_dir="."):
        """
        Initialize the pipeline with directory structure

        Args:
            base_dir (str): Base directory for all data and output
        """
        self.base_dir = base_dir

        # Set up directory structure
        self.directories = {
```

```
        "raw_data": os.path.join(base_dir, "data", "raw"),
        "processed_data": os.path.join(base_dir, "data", "processed"),
        "models": os.path.join(base_dir, "models"),
        "output": os.path.join(base_dir, "output"),
        "visualizations": os.path.join(base_dir, "visualizations")
    }

    # Create directories if they don't exist
    for directory in self.directories.values():
        os.makedirs(directory, exist_ok=True)

    # Default training data file
    self.training_data_file = os.path.join(self.directories["processed_data"], "training_data.csv")

    # Initialize components
    self.storage = TEDDataStorage(
        db_path=os.path.join(self.directories["processed_data"], "ted_results.db")
    )

    self.visualizer = TEDVisualizer(
        output_dir=self.directories["visualizations"]
    )

    self.model = None

    logger.info("TEDMLPipeline initialized")

def sync_data(self, days_back=30, country=None, max_pages=5, store=True):
    """
    Synchronize data from TED API

    Args:
        days_back (int): Number of days to look back for data
        country (str): Optional country code filter
        max_pages (int): Maximum number of pages to fetch from API
        store (bool): Whether to store data in the database

    Returns:
        dict: Dictionary with paths to output files
    """
    logger.info(f"Starting data synchronization (days_back={days_back}, country={country})")

    # Calculate date range
    end_date = datetime.now()
    start_date = end_date - timedelta(days=days_back)

    # Format dates for API
    start_date_str = start_date.strftime("%Y%m%d")
    end_date_str = end_date.strftime("%Y%m%d")

    print(f"Synchronizing data from {start_date_str} to {end_date_str}")
```

```
    try:
        # Step 1: Retrieve data from TED API
        data_retriever = TEDDataRetriever(data_dir=self.directories["raw_data"])

        # Fetch data
        df, raw_data_file = data_retriever.fetch_notices(
            start_date=start_date_str,
            end_date=end_date_str,
            country=country,
            max_pages=max_pages
        )

        if df.empty or not raw_data_file:
            print("No new data retrieved from API")
            return None

        # Step 2: Preprocess data
        print(f"Processing {len(df)} records...")

        preprocessor = TEDDataPreprocessor(
            input_file=raw_data_file,
            output_dir=self.directories["processed_data"]
        )

        # Process and save data
        output_files = preprocessor.save_output()

        # Step 3: Store processed data if requested
        if store and 'ml_dataset' in output_files:
            print("Data ready for model processing and storage")

        print(f"Data synchronization completed successfully: {len(df)} records processed")

        # Return information about the outputs
        result = {
            "raw_file": raw_data_file,
            "record_count": len(df),
            "processed_files": output_files,
            "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        }

        return result

    except Exception as e:
        logger.error(f"Error during data synchronization: {e}")
        print(f"Error: {str(e)}")
        return None
def train(self, input_file=None, sample_size=None, contamination=0.05):
    """
    Train an outlier detection model

    Args:
```

```
    input_file (str): Optional path to training data file
    sample_size (int): Optional sample size to use for training
    contamination (float): Expected proportion of outliers (0.0-0.5)

Returns:
    str: Path to the saved model file
"""
logger.info(f"Starting model training (sample_size={sample_size}, contamination={contamination})")

try:
    # Step 1: Determine training file to use
    if input_file:
        training_file = input_file
        print(f"Using specified training file: {training_file}")
    else:
        # Use default training file
        training_file = self.training_data_file
        print(f"Using default training file: {training_file}")

    # Check if file exists
    if not os.path.exists(training_file):
        print(f"Error: Training file {training_file} does not exist")
        return None

    # Step 2: Load training data
    print("Loading training data...")
    try:
        ml_df = pd.read_csv(training_file)
        print(f"Loaded {len(ml_df)} records with {len(ml_df.columns)} features")
    except Exception as e:
        print(f"Error loading training data: {e}")
        return None

    # Step 3: Initialize and train the model
    print("Training Isolation Forest model...")
    self.model = IsolationForestModel(
        model_path=os.path.join(self.directories["models"], "isolation_forest_model.pkl"),
        contamination=contamination
    )

    # Train the model
    self.model.train(ml_df, sample_size=sample_size)

    # Step 4: Save the trained model
    print("Saving trained model...")
    model_path = self.model.save_model()

    print(f"Model training completed successfully: {model_path}")
    return model_path

except Exception as e:
    logger.error(f"Error during model training: {e}")
```

```
        print(f"Error: {str(e)}")
        return None

def predict(self, input_file=None, country=None, start_date=None, end_date=None,
            max_bid_amount=None, visualize=False):
    """
    Run prediction to detect outliers in procurement data

    Args:
        input_file (str): Optional path to input file (if not provided, will fetch from API)
        country (str): Optional country code filter (for API fetching)
        start_date (str): Start date in format YYYYMMDD (for API fetching)
        end_date (str): End date in format YYYYMMDD (for API fetching)
        max_bid_amount (float): Optional maximum bid amount filter (for API fetching)
        visualize (bool): Whether to generate visualizations

    Returns:
        dict: Dictionary with paths to output files
    """
    logger.info(f"Starting prediction (visualize={visualize})")

    # Create timestamp for file naming
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

    try:
        # Step 1: Get data (either from file or API)
        if input_file and os.path.exists(input_file):
            print(f"Using provided input file: {input_file}")
            ml_dataset_file = input_file
        else:
            print("Fetching data from TED API...")

            if not start_date or not end_date:
                print("Error: start_date and end_date are required for API data fetching")
                return None

            # Fetch from API
            data_retriever = TEDDataRetriever(data_dir=self.directories["raw_data"])

            _, raw_data_file = data_retriever.fetch_notices(
                start_date=start_date,
                end_date=end_date,
                max_bid_amount=max_bid_amount,
                country=country,
                max_pages=5
            )

            if not raw_data_file:
                print("Error: No data retrieved from API")
                return None

            # Preprocess the data
```

```
        print("Preprocessing data...")
        preprocessor = TEDDataPreprocessor(
            input_file=raw_data_file,
            output_dir=self.directories["processed_data"]
        )

        output_files = preprocessor.save_output()
        ml_dataset_file = output_files["ml_dataset"]

    # Step 2: Load the dataset
    print(f"Loading dataset: {ml_dataset_file}")
    try:
        ml_df = pd.read_csv(ml_dataset_file)
        print(f"Loaded {len(ml_df)} records with {len(ml_df.columns)} features")
    except Exception as e:
        print(f"Error loading dataset: {e}")
        return None

    # Step 3: Load or create the model
    if not self.model:
        print("Loading trained model...")
        model_path = os.path.join(self.directories["models"], "isolation_forest_model.pkl")

        if not os.path.exists(model_path):
            print("No trained model found. Training new model...")
            self.train(input_file=ml_dataset_file)

        self.model = IsolationForestModel(model_path=model_path)
        self.model.load_model()

    # Step 4: Make predictions
    print("Detecting outliers...")
    result_df = self.model.predict(ml_df)

    # Step 5: Save results
    results_file = os.path.join(self.directories["output"], f"outliers_{timestamp}.csv")

    print(f"Saving results to {results_file}")
    result_df.to_csv(results_file, index=False)

    # Count outliers
    if 'is_outlier' in result_df.columns:
        outlier_count = result_df['is_outlier'].sum()
        outlier_pct = outlier_count / len(result_df) * 100
        print(f"Detected {outlier_count} outliers out of {len(result_df)} records ({outlier_pct:.2f}%)")

    # Step 6: Generate visualizations if requested
    viz_files = []
    if visualize:
        print("Generating visualizations...")
        viz_files = self.visualizer.create_visualizations(
            result_df,
```

```
                base_filename=f"ted_analysis_{timestamp}"
            )

            if viz_files:
                print(f"Generated {len(viz_files)} visualizations")

        # Save outliers to storage
        try:
            run_id = self.storage.store_results(
                model_type="isolation_forest",
                result_df=result_df,
                parameters={"contamination": self.model.contamination if self.model else 0.05},
                notes=f"Prediction run from {start_date or 'file'} to {end_date or 'file'}"
            )
            if run_id:
                print(f"Results stored in database with run ID: {run_id}")
        except Exception as e:
            logger.warning(f"Could not store results in database: {e}")

        # Prepare result information
        result = {
            "results_file": results_file,
            "record_count": len(result_df),
            "outlier_count": outlier_count if 'is_outlier' in result_df.columns else 0,
            "visualizations": viz_files,
            "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        }

        print("Prediction completed successfully")
        return result

    except Exception as e:
        logger.error(f"Error during prediction: {e}")
        print(f"Error: {str(e)}")
        return None

def evaluate(self, input_file=None):
    """
    Evaluate model performance

    Args:
        input_file (str): Path to evaluation data file

    Returns:
        dict: Dictionary with evaluation metrics
    """
    logger.info("Starting model evaluation")

    if not input_file or not os.path.exists(input_file):
        print(f"Error: Evaluation file {input_file} does not exist")
        return None
```

```
try:
    # Step 1: Load the dataset
    print(f"Loading evaluation dataset: {input_file}")
    eval_df = pd.read_csv(input_file)

    # Step 2: Load the model if not already loaded
    if not self.model:
        print("Loading model...")
        model_path = os.path.join(self.directories["models"], "isolation_forest_model.pkl")

        if not os.path.exists(model_path):
            print("Error: No trained model found")
            return None

        self.model = IsolationForestModel(model_path=model_path)
        self.model.load_model()

    # Step 3: Make predictions
    print("Running prediction for evaluation...")
    result_df = self.model.predict(eval_df)

    # Step 4: Calculate metrics
    # Note: For unsupervised outlier detection, we don't have ground truth labels
    # So we'll report basic statistics

    outlier_count = result_df['is_outlier'].sum()
    total_count = len(result_df)
    outlier_pct = outlier_count / total_count * 100

    # If value column exists, calculate value statistics
    value_col = None
    for col in ['total-value-eur', 'total-value-eur-capped', 'value_eur']:
        if col in result_df.columns:
            value_col = col
            break

    value_stats = {}
    if value_col:
        normal_avg = result_df[~result_df['is_outlier']][value_col].mean()
        outlier_avg = result_df[result_df['is_outlier']][value_col].mean() if outlier_count > 0 else 0

        value_stats = {
            "normal_avg_value": normal_avg,
            "outlier_avg_value": outlier_avg,
            "value_ratio": outlier_avg / normal_avg if normal_avg > 0 else 0
        }

    # Prepare evaluation metrics
    metrics = {
        "total_records": total_count,
        "outlier_count": outlier_count,
        "outlier_percentage": outlier_pct,
```

```
                "value_statistics": value_stats,
                "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            }

            # Save evaluation results
            eval_file = os.path.join(self.directories["output"], f"evaluation_{datetime.now().strftime('%Y%m%d_%H%M%S'
.json")

            with open(eval_file, 'w') as f:
                json.dump(metrics, f, indent=4)

            print(f"Evaluation completed: {outlier_count} outliers detected ({outlier_pct:.2f}%)")
            print(f"Results saved to {eval_file}")

            return metrics

        except Exception as e:
            logger.error(f"Error during evaluation: {e}")
            print(f"Error: {str(e)}")
            return None

    def generate_report(self, data_file=None, days=30, output_format='pdf'):
        """
        Generate a comprehensive procurement analysis report

        Args:
            data_file (str): Optional path to data file (if not provided, will use recent data)
            days (int): Number of days of data to include if data_file not provided
            output_format (str): Output format ('pdf', 'html', or 'json')

        Returns:
            str: Path to the generated report
        """
        logger.info(f"Generating report (days={days}, format={output_format})")

        try:
            # Step 1: Get data
            if data_file and os.path.exists(data_file):
                print(f"Using provided data file: {data_file}")
                df = pd.read_csv(data_file)
            else:
                print(f"Retrieving data from the last {days} days...")

                # Get data from storage
                filters = {
                    'start_date': (datetime.now() - timedelta(days=days)).strftime("%Y-%m-%d"),
                    'end_date': datetime.now().strftime("%Y-%m-%d")
                }

                df = self.storage.retrieve_data(filters=filters, limit=10000)

                if df.empty:
```

```
        print("No data available for report")
        return None

# Step 2: Run outlier detection if needed
if 'is_outlier' not in df.columns:
    print("Running outlier detection...")

    # Load model
    if not self.model:
        model_path = os.path.join(self.directories["models"], "isolation_forest_model.pkl")

        if os.path.exists(model_path):
            self.model = IsolationForestModel(model_path=model_path)
            self.model.load_model()
        else:
            print("No trained model found. Training new model...")
            self.train(input_file=self.training_data_file)

    # Make predictions
    df = self.model.predict(df)

# Step 3: Generate visualizations
print("Generating visualizations...")
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

viz_files = self.visualizer.create_visualizations(
    df,
    base_filename=f"report_{timestamp}"
)

# Step 4: Generate report based on format
report_dir = os.path.join(self.base_dir, "reports")
os.makedirs(report_dir, exist_ok=True)

report_file = os.path.join(report_dir, f"procurement_report_{timestamp}.{output_format}")

# Generate different formats
if output_format == 'json':
    # Create JSON report
    report_data = {
        "report_date": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        "data_period": f"{days} days",
        "record_count": len(df),
        "outlier_count": int(df['is_outlier'].sum()) if 'is_outlier' in df.columns else 0,
        "visualizations": viz_files,
        "summary_statistics": self.storage.get_statistics()
    }

    with open(report_file, 'w') as f:
        json.dump(report_data, f, indent=4)

else:
```

```
                # For PDF/HTML, we'll just use text for now
                # In a real implementation, you would generate proper PDF/HTML
                with open(report_file, 'w') as f:
                    f.write(f"EU Procurement Analysis Report\n")
                    f.write(f"Generated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}\n")
                    f.write(f"Data period: Last {days} days\n")
                    f.write(f"Records analyzed: {len(df)}\n")

                    if 'is_outlier' in df.columns:
                        outlier_count = df['is_outlier'].sum()
                        outlier_pct = outlier_count / len(df) * 100
                        f.write(f"Outliers detected: {outlier_count} ({outlier_pct:.2f}%)\n")

                    f.write(f"\nVisualizations:\n")
                    for viz in viz_files:
                        f.write(f"- {viz}\n")

            print(f"Report generated: {report_file}")
            return report_file

        except Exception as e:
            logger.error(f"Error generating report: {e}")
            print(f"Error: {str(e)}")
            return None
def main():
    """Main entry point for the command line interface"""
    # Create argument parser
    parser = argparse.ArgumentParser(
        description='EU Procurement Monitoring System',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
  # Synchronize data from the last 30 days
  python ted_ml_pipeline.py sync --days 30

  # Train a model using default training data
  python ted_ml_pipeline.py train

  # Detect outliers in data from a specific date range
  python ted_ml_pipeline.py predict --start-date 20250101 --end-date 20250131 --visualize

  # Generate a report for the last 30 days
  python ted_ml_pipeline.py report --days 30
"""
    )

    # Create subparsers for commands
    subparsers = parser.add_subparsers(dest='command', help='Command to run')

    # Sync command
    sync_parser = subparsers.add_parser('sync', help='Synchronize data from TED API')
    sync_parser.add_argument('--days', type=int, default=30, help='Number of days to look back')
```

```
sync_parser.add_argument('--country', type=str, help='Country code filter (ISO)')
sync_parser.add_argument('--max-pages', type=int, default=5, help='Maximum number of pages to fetch')

# Train command
train_parser = subparsers.add_parser('train', help='Train outlier detection model')
train_parser.add_argument('--input', type=str, help='Path to training data file')
train_parser.add_argument('--sample', type=int, help='Sample size for training')
train_parser.add_argument('--contamination', type=float, default=0.05,
                          help='Expected proportion of outliers (0.0-0.5)')

# Predict command
predict_parser = subparsers.add_parser('predict', help='Detect outliers in procurement data')
predict_parser.add_argument('--input', type=str, help='Path to input data file')
predict_parser.add_argument('--start-date', type=str, help='Start date (YYYYMMDD)')
predict_parser.add_argument('--end-date', type=str, help='End date (YYYYMMDD)')
predict_parser.add_argument('--country', type=str, help='Country code filter (ISO)')
predict_parser.add_argument('--max-bid-amount', type=float, help='Maximum bid amount')
predict_parser.add_argument('--visualize', action='store_true', help='Generate visualizations')

# Evaluate command
evaluate_parser = subparsers.add_parser('evaluate', help='Evaluate model performance')
evaluate_parser.add_argument('--input', type=str, required=True, help='Path to evaluation data file')

# Report command
report_parser = subparsers.add_parser('report', help='Generate procurement analysis report')
report_parser.add_argument('--input', type=str, help='Path to input data file')
report_parser.add_argument('--days', type=int, default=30, help='Number of days to include in report')
report_parser.add_argument('--format', type=str, choices=['pdf', 'html', 'json'],
                           default='pdf', help='Report format')

# Parse arguments
args = parser.parse_args()

# Create pipeline instance
pipeline = TEDMLPipeline()

# Execute command
if args.command == 'sync':
    pipeline.sync_data(
        days_back=args.days,
        country=args.country,
        max_pages=args.max_pages
    )
elif args.command == 'train':
    pipeline.train(
        input_file=args.input,
        sample_size=args.sample,
        contamination=args.contamination
    )
elif args.command == 'predict':
    pipeline.predict(
        input_file=args.input,
```

```
            country=args.country,
            start_date=args.start_date,
            end_date=args.end_date,
            max_bid_amount=args.max_bid_amount,
            visualize=args.visualize
        )
    elif args.command == 'evaluate':
        pipeline.evaluate(
            input_file=args.input
        )
    elif args.command == 'report':
        pipeline.generate_report(
            data_file=args.input,
            days=args.days,
            output_format=args.format
        )
    else:
        parser.print_help()
        sys.exit(1)
if __name__ == "__main__":
    main()
```

FILE: clustering_pipeline.py

```
#!/usr/bin/env python3
"""
Example implementation of a ClusteringPipeline for the EU Procurement Monitoring System.
This extension adds clustering-based outlier detection to the system.
"""
import os
import sys
import numpy as np
import pandas as pd
from datetime import datetime
import logging
from sklearn.cluster import DBSCAN, KMeans
from sklearn.preprocessing import StandardScaler
from ted_ml_pipeline import TEDMLPipeline
from visualization.visualizer import TEDVisualizer
# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
class ClusteringPipeline(TEDMLPipeline):
    """
    Extension of TEDMLPipeline that uses clustering-based methods for outlier detection.
    Supports both DBSCAN and KMeans clustering algorithms.
    """
```

```
def __init__(self, base_dir="."):
    """Initialize the clustering pipeline"""
    super().__init__(base_dir)

    # Add clustering-specific directories
    self.directories["clusters"] = os.path.join(base_dir, "clusters")
    os.makedirs(self.directories["clusters"], exist_ok=True)

    # Initialize clustering parameters with defaults
    self.clustering_params = {
        "method": "dbscan",  # 'dbscan' or 'kmeans'
        "eps": 0.5,          # For DBSCAN: maximum distance between samples
        "min_samples": 5,    # For DBSCAN: number of samples in neighborhood
        "n_clusters": 5      # For KMeans: number of clusters
    }

def train_clustering_model(self, input_file=None, method="dbscan", **kwargs):
    """
    Train a clustering model for outlier detection

    Args:
        input_file (str): Optional path to training data file
        method (str): Clustering method ('dbscan' or 'kmeans')
        **kwargs: Additional parameters for the clustering algorithm
            - eps: DBSCAN epsilon parameter (default: 0.5)
            - min_samples: DBSCAN min_samples parameter (default: 5)
            - n_clusters: KMeans number of clusters (default: 5)

    Returns:
        dict: Dictionary with model information
    """
    logger.info(f"Training clustering model: method={method}")

    # Update clustering parameters
    self.clustering_params["method"] = method
    if "eps" in kwargs:
        self.clustering_params["eps"] = kwargs["eps"]
    if "min_samples" in kwargs:
        self.clustering_params["min_samples"] = kwargs["min_samples"]
    if "n_clusters" in kwargs:
        self.clustering_params["n_clusters"] = kwargs["n_clusters"]

    try:
        # Step 1: Determine training file to use
        if input_file:
            training_file = input_file
            print(f"Using specified training file: {training_file}")
        else:
            # Use default training file
            training_file = self.training_data_file
            print(f"Using default training file: {training_file}")
```

```
# Check if file exists
if not os.path.exists(training_file):
    print(f"Error: Training file {training_file} does not exist")
    return None

# Step 2: Load training data
print("Loading training data...")
try:
    df = pd.read_csv(training_file)
    print(f"Loaded {len(df)} records with {len(df.columns)} features")
except Exception as e:
    print(f"Error loading training data: {e}")
    return None

# Step 3: Select and prepare features for clustering
print("Preparing features for clustering...")

# Get numerical features only
num_cols = df.select_dtypes(include=['int64', 'float64']).columns.tolist()

# Remove ID columns and other irrelevant features
exclude_patterns = ['id', 'identifier', 'code', 'status', 'is_outlier']
feature_cols = [col for col in num_cols if not any(pattern in col.lower() for pattern in exclude_patterns)

# Ensure we have valid numerical data
X = df[feature_cols].fillna(0).values

# Scale the data
print("Scaling features...")
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 4: Train the clustering model
print(f"Training {method.upper()} clustering model...")
if method.lower() == "dbscan":
    # Train DBSCAN model
    eps = self.clustering_params["eps"]
    min_samples = self.clustering_params["min_samples"]

    print(f"Parameters: eps={eps}, min_samples={min_samples}")
    model = DBSCAN(eps=eps, min_samples=min_samples, n_jobs=-1)

elif method.lower() == "kmeans":
    # Train KMeans model
    n_clusters = self.clustering_params["n_clusters"]

    print(f"Parameters: n_clusters={n_clusters}")
    model = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)

else:
    print(f"Error: Unknown clustering method '{method}'")
    return None
```

```
# Fit the model
model.fit(X_scaled)

# Step 5: Process clustering results
labels = model.labels_

# For DBSCAN: -1 indicates outliers
if method.lower() == "dbscan":
    n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
    n_outliers = list(labels).count(-1)
    outlier_pct = n_outliers / len(labels) * 100

    print(f"DBSCAN results: {n_clusters} clusters, {n_outliers} outliers ({outlier_pct:.2f}%)")

    # Add cluster labels to the dataframe
    df["cluster"] = labels
    df["is_outlier"] = labels == -1

# For KMeans: find outliers as points far from their centroids
else:
    # Calculate distance to assigned centroid
    distances = np.min(
        np.sqrt(np.sum((X_scaled - model.cluster_centers_[labels.reshape(-1, 1)])**2, axis=2)),
        axis=1
    )

    # Define outliers as points with distance > threshold (95th percentile)
    threshold = np.percentile(distances, 95)
    outliers = distances > threshold
    n_outliers = sum(outliers)
    outlier_pct = n_outliers / len(distances) * 100

    print(f"KMeans results: {n_clusters} clusters, {n_outliers} outliers ({outlier_pct:.2f}%)")

    # Add cluster labels and outlier flags to the dataframe
    df["cluster"] = labels
    df["is_outlier"] = outliers
    df["distance"] = distances

# Also add numerical anomaly score (0-1 scale)
if method.lower() == "dbscan":
    # For DBSCAN: Points marked as outliers get max score, others based on nearest neighbors
    from sklearn.neighbors import NearestNeighbors

    # Calculate distance to k-nearest neighbors for non-outliers
    nbrs = NearestNeighbors(n_neighbors=min_samples).fit(X_scaled)
    distances, _ = nbrs.kneighbors(X_scaled)

    # Average distance to k nearest neighbors as score
    avg_distances = np.mean(distances, axis=1)
    max_dist = np.max(avg_distances)
```

```
            # Normalize scores to 0-1 range
            scores = avg_distances / max_dist

            # Ensure outliers have highest scores
            scores[labels == -1] = 1.0

        else:
            # For KMeans: Normalize distances to 0-1 range
            scores = distances / np.max(distances)

        # Add anomaly scores to dataframe
        df["anomaly_score"] = scores

        # Step 6: Save results
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        results_file = os.path.join(self.directories["clusters"], f"cluster_results_{timestamp}.csv")

        print(f"Saving clustering results to {results_file}")
        df.to_csv(results_file, index=False)

        # Save model information
        model_info = {
            "method": method,
            "parameters": self.clustering_params,
            "feature_columns": feature_cols,
            "scaler": scaler,
            "model": model,
            "results_file": results_file,
            "timestamp": timestamp
        }

        print("Clustering model training completed successfully")

        # Generate visualizations if requested
        if kwargs.get("visualize", False):
            print("Generating visualizations...")

            visualizer = TEDVisualizer(output_dir=self.directories["visualizations"])
            viz_files = visualizer.create_visualizations(
                df,
                base_filename=f"clustering_{method}_{timestamp}"
            )

            if viz_files:
                print(f"Generated {len(viz_files)} visualizations")
                model_info["visualizations"] = viz_files

        return model_info

    except Exception as e:
        logger.error(f"Error during clustering model training: {e}")
```

```
        print(f"Error: {str(e)}")
        return None

def predict_with_clustering(self, input_file=None, method="dbscan", visualize=False, **kwargs):
    """
    Detect outliers using a clustering approach

    Args:
        input_file (str): Path to input data file
        method (str): Clustering method ('dbscan' or 'kmeans')
        visualize (bool): Whether to generate visualizations
        **kwargs: Additional parameters for the clustering algorithm

    Returns:
        dict: Dictionary with prediction results
    """
    logger.info(f"Detecting outliers with clustering: method={method}")

    # Update parameters for this run only (don't change stored parameters)
    clustering_params = dict(self.clustering_params)
    if "eps" in kwargs:
        clustering_params["eps"] = kwargs["eps"]
    if "min_samples" in kwargs:
        clustering_params["min_samples"] = kwargs["min_samples"]
    if "n_clusters" in kwargs:
        clustering_params["n_clusters"] = kwargs["n_clusters"]

    try:
        # Step 1: Load the dataset
        if not input_file or not os.path.exists(input_file):
            print(f"Error: Input file {input_file} does not exist")
            return None

        print(f"Loading dataset: {input_file}")
        try:
            df = pd.read_csv(input_file)
            print(f"Loaded {len(df)} records with {len(df.columns)} features")
        except Exception as e:
            print(f"Error loading dataset: {e}")
            return None

        # Step 2: Select and prepare features
        print("Preparing features for clustering...")

        # Get numerical features only
        num_cols = df.select_dtypes(include=['int64', 'float64']).columns.tolist()

        # Remove ID columns and other irrelevant features
        exclude_patterns = ['id', 'identifier', 'code', 'status', 'is_outlier']
        feature_cols = [col for col in num_cols if not any(pattern in col.lower() for pattern in exclude_patterns)

        # Ensure we have valid numerical data
```

```
X = df[feature_cols].fillna(0).values

# Scale the data
print("Scaling features...")
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Step 3: Apply clustering algorithm
print(f"Applying {method.upper()} clustering...")
if method.lower() == "dbscan":
    # Apply DBSCAN
    eps = clustering_params["eps"]
    min_samples = clustering_params["min_samples"]

    print(f"Parameters: eps={eps}, min_samples={min_samples}")
    model = DBSCAN(eps=eps, min_samples=min_samples, n_jobs=-1)

elif method.lower() == "kmeans":
    # Apply KMeans
    n_clusters = clustering_params["n_clusters"]

    print(f"Parameters: n_clusters={n_clusters}")
    model = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)

else:
    print(f"Error: Unknown clustering method '{method}'")
    return None

# Fit the model
model.fit(X_scaled)

# Step 4: Process clustering results
labels = model.labels_

# For DBSCAN: -1 indicates outliers
if method.lower() == "dbscan":
    n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
    n_outliers = list(labels).count(-1)
    outlier_pct = n_outliers / len(labels) * 100

    print(f"DBSCAN results: {n_clusters} clusters, {n_outliers} outliers ({outlier_pct:.2f}%)")

    # Add cluster labels to the dataframe
    df["cluster"] = labels
    df["is_outlier"] = labels == -1

# For KMeans: find outliers as points far from their centroids
else:
    # Calculate distance to assigned centroid
    distances = np.min(
        np.sqrt(np.sum((X_scaled - model.cluster_centers_[labels.reshape(-1, 1)])**2, axis=2)),
        axis=1
```

```
        )

        # Define outliers as points with distance > threshold (95th percentile)
        threshold = np.percentile(distances, 95)
        outliers = distances > threshold
        n_outliers = sum(outliers)
        outlier_pct = n_outliers / len(distances) * 100

        print(f"KMeans results: {n_clusters} clusters, {n_outliers} outliers ({outlier_pct:.2f}%)")

        # Add cluster labels and outlier flags to the dataframe
        df["cluster"] = labels
        df["is_outlier"] = outliers
        df["distance"] = distances

    # Also add numerical anomaly score (0-1 scale)
    if method.lower() == "dbscan":
        # For DBSCAN: Points marked as outliers get max score, others based on nearest neighbors
        from sklearn.neighbors import NearestNeighbors

        # Calculate distance to k-nearest neighbors for non-outliers
        nbrs = NearestNeighbors(n_neighbors=min_samples).fit(X_scaled)
        distances, _ = nbrs.kneighbors(X_scaled)

        # Average distance to k nearest neighbors as score
        avg_distances = np.mean(distances, axis=1)
        max_dist = np.max(avg_distances)

        # Normalize scores to 0-1 range
        scores = avg_distances / max_dist

        # Ensure outliers have highest scores
        scores[labels == -1] = 1.0

    else:
        # For KMeans: Normalize distances to 0-1 range
        scores = distances / np.max(distances)

    # Add anomaly scores to dataframe
    df["anomaly_score"] = scores

    # Step 5: Save results
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    results_file = os.path.join(self.directories["output"], f"cluster_outliers_{timestamp}.csv")

    print(f"Saving results to {results_file}")
    df.to_csv(results_file, index=False)

    # Step 6: Generate visualizations if requested
    viz_files = []
    if visualize:
        print("Generating visualizations...")
```

```
                visualizer = TEDVisualizer(output_dir=self.directories["visualizations"])
                viz_files = visualizer.create_visualizations(
                    df,
                    base_filename=f"clustering_{method}_{timestamp}"
                )

                if viz_files:
                    print(f"Generated {len(viz_files)} visualizations")

            # Prepare result information
            result = {
                "method": method,
                "parameters": clustering_params,
                "results_file": results_file,
                "record_count": len(df),
                "outlier_count": n_outliers,
                "outlier_percentage": outlier_pct,
                "visualizations": viz_files,
                "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            }

            print(f"Clustering-based outlier detection completed successfully")
            return result

        except Exception as e:
            logger.error(f"Error during clustering-based outlier detection: {e}")
            print(f"Error: {str(e)}")
            return None
# Example usage when run directly
if __name__ == "__main__":
    import argparse

    # Parse arguments
    parser = argparse.ArgumentParser(description='EU Procurement Monitoring - Clustering Analysis')

    # Add subparsers for different commands
    subparsers = parser.add_subparsers(dest='command', help='Command to run')

    # Train command
    train_parser = subparsers.add_parser('train', help='Train a clustering model')
    train_parser.add_argument('--input', type=str, help='Path to input CSV file')
    train_parser.add_argument('--method', type=str, choices=['dbscan', 'kmeans'],
                              default='dbscan', help='Clustering method')
    train_parser.add_argument('--eps', type=float, default=0.5,
                              help='DBSCAN: distance threshold')
    train_parser.add_argument('--min-samples', type=int, default=5,
                              help='DBSCAN: minimum samples in neighborhood')
    train_parser.add_argument('--n-clusters', type=int, default=5,
                              help='KMeans: number of clusters')
    train_parser.add_argument('--visualize', action='store_true',
                              help='Generate visualizations')
```

```
# Predict command
predict_parser = subparsers.add_parser('predict', help='Detect outliers using clustering')
predict_parser.add_argument('--input', type=str, required=True,
                            help='Path to input CSV file')
predict_parser.add_argument('--method', type=str, choices=['dbscan', 'kmeans'],
                            default='dbscan', help='Clustering method')
predict_parser.add_argument('--eps', type=float, default=0.5,
                            help='DBSCAN: distance threshold')
predict_parser.add_argument('--min-samples', type=int, default=5,
                            help='DBSCAN: minimum samples in neighborhood')
predict_parser.add_argument('--n-clusters', type=int, default=5,
                            help='KMeans: number of clusters')
predict_parser.add_argument('--visualize', action='store_true',
                            help='Generate visualizations')

# Parse arguments
args = parser.parse_args()

# Create pipeline instance
pipeline = ClusteringPipeline()

# Execute command
if args.command == 'train':
    if args.method == 'dbscan':
        pipeline.train_clustering_model(
            input_file=args.input,
            method=args.method,
            eps=args.eps,
            min_samples=args.min_samples,
            visualize=args.visualize
        )
    else:  # kmeans
        pipeline.train_clustering_model(
            input_file=args.input,
            method=args.method,
            n_clusters=args.n_clusters,
            visualize=args.visualize
        )
elif args.command == 'predict':
    if args.method == 'dbscan':
        pipeline.predict_with_clustering(
            input_file=args.input,
            method=args.method,
            eps=args.eps,
            min_samples=args.min_samples,
            visualize=args.visualize
        )
    else:  # kmeans
        pipeline.predict_with_clustering(
            input_file=args.input,
            method=args.method,
```

```
                n_clusters=args.n_clusters,
                visualize=args.visualize
            )
    else:
        parser.print_help()
        sys.exit(1)
```

FILE: script.py

```
#!/usr/bin/env python3
import os
import sys
import argparse
from fpdf import FPDF
import mimetypes
class SimpleFileExporter:
    """
    Simple PDF exporter that lists file routes and their content sequentially
    Optimized for SvelteKit and React JSX files
    """

    def __init__(self, start_dir='.', output_file='output.pdf', max_file_size=1048576):
        """
        Initialize the exporter

        Args:
            start_dir (str): Directory to start scanning from
            output_file (str): Output PDF filename
            max_file_size (int): Maximum file size in bytes to include
        """
        self.start_dir = os.path.abspath(start_dir)
        self.output_file = output_file
        self.max_file_size = max_file_size
        self.processed_files = 0

        # Initialize PDF
        self.pdf = FPDF()
        self.pdf.set_auto_page_break(True, margin=15)
        self.pdf.set_font('Arial', '', 10)
        self.pdf.add_page()

        # Add title
        self.pdf.set_font('Arial', 'B', 16)
        self.pdf.cell(0, 10, 'Project Files Content', 0, 1, 'C')
        self.pdf.ln(5)

    def is_text_file(self, filepath):
        """Check if file is a text file that should be included"""
        # List of extensions to include
        code_extensions = {
            # Svelte/React
```

```
            '.svelte', '.jsx', '.tsx', '.js', '.ts',
            # Web
            '.html', '.css', '.json', '.md',
            # Config
            '.config.js', '.config.ts', '.json', '.yaml', '.yml',
            # Other common code files
            '.py', '.php', '.rb', '.go', '.java', '.c', '.cpp', '.cs'
        }

        ext = os.path.splitext(filepath)[1].lower()
        # Special handling for config files
        if filepath.endswith('.config.js') or filepath.endswith('.config.ts'):
            return True

        # Check extension
        if ext in code_extensions:
            return True

        # Check mime type as fallback
        mime_type, _ = mimetypes.guess_type(filepath)
        if mime_type and mime_type.startswith('text/'):
            return True

        return False

    def process_folder(self, folder_path):
        """
        Process all files in a folder recursively

        Args:
            folder_path (str): Path to the folder
        """
        try:
            for root, dirs, files in os.walk(folder_path):
                # Skip hidden folders, node_modules, and .svelte-kit
                dirs[:] = [d for d in dirs if not d.startswith('.') and d != 'node_modules' and d != '.svelte-kit']

                for file in files:
                    # Skip hidden files and package-lock.json
                    if file.startswith('.') or file == 'package-lock.json':
                        continue

                    file_path = os.path.join(root, file)
                    rel_path = os.path.relpath(file_path, self.start_dir)

                    # Skip large files
                    if os.path.getsize(file_path) > self.max_file_size:
                        continue

                    # Process text files
                    if self.is_text_file(file_path):
                        self.add_file_content(file_path, rel_path)
```

```
                    self.processed_files += 1

        except Exception as e:
            print(f"Error processing folder {folder_path}: {str(e)}")

    def add_file_content(self, file_path, rel_path):
        """
        Add file route and content to PDF

        Args:
            file_path (str): Path to the file
            rel_path (str): Relative path from start directory
        """
        try:
            # Try different encodings to read the file
            content = None
            for encoding in ['utf-8', 'latin-1', 'cp1252']:
                try:
                    with open(file_path, 'r', encoding=encoding) as f:
                        content = f.read()
                    break
                except UnicodeDecodeError:
                    continue

            if content is None:
                print(f"Warning: Could not decode file {rel_path}")
                return

            # Clean content of any non-ASCII characters
            clean_content = ''.join(c if ord(c) < 128 else '_' for c in content)
            clean_path = ''.join(c if ord(c) < 128 else '_' for c in rel_path)

            # Add file header - ensure we have enough space
            if self.pdf.get_y() > 250:
                self.pdf.add_page()

            # Route header with background
            self.pdf.set_font('Arial', 'B', 12)
            self.pdf.set_fill_color(220, 220, 220)
            self.pdf.multi_cell(0, 10, f'FILE: {clean_path}', 1, 'L', True)

            # Content
            self.pdf.set_font('Courier', '', 8)

            # Split content into lines and add to PDF
            lines = clean_content.split('\n')
            for line in lines:
                current_y = self.pdf.get_y()
                if current_y > 270:  # Check if near bottom of page
                    self.pdf.add_page()

                # Wrap long lines
```

```
                while len(line) > 0:
                    line_width = min(120, len(line))
                    self.pdf.cell(0, 5, line[:line_width], 0, 1)
                    line = line[line_width:]

            # Add separator
            self.pdf.ln(5)
            self.pdf.cell(0, 0, '', 'T', 1)
            self.pdf.ln(5)

        except Exception as e:
            print(f"Error processing file {rel_path}: {str(e)}")

    def generate(self):
        """Generate the PDF file"""
        print(f"Scanning folder: {self.start_dir}")
        self.process_folder(self.start_dir)

        # Add summary at the end
        self.pdf.add_page()
        self.pdf.set_font('Arial', 'B', 14)
        self.pdf.cell(0, 10, 'Summary', 0, 1, 'C')
        self.pdf.set_font('Arial', '', 12)
        self.pdf.cell(0, 10, f'Files processed: {self.processed_files}', 0, 1)

        # Save PDF
        self.pdf.output(self.output_file)
        print(f"PDF generated: {os.path.abspath(self.output_file)}")
        print(f"Processed {self.processed_files} files.")
def main():
    """Main function to run the script"""
    parser = argparse.ArgumentParser(description='Generate a PDF with file routes and their content')
    parser.add_argument('-d', '--directory', default='.',
                        help='Directory to scan (default: current directory)')
    parser.add_argument('-o', '--output', default='output.pdf',
                        help='Output PDF filename (default: output.pdf)')
    parser.add_argument('-m', '--max-size', type=int, default=1048576,
                        help='Maximum file size in bytes (default: 1MB)')
    args = parser.parse_args()

    try:
        exporter = SimpleFileExporter(args.directory, args.output, args.max_size)
        exporter.generate()
    except Exception as e:
        print(f"Error: {str(e)}")
        return 1

    return 0
if __name__ == "__main__":
    sys.exit(main())
```

FILE: visualization/__init__.py

FILE: visualization/visualizer.py

```
#!/usr/bin/env python3
"""
TED Data Visualization Module
This module provides simple, straightforward visualization capabilities for EU procurement data.
It creates clear charts for understanding outlier detection results.
"""
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime
import logging
# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)
class TEDVisualizer:
    """Class for creating visualizations of TED procurement data analysis"""

    def __init__(self, output_dir="visualizations", dpi=100, figsize=(10, 6)):
        """
        Initialize the visualizer

        Args:
            output_dir (str): Directory for saving visualizations
            dpi (int): Resolution for saved images
            figsize (tuple): Default figure size (width, height)
        """
        self.output_dir = output_dir
        self.dpi = dpi
        self.figsize = figsize

        # Ensure output directory exists
        os.makedirs(output_dir, exist_ok=True)

        # Set visualization style
        sns.set_style("whitegrid")

        # Define colors for consistent look
        self.colors = {
            "normal": "#1f77b4",  # Blue
            "outlier": "#d62728"   # Red
        }
```

```
    # Set basic pyplot parameters
    plt.rcParams.update({
        'font.size': 10,
        'axes.titlesize': 14,
        'axes.labelsize': 12
    })

def _format_currency(self, x, pos):
    """Format values as Euro currency with appropriate scale"""
    if x >= 1e9:
        return '_{:.1f}B'.format(x / 1e9)
    elif x >= 1e6:
        return '_{:.1f}M'.format(x / 1e6)
    elif x >= 1e3:
        return '_{:.1f}K'.format(x / 1e3)
    else:
        return '_{:.0f}'.format(x)

def create_visualizations(self, df, base_filename="ted_analysis"):
    """
    Create a set of visualizations for procurement data analysis

    Args:
        df (pd.DataFrame): DataFrame with outlier predictions
        base_filename (str): Base name for output files

    Returns:
        list: Paths to saved visualization files
    """
    if df.empty:
        logger.warning("Empty dataframe provided, cannot create visualizations")
        return []

    saved_files = []

    try:
        # 1. Value Distribution
        value_path = self.plot_value_distribution(df, f"{base_filename}_values.png")
        if value_path:
            saved_files.append(value_path)

        # 2. Outlier Analysis
        if 'is_outlier' in df.columns:
            outlier_path = self.plot_outlier_analysis(df, f"{base_filename}_outliers.png")
            if outlier_path:
                saved_files.append(outlier_path)

        # 3. Country Distribution
        country_field = None
        for col in ['organisation-country-buyer', 'country']:
            if col in df.columns:
```

```
                country_field = col
                break

        if country_field:
            country_path = self.plot_country_distribution(df, country_field, f"{base_filename}_countries.png")
            if country_path:
                saved_files.append(country_path)

        # 4. Summary Report
        report_path = self.create_summary_report(df, f"{base_filename}_report.png")
        if report_path:
            saved_files.append(report_path)

        logger.info(f"Created {len(saved_files)} visualizations")
        return saved_files

    except Exception as e:
        logger.error(f"Error creating visualizations: {e}")
        return saved_files

def plot_value_distribution(self, df, filename):
    """
    Create visualization of procurement value distribution

    Args:
        df (pd.DataFrame): DataFrame with procurement data
        filename (str): Output filename

    Returns:
        str: Path to saved visualization
    """
    try:
        # Find value column
        value_col = None
        for col in ['total-value-eur', 'total-value-eur-capped', 'value_eur', 'total-value']:
            if col in df.columns:
                value_col = col
                break

        if not value_col:
            logger.warning("No value column found for value distribution plot")
            return None

        # Create figure with two subplots
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=self.figsize)

        # Filter to valid values only
        plot_df = df[df[value_col] > 0].copy()

        # Histogram with log scale
        if 'is_outlier' in df.columns:
            # Separate histograms for normal and outlier values
```

```
        sns.histplot(
            data=plot_df[~plot_df['is_outlier']],
            x=value_col,
            bins=20,
            log_scale=True,
            color=self.colors['normal'],
            alpha=0.7,
            label='Normal',
            ax=ax1
        )

        sns.histplot(
            data=plot_df[plot_df['is_outlier']],
            x=value_col,
            bins=20,
            log_scale=True,
            color=self.colors['outlier'],
            alpha=0.7,
            label='Outlier',
            ax=ax1
        )

        ax1.legend()
    else:
        # Single histogram
        sns.histplot(
            data=plot_df,
            x=value_col,
            bins=30,
            log_scale=True,
            color=self.colors['normal'],
            ax=ax1
        )

    ax1.set_title('Value Distribution (Log Scale)')
    ax1.set_xlabel('Value (EUR)')
    ax1.set_ylabel('Count')

    # Boxplot of values
    if 'is_outlier' in df.columns:
        # Create boxplot by outlier status
        plot_df['Status'] = plot_df['is_outlier'].map({True: 'Outlier', False: 'Normal'})
        sns.boxplot(
            x='Status',
            y=value_col,
            data=plot_df,
            palette={
                'Normal': self.colors['normal'],
                'Outlier': self.colors['outlier']
            },
            ax=ax2
        )
```

```
        else:
            # Single boxplot
            sns.boxplot(
                y=value_col,
                data=plot_df,
                color=self.colors['normal'],
                ax=ax2
            )

        ax2.set_title('Value Range')
        ax2.set_ylabel('Value (EUR)')
        ax2.set_yscale('log')

        # Format y-axis as currency
        for axis in [ax1.xaxis, ax2.yaxis]:
            axis.set_major_formatter(plt.FuncFormatter(self._format_currency))

        plt.suptitle('Procurement Value Analysis', fontsize=16)
        plt.tight_layout()

        # Save figure
        output_path = os.path.join(self.output_dir, filename)
        fig.savefig(output_path, dpi=self.dpi, bbox_inches='tight')
        plt.close(fig)

        logger.info(f"Saved value distribution plot to {output_path}")
        return output_path

    except Exception as e:
        logger.error(f"Error creating value distribution plot: {e}")
        return None

def plot_outlier_analysis(self, df, filename):
    """
    Create visualization focused on outlier analysis

    Args:
        df (pd.DataFrame): DataFrame with outlier predictions
        filename (str): Output filename

    Returns:
        str: Path to saved visualization
    """
    try:
        if 'is_outlier' not in df.columns:
            logger.warning("No is_outlier column found for outlier analysis plot")
            return None

        # Create figure with two subplots
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=self.figsize)

        # 1. Pie chart of outlier distribution
```

```
outlier_count = df['is_outlier'].sum()
normal_count = len(df) - outlier_count

ax1.pie(
    [normal_count, outlier_count],
    labels=['Normal', 'Outlier'],
    autopct='%1.1f%%',
    colors=[self.colors['normal'], self.colors['outlier']],
    startangle=90,
    explode=(0, 0.1)
)

ax1.set_title('Outlier Distribution')

# 2. Anomaly score plot if available
if 'anomaly_score' in df.columns:
    # Find value column for scatter plot
    value_col = None
    for col in ['total-value-eur', 'total-value-eur-capped', 'value_eur', 'total-value']:
        if col in df.columns:
            value_col = col
            break

    if value_col:
        # Scatter plot of value vs anomaly score
        ax2.scatter(
            df[~df['is_outlier']]['anomaly_score'],
            df[~df['is_outlier']][value_col],
            color=self.colors['normal'],
            alpha=0.6,
            label='Normal',
            s=20
        )

        ax2.scatter(
            df[df['is_outlier']]['anomaly_score'],
            df[df['is_outlier']][value_col],
            color=self.colors['outlier'],
            alpha=0.8,
            label='Outlier',
            s=40,
            marker='x'
        )

        ax2.set_title('Value vs Anomaly Score')
        ax2.set_xlabel('Anomaly Score')
        ax2.set_ylabel('Value (EUR)')
        ax2.set_yscale('log')
        ax2.legend()

        # Format y-axis as currency
        ax2.yaxis.set_major_formatter(plt.FuncFormatter(self._format_currency))
```

```
            # Add threshold line if possible
            if df['is_outlier'].sum() > 0:
                threshold = df[df['is_outlier']]['anomaly_score'].min()
                ax2.axvline(x=threshold, color='red', linestyle='--', label=f'Threshold: {threshold:.3f}')
                ax2.legend()
        else:
            # Simple histogram of anomaly scores
            sns.histplot(
                data=df,
                x='anomaly_score',
                hue='is_outlier',
                palette=[self.colors['normal'], self.colors['outlier']],
                bins=20,
                ax=ax2
            )

            ax2.set_title('Anomaly Score Distribution')
            ax2.set_xlabel('Anomaly Score')
            ax2.set_ylabel('Count')
    else:
        # Show stats if no anomaly score
        ax2.axis('off')

        stats_text = [
            f"Total Records: {len(df):,}",
            f"Normal Records: {normal_count:,} ({normal_count/len(df)*100:.1f}%)",
            f"Outlier Records: {outlier_count:,} ({outlier_count/len(df)*100:.1f}%)"
        ]

        # Add value stats if available
        value_col = None
        for col in ['total-value-eur', 'total-value-eur-capped', 'value_eur', 'total-value']:
            if col in df.columns:
                value_col = col
                break

        if value_col:
            normal_avg = df[~df['is_outlier']][value_col].mean()
            outlier_avg = df[df['is_outlier']][value_col].mean() if outlier_count > 0 else 0

            stats_text.extend([
                f"",
                f"Avg Normal Value: {self._format_currency(normal_avg, 0)}",
                f"Avg Outlier Value: {self._format_currency(outlier_avg, 0)}",
                f"Outlier/Normal Ratio: {outlier_avg/normal_avg:.1f}x" if normal_avg > 0 else ""
            ])

        ax2.text(
            0.5, 0.5,
            '\n'.join(stats_text),
            ha='center',
```

```
                va='center',
                transform=ax2.transAxes,
                bbox=dict(boxstyle='round', facecolor='#f9f9f9', alpha=0.5)
            )

        plt.suptitle('Procurement Outlier Analysis', fontsize=16)
        plt.tight_layout()

        # Save figure
        output_path = os.path.join(self.output_dir, filename)
        fig.savefig(output_path, dpi=self.dpi, bbox_inches='tight')
        plt.close(fig)

        logger.info(f"Saved outlier analysis plot to {output_path}")
        return output_path

    except Exception as e:
        logger.error(f"Error creating outlier analysis plot: {e}")
        return None

def plot_country_distribution(self, df, country_field, filename):
    """
    Create visualization of procurement data by country

    Args:
        df (pd.DataFrame): DataFrame with procurement data
        country_field (str): Name of column containing country data
        filename (str): Output filename

    Returns:
        str: Path to saved visualization
    """
    try:
        if country_field not in df.columns:
            logger.warning(f"Column {country_field} not found for country distribution plot")
            return None

        # Get top countries by count
        top_countries = df[country_field].value_counts().head(10)

        # Create figure with two subplots
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=self.figsize)

        # 1. Count by country
        sns.barplot(
            x=top_countries.index,
            y=top_countries.values,
            palette='viridis',
            ax=ax1
        )

        ax1.set_title('Record Count by Country')
```

```
ax1.set_xlabel('')
ax1.set_ylabel('Number of Records')
plt.setp(ax1.get_xticklabels(), rotation=45, ha='right')

# 2. Outlier percentage by country
if 'is_outlier' in df.columns:
    # Calculate percentage of outliers by country
    country_data = df.groupby(country_field)['is_outlier'].agg(['count', 'sum'])
    country_data['percentage'] = country_data['sum'] / country_data['count'] * 100

    # Sort by count and get top countries
    country_data = country_data.sort_values('count', ascending=False).head(10)

    sns.barplot(
        x=country_data.index,
        y=country_data['percentage'],
        palette='rocket_r',
        ax=ax2
    )

    # Add horizontal line for overall percentage
    overall_pct = df['is_outlier'].mean() * 100
    ax2.axhline(
        y=overall_pct,
        color='red',
        linestyle='--',
        label=f'Overall: {overall_pct:.1f}%'
    )

    ax2.set_title('Outlier Percentage by Country')
    ax2.set_xlabel('')
    ax2.set_ylabel('Outlier Percentage (%)')
    ax2.legend()
    plt.setp(ax2.get_xticklabels(), rotation=45, ha='right')
else:
    # Value distribution by country
    value_col = None
    for col in ['total-value-eur', 'total-value-eur-capped', 'value_eur', 'total-value']:
        if col in df.columns:
            value_col = col
            break

    if value_col:
        # Calculate average value by country
        country_values = df.groupby(country_field)[value_col].mean().sort_values(ascending=False).head(10)

        sns.barplot(
            x=country_values.index,
            y=country_values.values,
            palette='viridis',
            ax=ax2
        )
```

```
                ax2.set_title('Average Value by Country')
                ax2.set_xlabel('')
                ax2.set_ylabel('Average Value (EUR)')
                plt.setp(ax2.get_xticklabels(), rotation=45, ha='right')

                # Format y-axis as currency
                ax2.yaxis.set_major_formatter(plt.FuncFormatter(self._format_currency))
            else:
                ax2.set_title('No value data available')
                ax2.axis('off')

        plt.suptitle('Procurement Analysis by Country', fontsize=16)
        plt.tight_layout()

        # Save figure
        output_path = os.path.join(self.output_dir, filename)
        fig.savefig(output_path, dpi=self.dpi, bbox_inches='tight')
        plt.close(fig)

        logger.info(f"Saved country distribution plot to {output_path}")
        return output_path

    except Exception as e:
        logger.error(f"Error creating country distribution plot: {e}")
        return None

def create_summary_report(self, df, filename):
    """
    Create a summary report visualization

    Args:
        df (pd.DataFrame): DataFrame with procurement data
        filename (str): Output filename

    Returns:
        str: Path to saved visualization
    """
    try:
        # Create figure with 2x2 grid
        fig, axes = plt.subplots(2, 2, figsize=(12, 10))

        # Find value column
        value_col = None
        for col in ['total-value-eur', 'total-value-eur-capped', 'value_eur', 'total-value']:
            if col in df.columns:
                value_col = col
                break

        # 1. Value histogram (top-left)
        ax1 = axes[0, 0]
```

```
if value_col:
    # Filter to valid values
    plot_df = df[df[value_col] > 0].copy()

    if 'is_outlier' in df.columns:
        # Separate histograms for normal and outlier values
        sns.histplot(
            data=plot_df[~plot_df['is_outlier']],
            x=value_col,
            bins=20,
            log_scale=True,
            color=self.colors['normal'],
            alpha=0.7,
            label='Normal',
            ax=ax1
        )

        sns.histplot(
            data=plot_df[plot_df['is_outlier']],
            x=value_col,
            bins=20,
            log_scale=True,
            color=self.colors['outlier'],
            alpha=0.7,
            label='Outlier',
            ax=ax1
        )

        ax1.legend()
    else:
        # Single histogram
        sns.histplot(
            data=plot_df,
            x=value_col,
            bins=30,
            log_scale=True,
            color=self.colors['normal'],
            ax=ax1
        )

    ax1.set_title('Value Distribution')
    ax1.set_xlabel('Value (EUR, log scale)')
    ax1.set_ylabel('Count')

    # Format x-axis as currency
    ax1.xaxis.set_major_formatter(plt.FuncFormatter(self._format_currency))
else:
    ax1.set_title('No value data available')
    ax1.axis('off')

# 2. Country bar chart (top-right)
ax2 = axes[0, 1]
```

```
# Find country field
country_field = None
for col in ['organisation-country-buyer', 'country']:
    if col in df.columns:
        country_field = col
        break

if country_field:
    # Get top countries by count
    top_countries = df[country_field].value_counts().head(10)

    sns.barplot(
        x=top_countries.index,
        y=top_countries.values,
        palette='viridis',
        ax=ax2
    )

    ax2.set_title('Top Countries')
    ax2.set_xlabel('')
    ax2.set_ylabel('Number of Records')
    plt.setp(ax2.get_xticklabels(), rotation=45, ha='right')
else:
    ax2.set_title('No country data available')
    ax2.axis('off')

# 3. Outlier analysis (bottom-left)
ax3 = axes[1, 0]

if 'is_outlier' in df.columns:
    # Pie chart of outlier distribution
    outlier_count = df['is_outlier'].sum()
    normal_count = len(df) - outlier_count

    ax3.pie(
        [normal_count, outlier_count],
        labels=['Normal', 'Outlier'],
        autopct='%1.1f%%',
        colors=[self.colors['normal'], self.colors['outlier']],
        startangle=90,
        explode=(0, 0.1)
    )

    ax3.set_title('Outlier Distribution')
else:
    ax3.set_title('No outlier data available')
    ax3.axis('off')

# 4. Summary statistics text (bottom-right)
ax4 = axes[1, 1]
ax4.axis('off')
```

```
# Prepare statistics text
stats = [
    f"SUMMARY STATISTICS",
    f"",
    f"Total Records: {len(df):,}"
]

if 'is_outlier' in df.columns:
    outlier_count = df['is_outlier'].sum()
    outlier_pct = outlier_count / len(df) * 100
    stats.extend([
        f"Outliers: {outlier_count:,} ({outlier_pct:.1f}%)",
        f"Normal Records: {len(df) - outlier_count:,} ({100 - outlier_pct:.1f}%)"
    ])

if value_col:
    valid_values = df[df[value_col] > 0][value_col]
    stats.extend([
        f"",
        f"Value Statistics:",
        f"Minimum: {self._format_currency(valid_values.min(), 0)}",
        f"Maximum: {self._format_currency(valid_values.max(), 0)}",
        f"Average: {self._format_currency(valid_values.mean(), 0)}",
        f"Median: {self._format_currency(valid_values.median(), 0)}"
    ])

    if 'is_outlier' in df.columns and outlier_count > 0:
        normal_avg = df[~df['is_outlier']][value_col].mean()
        outlier_avg = df[df['is_outlier']][value_col].mean()
        stats.extend([
            f"",
            f"Average Normal Value: {self._format_currency(normal_avg, 0)}",
            f"Average Outlier Value: {self._format_currency(outlier_avg, 0)}",
            f"Outlier/Normal Ratio: {outlier_avg/normal_avg:.1f}x"
        ])

# Add timestamp
stats.extend([
    f"",
    f"Report Generated: {datetime.now().strftime('%Y-%m-%d %H:%M')}"
])

# Display statistics
ax4.text(
    0.5, 0.5,
    '\n'.join(stats),
    ha='center',
    va='center',
    transform=ax4.transAxes,
    bbox=dict(boxstyle='round', facecolor='#f9f9f9', alpha=0.5)
)
```

```
            plt.suptitle('EU Procurement Analysis Summary', fontsize=16)
            plt.tight_layout(rect=[0, 0, 1, 0.97])  # Make room for suptitle

            # Save figure
            output_path = os.path.join(self.output_dir, filename)
            fig.savefig(output_path, dpi=self.dpi, bbox_inches='tight')
            plt.close(fig)

            logger.info(f"Saved summary report to {output_path}")
            return output_path

        except Exception as e:
            logger.error(f"Error creating summary report: {e}")
            return None
# If run directly, perform a simple test
if __name__ == "__main__":
    import argparse

    # Parse arguments
    parser = argparse.ArgumentParser(description='TED Data Visualization')
    parser.add_argument('--input', '-i', type=str, required=True,
                        help='Path to input CSV file with procurement data')
    parser.add_argument('--output', '-o', type=str, default='visualizations',
                        help='Output directory for visualizations')
    parser.add_argument('--name', '-n', type=str, default='ted_analysis',
                        help='Base name for output files')

    args = parser.parse_args()

    # Check if input file exists
    if not os.path.exists(args.input):
        print(f"Error: Input file {args.input} does not exist")
        exit(1)

    # Create visualizer
    visualizer = TEDVisualizer(output_dir=args.output)

    # Load data
    try:
        df = pd.read_csv(args.input)
        print(f"Loaded {len(df)} records from {args.input}")
    except Exception as e:
        print(f"Error loading data: {e}")
        exit(1)

    # Create visualizations
    viz_files = visualizer.create_visualizations(df, args.name)

    # Print results
    if viz_files:
        print(f"\nCreated {len(viz_files)} visualizations:")
```

```
        for file in viz_files:
            print(f" - {file}")
    else:
        print("\nNo visualizations were created")
```

FILE: transforming/requirements.txt

```
# EU Procurement Monitoring System Requirements
# Core dependencies
pandas>=1.5.0
numpy>=1.23.0
scikit-learn>=1.2.0
matplotlib>=3.7.0
seaborn>=0.12.0
requests>=2.28.0
# Data handling
python-dateutil>=2.8.2
pytz>=2023.3
# Database
SQLAlchemy>=2.0.0  # Optional: For more advanced database operations
# Visualization
plotly>=5.13.0  # Optional: For interactive visualizations
# Testing
pytest>=7.3.1  # Optional: For unit testing
# Documentation
sphinx>=6.1.3  # Optional: For generating documentation
# Code quality
black>=23.3.0  # Optional: For code formatting
flake8>=6.0.0  # Optional: For code linting
```

FILE: transforming/clustering_model.py

```
#!/usr/bin/env python3
"""
Clustering Model Module for TED Procurement Outlier Detection
This module provides an alternative approach to outlier detection using clustering techniques.
It handles training, prediction, model serialization, and evaluation.
Author: Your Name
Date: May 21, 2025
"""
import os
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
from datetime import datetime
from sklearn.cluster import DBSCAN, KMeans
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.metrics import silhouette_score
from sklearn.neighbors import NearestNeighbors
# Set random seed for reproducibility
np.random.seed(42)
class ClusteringModel:
    """Class for building and using clustering models for outlier detection"""

    def __init__(self, model_path=None, eps=0.5, min_samples=5, model_dir="models", viz_dir="visualizations"):
        """
        Initialize the clustering model

        Args:
            model_path (str): Path to save/load model
            eps (float): DBSCAN parameter - maximum distance between samples
            min_samples (int): DBSCAN parameter - minimum samples in neighborhood
            model_dir (str): Directory for model storage
            viz_dir (str): Directory for visualizations
        """
        self.model_dir = model_dir
        self.viz_dir = viz_dir
        self.model_path = model_path or os.path.join(model_dir, "clustering_model.pkl")
        self.eps = eps
        self.min_samples = min_samples
        self.model = None
        self.feature_columns = None
        self.numerical_features = None
        self.categorical_features = None
        self.scaler = None

        # Ensure directories exist
        os.makedirs(model_dir, exist_ok=True)
        os.makedirs(viz_dir, exist_ok=True)

    def prepare_features(self, df):
        """
        Prepare features for the clustering model

        Args:
            df (pd.DataFrame): Input DataFrame

        Returns:
            tuple: (numerical_features, categorical_features)
        """
        print("\nPreparing features for clustering...")

        # Identify numerical and categorical features
        numerical_features = df.select_dtypes(include=['int64', 'float64']).columns.tolist()
        categorical_features = df.select_dtypes(include=['object', 'bool']).columns.tolist()
```

```
        # Remove ID columns from features
        id_patterns = ['identifier', 'id', 'code', 'date']
        numerical_features = [col for col in numerical_features
                              if not any(pat in col.lower() for pat in id_patterns)]
        categorical_features = [col for col in categorical_features
                                if not any(pat in col.lower() for pat in id_patterns)]

        print(f"Selected {len(numerical_features)} numerical features and {len(categorical_features)} categorical feat
es")
        print(f"Numerical features: {', '.join(numerical_features)}")
        print(f"Categorical features: {', '.join(categorical_features)}")

        # Check for missing values
        missing_values = df[numerical_features + categorical_features].isnull().sum()
        features_with_missing = missing_values[missing_values > 0]

        if not features_with_missing.empty:
            print("\nFeatures with missing values:")
            for feature, count in features_with_missing.items():
                print(f" {feature}: {count} missing values ({count/len(df)*100:.2f}%)")

        self.numerical_features = numerical_features
        self.categorical_features = categorical_features

        return numerical_features, categorical_features

    def build_pipeline(self, numerical_features, categorical_features):
        """
        Build a preprocessing pipeline for clustering

        Args:
            numerical_features (list): List of numerical feature names
            categorical_features (list): List of categorical feature names

        Returns:
            ColumnTransformer: Preprocessing pipeline
        """
        print("\nBuilding preprocessing pipeline...")

        # Numerical preprocessing
        numerical_transformer = Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='median')),
            ('scaler', StandardScaler())
        ])

        # Categorical preprocessing
        categorical_transformer = Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),
            ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
        ])

        # Column transformer for preprocessing
```

```
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numerical_transformer, numerical_features),
            ('cat', categorical_transformer, categorical_features)
        ],
        remainder='drop'
    )

    return preprocessor

def find_optimal_eps(self, X, n_samples=1000):
    """
    Find optimal epsilon parameter for DBSCAN

    Args:
        X (np.ndarray): Preprocessed feature array
        n_samples (int): Number of samples to use for estimation

    Returns:
        float: Estimated optimal epsilon value
    """
    # Sample data if needed
    if len(X) > n_samples:
        indices = np.random.choice(len(X), n_samples, replace=False)
        X_sample = X[indices]
    else:
        X_sample = X

    # Compute nearest neighbors
    nbrs = NearestNeighbors(n_neighbors=self.min_samples).fit(X_sample)
    distances, _ = nbrs.kneighbors(X_sample)

    # Sort and get distance to kth neighbor
    distances = np.sort(distances[:, self.min_samples-1])

    # Estimate optimal epsilon using the "elbow" method
    # Calculate the rate of change in distances
    diffs = np.diff(distances)

    # Find the point where the rate of change is greatest
    elbow_idx = np.argmax(diffs) + 1
    optimal_eps = distances[elbow_idx]

    print(f"Estimated optimal epsilon: {optimal_eps:.4f}")
    return optimal_eps

def train(self, df, sample_size=None, use_kmeans=False, n_clusters=5):
    """
    Train a clustering model on the dataset

    Args:
        df (pd.DataFrame): Input DataFrame for training
```

```
    sample_size (int): Optional sample size to use
    use_kmeans (bool): Whether to use KMeans instead of DBSCAN
    n_clusters (int): Number of clusters for KMeans

Returns:
    bool: True if successful, False otherwise
"""
print("\nTraining clustering model...")

# Sample data if needed
if sample_size and len(df) > sample_size:
    df_sample = df.sample(sample_size, random_state=42)
    print(f"Sampled {len(df_sample)} rows from {len(df)} total rows")
else:
    df_sample = df
    print(f"Using all {len(df)} available rows for training")

# Prepare features
numerical_features, categorical_features = self.prepare_features(df_sample)

# Create features dataframe
X_df = df_sample[numerical_features + categorical_features].copy()

try:
    # Build preprocessing pipeline
    preprocessor = self.build_pipeline(numerical_features, categorical_features)

    # Fit and transform the data
    X = preprocessor.fit_transform(X_df)
    print(f"Preprocessed data shape: {X.shape}")

    # Choose clustering algorithm
    if use_kmeans:
        print(f"Training KMeans with {n_clusters} clusters...")
        model = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
    else:
        # Find optimal epsilon if not manually set
        if self.eps is None or self.eps <= 0:
            self.eps = self.find_optimal_eps(X)

        print(f"Training DBSCAN with eps={self.eps}, min_samples={self.min_samples}...")
        model = DBSCAN(eps=self.eps, min_samples=self.min_samples, n_jobs=-1)

    # Fit the model
    model.fit(X)

    # Store model and features
    self.model = model
    self.feature_columns = X_df.columns.tolist()
    self.numerical_features = numerical_features
    self.categorical_features = categorical_features
    self.preprocessor = preprocessor
```

```
        # Get cluster labels
        if use_kmeans:
            labels = model.labels_
            n_clusters = len(set(labels))
            print(f"Model trained with {n_clusters} clusters")
        else:
            labels = model.labels_
            n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
            n_outliers = list(labels).count(-1)
            print(f"Model trained with {n_clusters} clusters and {n_outliers} outliers")
            print(f"Outlier percentage: {n_outliers/len(labels)*100:.2f}%")

        print("Model training completed successfully.")
        return True

    except Exception as e:
        print(f"Error during model training: {e}")

        # Try with only numerical features if there was an error
        print("Attempting to train with only numerical features...")
        try:
            preprocessor = self.build_pipeline(numerical_features, [])
            X_num = df_sample[numerical_features].copy()
            X = preprocessor.fit_transform(X_num)

            if use_kmeans:
                model = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)
            else:
                # Find optimal epsilon if not manually set
                if self.eps is None or self.eps <= 0:
                    self.eps = self.find_optimal_eps(X)

                model = DBSCAN(eps=self.eps, min_samples=self.min_samples, n_jobs=-1)

            model.fit(X)

            self.model = model
            self.feature_columns = numerical_features
            self.numerical_features = numerical_features
            self.categorical_features = []
            self.preprocessor = preprocessor

            print("Model training with numerical features only completed successfully.")
            return True

        except Exception as e2:
            print(f"Error during fallback training: {e2}")
            return False

def predict(self, df):
    """
```

```
Use the trained model to detect outliers in the dataset

Args:
    df (pd.DataFrame): Dataset for prediction

Returns:
    pd.DataFrame: DataFrame with added prediction results
"""
print("\nDetecting outliers using clustering...")

if self.model is None:
    raise ValueError("Model not trained or loaded. Please train or load a model first.")

try:
    # Ensure we have all required columns
    missing_columns = [col for col in self.feature_columns if col not in df.columns]
    if missing_columns:
        print(f"Warning: Missing columns in dataset: {missing_columns}")

        # Add missing columns with default values
        for col in missing_columns:
            df[col] = 0
        print(f"Added missing columns with default values")

    # Prepare features for prediction
    all_feature_columns = [col for col in self.feature_columns if col in df.columns]
    X_df = df[all_feature_columns].copy()

    # Apply preprocessing
    X = self.preprocessor.transform(X_df)

    # Check if model is KMeans or DBSCAN
    if hasattr(self.model, 'predict'):
        # For KMeans, predict clusters and calculate distances to centroids
        clusters = self.model.predict(X)

        # Calculate distance to closest centroid
        distances = np.min(
            np.sqrt(np.sum((X - self.model.cluster_centers_[clusters.reshape(-1, 1)])**2, axis=2)),
            axis=1
        )

        # Define outliers as points with distance > threshold
        # Using 95th percentile as threshold
        threshold = np.percentile(distances, 95)
        outliers = distances > threshold

        # Calculate anomaly scores (normalized distances)
        scores = distances / np.max(distances)

    else:
        # For DBSCAN, use the model's labels
```

```
            # First, need to run fit_predict as DBSCAN doesn't have predict method
            clusters = self.model.fit_predict(X)

            # Points with cluster label -1 are outliers
            outliers = clusters == -1

            # Calculate anomaly scores based on distances to nearest core points
            # Note: This is a simplified approach
            nbrs = NearestNeighbors(n_neighbors=self.min_samples).fit(X)
            distances, _ = nbrs.kneighbors(X)

            # Use average distance to k nearest neighbors as score
            scores = np.mean(distances, axis=1)
            scores = scores / np.max(scores)  # Normalize

        # Add results to the dataframe
        result_df = df.copy()
        result_df['cluster'] = clusters
        result_df['is_outlier'] = outliers

        # Add clear text status
        result_df['outlier_status'] = result_df['is_outlier'].apply(
            lambda x: 'OUTLIER' if x else 'NORMAL'
        )

        # Add anomaly scores
        result_df['anomaly_score'] = scores.round(4)

        # Add timestamp of prediction
        result_df['prediction_time'] = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        # Print outlier summary
        outlier_count = outliers.sum()
        print(f"Detected {outlier_count} outliers out of {len(df)} records ({outlier_count/len(df)*100:.2f}%)")

        return result_df

    except Exception as e:
        print(f"Error detecting outliers: {e}")
        raise

def save_model(self):
    """
    Save the trained model and feature information

    Returns:
        bool: True if successful, False otherwise
    """
    print(f"\nSaving model to {self.model_path}...")

    if self.model is None:
        raise ValueError("No model to save. Please train a model first.")
```

```
    # Create directory if it doesn't exist
    os.makedirs(os.path.dirname(os.path.abspath(self.model_path)), exist_ok=True)

    # Create a package with all necessary components
    model_package = {
        'model': self.model,
        'preprocessor': self.preprocessor,
        'feature_columns': self.feature_columns,
        'numerical_features': self.numerical_features,
        'categorical_features': self.categorical_features,
        'eps': self.eps,
        'min_samples': self.min_samples,
        'date_trained': datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    }

    try:
        with open(self.model_path, 'wb') as f:
            pickle.dump(model_package, f)
        print(f"Model saved successfully to {self.model_path}")
        return True
    except Exception as e:
        print(f"Error saving model: {e}")
        return False

def load_model(self):
    """
    Load a previously trained clustering model

    Returns:
        bool: True if successful, False otherwise
    """
    print(f"Loading model from {self.model_path}...")

    try:
        with open(self.model_path, 'rb') as f:
            model_package = pickle.load(f)

        self.model = model_package['model']
        self.preprocessor = model_package['preprocessor']
        self.feature_columns = model_package['feature_columns']
        self.numerical_features = model_package['numerical_features']
        self.categorical_features = model_package['categorical_features']
        self.eps = model_package.get('eps', 0.5)
        self.min_samples = model_package.get('min_samples', 5)

        date_trained = model_package.get('date_trained', 'unknown')
        print(f"Model loaded successfully. Trained on: {date_trained}")

        print(f"Features: {len(self.feature_columns)} total features")
        print(f" - {len(self.numerical_features)} numerical features")
        print(f" - {len(self.categorical_features)} categorical features")
```

```
        # Check if model is KMeans or DBSCAN
        if hasattr(self.model, 'n_clusters'):
            print(f"KMeans model with {self.model.n_clusters} clusters")
        else:
            # For DBSCAN, count unique clusters
            if hasattr(self.model, 'labels_'):
                labels = self.model.labels_
                n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
                n_outliers = list(labels).count(-1)
                print(f"DBSCAN model with {n_clusters} clusters")
                print(f"Parameters: eps={self.eps}, min_samples={self.min_samples}")
                print(f"Detected {n_outliers} outliers in training data ({n_outliers/len(labels)*100:.2f}%)")

        return True

    except Exception as e:
        print(f"Error loading model: {e}")
        return False

def evaluate(self, df):
    """
    Evaluate the clustering model performance

    Args:
        df (pd.DataFrame): Evaluation dataset

    Returns:
        dict: Dictionary of evaluation metrics
    """
    print("\nEvaluating clustering model...")

    if self.model is None:
        raise ValueError("Model not trained or loaded. Please train or load a model first.")

    try:
        # Prepare features for evaluation
        all_feature_columns = [col for col in self.feature_columns if col in df.columns]
        X_df = df[all_feature_columns].copy()

        # Apply preprocessing
        X = self.preprocessor.transform(X_df)

        # Check if model is KMeans or DBSCAN
        if hasattr(self.model, 'predict'):
            # For KMeans
            labels = self.model.predict(X)

            # Calculate silhouette score
            silhouette = silhouette_score(X, labels) if len(set(labels)) > 1 else 0

            # Calculate inertia (sum of squared distances to centroids)
```

```
            inertia = self.model.inertia_

            metrics = {
                'silhouette_score': silhouette,
                'inertia': inertia,
                'n_clusters': self.model.n_clusters
            }

        else:
            # For DBSCAN, refit on evaluation data
            labels = self.model.fit_predict(X)

            # Count clusters and outliers
            n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
            n_outliers = list(labels).count(-1)
            outlier_percentage = n_outliers / len(labels) * 100

            # Calculate silhouette score for non-outlier points
            if n_clusters > 1:
                # Filter out outliers for silhouette calculation
                mask = labels != -1
                if sum(mask) > 1 and len(set(labels[mask])) > 1:
                    silhouette = silhouette_score(X[mask], labels[mask])
                else:
                    silhouette = 0
            else:
                silhouette = 0

            metrics = {
                'n_clusters': n_clusters,
                'n_outliers': n_outliers,
                'outlier_percentage': outlier_percentage,
                'silhouette_score': silhouette,
                'eps': self.eps,
                'min_samples': self.min_samples
            }

        print(f"Model evaluation completed:")
        for metric, value in metrics.items():
            print(f" - {metric}: {value}")

        return metrics

    except Exception as e:
        print(f"Error during model evaluation: {e}")
        return {'error': str(e)}

def save_predictions(self, result_df, output_file=None):
    """
    Save the prediction results to a CSV file

    Args:
```

```
            result_df (pd.DataFrame): DataFrame with prediction results
            output_file (str): Path to save the CSV file

        Returns:
            str: Path to the saved file
        """
        # If no specific output file is provided, create one with timestamp
        if output_file is None:
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            output_file = os.path.join("results", f"clustering_outliers_{timestamp}.csv")

        # Ensure output directory exists
        os.makedirs(os.path.dirname(output_file), exist_ok=True)

        print(f"\nSaving predictions to {output_file}...")

        try:
            result_df.to_csv(output_file, index=False)
            print(f"Results saved successfully.")

            # Print summary
            outlier_count = result_df['is_outlier'].sum()
            total_count = len(result_df)
            print(f"Summary: {outlier_count} outliers detected out of {total_count} records "
                 f"({outlier_count/total_count*100:.2f}%)")

            return output_file

        except Exception as e:
            print(f"Error saving predictions: {e}")
            return None
# If run directly, perform a test train and predict
if __name__ == "__main__":
    import argparse

    # Parse arguments
    parser = argparse.ArgumentParser(description='TED Procurement Clustering for Outlier Detection')
    parser.add_argument('--input', type=str, required=True, help='Path to preprocessed CSV file')
    parser.add_argument('--output', type=str, default='results/clustering_outliers.csv',
                        help='Path to output CSV file')
    parser.add_argument('--model', type=str, default='models/clustering_model.pkl',
                        help='Path to save/load model')
    parser.add_argument('--train', action='store_true', help='Train a new model')
    parser.add_argument('--predict', action='store_true', help='Make predictions')
    parser.add_argument('--evaluate', action='store_true', help='Evaluate the model')
    parser.add_argument('--sample', type=int, default=None, help='Sample size for training')
    parser.add_argument('--eps', type=float, default=None, help='DBSCAN epsilon parameter')
    parser.add_argument('--min-samples', type=int, default=5, help='DBSCAN min_samples parameter')
    parser.add_argument('--kmeans', action='store_true', help='Use KMeans instead of DBSCAN')
    parser.add_argument('--clusters', type=int, default=5, help='Number of clusters for KMeans')

    args = parser.parse_args()
```

```
    # Create model instance
    model = ClusteringModel(
        model_path=args.model,
        eps=args.eps,
        min_samples=args.min_samples
    )

    # Load data
    df = pd.read_csv(args.input)
    print(f"Loaded dataset with {len(df)} rows and {len(df.columns)} columns")

    # Training, prediction, or evaluation
    if args.train:
        print("=== Training Mode ===")
        model.train(df, sample_size=args.sample, use_kmeans=args.kmeans, n_clusters=args.clusters)
        model.save_model()

    if args.predict:
        print("=== Prediction Mode ===")
        if not model.model and not args.train:
            model.load_model()
        result_df = model.predict(df)
        model.save_predictions(result_df, args.output)

    if args.evaluate:
        print("=== Evaluation Mode ===")
        if not model.model and not args.train:
            model.load_model()
        metrics = model.evaluate(df)
```

FILE: transforming/isolation_forest_model.py

```
#!/usr/bin/env python3
"""
Isolation Forest Model Module
This module provides the machine learning functionality for TED procurement outlier detection.
It handles training, prediction, model serialization, and visualization.
Author: Your Name
Date: May 16, 2025
"""
import os
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
from datetime import datetime
from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
# Set random seed for reproducibility
np.random.seed(42)
class IsolationForestModel:
    """Class for building and using the Isolation Forest model"""

    def __init__(self, model_path=None, contamination=0.05, model_dir="models", viz_dir="visualizations"):
        self.model_dir = model_dir
        self.viz_dir = viz_dir
        self.model_path = model_path or os.path.join(model_dir, "isolation_forest_model.pkl")
        self.contamination = contamination
        self.model = None
        self.feature_columns = None
        self.numerical_features = None
        self.categorical_features = None

        # Ensure directories exist
        os.makedirs(model_dir, exist_ok=True)
        os.makedirs(viz_dir, exist_ok=True)

    def prepare_features(self, df):
        """
        Prepare features for the isolation forest model
        """
        print("\nPreparing features for outlier detection...")

        # Identify numerical and categorical features
        numerical_features = df.select_dtypes(include=['int64', 'float64']).columns.tolist()
        categorical_features = df.select_dtypes(include=['object', 'bool']).columns.tolist()

        # Remove ID columns from features
        id_patterns = ['identifier', 'id', 'code', 'date']
        numerical_features = [col for col in numerical_features
                              if not any(pat in col.lower() for pat in id_patterns)]
        categorical_features = [col for col in categorical_features
                                if not any(pat in col.lower() for pat in id_patterns)]

        print(f"Selected {len(numerical_features)} numerical features and {len(categorical_features)} categorical feat
es")
        print(f"Numerical features: {', '.join(numerical_features)}")
        print(f"Categorical features: {', '.join(categorical_features)}")

        # Check for missing values
        missing_values = df[numerical_features + categorical_features].isnull().sum()
        features_with_missing = missing_values[missing_values > 0]
        if not features_with_missing.empty:
            print("\nFeatures with missing values:")
            for feature, count in features_with_missing.items():
                print(f" {feature}: {count} missing values ({count/len(df)*100:.2f}%)")
```

```
        self.numerical_features = numerical_features
        self.categorical_features = categorical_features

        return numerical_features, categorical_features

    def build_pipeline(self, numerical_features, categorical_features):
        """
        Build a preprocessing and isolation forest pipeline
        """
        print("\nBuilding model pipeline...")

        # Numerical preprocessing
        numerical_transformer = Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='median')),
            ('scaler', StandardScaler())
        ])

        # Categorical preprocessing
        categorical_transformer = Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),
            ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
        ])

        # Column transformer for preprocessing
        preprocessor = ColumnTransformer(
            transformers=[
                ('num', numerical_transformer, numerical_features),
                ('cat', categorical_transformer, categorical_features)
            ], remainder='drop'
        )

        # Create the full pipeline with isolation forest
        pipeline = Pipeline(steps=[
            ('preprocessor', preprocessor),
            ('outlier_detector', IsolationForest(
                n_estimators=100,
                max_samples='auto',
                contamination=self.contamination,
                random_state=42,
                n_jobs=-1  # Use all available cores
            ))
        ])

        return pipeline

    def train(self, df, sample_size=None):
        """
        Train an isolation forest model on the dataset
        Parameters:
        -----------
        df : pd.DataFrame
            Input DataFrame for training
```

```
sample_size : int, optional
    Number of rows to sample for training
"""
print("\nTraining Isolation Forest model...")

# Sample data if needed
if sample_size and len(df) > sample_size:
    df_sample = df.sample(sample_size, random_state=42)
    print(f"Sampled {len(df_sample)} rows from {len(df)} total rows")
else:
    df_sample = df
    print(f"Using all {len(df)} available rows for training")

# Prepare features
numerical_features, categorical_features = self.prepare_features(df_sample)

# Create features dataframe
X = df_sample[numerical_features + categorical_features].copy()

# Build and train the pipeline
try:
    pipeline = self.build_pipeline(numerical_features, categorical_features)

    # Fit the model
    pipeline.fit(X)
    print("Model training completed successfully.")

    self.model = pipeline
    self.feature_columns = X.columns.tolist()
    self.numerical_features = numerical_features
    self.categorical_features = categorical_features

    return True
except Exception as e:
    print(f"Error during model training: {e}")

    # Try with only numerical features if there was an error
    print("Attempting to train with only numerical features...")
    try:
        pipeline = self.build_pipeline(numerical_features, [])
        X_num = df_sample[numerical_features].copy()
        pipeline.fit(X_num)

        print("Model training with numerical features only completed successfully.")

        self.model = pipeline
        self.feature_columns = numerical_features
        self.numerical_features = numerical_features
        self.categorical_features = []

        return True
    except Exception as e2:
```

```
            print(f"Error during fallback training: {e2}")
            return False

def predict(self, df):
    """
    Use the trained model to detect outliers in the dataset
    Parameters:
    -----------
    df : pd.DataFrame
        Dataset for prediction
    Returns:
    --------
    pd.DataFrame
        DataFrame with added prediction results
    """
    print("\nDetecting outliers...")

    if self.model is None:
        raise ValueError("Model not trained or loaded. Please train or load a model first.")

    try:
        # Ensure we have all required columns
        missing_columns = [col for col in self.feature_columns if col not in df.columns]
        if missing_columns:
            print(f"Warning: Missing columns in dataset: {missing_columns}")

            # Add missing columns with default values (0 for numeric columns)
            for col in missing_columns:
                df[col] = 0
            print(f"Added missing columns with default values")

        # Ensure columns are in the right order
        all_feature_columns = [col for col in self.feature_columns if col in df.columns]

        # Prepare features
        X = df[all_feature_columns].copy()

        # Predict outliers (1: inlier, -1: outlier)
        predictions = self.model.predict(X)
        outliers = predictions == -1

        # Get anomaly scores if possible
        try:
            if hasattr(self.model, 'decision_function'):
                scores = self.model.decision_function(X)
            elif hasattr(self.model[-1], 'decision_function'):  # For pipeline
                scores = self.model[-1].decision_function(self.model[:-1].transform(X))
            else:
                scores = None
        except Exception as e:
            print(f"Warning: Could not compute anomaly scores: {e}")
            scores = None
```

```
        # Add results to the dataframe
        result_df = df.copy()
        result_df['is_outlier'] = outliers

        # Add clear text status
        result_df['outlier_status'] = result_df['is_outlier'].apply(
            lambda x: 'OUTLIER' if x else 'NORMAL'
        )

        # Add anomaly scores if available
        if scores is not None:
            result_df['anomaly_score'] = scores.round(4)

        # Add timestamp of prediction
        result_df['prediction_time'] = datetime.now().strftime("%Y-%m-%d %H:%M:%S")

        # Print outlier summary
        outlier_count = outliers.sum()
        print(f"Detected {outlier_count} outliers out of {len(df)} records ({outlier_count/len(df)*100:.2f}%)")

        return result_df
    except Exception as e:
        print(f"Error detecting outliers: {e}")
        raise

def save_model(self):
    """
    Save the trained model and feature information
    """
    print(f"\nSaving model to {self.model_path}...")

    if self.model is None:
        raise ValueError("No model to save. Please train a model first.")

    # Create directory if it doesn't exist
    os.makedirs(os.path.dirname(os.path.abspath(self.model_path)), exist_ok=True)

    # Create a package with all necessary components
    model_package = {
        'model': self.model,
        'feature_columns': self.feature_columns,
        'numerical_features': self.numerical_features,
        'categorical_features': self.categorical_features,
        'date_trained': datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    }

    try:
        with open(self.model_path, 'wb') as f:
            pickle.dump(model_package, f)
        print(f"Model saved successfully to {self.model_path}")
        return True
```

```
    except Exception as e:
        print(f"Error saving model: {e}")
        return False

def load_model(self):
    """
    Load a previously trained isolation forest model
    Returns:
    --------
    bool
        True if successful, False otherwise
    """
    print(f"Loading model from {self.model_path}...")

    try:
        with open(self.model_path, 'rb') as f:
            model_package = pickle.load(f)

        self.model = model_package['model']
        self.feature_columns = model_package['feature_columns']
        self.numerical_features = model_package['numerical_features']
        self.categorical_features = model_package['categorical_features']

        date_trained = model_package.get('date_trained', 'unknown')
        print(f"Model loaded successfully. Trained on: {date_trained}")
        print(f"Features: {len(self.feature_columns)} total features")
        print(f" - {len(self.numerical_features)} numerical features")
        print(f" - {len(self.categorical_features)} categorical features")

        return True
    except Exception as e:
        print(f"Error loading model: {e}")
        return False

def visualize_outliers(self, result_df, output_file=None):
    """
    Skip visualization and just return empty dict
    Parameters:
    -----------
    result_df : pd.DataFrame
        DataFrame with prediction results
    output_file : str, optional
        Path to save the visualizations (not used)
    Returns:
    --------
    dict
        Empty dictionary
    """
    print("\nSkipping outlier visualizations...")
    return {}

def save_predictions(self, result_df, output_file=None):
```

```
        """
        Save the prediction results to a CSV file
        Parameters:
        -----------
        result_df : pd.DataFrame
            DataFrame with prediction results
        output_file : str, optional
            Path to save the CSV file
        Returns:
        --------
        str
            Path to the saved file
        """
        # If no specific output file is provided, create one with timestamp
        if output_file is None:
            timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
            output_file = os.path.join("results", f"outliers_{timestamp}.csv")

        # Ensure output directory exists
        os.makedirs(os.path.dirname(output_file), exist_ok=True)

        print(f"\nSaving predictions to {output_file}...")
        try:
            result_df.to_csv(output_file, index=False)
            print(f"Results saved successfully.")

            # Print summary
            outlier_count = result_df['is_outlier'].sum()
            total_count = len(result_df)
            print(f"Summary: {outlier_count} outliers detected out of {total_count} records ({outlier_count/total_cour
100:.2f}%)")

            return output_file
        except Exception as e:
            print(f"Error saving predictions: {e}")
            return None
# If run directly, perform a test train and predict
if __name__ == "__main__":
    import argparse

    # Parse arguments
    parser = argparse.ArgumentParser(description='TED Procurement Outlier Detection')
    parser.add_argument('--input', type=str, required=True, help='Path to preprocessed CSV file')
    parser.add_argument('--output', type=str, default='results/outliers.csv', help='Path to output CSV file')
    parser.add_argument('--model', type=str, default='models/isolation_forest_model.pkl', help='Path to save/load mode
)
    parser.add_argument('--train', action='store_true', help='Train a new model')
    parser.add_argument('--predict', action='store_true', help='Make predictions')
    parser.add_argument('--sample', type=int, default=None, help='Sample size for training')
    parser.add_argument('--contamination', type=float, default=0.05, help='Expected proportion of outliers (0.0-0.5)')

    args = parser.parse_args()
```

```
    # Create model instance
    model = IsolationForestModel(
        model_path=args.model,
        contamination=args.contamination
    )

    # Load data
    df = pd.read_csv(args.input)
    print(f"Loaded dataset with {len(df)} rows and {len(df.columns)} columns")

    # Training or prediction
    if args.train:
        print("=== Training Mode ===")
        model.train(df, sample_size=args.sample)
        model.save_model()

    if args.predict:
        print("=== Prediction Mode ===")
        if not model.model and not args.train:
            model.load_model()

        result_df = model.predict(df)
        model.save_predictions(result_df, args.output)
```

FILE: components/ted_data_storage.py

```
#!/usr/bin/env python3
"""
Simplified TED Data Storage Module
A clean and simple database for storing model results and statistics.
"""
import os
import sqlite3
import pandas as pd
import json
import uuid
from datetime import datetime
import logging
logger = logging.getLogger(__name__)
class TEDDataStorage:
    """Simple storage for TED procurement analysis results"""

    def __init__(self, db_path="data/ted_results.db"):
        """Initialize storage with database path"""
        self.db_path = db_path

        # Ensure directory exists
        os.makedirs(os.path.dirname(os.path.abspath(db_path)), exist_ok=True)

        # Initialize database
```

```
    self._init_database()
    logger.info(f"Storage initialized: {self.db_path}")

def _init_database(self):
    """Create database tables if they don't exist"""
    with sqlite3.connect(self.db_path) as conn:
        cursor = conn.cursor()

        # Table 1: Model runs metadata
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS model_runs (
                run_id TEXT PRIMARY KEY,
                model_type TEXT NOT NULL,
                parameters TEXT,
                execution_date TEXT NOT NULL,
                record_count INTEGER,
                outlier_count INTEGER,
                outlier_percentage REAL,
                notes TEXT
            )
        ''')

        # Table 2: All procurement results with outlier info
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS procurement_results (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                run_id TEXT NOT NULL,
                notice_id TEXT,
                value_eur REAL,
                country TEXT,
                notice_type TEXT,
                is_outlier INTEGER NOT NULL,
                anomaly_score REAL,
                FOREIGN KEY (run_id) REFERENCES model_runs(run_id)
            )
        ''')

        # Create indexes for faster queries
        cursor.execute('CREATE INDEX IF NOT EXISTS idx_run_id ON procurement_results(run_id)')
        cursor.execute('CREATE INDEX IF NOT EXISTS idx_outlier ON procurement_results(is_outlier)')
        cursor.execute('CREATE INDEX IF NOT EXISTS idx_notice_id ON procurement_results(notice_id)')

        conn.commit()

def store_results(self, model_type, result_df, parameters=None, notes=None):
    """
    Store complete model results in one operation

    Args:
        model_type (str): Type of model ('isolation_forest', 'dbscan', 'kmeans')
        result_df (pd.DataFrame): DataFrame with outlier detection results
        parameters (dict): Model parameters
```

```
    notes (str): Additional notes

Returns:
    str: Run ID for the stored results
"""
run_id = str(uuid.uuid4())

try:
    with sqlite3.connect(self.db_path) as conn:
        cursor = conn.cursor()

        # Calculate statistics
        total_records = len(result_df)
        outlier_count = int(result_df.get('is_outlier', pd.Series([False])).sum())
        outlier_percentage = (outlier_count / total_records * 100) if total_records > 0 else 0

        # Store run metadata
        cursor.execute('''
            INSERT INTO model_runs
            (run_id, model_type, parameters, execution_date, record_count,
             outlier_count, outlier_percentage, notes)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?)
        ''', (
            run_id,
            model_type,
            json.dumps(parameters) if parameters else None,
            datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
            total_records,
            outlier_count,
            outlier_percentage,
            notes
        ))

        # Prepare procurement results data
        results_data = []
        for _, row in result_df.iterrows():
            # Extract key fields with fallbacks
            notice_id = self._extract_field(row, ['notice-identifier', 'notice_id', 'identifier'])
            value_eur = self._extract_numeric_field(row, ['total-value-eur', 'value_eur', 'total-value'])
            country = self._extract_field(row, ['organisation-country-buyer', 'country'])
            notice_type = self._extract_field(row, ['notice-type', 'notice_type'])
            is_outlier = int(row.get('is_outlier', False))
            anomaly_score = float(row.get('anomaly_score', 0) or 0)

            results_data.append((
                run_id, notice_id, value_eur, country, notice_type,
                is_outlier, anomaly_score
            ))

        # Store all procurement results
        cursor.executemany('''
            INSERT INTO procurement_results
```

```
                    (run_id, notice_id, value_eur, country, notice_type, is_outlier, anomaly_score)
                    VALUES (?, ?, ?, ?, ?, ?, ?)
                ''', results_data)

                conn.commit()

            logger.info(f"Stored results: {run_id} ({model_type}) - {total_records} records, {outlier_count} outliers
outlier_percentage:.2f}%)")
            return run_id

        except Exception as e:
            logger.error(f"Error storing results: {e}")
            return None

    def _extract_field(self, row, possible_names, default='unknown'):
        """Extract field value trying multiple possible column names"""
        for name in possible_names:
            if name in row.index and pd.notna(row[name]):
                return str(row[name])
        return default

    def _extract_numeric_field(self, row, possible_names, default=0.0):
        """Extract numeric field value trying multiple possible column names"""
        for name in possible_names:
            if name in row.index and pd.notna(row[name]):
                try:
                    return float(row[name])
                except (ValueError, TypeError):
                    continue
        return default

    def get_run_summary(self, run_id):
        """Get summary statistics for a specific run"""
        with sqlite3.connect(self.db_path) as conn:
            conn.row_factory = sqlite3.Row
            cursor = conn.cursor()

            # Get run metadata
            cursor.execute('''
                SELECT * FROM model_runs WHERE run_id = ?
            ''', (run_id,))

            run_info = cursor.fetchone()
            if not run_info:
                return None

            # Convert to dict and parse parameters
            result = dict(run_info)
            if result['parameters']:
                try:
                    result['parameters'] = json.loads(result['parameters'])
                except:
```

```
                pass

        # Get outlier details
        cursor.execute('''
            SELECT
                COUNT(*) as total_outliers,
                AVG(anomaly_score) as avg_score,
                MIN(value_eur) as min_value,
                MAX(value_eur) as max_value,
                AVG(value_eur) as avg_value
            FROM procurement_results
            WHERE run_id = ? AND is_outlier = 1
        ''', (run_id,))

        outlier_stats = dict(cursor.fetchone())
        result['outlier_stats'] = outlier_stats

        # Get country breakdown
        cursor.execute('''
            SELECT
                country,
                COUNT(*) as total,
                SUM(is_outlier) as outliers,
                ROUND(AVG(is_outlier) * 100, 2) as outlier_pct
            FROM procurement_results
            WHERE run_id = ?
            GROUP BY country
            ORDER BY total DESC
            LIMIT 10
        ''', (run_id,))

        result['country_breakdown'] = [dict(row) for row in cursor.fetchall()]

        return result

def get_recent_runs(self, limit=10):
    """Get recent model runs with basic statistics"""
    with sqlite3.connect(self.db_path) as conn:
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        cursor.execute('''
            SELECT
                run_id,
                model_type,
                execution_date,
                record_count,
                outlier_count,
                outlier_percentage,
                notes
            FROM model_runs
            ORDER BY execution_date DESC
```

```
            LIMIT ?
        ''', (limit,))

        return [dict(row) for row in cursor.fetchall()]

def get_outliers(self, run_id, limit=100):
    """Get outliers from a specific run"""
    with sqlite3.connect(self.db_path) as conn:
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        cursor.execute('''
            SELECT *
            FROM procurement_results
            WHERE run_id = ? AND is_outlier = 1
            ORDER BY anomaly_score DESC
            LIMIT ?
        ''', (run_id, limit))

        return [dict(row) for row in cursor.fetchall()]

def get_run_data(self, run_id):
    """Get all data from a specific run as DataFrame"""
    with sqlite3.connect(self.db_path) as conn:
        query = '''
            SELECT * FROM procurement_results WHERE run_id = ?
        '''
        return pd.read_sql_query(query, conn, params=(run_id,))

def export_run_csv(self, run_id, output_file):
    """Export run data to CSV"""
    try:
        df = self.get_run_data(run_id)
        if df.empty:
            logger.warning(f"No data found for run {run_id}")
            return False

        df.to_csv(output_file, index=False)
        logger.info(f"Exported {len(df)} records to {output_file}")
        return True

    except Exception as e:
        logger.error(f"Error exporting data: {e}")
        return False

def get_statistics(self):
    """Get overall database statistics"""
    with sqlite3.connect(self.db_path) as conn:
        conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        # Basic counts
```

```
        cursor.execute('SELECT COUNT(*) as total_runs FROM model_runs')
        total_runs = cursor.fetchone()['total_runs']

        cursor.execute('SELECT COUNT(*) as total_records FROM procurement_results')
        total_records = cursor.fetchone()['total_records']

        cursor.execute('SELECT COUNT(*) as total_outliers FROM procurement_results WHERE is_outlier = 1')
        total_outliers = cursor.fetchone()['total_outliers']

        # Date range
        cursor.execute('SELECT MIN(execution_date) as first_run, MAX(execution_date) as last_run FROM model_runs')
        date_range = dict(cursor.fetchone())

        # Model type distribution
        cursor.execute('''
            SELECT model_type, COUNT(*) as count
            FROM model_runs
            GROUP BY model_type
            ORDER BY count DESC
        ''')
        model_types = {row['model_type']: row['count'] for row in cursor.fetchall()}

        return {
            'total_runs': total_runs,
            'total_records': total_records,
            'total_outliers': total_outliers,
            'overall_outlier_percentage': (total_outliers / total_records * 100) if total_records > 0 else 0,
            'date_range': date_range,
            'model_types': model_types
        }

def cleanup_old_runs(self, days_to_keep=30):
    """Remove runs older than specified days"""
    from datetime import datetime, timedelta

    cutoff_date = (datetime.now() - timedelta(days=days_to_keep)).strftime("%Y-%m-%d")

    with sqlite3.connect(self.db_path) as conn:
        cursor = conn.cursor()

        # Get runs to delete
        cursor.execute('''
            SELECT run_id FROM model_runs WHERE execution_date < ?
        ''', (cutoff_date,))

        old_runs = [row[0] for row in cursor.fetchall()]

        if old_runs:
            # Delete procurement results first (foreign key constraint)
            placeholders = ','.join(['?'] * len(old_runs))
            cursor.execute(f'''
                DELETE FROM procurement_results WHERE run_id IN ({placeholders})
```

```
                ''', old_runs)

                # Delete model runs
                cursor.execute(f'''
                    DELETE FROM model_runs WHERE run_id IN ({placeholders})
                ''', old_runs)

                conn.commit()
                logger.info(f"Cleaned up {len(old_runs)} old runs")

            return len(old_runs)
# Example usage and testing
if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description='TED Data Storage Management')
    parser.add_argument('--stats', action='store_true', help='Show database statistics')
    parser.add_argument('--runs', action='store_true', help='Show recent runs')
    parser.add_argument('--run-details', type=str, help='Show details for specific run ID')
    parser.add_argument('--outliers', type=str, help='Show outliers for specific run ID')
    parser.add_argument('--export', nargs=2, help='Export run data: run_id output_file')
    parser.add_argument('--cleanup', type=int, help='Clean up runs older than X days')

    args = parser.parse_args()

    # Initialize storage
    storage = TEDDataStorage()

    if args.stats:
        stats = storage.get_statistics()
        print("\n=== DATABASE STATISTICS ===")
        print(f"Total Runs: {stats['total_runs']}")
        print(f"Total Records: {stats['total_records']:,}")
        print(f"Total Outliers: {stats['total_outliers']:,} ({stats['overall_outlier_percentage']:.2f}%)")
        print(f"Date Range: {stats['date_range']['first_run']} to {stats['date_range']['last_run']}")
        print(f"Model Types: {stats['model_types']}")

    elif args.runs:
        runs = storage.get_recent_runs(20)
        print("\n=== RECENT RUNS ===")
        for run in runs:
            print(f"{run['execution_date']} | {run['model_type']} | {run['record_count']:,} records | {run['outlier_co
t']:,} outliers ({run['outlier_percentage']:.2f}%)")
            print(f"  Run ID: {run['run_id']}")
            if run['notes']:
                print(f"  Notes: {run['notes']}")
            print()

    elif args.run_details:
        details = storage.get_run_summary(args.run_details)
        if details:
            print(f"\n=== RUN DETAILS: {args.run_details} ===")
```

```
            print(f"Model Type: {details['model_type']}")
            print(f"Date: {details['execution_date']}")
            print(f"Records: {details['record_count']:,}")
            print(f"Outliers: {details['outlier_count']:,} ({details['outlier_percentage']:.2f}%)")

            if details['outlier_stats']['total_outliers']:
                print(f"\nOutlier Statistics:")
                print(f"  Average Score: {details['outlier_stats']['avg_score']:.4f}")
                print(f"  Value Range: _{details['outlier_stats']['min_value']:,.2f} - _{details['outlier_stats']['max
alue']:,.2f}")
                print(f"  Average Value: _{details['outlier_stats']['avg_value']:,.2f}")

            if details['country_breakdown']:
                print(f"\nTop Countries:")
                for country in details['country_breakdown'][:5]:
                    print(f"  {country['country']}: {country['total']:,} records, {country['outliers']} outliers ({cou
ry['outlier_pct']}%)")
        else:
            print(f"Run ID not found: {args.run_details}")

    elif args.outliers:
        outliers = storage.get_outliers(args.outliers, 20)
        if outliers:
            print(f"\n=== OUTLIERS FOR RUN: {args.outliers} ===")
            for outlier in outliers:
                print(f"Notice: {outlier['notice_id']} | Value: _{outlier['value_eur']:,.2f} | Score: {outlier['anomal
score']:.4f} | Country: {outlier['country']}")
        else:
            print(f"No outliers found for run: {args.outliers}")

    elif args.export:
        run_id, output_file = args.export
        success = storage.export_run_csv(run_id, output_file)
        if success:
            print(f"Data exported to: {output_file}")
        else:
            print("Export failed")

    elif args.cleanup:
        deleted = storage.cleanup_old_runs(args.cleanup)
        print(f"Cleaned up {deleted} runs older than {args.cleanup} days")

    else:
        parser.print_help()
```

FILE: components/ted_data_preprocessor.py

```
#!/usr/bin/env python3
"""
TED Data Preprocessor Module
This module handles the preprocessing of TED procurement data for machine learning.
```

```
It cleans, normalizes, and transforms the data to make it suitable for outlier detection.
Author: Your Name
Date: May 16, 2025
"""
import os
import csv
import pandas as pd
import numpy as np
from datetime import datetime
class TEDDataPreprocessor:
    """Class for preprocessing TED procurement data"""

    def __init__(self, input_file=None, output_dir="processed_data"):
        self.input_file = input_file
        self.output_dir = output_dir

        # Ensure output directory exists
        os.makedirs(output_dir, exist_ok=True)

    def load_csv_safely(self, file_path):
        """
        Load a CSV file with robust error handling for inconsistent field counts
        """
        print(f"Loading file: {file_path}")
        try:
            # First try pandas with default settings
            df = pd.read_csv(file_path)
            print(f"Successfully loaded with pandas: {len(df)} rows")
            return df
        except Exception as e:
            print(f"Standard loading failed: {str(e)}")
            print("Trying alternative loading method...")

            # Manual loading using csv module
            rows = []
            header = None
            max_fields = 0

            # First pass to get header and max field count
            with open(file_path, 'r', newline='', encoding='utf-8', errors='replace') as f:
                reader = csv.reader(f)
                for i, row in enumerate(reader):
                    if i == 0:
                        header = row
                    else:
                        max_fields = max(max_fields, len(row))

            max_fields = max(max_fields, len(header))
            print(f"Max field count: {max_fields}")

            # Second pass to read the data
            with open(file_path, 'r', newline='', encoding='utf-8', errors='replace') as f:
```

```
            reader = csv.reader(f)
            next(reader)  # Skip header
            for row in reader:
                # Pad or truncate row
                if len(row) < max_fields:
                    row = row + [''] * (max_fields - len(row))
                elif len(row) > max_fields:
                    row = row[:max_fields]
                rows.append(row)

        # Ensure header has the right length
        if len(header) < max_fields:
            header.extend([f"unknown_{i}" for i in range(len(header), max_fields)])
        elif len(header) > max_fields:
            header = header[:max_fields]

        # Create DataFrame
        df = pd.DataFrame(rows, columns=header)
        print(f"Successfully loaded with manual method: {len(df)} rows, {len(df.columns)} columns")
        return df

def load_data(self):
    """Load the TED procurement data"""
    if not self.input_file:
        raise ValueError("Input file not specified")

    return self.load_csv_safely(self.input_file)

def clean_data_for_ml(self, df):
    """
    Clean and normalize TED procurement data for machine learning
    Parameters:
    -----------
    df : pd.DataFrame
        Raw TED procurement data
    Returns:
    --------
    pd.DataFrame
        Cleaned data ready for ML
    """
    print("\nCleaning and normalizing data...")
    cleaned_df = df.copy()

    # Step 1: Remove link fields
    link_cols = [col for col in cleaned_df.columns if
                 'link' in col.lower() or
                 'url' in col.lower() or
                 'xml' in col.lower() or
                 'html' in col.lower() or
                 'pdf' in col.lower()]

    if link_cols:
```

```
    print(f"Removing {len(link_cols)} link-related columns")
    cleaned_df = cleaned_df.drop(columns=link_cols)

# Step 2: Extract clean currency information
if 'estimated-value-cur-proc' in cleaned_df.columns:
    valid_currencies = ['EUR', 'SEK', 'BGN', 'NOK', 'PLN', 'CZK', 'HUF', 'DKK', 'RON']

    def extract_currency(value):
        if pd.isna(value) or not isinstance(value, str):
            return 'EUR'  # Default currency
        for curr in valid_currencies:
            if curr in value:
                return curr
        return 'EUR'

    cleaned_df['currency'] = cleaned_df['estimated-value-cur-proc'].apply(extract_currency)
    currency_counts = cleaned_df['currency'].value_counts()
    print(f"Currency distribution: {dict(currency_counts)}")
else:
    # Default currency if not present
    cleaned_df['currency'] = 'EUR'

# Step 3: Process monetary values
for col in ['total-value', 'framework-value-notice', 'subcontracting-value']:
    if col in cleaned_df.columns:
        # Convert to string first
        cleaned_df[col] = cleaned_df[col].astype(str)
        # Clean up the values
        cleaned_df[col] = cleaned_df[col].str.replace(',', '.')
        cleaned_df[col] = cleaned_df[col].str.replace(r'[^\d.]', '', regex=True)
        # Convert to numeric
        cleaned_df[col] = pd.to_numeric(cleaned_df[col], errors='coerce')
        valid_count = cleaned_df[col].count()
        print(f"Processed {col}: {valid_count} valid values")

# Step 4: Normalize monetary values to EUR
exchange_rates = {
    'EUR': 1.0,
    'SEK': 0.087,
    'BGN': 0.51,
    'NOK': 0.086,
    'PLN': 0.23,
    'CZK': 0.039,
    'HUF': 0.0026,
    'DKK': 0.13,
    'RON': 0.20
}

if 'total-value' in cleaned_df.columns:
    cleaned_df['total-value-eur'] = cleaned_df.apply(
        lambda row: row['total-value'] * exchange_rates.get(row['currency'], 1.0)
        if pd.notna(row['total-value']) else np.nan,
```

```
        axis=1
    )
    print(f"Normalized total values to EUR: {cleaned_df['total-value-eur'].count()} values")

    # Cap outliers for better model stability
    # Calculate 95th percentile for capping
    percentile_95 = cleaned_df['total-value-eur'].quantile(0.95)
    cleaned_df['total-value-eur-capped'] = cleaned_df['total-value-eur'].apply(
        lambda x: min(x, percentile_95) if pd.notna(x) else x
    )

    # Add outlier flag based on simple threshold for initial filtering
    if 'total-value-eur' in cleaned_df.columns:
        # Flag values above 95th percentile as potential outliers
        cleaned_df['is_outlier'] = (cleaned_df['total-value-eur'] > percentile_95).astype(bool)

    # Log transform of monetary values (useful for ML)
    cleaned_df['total-value-eur-log'] = np.log1p(
        cleaned_df['total-value-eur'].replace([np.inf, -np.inf, np.nan], 0)
    )

# Step 5: Extract bidder information
if 'winner-size' in cleaned_df.columns:
    try:
        # Count bidders
        cleaned_df['bidder-count'] = cleaned_df['winner-size'].astype(str).apply(
            lambda x: len(x.split('|')) if pd.notna(x) and x != 'nan' and x != 'None' else 0
        )

        # Extract primary bidder size
        cleaned_df['primary-bidder-size'] = cleaned_df['winner-size'].astype(str).apply(
            lambda x: x.split('|')[0] if pd.notna(x) and x != 'nan' and x != 'None' else np.nan
        )

        # Convert size to numeric representation
        size_mapping = {
            'micro': 1,
            'small': 2,
            'sme': 2.5,  # between small and medium
            'medium': 3,
            'large': 4
        }
        cleaned_df['bidder-size-numeric'] = cleaned_df['primary-bidder-size'].map(size_mapping)

        print(f"Extracted bidder info: max bidders = {cleaned_df['bidder-count'].max()}")
    except Exception as e:
        print(f"Error extracting bidder information: {e}")

# Step 6: Format categorical features
if 'notice-type' in cleaned_df.columns:
    # Get top notice types
    top_types = cleaned_df['notice-type'].value_counts().head(10).index.tolist()
```

```
        # Create dummy variables for top types
        for notice_type in top_types:
            col_name = f"notice_is_{notice_type}"
            cleaned_df[col_name] = (cleaned_df['notice-type'] == notice_type).astype(int)

        print(f"Created dummy variables for {len(top_types)} notice types")

    return cleaned_df

def prepare_for_ml(self, df):
    """
    Final preparation to make the data compatible with the ML algorithm
    Parameters:
    -----------
    df : pd.DataFrame
        Cleaned DataFrame
    Returns:
    --------
    pd.DataFrame
        ML-ready DataFrame with only relevant features
    """
    print("\nPreparing final ML-ready dataset...")

    # Focus on rows with valid monetary values
    if 'total-value-eur' in df.columns:
        ml_df = df.dropna(subset=['total-value-eur']).copy()
        print(f"Kept {len(ml_df)}/{len(df)} rows with valid monetary values")
    else:
        ml_df = df.copy()
        print("Warning: No monetary values found")

    # Select features important for ML
    keep_columns = []

    # Always include ID if available
    id_cols = [col for col in ml_df.columns if 'identifier' in col.lower() or 'id' in col.lower()]
    if id_cols:
        keep_columns.extend(id_cols[:1])  # Take the first ID column

    # Include monetary values
    money_cols = ['total-value-eur', 'total-value-eur-capped', 'total-value-eur-log']
    keep_columns.extend([col for col in money_cols if col in ml_df.columns])

    # Include bidder info
    bidder_cols = ['bidder-count', 'bidder-size-numeric']
    keep_columns.extend([col for col in bidder_cols if col in ml_df.columns])

    # Include notice type dummies
    notice_dummies = [col for col in ml_df.columns if col.startswith('notice_is_')]
    keep_columns.extend(notice_dummies)
```

```
    # Filter to only existing columns
    keep_columns = [col for col in keep_columns if col in ml_df.columns]

    # Keep other potentially useful columns
    remain_cols = []
    for col in ml_df.columns:
        # Skip already included columns
        if col in keep_columns:
            continue

        # Skip text fields and other less useful columns
        if ('text' in col.lower() or
            'description' in col.lower() or
            'currency' in col.lower() or
            'link' in col.lower()):
            continue

        # Keep numeric columns with reasonable non-null counts
        if ml_df[col].dtype in ['int64', 'float64']:
            non_null_pct = ml_df[col].count() / len(ml_df)
            if non_null_pct > 0.5:  # At least 50% non-null
                remain_cols.append(col)

    # Add remaining useful columns
    keep_columns.extend(remain_cols[:5])  # Limit to 5 additional columns

    # Create final dataset
    final_df = ml_df[keep_columns].copy()
    print(f"Final ML dataset: {len(final_df)} rows, {len(keep_columns)} columns")
    print(f"Features included: {keep_columns}")

    return final_df

def preprocess_data(self):
    """
    Run the full preprocessing pipeline
    Returns:
    --------
    tuple
        (normalized_data, ml_ready_data)
    """
    # Load data
    df = self.load_data()

    # Clean and normalize
    normalized_df = self.clean_data_for_ml(df)

    # Prepare for ML
    ml_df = self.prepare_for_ml(normalized_df)

    return normalized_df, ml_df
```

```
    def save_output(self):
        """
        Save the preprocessed data to CSV files
        Returns:
        --------
        dict
            Paths to the saved files
        """
        # Run preprocessing
        normalized_df, ml_df = self.preprocess_data()

        # Generate timestamp
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

        # Save normalized data
        normalized_path = os.path.join(self.output_dir, f"ted_normalized_{timestamp}.csv")
        normalized_df.to_csv(normalized_path, index=False)
        print(f"Saved normalized data to: {normalized_path}")

        # Save ML-ready data
        ml_path = os.path.join(self.output_dir, f"ted_ml_dataset_{timestamp}.csv")
        ml_df.to_csv(ml_path, index=False)
        print(f"Saved ML-ready data to: {ml_path}")

        return {
            "normalized": normalized_path,
            "ml_dataset": ml_path,
            "ml_df": ml_df
        }
# If run directly, perform a test preprocessing
if __name__ == "__main__":
    import argparse

    # Parse arguments
    parser = argparse.ArgumentParser(description='Preprocess TED procurement data for ML')
    parser.add_argument('--input', required=True, help='Path to input CSV file')
    parser.add_argument('--output', default='processed_data', help='Directory to save output files')

    args = parser.parse_args()

    # Run preprocessing
    preprocessor = TEDDataPreprocessor(args.input, args.output)
    result = preprocessor.save_output()

    print("\nPreprocessing summary:")
    print(f"Input file: {args.input}")
    print(f"Normalized data saved to: {result['normalized']}")
    print(f"ML-ready data saved to: {result['ml_dataset']}")
    print(f"ML dataset shape: {result['ml_df'].shape}")
```

FILE: components/ted_data_retriever.py

```
#!/usr/bin/env python3
"""
TED Data Retriever Module
This module handles retrieving data from the TED API based on specified criteria.
It provides functionality to fetch procurement notices and convert them to a usable format.
Author: Your Name
Date: May 16, 2025
"""
import os
import json
import requests
import pandas as pd
import csv
import time
from datetime import datetime
class TEDDataRetriever:
    """Class for retrieving data from the TED API"""

    def __init__(self, data_dir="data"):
        self.headers = {'Content-type': 'application/json', 'Accept': 'text/plain', "Charset": "UTF-8"}
        self.base_url = "https://tedweb.api.ted.europa.eu/v3/notices/search"
        self.data_dir = data_dir

        # Ensure data directory exists
        os.makedirs(data_dir, exist_ok=True)

    def get_notices_page(self, page_number, start_date, end_date, max_bid_amount=None, country=None, limit=100):
        """
        Get a single page of notices from the TED API and return the results.
        Args:
            page_number (int): Page number to retrieve
            start_date (str): Start date in format YYYYMMDD
            end_date (str): End date in format YYYYMMDD
            max_bid_amount (float, optional): Maximum bid amount to filter by
            country (str, optional): Country code to filter by (e.g., 'GRC')
            limit (int): Number of results per page
        Returns:
            list: List of notice data dictionaries
        """
        # Use the exact same query format as in test.py
        query = f"publication-date>={start_date}<={end_date}"

        # Add country filter if specified
        if country:
            query += f" AND organisation-country-buyer={country}"

        # Prepare request parameters
        params = {
            "query": query,
            "fields": [
```

```
        "notice-identifier",
        "estimated-value-cur-lot",
        "no-negocaition-necessary-lot",
        "direct-award-justification-text-proc",
        "legal-basis",
        "procedure-identifier",
        "winner-size",
        "winner-selection-status",
        "notice-type",
        "estimated-value-cur-proc",
        "total-value",
        "framework-value-notice",
        "subcontracting-percentage",
        "subcontracting-value",
        "direct-award-justification-proc",
        "ipi-measures-applicable-lot",
        "procedure-accelerated",
        "legal-basis-proc",
        "legal-basis-text",
        "eu-registration-number",
        "exclusion-grounds",
        "framework-buyer-categories-lot",
        "dps-usage-lot",
        "accessibility-lot",
        "winner-owner-nationality",
        "organisation-country-buyer",
    ],
    "page": page_number,
    "limit": limit
}

# Prepare request parameters
params = {
    "query": query,
    "fields": [
        "notice-identifier",
        "estimated-value-cur-lot",
        "no-negocaition-necessary-lot",
        "direct-award-justification-text-proc",
        "legal-basis",
        "procedure-identifier",
        "winner-size",
        "winner-selection-status",
        "notice-type",
        "estimated-value-cur-proc",
        "total-value",
        "framework-value-notice",
        "subcontracting-percentage",
        "subcontracting-value",
        "direct-award-justification-proc",
        "ipi-measures-applicable-lot",
        "procedure-accelerated",
```

```
            "legal-basis-proc",
            "legal-basis-text",
            "eu-registration-number",
            "exclusion-grounds",
            "framework-buyer-categories-lot",
            "dps-usage-lot",
            "accessibility-lot",
            "winner-owner-nationality",
            "organisation-country-buyer",
        ],
        "page": page_number,
        "limit": limit
    }

    print(f"Making API request for page {page_number}...")
    try:
        response = requests.post(self.base_url, json=params, headers=self.headers)
        if response.status_code == 200:
            data = json.loads(response.text)
            notices = data.get("notices", [])
            print(f"Successfully retrieved {len(notices)} notices from page {page_number}")

            # Filter by max_bid_amount if specified
            if max_bid_amount is not None and notices:
                filtered_notices = []
                for notice in notices:
                    total_value = notice.get("total-value", 0)
                    if total_value is None or float(total_value or 0) <= float(max_bid_amount):
                        filtered_notices.append(notice)
                print(f"Filtered to {len(filtered_notices)} notices within budget {max_bid_amount}")
                return filtered_notices
            return notices
        else:
            print(f"Error on page {page_number}: {response.status_code}")
            print(response.text)
            return []
    except Exception as e:
        print(f"Exception during API request: {str(e)}")
        return []
def flatten_notice(self, notice):
    """
    Flatten a nested notice structure into a single-level dictionary.
    Args:
        notice (dict): Notice data dictionary
    Returns:
        dict: Flattened notice dictionary
    """
    flat_notice = {}
    for key, value in notice.items():
        if isinstance(value, dict):
            # Flatten nested dictionaries with dot notation
            for nested_key, nested_value in value.items():
```

```
                flat_notice[f"{key}.{nested_key}"] = nested_value
        elif isinstance(value, list):
            # Join list values with a separator
            flat_notice[key] = "|".join(str(item) for item in value)
        else:
            flat_notice[key] = value
    return flat_notice

def save_to_csv(self, df, timestamp=None):
    """
    Save DataFrame to CSV with timestamp
    Args:
        df (pd.DataFrame): DataFrame to save
        timestamp (str, optional): Timestamp to use in filename, defaults to current time
    Returns:
        str: Path to saved file
    """
    if timestamp is None:
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

    output_file = os.path.join(self.data_dir, f"ted_notices_{timestamp}.csv")
    df.to_csv(output_file, index=False)
    print(f"Raw data saved to {output_file}")

    return output_file

def fetch_notices(self, start_date, end_date, max_bid_amount=None, country=None, max_pages=5):
    """
    Fetch notices from the TED API based on specified criteria
    Args:
        start_date (str): Start date in format YYYYMMDD
        end_date (str): End date in format YYYYMMDD
        max_bid_amount (float, optional): Maximum bid amount
        country (str, optional): Country code
        max_pages (int): Maximum number of pages to fetch
    Returns:
        tuple: (pd.DataFrame, str) - DataFrame containing notices and path to saved CSV
    """
    all_notices = []

    # Process pages one by one
    for page_num in range(1, max_pages + 1):
        notices = self.get_notices_page(
            page_num,
            start_date,
            end_date,
            max_bid_amount,
            country
        )

        if not notices:
            print(f"No notices found on page {page_num}, stopping pagination")
```

```
                break

            # Flatten notices and add to the list
            flattened_notices = [self.flatten_notice(notice) for notice in notices]
            all_notices.extend(flattened_notices)

            # Add a delay between requests to avoid rate limiting
            time.sleep(1)

        if not all_notices:
            print("No notices were found with the specified criteria")
            return pd.DataFrame(), None

        # Convert to DataFrame
        df = pd.DataFrame(all_notices)
        print(f"Successfully fetched {len(df)} notices")

        # Save raw data to CSV
        output_file = self.save_to_csv(df)

        return df, output_file
# If run directly, perform a test fetch
if __name__ == "__main__":
    # Example usage
    retriever = TEDDataRetriever()

    # Fetch notices for the last month
    today = datetime.now()
    end_date = today.strftime("%Y%m%d")
    start_date = (today.replace(day=1) - pd.DateOffset(months=1)).strftime("%Y%m%d")

    print(f"Fetching notices from {start_date} to {end_date}")
    df, output_file = retriever.fetch_notices(
        start_date=start_date,
        end_date=end_date,
        max_pages=2
    )

    if not df.empty:
        print(f"Retrieved {len(df)} notices")
        print(f"Sample columns: {', '.join(df.columns[:5])}")
        print("\nSample data:")
        print(df.head(2))
    else:
        print("No data retrieved")
```

FILE: data/processed/metadata.json


```
{
    "last_update": "2025-05-21 13:25:46",
    "total_records": 446,
```

```
    "data_files": [
        "ted_ml_dataset_20250521_132539_20250521_132539.csv",
        "ted_ml_dataset_20250521_132546_20250521_132546.csv"
    ],
    "models": []
}
```

Summary

Files processed: 13