Project Files Content

FILE: ted_ml_pipeline.py

```
#!/usr/bin/env python3
"""
TED ML Pipeline
This script provides the main entry point for the TED procurement data processing and outlier detection.
It coordinates the different stages of the pipeline: data fetching, preprocessing, model training/prediction.
Author: Your Name
Date: May 16, 2025
"""
import os
import sys
import argparse
import pandas as pd
from datetime import datetime
# Import pipeline components from correct locations
from components.ted_data_retriever import TEDDataRetriever
from components.ted_data_preprocessor import TEDDataPreprocessor
from transforming.isolation_forest_model import IsolationForestModel
class TEDMLPipeline:
    """Main class for the TED procurement data processing and outlier detection pipeline"""

    def __init__(self, base_dir="."):
        self.base_dir = base_dir
        self.raw_data_dir = os.path.join(base_dir, "data")
        self.processed_data_dir = os.path.join(base_dir, "data")
        self.model_dir = os.path.join(base_dir, "models")
        self.output_dir = os.path.join(base_dir, "output")
        self.training_data_file = os.path.join(base_dir, "data", "training_data.csv")

        # Ensure directories exist
        for directory in [self.raw_data_dir, self.processed_data_dir,
                          self.model_dir, self.output_dir]:
            os.makedirs(directory, exist_ok=True)

        self.model = None

    def train(self, input_file=None, sample_size=None, contamination=0.05):
        """
        Run the training pipeline:
        1. Use the specified training data file or default to training_data.csv
        2. Train the model
        3. Save the model
        """
        print("=== Starting Training Pipeline ===")

        # Step 1: Determine which training file to use
        if input_file:
            # Use the file specified by the user
```

```
            training_file = input_file
            print(f"\nUsing specified training file: {training_file}")
        else:
            # Use the default training file
            training_file = self.training_data_file
            print(f"\nUsing default training file: {training_file}")

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

        # Step 2: Train the model
        print("\nStep 2: Training the model...")
        self.model = IsolationForestModel(
            model_path=os.path.join(self.model_dir, "isolation_forest_model.pkl"),
            contamination=contamination
        )

        # Load the training data
        print(f"Loading training data from: {training_file}")
        ml_df = pd.read_csv(training_file)
        print(f"Loaded training dataset with {len(ml_df)} rows and {len(ml_df.columns)} columns")

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

        # Step 3: Save the model
        print("\nStep 3: Saving trained model...")
        self.model.save_model()

        print("\nTraining completed successfully.")
        return self.model.model_path

    def predict(self, country=None, start_date=None, end_date=None, max_bid_amount=None):
        """
        Run the prediction pipeline:
        1. Retrieve data from TED API based on filters
        2. Preprocess the data
        3. Load the trained model
        4. Make predictions
        5. Save the results to CSV (no visualizations)
        """
        print("=== Starting Prediction Pipeline ===")
        print(f"Date range: {start_date} to {end_date}")
        print(f"Max bid amount: {max_bid_amount}")
        print(f"Country filter: {country}")

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

        # Step 1: Retrieve data from TED API
```

```
    print("\nStep 1: Retrieving data from TED API...")
    data_retriever = TEDDataRetriever(data_dir=self.raw_data_dir)

    # Get data from TED API
    _, 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

    # Step 2: Preprocess the data
    print("\nStep 2: Preprocessing retrieved data...")
    preprocessor = TEDDataPreprocessor(
        input_file=raw_data_file,
        output_dir=self.processed_data_dir
    )

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

    # Step 3: Load the trained model
    print("\nStep 3: Loading trained model...")
    self.model = IsolationForestModel(
        model_path=os.path.join(self.model_dir, "isolation_forest_model.pkl")
    )

    self.model.load_model()

    # Step 4: Making predictions
    print("\nStep 4: Making predictions...")
    ml_df = pd.read_csv(ml_dataset_file)
    result_df = self.model.predict(ml_df)

    # Step 5: Save results to CSV (no visualizations)
    print("\nStep 5: Saving results...")

    # Save predictions to CSV
    csv_path = os.path.join(self.output_dir, f"outliers_{timestamp}.csv")
    self.model.save_predictions(result_df, csv_path)

    print("\nPrediction completed successfully.")
    return csv_path

def evaluate(self, input_file=None):
    """
    Run the evaluation pipeline:
```

```
        1. Load the data
        2. Load the trained model
        3. Make predictions and evaluate performance
        """
        print("=== Starting Evaluation Pipeline ===")

        # Not implemented yet
        print("Evaluation pipeline not implemented yet.")
        return None
def main():
    # Create argument parser
    parser = argparse.ArgumentParser(description='TED ML Pipeline for procurement outlier detection')

    # Create 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 new model')
    train_parser.add_argument('--input', type=str, help='Path to input CSV file (default: ./data/training_data.csv)')
    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='Make predictions on new data')
    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('--max_bid_amount', type=float, help='Maximum bid amount')
    predict_parser.add_argument('--country', type=str, help='Country code (ISO)')

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

    # Parse arguments
    args = parser.parse_args()

    # Create pipeline instance
    pipeline = TEDMLPipeline()

    # Execute command
    if args.command == 'train':
        pipeline.train(
            input_file=args.input,
            sample_size=args.sample,
            contamination=args.contamination
        )
    elif args.command == 'predict':
        pipeline.predict(
            country=args.country,
            start_date=args.start_date,
            end_date=args.end_date,
```

```
            max_bid_amount=args.max_bid_amount
        )
    elif args.command == 'evaluate':
        pipeline.evaluate(input_file=args.input)
    else:
        parser.print_help()
        sys.exit(1)
if __name__ == "__main__":
    main()
```

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: transforming/requirements.txt

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_cou
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_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")
```

Summary

Files processed: 6