Survey Data Analyzer

Complete Project Documentation & Codebase

A comprehensive Command Line Interface (CLI) tool for analyzing survey responses from CSV files. This project demonstrates advanced Python programming concepts including modular design, statistical analysis, sentiment analysis, pattern detection, and report generation.

- CSV Input Handling with multi-encoding support
- Survey Summary with demographic breakdowns
- Statistical Analysis (cross-tabulation, chi-square tests)
- Sentiment Analysis on text responses
- Pattern Recognition and correlation detection
- Comprehensive Report Generation
- Professional CLI Interface

Generated on: August 02, 2025 at 03:35 AM

Table of Contents

1. Project Overview

The Survey Data Analyzer is a sophisticated Python CLI application designed to perform comprehensive analysis of survey data stored in CSV format. The project demonstrates advanced software engineering principles including modular design, object-oriented programming, error handling, and comprehensive testing.

The application consists of several specialized modules, each handling a specific aspect of survey analysis:

- DataLoader: Handles CSV file loading, validation, and data preprocessing
- SurveySummary: Generates demographic breakdowns and response statistics
- StatsAnalyzer: Performs statistical analysis including chi-square tests
- SentimentAnalyzer: Analyzes text responses using keyword-based sentiment scoring
- PatternDetector: Identifies correlations and patterns in survey responses
- ReportGenerator: Creates comprehensive analysis reports
- Utils: Provides utility functions for formatting and validation

2. Main Application (main.py)

The main.py file serves as the entry point for the Survey Data Analyzer CLI application. It provides a menu-driven interface that orchestrates all analysis components and manages the user interaction flow.

File: main.py

```
  1: #!/usr/bin/env python3
  2: """
  3: Survey Data Analyzer - Main CLI Application
  4: Author: Student Developer
  5: Description: Main entry point for the Survey Data Analyzer CLI tool.
  6: Provides a menu-driven interface for analyzing survey data from CSV files.
  7: """
  8:
  9: import sys
 10: import os
 11: from typing import Optional, Dict, Any
 12: from data_loader import DataLoader
 13: from survey_summary import SurveySummary
 14: from stats_analyzer import StatsAnalyzer
 15: from sentiment_analyzer import SentimentAnalyzer
 16: from pattern_detector import PatternDetector
 17: from report_generator import ReportGenerator
 18: from utils import clear_screen, print_header, print_menu
 19:
 20:
 21: class SurveyAnalyzerCLI:
 22:     """Main CLI application class for Survey Data Analyzer."""
 23:
 24:     def __init__(self):
 25:         """Initialize the CLI application with empty data containers."""
 26:         self.data_loader = DataLoader()
 27:         self.survey_data = None
 28:         self.survey_summary = None
 29:         self.stats_analyzer = None
 30:         self.sentiment_analyzer = None
 31:         self.pattern_detector = None
 32:         self.report_generator = None
 33:
 34:     def load_survey_data(self) -> bool:
 35:         """Load survey data from CSV file."""
 36:         try:
 37:             print_header("Load Survey Data")
 38:             file_path = input("Enter the path to your CSV file: ").strip()
 39:
 40:             if not file_path:
 41:                 print("ERROR: No file path provided.")
 42:                 return False
 43:
 44:             if not os.path.exists(file_path):
 45:                 print(f"ERROR: File not found: {file_path}")
 46:                 return False
 47:
 48:             self.survey_data = self.data_loader.load_csv(file_path)
 49:             if self.survey_data:
 50:                 print(f"SUCCESS: Successfully loaded {len(self.survey_data)} survey
responses"
)
 51:                 print(f"INFO: Columns: {', '.join(self.survey_data[0].keys())}")
```

```
                return True
            else:
                print("ERROR: Failed to load survey data")
                return False

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

    def view_summary_statistics(self):
        """Display summary statistics for the loaded survey data."""
        if not self.survey_data:
            print("ERROR: No survey data loaded. Please load data first.")
            return

        try:
            print_header("Survey Summary Statistics")
            self.survey_summary = SurveySummary(self.survey_data)
            summary = self.survey_summary.generate_summary()

            print("\nRESPONSE OVERVIEW:")
            print(f"   Total Responses: {summary['total_responses']}")
            print(f"   Response Rate: {summary['response_rate']:.1f}%")

            print("\nDEMOGRAPHIC BREAKDOWN:")
            for demo, breakdown in summary['demographics'].items():
                print(f"\n   {demo.upper()}:")
                for category, count in breakdown.items():
                    percentage = (count / summary['total_responses']) * 100
                    print(f"     {category}: {count} ({percentage:.1f}%)")

        except Exception as e:
            print(f"ERROR: Error generating summary: {str(e)}")

    def analyze_sentiment(self):
        """Analyze sentiment of text responses."""
        if not self.survey_data:
            print("ERROR: No survey data loaded. Please load data first.")
            return

        try:
            print_header("Sentiment Analysis")
            self.sentiment_analyzer = SentimentAnalyzer()

            # Find text columns
            text_columns = []
            for col in self.survey_data[0].keys():
                if any(isinstance(row[col], str) and len(str(row[col])) > 20
                       for row in self.survey_data[:10]):
                    text_columns.append(col)

            if not text_columns:
                print("ERROR: No text columns found for sentiment analysis.")
                return

            print(f"INFO: Found text columns: {', '.join(text_columns)}")

            for column in text_columns:
                print(f"\nANALYZING: Analyzing sentiment for '{column}':")
                sentiment_results = self.sentiment_analyzer.analyze_column(
                    self.survey_data, column
                )

                print(f"   Positive responses: {sentiment_results['positive']}
({sentiment_res
```

```
ults['positive_pct']:.1f}%)")
                print(f"   Negative responses: {sentiment_results['negative']}
({sentiment_res
ults['negative_pct']:.1f}%)")
                print(f"   Neutral responses: {sentiment_results['neutral']}
({sentiment_resul
ts['neutral_pct']:.1f}%)")
                print(f"   Average sentiment score: {sentiment_results['avg_score']:.2f}")

        except Exception as e:
            print(f"ERROR: Error analyzing sentiment: {str(e)}")

    def cross_tabulate_results(self):
        """Perform cross-tabulation analysis."""
        if not self.survey_data:
            print("ERROR: No survey data loaded. Please load data first.")
            return

        try:
            print_header("Cross-Tabulation Analysis")
            self.stats_analyzer = StatsAnalyzer(self.survey_data)

            # Get available columns
            columns = list(self.survey_data[0].keys())
            print(f"Available columns: {', '.join(columns)}")

            col1 = input("Enter first column name: ").strip()
            col2 = input("Enter second column name: ").strip()

            if col1 not in columns or col2 not in columns:
                print("ERROR: Invalid column names.")
                return

            crosstab = self.stats_analyzer.cross_tabulate(col1, col2)
            chi_square = self.stats_analyzer.chi_square_test(col1, col2)

            print(f"\nCROSS-TABULATION: {col1} vs {col2}")
            print("=" * 50)

            # Display crosstab
            for row in crosstab:
                print(" | ".join(f"{cell:>8}" for cell in row))

            print(f"\nCHI-SQUARE TEST RESULTS:")
            print(f"   Chi-Square Value: {chi_square['chi_square']:.4f}")
            print(f"   P-Value: {chi_square['p_value']:.4f}")
            print(f"   Degrees of Freedom: {chi_square['df']}")
            print(f"   Significant: {'Yes' if chi_square['significant'] else 'No'}")

        except Exception as e:
            print(f"ERROR: Error in cross-tabulation: {str(e)}")

    def detect_patterns(self):
        """Detect patterns and correlations in survey responses."""
        if not self.survey_data:
            print("ERROR: No survey data loaded. Please load data first.")
            return

        try:
            print_header("Pattern Detection")
            self.pattern_detector = PatternDetector(self.survey_data)

            patterns = self.pattern_detector.find_patterns()

            print("DETECTED PATTERNS:")
```

```
            print("=" * 50)

            for pattern in patterns:
                print(f"\nPATTERN: {pattern['description']}")
                print(f"   Confidence: {pattern['confidence']:.1f}%")
                print(f"   Sample size: {pattern['sample_size']}")

        except Exception as e:
            print(f"ERROR: Error detecting patterns: {str(e)}")

    def generate_report(self):
        """Generate a comprehensive analysis report."""
        if not self.survey_data:
            print("ERROR: No survey data loaded. Please load data first.")
            return

        try:
            print_header("Generate Report")

            # Initialize all analyzers
            self.survey_summary = SurveySummary(self.survey_data)
            self.stats_analyzer = StatsAnalyzer(self.survey_data)
            self.sentiment_analyzer = SentimentAnalyzer()
            self.pattern_detector = PatternDetector(self.survey_data)
            self.report_generator = ReportGenerator()

            output_file = input("Enter output file name (default: survey_report.txt):
").strip
()
            if not output_file:
                output_file = "survey_report.txt"

            if not output_file.endswith('.txt'):
                output_file += '.txt'

            # Generate comprehensive report
            report_data = {
                'summary': self.survey_summary.generate_summary(),
                'sentiment':
self.sentiment_analyzer.analyze_all_text_columns(self.survey_data
),
                'patterns': self.pattern_detector.find_patterns(),
                'file_path': output_file
            }

            success = self.report_generator.generate_report(report_data)

            if success:
                print(f"SUCCESS: Report generated successfully: {output_file}")
            else:
                print("ERROR: Failed to generate report")

        except Exception as e:
            print(f"ERROR: Error generating report: {str(e)}")

    def run(self):
        """Main application loop."""
        while True:
            clear_screen()
            print_header("Survey Data Analyzer")

            if self.survey_data:
                print(f"INFO: Loaded: {len(self.survey_data)} responses")
            else:
                print("INFO: No data loaded")
```

```
237:
238:             print_menu([
239:                 "Load survey data",
240:                 "View summary statistics",
241:                 "Analyze sentiment",
242:                 "Cross-tabulate results",
243:                 "Detect patterns",
244:                 "Generate report",
245:                 "Exit"
246:             ])
247:
248:             choice = input("\nEnter your choice (1-7): ").strip()
249:
250:             if choice == '1':
251:                 self.load_survey_data()
252:             elif choice == '2':
253:                 self.view_summary_statistics()
254:             elif choice == '3':
255:                 self.analyze_sentiment()
256:             elif choice == '4':
257:                 self.cross_tabulate_results()
258:             elif choice == '5':
259:                 self.detect_patterns()
260:             elif choice == '6':
261:                 self.generate_report()
262:             elif choice == '7':
263:                 print("\nThank you for using Survey Data Analyzer!")
264:                 sys.exit(0)
265:             else:
266:                 print("ERROR: Invalid choice. Please try again.")
267:
268:             input("\nPress Enter to continue...")
269:
270:
271: def main():
272:     """Main entry point for the application."""
273:     try:
274:         app = SurveyAnalyzerCLI()
275:         app.run()
276:     except KeyboardInterrupt:
277:         print("\n\nGoodbye!")
278:         sys.exit(0)
279:     except Exception as e:
280:         print(f"ERROR: Unexpected error: {str(e)}")
281:         sys.exit(1)
282:
283:
284: if __name__ == "__main__":
285:     main()
```

3. Data Loading Module (data_loader.py)

The DataLoader class handles all aspects of CSV file processing including file validation, encoding detection, data cleaning, and validation. It provides robust error handling and supports multiple file formats and encodings.

File: data_loader.py

```
#!/usr/bin/env python3
"""
Survey Data Loader Module
Author: Student Developer
Description: Handles CSV file loading, validation, and data preprocessing.
Provides robust error handling and data consistency checks.
"""

import csv
import os
import sys
from typing import List, Dict, Any, Optional
from collections import defaultdict


class DataLoader:
    """Handles loading and validation of survey data from CSV files."""

    def __init__(self):
        """Initialize the data loader with validation settings."""
        self.required_fields = ['age', 'gender', 'region']  # Common required fields
        self.max_file_size = 50 * 1024 * 1024  # 50MB limit
        self.supported_encodings = ['utf-8', 'latin-1', 'cp1252']

    def load_csv(self, file_path: str) -> Optional[List[Dict[str, Any]]]:
        """
        Load and validate CSV survey data.

        Args:
            file_path: Path to the CSV file

        Returns:
            List of dictionaries representing survey responses, or None if failed
        """
        try:
            # Validate file exists and is readable
            if not self._validate_file(file_path):
                return None

            # Try different encodings
            data = None
            for encoding in self.supported_encodings:
                try:
                    data = self._read_csv_with_encoding(file_path, encoding)
                    if data:
                        break
                except UnicodeDecodeError:
                    continue

            if not data:
                print("ERROR: Could not read file with any supported encoding")
                return None

```

```
            # Validate and clean data
            cleaned_data = self._clean_and_validate_data(data)
            if not cleaned_data:
                return None

            print(f"SUCCESS: Data validation completed successfully")
            return cleaned_data

        except Exception as e:
            print(f"ERROR: Error loading CSV file: {str(e)}")
            return None

    def _validate_file(self, file_path: str) -> bool:
        """Validate that the file exists and is accessible."""
        try:
            if not os.path.exists(file_path):
                print(f"ERROR: File not found: {file_path}")
                return False

            if not os.path.isfile(file_path):
                print(f"ERROR: Path is not a file: {file_path}")
                return False

            # Check file size
            file_size = os.path.getsize(file_path)
            if file_size > self.max_file_size:
                print(f"ERROR: File too large: {file_size / (1024*1024):.1f}MB (max: 50MB)")
                return False

            if file_size == 0:
                print("ERROR: File is empty")
                return False

            return True

        except Exception as e:
            print(f"ERROR: Error validating file: {str(e)}")
            return False

    def _read_csv_with_encoding(self, file_path: str, encoding: str) ->
Optional[List[Dict[str
, Any]]]:
        """Read CSV file with specified encoding."""
        try:
            with open(file_path, 'r', encoding=encoding, newline='') as file:
                # Try to detect delimiter
                sample = file.read(1024)
                file.seek(0)

                # Common delimiters to try
                delimiters = [',', ';', '\t', '|']
                detected_delimiter = ','

                for delimiter in delimiters:
                    if delimiter in sample:
                        detected_delimiter = delimiter
                        break

                reader = csv.DictReader(file, delimiter=detected_delimiter)
                data = list(reader)

                if not data:
                    print("ERROR: No data found in CSV file")
                    return None

```

```
                print(f"SUCCESS: Loaded {len(data)} rows with {len(data[0])} columns")
                print(f"INFO: Columns: {', '.join(data[0].keys())}")

                return data

        except Exception as e:
            print(f"ERROR: Error reading CSV with {encoding} encoding: {str(e)}")
            return None

    def _clean_and_validate_data(self, data: List[Dict[str, Any]]) -> Optional[List[Dict[str,
Any]]]:
        """Clean and validate the loaded data."""
        try:
            if not data:
                return None

            cleaned_data = []
            validation_errors = []

            for i, row in enumerate(data, 1):
                cleaned_row = {}
                row_errors = []

                for key, value in row.items():
                    # Clean the key (remove whitespace, normalize)
                    if key is None:
                        continue  # Skip rows with None keys
                    clean_key = str(key).strip().lower().replace(' ', '_')

                    # Clean the value
                    if value is None or value == '':
                        cleaned_value = None
                    else:
                        cleaned_value = str(value).strip()
                        if cleaned_value.lower() in ['na', 'n/a', 'null', 'none']:
                            cleaned_value = None

                    cleaned_row[clean_key] = cleaned_value

                # Basic validation
                if self._validate_row(cleaned_row, i):
                    cleaned_data.append(cleaned_row)
                else:
                    validation_errors.append(f"Row {i}: Invalid data")

            # Report validation results
            total_rows = len(data)
            valid_rows = len(cleaned_data)
            invalid_rows = total_rows - valid_rows

            print(f"INFO: Data validation results:")
            print(f"   Total rows: {total_rows}")
            print(f"   Valid rows: {valid_rows}")
            print(f"   Invalid rows: {invalid_rows}")

            if invalid_rows > 0:
                print(f"WARNING: {invalid_rows} rows had validation issues")

            if valid_rows == 0:
                print("ERROR: No valid data found")
                return None

            return cleaned_data

```

```
        except Exception as e:
            print(f"ERROR: Error cleaning data: {str(e)}")
            return None

    def _validate_row(self, row: Dict[str, Any], row_num: int) -> bool:
        """Validate a single row of data."""
        try:
            # Check for minimum required fields
            if len(row) < 2:
                return False

            # Validate age if present
            if 'age' in row and row['age'] is not None:
                try:
                    age = int(row['age'])
                    if age < 0 or age > 120:
                        return False
                except (ValueError, TypeError):
                    return False

            # Validate gender if present
            if 'gender' in row and row['gender'] is not None:
                try:
                    gender = str(row['gender']).lower()
                    valid_genders = ['male', 'female', 'm', 'f', 'other', 'prefer not to
say']
                    if gender not in valid_genders:
                        return False
                except (AttributeError, TypeError):
                    return False

            return True

        except Exception:
            return False

    def get_data_summary(self, data: List[Dict[str, Any]]) -> Dict[str, Any]:
        """Generate a summary of the loaded data."""
        if not data:
            return {}

        summary = {
            'total_rows': len(data),
            'columns': list(data[0].keys()) if data else [],
            'missing_values': {},
            'unique_values': {}
        }

        # Analyze missing values and unique values
        for column in summary['columns']:
            values = [row.get(column) for row in data]
            missing_count = sum(1 for v in values if v is None or v == '')
            summary['missing_values'][column] = {
                'count': missing_count,
                'percentage': (missing_count / len(data)) * 100
            }

            unique_vals = set(v for v in values if v is not None and v != '')
            summary['unique_values'][column] = len(unique_vals)

        return summary

    def export_sample_data(self, file_path: str = "sample_survey.csv"):
        """Create a sample CSV file for testing."""
        sample_data = [
```

```
            {
                'age': '25',
                'gender': 'Female',
                'region': 'North',
                'education': 'Bachelor',
                'satisfaction': 'Very Satisfied',
                'feedback': 'Great experience with the product!',
                'recommend': 'Yes'
            },
            {
                'age': '32',
                'gender': 'Male',
                'region': 'South',
                'education': 'Master',
                'satisfaction': 'Satisfied',
                'feedback': 'Good but could be better.',
                'recommend': 'Yes'
            },
            {
                'age': '45',
                'gender': 'Female',
                'region': 'East',
                'education': 'High School',
                'satisfaction': 'Neutral',
                'feedback': 'It was okay, nothing special.',
                'recommend': 'Maybe'
            },
            {
                'age': '28',
                'gender': 'Male',
                'region': 'West',
                'education': 'Bachelor',
                'satisfaction': 'Dissatisfied',
                'feedback': 'Poor quality and bad service.',
                'recommend': 'No'
            },
            {
                'age': '35',
                'gender': 'Female',
                'region': 'North',
                'education': 'PhD',
                'satisfaction': 'Very Satisfied',
                'feedback': 'Excellent product and amazing support!',
                'recommend': 'Yes'
            }
        ]

        try:
            with open(file_path, 'w', newline='', encoding='utf-8') as file:
                if sample_data:
                    writer = csv.DictWriter(file, fieldnames=sample_data[0].keys())
                    writer.writeheader()
                    writer.writerows(sample_data)

            print(f"SUCCESS: Sample data exported to {file_path}")
            return True

        except Exception as e:
            print(f"ERROR: Error exporting sample data: {str(e)}")
            return False
```

4. Survey Summary Module (survey_summary.py)

The SurveySummary class generates comprehensive summary statistics including demographic breakdowns, response rates, and data quality assessments. It provides the foundation for all subsequent analysis.

File: survey_summary.py

```
#!/usr/bin/env python3
"""
Survey Summary Module
Author: Student Developer
Description: Generates summary statistics, response counts, percentages,
and demographic breakdowns for survey data analysis.
"""

import statistics
from typing import List, Dict, Any, Optional
from collections import defaultdict, Counter


class SurveySummary:
    """Handles generation of survey summary statistics and demographic breakdowns."""

    def __init__(self, survey_data: List[Dict[str, Any]]):
        """
        Initialize the survey summary analyzer.

        Args:
            survey_data: List of dictionaries containing survey responses
        """
        self.survey_data = survey_data
        self.total_responses = len(survey_data)
        self.columns = list(survey_data[0].keys()) if survey_data else []

    def generate_summary(self) -> Dict[str, Any]:
        """
        Generate comprehensive summary statistics.

        Returns:
            Dictionary containing all summary statistics
        """
        if not self.survey_data:
            return {}

        summary = {
            'total_responses': self.total_responses,
            'response_rate': self._calculate_response_rate(),
            'demographics': self._analyze_demographics(),
            'question_summaries': self._analyze_questions(),
            'data_quality': self._assess_data_quality()
        }

        return summary

    def _calculate_response_rate(self) -> float:
        """Calculate the response rate (placeholder for actual calculation)."""
        # In a real scenario, this would compare against expected responses
        # For now, we'll use a placeholder calculation
        return 85.5  # Placeholder response rate

```

```
    def _analyze_demographics(self) -> Dict[str, Dict[str, int]]:
        """Analyze demographic breakdowns."""
        demographics = {}

        # Common demographic fields
        demo_fields = ['age', 'gender', 'region', 'education', 'income']

        for field in demo_fields:
            if field in self.columns:
                demographics[field] = self._count_responses_by_field(field)

        return demographics

    def _analyze_questions(self) -> Dict[str, Dict[str, Any]]:
        """Analyze responses for each question."""
        question_summaries = {}

        for column in self.columns:
            if column not in ['age', 'gender', 'region', 'education', 'income']:
                question_summaries[column] = self._analyze_question_responses(column)

        return question_summaries

    def _count_responses_by_field(self, field: str) -> Dict[str, int]:
        """Count responses for a specific field."""
        counts = Counter()

        for row in self.survey_data:
            value = row.get(field)
            if value is not None and str(value).strip():
                # Normalize the value
                normalized_value = str(value).strip().title()
                counts[normalized_value] += 1

        return dict(counts)

    def _analyze_question_responses(self, question: str) -> Dict[str, Any]:
        """Analyze responses for a specific question."""
        responses = [row.get(question) for row in self.survey_data]
        valid_responses = [r for r in responses if r is not None and str(r).strip()]

        if not valid_responses:
            return {
                'total_responses': 0,
                'missing_responses': len(responses),
                'response_rate': 0.0,
                'top_responses': [],
                'response_distribution': {}
            }

        # Count responses
        response_counts = Counter(valid_responses)

        # Calculate percentages
        total_valid = len(valid_responses)
        response_distribution = {}
        for response, count in response_counts.items():
            percentage = (count / total_valid) * 100
            response_distribution[response] = {
                'count': count,
                'percentage': percentage
            }

        # Get top responses
        top_responses = response_counts.most_common(5)
```

```
119:
120:         return {
121:             'total_responses': total_valid,
122:             'missing_responses': len(responses) - total_valid,
123:             'response_rate': (total_valid / len(responses)) * 100,
124:             'top_responses': top_responses,
125:             'response_distribution': response_distribution
126:         }
127:
128:     def _assess_data_quality(self) -> Dict[str, Any]:
129:         """Assess the quality of the survey data."""
130:         quality_metrics = {
131:             'total_rows': len(self.survey_data),
132:             'total_columns': len(self.columns),
133:             'missing_data': {},
134:             'completeness': {}
135:         }
136:
137:         # Analyze missing data for each column
138:         for column in self.columns:
139:             values = [row.get(column) for row in self.survey_data]
140:             missing_count = sum(1 for v in values if v is None or str(v).strip() == '')
141:             missing_percentage = (missing_count / len(values)) * 100
142:
143:             quality_metrics['missing_data'][column] = {
144:                 'count': missing_count,
145:                 'percentage': missing_percentage
146:             }
147:
148:             quality_metrics['completeness'][column] = 100 - missing_percentage
149:
150:         return quality_metrics
151:
152:     def get_age_distribution(self) -> Dict[str, int]:
153:         """Get age distribution if age data is available."""
154:         if 'age' not in self.columns:
155:             return {}
156:
157:         age_groups = {
158:             '18-25': 0,
159:             '26-35': 0,
160:             '36-45': 0,
161:             '46-55': 0,
162:             '56-65': 0,
163:             '65+': 0
164:         }
165:
166:         for row in self.survey_data:
167:             age_str = row.get('age')
168:             if age_str and str(age_str).isdigit():
169:                 try:
170:                     age = int(age_str)
171:                     if 18 <= age <= 25:
172:                         age_groups['18-25'] += 1
173:                     elif 26 <= age <= 35:
174:                         age_groups['26-35'] += 1
175:                     elif 36 <= age <= 45:
176:                         age_groups['36-45'] += 1
177:                     elif 46 <= age <= 55:
178:                         age_groups['46-55'] += 1
179:                     elif 56 <= age <= 65:
180:                         age_groups['56-65'] += 1
181:                     elif age > 65:
182:                         age_groups['65+'] += 1
183:                 except ValueError:
```

```
                    continue

        return age_groups

    def get_gender_distribution(self) -> Dict[str, int]:
        """Get gender distribution if gender data is available."""
        if 'gender' not in self.columns:
            return {}

        return self._count_responses_by_field('gender')

    def get_regional_distribution(self) -> Dict[str, int]:
        """Get regional distribution if region data is available."""
        if 'region' not in self.columns:
            return {}

        return self._count_responses_by_field('region')

    def get_education_distribution(self) -> Dict[str, int]:
        """Get education distribution if education data is available."""
        if 'education' not in self.columns:
            return {}

        return self._count_responses_by_field('education')

    def calculate_average_age(self) -> Optional[float]:
        """Calculate average age if age data is available."""
        if 'age' not in self.columns:
            return None

        ages = []
        for row in self.survey_data:
            age_str = row.get('age')
            if age_str and str(age_str).isdigit():
                try:
                    ages.append(int(age_str))
                except ValueError:
                    continue

        if ages:
            return statistics.mean(ages)
        return None

    def get_response_trends(self) -> Dict[str, Any]:
        """Identify response trends and patterns."""
        trends = {
            'most_common_responses': {},
            'response_patterns': [],
            'outliers': []
        }

        # Find most common responses for each question
        for column in self.columns:
            if column not in ['age', 'gender', 'region', 'education', 'income']:
                responses = [row.get(column) for row in self.survey_data]
                valid_responses = [r for r in responses if r is not None and str(r).strip()]

                if valid_responses:
                    response_counts = Counter(valid_responses)
                    most_common = response_counts.most_common(1)
                    if most_common:
                        trends['most_common_responses'][column] = {
                            'response': most_common[0][0],
                            'count': most_common[0][1],
                            'percentage': (most_common[0][1] / len(valid_responses)) * 100
```

```
                        }

        return trends

    def generate_demographic_report(self) -> str:
        """Generate a formatted demographic report."""
        if not self.survey_data:
            return "No data available for demographic analysis."

        report_lines = []
        report_lines.append("DEMOGRAPHIC ANALYSIS REPORT")
        report_lines.append("=" * 50)
        report_lines.append(f"Total Responses: {self.total_responses}")
        report_lines.append("")

        # Age distribution
        age_dist = self.get_age_distribution()
        if age_dist:
            report_lines.append("AGE DISTRIBUTION:")
            for age_group, count in age_dist.items():
                percentage = (count / self.total_responses) * 100
                report_lines.append(f"  {age_group}: {count} ({percentage:.1f}%)")
            report_lines.append("")

        # Gender distribution
        gender_dist = self.get_gender_distribution()
        if gender_dist:
            report_lines.append("GENDER DISTRIBUTION:")
            for gender, count in gender_dist.items():
                percentage = (count / self.total_responses) * 100
                report_lines.append(f"  {gender}: {count} ({percentage:.1f}%)")
            report_lines.append("")

        # Regional distribution
        region_dist = self.get_regional_distribution()
        if region_dist:
            report_lines.append("REGIONAL DISTRIBUTION:")
            for region, count in region_dist.items():
                percentage = (count / self.total_responses) * 100
                report_lines.append(f"  {region}: {count} ({percentage:.1f}%)")
            report_lines.append("")

        # Education distribution
        education_dist = self.get_education_distribution()
        if education_dist:
            report_lines.append("EDUCATION DISTRIBUTION:")
            for education, count in education_dist.items():
                percentage = (count / self.total_responses) * 100
                report_lines.append(f"  {education}: {count} ({percentage:.1f}%)")
            report_lines.append("")

        return "\n".join(report_lines)
```

5. Statistical Analysis Module (stats_analyzer.py)

The StatsAnalyzer class performs advanced statistical analysis including cross-tabulation, chi-square tests, correlation analysis, and significance testing. It provides the statistical foundation for data-driven insights.

File: stats_analyzer.py

```
#!/usr/bin/env python3
"""
Statistical Analyzer Module
Author: Student Developer
Description: Performs statistical analysis including cross-tabulation,
chi-square tests, and correlation analysis on survey data.
"""

import math
from typing import List, Dict, Any, Optional, Tuple
from collections import defaultdict, Counter
import itertools


class StatsAnalyzer:
    """Handles statistical analysis of survey data including cross-tabulation and chi-square t
ests."""

    def __init__(self, survey_data: List[Dict[str, Any]]):
        """
        Initialize the statistical analyzer.

        Args:
            survey_data: List of dictionaries containing survey responses
        """
        self.survey_data = survey_data
        self.total_responses = len(survey_data)
        self.columns = list(survey_data[0].keys()) if survey_data else []

    def cross_tabulate(self, col1: str, col2: str) -> List[List]:
        """
        Perform cross-tabulation between two columns.

        Args:
            col1: First column name
            col2: Second column name

        Returns:
            Cross-tabulation matrix as a list of lists
        """
        if col1 not in self.columns or col2 not in self.columns:
            raise ValueError(f"Column not found: {col1} or {col2}")

        # Get unique values for each column
        values1 = set()
        values2 = set()

        for row in self.survey_data:
            val1 = row.get(col1)
            val2 = row.get(col2)

            if val1 is not None and str(val1).strip():
```

```
                values1.add(str(val1).strip())
            if val2 is not None and str(val2).strip():
                values2.add(str(val2).strip())

        # Sort values for consistent ordering
        values1 = sorted(list(values1))
        values2 = sorted(list(values2))

        # Create cross-tabulation matrix
        crosstab = []

        # Header row
        header = [''] + values2
        crosstab.append(header)

        # Data rows
        for val1 in values1:
            row = [val1]
            for val2 in values2:
                count = 0
                for survey_row in self.survey_data:
                    if (str(survey_row.get(col1, '')).strip() == val1 and
                        str(survey_row.get(col2, '')).strip() == val2):
                        count += 1
                row.append(count)
            crosstab.append(row)

        return crosstab

    def chi_square_test(self, col1: str, col2: str) -> Dict[str, Any]:
        """
        Perform chi-square test of independence between two categorical variables.

        Args:
            col1: First column name
            col2: Second column name

        Returns:
            Dictionary containing chi-square test results
        """
        # Get cross-tabulation
        crosstab = self.cross_tabulate(col1, col2)

        if len(crosstab) < 2 or len(crosstab[0]) < 2:
            return {
                'chi_square': 0.0,
                'p_value': 1.0,
                'df': 0,
                'significant': False,
                'error': 'Insufficient data for chi-square test'
            }

        # Extract observed frequencies (skip header row and column)
        observed = []
        for i in range(1, len(crosstab)):
            row = []
            for j in range(1, len(crosstab[i])):
                row.append(int(crosstab[i][j]))
            observed.append(row)

        # Check if we have enough data for chi-square test
        total_observations = sum(sum(row) for row in observed)
        if total_observations < 5:  # Chi-square test requires at least 5 observations
            return {
                'chi_square': 0.0,
```

```
                'p_value': 1.0,
                'df': 0,
                'significant': False,
                'error': 'Insufficient data for chi-square test'
            }

        # Calculate expected frequencies
        expected = self._calculate_expected_frequencies(observed)

        # Calculate chi-square statistic
        chi_square = 0.0
        for i in range(len(observed)):
            for j in range(len(observed[i])):
                if expected[i][j] > 0:
                    chi_square += ((observed[i][j] - expected[i][j]) ** 2) / expected[i][j]

        # Calculate degrees of freedom
        df = (len(observed) - 1) * (len(observed[0]) - 1)

        # Calculate p-value (approximation using chi-square distribution)
        p_value = self._chi_square_p_value(chi_square, df)

        # Determine significance (alpha = 0.05)
        significant = p_value < 0.05

        return {
            'chi_square': chi_square,
            'p_value': p_value,
            'df': df,
            'significant': significant,
            'observed': observed,
            'expected': expected
        }

    def _calculate_expected_frequencies(self, observed: List[List[int]]) ->
List[List[float]]:
        """Calculate expected frequencies for chi-square test."""
        if not observed or not observed[0]:
            return []

        rows = len(observed)
        cols = len(observed[0])

        # Calculate row and column totals
        row_totals = [sum(row) for row in observed]
        col_totals = []
        for j in range(cols):
            col_totals.append(sum(observed[i][j] for i in range(rows)))

        total = sum(row_totals)

        # Calculate expected frequencies
        expected = []
        for i in range(rows):
            row = []
            for j in range(cols):
                expected_freq = (row_totals[i] * col_totals[j]) / total if total > 0 else 0
                row.append(expected_freq)
            expected.append(row)

        return expected

    def _chi_square_p_value(self, chi_square: float, df: int) -> float:
        """
        Calculate approximate p-value for chi-square statistic.
```

```
        This is a simplified approximation - in practice, you'd use a proper chi-square
distri
bution.
        """
        if df <= 0:
            return 1.0

        # Simple approximation for chi-square p-value
        # For small chi-square values, p-value is close to 1
        # For large chi-square values, p-value approaches 0
        if chi_square < df:
            return 1.0 - (chi_square / (df * 2))
        else:
            return max(0.0, 1.0 - (chi_square / (df * 10)))

    def correlation_analysis(self, col1: str, col2: str) -> Dict[str, Any]:
        """
        Perform correlation analysis between two variables.

        Args:
            col1: First column name
            col2: Second column name

        Returns:
            Dictionary containing correlation analysis results
        """
        # Extract numeric values
        values1 = []
        values2 = []

        for row in self.survey_data:
            val1 = row.get(col1)
            val2 = row.get(col2)

            # Try to convert to numeric
            try:
                if val1 is not None and str(val1).strip():
                    num1 = float(val1)
                    if val2 is not None and str(val2).strip():
                        num2 = float(val2)
                        values1.append(num1)
                        values2.append(num2)
            except (ValueError, TypeError):
                continue

        if len(values1) < 2:
            return {
                'correlation': 0.0,
                'sample_size': 0,
                'error': 'Insufficient numeric data for correlation analysis'
            }

        # Calculate correlation coefficient
        correlation = self._calculate_correlation(values1, values2)

        return {
            'correlation': correlation,
            'sample_size': len(values1),
            'strength': self._interpret_correlation(correlation)
        }

    def _calculate_correlation(self, x: List[float], y: List[float]) -> float:
        """Calculate Pearson correlation coefficient."""
        if len(x) != len(y) or len(x) < 2:
            return 0.0
```

```
244:
245:         n = len(x)
246:
247:         # Calculate means
248:         mean_x = sum(x) / n
249:         mean_y = sum(y) / n
250:
251:         # Calculate correlation coefficient
252:         numerator = sum((x[i] - mean_x) * (y[i] - mean_y) for i in range(n))
253:         denominator_x = sum((x[i] - mean_x) ** 2 for i in range(n))
254:         denominator_y = sum((y[i] - mean_y) ** 2 for i in range(n))
255:
256:         if denominator_x == 0 or denominator_y == 0:
257:             return 0.0
258:
259:         correlation = numerator / math.sqrt(denominator_x * denominator_y)
260:         return correlation
261:
262:     def _interpret_correlation(self, correlation: float) -> str:
263:         """Interpret correlation coefficient strength."""
264:         abs_corr = abs(correlation)
265:
266:         if abs_corr >= 0.8:
267:             return "Very Strong"
268:         elif abs_corr >= 0.6:
269:             return "Strong"
270:         elif abs_corr >= 0.4:
271:             return "Moderate"
272:         elif abs_corr >= 0.2:
273:             return "Weak"
274:         else:
275:             return "Very Weak"
276:
277:     def analyze_response_patterns(self) -> Dict[str, Any]:
278:         """Analyze patterns in survey responses."""
279:         patterns = {
280:             'common_combinations': [],
281:             'response_clusters': [],
282:             'outliers': []
283:         }
284:
285:         # Find common combinations of responses
286:         response_combinations = []
287:         for row in self.survey_data:
288:             combination = []
289:             for col in self.columns:
290:                 value = row.get(col)
291:                 if value is not None and str(value).strip():
292:                     combination.append(f"{col}:{str(value).strip()}")
293:             if combination:
294:                 response_combinations.append(tuple(sorted(combination)))
295:
296:         # Count combinations
297:         combination_counts = Counter(response_combinations)
298:         common_combinations = combination_counts.most_common(5)
299:
300:         patterns['common_combinations'] = [
301:             {
302:                 'combination': list(combo),
303:                 'count': count,
304:                 'percentage': (count / len(response_combinations)) * 100
305:             }
306:             for combo, count in common_combinations
307:         ]
308:
```

```
        return patterns

    def get_statistical_summary(self) -> Dict[str, Any]:
        """Generate a comprehensive statistical summary."""
        summary = {
            'total_responses': self.total_responses,
            'total_columns': len(self.columns),
            'numeric_columns': [],
            'categorical_columns': [],
            'statistical_tests': []
        }

        # Categorize columns
        for column in self.columns:
            numeric_count = 0
            total_count = 0

            for row in self.survey_data:
                value = row.get(column)
                if value is not None and str(value).strip():
                    total_count += 1
                    try:
                        float(str(value))
                        numeric_count += 1
                    except (ValueError, TypeError):
                        pass

            if total_count > 0 and (numeric_count / total_count) > 0.5:
                summary['numeric_columns'].append(column)
            else:
                summary['categorical_columns'].append(column)

        return summary

    def perform_multiple_chi_square_tests(self, target_column: str) -> List[Dict[str, Any]]:
        """
        Perform chi-square tests between a target column and all other categorical columns.

        Args:
            target_column: The target column to test against

        Returns:
            List of chi-square test results
        """
        results = []

        for column in self.columns:
            if column != target_column:
                try:
                    test_result = self.chi_square_test(target_column, column)
                    test_result['column1'] = target_column
                    test_result['column2'] = column
                    results.append(test_result)
                except Exception as e:
                    results.append({
                        'column1': target_column,
                        'column2': column,
                        'error': str(e)
                    })

        # Sort by significance
        results.sort(key=lambda x: x.get('p_value', 1.0))

        return results
```

6. Sentiment Analysis Module (sentiment_analyzer.py)

The SentimentAnalyzer class performs text-based sentiment analysis using keyword dictionaries, negation handling, and intensifier recognition. It provides insights into the emotional content of open-ended survey responses.

File: sentiment_analyzer.py

```
#!/usr/bin/env python3
"""
Sentiment Analyzer Module
Description: Performs basic sentiment analysis on text responses using
keyword matching and scoring systems without external libraries.
"""

import re
from typing import List, Dict, Any, Optional
from collections import Counter


class SentimentAnalyzer:
    """Handles sentiment analysis of text responses using keyword-based approach."""

    def __init__(self):
        """Initialize the sentiment analyzer with keyword dictionaries."""
        # Positive keywords and their weights
        self.positive_keywords = {
            'excellent': 3, 'amazing': 3, 'great': 2, 'good': 2, 'wonderful': 3,
            'fantastic': 3, 'outstanding': 3, 'perfect': 3, 'love': 2, 'like': 1,
            'enjoy': 2, 'happy': 2, 'satisfied': 2, 'pleased': 2, 'impressed': 2,
            'recommend': 2, 'helpful': 2, 'useful': 1, 'effective': 2, 'quality': 1,
            'best': 2, 'awesome': 3, 'brilliant': 3, 'superb': 3, 'terrific': 3,
            'delighted': 3, 'thrilled': 3, 'excited': 2, 'positive': 1, 'successful': 2,
            'improved': 1, 'better': 1, 'exceeded': 2, 'surpassed': 2, 'outstanding': 3
        }

        # Negative keywords and their weights
        self.negative_keywords = {
            'terrible': 3, 'awful': 3, 'horrible': 3, 'bad': 2, 'poor': 2,
            'disappointing': 2, 'frustrated': 2, 'angry': 2, 'upset': 2, 'annoyed': 2,
            'hate': 3, 'dislike': 2, 'worst': 3, 'useless': 2, 'waste': 2,
            'problem': 1, 'issue': 1, 'difficult': 1, 'confusing': 1, 'complicated': 1,
            'broken': 2, 'failed': 2, 'error': 1, 'bug': 1, 'crash': 2,
            'slow': 1, 'expensive': 1, 'overpriced': 2, 'cheap': 1, 'low quality': 2,
            'unreliable': 2, 'unstable': 2, 'inconsistent': 1, 'disorganized': 1,
            'messy': 1, 'chaotic': 2, 'stressful': 2, 'overwhelming': 2
        }

        # Neutral keywords (used for context)
        self.neutral_keywords = {
            'okay': 0, 'fine': 0, 'average': 0, 'normal': 0, 'standard': 0,
            'usual': 0, 'typical': 0, 'regular': 0, 'common': 0, 'basic': 0,
            'simple': 0, 'straightforward': 0, 'clear': 0, 'understandable': 0,
            'adequate': 0, 'sufficient': 0, 'acceptable': 0, 'reasonable': 0
        }

        # Negation words that can flip sentiment
        self.negation_words = {
            'not', 'no', 'never', 'none', 'neither', 'nor', 'nobody', 'nothing',
            'nowhere', 'hardly', 'barely', 'scarcely', 'doesn\'t', 'don\'t',
            'didn\'t', 'won\'t', 'can\'t', 'couldn\'t', 'wouldn\'t', 'shouldn\'t',
```

```
            'isn\'t', 'aren\'t', 'wasn\'t', 'weren\'t', 'hasn\'t', 'haven\'t',
            'hadn\'t', 'doesnt', 'dont', 'didnt', 'wont', 'cant', 'couldnt',
            'wouldnt', 'shouldnt', 'isnt', 'arent', 'wasnt', 'werent', 'hasnt',
            'havent', 'hadnt'
        }

        # Intensifier words that amplify sentiment
        self.intensifier_words = {
            'very': 1.5, 'really': 1.5, 'extremely': 2.0, 'absolutely': 2.0,
            'completely': 2.0, 'totally': 2.0, 'entirely': 2.0, 'thoroughly': 1.5,
            'highly': 1.5, 'incredibly': 2.0, 'amazingly': 2.0, 'exceptionally': 2.0,
            'particularly': 1.2, 'especially': 1.2, 'notably': 1.2, 'remarkably': 1.5
        }

    def analyze_text(self, text: str) -> Dict[str, Any]:
        """
        Analyze sentiment of a single text response.

        Args:
            text: Text string to analyze

        Returns:
            Dictionary containing sentiment analysis results
        """
        if not text or not isinstance(text, str):
            return {
                'sentiment': 'neutral',
                'score': 0,
                'positive_words': [],
                'negative_words': [],
                'confidence': 0.0
            }

        # Clean and normalize text
        cleaned_text = self._clean_text(text)

        # Extract words
        words = self._extract_words(cleaned_text)

        # Analyze sentiment
        sentiment_score = 0
        positive_words = []
        negative_words = []
        intensifier_count = 0

        for i, word in enumerate(words):
            word_lower = word.lower()

            # Check for intensifiers
            if word_lower in self.intensifier_words:
                intensifier_count += 1
                continue

            # Check for negations
            is_negated = self._is_negated(words, i)

            # Check positive keywords
            if word_lower in self.positive_keywords:
                weight = self.positive_keywords[word_lower]
                if is_negated:
                    sentiment_score -= weight
                    negative_words.append(word)
                else:
                    sentiment_score += weight
                    positive_words.append(word)
```

```
119:
120:             # Check negative keywords
121:             elif word_lower in self.negative_keywords:
122:                 weight = self.negative_keywords[word_lower]
123:                 if is_negated:
124:                     sentiment_score += weight
125:                     positive_words.append(word)
126:                 else:
127:                     sentiment_score -= weight
128:                     negative_words.append(word)
129:
130:         # Apply intensifier multiplier
131:         if intensifier_count > 0:
132:             sentiment_score *= (1 + (intensifier_count * 0.2))
133:
134:         # Determine sentiment category
135:         sentiment = self._categorize_sentiment(sentiment_score)
136:
137:         # Calculate confidence
138:         total_words = len(words)
139:         sentiment_words = len(positive_words) + len(negative_words)
140:         confidence = min(1.0, sentiment_words / max(total_words, 1))
141:
142:         return {
143:             'sentiment': sentiment,
144:             'score': sentiment_score,
145:             'positive_words': positive_words,
146:             'negative_words': negative_words,
147:             'confidence': confidence,
148:             'total_words': total_words,
149:             'sentiment_words': sentiment_words
150:         }
151:
152:     def analyze_column(self, survey_data: List[Dict[str, Any]], column: str) -> Dict[str,
Any]
:
153:         """
154:         Analyze sentiment for all responses in a specific column.
155:
156:         Args:
157:             survey_data: List of survey responses
158:             column: Column name to analyze
159:
160:         Returns:
161:             Dictionary containing aggregated sentiment analysis results
162:         """
163:         if not survey_data or column not in survey_data[0]:
164:             return {
165:                 'positive': 0,
166:                 'negative': 0,
167:                 'neutral': 0,
168:                 'positive_pct': 0.0,
169:                 'negative_pct': 0.0,
170:                 'neutral_pct': 0.0,
171:                 'avg_score': 0.0,
172:                 'total_responses': 0
173:             }
174:
175:         sentiment_results = []
176:         total_responses = 0
177:
178:         for row in survey_data:
179:             text = row.get(column)
180:             if text is not None and str(text).strip():
181:                 result = self.analyze_text(str(text))
```

```
                sentiment_results.append(result)
                total_responses += 1

        if not sentiment_results:
            return {
                'positive': 0,
                'negative': 0,
                'neutral': 0,
                'positive_pct': 0.0,
                'negative_pct': 0.0,
                'neutral_pct': 0.0,
                'avg_score': 0.0,
                'total_responses': 0
            }

        # Count sentiments
        positive_count = sum(1 for r in sentiment_results if r['sentiment'] == 'positive')
        negative_count = sum(1 for r in sentiment_results if r['sentiment'] == 'negative')
        neutral_count = sum(1 for r in sentiment_results if r['sentiment'] == 'neutral')

        # Calculate percentages
        total = len(sentiment_results)
        positive_pct = (positive_count / total) * 100
        negative_pct = (negative_count / total) * 100
        neutral_pct = (neutral_count / total) * 100

        # Calculate average score
        avg_score = sum(r['score'] for r in sentiment_results) / total

        return {
            'positive': positive_count,
            'negative': negative_count,
            'neutral': neutral_count,
            'positive_pct': positive_pct,
            'negative_pct': negative_pct,
            'neutral_pct': neutral_pct,
            'avg_score': avg_score,
            'total_responses': total_responses,
            'detailed_results': sentiment_results
        }

    def analyze_all_text_columns(self, survey_data: List[Dict[str, Any]]) -> Dict[str, Any]:
        """
        Analyze sentiment for all text columns in the survey data.

        Args:
            survey_data: List of survey responses

        Returns:
            Dictionary containing sentiment analysis for all text columns
        """
        if not survey_data:
            return {}

        text_columns = []
        for col in survey_data[0].keys():
            # Check if column contains text data
            text_count = 0
            total_count = 0

            for row in survey_data[:10]:  # Sample first 10 rows
                value = row.get(col)
                if value is not None and str(value).strip():
                    total_count += 1
                    if len(str(value)) > 20:  # Consider it text if longer than 20 chars
```

```
                        text_count += 1

            if total_count > 0 and (text_count / total_count) > 0.3:
                text_columns.append(col)

        results = {}
        for column in text_columns:
            results[column] = self.analyze_column(survey_data, column)

        return results

    def _clean_text(self, text: str) -> str:
        """Clean and normalize text for analysis."""
        # Convert to lowercase
        text = text.lower()

        # Remove extra whitespace
        text = re.sub(r'\s+', ' ', text)

        # Remove punctuation (keep apostrophes for contractions)
        text = re.sub(r'[^\w\s\']', ' ', text)

        return text.strip()

    def _extract_words(self, text: str) -> List[str]:
        """Extract words from text."""
        return text.split()

    def _is_negated(self, words: List[str], current_index: int) -> bool:
        """Check if current word is negated by previous words."""
        # Look back up to 3 words for negation
        start_index = max(0, current_index - 3)

        for i in range(start_index, current_index):
            if i < len(words) and words[i].lower() in self.negation_words:
                return True

        return False

    def _categorize_sentiment(self, score: float) -> str:
        """Categorize sentiment based on score."""
        if score > 1.0:
            return 'positive'
        elif score < -1.0:
            return 'negative'
        else:
            return 'neutral'

    def get_sentiment_summary(self, sentiment_results: List[Dict[str, Any]]) -> Dict[str,
Any]
:
        """Generate a summary of sentiment analysis results."""
        if not sentiment_results:
            return {}

        total_responses = len(sentiment_results)
        positive_count = sum(1 for r in sentiment_results if r['sentiment'] == 'positive')
        negative_count = sum(1 for r in sentiment_results if r['sentiment'] == 'negative')
        neutral_count = sum(1 for r in sentiment_results if r['sentiment'] == 'neutral')

        # Most common positive and negative words
        all_positive_words = []
        all_negative_words = []

        for result in sentiment_results:
```

```
            all_positive_words.extend(result['positive_words'])
            all_negative_words.extend(result['negative_words'])

        positive_word_counts = Counter(all_positive_words)
        negative_word_counts = Counter(all_negative_words)

        return {
            'total_responses': total_responses,
            'sentiment_distribution': {
                'positive': {'count': positive_count, 'percentage': (positive_count / total_responses) * 100},
                'negative': {'count': negative_count, 'percentage': (negative_count / total_responses) * 100},
                'neutral': {'count': neutral_count, 'percentage': (neutral_count / total_responses) * 100}
            },
            'top_positive_words': positive_word_counts.most_common(5),
            'top_negative_words': negative_word_counts.most_common(5),
            'average_confidence': sum(r['confidence'] for r in sentiment_results) / total_responses,
            'average_score': sum(r['score'] for r in sentiment_results) / total_responses
        }

    def export_sentiment_report(self, sentiment_results: Dict[str, Any], filename: str = "sentiment_report.txt") -> bool:
        """Export sentiment analysis results to a text file."""
        try:
            with open(filename, 'w', encoding='utf-8') as file:
                file.write("SENTIMENT ANALYSIS REPORT\n")
                file.write("=" * 50 + "\n\n")

                for column, results in sentiment_results.items():
                    file.write(f"Column: {column}\n")
                    file.write("-" * 30 + "\n")
                    file.write(f"Total Responses: {results['total_responses']}\n")
                    file.write(f"Positive: {results['positive']} ({results['positive_pct']:.1f}%)\n")
                    file.write(f"Negative: {results['negative']} ({results['negative_pct']:.1f}%)\n")
                    file.write(f"Neutral: {results['neutral']} ({results['neutral_pct']:.1f}%)\n")
                    file.write(f"Average Score: {results['avg_score']:.2f}\n\n")

            return True

        except Exception as e:
            print(f"Error exporting sentiment report: {str(e)}")
            return False
```

7. Pattern Detection Module (pattern_detector.py)

The PatternDetector class identifies correlations, trends, and patterns in survey responses. It analyzes demographic patterns, response combinations, and outlier detection to uncover meaningful insights in the data.

File: pattern_detector.py

```
#!/usr/bin/env python3
"""
Pattern Detector Module
Author: Student Developer
Description: Identifies correlations, patterns, and trends in survey responses
using statistical analysis and pattern recognition techniques.
"""

import math
from typing import List, Dict, Any, Optional, Tuple
from collections import defaultdict, Counter
import itertools


class PatternDetector:
    """Handles pattern detection and correlation analysis in survey data."""

    def __init__(self, survey_data: List[Dict[str, Any]]):
        """
        Initialize the pattern detector.

        Args:
            survey_data: List of dictionaries containing survey responses
        """
        self.survey_data = survey_data
        self.total_responses = len(survey_data)
        self.columns = list(survey_data[0].keys()) if survey_data else []

    def find_patterns(self) -> List[Dict[str, Any]]:
        """
        Find patterns and correlations in survey responses.

        Returns:
            List of detected patterns with descriptions and confidence levels
        """
        if not self.survey_data:
            return []

        patterns = []

        # Find demographic patterns
        patterns.extend(self._find_demographic_patterns())

        # Find response correlation patterns
        patterns.extend(self._find_correlation_patterns())

        # Find response combination patterns
        patterns.extend(self._find_combination_patterns())

        # Find outlier patterns
        patterns.extend(self._find_outlier_patterns())

        # Sort patterns by confidence
```

```
        patterns.sort(key=lambda x: x['confidence'], reverse=True)

        return patterns

    def _find_demographic_patterns(self) -> List[Dict[str, Any]]:
        """Find patterns related to demographics."""
        patterns = []

        # Age-based patterns
        if 'age' in self.columns:
            age_patterns = self._analyze_age_patterns()
            patterns.extend(age_patterns)

        # Gender-based patterns
        if 'gender' in self.columns:
            gender_patterns = self._analyze_gender_patterns()
            patterns.extend(gender_patterns)

        # Regional patterns
        if 'region' in self.columns:
            regional_patterns = self._analyze_regional_patterns()
            patterns.extend(regional_patterns)

        # Education-based patterns
        if 'education' in self.columns:
            education_patterns = self._analyze_education_patterns()
            patterns.extend(education_patterns)

        return patterns

    def _analyze_age_patterns(self) -> List[Dict[str, Any]]:
        """Analyze patterns based on age groups."""
        patterns = []

        # Group responses by age
        age_groups = {
            '18-25': [],
            '26-35': [],
            '36-45': [],
            '46-55': [],
            '56-65': [],
            '65+': []
        }

        for row in self.survey_data:
            age_str = row.get('age')
            if age_str and str(age_str).isdigit():
                try:
                    age = int(age_str)
                    if 18 <= age <= 25:
                        age_groups['18-25'].append(row)
                    elif 26 <= age <= 35:
                        age_groups['26-35'].append(row)
                    elif 36 <= age <= 45:
                        age_groups['36-45'].append(row)
                    elif 46 <= age <= 55:
                        age_groups['46-55'].append(row)
                    elif 56 <= age <= 65:
                        age_groups['56-65'].append(row)
                    elif age > 65:
                        age_groups['65+'].append(row)
                except ValueError:
                    continue

        # Analyze patterns for each question
```

```
        for column in self.columns:
            if column not in ['age', 'gender', 'region', 'education']:
                column_patterns = self._analyze_column_by_age_groups(column, age_groups)
                patterns.extend(column_patterns)

        return patterns

    def _analyze_column_by_age_groups(self, column: str, age_groups: Dict[str, List[Dict[str,
Any]]]) -> List[Dict[str, Any]]:
        """Analyze a specific column's responses by age groups."""
        patterns = []

        # Get most common response for each age group
        age_group_responses = {}
        for age_group, responses in age_groups.items():
            if responses:
                values = [r.get(column) for r in responses if r.get(column)]
                if values:
                    value_counts = Counter(values)
                    most_common = value_counts.most_common(1)[0]
                    age_group_responses[age_group] = {
                        'response': most_common[0],
                        'count': most_common[1],
                        'percentage': (most_common[1] / len(values)) * 100
                    }

        # Find patterns
        if len(age_group_responses) > 1:
            # Find age groups with similar responses
            response_groups = defaultdict(list)
            for age_group, data in age_group_responses.items():
                response_groups[data['response']].append(age_group)

            for response, age_groups_list in response_groups.items():
                if len(age_groups_list) > 1:
                    confidence = min(90, len(age_groups_list) * 20)
                    patterns.append({
                        'type': 'age_pattern',
                        'description': f"Age groups {', '.join(age_groups_list)} most
commonly
responded '{response}' to {column}",
                        'confidence': confidence,
                        'sample_size': sum(len(age_groups[ag]) for ag in age_groups_list),
                        'response': response,
                        'affected_groups': age_groups_list
                    })

        return patterns

    def _analyze_gender_patterns(self) -> List[Dict[str, Any]]:
        """Analyze patterns based on gender."""
        patterns = []

        # Group responses by gender
        gender_groups = defaultdict(list)
        for row in self.survey_data:
            gender = row.get('gender')
            if gender and str(gender).strip():
                gender_groups[str(gender).strip().title()].append(row)

        # Analyze patterns for each question
        for column in self.columns:
            if column not in ['age', 'gender', 'region', 'education']:
                column_patterns = self._analyze_column_by_gender(column, gender_groups)
```

```
                patterns.extend(column_patterns)

        return patterns

    def _analyze_column_by_gender(self, column: str, gender_groups: Dict[str, List[Dict[str,
A
ny]]]) -> List[Dict[str, Any]]:
        """Analyze a specific column's responses by gender."""
        patterns = []

        # Get most common response for each gender
        gender_responses = {}
        for gender, responses in gender_groups.items():
            if responses:
                values = [r.get(column) for r in responses if r.get(column)]
                if values:
                    value_counts = Counter(values)
                    most_common = value_counts.most_common(1)[0]
                    gender_responses[gender] = {
                        'response': most_common[0],
                        'count': most_common[1],
                        'percentage': (most_common[1] / len(values)) * 100
                    }

        # Find gender differences
        if len(gender_responses) > 1:
            responses = list(gender_responses.values())
            if len(set(r['response'] for r in responses)) > 1:
                # Different genders have different most common responses
                confidence = 75
                patterns.append({
                    'type': 'gender_pattern',
                    'description': f"Gender differences detected in {column} responses",
                    'confidence': confidence,
                    'sample_size': sum(len(gender_groups[g]) for g in
gender_responses.keys())
,
                    'details': gender_responses
                })

        return patterns

    def _analyze_regional_patterns(self) -> List[Dict[str, Any]]:
        """Analyze patterns based on region."""
        patterns = []

        # Group responses by region
        regional_groups = defaultdict(list)
        for row in self.survey_data:
            region = row.get('region')
            if region and str(region).strip():
                regional_groups[str(region).strip().title()].append(row)

        # Analyze patterns for each question
        for column in self.columns:
            if column not in ['age', 'gender', 'region', 'education']:
                column_patterns = self._analyze_column_by_region(column, regional_groups)
                patterns.extend(column_patterns)

        return patterns

    def _analyze_column_by_region(self, column: str, regional_groups: Dict[str,
List[Dict[str,
Any]]]) -> List[Dict[str, Any]]:
        """Analyze a specific column's responses by region."""
```

```
        patterns = []

        # Get most common response for each region
        regional_responses = {}
        for region, responses in regional_groups.items():
            if responses:
                values = [r.get(column) for r in responses if r.get(column)]
                if values:
                    value_counts = Counter(values)
                    most_common = value_counts.most_common(1)[0]
                    regional_responses[region] = {
                        'response': most_common[0],
                        'count': most_common[1],
                        'percentage': (most_common[1] / len(values)) * 100
                    }

        # Find regional patterns
        if len(regional_responses) > 1:
            # Find regions with similar responses
            response_groups = defaultdict(list)
            for region, data in regional_responses.items():
                response_groups[data['response']].append(region)

            for response, regions_list in response_groups.items():
                if len(regions_list) > 1:
                    confidence = min(85, len(regions_list) * 25)
                    patterns.append({
                        'type': 'regional_pattern',
                        'description': f"Regions {', '.join(regions_list)} most commonly
respo
nded '{response}' to {column}",
                        'confidence': confidence,
                        'sample_size': sum(len(regional_groups[r]) for r in regions_list),
                        'response': response,
                        'affected_regions': regions_list
                    })

        return patterns

    def _analyze_education_patterns(self) -> List[Dict[str, Any]]:
        """Analyze patterns based on education level."""
        patterns = []

        # Group responses by education
        education_groups = defaultdict(list)
        for row in self.survey_data:
            education = row.get('education')
            if education and str(education).strip():
                education_groups[str(education).strip().title()].append(row)

        # Analyze patterns for each question
        for column in self.columns:
            if column not in ['age', 'gender', 'region', 'education']:
                column_patterns = self._analyze_column_by_education(column, education_groups)
                patterns.extend(column_patterns)

        return patterns

    def _analyze_column_by_education(self, column: str, education_groups: Dict[str,
List[Dict[
str, Any]]]) -> List[Dict[str, Any]]:
        """Analyze a specific column's responses by education level."""
        patterns = []

        # Get most common response for each education level
```

```
        education_responses = {}
        for education, responses in education_groups.items():
            if responses:
                values = [r.get(column) for r in responses if r.get(column)]
                if values:
                    value_counts = Counter(values)
                    most_common = value_counts.most_common(1)[0]
                    education_responses[education] = {
                        'response': most_common[0],
                        'count': most_common[1],
                        'percentage': (most_common[1] / len(values)) * 100
                    }

        # Find education-based patterns
        if len(education_responses) > 1:
            # Find education levels with similar responses
            response_groups = defaultdict(list)
            for education, data in education_responses.items():
                response_groups[data['response']].append(education)

            for response, education_levels in response_groups.items():
                if len(education_levels) > 1:
                    confidence = min(80, len(education_levels) * 20)
                    patterns.append({
                        'type': 'education_pattern',
                        'description': f"Education levels {', '.join(education_levels)} most c
ommonly responded '{response}' to {column}",
                        'confidence': confidence,
                        'sample_size': sum(len(education_groups[e]) for e in
education_levels)
,
                        'response': response,
                        'affected_education_levels': education_levels
                    })

        return patterns

    def _find_correlation_patterns(self) -> List[Dict[str, Any]]:
        """Find correlation patterns between different questions."""
        patterns = []

        # Analyze correlations between categorical variables
        categorical_columns = [col for col in self.columns if col not in ['age', 'gender',
're
gion', 'education']]

        for i, col1 in enumerate(categorical_columns):
            for col2 in categorical_columns[i+1:]:
                correlation_pattern = self._analyze_correlation(col1, col2)
                if correlation_pattern:
                    patterns.append(correlation_pattern)

        return patterns

    def _analyze_correlation(self, col1: str, col2: str) -> Optional[Dict[str, Any]]:
        """Analyze correlation between two columns."""
        # Get unique values for both columns
        values1 = set()
        values2 = set()

        for row in self.survey_data:
            val1 = row.get(col1)
            val2 = row.get(col2)

```

```
 359:             if val1 is not None and str(val1).strip():
 360:                 values1.add(str(val1).strip())
 361:             if val2 is not None and str(val2).strip():
 362:                 values2.add(str(val2).strip())
 363:
 364:         if len(values1) < 2 or len(values2) < 2:
 365:             return None
 366:
 367:         # Calculate correlation strength
 368:         total_responses = 0
 369:         matching_responses = 0
 370:
 371:         for row in self.survey_data:
 372:             val1 = row.get(col1)
 373:             val2 = row.get(col2)
 374:
 375:             if val1 is not None and str(val1).strip() and val2 is not None and
str(val2).strip
():
 376:                 total_responses += 1
 377:                 # Check if responses are related (simplified correlation)
 378:                 if self._are_responses_related(str(val1).strip(), str(val2).strip()):
 379:                     matching_responses += 1
 380:
 381:         if total_responses == 0:
 382:             return None
 383:
 384:         correlation_strength = matching_responses / total_responses
 385:
 386:         if correlation_strength > 0.6:  # Strong correlation threshold
 387:             return {
 388:                 'type': 'correlation_pattern',
 389:                 'description': f"Strong correlation detected between {col1} and {col2}",
 390:                 'confidence': min(90, correlation_strength * 100),
 391:                 'sample_size': total_responses,
 392:                 'correlation_strength': correlation_strength,
 393:                 'columns': [col1, col2]
 394:             }
 395:
 396:         return None
 397:
 398:     def _are_responses_related(self, response1: str, response2: str) -> bool:
 399:         """Check if two responses are related (simplified logic)."""
 400:         # This is a simplified correlation check
 401:         # In a real implementation, you might use more sophisticated methods
 402:
 403:         # Check for similar sentiment
 404:         positive_words = ['good', 'great', 'excellent', 'satisfied', 'happy', 'like', 'love']
 405:         negative_words = ['bad', 'poor', 'terrible', 'dissatisfied', 'unhappy', 'dislike',
'ha
te']
 406:
 407:         response1_lower = response1.lower()
 408:         response2_lower = response2.lower()
 409:
 410:         # Check if both responses have similar sentiment
 411:         response1_positive = any(word in response1_lower for word in positive_words)
 412:         response1_negative = any(word in response1_lower for word in negative_words)
 413:         response2_positive = any(word in response2_lower for word in positive_words)
 414:         response2_negative = any(word in response2_lower for word in negative_words)
 415:
 416:         if response1_positive and response2_positive:
 417:             return True
 418:         if response1_negative and response2_negative:
 419:             return True
```

```
420:
421:         # Check for exact matches
422:         if response1_lower == response2_lower:
423:             return True
424:
425:         return False
426:
427:     def _find_combination_patterns(self) -> List[Dict[str, Any]]:
428:         """Find patterns in response combinations."""
429:         patterns = []
430:
431:         # Find common response combinations
432:         response_combinations = []
433:         for row in self.survey_data:
434:             combination = []
435:             for col in self.columns:
436:                 value = row.get(col)
437:                 if value is not None and str(value).strip():
438:                     combination.append(f"{col}:{str(value).strip()}")
439:             if combination:
440:                 response_combinations.append(tuple(sorted(combination)))
441:
442:         # Count combinations
443:         combination_counts = Counter(response_combinations)
444:         common_combinations = combination_counts.most_common(3)
445:
446:         for combo, count in common_combinations:
447:             if count > 1:  # Only report if combination appears more than once
448:                 percentage = (count / len(response_combinations)) * 100
449:                 if percentage > 10:  # Only report if more than 10% of responses
450:                     patterns.append({
451:                         'type': 'combination_pattern',
452:                         'description': f"Common response combination: {', '.join(combo)}",
453:                         'confidence': min(85, percentage * 2),
454:                         'sample_size': count,
455:                         'percentage': percentage,
456:                         'combination': list(combo)
457:                     })
458:
459:         return patterns
460:
461:     def _find_outlier_patterns(self) -> List[Dict[str, Any]]:
462:         """Find outlier patterns in responses."""
463:         patterns = []
464:
465:         # Find responses that are significantly different from the norm
466:         for column in self.columns:
467:             if column not in ['age', 'gender', 'region', 'education']:
468:                 outlier_pattern = self._analyze_outliers(column)
469:                 if outlier_pattern:
470:                     patterns.append(outlier_pattern)
471:
472:         return patterns
473:
474:     def _analyze_outliers(self, column: str) -> Optional[Dict[str, Any]]:
475:         """Analyze outliers in a specific column."""
476:         values = [row.get(column) for row in self.survey_data if row.get(column)]
477:
478:         if not values:
479:             return None
480:
481:         # Count responses
482:         value_counts = Counter(values)
483:         total_responses = len(values)
484:
```

```
        # Find responses that appear very rarely (outliers)
        outlier_threshold = total_responses * 0.05  # 5% threshold

        outliers = []
        for value, count in value_counts.items():
            if count <= outlier_threshold and count > 0:
                outliers.append({
                    'value': value,
                    'count': count,
                    'percentage': (count / total_responses) * 100
                })

        if outliers:
            return {
                'type': 'outlier_pattern',
                'description': f"Outlier responses detected in {column}",
                'confidence': 70,
                'sample_size': total_responses,
                'outliers': outliers
            }

        return None

    def get_pattern_summary(self) -> Dict[str, Any]:
        """Generate a summary of detected patterns."""
        patterns = self.find_patterns()

        summary = {
            'total_patterns': len(patterns),
            'pattern_types': Counter(p['type'] for p in patterns),
            'high_confidence_patterns': [p for p in patterns if p['confidence'] >= 80],
            'medium_confidence_patterns': [p for p in patterns if 60 <= p['confidence'] <
80],
            'low_confidence_patterns': [p for p in patterns if p['confidence'] < 60]
        }

        return summary
```

8. Report Generation Module (report_generator.py)

The ReportGenerator class creates comprehensive, professionally formatted analysis reports. It combines all analysis results into structured reports with executive summaries, key findings, and actionable recommendations.

File: report_generator.py

```
#!/usr/bin/env python3
"""
Report Generator Module
Description: Generates comprehensive analysis reports from survey data
and saves them to text files with proper formatting and structure.
"""

import os
from datetime import datetime
from typing import List, Dict, Any, Optional


class ReportGenerator:
    """Handles generation of comprehensive survey analysis reports."""

    def __init__(self):
        """Initialize the report generator."""
        self.report_sections = []
        self.current_datetime = datetime.now()

    def generate_report(self, report_data: Dict[str, Any]) -> bool:
        """
        Generate a comprehensive survey analysis report.

        Args:
            report_data: Dictionary containing all analysis results

        Returns:
            True if report was generated successfully, False otherwise
        """
        try:
            # Extract data from report_data
            summary = report_data.get('summary', {})
            sentiment = report_data.get('sentiment', {})
            patterns = report_data.get('patterns', [])
            file_path = report_data.get('file_path', 'survey_report.txt')

            # Build report content
            report_content = self._build_report_content(summary, sentiment, patterns)

            # Write report to file
            success = self._write_report_to_file(report_content, file_path)

            return success

        except Exception as e:
            print(f"Error generating report: {str(e)}")
            return False

    def _build_report_content(self, summary: Dict[str, Any], sentiment: Dict[str, Any],
patter
ns: List[Dict[str, Any]]) -> str:
        """Build the complete report content."""
```

```
        report_lines = []

        # Header
        report_lines.extend(self._generate_header())

        # Executive Summary
        report_lines.extend(self._generate_executive_summary(summary))

        # Survey Overview
        report_lines.extend(self._generate_survey_overview(summary))

        # Demographic Analysis
        report_lines.extend(self._generate_demographic_analysis(summary))

        # Sentiment Analysis
        report_lines.extend(self._generate_sentiment_analysis(sentiment))

        # Pattern Analysis
        report_lines.extend(self._generate_pattern_analysis(patterns))

        # Statistical Analysis
        report_lines.extend(self._generate_statistical_analysis(summary))

        # Key Findings
        report_lines.extend(self._generate_key_findings(summary, sentiment, patterns))

        # Recommendations
        report_lines.extend(self._generate_recommendations(summary, sentiment, patterns))

        # Footer
        report_lines.extend(self._generate_footer())

        return "\n".join(report_lines)

    def _generate_header(self) -> List[str]:
        """Generate the report header."""
        header = [
            "=" * 80,
            "SURVEY DATA ANALYSIS REPORT",
            "=" * 80,
            f"Generated on: {self.current_datetime.strftime('%B %d, %Y at %I:%M %p')}",
            f"Report ID: SUR-{self.current_datetime.strftime('%Y%m%d-%H%M%S')}",
            "",
            "This report provides a comprehensive analysis of survey responses including",
            "demographic breakdowns, sentiment analysis, pattern detection, and",
            "statistical insights to support data-driven decision making.",
            "",
            "=" * 80,
            ""
        ]
        return header

    def _generate_executive_summary(self, summary: Dict[str, Any]) -> List[str]:
        """Generate the executive summary section."""
        lines = [
            "EXECUTIVE SUMMARY",
            "-" * 50,
            ""
        ]

        if summary:
            total_responses = summary.get('total_responses', 0)
            response_rate = summary.get('response_rate', 0)

            lines.extend([
```

```
                f" Total Survey Responses: {total_responses:,}",
                f" Response Rate: {response_rate:.1f}%",
                "",
                "Key Highlights:",
                " Comprehensive analysis of survey data across multiple dimensions",
                " Demographic breakdowns reveal respondent characteristics",
                " Sentiment analysis provides insights into respondent attitudes",
                " Pattern detection identifies correlations and trends",
                " Statistical analysis supports evidence-based conclusions",
                ""
            ])

        return lines

    def _generate_survey_overview(self, summary: Dict[str, Any]) -> List[str]:
        """Generate the survey overview section."""
        lines = [
            "SURVEY OVERVIEW",
            "-" * 50,
            ""
        ]

        if summary:
            total_responses = summary.get('total_responses', 0)
            data_quality = summary.get('data_quality', {})

            lines.extend([
                f" Survey Details:",
                f"    Total Responses: {total_responses:,}",
                f"    Data Quality: {self._assess_data_quality(data_quality)}",
                ""
            ])

            # Data quality metrics
            if data_quality:
                completeness = data_quality.get('completeness', {})
                if completeness:
                    lines.append(" Data Completeness by Column:")
                    for column, completeness_pct in completeness.items():
                        lines.append(f"    {column}: {completeness_pct:.1f}%")
                    lines.append("")

        return lines

    def _generate_demographic_analysis(self, summary: Dict[str, Any]) -> List[str]:
        """Generate the demographic analysis section."""
        lines = [
            "DEMOGRAPHIC ANALYSIS",
            "-" * 50,
            ""
        ]

        demographics = summary.get('demographics', {})

        if demographics:
            lines.append(" Respondent Demographics:")
            lines.append("")

            for demo_field, breakdown in demographics.items():
                if breakdown:
                    lines.append(f" {demo_field.title()}:")
                    total_demo = sum(breakdown.values())

                    for category, count in breakdown.items():
                        percentage = (count / total_demo) * 100
```

```
                        lines.append(f"    {category}: {count} ({percentage:.1f}%)")
                lines.append("")
        else:
            lines.append("No demographic data available for analysis.")
            lines.append("")

        return lines

    def _generate_sentiment_analysis(self, sentiment: Dict[str, Any]) -> List[str]:
        """Generate the sentiment analysis section."""
        lines = [
            "SENTIMENT ANALYSIS",
            "-" * 50,
            ""
        ]

        if sentiment:
            lines.append(" Text Response Sentiment Analysis:")
            lines.append("")

            for column, results in sentiment.items():
                if isinstance(results, dict) and 'total_responses' in results:
                    lines.append(f" {column}:")
                    lines.append(f"    Total Responses: {results['total_responses']}")
                    lines.append(f"    Positive: {results['positive']}
({results['positive_pct
']:.1f}%)")
                    lines.append(f"    Negative: {results['negative']}
({results['negative_pct
']:.1f}%)")
                    lines.append(f"    Neutral: {results['neutral']}
({results['neutral_pct']:
.1f}%)")
                    lines.append(f"    Average Sentiment Score: {results['avg_score']:.2f}")
                    lines.append("")
        else:
            lines.append("No text responses available for sentiment analysis.")
            lines.append("")

        return lines

    def _generate_pattern_analysis(self, patterns: List[Dict[str, Any]]) -> List[str]:
        """Generate the pattern analysis section."""
        lines = [
            "PATTERN ANALYSIS",
            "-" * 50,
            ""
        ]

        if patterns:
            lines.append(" Detected Patterns and Correlations:")
            lines.append("")

            # Group patterns by type
            pattern_types = {}
            for pattern in patterns:
                pattern_type = pattern.get('type', 'unknown')
                if pattern_type not in pattern_types:
                    pattern_types[pattern_type] = []
                pattern_types[pattern_type].append(pattern)

            for pattern_type, type_patterns in pattern_types.items():
                lines.append(f" {pattern_type.replace('_', ' ').title()} Patterns:")
                for pattern in type_patterns:
```

```
                    lines.append(f"    {pattern['description']}")
                    lines.append(f"     Confidence: {pattern['confidence']:.1f}%")
                    lines.append(f"     Sample Size: {pattern['sample_size']}")
                    lines.append("")
        else:
            lines.append("No significant patterns detected in the survey data.")
            lines.append("")

        return lines

    def _generate_statistical_analysis(self, summary: Dict[str, Any]) -> List[str]:
        """Generate the statistical analysis section."""
        lines = [
            "STATISTICAL ANALYSIS",
            "-" * 50,
            ""
        ]

        question_summaries = summary.get('question_summaries', {})

        if question_summaries:
            lines.append(" Response Distribution Analysis:")
            lines.append("")

            for question, q_summary in question_summaries.items():
                if isinstance(q_summary, dict):
                    lines.append(f" {question}:")
                    lines.append(f"    Total Responses: {q_summary.get('total_responses',
0)}"
)
                    lines.append(f"    Response Rate: {q_summary.get('response_rate',
0):.1f}%
")

                    top_responses = q_summary.get('top_responses', [])
                    if top_responses:
                        lines.append("    Top Responses:")
                        for response, count in top_responses[:3]:
                            percentage = (count / q_summary['total_responses']) * 100
                            lines.append(f"     - {response}: {count} ({percentage:.1f}%)")

                    lines.append("")
        else:
            lines.append("No question response data available for statistical analysis.")
            lines.append("")

        return lines

    def _generate_key_findings(self, summary: Dict[str, Any], sentiment: Dict[str, Any],
patte
rns: List[Dict[str, Any]]) -> List[str]:
        """Generate the key findings section."""
        lines = [
            "KEY FINDINGS",
            "-" * 50,
            ""
        ]

        findings = []

        # Demographic findings
        demographics = summary.get('demographics', {})
        if demographics:
            for demo_field, breakdown in demographics.items():
                if breakdown:
```

```
 300:                     most_common = max(breakdown.items(), key=lambda x: x[1])
 301:                     findings.append(f" {demo_field.title()}: {most_common[0]} is the most
comm
on category ({most_common[1]} responses)")
 302:
 303:         # Sentiment findings
 304:         if sentiment:
 305:             for column, results in sentiment.items():
 306:                 if isinstance(results, dict):
 307:                     dominant_sentiment = self._get_dominant_sentiment(results)
 308:                     findings.append(f" {column}: {dominant_sentiment} sentiment dominates the

responses")
 309:
 310:         # Pattern findings
 311:         if patterns:
 312:             high_confidence_patterns = [p for p in patterns if p.get('confidence', 0) >= 80]
 313:             if high_confidence_patterns:
 314:                 findings.append(f" {len(high_confidence_patterns)} high-confidence patterns
de
tected in the data")
 315:
 316:         if findings:
 317:             lines.extend(findings)
 318:         else:
 319:             lines.append("No significant findings to report at this time.")
 320:
 321:         lines.append("")
 322:         return lines
 323:
 324:     def _generate_recommendations(self, summary: Dict[str, Any], sentiment: Dict[str, Any],
pa
tterns: List[Dict[str, Any]]) -> List[str]:
 325:         """Generate the recommendations section."""
 326:         lines = [
 327:             "RECOMMENDATIONS",
 328:             "-" * 50,
 329:             ""
 330:         ]
 331:
 332:         recommendations = []
 333:
 334:         # Data quality recommendations
 335:         data_quality = summary.get('data_quality', {})
 336:         if data_quality:
 337:             completeness = data_quality.get('completeness', {})
 338:             low_completeness = [col for col, comp in completeness.items() if comp < 80]
 339:             if low_completeness:
 340:                 recommendations.append(f" Improve data collection for columns with low
complet
eness: {', '.join(low_completeness)}")
 341:
 342:         # Sentiment-based recommendations
 343:         if sentiment:
 344:             negative_sentiment_columns = []
 345:             for column, results in sentiment.items():
 346:                 if isinstance(results, dict) and results.get('negative_pct', 0) > 30:
 347:                     negative_sentiment_columns.append(column)
 348:
 349:             if negative_sentiment_columns:
 350:                 recommendations.append(f" Address concerns in columns with high negative
senti
ment: {', '.join(negative_sentiment_columns)}")
 351:
 352:         # Pattern-based recommendations
```

```
        if patterns:
            high_confidence_patterns = [p for p in patterns if p.get('confidence', 0) >= 80]
            if high_confidence_patterns:
                recommendations.append(f" Investigate {len(high_confidence_patterns)}
high-con
fidence patterns for actionable insights")

        # General recommendations
        recommendations.extend([
            " Consider conducting follow-up surveys to validate findings",
            " Implement targeted improvements based on demographic insights",
            " Monitor sentiment trends over time for continuous improvement",
            " Use statistical insights to inform decision-making processes"
        ])

        lines.extend(recommendations)
        lines.append("")

        return lines

    def _generate_footer(self) -> List[str]:
        """Generate the report footer."""
        footer = [
            "=" * 80,
            "REPORT END",
            "=" * 80,
            "",
            "This report was generated automatically by the Survey Data Analyzer.",
            "For questions or additional analysis, please contact the development team.",
            "",
            f"Report generated on: {self.current_datetime.strftime('%B %d, %Y at %I:%M
%p')}",
            "=" * 80
        ]
        return footer

    def _assess_data_quality(self, data_quality: Dict[str, Any]) -> str:
        """Assess overall data quality."""
        if not data_quality:
            return "Unknown"

        completeness = data_quality.get('completeness', {})
        if not completeness:
            return "Unknown"

        avg_completeness = sum(completeness.values()) / len(completeness)

        if avg_completeness >= 90:
            return "Excellent"
        elif avg_completeness >= 80:
            return "Good"
        elif avg_completeness >= 70:
            return "Fair"
        else:
            return "Poor"

    def _get_dominant_sentiment(self, sentiment_results: Dict[str, Any]) -> str:
        """Get the dominant sentiment from results."""
        positive_pct = sentiment_results.get('positive_pct', 0)
        negative_pct = sentiment_results.get('negative_pct', 0)
        neutral_pct = sentiment_results.get('neutral_pct', 0)

        if positive_pct > negative_pct and positive_pct > neutral_pct:
            return "Positive"
        elif negative_pct > positive_pct and negative_pct > neutral_pct:
```

```
            return "Negative"
        else:
            return "Neutral"

    def _write_report_to_file(self, content: str, file_path: str) -> bool:
        """Write the report content to a file."""
        try:
            # Ensure directory exists
            directory = os.path.dirname(file_path)
            if directory and not os.path.exists(directory):
                os.makedirs(directory)

            # Write content to file
            with open(file_path, 'w', encoding='utf-8') as file:
                file.write(content)

            return True

        except Exception as e:
            print(f"Error writing report to file: {str(e)}")
            return False

    def generate_summary_report(self, summary_data: Dict[str, Any], file_path: str =
"summary_
report.txt") -> bool:
        """Generate a simplified summary report."""
        try:
            lines = [
                "SURVEY SUMMARY REPORT",
                "=" * 50,
                f"Generated: {self.current_datetime.strftime('%B %d, %Y')}",
                "",
                f"Total Responses: {summary_data.get('total_responses', 0)}",
                f"Response Rate: {summary_data.get('response_rate', 0):.1f}%",
                "",
                "Key Metrics:",
                " Data quality assessment",
                " Demographic breakdowns",
                " Response distributions",
                " Pattern detection results",
                "",
                "=" * 50
            ]

            content = "\n".join(lines)
            return self._write_report_to_file(content, file_path)

        except Exception as e:
            print(f"Error generating summary report: {str(e)}")
            return False
```

9. Utilities Module (utils.py)

The utils.py file provides utility functions for formatting, validation, and common operations used throughout the application. It includes display formatting, input validation, and helper functions.

File: utils.py

```
#!/usr/bin/env python3
"""
Utilities Module
Description: Provides utility functions for the Survey Data Analyzer CLI,
including display formatting, input validation, and common operations.
"""

import os
import sys
from typing import List, Dict, Any, Optional


def clear_screen():
    """Clear the terminal screen."""
    os.system('cls' if os.name == 'nt' else 'clear')


def print_header(title: str):
    """Print a formatted header."""
    print("\n" + "=" * 60)
    print(f"TARGET: {title}")
    print("=" * 60)


def print_menu(options: List[str]):
    """Print a numbered menu."""
    print("\nAvailable Options:")
    for i, option in enumerate(options, 1):
        print(f"   {i}. {option}")


def print_success(message: str):
    """Print a success message."""
    print(f"SUCCESS: {message}")


def print_error(message: str):
    """Print an error message."""
    print(f"ERROR: {message}")


def print_warning(message: str):
    """Print a warning message."""
    print(f"WARNING: {message}")


def print_info(message: str):
    """Print an info message."""
    print(f"INFO: {message}")


def validate_file_path(file_path: str) -> bool:
    """
    Validate if a file path exists and is accessible.

```

```
    Args:
        file_path: Path to the file to validate

    Returns:
        True if file is valid, False otherwise
    """
    if not file_path or not isinstance(file_path, str):
        return False

    file_path = file_path.strip()
    if not file_path:
        return False

    return os.path.exists(file_path) and os.path.isfile(file_path)


def validate_csv_file(file_path: str) -> bool:
    """
    Validate if a file is a valid CSV file.

    Args:
        file_path: Path to the CSV file

    Returns:
        True if file is a valid CSV, False otherwise
    """
    if not validate_file_path(file_path):
        return False

    # Check file extension
    if not file_path.lower().endswith('.csv'):
        return False

    # Check file size (not empty and not too large)
    try:
        file_size = os.path.getsize(file_path)
        if file_size == 0:
            return False
        if file_size > 50 * 1024 * 1024:  # 50MB limit
            return False
    except OSError:
        return False

    return True


def format_number(number: float, decimal_places: int = 2) -> str:
    """
    Format a number with specified decimal places.

    Args:
        number: Number to format
        decimal_places: Number of decimal places to show

    Returns:
        Formatted number string
    """
    return f"{number:.{decimal_places}f}"


def format_percentage(value: float, total: float) -> str:
    """
    Format a percentage value.

    Args:
```

```
        value: The value to calculate percentage for
        total: The total value

    Returns:
        Formatted percentage string
    """
    if total == 0:
        return "0.0%"
    percentage = (value / total) * 100
    return f"{percentage:.1f}%"


def format_table(data: List[List[str]], headers: List[str] = None) -> str:
    """
    Format data as a table.

    Args:
        data: List of rows, each row is a list of values
        headers: Optional list of column headers

    Returns:
        Formatted table string
    """
    if not data:
        return "No data to display"

    # Determine column widths
    if headers:
        all_rows = [headers] + data
    else:
        all_rows = data

    col_widths = []
    for col in range(len(all_rows[0])):
        max_width = max(len(str(row[col])) for row in all_rows)
        col_widths.append(max_width)

    # Build table
    lines = []

    # Header
    if headers:
        header_line = " | ".join(f"{headers[i]:<{col_widths[i]}}" for i in
range(len(headers))
)
        lines.append(header_line)
        lines.append("-" * len(header_line))

    # Data rows
    for row in data:
        row_line = " | ".join(f"{row[i]:<{col_widths[i]}}" for i in range(len(row)))
        lines.append(row_line)

    return "\n".join(lines)


def get_user_input(prompt: str, default: str = None) -> str:
    """
    Get user input with optional default value.

    Args:
        prompt: Input prompt to display
        default: Optional default value

    Returns:
```

```
        User input string
    """
    if default:
        user_input = input(f"{prompt} (default: {default}): ").strip()
        return user_input if user_input else default
    else:
        return input(f"{prompt}: ").strip()


def confirm_action(prompt: str = "Are you sure?") -> bool:
    """
    Get user confirmation for an action.

    Args:
        prompt: Confirmation prompt

    Returns:
        True if user confirms, False otherwise
    """
    response = input(f"{prompt} (y/N): ").strip().lower()
    return response in ['y', 'yes']


def display_progress(current: int, total: int, description: str = "Processing"):
    """
    Display a progress bar.

    Args:
        current: Current progress value
        total: Total value
        description: Description of the operation
    """
    if total == 0:
        return

    percentage = (current / total) * 100
    bar_length = 30
    filled_length = int(bar_length * current // total)
    bar = '' * filled_length + '-' * (bar_length - filled_length)

    print(f"\r{description}: |{bar}| {percentage:.1f}% ({current}/{total})", end='')

    if current == total:
        print()  # New line when complete


def safe_divide(numerator: float, denominator: float, default: float = 0.0) -> float:
    """
    Safely divide two numbers, returning default if denominator is zero.

    Args:
        numerator: The numerator
        denominator: The denominator
        default: Default value if division by zero

    Returns:
        Result of division or default value
    """
    try:
        return numerator / denominator if denominator != 0 else default
    except (TypeError, ValueError):
        return default


def truncate_text(text: str, max_length: int = 50) -> str:
```

```
    """
    Truncate text to a maximum length.

    Args:
        text: Text to truncate
        max_length: Maximum length

    Returns:
        Truncated text
    """
    if len(text) <= max_length:
        return text
    return text[:max_length-3] + "..."


def format_file_size(size_bytes: int) -> str:
    """
    Format file size in human-readable format.

    Args:
        size_bytes: Size in bytes

    Returns:
        Formatted size string
    """
    if size_bytes == 0:
        return "0 B"

    size_names = ["B", "KB", "MB", "GB"]
    i = 0
    while size_bytes >= 1024 and i < len(size_names) - 1:
        size_bytes /= 1024.0
        i += 1

    return f"{size_bytes:.1f} {size_names[i]}"


def validate_age(age_str: str) -> bool:
    """
    Validate if a string represents a valid age.

    Args:
        age_str: String to validate as age

    Returns:
        True if valid age, False otherwise
    """
    try:
        age = int(age_str)
        return 0 <= age <= 120
    except (ValueError, TypeError):
        return False


def validate_gender(gender_str: str) -> bool:
    """
    Validate if a string represents a valid gender.

    Args:
        gender_str: String to validate as gender

    Returns:
        True if valid gender, False otherwise
    """
    if not gender_str:
```

```
        return False

    valid_genders = [
        'male', 'female', 'm', 'f', 'other', 'prefer not to say',
        'Male', 'Female', 'M', 'F', 'Other', 'Prefer not to say'
    ]

    return gender_str.strip() in valid_genders


def normalize_text(text: str) -> str:
    """
    Normalize text by removing extra whitespace and converting to lowercase.

    Args:
        text: Text to normalize

    Returns:
        Normalized text
    """
    if not text:
        return ""

    # Remove extra whitespace and convert to lowercase
    normalized = " ".join(text.strip().split()).lower()
    return normalized


def count_words(text: str) -> int:
    """
    Count words in text.

    Args:
        text: Text to count words in

    Returns:
        Number of words
    """
    if not text:
        return 0

    return len(text.split())


def get_file_extension(file_path: str) -> str:
    """
    Get file extension from file path.

    Args:
        file_path: Path to file

    Returns:
        File extension (including dot)
    """
    return os.path.splitext(file_path)[1].lower()


def ensure_directory_exists(directory_path: str) -> bool:
    """
    Ensure a directory exists, creating it if necessary.

    Args:
        directory_path: Path to directory

    Returns:
```

```
        True if directory exists or was created, False otherwise
    """
    try:
        if not os.path.exists(directory_path):
            os.makedirs(directory_path)
        return True
    except Exception:
        return False


def is_valid_filename(filename: str) -> bool:
    """
    Check if a filename is valid.

    Args:
        filename: Filename to validate

    Returns:
        True if valid filename, False otherwise
    """
    if not filename:
        return False

    # Check for invalid characters
    invalid_chars = '<>:"/\\|?*'
    return not any(char in filename for char in invalid_chars)


def format_duration(seconds: float) -> str:
    """
    Format duration in seconds to human-readable format.

    Args:
        seconds: Duration in seconds

    Returns:
        Formatted duration string
    """
    if seconds < 60:
        return f"{seconds:.1f} seconds"
    elif seconds < 3600:
        minutes = seconds / 60
        return f"{minutes:.1f} minutes"
    else:
        hours = seconds / 3600
        return f"{hours:.1f} hours"


def print_separator(char: str = "-", length: int = 60):
    """Print a separator line."""
    print(char * length)


def print_bullet_list(items: List[str], indent: int = 2):
    """
    Print a bullet list.

    Args:
        items: List of items to print
        indent: Number of spaces to indent
    """
    indent_str = " " * indent
    for item in items:
        print(f"{indent_str} {item}")

```

```

def print_key_value_pairs(data: Dict[str, Any], indent: int = 2):
    """
    Print key-value pairs in a formatted way.

    Args:
        data: Dictionary of key-value pairs
        indent: Number of spaces to indent
    """
    indent_str = " " * indent
    for key, value in data.items():
        print(f"{indent_str}{key}: {value}")
```

10. Test Suite

The test suite provides comprehensive unit testing for all major components of the application.

Each module has corresponding test files that validate functionality, edge cases, and error handling.

File: test_data_loader.py

```
#!/usr/bin/env python3
"""
Unit Tests for Data Loader Module
Author: Student Developer
Description: Comprehensive unit tests for CSV data loading and validation functionality.
"""

import unittest
import tempfile
import os
import csv
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from data_loader import DataLoader


class TestDataLoader(unittest.TestCase):
    """Test cases for the DataLoader class."""

    def setUp(self):
        """Set up test fixtures."""
        self.data_loader = DataLoader()
        self.temp_dir = tempfile.mkdtemp()

    def tearDown(self):
        """Clean up test fixtures."""
        # Clean up temporary files
        for file in os.listdir(self.temp_dir):
            os.remove(os.path.join(self.temp_dir, file))
        os.rmdir(self.temp_dir)

    def test_validate_file_existing(self):
        """Test file validation with existing file."""
        # Create a temporary file
        temp_file = os.path.join(self.temp_dir, "test.csv")
        with open(temp_file, 'w') as f:
            f.write("test data")

        result = self.data_loader._validate_file(temp_file)
        self.assertTrue(result)

    def test_validate_file_nonexistent(self):
        """Test file validation with non-existent file."""
        result = self.data_loader._validate_file("nonexistent_file.csv")
        self.assertFalse(result)

    def test_validate_file_empty(self):
        """Test file validation with empty file."""
```

```
        temp_file = os.path.join(self.temp_dir, "empty.csv")
        with open(temp_file, 'w') as f:
            pass  # Create empty file

        result = self.data_loader._validate_file(temp_file)
        self.assertFalse(result)

    def test_validate_file_too_large(self):
        """Test file validation with file too large."""
        temp_file = os.path.join(self.temp_dir, "large.csv")
        # Create a file larger than 50MB
        with open(temp_file, 'w') as f:
            f.write("x" * (51 * 1024 * 1024))  # 51MB

        result = self.data_loader._validate_file(temp_file)
        self.assertFalse(result)

    def test_read_csv_with_encoding_utf8(self):
        """Test CSV reading with UTF-8 encoding."""
        temp_file = os.path.join(self.temp_dir, "test_utf8.csv")

        # Create test CSV data
        test_data = [
            {'name': 'John', 'age': '25', 'city': 'New York'},
            {'name': 'Jane', 'age': '30', 'city': 'Los Angeles'},
            {'name': 'Bob', 'age': '35', 'city': 'Chicago'}
        ]

        with open(temp_file, 'w', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=['name', 'age', 'city'])
            writer.writeheader()
            writer.writerows(test_data)

        result = self.data_loader._read_csv_with_encoding(temp_file, 'utf-8')
        self.assertIsNotNone(result)
        self.assertEqual(len(result), 3)
        self.assertEqual(result[0]['name'], 'John')

    def test_read_csv_with_encoding_latin1(self):
        """Test CSV reading with Latin-1 encoding."""
        temp_file = os.path.join(self.temp_dir, "test_latin1.csv")

        # Create test CSV data
        test_data = [
            {'name': 'Jos', 'age': '25', 'city': 'Madrid'},
            {'name': 'Franois', 'age': '30', 'city': 'Paris'}
        ]

        with open(temp_file, 'w', newline='', encoding='latin-1') as f:
            writer = csv.DictWriter(f, fieldnames=['name', 'age', 'city'])
            writer.writeheader()
            writer.writerows(test_data)

        result = self.data_loader._read_csv_with_encoding(temp_file, 'latin-1')
        self.assertIsNotNone(result)
        self.assertEqual(len(result), 2)

    def test_clean_and_validate_data_valid(self):
        """Test data cleaning and validation with valid data."""
        test_data = [
            {'age': '25', 'gender': 'Male', 'region': 'North'},
            {'age': '30', 'gender': 'Female', 'region': 'South'},
            {'age': '35', 'gender': 'Male', 'region': 'East'}
        ]

```

```
        result = self.data_loader._clean_and_validate_data(test_data)
        self.assertIsNotNone(result)
        self.assertEqual(len(result), 3)

    def test_clean_and_validate_data_invalid_age(self):
        """Test data cleaning with invalid age data."""
        test_data = [
            {'age': '25', 'gender': 'Male', 'region': 'North'},
            {'age': '150', 'gender': 'Female', 'region': 'South'},  # Invalid age
            {'age': '35', 'gender': 'Male', 'region': 'East'}
        ]

        result = self.data_loader._clean_and_validate_data(test_data)
        self.assertIsNotNone(result)
        # Should filter out invalid age
        self.assertEqual(len(result), 2)

    def test_clean_and_validate_data_invalid_gender(self):
        """Test data cleaning with invalid gender data."""
        test_data = [
            {'age': '25', 'gender': 'Male', 'region': 'North'},
            {'age': '30', 'gender': 'Invalid', 'region': 'South'},  # Invalid gender
            {'age': '35', 'gender': 'Female', 'region': 'East'}
        ]

        result = self.data_loader._clean_and_validate_data(test_data)
        self.assertIsNotNone(result)
        # Should filter out invalid gender
        self.assertEqual(len(result), 2)

    def test_clean_and_validate_data_empty(self):
        """Test data cleaning with empty data."""
        result = self.data_loader._clean_and_validate_data([])
        self.assertIsNone(result)

    def test_validate_row_valid(self):
        """Test row validation with valid data."""
        valid_row = {'age': '25', 'gender': 'Male', 'region': 'North'}
        result = self.data_loader._validate_row(valid_row, 1)
        self.assertTrue(result)

    def test_validate_row_invalid_age(self):
        """Test row validation with invalid age."""
        invalid_row = {'age': '150', 'gender': 'Male', 'region': 'North'}
        result = self.data_loader._validate_row(invalid_row, 1)
        self.assertFalse(result)

    def test_validate_row_invalid_gender(self):
        """Test row validation with invalid gender."""
        invalid_row = {'age': '25', 'gender': 'Invalid', 'region': 'North'}
        result = self.data_loader._validate_row(invalid_row, 1)
        self.assertFalse(result)

    def test_validate_row_insufficient_fields(self):
        """Test row validation with insufficient fields."""
        invalid_row = {'age': '25'}  # Only one field
        result = self.data_loader._validate_row(invalid_row, 1)
        self.assertFalse(result)

    def test_get_data_summary(self):
        """Test data summary generation."""
        test_data = [
            {'age': '25', 'gender': 'Male', 'region': 'North'},
            {'age': '30', 'gender': 'Female', 'region': 'South'},
            {'age': '35', 'gender': 'Male', 'region': 'East'}
```

```
        ]

        summary = self.data_loader.get_data_summary(test_data)

        self.assertIn('total_rows', summary)
        self.assertIn('columns', summary)
        self.assertIn('missing_values', summary)
        self.assertIn('unique_values', summary)

        self.assertEqual(summary['total_rows'], 3)
        self.assertEqual(len(summary['columns']), 3)

    def test_export_sample_data(self):
        """Test sample data export functionality."""
        temp_file = os.path.join(self.temp_dir, "sample_export.csv")

        result = self.data_loader.export_sample_data(temp_file)
        self.assertTrue(result)
        self.assertTrue(os.path.exists(temp_file))

        # Verify the exported file has content
        with open(temp_file, 'r', encoding='utf-8') as f:
            content = f.read()
            self.assertIn('age', content)
            self.assertIn('gender', content)
            self.assertIn('region', content)

    def test_load_csv_integration(self):
        """Test complete CSV loading integration."""
        temp_file = os.path.join(self.temp_dir, "integration_test.csv")

        # Create test CSV data
        test_data = [
            {'age': '25', 'gender': 'Male', 'region': 'North', 'satisfaction': 'High'},
            {'age': '30', 'gender': 'Female', 'region': 'South', 'satisfaction': 'Medium'},
            {'age': '35', 'gender': 'Male', 'region': 'East', 'satisfaction': 'Low'}
        ]

        with open(temp_file, 'w', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=['age', 'gender', 'region',
'satisfaction'])
            writer.writeheader()
            writer.writerows(test_data)

        result = self.data_loader.load_csv(temp_file)
        self.assertIsNotNone(result)
        self.assertEqual(len(result), 3)
        self.assertEqual(result[0]['age'], '25')
        self.assertEqual(result[0]['gender'], 'Male')


if __name__ == '__main__':
    unittest.main()
```

File: test_sentiment_analyzer.py

```
#!/usr/bin/env python3
"""
Unit Tests for Sentiment Analyzer Module
Author: Student Developer
Description: Comprehensive unit tests for sentiment analysis functionality.
"""

import unittest
```

```
   9: import sys
  10: import os
  11: sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
  12: from sentiment_analyzer import SentimentAnalyzer
  13:
  14:
  15: class TestSentimentAnalyzer(unittest.TestCase):
  16:     """Test cases for the SentimentAnalyzer class."""
  17:
  18:     def setUp(self):
  19:         """Set up test fixtures."""
  20:         self.sentiment_analyzer = SentimentAnalyzer()
  21:         self.test_data = [
  22:             {'feedback': 'This is excellent! I love it.', 'satisfaction': 'Very Satisfied'},
  23:             {'feedback': 'Terrible experience, very bad service.', 'satisfaction':
'Dissatisfi
ed'},
  24:             {'feedback': 'It was okay, nothing special.', 'satisfaction': 'Neutral'},
  25:             {'feedback': 'Great product and amazing support!', 'satisfaction': 'Very
Satisfied
'},
  26:             {'feedback': 'Poor quality and bad service.', 'satisfaction': 'Dissatisfied'}
  27:         ]
  28:
  29:     def test_initialization(self):
  30:         """Test SentimentAnalyzer initialization."""
  31:         self.assertIsInstance(self.sentiment_analyzer.positive_keywords, dict)
  32:         self.assertIsInstance(self.sentiment_analyzer.negative_keywords, dict)
  33:         self.assertIsInstance(self.sentiment_analyzer.neutral_keywords, dict)
  34:         self.assertIsInstance(self.sentiment_analyzer.negation_words, set)
  35:         self.assertIsInstance(self.sentiment_analyzer.intensifier_words, dict)
  36:
  37:         # Check that keywords are loaded
  38:         self.assertGreater(len(self.sentiment_analyzer.positive_keywords), 0)
  39:         self.assertGreater(len(self.sentiment_analyzer.negative_keywords), 0)
  40:
  41:     def test_analyze_text_positive(self):
  42:         """Test sentiment analysis with positive text."""
  43:         text = "This is excellent! I love the product."
  44:         result = self.sentiment_analyzer.analyze_text(text)
  45:
  46:         self.assertIsInstance(result, dict)
  47:         self.assertIn('sentiment', result)
  48:         self.assertIn('score', result)
  49:         self.assertIn('positive_words', result)
  50:         self.assertIn('negative_words', result)
  51:         self.assertIn('confidence', result)
  52:
  53:         self.assertEqual(result['sentiment'], 'positive')
  54:         self.assertGreater(result['score'], 0)
  55:         self.assertGreater(len(result['positive_words']), 0)
  56:
  57:     def test_analyze_text_negative(self):
  58:         """Test sentiment analysis with negative text."""
  59:         text = "This is terrible! I hate the service."
  60:         result = self.sentiment_analyzer.analyze_text(text)
  61:
  62:         self.assertEqual(result['sentiment'], 'negative')
  63:         self.assertLess(result['score'], 0)
  64:         self.assertGreater(len(result['negative_words']), 0)
  65:
  66:     def test_analyze_text_neutral(self):
  67:         """Test sentiment analysis with neutral text."""
  68:         text = "It was okay, nothing special."
  69:         result = self.sentiment_analyzer.analyze_text(text)
```

```
70:
71:         self.assertEqual(result['sentiment'], 'neutral')
72:         self.assertAlmostEqual(result['score'], 0, places=1)
73:
74:     def test_analyze_text_with_negation(self):
75:         """Test sentiment analysis with negation."""
76:         text = "This is not good at all."
77:         result = self.sentiment_analyzer.analyze_text(text)
78:
79:         # Should be negative due to negation
80:         self.assertEqual(result['sentiment'], 'negative')
81:         self.assertLess(result['score'], 0)
82:
83:     def test_analyze_text_with_intensifier(self):
84:         """Test sentiment analysis with intensifier."""
85:         text = "This is very excellent!"
86:         result = self.sentiment_analyzer.analyze_text(text)
87:
88:         self.assertEqual(result['sentiment'], 'positive')
89:         self.assertGreater(result['score'], 0)
90:
91:     def test_analyze_text_empty(self):
92:         """Test sentiment analysis with empty text."""
93:         result = self.sentiment_analyzer.analyze_text("")
94:
95:         self.assertEqual(result['sentiment'], 'neutral')
96:         self.assertEqual(result['score'], 0)
97:         self.assertEqual(len(result['positive_words']), 0)
98:         self.assertEqual(len(result['negative_words']), 0)
99:
100:     def test_analyze_text_none(self):
101:         """Test sentiment analysis with None text."""
102:         result = self.sentiment_analyzer.analyze_text(None)
103:
104:         self.assertEqual(result['sentiment'], 'neutral')
105:         self.assertEqual(result['score'], 0)
106:
107:     def test_analyze_column(self):
108:         """Test sentiment analysis for a column."""
109:         result = self.sentiment_analyzer.analyze_column(self.test_data, 'feedback')
110:
111:         self.assertIsInstance(result, dict)
112:         self.assertIn('positive', result)
113:         self.assertIn('negative', result)
114:         self.assertIn('neutral', result)
115:         self.assertIn('positive_pct', result)
116:         self.assertIn('negative_pct', result)
117:         self.assertIn('neutral_pct', result)
118:         self.assertIn('avg_score', result)
119:         self.assertIn('total_responses', result)
120:
121:         # Check that percentages sum to approximately 100
122:         total_pct = result['positive_pct'] + result['negative_pct'] + result['neutral_pct']
123:         self.assertAlmostEqual(total_pct, 100.0, places=1)
124:
125:     def test_analyze_column_empty_data(self):
126:         """Test sentiment analysis with empty data."""
127:         result = self.sentiment_analyzer.analyze_column([], 'feedback')
128:
129:         self.assertEqual(result['positive'], 0)
130:         self.assertEqual(result['negative'], 0)
131:         self.assertEqual(result['neutral'], 0)
132:         self.assertEqual(result['total_responses'], 0)
133:
134:     def test_analyze_column_missing_column(self):
```

```
        """Test sentiment analysis with missing column."""
        result = self.sentiment_analyzer.analyze_column(self.test_data, 'nonexistent_column')

        self.assertEqual(result['positive'], 0)
        self.assertEqual(result['negative'], 0)
        self.assertEqual(result['neutral'], 0)
        self.assertEqual(result['total_responses'], 0)

    def test_analyze_all_text_columns(self):
        """Test sentiment analysis for all text columns."""
        result = self.sentiment_analyzer.analyze_all_text_columns(self.test_data)

        self.assertIsInstance(result, dict)
        self.assertIn('feedback', result)

        feedback_result = result['feedback']
        self.assertIn('positive', feedback_result)
        self.assertIn('negative', feedback_result)
        self.assertIn('neutral', feedback_result)

    def test_clean_text(self):
        """Test text cleaning functionality."""
        dirty_text = "  This   is   a   TEST!!!   "
        cleaned = self.sentiment_analyzer._clean_text(dirty_text)

        self.assertEqual(cleaned, "this is a test")

    def test_extract_words(self):
        """Test word extraction."""
        text = "This is a test sentence."
        words = self.sentiment_analyzer._extract_words(text)

        self.assertIsInstance(words, list)
        self.assertEqual(len(words), 5)
        self.assertIn('This', words)
        self.assertIn('is', words)
        self.assertIn('a', words)
        self.assertIn('test', words)
        self.assertIn('sentence.', words)

    def test_is_negated(self):
        """Test negation detection."""
        words = ['This', 'is', 'not', 'good']

        # Check if 'good' is negated
        is_negated = self.sentiment_analyzer._is_negated(words, 3)  # Index of 'good'
        self.assertTrue(is_negated)

        # Check if 'This' is negated (should not be)
        is_negated = self.sentiment_analyzer._is_negated(words, 0)  # Index of 'This'
        self.assertFalse(is_negated)

    def test_categorize_sentiment(self):
        """Test sentiment categorization."""
        # Test positive sentiment
        sentiment = self.sentiment_analyzer._categorize_sentiment(2.0)
        self.assertEqual(sentiment, 'positive')

        # Test negative sentiment
        sentiment = self.sentiment_analyzer._categorize_sentiment(-2.0)
        self.assertEqual(sentiment, 'negative')

        # Test neutral sentiment
        sentiment = self.sentiment_analyzer._categorize_sentiment(0.5)
        self.assertEqual(sentiment, 'neutral')
```

```
 200:
 201:         sentiment = self.sentiment_analyzer._categorize_sentiment(-0.5)
 202:         self.assertEqual(sentiment, 'neutral')
 203:
 204:     def test_get_sentiment_summary(self):
 205:         """Test sentiment summary generation."""
 206:         # Create sentiment results
 207:         sentiment_results = [
 208:             {'sentiment': 'positive', 'score': 2.0, 'positive_words': ['excellent'],
'negative
_words': [], 'confidence': 0.8},
 209:             {'sentiment': 'negative', 'score': -1.5, 'positive_words': [], 'negative_words':
[
'terrible'], 'confidence': 0.7},
 210:             {'sentiment': 'neutral', 'score': 0.0, 'positive_words': [], 'negative_words':
[],
 'confidence': 0.5}
 211:         ]
 212:
 213:         summary = self.sentiment_analyzer.get_sentiment_summary(sentiment_results)
 214:
 215:         self.assertIsInstance(summary, dict)
 216:         self.assertIn('total_responses', summary)
 217:         self.assertIn('sentiment_distribution', summary)
 218:         self.assertIn('top_positive_words', summary)
 219:         self.assertIn('top_negative_words', summary)
 220:         self.assertIn('average_confidence', summary)
 221:         self.assertIn('average_score', summary)
 222:
 223:         self.assertEqual(summary['total_responses'], 3)
 224:         self.assertEqual(summary['sentiment_distribution']['positive']['count'], 1)
 225:         self.assertEqual(summary['sentiment_distribution']['negative']['count'], 1)
 226:         self.assertEqual(summary['sentiment_distribution']['neutral']['count'], 1)
 227:
 228:     def test_export_sentiment_report(self):
 229:         """Test sentiment report export."""
 230:         sentiment_results = {
 231:             'feedback': {
 232:                 'positive': 2,
 233:                 'negative': 2,
 234:                 'neutral': 1,
 235:                 'positive_pct': 40.0,
 236:                 'negative_pct': 40.0,
 237:                 'neutral_pct': 20.0,
 238:                 'avg_score': 0.2,
 239:                 'total_responses': 5
 240:             }
 241:         }
 242:
 243:         # Test successful export
 244:         result = self.sentiment_analyzer.export_sentiment_report(sentiment_results,
"test_sent
iment_report.txt")
 245:         self.assertTrue(result)
 246:
 247:         # Clean up
 248:         import os
 249:         if os.path.exists("test_sentiment_report.txt"):
 250:             os.remove("test_sentiment_report.txt")
 251:
 252:     def test_analyze_text_complex_sentence(self):
 253:         """Test sentiment analysis with complex sentence."""
 254:         text = "This product is absolutely fantastic and I would highly recommend it to
everyo
ne!"
```

```
255:         result = self.sentiment_analyzer.analyze_text(text)
256:
257:         self.assertEqual(result['sentiment'], 'positive')
258:         self.assertGreater(result['score'], 0)
259:         self.assertGreater(len(result['positive_words']), 0)
260:
261:     def test_analyze_text_mixed_sentiment(self):
262:         """Test sentiment analysis with mixed sentiment."""
263:         text = "The product is good but the service is terrible."
264:         result = self.sentiment_analyzer.analyze_text(text)
265:
266:         # Should have both positive and negative words
267:         self.assertGreater(len(result['positive_words']), 0)
268:         self.assertGreater(len(result['negative_words']), 0)
269:
270:     def test_analyze_text_no_sentiment_words(self):
271:         """Test sentiment analysis with no sentiment words."""
272:         text = "The product arrived on time and was delivered to the correct address."
273:         result = self.sentiment_analyzer.analyze_text(text)
274:
275:         self.assertEqual(result['sentiment'], 'neutral')
276:         self.assertEqual(len(result['positive_words']), 0)
277:         self.assertEqual(len(result['negative_words']), 0)
278:
279:
280: if __name__ == '__main__':
281:     unittest.main()
```

File: test_stats_analyzer.py

```
 1: #!/usr/bin/env python3
 2: """
 3: Unit Tests for Statistical Analyzer Module
 4: Author: Student Developer
 5: Description: Comprehensive unit tests for statistical analysis functionality.
 6: """
 7:
 8: import unittest
 9: import sys
10: import os
11: sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
12: from stats_analyzer import StatsAnalyzer
13:
14:
15: class TestStatsAnalyzer(unittest.TestCase):
16:     """Test cases for the StatsAnalyzer class."""
17:
18:     def setUp(self):
19:         """Set up test fixtures."""
20:         self.test_data = [
21:             {'age': '25', 'gender': 'Male', 'region': 'North', 'satisfaction': 'High'},
22:             {'age': '30', 'gender': 'Female', 'region': 'South', 'satisfaction': 'Medium'},
23:             {'age': '35', 'gender': 'Male', 'region': 'East', 'satisfaction': 'Low'},
24:             {'age': '28', 'gender': 'Female', 'region': 'North', 'satisfaction': 'High'},
25:             {'age': '42', 'gender': 'Male', 'region': 'South', 'satisfaction': 'Medium'},
26:             {'age': '33', 'gender': 'Female', 'region': 'East', 'satisfaction': 'Low'}
27:         ]
28:         self.stats_analyzer = StatsAnalyzer(self.test_data)
29:
30:     def test_initialization(self):
31:         """Test StatsAnalyzer initialization."""
32:         self.assertEqual(self.stats_analyzer.total_responses, 6)
33:         self.assertEqual(len(self.stats_analyzer.columns), 4)
34:         self.assertIn('age', self.stats_analyzer.columns)
```

```
        self.assertIn('gender', self.stats_analyzer.columns)
        self.assertIn('region', self.stats_analyzer.columns)
        self.assertIn('satisfaction', self.stats_analyzer.columns)

    def test_cross_tabulate_valid_columns(self):
        """Test cross-tabulation with valid columns."""
        crosstab = self.stats_analyzer.cross_tabulate('gender', 'satisfaction')

        self.assertIsInstance(crosstab, list)
        self.assertGreater(len(crosstab), 1)  # Should have header + data rows

        # Check header row
        header = crosstab[0]
        self.assertIn('', header)  # First column should be empty (row labels)

        # Check data rows
        for row in crosstab[1:]:
            self.assertIsInstance(row, list)
            self.assertEqual(len(row), len(header))

    def test_cross_tabulate_invalid_columns(self):
        """Test cross-tabulation with invalid columns."""
        with self.assertRaises(ValueError):
            self.stats_analyzer.cross_tabulate('invalid_column', 'gender')

    def test_chi_square_test_valid_data(self):
        """Test chi-square test with valid data."""
        result = self.stats_analyzer.chi_square_test('gender', 'satisfaction')

        self.assertIsInstance(result, dict)
        self.assertIn('chi_square', result)
        self.assertIn('p_value', result)
        self.assertIn('df', result)
        self.assertIn('significant', result)

        # Check data types
        self.assertIsInstance(result['chi_square'], float)
        self.assertIsInstance(result['p_value'], float)
        self.assertIsInstance(result['df'], int)
        self.assertIsInstance(result['significant'], bool)

    def test_chi_square_test_insufficient_data(self):
        """Test chi-square test with insufficient data."""
        # Create data with only one value per category
        limited_data = [
            {'gender': 'Male', 'satisfaction': 'High'},
            {'gender': 'Female', 'satisfaction': 'Low'}
        ]
        limited_analyzer = StatsAnalyzer(limited_data)

        result = limited_analyzer.chi_square_test('gender', 'satisfaction')

        self.assertIn('error', result)
        self.assertEqual(result['chi_square'], 0.0)
        self.assertEqual(result['p_value'], 1.0)

    def test_correlation_analysis_numeric_data(self):
        """Test correlation analysis with numeric data."""
        numeric_data = [
            {'age': '25', 'satisfaction_score': '8'},
            {'age': '30', 'satisfaction_score': '7'},
            {'age': '35', 'satisfaction_score': '6'},
            {'age': '28', 'satisfaction_score': '9'},
            {'age': '42', 'satisfaction_score': '5'}
        ]
```

```
        numeric_analyzer = StatsAnalyzer(numeric_data)

        result = numeric_analyzer.correlation_analysis('age', 'satisfaction_score')

        self.assertIsInstance(result, dict)
        self.assertIn('correlation', result)
        self.assertIn('sample_size', result)
        self.assertIn('strength', result)

        self.assertIsInstance(result['correlation'], float)
        self.assertIsInstance(result['sample_size'], int)
        self.assertIsInstance(result['strength'], str)

    def test_correlation_analysis_insufficient_data(self):
        """Test correlation analysis with insufficient data."""
        insufficient_data = [
            {'age': '25', 'score': '8'}
        ]
        insufficient_analyzer = StatsAnalyzer(insufficient_data)

        result = insufficient_analyzer.correlation_analysis('age', 'score')

        self.assertIn('error', result)
        self.assertEqual(result['correlation'], 0.0)
        self.assertEqual(result['sample_size'], 0)

    def test_calculate_correlation(self):
        """Test correlation calculation."""
        x = [1, 2, 3, 4, 5]
        y = [2, 4, 6, 8, 10]  # Perfect positive correlation

        correlation = self.stats_analyzer._calculate_correlation(x, y)
        self.assertAlmostEqual(correlation, 1.0, places=5)

    def test_calculate_correlation_negative(self):
        """Test negative correlation calculation."""
        x = [1, 2, 3, 4, 5]
        y = [10, 8, 6, 4, 2]  # Perfect negative correlation

        correlation = self.stats_analyzer._calculate_correlation(x, y)
        self.assertAlmostEqual(correlation, -1.0, places=5)

    def test_calculate_correlation_no_correlation(self):
        """Test correlation calculation with no correlation."""
        x = [1, 2, 3, 4, 5]
        y = [1, 1, 1, 1, 1]  # No variation in y

        correlation = self.stats_analyzer._calculate_correlation(x, y)
        self.assertEqual(correlation, 0.0)

    def test_interpret_correlation(self):
        """Test correlation interpretation."""
        # Test very strong positive correlation
        strength = self.stats_analyzer._interpret_correlation(0.9)
        self.assertEqual(strength, "Very Strong")

        # Test strong positive correlation
        strength = self.stats_analyzer._interpret_correlation(0.7)
        self.assertEqual(strength, "Strong")

        # Test moderate correlation
        strength = self.stats_analyzer._interpret_correlation(0.5)
        self.assertEqual(strength, "Moderate")

        # Test weak correlation
```

```
        strength = self.stats_analyzer._interpret_correlation(0.3)
        self.assertEqual(strength, "Weak")

        # Test very weak correlation
        strength = self.stats_analyzer._interpret_correlation(0.1)
        self.assertEqual(strength, "Very Weak")

    def test_analyze_response_patterns(self):
        """Test response pattern analysis."""
        patterns = self.stats_analyzer.analyze_response_patterns()

        self.assertIsInstance(patterns, dict)
        self.assertIn('common_combinations', patterns)
        self.assertIn('response_clusters', patterns)
        self.assertIn('outliers', patterns)

        self.assertIsInstance(patterns['common_combinations'], list)
        self.assertIsInstance(patterns['response_clusters'], list)
        self.assertIsInstance(patterns['outliers'], list)

    def test_get_statistical_summary(self):
        """Test statistical summary generation."""
        summary = self.stats_analyzer.get_statistical_summary()

        self.assertIsInstance(summary, dict)
        self.assertIn('total_responses', summary)
        self.assertIn('total_columns', summary)
        self.assertIn('numeric_columns', summary)
        self.assertIn('categorical_columns', summary)
        self.assertIn('statistical_tests', summary)

        self.assertEqual(summary['total_responses'], 6)
        self.assertEqual(summary['total_columns'], 4)
        self.assertIsInstance(summary['numeric_columns'], list)
        self.assertIsInstance(summary['categorical_columns'], list)

    def test_perform_multiple_chi_square_tests(self):
        """Test multiple chi-square tests."""
        results = self.stats_analyzer.perform_multiple_chi_square_tests('gender')

        self.assertIsInstance(results, list)

        for result in results:
            self.assertIn('column1', result)
            self.assertIn('column2', result)
            self.assertEqual(result['column1'], 'gender')
            self.assertNotEqual(result['column2'], 'gender')

    def test_calculate_expected_frequencies(self):
        """Test expected frequency calculation."""
        observed = [[2, 1], [1, 2]]  # 2x2 contingency table

        expected = self.stats_analyzer._calculate_expected_frequencies(observed)

        self.assertIsInstance(expected, list)
        self.assertEqual(len(expected), 2)
        self.assertEqual(len(expected[0]), 2)

        # Check that expected frequencies sum to total
        total_observed = sum(sum(row) for row in observed)
        total_expected = sum(sum(row) for row in expected)
        self.assertAlmostEqual(total_expected, total_observed, places=5)

    def test_chi_square_p_value(self):
        """Test chi-square p-value calculation."""
```

```
        # Test with small chi-square value
        p_value = self.stats_analyzer._chi_square_p_value(1.0, 2)
        self.assertGreater(p_value, 0.0)
        self.assertLessEqual(p_value, 1.0)

        # Test with large chi-square value
        p_value = self.stats_analyzer._chi_square_p_value(10.0, 2)
        self.assertGreaterEqual(p_value, 0.0)
        self.assertLess(p_value, 1.0)

        # Test with zero degrees of freedom
        p_value = self.stats_analyzer._chi_square_p_value(5.0, 0)
        self.assertEqual(p_value, 1.0)


if __name__ == '__main__':
    unittest.main()
```

11. Sample Data

The project includes sample survey data for testing and demonstration purposes. The sample data contains realistic survey responses with various demographic and response patterns.

File: sample_survey.csv

```
  1: age,gender,region,education,satisfaction,feedback,recommend
  2: 25,Female,North,Bachelor,Very Satisfied,Great experience with the product!,Yes
  3: 32,Male,South,Master,Satisfied,Good but could be better.,Yes
  4: 45,Female,East,High School,Neutral,It was okay, nothing special.,Maybe
  5: 28,Male,West,Bachelor,Dissatisfied,Poor quality and bad service.,No
  6: 35,Female,North,PhD,Very Satisfied,Excellent product and amazing support!,Yes
  7: 42,Male,South,Bachelor,Satisfied,Overall good experience.,Yes
  8: 29,Female,East,Master,Neutral,Average service, nothing exceptional.,Maybe
  9: 38,Male,West,High School,Dissatisfied,Very disappointed with the service.,No
 10: 31,Female,North,Bachelor,Very Satisfied,Love the product! Highly recommend.,Yes
 11: 27,Male,South,Master,Satisfied,Good value for money.,Yes
 12: 50,Female,East,PhD,Neutral,It meets basic expectations.,Maybe
 13: 33,Male,West,Bachelor,Dissatisfied,Not worth the money at all.,No
 14: 26,Female,North,High School,Very Satisfied,Amazing experience!,Yes
 15: 39,Male,South,Bachelor,Satisfied,Decent product, would buy again.,Yes
 16: 44,Female,East,Master,Neutral,It's okay, nothing special.,Maybe
 17: 36,Male,West,PhD,Dissatisfied,Terrible customer service.,No
 18: 23,Female,North,Bachelor,Very Satisfied,Best purchase ever!,Yes
 19: 41,Male,South,High School,Satisfied,Good quality product.,Yes
 20: 30,Female,East,Bachelor,Neutral,Average experience.,Maybe
 21: 47,Male,West,Master,Dissatisfied,Complete waste of money.,No
```

13. Project Documentation

The README.md file provides comprehensive documentation including installation instructions, usage examples, technical specifications, and development guidelines.

File: README.md

```
  1: # Survey Data Analyzer and Response Processor
  2:
  3: **A comprehensive Command Line Interface (CLI) tool for analyzing survey responses from CSV
fi
les**
  4:
  5: ## Project Overview
  6:
  7: The Survey Data Analyzer is a modular Python CLI application designed to perform structured
st
atistical analysis, basic text sentiment analysis, identify response patterns, and generate
readable
 reports from survey data stored in CSV format.
  8:
  9: ### Key Features
 10:
 11: - **CSV Input Handling**: Load and validate structured survey data from CSV files
 12: - **Survey Summary**: Calculate response counts, percentages, and demographic breakdowns
 13: - **Statistical Analysis**: Perform cross-tabulation and Chi-Square tests
 14: - **Sentiment Analysis**: Analyze text responses using keyword-based sentiment scoring
 15: - **Pattern Recognition**: Identify correlations and trends between survey responses
 16: - **Report Generation**: Create comprehensive analysis reports in text format
```

Project Structure

```
SurveyAnalyzer/
 main.py                    # CLI menu and application entry point
 data_loader.py            # CSV loading and validation
 survey_summary.py         # Response counts and demographics
 stats_analyzer.py         # Cross-tabulation and chi-square tests
 sentiment_analyzer.py     # Text sentiment analysis
 pattern_detector.py       # Pattern correlation detection
 report_generator.py       # Report generation and file output
 utils.py                  # Utility functions and helpers
 sample_survey.csv         # Example input file
 requirements.txt          # Project dependencies (stdlib only)
 README.md                 # This documentation
 test/                     # Unit tests
     test_data_loader.py
     test_stats_analyzer.py
     test_sentiment_analyzer.py
```

Quick Start

Prerequisites

- Python 3.8 or higher
- No external dependencies required (uses only standard library)

Installation

1. Clone or download the project files
2. Navigate to the SurveyAnalyzer directory
3. Run the application:

```bash
python main.py
```

Usage Example

1. **Load Survey Data**: Choose option 1 and provide the path to your CSV file
2. **View Summary Statistics**: Choose option 2 to see demographic breakdowns
3. **Analyze Sentiment**: Choose option 3 for text sentiment analysis
4. **Cross-tabulate Results**: Choose option 4 for statistical analysis
5. **Detect Patterns**: Choose option 5 to find correlations
6. **Generate Report**: Choose option 6 to create a comprehensive report

Sample Data

The project includes `sample_survey.csv` with example data containing:
- Age, gender, region, education demographics
- Satisfaction ratings
- Text feedback responses
- Recommendation preferences

Technical Requirements

Python Version
- **Minimum**: Python 3.8
- **Recommended**: Python 3.9+

Standard Library Modules Used
- `csv`: CSV file handling
- `re`: Regular expressions for text processing

- `math`: Mathematical calculations
- `statistics`: Statistical functions
- `os`: Operating system interface
- `sys`: System-specific parameters
- `datetime`: Date and time handling
- `itertools`: Iterator building blocks
- `collections`: Specialized container datatypes
- `typing`: Type hints support

Object-Oriented Design
- **Class Hierarchies**: Modular class structure for each component
- **Inheritance**: Base classes for common functionality
- **Polymorphism**: Flexible method implementations

User Guide

1. Loading Survey Data

The application accepts CSV files with the following requirements:
- **File Format**: Comma-separated values (CSV)
- **Encoding**: UTF-8, Latin-1, or CP1252
- **Size Limit**: Maximum 50MB
- **Required Columns**: At least 2 columns with valid data

2. Survey Summary Analysis

View comprehensive statistics including:
- Total response count and response rate
- Demographic breakdowns (age, gender, region, education)
- Response distributions for each question
- Data quality metrics

3. Sentiment Analysis

Analyze text responses using:
- **Positive Keywords**: excellent, amazing, great, good, etc.
- **Negative Keywords**: terrible, awful, bad, poor, etc.
- **Negation Handling**: "not good" negative sentiment
- **Intensifier Recognition**: "very good" amplified positive

4. Statistical Analysis

Perform advanced statistical tests:
- **Cross-tabulation**: Compare responses across different questions
- **Chi-Square Tests**: Determine statistical significance
- **Correlation Analysis**: Find relationships between variables

5. Pattern Detection

Identify meaningful patterns:
- **Demographic Patterns**: Age, gender, regional differences
- **Response Correlations**: Related answers across questions
- **Combination Patterns**: Common response combinations
- **Outlier Detection**: Unusual or rare responses

6. Report Generation

Generate comprehensive reports including:
- Executive summary
- Demographic analysis
- Sentiment analysis results
- Pattern detection findings
- Statistical insights
- Key findings and recommendations

```
147: ## Testing
148:
149: ### Running Tests
150:
151: ```bash
152: # Run all tests
153: python -m unittest discover test/
154:
155: # Run specific test file
156: python -m unittest test.test_data_loader
157:
158: # Run with verbose output
159: python -m unittest discover test/ -v
160: ```
161:
162: ### Test Coverage
163:
164: - **Data Loading**: File validation, encoding detection, data cleaning
165: - **Statistical Analysis**: Cross-tabulation, chi-square calculations
166: - **Sentiment Analysis**: Keyword matching, negation handling
167: - **Pattern Detection**: Correlation analysis, demographic patterns
168: - **Report Generation**: File output, formatting
169:
170: ## Technical Documentation
171:
172: ### Core Classes
173:
174: #### DataLoader
175: - **Purpose**: Handle CSV file loading and validation
176: - **Key Methods**: `load_csv()`, `_validate_file()`, `_clean_and_validate_data()`
177: - **Features**: Multi-encoding support, delimiter detection, data cleaning
178:
179: #### SurveySummary
180: - **Purpose**: Generate summary statistics and demographic breakdowns
181: - **Key Methods**: `generate_summary()`, `_analyze_demographics()`, `get_age_distribution()`
182: - **Features**: Response counting, percentage calculations, trend analysis
183:
184: #### StatsAnalyzer
185: - **Purpose**: Perform statistical analysis and hypothesis testing
186: - **Key Methods**: `cross_tabulate()`, `chi_square_test()`, `correlation_analysis()`
187: - **Features**: Cross-tabulation matrices, chi-square tests, correlation coefficients
188:
189: #### SentimentAnalyzer
190: - **Purpose**: Analyze text sentiment using keyword-based approach
191: - **Key Methods**: `analyze_text()`, `analyze_column()`, `_clean_text()`
192: - **Features**: Keyword dictionaries, negation handling, intensifier recognition
193:
194: #### PatternDetector
195: - **Purpose**: Identify patterns and correlations in survey responses
196: - **Key Methods**: `find_patterns()`, `_analyze_age_patterns()`,
`_find_correlation_patterns()
`
197: - **Features**: Demographic analysis, correlation detection, outlier identification
198:
199: #### ReportGenerator
200: - **Purpose**: Generate comprehensive analysis reports
201: - **Key Methods**: `generate_report()`, `_build_report_content()`, `_write_report_to_file()`
202: - **Features**: Structured reports, multiple sections, file output
203:
204: ### Error Handling
205:
206: The application implements robust error handling:
207: - **File Validation**: Check file existence, size, and format
208: - **Data Validation**: Validate age ranges, gender values, required fields
209: - **Graceful Degradation**: Continue processing even with partial data
```

```
 210: - **User-Friendly Messages**: Clear error messages with suggestions
 211:
 212: ### Performance Considerations
 213:
 214: - **Memory Efficient**: Process data in chunks for large files
 215: - **Fast Processing**: Optimized algorithms for statistical calculations
 216: - **Responsive UI**: Non-blocking operations with progress indicators
 217:
 218: ## Educational Features
 219:
 220: ### Learning Objectives
 221:
 222: This project demonstrates:
 223: - **Modular Design**: Clean separation of concerns
 224: - **Object-Oriented Programming**: Classes, inheritance, polymorphism
 225: - **Error Handling**: Robust exception management
 226: - **Documentation**: Comprehensive docstrings and comments
 227: - **Testing**: Unit tests for all major components
 228: - **CLI Design**: User-friendly command-line interface
 229:
 230: ### Code Quality
 231:
 232: - **Readability**: Clear variable names and function purposes
 233: - **Maintainability**: Modular structure for easy updates
 234: - **Extensibility**: Easy to add new analysis features
 235: - **Documentation**: Inline comments and comprehensive docstrings
 236:
 237: ## Contributing
 238:
 239: ### Development Guidelines
 240:
 241: 1. **Code Style**: Follow PEP 8 conventions
 242: 2. **Documentation**: Add docstrings for all functions and classes
 243: 3. **Testing**: Write unit tests for new features
 244: 4. **Error Handling**: Implement proper exception handling
 245: 5. **User Experience**: Ensure intuitive CLI interface
 246:
 247: ### Adding New Features
 248:
 249: 1. Create new module in appropriate directory
 250: 2. Implement class with clear interface
 251: 3. Add comprehensive unit tests
 252: 4. Update main.py to integrate new feature
 253: 5. Update documentation
 254:
 255: ## License
 256:
 257: This project is designed for educational purposes. Feel free to use, modify, and distribute
fo
r learning and teaching.
 258:
 259: ## Author
 260:
 261: **Student Developer** - Survey Data Analyzer Project
 262:
 263: ---
 264:
 265: **Note**: This application uses only Python standard library modules, making it easy to run
wi
thout external dependencies. Perfect for educational environments and learning Python programming
co
ncepts.
```