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 03:03 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.

Main Application - main.py

```
#!/usr/bin/env python3
"""
Survey Data Analyzer - Main CLI Application
Author: Student Developer
Description: Main entry point for the Survey Data Analyzer CLI tool.
Provides a menu-driven interface for analyzing survey data from CSV files.
"""

import sys
import os
from typing import Optional, Dict, Any
from data_loader import DataLoader
from survey_summary import SurveySummary
from stats_analyzer import StatsAnalyzer
from sentiment_analyzer import SentimentAnalyzer
from pattern_detector import PatternDetector
from report_generator import ReportGenerator
from utils import clear_screen, print_header, print_menu


class SurveyAnalyzerCLI:
"""Main CLI application class for Survey Data Analyzer."""

def __init__(self):
"""Initialize the CLI application with empty data containers."""
self.data_loader = DataLoader()
self.survey_data = None
self.survey_summary = None
self.stats_analyzer = None
self.sentiment_analyzer = None
self.pattern_detector = None
self.report_generator = None

def load_survey_data(self) -> bool:
"""Load survey data from CSV file."""
try:
print_header("Load Survey Data")
file_path = input("Enter the path to your CSV file: ").strip()

if not file_path:
print("ERROR: No file path provided.")
return False

if not os.path.exists(file_path):
print(f"ERROR: File not found: {file_path}")
return False

self.survey_data = self.data_loader.load_csv(file_path)
if self.survey_data:
print(f"SUCCESS: Successfully loaded {len(self.survey_data)} survey responses")
```

```
51: print(f"INFO: Columns: {', '.join(self.survey_data[0].keys())}") 52: return
True 53: else: 54: print("ERROR: Failed to load survey data") 55: return False
56: 57: except Exception as e: 58: print(f"ERROR: Error loading data: {str(e)}")
59: return False 60: 61: def view_summary_statistics(self): 62: """Display
summary statistics for the loaded survey data.""" 63: if not self.survey_data:
64: print("ERROR: No survey data loaded. Please load data first.") 65: return 66:
67: try: 68: print_header("Survey Summary Statistics") 69: self.survey_summary =
SurveySummary(self.survey_data) 70: summary =
self.survey_summary.generate_summary() 71: 72: print("\nRESPONSE OVERVIEW:") 73:
print(f" Total Responses: {summary['total_responses']}") 74: print(f" Response
Rate: {summary['response_rate']:.1f}%") 75: 76: print("\nDEMOGRAPHIC
BREAKDOWN:") 77: for demo, breakdown in summary['demographics'].items(): 78:
print(f"\n {demo.upper()}:") 79: for category, count in breakdown.items(): 80:
percentage = (count / summary['total_responses']) * 100 81: print(f" {category}:
{count} ({percentage:.1f}%)") 82: 83: except Exception as e: 84: print(f"ERROR:
Error generating summary: {str(e)}") 85: 86: def analyze_sentiment(self): 87:
"""Analyze sentiment of text responses.""" 88: if not self.survey_data: 89:
print("ERROR: No survey data loaded. Please load data first.") 90: return 91: 92:
try: 93: print_header("Sentiment Analysis") 94: self.sentiment_analyzer =
SentimentAnalyzer() 95: 96: # Find text columns 97: text_columns = [] 98: for col
in self.survey_data[0].keys(): 99: if any(isinstance(row[col], str) and
len(str(row[col])) > 20 100: 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_results['positive_pct']:.1f}%)")
print(f" Negative responses: {sentiment_results['negative']} ({sentiment_results['negative_pct']:.1f}%)")
print(f" Neutral responses: {sentiment_results['neutral']} ({sentiment_results['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
```

```
151: for row in crosstab: 152: print(" | ".join(f"{cell:>8}" for cell in row))
153: 154: print(f"\nCHI-SQUARE TEST RESULTS:") 155: print(f" Chi-Square Value:
{chi_square['chi_square']:.4f}") 156: print(f" P-Value:
{chi_square['p_value']:.4f}") 157: print(f" Degrees of Freedom:
{chi_square['df']}") 158: print(f" Significant: {'Yes' if
chi_square['significant'] else 'No'}") 159: 160: except Exception as e: 161:
print(f"ERROR: Error in cross-tabulation: {str(e)}") 162: 163: def
detect_patterns(self): 164: """Detect patterns and correlations in survey
responses.""" 165: if not self.survey_data: 166: print("ERROR: No survey data
loaded. Please load data first.") 167: return 168: 169: try: 170:
print_header("Pattern Detection") 171: self.pattern_detector =
PatternDetector(self.survey_data) 172: 173: patterns =
self.pattern_detector.find_patterns() 174: 175: print("DETECTED PATTERNS:") 176:
print("=" * 50) 177: 178: for pattern in patterns: 179: print(f"\nPATTERN:
{pattern['description']}") 180: print(f" Confidence:
{pattern['confidence']:.1f}%") 181: print(f" Sample size:
{pattern['sample_size']}") 182: 183: except Exception as e: 184: print(f"ERROR:
Error detecting patterns: {str(e)}") 185: 186: def generate_report(self): 187:
"""Generate a comprehensive analysis report.""" 188: if not self.survey_data:
189: print("ERROR: No survey data loaded. Please load data first.") 190: return
191: 192: try: 193: print_header("Generate Report") 194: 195: # Initialize all
analyzers 196: self.survey_summary = SurveySummary(self.survey_data) 197:
self.stats_analyzer = StatsAnalyzer(self.survey_data) 198:
self.sentiment_analyzer = SentimentAnalyzer() 199: self.pattern_detector =
PatternDetector(self.survey_data) 200: 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")

print_menu([
"Load survey data",
"View summary statistics",
"Analyze sentiment",
"Cross-tabulate results",
"Detect patterns",
"Generate report",
"Exit"
])

choice = input("\nEnter your choice (1-7): ").strip()

if choice == '1':
```

```
self.load_survey_data()
elif choice == '2':
self.view_summary_statistics()
elif choice == '3':
self.analyze_sentiment()
elif choice == '4':
self.cross_tabulate_results()
elif choice == '5':
self.detect_patterns()
elif choice == '6':
self.generate_report()
elif choice == '7':
print("\nThank you for using Survey Data Analyzer!")
sys.exit(0)
else:
print("ERROR: Invalid choice. Please try again.")

input("\nPress Enter to continue...")


def main():
"""Main entry point for the application."""
try:
app = SurveyAnalyzerCLI()
app.run()
except KeyboardInterrupt:
print("\n\nGoodbye!")
sys.exit(0)
except Exception as e:
print(f"ERROR: Unexpected error: {str(e)}")
sys.exit(1)


if __name__ == "__main__":
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.

Data Loading Module - 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:
```

```
51: print("ERROR: Could not read file with any supported encoding") 52: return
None 53: 54: # Validate and clean data 55: cleaned_data =
self._clean_and_validate_data(data) 56: if not cleaned_data: 57: return None 58:
59: print(f"SUCCESS: Data validation completed successfully") 60: return
cleaned_data 61: 62: except Exception as e: 63: print(f"ERROR: Error loading CSV
file: {str(e)}") 64: return None 65: 66: def _validate_file(self, file_path: str)
-> bool: 67: """Validate that the file exists and is accessible.""" 68: try: 69:
if not os.path.exists(file_path): 70: print(f"ERROR: File not found:
{file_path}") 71: return False 72: 73: if not os.path.isfile(file_path): 74:
print(f"ERROR: Path is not a file: {file_path}") 75: return False 76: 77: # Check
file size 78: file_size = os.path.getsize(file_path) 79: if file_size >
self.max_file_size: 80: print(f"ERROR: File too large: {file_size /
(1024*1024):.1f}MB (max: 50MB)") 81: return False 82: 83: if file_size == 0: 84:
print("ERROR: File is empty") 85: return False 86: 87: return True 88: 89: except
Exception as e: 90: print(f"ERROR: Error validating file: {str(e)}") 91: return
False 92: 93: def _read_csv_with_encoding(self, file_path: str, encoding: str) ->
Optional[List[Dict[str, Any]]]: 94: """Read CSV file with specified encoding."""
95: try: 96: with open(file_path, 'r', encoding=encoding, newline='') as file:
97: # Try to detect delimiter 98: sample = file.read(1024) 99: file.seek(0) 100:
```

```
101: # Common delimiters to try 102: delimiters = [',', ';', '\t', '|'] 103:
detected_delimiter = ',' 104: 105: for delimiter in delimiters: 106: if delimiter
in sample: 107: detected_delimiter = delimiter 108: break 109: 110: reader =
csv.DictReader(file, delimiter=detected_delimiter) 111: data = list(reader) 112:
113: if not data: 114: print("ERROR: No data found in CSV file") 115: return None
116: 117: print(f"SUCCESS: Loaded {len(data)} rows with {len(data[0])} columns")
118: print(f"INFO: Columns: {', '.join(data[0].keys())}") 119: 120: return data
121: 122: except Exception as e: 123: print(f"ERROR: Error reading CSV with
{encoding} encoding: {str(e)}") 124: return None 125: 126: def
_clean_and_validate_data(self, data: List[Dict[str, Any]]) ->
Optional[List[Dict[str, Any]]]: 127: """Clean and validate the loaded data."""
128: try: 129: if not data: 130: return None 131: 132: cleaned_data = [] 133:
validation_errors = [] 134: 135: for i, row in enumerate(data, 1): 136:
cleaned_row = {} 137: row_errors = [] 138: 139: for key, value in row.items():
140: # Clean the key (remove whitespace, normalize) 141: if key is None: 142:
continue # Skip rows with None keys 143: clean_key =
str(key).strip().lower().replace(' ', '_') 144: 145: # Clean the value 146: if
value is None or value == '': 147: cleaned_value = None 148: else: 149:
cleaned_value = str(value).strip() 150: if cleaned_value.lower() in ['na', 'n/a',
'null', 'none']:
```

```
151: cleaned_value = None 152: 153: cleaned_row[clean_key] = cleaned_value 154:
155: # Basic validation 156: if self._validate_row(cleaned_row, i): 157:
cleaned_data.append(cleaned_row) 158: else: 159: validation_errors.append(f"Row
{i}: Invalid data") 160: 161: # Report validation results 162: total_rows =
len(data) 163: valid_rows = len(cleaned_data) 164: invalid_rows = total_rows -
valid_rows 165: 166: print(f"INFO: Data validation results:") 167: print(f" Total
rows: {total_rows}") 168: print(f" Valid rows: {valid_rows}") 169: print(f"
Invalid rows: {invalid_rows}") 170: 171: if invalid_rows > 0: 172:
print(f"WARNING: {invalid_rows} rows had validation issues") 173: 174: if
valid_rows == 0: 175: print("ERROR: No valid data found") 176: return None 177:
178: return cleaned_data 179: 180: except Exception as e: 181: print(f"ERROR:
Error cleaning data: {str(e)}") 182: return None 183: 184: def
_validate_row(self, row: Dict[str, Any], row_num: int) -> bool: 185: """Validate
a single row of data.""" 186: try: 187: # Check for minimum required fields 188:
if len(row) < 2: 189: return False 190: 191: # Validate age if present 192: if
'age' in row and row['age'] is not None: 193: try: 194: age = int(row['age'])
195: if age < 0 or age > 120: 196: return False 197: except (ValueError,
TypeError): 198: return False 199: 200: # 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.

Survey Summary Module - 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
```

```
51: # For now, we'll use a placeholder calculation 52: return 85.5 # Placeholder
response rate 53: 54: def _analyze_demographics(self) -> Dict[str, Dict[str,
int]]: 55: """Analyze demographic breakdowns.""" 56: demographics = {} 57: 58: #
Common demographic fields 59: demo_fields = ['age', 'gender', 'region',
'education', 'income'] 60: 61: for field in demo_fields: 62: if field in
self.columns: 63: demographics[field] = self._count_responses_by_field(field)
64: 65: return demographics 66: 67: def _analyze_questions(self) -> Dict[str,
Dict[str, Any]]: 68: """Analyze responses for each question.""" 69:
question_summaries = {} 70: 71: for column in self.columns: 72: if column not in
['age', 'gender', 'region', 'education', 'income']: 73:
question_summaries[column] = self._analyze_question_responses(column) 74: 75:
return question_summaries 76: 77: def _count_responses_by_field(self, field:
str) -> Dict[str, int]: 78: """Count responses for a specific field.""" 79:
counts = Counter() 80: 81: for row in self.survey_data: 82: value =
row.get(field) 83: if value is not None and str(value).strip(): 84: # Normalize
the value 85: normalized_value = str(value).strip().title() 86:
counts[normalized_value] += 1 87: 88: return dict(counts) 89: 90: def
_analyze_question_responses(self, question: str) -> Dict[str, Any]: 91:
"""Analyze responses for a specific question.""" 92: responses =
[row.get(question) for row in self.survey_data] 93: valid_responses = [r for r in
responses if r is not None and str(r).strip()] 94: 95: if not valid_responses:
96: return { 97: 'total_responses': 0, 98: 'missing_responses': len(responses),
99: 'response_rate': 0.0, 100: '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)

return {
'total_responses': total_valid,
'missing_responses': len(responses) - total_valid,
'response_rate': (total_valid / len(responses)) * 100,
'top_responses': top_responses,
'response_distribution': response_distribution
}

def _assess_data_quality(self) -> Dict[str, Any]:
"""Assess the quality of the survey data."""
quality_metrics = {
'total_rows': len(self.survey_data),
'total_columns': len(self.columns),
'missing_data': {},
'completeness': {}
}

# Analyze missing data for each column
for column in self.columns:
values = [row.get(column) for row in self.survey_data]
missing_count = sum(1 for v in values if v is None or str(v).strip() == '')
missing_percentage = (missing_count / len(values)) * 100

quality_metrics['missing_data'][column] = {
'count': missing_count,
'percentage': missing_percentage
}

quality_metrics['completeness'][column] = 100 - missing_percentage

return quality_metrics
```

```
def get_age_distribution(self) -> Dict[str, int]: """Get age
distribution if age data is available.""" if 'age' not in self.columns:
return {} age_groups = { '18-25': 0, '26-35': 0,
'36-45': 0, '46-55': 0, '56-65': 0, '65+': 0 } 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'] += 1 elif 26 <= age <= 35:
age_groups['26-35'] += 1 elif 36 <= age <= 45: age_groups['36-45'] += 1
elif 46 <= age <= 55: age_groups['46-55'] += 1 elif 56 <= age <=
65: age_groups['56-65'] += 1 elif age > 65: age_groups['65+'] += 1
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.

Statistical Analysis Module - 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 tests."""

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
}

```

```
151: def _calculate_expected_frequencies(self, observed: List[List[int]]) ->
List[List[float]]: 152: """Calculate expected frequencies for chi-square
test.""" 153: if not observed or not observed[0]: 154: return [] 155: 156: rows =
len(observed) 157: cols = len(observed[0]) 158: 159: # Calculate row and column
totals 160: row_totals = [sum(row) for row in observed] 161: col_totals = [] 162:
for j in range(cols): 163: col_totals.append(sum(observed[i][j] for i in
range(rows))) 164: 165: total = sum(row_totals) 166: 167: # Calculate expected
frequencies 168: expected = [] 169: for i in range(rows): 170: row = [] 171: for
j in range(cols): 172: expected_freq = (row_totals[i] * col_totals[j]) / total if
total > 0 else 0 173: row.append(expected_freq) 174: expected.append(row) 175:
176: return expected 177: 178: def _chi_square_p_value(self, chi_square: float,
df: int) -> float: 179: """ 180: Calculate approximate p-value for chi-square
statistic. 181: This is a simplified approximation - in practice, you'd use a
proper chi-square distribution. 182: """ 183: if df <= 0: 184: return 1.0 185:
186: # Simple approximation for chi-square p-value 187: # For small chi-square
values, p-value is close to 1 188: # For large chi-square values, p-value
approaches 0 189: if chi_square < df: 190: return 1.0 - (chi_square / (df * 2))
191: else: 192: return max(0.0, 1.0 - (chi_square / (df * 10))) 193: 194: def
correlation_analysis(self, col1: str, col2: str) -> Dict[str, Any]: 195: """ 196:
Perform correlation analysis between two variables. 197: 198: Args: 199: col1:
First column name 200: 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

n = len(x)

# Calculate means
mean_x = sum(x) / n
mean_y = sum(y) / n

```

```
# Calculate correlation coefficient
numerator = sum((x[i] - mean_x) * (y[i] - mean_y) for i in range(n))
denominator_x = sum((x[i] - mean_x) ** 2 for i in range(n))
denominator_y = sum((y[i] - mean_y) ** 2 for i in range(n))

if denominator_x == 0 or denominator_y == 0:
return 0.0

correlation = numerator / math.sqrt(denominator_x * denominator_y)
return correlation

def _interpret_correlation(self, correlation: float) -> str:
"""Interpret correlation coefficient strength."""
abs_corr = abs(correlation)

if abs_corr >= 0.8:
return "Very Strong"
elif abs_corr >= 0.6:
return "Strong"
elif abs_corr >= 0.4:
return "Moderate"
elif abs_corr >= 0.2:
return "Weak"
else:
return "Very Weak"

def analyze_response_patterns(self) -> Dict[str, Any]:
"""Analyze patterns in survey responses."""
patterns = {
'common_combinations': [],
'response_clusters': [],
'outliers': []
}

# Find common combinations of responses
response_combinations = []
for row in self.survey_data:
combination = []
for col in self.columns:
value = row.get(col)
if value is not None and str(value).strip():
combination.append(f"{col}:{str(value).strip()}")
if combination:
response_combinations.append(tuple(sorted(combination)))

# Count combinations
combination_counts = Counter(response_combinations)
common_combinations = combination_counts.most_common(5)

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: 309: return patterns 310: 311: def
get_statistical_summary(self) -> Dict[str, Any]: 312: """Generate a
comprehensive statistical summary.""" 313: summary = { 314: 'total_responses':
self.total_responses, 315: 'total_columns': len(self.columns), 316:
'numeric_columns': [], 317: 'categorical_columns': [], 318: 'statistical_tests':
[] 319: } 320: 321: # Categorize columns 322: for column in self.columns: 323:
numeric_count = 0 324: total_count = 0 325: 326: for row in self.survey_data:
327: value = row.get(column) 328: if value is not None and str(value).strip():
329: total_count += 1 330: try: 331: float(str(value)) 332: numeric_count += 1
333: except (ValueError, TypeError): 334: pass 335: 336: if total_count > 0 and
(numeric_count / total_count) > 0.5: 337:
summary['numeric_columns'].append(column) 338: else: 339:
summary['categorical_columns'].append(column) 340: 341: return summary 342: 343:
def perform_multiple_chi_square_tests(self, target_column: str) ->
List[Dict[str, Any]]: 344: """ 345: Perform chi-square tests between a target
column and all other categorical columns. 346: 347: Args: 348: target_column: The
target column to test against 349: 350: 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.

Sentiment Analysis Module - 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()
```

```
101: 102: # Check for intensifiers 103: if word_lower in self.intensifier_words:
104: intensifier_count += 1 105: continue 106: 107: # Check for negations 108:
is_negated = self._is_negated(words, i) 109: 110: # Check positive keywords 111:
if word_lower in self.positive_keywords: 112: weight =
self.positive_keywords[word_lower] 113: if is_negated: 114: sentiment_score -=
weight 115: negative_words.append(word) 116: else: 117: sentiment_score += weight
118: 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)) 182:
sentiment_results.append(result) 183: total_responses += 1 184: 185: if not
sentiment_results: 186: return { 187: 'positive': 0, 188: 'negative': 0, 189:
'neutral': 0, 190: 'positive_pct': 0.0, 191: 'negative_pct': 0.0, 192:
'neutral_pct': 0.0, 193: 'avg_score': 0.0, 194: 'total_responses': 0 195: } 196:
197: # Count sentiments 198: positive_count = sum(1 for r in sentiment_results if
r['sentiment'] == 'positive') 199: negative_count = sum(1 for r in
sentiment_results if r['sentiment'] == 'negative') 200: neutral_count = sum(1 for
r in sentiment_results if r['sentiment'] == 'neutral')
```

```
201: 202: # Calculate percentages 203: total = len(sentiment_results) 204:
positive_pct = (positive_count / total) * 100 205: negative_pct = (negative_count
/ total) * 100 206: neutral_pct = (neutral_count / total) * 100 207: 208: #
Calculate average score 209: avg_score = sum(r['score'] for r in
sentiment_results) / total 210: 211: return { 212: 'positive': positive_count,
213: 'negative': negative_count, 214: 'neutral': neutral_count, 215:
'positive_pct': positive_pct, 216: 'negative_pct': negative_pct, 217:
'neutral_pct': neutral_pct, 218: 'avg_score': avg_score, 219: 'total_responses':
total_responses, 220: 'detailed_results': sentiment_results 221: } 222: 223: def
analyze_all_text_columns(self, survey_data: List[Dict[str, Any]]) -> Dict[str,
Any]: 224: """ 225: Analyze sentiment for all text columns in the survey data.
226: 227: Args: 228: survey_data: List of survey responses 229: 230: Returns:
231: Dictionary containing sentiment analysis for all text columns 232: """ 233:
if not survey_data: 234: return {} 235: 236: text_columns = [] 237: for col in
survey_data[0].keys(): 238: # Check if column contains text data 239: text_count
= 0 240: total_count = 0 241: 242: for row in survey_data[:10]: # Sample first 10
rows 243: value = row.get(col) 244: if value is not None and str(value).strip():
245: total_count += 1 246: if len(str(value)) > 20: # Consider it text if longer
than 20 chars 247: text_count += 1 248: 249: if total_count > 0 and (text_count /
total_count) > 0.3: 250: text_columns.append(col)
```

```
251: 252: results = {} 253: for column in text_columns: 254: results[column] =
self.analyze_column(survey_data, column) 255: 256: return results 257: 258: def
_clean_text(self, text: str) -> str: 259: """Clean and normalize text for
analysis.""" 260: # Convert to lowercase 261: text = text.lower() 262: 263: #
Remove extra whitespace 264: text = re.sub(r'\s+', ' ', text) 265: 266: # Remove
punctuation (keep apostrophes for contractions) 267: text = re.sub(r'[^\w\s\']',
' ', text) 268: 269: return text.strip() 270: 271: def _extract_words(self, text:
str) -> List[str]: 272: """Extract words from text.""" 273: return text.split()
274: 275: def _is_negated(self, words: List[str], current_index: int) -> bool:
276: """Check if current word is negated by previous words.""" 277: # Look back
up to 3 words for negation 278: start_index = max(0, current_index - 3) 279: 280:
for i in range(start_index, current_index): 281: if i < len(words) and
words[i].lower() in self.negation_words: 282: return True 283: 284: return False
285: 286: def _categorize_sentiment(self, score: float) -> str: 287:
"""Categorize sentiment based on score.""" 288: if score > 1.0: 289: return
'positive' 290: elif score < -1.0: 291: return 'negative' 292: else: 293: return
'neutral' 294: 295: def get_sentiment_summary(self, sentiment_results:
List[Dict[str, Any]]) -> Dict[str, Any]: 296: """Generate a summary of sentiment
analysis results.""" 297: if not sentiment_results: 298: return {} 299: 300:
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.

Pattern Detection Module - 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
```

```
51: patterns.extend(self._find_outlier_patterns()) 52: 53: # Sort patterns by
confidence 54: patterns.sort(key=lambda x: x['confidence'], reverse=True) 55:
56: return patterns 57: 58: def _find_demographic_patterns(self) ->
List[Dict[str, Any]]: 59: """Find patterns related to demographics.""" 60:
patterns = [] 61: 62: # Age-based patterns 63: if 'age' in self.columns: 64:
age_patterns = self._analyze_age_patterns() 65: patterns.extend(age_patterns)
66: 67: # Gender-based patterns 68: if 'gender' in self.columns: 69:
gender_patterns = self._analyze_gender_patterns() 70:
patterns.extend(gender_patterns) 71: 72: # Regional patterns 73: if 'region' in
self.columns: 74: regional_patterns = self._analyze_regional_patterns() 75:
patterns.extend(regional_patterns) 76: 77: # Education-based patterns 78: if
'education' in self.columns: 79: education_patterns =
self._analyze_education_patterns() 80: patterns.extend(education_patterns) 81:
82: return patterns 83: 84: def _analyze_age_patterns(self) -> List[Dict[str,
Any]]: 85: """Analyze patterns based on age groups.""" 86: patterns = [] 87: 88:
# Group responses by age 89: age_groups = { 90: '18-25': [], 91: '26-35': [], 92:
'36-45': [], 93: '46-55': [], 94: '56-65': [], 95: '65+': [] 96: } 97: 98: for
row in self.survey_data: 99: age_str = row.get('age') 100: 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)

```

```
151: for response, age_groups_list in response_groups.items(): 152: if
len(age_groups_list) > 1: 153: confidence = min(90, len(age_groups_list) * 20)
154: patterns.append({ 155: 'type': 'age_pattern', 156: 'description': f"Age
groups {', '.join(age_groups_list)} most commonly responded '{response}' to
{column}", 157: 'confidence': confidence, 158: 'sample_size':
sum(len(age_groups[ag]) for ag in age_groups_list), 159: 'response': response,
160: 'affected_groups': age_groups_list 161: }) 162: 163: return patterns 164:
165: def _analyze_gender_patterns(self) -> List[Dict[str, Any]]: 166: """Analyze
patterns based on gender.""" 167: patterns = [] 168: 169: # Group responses by
gender 170: gender_groups = defaultdict(list) 171: for row in self.survey_data:
172: gender = row.get('gender') 173: if gender and str(gender).strip(): 174:
gender_groups[str(gender).strip().title()].append(row) 175: 176: # Analyze
patterns for each question 177: for column in self.columns: 178: if column not in
['age', 'gender', 'region', 'education']: 179: column_patterns =
self._analyze_column_by_gender(column, gender_groups) 180:
patterns.extend(column_patterns) 181: 182: return patterns 183: 184: def
_analyze_column_by_gender(self, column: str, gender_groups: Dict[str,
List[Dict[str, Any]]]) -> List[Dict[str, Any]]: 185: """Analyze a specific
column's responses by gender.""" 186: patterns = [] 187: 188: # Get most common
response for each gender 189: gender_responses = {} 190: for gender, responses in
gender_groups.items(): 191: if responses: 192: values = [r.get(column) for r in
responses if r.get(column)] 193: if values: 194: value_counts = Counter(values)
195: most_common = value_counts.most_common(1)[0] 196: gender_responses[gender]
= { 197: 'response': most_common[0], 198: 'count': most_common[1], 199:
'percentage': (most_common[1] / len(values)) * 100 200: }
```

```
201: 202: # Find gender differences 203: if len(gender_responses) > 1: 204:
responses = list(gender_responses.values()) 205: if len(set(r['response'] for r
in responses)) > 1: 206: # Different genders have different most common responses
207: confidence = 75 208: patterns.append({ 209: 'type': 'gender_pattern', 210:
'description': f"Gender differences detected in {column} responses", 211:
'confidence': confidence, 212: 'sample_size': sum(len(gender_groups[g]) for g in
gender_responses.keys()), 213: 'details': gender_responses 214: }) 215: 216:
return patterns 217: 218: def _analyze_regional_patterns(self) -> List[Dict[str,
Any]]: 219: """Analyze patterns based on region.""" 220: patterns = [] 221: 222:
# Group responses by region 223: regional_groups = defaultdict(list) 224: for row
in self.survey_data: 225: region = row.get('region') 226: if region and
str(region).strip(): 227:
regional_groups[str(region).strip().title()].append(row) 228: 229: # Analyze
patterns for each question 230: for column in self.columns: 231: if column not in
['age', 'gender', 'region', 'education']: 232: column_patterns =
self._analyze_column_by_region(column, regional_groups) 233:
patterns.extend(column_patterns) 234: 235: return patterns 236: 237: def
_analyze_column_by_region(self, column: str, regional_groups: Dict[str,
List[Dict[str, Any]]]) -> List[Dict[str, Any]]: 238: """Analyze a specific
column's responses by region.""" 239: patterns = [] 240: 241: # Get most common
response for each region 242: regional_responses = {} 243: for region, responses
in regional_groups.items(): 244: if responses: 245: values = [r.get(column) for r
in responses if r.get(column)] 246: if values: 247: value_counts =
Counter(values) 248: most_common = value_counts.most_common(1)[0] 249:
regional_responses[region] = { 250: 'response': most_common[0],
251: 'count': most_common[1], 252: 'percentage': (most_common[1] / len(values)) *
100 253: } 254: 255: # Find regional patterns 256: if len(regional_responses) >
1: 257: # Find regions with similar responses 258: response_groups =
defaultdict(list) 259: for region, data in regional_responses.items(): 260:
response_groups[data['response']].append(region) 261: 262: for response,
regions_list in response_groups.items(): 263: if len(regions_list) > 1: 264:
confidence = min(85, len(regions_list) * 25) 265: patterns.append({ 266: 'type':
'regional_pattern', 267: 'description': f"Regions {', '.join(regions_list)} most
commonly responded '{response}' to {column}", 268: 'confidence': confidence, 269:
'sample_size': sum(len(regional_groups[r]) for r in regions_list), 270:
'response': response, 271: 'affected_regions': regions_list 272: }) 273: 274:
return patterns 275: 276: def _analyze_education_patterns(self) ->
List[Dict[str, Any]]: 277: """Analyze patterns based on education level.""" 278:
patterns = [] 279: 280: # Group responses by education 281: education_groups =
defaultdict(list) 282: for row in self.survey_data: 283: education =
row.get('education') 284: if education and str(education).strip(): 285:
education_groups[str(education).strip().title()].append(row) 286: 287: # Analyze
patterns for each question 288: for column in self.columns: 289: if column not in
['age', 'gender', 'region', 'education']: 290: column_patterns =
self._analyze_column_by_education(column, education_groups) 291:
patterns.extend(column_patterns) 292: 293: return patterns 294: 295: def
_analyze_column_by_education(self, column: str, education_groups: Dict[str,
List[Dict[str, Any]]]) -> List[Dict[str, Any]]: 296: """Analyze a specific
column's responses by education level.""" 297: patterns = [] 298: 299: # Get most
common response for each education level 300: 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 commonly 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', 'region', '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."""
```

```
351: # Get unique values for both columns 352: values1 = set() 353: values2 =
set() 354: 355: for row in self.survey_data: 356: val1 = row.get(col1) 357: val2
= row.get(col2) 358: 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', 'hate'] 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({
```

```
'type': 'combination_pattern',
'description': f"Common response combination: {', '.join(combo)}",
'confidence': min(85, percentage * 2),
'sample_size': count,
'percentage': percentage,
'combination': list(combo)
})

return patterns

def _find_outlier_patterns(self) -> List[Dict[str, Any]]:
"""Find outlier patterns in responses."""
patterns = []

# Find responses that are significantly different from the norm
for column in self.columns:
if column not in ['age', 'gender', 'region', 'education']:
outlier_pattern = self._analyze_outliers(column)
if outlier_pattern:
patterns.append(outlier_pattern)

return patterns

def _analyze_outliers(self, column: str) -> Optional[Dict[str, Any]]:
"""Analyze outliers in a specific column."""
values = [row.get(column) for row in self.survey_data if row.get(column)]

if not values:
return None

# Count responses
value_counts = Counter(values)
total_responses = len(values)

# 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.

Report Generation Module - 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], patterns: 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
```

```
151: if data_quality: 152: completeness = data_quality.get('completeness', {})
153: if completeness: 154: lines.append("■ Data Completeness by Column:") 155:
for column, completeness_pct in completeness.items(): 156: lines.append(f" •
{column}: {completeness_pct:.1f}%") 157: lines.append("") 158: 159: return lines
160: 161: def _generate_demographic_analysis(self, summary: Dict[str, Any]) ->
List[str]: 162: """Generate the demographic analysis section.""" 163: lines = [
164: "DEMOGRAPHIC ANALYSIS", 165: "-" * 50, 166: "" 167: ] 168: 169: demographics
= summary.get('demographics', {}) 170: 171: if demographics: 172:
lines.append("■ Respondent Demographics:") 173: lines.append("") 174: 175: for
demo_field, breakdown in demographics.items(): 176: if breakdown: 177:
lines.append(f"■ {demo_field.title()}:") 178: total_demo =
sum(breakdown.values()) 179: 180: for category, count in breakdown.items(): 181:
percentage = (count / total_demo) * 100 182: lines.append(f" • {category}:
{count} ({percentage:.1f}%)") 183: 184: lines.append("") 185: else: 186:
lines.append("No demographic data available for analysis.") 187:
lines.append("") 188: 189: return lines 190: 191: def
_generate_sentiment_analysis(self, sentiment: Dict[str, Any]) -> List[str]: 192:
"""Generate the sentiment analysis section.""" 193: lines = [ 194: "SENTIMENT
ANALYSIS", 195: "-" * 50, 196: "" 197: ] 198: 199: if sentiment: 200:
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

```

```
251: def _generate_statistical_analysis(self, summary: Dict[str, Any]) ->
List[str]: 252: """Generate the statistical analysis section.""" 253: lines = [
254: "STATISTICAL ANALYSIS", 255: "-" * 50, 256: "" 257: ] 258: 259:
question_summaries = summary.get('question_summaries', {}) 260: 261: if
question_summaries: 262: lines.append("■ Response Distribution Analysis:") 263:
lines.append("") 264: 265: for question, q_summary in
question_summaries.items(): 266: if isinstance(q_summary, dict): 267:
lines.append(f"■ {question}:") 268: lines.append(f" • Total Responses:
{q_summary.get('total_responses', 0)}") 269: lines.append(f" • Response Rate:
{q_summary.get('response_rate', 0):.1f}%") 270: 271: top_responses =
q_summary.get('top_responses', []) 272: if top_responses: 273: lines.append(" •
Top Responses:") 274: for response, count in top_responses[:3]: 275: percentage =
(count / q_summary['total_responses']) * 100 276: lines.append(f" - {response}:
{count} ({percentage:.1f}%)") 277: 278: lines.append("") 279: else: 280:
lines.append("No question response data available for statistical analysis.")
281: lines.append("") 282: 283: return lines 284: 285: def
_generate_key_findings(self, summary: Dict[str, Any], sentiment: Dict[str, Any],
patterns: List[Dict[str, Any]]) -> List[str]: 286: """Generate the key findings
section.""" 287: lines = [ 288: "KEY FINDINGS", 289: "-" * 50, 290: "" 291: ]
292: 293: findings = [] 294: 295: # Demographic findings 296: demographics =
summary.get('demographics', {}) 297: if demographics: 298: for demo_field,
breakdown in demographics.items(): 299: 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
common 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
detected 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], patterns: 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 completeness: {', '.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 sentiment: {', '.join(negative_sentiment_columns)}")
```

```
# 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-confidence 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"
```

```
401: elif avg_completeness >= 70: 402: return "Fair" 403: else: 404: return
"Poor" 405: 406: def _get_dominant_sentiment(self, sentiment_results: Dict[str,
Any]) -> str: 407: """Get the dominant sentiment from results.""" 408:
positive_pct = sentiment_results.get('positive_pct', 0) 409: negative_pct =
sentiment_results.get('negative_pct', 0) 410: neutral_pct =
sentiment_results.get('neutral_pct', 0) 411: 412: if positive_pct > negative_pct
and positive_pct > neutral_pct: 413: return "Positive" 414: elif negative_pct >
positive_pct and negative_pct > neutral_pct: 415: return "Negative" 416: else:
417: return "Neutral" 418: 419: def _write_report_to_file(self, content: str,
file_path: str) -> bool: 420: """Write the report content to a file.""" 421: try:
422: # Ensure directory exists 423: directory = os.path.dirname(file_path) 424:
if directory and not os.path.exists(directory): 425: os.makedirs(directory) 426:
427: # Write content to file 428: with open(file_path, 'w', encoding='utf-8') as
file: 429: file.write(content) 430: 431: return True 432: 433: except Exception
as e: 434: print(f"Error writing report to file: {str(e)}") 435: return False
436: 437: def generate_summary_report(self, summary_data: Dict[str, Any],
file_path: str = "summary_report.txt") -> bool: 438: """Generate a simplified
summary report.""" 439: try: 440: lines = [ 441: "SURVEY SUMMARY REPORT", 442:
"=" * 50, 443: f"Generated: {self.current_datetime.strftime('%B %d, %Y')}", 444:
"", 445: f"Total Responses: {summary_data.get('total_responses', 0)}", 446:
f"Response Rate: {summary_data.get('response_rate', 0):.1f}%", 447: "", 448:
"Key Metrics:", 449: "• Data quality assessment", 450: "• 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.

Utilities Module - 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:
```

```
201: True if user confirms, False otherwise 202: """ 203: response =
input(f"{prompt} (y/N): ").strip().lower() 204: return response in ['y', 'yes']
205: 206: 207: def display_progress(current: int, total: int, description: str =
"Processing"): 208: """ 209: Display a progress bar. 210: 211: Args: 212:
current: Current progress value 213: total: Total value 214: description:
Description of the operation 215: """ 216: if total == 0: 217: return 218: 219:
percentage = (current / total) * 100 220: bar_length = 30 221: filled_length =
int(bar_length * current // total) 222: bar = '■' * filled_length + '-' *
(bar_length - filled_length) 223: 224: print(f"\r{description}: |{bar}|
{percentage:.1f}% ({current}/{total})", end='') 225: 226: if current == total:
227: print() # New line when complete 228: 229: 230: def safe_divide(numerator:
float, denominator: float, default: float = 0.0) -> float: 231: """ 232: Safely
divide two numbers, returning default if denominator is zero. 233: 234: Args:
235: numerator: The numerator 236: denominator: The denominator 237: default:
Default value if division by zero 238: 239: Returns: 240: Result of division or
default value 241: """ 242: try: 243: return numerator / denominator if
denominator != 0 else default 244: except (TypeError, ValueError): 245: return
default 246: 247: 248: def truncate_text(text: str, max_length: int = 50) -> str:
249: """ 250: 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
```

```
351: """ 352: if not text: 353: return 0 354: 355: return len(text.split()) 356:
357: 358: def get_file_extension(file_path: str) -> str: 359: """ 360: Get file
extension from file path. 361: 362: Args: 363: file_path: Path to file 364: 365:
Returns: 366: File extension (including dot) 367: """ 368: return
os.path.splitext(file_path)[1].lower() 369: 370: 371: def
ensure_directory_exists(directory_path: str) -> bool: 372: """ 373: Ensure a
directory exists, creating it if necessary. 374: 375: Args: 376: directory_path:
Path to directory 377: 378: Returns: 379: True if directory exists or was
created, False otherwise 380: """ 381: try: 382: if not
os.path.exists(directory_path): 383: os.makedirs(directory_path) 384: return
True 385: except Exception: 386: return False 387: 388: 389: def
is_valid_filename(filename: str) -> bool: 390: """ 391: Check if a filename is
valid. 392: 393: Args: 394: filename: Filename to validate 395: 396: Returns:
397: True if valid filename, False otherwise 398: """ 399: if not filename: 400:
return False
```

```
401: 402: # Check for invalid characters 403: invalid_chars = '<>:"/\\|?*' 404:
return not any(char in filename for char in invalid_chars) 405: 406: 407: def
format_duration(seconds: float) -> str: 408: """ 409: Format duration in seconds
to human-readable format. 410: 411: Args: 412: seconds: Duration in seconds 413:
414: Returns: 415: Formatted duration string 416: """ 417: if seconds < 60: 418:
return f"{seconds:.1f} seconds" 419: elif seconds < 3600: 420: minutes = seconds
/ 60 421: return f"{minutes:.1f} minutes" 422: else: 423: hours = seconds / 3600
424: return f"{hours:.1f} hours" 425: 426: 427: def print_separator(char: str =
"-", length: int = 60): 428: """Print a separator line.""" 429: print(char *
length) 430: 431: 432: def print_bullet_list(items: List[str], indent: int = 2):
433: """ 434: Print a bullet list. 435: 436: Args: 437: items: List of items to
print 438: indent: Number of spaces to indent 439: """ 440: indent_str = " " *
indent 441: for item in items: 442: print(f"{indent_str}• {item}") 443: 444: 445:
def print_key_value_pairs(data: Dict[str, Any], indent: int = 2): 446: """ 447:
Print key-value pairs in a formatted way. 448: 449: Args: 450: data: Dictionary
of key-value pairs
```

```
451: indent: Number of spaces to indent 452: """ 453: indent_str = " " * indent
454: for key, value in data.items(): 455: 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.

Data Loader Tests - 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")
```

```
51: with open(temp_file, 'w') as f: 52: pass # Create empty file 53: 54: result =
self.data_loader._validate_file(temp_file) 55: self.assertFalse(result) 56: 57:
def test_validate_file_too_large(self): 58: """Test file validation with file too
large.""" 59: temp_file = os.path.join(self.temp_dir, "large.csv") 60: # Create a
file larger than 50MB 61: with open(temp_file, 'w') as f: 62: f.write("x" * (51 *
1024 * 1024)) # 51MB 63: 64: result = self.data_loader._validate_file(temp_file)
65: self.assertFalse(result) 66: 67: def test_read_csv_with_encoding_utf8(self):
68: """Test CSV reading with UTF-8 encoding.""" 69: temp_file =
os.path.join(self.temp_dir, "test_utf8.csv") 70: 71: # Create test CSV data 72:
test_data = [ 73: {'name': 'John', 'age': '25', 'city': 'New York'}, 74: {'name':
'Jane', 'age': '30', 'city': 'Los Angeles'}, 75: {'name': 'Bob', 'age': '35',
'city': 'Chicago'} 76: ] 77: 78: with open(temp_file, 'w', newline='',
encoding='utf-8') as f: 79: writer = csv.DictWriter(f, fieldnames=['name', 'age',
'city']) 80: writer.writeheader() 81: writer.writerows(test_data) 82: 83: result
= self.data_loader._read_csv_with_encoding(temp_file, 'utf-8') 84:
self.assertIsNotNone(result) 85: self.assertEqual(len(result), 3) 86:
self.assertEqual(result[0]['name'], 'John') 87: 88: def
test_read_csv_with_encoding_latin1(self): 89: """Test CSV reading with Latin-1
encoding.""" 90: temp_file = os.path.join(self.temp_dir, "test_latin1.csv") 91:
92: # Create test CSV data 93: test_data = [ 94: {'name': 'José', 'age': '25',
'city': 'Madrid'}, 95: {'name': 'François', 'age': '30', 'city': 'Paris'} 96: ]
97: 98: with open(temp_file, 'w', newline='', encoding='latin-1') as f: 99:
writer = csv.DictWriter(f, fieldnames=['name', 'age', 'city']) 100:
writer.writeheader()
```

```
101: writer.writerows(test_data) 102: 103: result =
self.data_loader._read_csv_with_encoding(temp_file, 'latin-1') 104:
self.assertIsNotNone(result) 105: self.assertEqual(len(result), 2) 106: 107: def
test_clean_and_validate_data_valid(self): 108: """Test data cleaning and
validation with valid data.""" 109: test_data = [ 110: {'age': '25', 'gender':
'Male', 'region': 'North'}, 111: {'age': '30', 'gender': 'Female', 'region':
'South'}, 112: {'age': '35', 'gender': 'Male', 'region': 'East'} 113: ] 114: 115:
result = self.data_loader._clean_and_validate_data(test_data) 116:
self.assertIsNotNone(result) 117: self.assertEqual(len(result), 3) 118: 119: def
test_clean_and_validate_data_invalid_age(self): 120: """Test data cleaning with
invalid age data.""" 121: test_data = [ 122: {'age': '25', 'gender': 'Male',
'region': 'North'}, 123: {'age': '150', 'gender': 'Female', 'region': 'South'}, #
Invalid age 124: {'age': '35', 'gender': 'Male', 'region': 'East'} 125: ] 126:
127: result = self.data_loader._clean_and_validate_data(test_data) 128:
self.assertIsNotNone(result) 129: # Should filter out invalid age 130:
self.assertEqual(len(result), 2) 131: 132: def
test_clean_and_validate_data_invalid_gender(self): 133: """Test data cleaning
with invalid gender data.""" 134: test_data = [ 135: {'age': '25', 'gender':
'Male', 'region': 'North'}, 136: {'age': '30', 'gender': 'Invalid', 'region':
'South'}, # Invalid gender 137: {'age': '35', 'gender': 'Female', 'region':
'East'} 138: ] 139: 140: result =
self.data_loader._clean_and_validate_data(test_data) 141:
self.assertIsNotNone(result) 142: # Should filter out invalid gender 143:
self.assertEqual(len(result), 2) 144: 145: def
test_clean_and_validate_data_empty(self): 146: """Test data cleaning with empty
data.""" 147: result = self.data_loader._clean_and_validate_data([]) 148:
self.assertIsNone(result) 149: 150: 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()
```

Sentiment Analyzer Tests - 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
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from sentiment_analyzer import SentimentAnalyzer


class TestSentimentAnalyzer(unittest.TestCase):
"""Test cases for the SentimentAnalyzer class."""

def setUp(self):
"""Set up test fixtures."""
self.sentiment_analyzer = SentimentAnalyzer()
self.test_data = [
{'feedback': 'This is excellent! I love it.', 'satisfaction': 'Very Satisfied'},
{'feedback': 'Terrible experience, very bad service.', 'satisfaction': 'Dissatisfied'},
{'feedback': 'It was okay, nothing special.', 'satisfaction': 'Neutral'},
{'feedback': 'Great product and amazing support!', 'satisfaction': 'Very Satisfied'},
{'feedback': 'Poor quality and bad service.', 'satisfaction': 'Dissatisfied'}
]

def test_initialization(self):
"""Test SentimentAnalyzer initialization."""
self.assertIsInstance(self.sentiment_analyzer.positive_keywords, dict)
self.assertIsInstance(self.sentiment_analyzer.negative_keywords, dict)
self.assertIsInstance(self.sentiment_analyzer.neutral_keywords, dict)
self.assertIsInstance(self.sentiment_analyzer.negation_words, set)
self.assertIsInstance(self.sentiment_analyzer.intensifier_words, dict)

# Check that keywords are loaded
self.assertGreater(len(self.sentiment_analyzer.positive_keywords), 0)
self.assertGreater(len(self.sentiment_analyzer.negative_keywords), 0)

def test_analyze_text_positive(self):
"""Test sentiment analysis with positive text."""
text = "This is excellent! I love the product."
result = self.sentiment_analyzer.analyze_text(text)

self.assertIsInstance(result, dict)
self.assertIn('sentiment', result)
self.assertIn('score', result)
self.assertIn('positive_words', result)
self.assertIn('negative_words', result)
self.assertIn('confidence', result)

self.assertEqual(result['sentiment'], 'positive')
self.assertGreater(result['score'], 0)
self.assertGreater(len(result['positive_words']), 0)

def test_analyze_text_negative(self):
"""Test sentiment analysis with negative text."""
text = "This is terrible! I hate the service."
result = self.sentiment_analyzer.analyze_text(text)

self.assertEqual(result['sentiment'], 'negative')
self.assertLess(result['score'], 0)
self.assertGreater(len(result['negative_words']), 0)

def test_analyze_text_neutral(self):
"""Test sentiment analysis with neutral text."""
text = "It was okay, nothing special."
result = self.sentiment_analyzer.analyze_text(text)

self.assertEqual(result['sentiment'], 'neutral')
self.assertAlmostEqual(result['score'], 0, places=1)

def test_analyze_text_with_negation(self):
"""Test sentiment analysis with negation."""
text = "This is not good at all."
result = self.sentiment_analyzer.analyze_text(text)

# Should be negative due to negation
self.assertEqual(result['sentiment'], 'negative')
self.assertLess(result['score'], 0)

def test_analyze_text_with_intensifier(self):
"""Test sentiment analysis with intensifier."""
text = "This is very excellent!"
result = self.sentiment_analyzer.analyze_text(text)

self.assertEqual(result['sentiment'], 'positive')
self.assertGreater(result['score'], 0)

def test_analyze_text_empty(self):
"""Test sentiment analysis with empty text."""
result = self.sentiment_analyzer.analyze_text("")

self.assertEqual(result['sentiment'], 'neutral')
self.assertEqual(result['score'], 0)
self.assertEqual(len(result['positive_words']), 0)
self.assertEqual(len(result['negative_words']), 0)

def test_analyze_text_none(self):
```

```
"""Test sentiment analysis with None text."""
result = self.sentiment_analyzer.analyze_text(None)

self.assertEqual(result['sentiment'], 'neutral')
self.assertEqual(result['score'], 0)

def test_analyze_column(self):
"""Test sentiment analysis for a column."""
result = self.sentiment_analyzer.analyze_column(self.test_data, 'feedback')

self.assertIsInstance(result, dict)
self.assertIn('positive', result)
self.assertIn('negative', result)
self.assertIn('neutral', result)
self.assertIn('positive_pct', result)
self.assertIn('negative_pct', result)
self.assertIn('neutral_pct', result)
self.assertIn('avg_score', result)
self.assertIn('total_responses', result)

# Check that percentages sum to approximately 100
total_pct = result['positive_pct'] + result['negative_pct'] + result['neutral_pct']
self.assertAlmostEqual(total_pct, 100.0, places=1)

def test_analyze_column_empty_data(self):
"""Test sentiment analysis with empty data."""
result = self.sentiment_analyzer.analyze_column([], 'feedback')

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

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')

```

```
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_sentiment_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")
```

```

def test_analyze_text_complex_sentence(self):
"""Test sentiment analysis with complex sentence."""
text = "This product is absolutely fantastic and I would highly recommend it to everyone!"
result = self.sentiment_analyzer.analyze_text(text)

self.assertEqual(result['sentiment'], 'positive')
self.assertGreater(result['score'], 0)
self.assertGreater(len(result['positive_words']), 0)

def test_analyze_text_mixed_sentiment(self):
"""Test sentiment analysis with mixed sentiment."""
text = "The product is good but the service is terrible."
result = self.sentiment_analyzer.analyze_text(text)

# Should have both positive and negative words
self.assertGreater(len(result['positive_words']), 0)
self.assertGreater(len(result['negative_words']), 0)

def test_analyze_text_no_sentiment_words(self):
"""Test sentiment analysis with no sentiment words."""
text = "The product arrived on time and was delivered to the correct address."
result = self.sentiment_analyzer.analyze_text(text)

self.assertEqual(result['sentiment'], 'neutral')
self.assertEqual(len(result['positive_words']), 0)
self.assertEqual(len(result['negative_words']), 0)


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

Statistical Analyzer Tests - test_stats_analyzer.py

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

import unittest
import sys
import os
sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
from stats_analyzer import StatsAnalyzer


class TestStatsAnalyzer(unittest.TestCase):
"""Test cases for the StatsAnalyzer class."""

def setUp(self):
"""Set up test fixtures."""
self.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'},
{'age': '28', 'gender': 'Female', 'region': 'North', 'satisfaction': 'High'},
{'age': '42', 'gender': 'Male', 'region': 'South', 'satisfaction': 'Medium'},
{'age': '33', 'gender': 'Female', 'region': 'East', 'satisfaction': 'Low'}
]
self.stats_analyzer = StatsAnalyzer(self.test_data)

def test_initialization(self):
"""Test StatsAnalyzer initialization."""
self.assertEqual(self.stats_analyzer.total_responses, 6)
self.assertEqual(len(self.stats_analyzer.columns), 4)
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.

Sample Survey Data - sample_survey.csv

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

12. Requirements and Dependencies

The project uses only Python standard library modules, making it easy to run without external dependencies. This ensures maximum compatibility and ease of deployment.

13. Project Documentation

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

Project Documentation - README.md

~~~
# Survey Data Analyzer and Response Processor

**A comprehensive Command Line Interface (CLI) tool for analyzing survey responses from CSV files**

## Project Overview

The Survey Data Analyzer is a modular Python CLI application designed to perform structured statistical analysis, basic text sentiment analysis, identify response patterns, and generate readable reports from survey data stored in CSV format.

### Key Features

- **CSV Input Handling**: Load and validate structured survey data from CSV files
- **Survey Summary**: Calculate response counts, percentages, and demographic breakdowns
- **Statistical Analysis**: Perform cross-tabulation and Chi-Square tests
- **Sentiment Analysis**: Analyze text responses using keyword-based sentiment scoring
- **Pattern Recognition**: Identify correlations and trends between survey responses
- **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:
~~~

~~~~
51: 52: ```bash 53: python main.py 54: ``` 55: 56: ### Usage Example 57: 58: 1.
**Load Survey Data**: Choose option 1 and provide the path to your CSV file 59:
2. **View Summary Statistics**: Choose option 2 to see demographic breakdowns 60:
3. **Analyze Sentiment**: Choose option 3 for text sentiment analysis 61: 4.
**Cross-tabulate Results**: Choose option 4 for statistical analysis 62: 5.
**Detect Patterns**: Choose option 5 to find correlations 63: 6. **Generate
Report**: Choose option 6 to create a comprehensive report 64: 65: ## Sample Data
66: 67: The project includes `sample_survey.csv` with example data containing:
68: - Age, gender, region, education demographics 69: - Satisfaction ratings 70:
- Text feedback responses 71: - Recommendation preferences 72: 73: ## Technical
Requirements 74: 75: ### Python Version 76: - **Minimum**: Python 3.8 77: -
**Recommended**: Python 3.9+ 78: 79: ### Standard Library Modules Used 80: -
`csv`: CSV file handling 81: - `re`: Regular expressions for text processing 82:
- `math`: Mathematical calculations 83: - `statistics`: Statistical functions 84:
- `os`: Operating system interface 85: - `sys`: System-specific parameters 86: -
`datetime`: Date and time handling 87: - `itertools`: Iterator building blocks
88: - `collections`: Specialized container datatypes 89: - `typing`: Type hints
support 90: 91: ### Object-Oriented Design 92: - **Class Hierarchies**: Modular
class structure for each component 93: - **Inheritance**: Base classes for common
functionality 94: - **Polymorphism**: Flexible method implementations 95: 96: ##
User Guide 97: 98: ### 1. Loading Survey Data 99: 100: The application accepts
CSV files with the following requirements:
~~~~

```
101: - **File Format**: Comma-separated values (CSV) 102: - **Encoding**: UTF-8,
Latin-1, or CP1252 103: - **Size Limit**: Maximum 50MB 104: - **Required
Columns**: At least 2 columns with valid data 105: 106: ### 2. Survey Summary
Analysis 107: 108: View comprehensive statistics including: 109: - Total response
count and response rate 110: - Demographic breakdowns (age, gender, region,
education) 111: - Response distributions for each question 112: - Data quality
metrics 113: 114: ### 3. Sentiment Analysis 115: 116: Analyze text responses
using: 117: - **Positive Keywords**: excellent, amazing, great, good, etc. 118: -
**Negative Keywords**: terrible, awful, bad, poor, etc. 119: - **Negation
Handling**: "not good" → negative sentiment 120: - **Intensifier Recognition**:
"very good" → amplified positive 121: 122: ### 4. Statistical Analysis 123: 124:
Perform advanced statistical tests: 125: - **Cross-tabulation**: Compare
responses across different questions 126: - **Chi-Square Tests**: Determine
statistical significance 127: - **Correlation Analysis**: Find relationships
between variables 128: 129: ### 5. Pattern Detection 130: 131: Identify
meaningful patterns: 132: - **Demographic Patterns**: Age, gender, regional
differences 133: - **Response Correlations**: Related answers across questions
134: - **Combination Patterns**: Common response combinations 135: - **Outlier
Detection**: Unusual or rare responses 136: 137: ### 6. Report Generation 138:
139: Generate comprehensive reports including: 140: - Executive summary 141: -
Demographic analysis 142: - Sentiment analysis results 143: - Pattern detection
findings 144: - Statistical insights 145: - Key findings and recommendations 146:
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
for 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 without external
dependencies. Perfect for educational environments and learning Python
programming concepts.
```