Computing for Social Sciences with Python - Lecture Notes

Table of Contents

Assignment 2: Exploring and Visualizing Data

Learning Objectives

- Load and examine datasets using pandas
- Create basic visualizations with matplotlib and seaborn
- Understand descriptive statistics
- Identify patterns and outliers in data

Key Libraries

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

Loading Data

```python
```

```
# Reading CSV files
df = pd.read_csv('data.csv')

# Reading Excel files
df = pd.read_excel('data.xlsx')

# Reading from URL
df = pd.read_csv('https://example.com/data.csv')
```

Basic Data Exploration

python

```
# First look at your data
print(df.head())      # First 5 rows
print(df.tail())      # Last 5 rows
print(df.info())      # Data types and missing values
print(df.describe())  # Summary statistics
print(df.shape)       # Dimensions (rows, columns)
```

Example: Exploring Survey Data

python

```
# Load sample data (replace with your dataset)
data = {
    'age': [22, 25, 30, 35, 28, 32, 29, 26, 31, 24],
    'income': [35000, 42000, 55000, 68000, 48000, 62000, 51000, 45000, 58000, 38000],
    'education': ['Bachelor', 'Master', 'PhD', 'Master', 'Bachelor', 'PhD', 'Master', 'Bachelor', 'PhD', 'Bachelor'],
    'satisfaction': [7, 8, 6, 9, 7, 8, 6, 7, 9, 8]
}
df = pd.DataFrame(data)

# Basic statistics
print("Mean age:", df['age'].mean())
print("Median income:", df['income'].median())
print("Education distribution:")
print(df['education'].value_counts())
```

Creating Basic Visualizations

python

```
# Histogram
plt.figure(figsize=(8, 6))
plt.hist(df['age'], bins=10, edgecolor='black')
plt.title('Distribution of Ages')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()

# Scatter plot
plt.figure(figsize=(8, 6))
plt.scatter(df['age'], df['income'])
plt.title('Age vs Income')
plt.xlabel('Age')
plt.ylabel('Income')
plt.show()

# Box plot with seaborn
plt.figure(figsize=(8, 6))
sns.boxplot(x='education', y='income', data=df)
plt.title('Income by Education Level')
plt.xticks(rotation=45)
plt.show()
```

Practice Tips

- Always start with `df.head()` and `df.info()` to understand your data
- Look for missing values with `df.isnull().sum()`
- Use `df.columns` to see all column names
- Try different plot types to find the most appropriate visualization

---

Assignment 3: Wrangling and Visualizing Data

Learning Objectives

- Clean and transform datasets
- Handle missing values
- Create new variables
- Produce publication-ready visualizations

Data Cleaning Basics

```python
# Handling missing values
df.dropna()                    # Remove rows with any missing values
df.dropna(subset=['column'])   # Remove rows with missing values in specific column
df.fillna(0)                   # Fill missing values with 0
df.fillna(df.mean())           # Fill with mean (numeric columns only)
df.fillna(method='forward')    # Forward fill
```

Data Transformation

```python
# Creating new columns
df['new_column'] = df['column1'] + df['column2']
df['log_income'] = np.log(df['income'])
df['age_group'] = pd.cut(df['age'], bins=[0, 25, 35, 50, 100],
                         labels=['18-25', '26-35', '36-50', '50+'])

# String operations
df['name_upper'] = df['name'].str.upper()
df['name_length'] = df['name'].str.len()

# Date operations
df['date'] = pd.to_datetime(df['date_string'])
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
```

Example: Cleaning Survey Data

```python
```

```
# Sample messy data
messy_data = {
    'Name': ['John Doe', 'jane smith', 'BOB JOHNSON', None, 'Mary Jane'],
    'Age': [25, 30, None, 28, 35],
    'Income': ['$50,000', '$60000', '45k', '$55,000', None],
    'Date': ['2023-01-15', '01/20/2023', '2023-02-28', '2023-03-10', '2023-04-05']
}
df = pd.DataFrame(messy_data)

# Clean the data
# Fix names
df['Name'] = df['Name'].str.title()  # Proper case
df['Name'].fillna('Unknown', inplace=True)

# Clean income column
df['Income'] = df['Income'].str.replace('$', '').str.replace(',', '').str.replace('k', '000')
df['Income'] = pd.to_numeric(df['Income'], errors='coerce')
df['Income'].fillna(df['Income'].mean(), inplace=True)

# Convert dates
df['Date'] = pd.to_datetime(df['Date'])

print(df)
```

Advanced Visualizations

```
python
```

```
# Subplots
fig, axes = plt.subplots(2, 2, figsize=(12, 10))

# Histogram
axes[0,0].hist(df['age'], bins=10)
axes[0,0].set_title('Age Distribution')

# Scatter plot
axes[0,1].scatter(df['age'], df['income'])
axes[0,1].set_title('Age vs Income')

# Bar plot
education_counts = df['education'].value_counts()
axes[1,0].bar(education_counts.index, education_counts.values)
axes[1,0].set_title('Education Levels')

# Box plot
df.boxplot(column='satisfaction', by='education', ax=axes[1,1])
axes[1,1].set_title('Satisfaction by Education')

plt.tight_layout()
plt.show()
```

Seaborn for Advanced Plots

python

```
# Correlation heatmap
plt.figure(figsize=(10, 8))
correlation_matrix = df.select_dtypes(include=[np.number]).corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Correlation Matrix')
plt.show()

# Pair plot
sns.pairplot(df, hue='education')
plt.show()
```

Assignment 4: Programming in Python

Learning Objectives

- Write functions for data analysis
- Use control structures (loops, conditionals)
- Apply list comprehensions
- Handle errors gracefully

Functions

python

```
def calculate_mean(numbers):
    """Calculate the mean of a list of numbers."""
    if len(numbers) == 0:
        return 0
    return sum(numbers) / len(numbers)

def categorize_age(age):
    """Categorize age into groups."""
    if age < 18:
        return 'Minor'
    elif age < 65:
        return 'Adult'
    else:
        return 'Senior'

# Using functions
ages = [25, 30, 17, 70, 45]
mean_age = calculate_mean(ages)
print(f"Mean age: {mean_age}")

# Apply function to DataFrame
df['age_category'] = df['age'].apply(categorize_age)
```

Loops and Conditionals

python

```
# For loops
total_income = 0
for income in df['income']:
    total_income += income
print(f"Total income: {total_income}")

# While loops
i = 0
while i < len(df) and df.iloc[i]['age'] < 30:
    print(f"Person {i} is under 30")
    i += 1

# Conditional processing
high_earners = []
for index, row in df.iterrows():
    if row['income'] > 50000:
        high_earners.append(row['name'])
```

List Comprehensions

python

```
# Basic list comprehension
squared_ages = [age**2 for age in df['age']]

# With conditions
high_earners = [name for name, income in zip(df['name'], df['income'])
                if income > 50000]

# Dictionary comprehension
income_by_education = {education: df[df['education'] == education]['income'].mean()
                      for education in df['education'].unique()}
```

Example: Data Analysis Functions

python

```python
def analyze_group(df, group_column, value_column):
    """Analyze a specific group in the dataset."""
    results = {}

    for group in df[group_column].unique():
        group_data = df[df[group_column] == group][value_column]

        results[group] = {
            'count': len(group_data),
            'mean': group_data.mean(),
            'median': group_data.median(),
            'std': group_data.std()
        }

    return results

# Use the function
education_analysis = analyze_group(df, 'education', 'income')
for education, stats in education_analysis.items():
    print(f"{education}: Mean income = ${stats['mean']:,.2f}")
```

Error Handling

python

```
def safe_divide(a, b):
    """Safely divide two numbers."""
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        print("Cannot divide by zero!")
        return None
    except TypeError:
        print("Both arguments must be numbers!")
        return None

def load_data_safely(filename):
    """Load data with error handling."""
    try:
        df = pd.read_csv(filename)
        print(f"Successfully loaded {len(df)} rows")
        return df
    except FileNotFoundError:
        print(f"File {filename} not found!")
        return None
    except pd.errors.EmptyDataError:
        print("File is empty!")
        return None
```

Assignment 5: Debugging and Troubleshooting

Learning Objectives

- Identify common Python errors
- Use debugging techniques
- Write defensive code
- Test your functions

Common Error Types

```
python
```

```
# NameError - using undefined variable
# print(undefined_variable)  # This will cause NameError

# TypeError - wrong data type
# result = "hello" + 5  # This will cause TypeError

# KeyError - accessing non-existent dictionary key
# data = {"name": "John"}
# print(data["age"])  # This will cause KeyError

# IndexError - accessing invalid list index
# my_list = [1, 2, 3]
# print(my_list[5])  # This will cause IndexError
```

Debugging Techniques

python

```
# 1. Print debugging
def calculate_bmi(weight, height):
    print(f"Input: weight={weight}, height={height}")  # Debug print

    bmi = weight / (height ** 2)
    print(f"Calculated BMI: {bmi}")  # Debug print

    return bmi

# 2. Using assert statements
def calculate_percentage(part, whole):
    assert whole != 0, "Whole cannot be zero"
    assert part >= 0, "Part cannot be negative"
    assert whole > 0, "Whole must be positive"

    return (part / whole) * 100

# 3. Logging
import logging

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

def process_data(df):
    logger.info(f"Processing {len(df)} rows")

    if df.empty:
        logger.warning("DataFrame is empty!")
        return df

    # Process data...
    logger.debug("Data processing complete")
    return df
```

Writing Defensive Code

```
python
```

```
def safe_mean(numbers):
    """Calculate mean with input validation."""

    # Check if input is empty
    if not numbers:
        return None

    # Check if all elements are numbers
    try:
        numeric_numbers = [float(x) for x in numbers]
    except (ValueError, TypeError):
        print("All elements must be numeric")
        return None

    # Calculate mean
    return sum(numeric_numbers) / len(numeric_numbers)

def load_and_validate_data(filename, required_columns):
    """Load data and validate it has required columns."""

    try:
        df = pd.read_csv(filename)
    except Exception as e:
        print(f"Error loading file: {e}")
        return None

    # Check for required columns
    missing_columns = set(required_columns) - set(df.columns)
    if missing_columns:
        print(f"Missing required columns: {missing_columns}")
        return None

    return df
```

Testing Your Functions

```
python
```

```
def test_calculate_bmi():
    """Test the BMI calculation function."""

    # Test normal case
    result = calculate_bmi(70, 1.75)
    expected = 22.86  # approximately
    assert abs(result - expected) < 0.01, f"Expected ~{expected}, got {result}"

    # Test edge cases
    try:
        calculate_bmi(70, 0)  # Should handle division by zero
        assert False, "Should have raised an error for zero height"
    except:
        pass  # Expected to fail

    print("All BMI tests passed!")

# Run tests
test_calculate_bmi()
```

Example: Debugging a Data Analysis Function

python

```
def analyze_survey_data(df, debug=False):
    """Analyze survey data with debugging capabilities."""

    if debug:
        print(f"Input DataFrame shape: {df.shape}")
        print(f"Columns: {list(df.columns)}")

    # Check for required columns
    required_cols = ['age', 'income', 'satisfaction']
    missing_cols = [col for col in required_cols if col not in df.columns]

    if missing_cols:
        raise ValueError(f"Missing required columns: {missing_cols}")

    # Remove rows with missing values
    original_length = len(df)
    df_clean = df.dropna(subset=required_cols)

    if debug:
        print(f"Removed {original_length - len(df_clean)} rows with missing values")

    # Calculate statistics
    results = {
        'sample_size': len(df_clean),
        'mean_age': df_clean['age'].mean(),
        'mean_income': df_clean['income'].mean(),
        'mean_satisfaction': df_clean['satisfaction'].mean()
    }

    if debug:
        print("Analysis results:", results)

    return results

# Use with debugging
# results = analyze_survey_data(df, debug=True)
```

Assignment 6: Generating Reproducible Analysis

Learning Objectives

- Create reproducible workflows

- Document your code effectively
- Use version control concepts
- Generate automated reports

Code Documentation

```python
```

```python
def analyze_demographics(df, group_var, outcome_var):
    """
    Analyze demographic differences in outcomes.

    Parameters:
    -----------
    df : pandas.DataFrame
        The input dataset
    group_var : str
        The column name for grouping variable
    outcome_var : str
        The column name for outcome variable

    Returns:
    --------
    dict
        Dictionary containing summary statistics for each group

    Examples:
    ---------
    >>> results = analyze_demographics(df, 'education', 'income')
    >>> print(results['Bachelor']['mean'])
    """

    if group_var not in df.columns:
        raise ValueError(f"Column '{group_var}' not found in DataFrame")

    if outcome_var not in df.columns:
        raise ValueError(f"Column '{outcome_var}' not found in DataFrame")

    results = {}

    for group in df[group_var].unique():
        group_data = df[df[group_var] == group][outcome_var]

        results[group] = {
            'n': len(group_data),
            'mean': group_data.mean(),
            'std': group_data.std(),
            'median': group_data.median(),
            'min': group_data.min(),
            'max': group_data.max()
        }
    return results
```

Creating Configuration Files

python

```python
# config.py
CONFIG = {
    'data_path': 'data/survey_data.csv',
    'output_path': 'results/',
    'figure_size': (10, 8),
    'random_seed': 42,
    'missing_value_threshold': 0.1,
    'categorical_columns': ['education', 'gender', 'region'],
    'numerical_columns': ['age', 'income', 'satisfaction']
}

# Using configuration
import config

def load_project_data():
    """Load data using configuration settings."""
    return pd.read_csv(config.CONFIG['data_path'])

def save_figure(fig, filename):
    """Save figure using configuration settings."""
    filepath = config.CONFIG['output_path'] + filename
    fig.savefig(filepath, dpi=300, bbox_inches='tight')
```

Reproducible Analysis Script

python

```python
#!/usr/bin/env python3
"""
Social Science Data Analysis Pipeline
=====================================

This script performs a complete analysis of survey data including:
1. Data loading and cleaning
2. Exploratory data analysis
3. Statistical testing
4. Visualization
5. Report generation

Author: Your Name
Date: 2024
Version: 1.0
"""

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
import config

# Set random seed for reproducibility
np.random.seed(config.CONFIG['random_seed'])

def main():
    """Main analysis pipeline."""

    print("=== Social Science Data Analysis ===")
    print("Starting analysis pipeline...")

    # Step 1: Load data
    print("\n1. Loading data...")
    df = load_project_data()
    print(f"Loaded {len(df)} observations with {len(df.columns)} variables")

    # Step 2: Clean data
    print("\n2. Cleaning data...")
    df_clean = clean_data(df)
    print(f"After cleaning: {len(df_clean)} observations")
    # Step 3: Descriptive analysis
    print("\n3. Performing descriptive analysis...")
    descriptive_stats = generate_descriptive_stats(df_clean)

    # Step 4: Statistical tests
    print("\n4. Running statistical tests...")
    test_results = run_statistical_tests(df_clean)

    # Step 5: Create visualizations
    print("\n5. Creating visualizations...")
    create_visualizations(df_clean)

    # Step 6: Generate report
    print("\n6. Generating report...")
    generate_report(descriptive_stats, test_results)

    print("\nAnalysis complete! Check the results/ directory for outputs.")

def clean_data(df):
    """Clean the input dataset."""
    df_clean = df.copy()

    # Remove rows with too many missing values
    threshold = len(df_clean.columns) * config.CONFIG['missing_value_threshold']
    df_clean = df_clean.dropna(thresh=len(df_clean.columns) - threshold)

    # Handle missing values in specific columns
    for col in config.CONFIG['numerical_columns']:
        if col in df_clean.columns:
            df_clean[col].fillna(df_clean[col].median(), inplace=True)

    return df_clean

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

Automated Report Generation

```
python
```

```python
def generate_html_report(data, analysis_results, output_file="report.html"):
    """Generate an HTML report of the analysis."""

    html_template = """
    <!DOCTYPE html>
    <html>
    <head>
        <title>Social Science Data Analysis Report</title>
        <style>
            body { font-family: Arial, sans-serif; margin: 40px; }
            h1 { color: #2c3e50; }
            h2 { color: #34495e; }
            table { border-collapse: collapse; width: 100%; }
            th, td { border: 1px solid #ddd; padding: 8px; text-align: left; }
            th { background-color: #f2f2f2; }
            .summary { background-color: #ecf0f1; padding: 20px; border-radius: 5px; }
        </style>
    </head>
    <body>
        <h1>Social Science Data Analysis Report</h1>

        <div class="summary">
            <h2>Summary</h2>
            <p>This report presents analysis of survey data with {n_observations} observations.</p>
            <p>Analysis completed on: {date}</p>
        </div>

        <h2>Descriptive Statistics</h2>
        {descriptive_table}

        <h2>Key Findings</h2>
        <ul>
            {findings_list}
        </ul>

    </body>
    </html>
    """

    from datetime import datetime

    # Create descriptive statistics table
    desc_stats = data.describe()
    descriptive_table = desc_stats.to_html(classes='table')

    # Format findings
    findings = [
        f"Average age: {data['age'].mean():.1f} years",
        f"Average income: ${data['income'].mean():,.0f}",
        f"Sample size: {len(data)} participants"
    ]
    findings_list = ''.join([f"<li>{finding}</li>" for finding in findings])

    # Fill template
    html_content = html_template.format(
        n_observations=len(data),
        date=datetime.now().strftime("%Y-%m-%d %H:%M"),
        descriptive_table=descriptive_table,
        findings_list=findings_list
    )

    # Save report
    with open(output_file, 'w') as f:
        f.write(html_content)

    print(f"HTML report saved to {output_file}")
```

Assignment 7: Machine Learning

Learning Objectives

- Understand basic machine learning concepts
- Implement supervised learning models
- Evaluate model performance
- Apply ML to social science questions

Essential Libraries

```
python
```

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, mean_squared_error, classification_report
from sklearn.preprocessing import StandardScaler, LabelEncoder
import matplotlib.pyplot as plt
import seaborn as sns
```

Data Preparation

python

```python
def prepare_ml_data(df, target_column, feature_columns=None):
    """Prepare data for machine learning."""

    # Select features
    if feature_columns is None:
        feature_columns = [col for col in df.columns if col != target_column]

    X = df[feature_columns].copy()
    y = df[target_column].copy()

    # Handle categorical variables
    categorical_cols = X.select_dtypes(include=['object']).columns
    for col in categorical_cols:
        le = LabelEncoder()
        X[col] = le.fit_transform(X[col].astype(str))

    # Handle missing values
    X.fillna(X.median(), inplace=True)

    return X, y

# Example usage
# X, y = prepare_ml_data(df, target_column='income',
#                        feature_columns=['age', 'education', 'experience'])
```

Linear Regression Example

python

```
def predict_income_linear(df):
    """Predict income using linear regression."""

    # Prepare data
    X = df[['age', 'years_education', 'experience']].copy()
    y = df['income'].copy()

    # Handle missing values
    X.fillna(X.mean(), inplace=True)
    y.fillna(y.mean(), inplace=True)

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42
    )

    # Create and train model
    model = LinearRegression()
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Evaluate model
    mse = mean_squared_error(y_test, y_pred)
    r2 = model.score(X_test, y_test)

    print(f"Mean Squared Error: {mse:,.2f}")
    print(f"R-squared Score: {r2:.3f}")

    # Feature importance
    feature_importance = pd.DataFrame({
        'feature': X.columns,
        'coefficient': model.coef_,
        'abs_coefficient': np.abs(model.coef_)
    }).sort_values('abs_coefficient', ascending=False)

    print("\nFeature Importance:")
    print(feature_importance)

    return model, feature_importance
```

Classification Example

python

```
def predict_satisfaction_category(df):
    """Predict satisfaction category using classification."""

    # Create satisfaction categories
    df['satisfaction_category'] = pd.cut(
        df['satisfaction'],
        bins=[0, 5, 7, 10],
        labels=['Low', 'Medium', 'High']
    )

    # Prepare features
    feature_columns = ['age', 'income', 'years_education']
    X = df[feature_columns].copy()
    y = df['satisfaction_category'].copy()

    # Handle missing values
    X.fillna(X.median(), inplace=True)

    # Remove rows where target is missing
    mask = y.notna()
    X, y = X[mask], y[mask]

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42, stratify=y
    )

    # Try different models
    models = {
        'Logistic Regression': LogisticRegression(random_state=42),
        'Decision Tree': DecisionTreeClassifier(random_state=42),
        'Random Forest': RandomForestClassifier(random_state=42, n_estimators=100)
    }

    results = {}

    for name, model in models.items():
        # Train model
        model.fit(X_train, y_train)

        # Make predictions
        y_pred = model.predict(X_test)
        # Calculate accuracy
        accuracy = accuracy_score(y_test, y_pred)
        results[name] = accuracy

        print(f"\n{name} Results:")
        print(f"Accuracy: {accuracy:.3f}")
        print(f"Classification Report:\n{classification_report(y_test, y_pred)}")

    return results

# Example: Compare models
# model_results = predict_satisfaction_category(df)
```

Cross-Validation

python

```
def evaluate_model_cv(X, y, model, cv=5):
    """Evaluate model using cross-validation."""

    # Perform cross-validation
    cv_scores = cross_val_score(model, X, y, cv=cv, scoring='accuracy')

    print(f"Cross-validation results:")
    print(f"Mean accuracy: {cv_scores.mean():.3f}")
    print(f"Standard deviation: {cv_scores.std():.3f}")
    print(f"Individual fold scores: {cv_scores}")

    return cv_scores

# Example usage
# X, y = prepare_ml_data(df, 'satisfaction_category')
# model = RandomForestClassifier(random_state=42)
# cv_scores = evaluate_model_cv(X, y, model)
```

Visualization of Results

python

```
def plot_model_results(y_true, y_pred, model_name):
    """Plot model results."""

    plt.figure(figsize=(12, 4))

    # Subplot 1: Actual vs Predicted
    plt.subplot(1, 3, 1)
    plt.scatter(y_true, y_pred, alpha=0.5)
    plt.plot([y_true.min(), y_true.max()], [y_true.min(), y_true.max()], 'r--')
    plt.xlabel('Actual Values')
    plt.ylabel('Predicted Values')
    plt.title(f'{model_name}: Actual vs Predicted')

    # Subplot 2: Residuals
    plt.subplot(1, 3, 2)
    residuals = y_true - y_pred
    plt.scatter(y_pred, residuals, alpha=0.5)
    plt.axhline(y=0, color='r', linestyle='--')
    plt.xlabel('Predicted Values')
    plt.ylabel('Residuals')
    plt.title('Residual Plot')

    # Subplot 3: Residual histogram
    plt.subplot(1, 3, 3)
    plt.hist(residuals, bins=20, edgecolor='black')
    plt.xlabel('Residuals')
    plt.ylabel('Frequency')
    plt.title('Residual Distribution')

    plt.tight_layout()
    plt.show()
```

Social Science Application Example

```
python
```

```
def analyze_education_income_relationship(df):
    """Analyze the relationship between education and income using ML."""

    print("=== Education-Income Relationship Analysis ===")

    # Prepare data
    # Convert education to numeric if it's categorical
    if df['education'].dtype == 'object':
        education_mapping = {'High School': 12, 'Bachelor': 16, 'Master': 18, 'PhD': 22}
        df['education_years'] = df['education'].map(education_mapping)
    else:
        df['education_years'] = df['education']

    # Features and target
    X = df[['age', 'education_years', 'experience']].copy()
    y = df['income'].copy()

    # Clean data
    mask = X.notna().all(axis=1) & y.notna()
    X, y = X[mask], y[mask]

    # Train model
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    model = LinearRegression()
    model.fit(X_train, y_train)

    # Results
    y_pred = model.predict(X_test)
    r2 = model.score(X_test, y_test)

    print(f"Model explains {r2:.1%} of income variation")
    print(f"Each additional year of education is associated with ${model.coef_[1]:,.0f} more income")

    return model

---

## Assignment 8: Collecting and Analyzing Web Data

### Learning Objectives
- Scrape data from websites ethically
- Work with APIs
```

~~~~
- Handle web data formats (JSON, XML)
- Analyze social media and web content

### Web Scraping Basics
```python
import requests
from bs4 import BeautifulSoup
import pandas as pd
import json
import time

def scrape_news_headlines(url, delay=1):
    """
    Scrape news headlines from a website.
    Always check robots.txt and terms of service first!
    """

    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'
    }

    try:
        # Make request with delay to be respectful
        time.sleep(delay)
        response = requests.get(url, headers=headers)
        response.raise_for_status()

        # Parse HTML
        soup = BeautifulSoup(response.content, 'html.parser')

        # Extract headlines (this will vary by website)
        headlines = []
        for headline_tag in soup.find_all('h2', class_='headline'):  # Example selector
            headlines.append(headline_tag.get_text().strip())

        return headlines

    except requests.RequestException as e:
        print(f"Error fetching data: {e}")
        return []

# Example usage (replace with actual news site)
# headlines = scrape_news_headlines('https://example-news-site.com')
~~~~

Working with APIs

python

```
def get_weather_data(city, api_key):
    """Get weather data from OpenWeatherMap API."""

    base_url = "http://api.openweathermap.org/data/2.5/weather"
    params = {
        'q': city,
        'appid': api_key,
        'units': 'metric'
    }

    try:
        response = requests.get(base_url, params=params)
        response.raise_for_status()

        data = response.json()

        # Extract relevant information
        weather_info = {
            'city': data['name'],
            'country': data['sys']['country'],
            'temperature': data['main']['temp'],
            'humidity': data['main']['humidity'],
            'description': data['weather'][0]['description'],
            'timestamp': pd.Timestamp.now()
        }

        return weather_info

    except requests.RequestException as e:
        print(f"Error fetching weather data: {e}")
        return None

def collect_multiple_cities_weather(cities, api_key, delay=1):
    """Collect weather data for multiple cities."""

    weather_data = []

    for city in cities:
        print(f"Fetching weather for {city}...")
        weather = get_weather_data(city, api_key)

        if weather:
            weather_data.append(weather)
        # Be respectful with API calls
        time.sleep(delay)

    return pd.DataFrame(weather_data)

# Example usage
# cities = ['New York', 'London', 'Tokyo', 'Sydney']
# weather_df = collect_multiple_cities_weather(cities, 'your_api_key')
```

Social Media Data Analysis (Twitter/X Example)

python

```python
def analyze_tweet_sentiment(tweets_df):
    """
    Analyze sentiment of tweets using TextBlob.
    Note: For real Twitter data, you'll need Twitter API access.
    """
    from textblob import TextBlob

    def get_sentiment(text):
        """Get sentiment polarity (-1 to 1)."""
        try:
            blob = TextBlob(str(text))
            return blob.sentiment.polarity
        except:
            return 0

    # Add sentiment scores
    tweets_df['sentiment'] = tweets_df['text'].apply(get_sentiment)

    # Categorize sentiment
    def categorize_sentiment(score):
        if score > 0.1:
            return 'Positive'
        elif score < -0.1:
            return 'Negative'
        else:
            return 'Neutral'

    tweets_df['sentiment_category'] = tweets_df['sentiment'].apply(categorize_sentiment)

    # Summary statistics
    sentiment_summary = tweets_df['sentiment_category'].value_counts()
    print("Sentiment Distribution:")
    print(sentiment_summary)

    return tweets_df

# Example with sample data
sample_tweets = {
    'text': [
        "I love this new policy!",
        "This is terrible news",
        "The weather is okay today",
        "Amazing results from the study",
        "Not sure about this decision"
    ],
    'user': ['user1', 'user2', 'user3', 'user4', 'user5'],
    'timestamp': pd.date_range('2024-01-01', periods=5, freq='1H')
}

tweets_df = pd.DataFrame(sample_tweets)
tweets_with_sentiment = analyze_tweet_sentiment(tweets_df)
```

Web Data Cleaning and Processing

```
python
```

```python
def clean_web_scraped_text(text_series):
    """Clean text data scraped from web."""
    import re

    def clean_text(text):
        if pd.isna(text):
            return ""

        # Convert to string
        text = str(text)

        # Remove HTML tags
        text = re.sub(r'<[^>]+>', '', text)

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

        # Remove special characters but keep basic punctuation
        text = re.sub(r'[^\w\s.,!?-]', '', text)

        return text

    return text_series.apply(clean_text)

def extract_urls_from_text(text_series):
    """Extract URLs from text data."""
    import re

    url_pattern = r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\\(\\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+'

    def find_urls(text):
        if pd.isna(text):
            return []
        return re.findall(url_pattern, str(text))

    return text_series.apply(find_urls)

# Example usage
# clean_headlines = clean_web_scraped_text(df['raw_headlines'])
# urls = extract_urls_from_text(df['text_content'])
```

Rate Limiting and Ethical Scraping

python

```python
import time
from datetime import datetime, timedelta

class RateLimiter:
    """Simple rate limiter for web scraping."""

    def __init__(self, max_requests=10, time_window=60):
        self.max_requests = max_requests
        self.time_window = time_window
        self.requests = []

    def wait_if_needed(self):
        """Wait if necessary to respect rate limits."""
        now = datetime.now()

        # Remove old requests outside the time window
        cutoff = now - timedelta(seconds=self.time_window)
        self.requests = [req_time for req_time in self.requests if req_time > cutoff]

        # Check if we need to wait
        if len(self.requests) >= self.max_requests:
            sleep_time = self.time_window - (now - self.requests[0]).total_seconds()
            if sleep_time > 0:
                print(f"Rate limit reached. Waiting {sleep_time:.1f} seconds...")
                time.sleep(sleep_time)

        # Record this request
        self.requests.append(now)

def scrape_responsibly(urls, scraping_function, delay=1, max_requests_per_minute=10):
    """Scrape URLs while respecting rate limits."""

    rate_limiter = RateLimiter(max_requests=max_requests_per_minute, time_window=60)
    results = []

    for i, url in enumerate(urls):
        print(f"Scraping {i+1}/{len(urls)}: {url}")

        # Respect rate limits
        rate_limiter.wait_if_needed()

        # Additional delay between requests
        if i > 0:
            time.sleep(delay)

        try:
            result = scraping_function(url)
            results.append(result)
        except Exception as e:
            print(f"Error scraping {url}: {e}")
            results.append(None)

    return results
```

Assignment 9: Geospatial Visualization

Learning Objectives

- Work with geographic data formats
- Create maps and spatial visualizations
- Analyze spatial patterns
- Understand coordinate systems

Essential Libraries

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import folium
import geopandas as gpd
from shapely.geometry import Point
import plotly.express as px
import plotly.graph_objects as go
```

Basic Map Creation with Folium

```python
```

```
def create_basic_map(center_lat=40.7128, center_lon=-74.0060, zoom=10):
    """Create a basic interactive map."""

    # Create base map
    m = folium.Map(
        location=[center_lat, center_lon],
        zoom_start=zoom,
        tiles='OpenStreetMap'
    )

    return m

def add_markers_to_map(map_obj, locations_df):
    """
    Add markers to a map.
    locations_df should have columns: 'latitude', 'longitude', 'name', 'info'
    """

    for idx, row in locations_df.iterrows():
        folium.Marker(
            location=[row['latitude'], row['longitude']],
            popup=f"<b>{row['name']}</b><br>{row['info']}",
            tooltip=row['name']
        ).add_to(map_obj)

    return map_obj

# Example: Create map with sample data
sample_locations = pd.DataFrame({
    'name': ['New York City', 'Los Angeles', 'Chicago', 'Houston'],
    'latitude': [40.7128, 34.0522, 41.8781, 29.7604],
    'longitude': [-74.0060, -118.2437, -87.6298, -95.3698],
    'info': ['Population: 8.3M', 'Population: 4M', 'Population: 2.7M', 'Population: 2.3M']
})

# Create map
city_map = create_basic_map(center_lat=39.8283, center_lon=-98.5795, zoom=4)
city_map = add_markers_to_map(city_map, sample_locations)

# Save map
# city_map.save('city_map.html')
```

Choropleth Maps

python

```python
def create_choropleth_map(data_df, geo_data, value_column, location_column):
    """
    Create a choropleth (filled) map.

    Parameters:
    - data_df: DataFrame with values to map
    - geo_data: GeoJSON data or file path
    - value_column: Column name with values to visualize
    - location_column: Column name with location identifiers
    """

    # Create base map
    m = folium.Map(location=[39.8283, -98.5795], zoom_start=4)

    # Create choropleth layer
    folium.Choropleth(
        geo_data=geo_data,
        name='choropleth',
        data=data_df,
        columns=[location_column, value_column],
        key_on='feature.properties.NAME',  # Adjust based on your GeoJSON
        fill_color='YlOrRd',
        fill_opacity=0.7,
        line_opacity=0.2,
        legend_name=value_column.title()
    ).add_to(m)

    # Add layer control
    folium.LayerControl().add_to(m)

    return m

# Example with state population data
state_data = pd.DataFrame({
    'state': ['California', 'Texas', 'Florida', 'New York'],
    'population': [39.5, 29.0, 21.5, 19.8],
    'median_income': [75000, 60000, 55000, 70000]
})

# Note: You would need actual GeoJSON data for US states
# choropleth_map = create_choropleth_map(state_data, 'us-states.json', 'population', 'state')
```

Point Data Analysis

```python
def analyze_spatial_distribution(df, lat_col='latitude', lon_col='longitude'):
    """Analyze the spatial distribution of points."""

    # Basic statistics
    print("Spatial Distribution Analysis")
    print("=" * 40)
    print(f"Number of points: {len(df)}")
    print(f"Latitude range: {df[lat_col].min():.4f} to {df[lat_col].max():.4f}")
    print(f"Longitude range: {df[lon_col].min():.4f} to {df[lon_col].max():.4f}")

    # Calculate center point
    center_lat = df[lat_col].mean()
    center_lon = df[lon_col].mean()
    print(f"Center point: ({center_lat:.4f}, {center_lon:.4f})")

    # Create visualization
    fig, axes = plt.subplots(1, 2, figsize=(15, 6))

    # Scatter plot
    axes[0].scatter(df[lon_col], df[lat_col], alpha=0.6)
    axes[0].set_xlabel('Longitude')
    axes[0].set_ylabel('Latitude')
    axes[0].set_title('Geographic Distribution')
    axes[0].grid(True, alpha=0.3)

    # Density plot
    axes[1].hexbin(df[lon_col], df[lat_col], gridsize=20, cmap='Blues')
    axes[1].set_xlabel('Longitude')
    axes[1].set_ylabel('Latitude')
    axes[1].set_title('Point Density')

    plt.tight_layout()
    plt.show()

    return center_lat, center_lon

# Example usage
# center = analyze_spatial_distribution(sample_locations)
```

Distance Calculations

```python
def calculate_distance(lat1, lon1, lat2, lon2):
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees).
    Returns distance in kilometers.
    """
    from math import radians, cos, sin, asin, sqrt

    # Convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # Haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))

    # Radius of earth in kilometers
    r = 6371

    return c * r

def find_nearest_points(df, target_lat, target_lon, n=5):
    """Find the n nearest points to a target location."""

    # Calculate distances
    df = df.copy()
    df['distance_km'] = df.apply(
        lambda row: calculate_distance(
            target_lat, target_lon,
            row['latitude'], row['longitude']
        ), axis=1
    )

    # Return nearest points
    return df.nsmallest(n, 'distance_km')

# Example usage
# nearest_to_nyc = find_nearest_points(sample_locations, 40.7128, -74.0060, n=3)
```

Heatmaps for Geographic Data

python

```python
def create_heatmap(df, lat_col='latitude', lon_col='longitude', weight_col=None):
    """Create a heatmap of geographic points."""

    # Create base map
    center_lat = df[lat_col].mean()
    center_lon = df[lon_col].mean()

    m = folium.Map(location=[center_lat, center_lon], zoom_start=6)

    # Prepare data for heatmap
    if weight_col:
        heat_data = [[row[lat_col], row[lon_col], row[weight_col]]
                    for idx, row in df.iterrows()]
    else:
        heat_data = [[row[lat_col], row[lon_col]]
                    for idx, row in df.iterrows()]

    # Add heatmap
    from folium.plugins import HeatMap
    HeatMap(heat_data).add_to(m)

    return m

# Example with weighted heatmap
sample_locations['weight'] = [100, 80, 60, 40]  # Population weights
# heatmap = create_heatmap(sample_locations, weight_col='weight')
```

Plotting with Plotly for Interactive Maps

python

```python
def create_interactive_scatter_map(df, lat_col='latitude', lon_col='longitude',
                                  size_col=None, color_col=None, hover_col=None):
    """Create an interactive scatter plot map using Plotly."""

    fig = px.scatter_mapbox(
        df,
        lat=lat_col,
        lon=lon_col,
        size=size_col,
        color=color_col,
        hover_name=hover_col,
        hover_data={lat_col: ':.4f', lon_col: ':.4f'},
        mapbox_style='open-street-map',
        zoom=3,
        height=600,
        title='Interactive Geographic Visualization'
    )

    fig.update_layout(
        mapbox=dict(
            center=dict(
                lat=df[lat_col].mean(),
                lon=df[lon_col].mean()
            )
        )
    )

    return fig

# Example usage
# Add some additional data for visualization
sample_locations['category'] = ['Major City', 'Major City', 'Major City', 'Major City']
sample_locations['population_millions'] = [8.3, 4.0, 2.7, 2.3]

# Create interactive map
# fig = create_interactive_scatter_map(
#     sample_locations,
#     size_col='population_millions',
#     color_col='category',
#     hover_col='name'
# )
# fig.show()
```

Assignment 10: Analyzing Text Data

Learning Objectives

- Process and clean text data
- Perform sentiment analysis
- Extract topics from text
- Visualize text analysis results

Text Processing Basics

```python
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter
import re
from wordcloud import WordCloud

# For advanced text analysis
try:
    from textblob import TextBlob
    from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
    from sklearn.decomposition import LatentDirichletAllocation
    import nltk
    # Download required NLTK data
    nltk.download('punkt', quiet=True)
    nltk.download('stopwords', quiet=True)
    nltk.download('wordnet', quiet=True)
except ImportError:
    print("Some libraries may need to be installed: pip install textblob scikit-learn wordcloud nltk")

def clean_text(text):
    """Clean and preprocess text data."""
    if pd.isna(text):
        return ""

    # Convert to string and lowercase
    text = str(text).lower()

    # Remove URLs
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)

    # Remove user mentions and hashtags (for social media text)
    text = re.sub(r'@\w+|#\w+', '', text)

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

    # Remove special characters but keep basic punctuation
    text = re.sub(r'[^\w\s.,!?]', '', text)

    return text
```

```
def remove_stopwords(text, custom_stopwords=None):
    """Remove common stopwords from text."""
    from nltk.corpus import stopwords

    stop_words = set(stopwords.words('english'))

    # Add custom stopwords if provided
    if custom_stopwords:
        stop_words.update(custom_stopwords)

    # Tokenize and remove stopwords
    words = text.split()
    filtered_words = [word for word in words if word.lower() not in stop_words]

    return ' '.join(filtered_words)

# Example text cleaning pipeline
def preprocess_text_column(df, text_column):
    """Complete text preprocessing pipeline."""

    print(f"Preprocessing {text_column} column...")

    # Clean text
    df[f'{text_column}_clean'] = df[text_column].apply(clean_text)

    # Remove stopwords
    df[f'{text_column}_no_stopwords'] = df[f'{text_column}_clean'].apply(remove_stopwords)

    # Calculate text length
    df[f'{text_column}_length'] = df[f'{text_column}_clean'].str.len()
    df[f'{text_column}_word_count'] = df[f'{text_column}_clean'].str.split().str.len()

    print("Text preprocessing complete!")
    return df
```

Sentiment Analysis

python

```python
def analyze_sentiment_textblob(text_series):
    """Analyze sentiment using TextBlob."""

    def get_sentiment(text):
        try:
            blob = TextBlob(str(text))
            return {
                'polarity': blob.sentiment.polarity,     # -1 (negative) to 1 (positive)
                'subjectivity': blob.sentiment.subjectivity  # 0 (objective) to 1 (subjective)
            }
        except:
            return {'polarity': 0, 'subjectivity': 0}

    # Apply sentiment analysis
    sentiment_data = text_series.apply(get_sentiment)

    # Extract polarity and subjectivity
    polarity = [s['polarity'] for s in sentiment_data]
    subjectivity = [s['subjectivity'] for s in sentiment_data]

    # Categorize sentiment
    def categorize_sentiment(polarity):
        if polarity > 0.1:
            return 'Positive'
        elif polarity < -0.1:
            return 'Negative'
        else:
            return 'Neutral'

    categories = [categorize_sentiment(p) for p in polarity]

    return pd.DataFrame({
        'polarity': polarity,
        'subjectivity': subjectivity,
        'sentiment_category': categories
    })

def visualize_sentiment(sentiment_df):
    """Create visualizations for sentiment analysis."""

    fig, axes = plt.subplots(2, 2, figsize=(15, 12))

    # Sentiment distribution
    sentiment_counts = sentiment_df['sentiment_category'].value_counts()
    axes[0,0].pie(sentiment_counts.values, labels=sentiment_counts.index, autopct='%1.1f%%')
    axes[0,0].set_title('Sentiment Distribution')

    # Polarity histogram
    axes[0,1].hist(sentiment_df['polarity'], bins=20, edgecolor='black')
    axes[0,1].set_xlabel('Polarity Score')
    axes[0,1].set_ylabel('Frequency')
    axes[0,1].set_title('Polarity Distribution')
    axes[0,1].axvline(x=0, color='red', linestyle='--', label='Neutral')
    axes[0,1].legend()

    # Polarity vs Subjectivity scatter plot
    colors = {'Positive': 'green', 'Negative': 'red', 'Neutral': 'gray'}
    for sentiment in sentiment_df['sentiment_category'].unique():
        mask = sentiment_df['sentiment_category'] == sentiment
        axes[1,0].scatter(
            sentiment_df[mask]['polarity'],
            sentiment_df[mask]['subjectivity'],
            c=colors[sentiment],
            label=sentiment,
            alpha=0.6
        )
    axes[1,0].set_xlabel('Polarity')
    axes[1,0].set_ylabel('Subjectivity')
    axes[1,0].set_title('Polarity vs Subjectivity')
    axes[1,0].legend()

    # Box plot of polarity by sentiment category
    sentiment_df.boxplot(column='polarity', by='sentiment_category', ax=axes[1,1])
    axes[1,1].set_title('Polarity by Sentiment Category')
    axes[1,1].set_xlabel('Sentiment Category')
    axes[1,1].set_ylabel('Polarity Score')

    plt.tight_layout()
    plt.show()

# Example usage
sample_texts = [
    "I absolutely love this new policy! It's amazing!",
    "This is the worst decision ever made.",
    "The weather is okay today, nothing special.",
    "Fantastic work by the team, very impressive results.",
    "I'm not sure how I feel about this change."
]

sample_df = pd.DataFrame({'text': sample_texts})
sentiment_results = analyze_sentiment_textblob(sample_df['text'])

# Combine with original data
sample_df = pd.concat([sample_df, sentiment_results], axis=1)
print(sample_df)
```

Word Frequency Analysis

python

```python
def analyze_word_frequency(text_series, top_n=20):
    """Analyze word frequency in text data."""

    # Combine all text
    all_text = ' '.join(text_series.astype(str))

    # Clean and tokenize
    all_text = clean_text(all_text)
    all_text = remove_stopwords(all_text)

    # Count words
    words = all_text.split()
    word_freq = Counter(words)

    # Get top N words
    top_words = word_freq.most_common(top_n)

    # Create DataFrame
    freq_df = pd.DataFrame(top_words, columns=['word', 'frequency'])

    # Visualize
    plt.figure(figsize=(12, 8))
    plt.subplot(2, 1, 1)
    plt.barh(range(len(freq_df)), freq_df['frequency'])
    plt.yticks(range(len(freq_df)), freq_df['word'])
    plt.xlabel('Frequency')
    plt.title(f'Top {top_n} Most Frequent Words')
    plt.gca().invert_yaxis()

    # Word cloud
    plt.subplot(2, 1, 2)
    wordcloud = WordCloud(width=800, height=400, background_color='white').generate(all_text)
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis('off')
    plt.title('Word Cloud')

    plt.tight_layout()
    plt.show()

    return freq_df
```

```
# Example usage
# word_freq = analyze_word_frequency(sample_df['text'])
```

Topic Modeling

```
python
```

```
def perform_topic_modeling(text_series, n_topics=5, max_features=100):
    """Perform topic modeling using Latent Dirichlet Allocation."""

    # Preprocess text
    clean_texts = [clean_text(text) for text in text_series]
    clean_texts = [remove_stopwords(text) for text in clean_texts]

    # Remove empty texts
    clean_texts = [text for text in clean_texts if text.strip()]

    if len(clean_texts) == 0:
        print("No valid texts found after cleaning!")
        return None, None

    # Vectorize text
    vectorizer = CountVectorizer(max_features=max_features, max_df=0.95, min_df=2)
    text_matrix = vectorizer.fit_transform(clean_texts)

    # Perform LDA
    lda = LatentDirichletAllocation(n_components=n_topics, random_state=42)
    lda.fit(text_matrix)

    # Get feature names
    feature_names = vectorizer.get_feature_names_out()

    # Display topics
    print(f"Discovered {n_topics} topics:")
    print("=" * 50)

    topics = {}
    for topic_idx, topic in enumerate(lda.components_):
        top_words_idx = topic.argsort()[-10:][::-1]  # Top 10 words
        top_words = [feature_names[i] for i in top_words_idx]
        top_scores = [topic[i] for i in top_words_idx]

        topics[f"Topic {topic_idx + 1}"] = {
            'words': top_words,
            'scores': top_scores
        }

        print(f"Topic {topic_idx + 1}: {', '.join(top_words[:5])}")

    return lda, topics
```

```
# Example with more text data
extended_texts = [
    "The economic policy has significant implications for unemployment rates.",
    "Healthcare reform is needed to address rising costs and accessibility issues.",
    "Education funding cuts will impact student performance and teacher retention.",
    "Climate change policies must balance environmental protection with economic growth.",
    "Social media platforms influence political discourse and public opinion formation.",
    "Income inequality continues to widen across different demographic groups.",
    "Technology adoption in healthcare improves patient outcomes and efficiency.",
    "Urban planning decisions affect community development and quality of life.",
    "Immigration policies have complex effects on labor markets and cultural integration.",
    "Criminal justice reform aims to reduce recidivism and improve rehabilitation."
]

extended_df = pd.DataFrame({'text': extended_texts})
# lda_model, topics = perform_topic_modeling(extended_df['text'], n_topics=3)
```

Text Classification

```
python
```

```python
def classify_text_by_keywords(text_series, keyword_categories):
    """
    Classify texts based on keyword categories.

    Parameters:
    text_series: pandas Series with text data
    keyword_categories: dict with category names as keys and lists of keywords as values
    """

    def classify_text(text):
        text_lower = str(text).lower()
        scores = {}

        for category, keywords in keyword_categories.items():
            score = sum(1 for keyword in keywords if keyword.lower() in text_lower)
            scores[category] = score

        # Return category with highest score, or 'Other' if no matches
        if max(scores.values()) > 0:
            return max(scores, key=scores.get)
        else:
            return 'Other'

    classifications = text_series.apply(classify_text)

    # Create summary
    classification_counts = classifications.value_counts()
    print("Text Classification Results:")
    print("=" * 30)
    for category, count in classification_counts.items():
        percentage = (count / len(classifications)) * 100
        print(f"{category}: {count} ({percentage:.1f}%)")

    return classifications

# Example keyword-based classification
policy_categories = {
    'Economic': ['economy', 'economic', 'unemployment', 'income', 'financial', 'budget'],
    'Healthcare': ['health', 'healthcare', 'medical', 'hospital', 'patient', 'treatment'],
    'Education': ['education', 'school', 'student', 'teacher', 'learning', 'university'],
    'Environment': ['climate', 'environment', 'green', 'pollution', 'sustainable', 'energy'],
    'Technology': ['technology', 'digital', 'internet', 'social media', 'innovation', 'tech']
}
```

```
# Classify the extended texts
# classifications = classify_text_by_keywords(extended_df['text'], policy_categories)
# extended_df['category'] = classifications
```

Advanced Text Analytics

```python
```

```
def extract_named_entities(text_series):
    """Extract named entities from text (requires spacy)."""
    try:
        import spacy
        # You would need to install: python -m spacy download en_core_web_sm
        nlp = spacy.load("en_core_web_sm")
    except:
        print("Named entity recognition requires spacy. Install with: pip install spacy")
        print("Then download model: python -m spacy download en_core_web_sm")
        return None

    entities_data = []

    for idx, text in enumerate(text_series):
        doc = nlp(str(text))

        for ent in doc.ents:
            entities_data.append({
                'text_id': idx,
                'entity': ent.text,
                'label': ent.label_,
                'description': spacy.explain(ent.label_)
            })

    entities_df = pd.DataFrame(entities_data)

    if not entities_df.empty:
        print("Most common entity types:")
        print(entities_df['label'].value_counts().head(10))

    return entities_df

def analyze_text_readability(text_series):
    """Analyze text readability using basic metrics."""

    def calculate_readability(text):
        if pd.isna(text):
            return {'sentences': 0, 'words': 0, 'avg_words_per_sentence': 0}

        text = str(text)

        # Count sentences (rough approximation)
        sentences = len([s for s in text.split('.') if s.strip()])
```

```python
        if sentences == 0:
            sentences = 1

        # Count words
        words = len(text.split())

        # Average words per sentence
        avg_words_per_sentence = words / sentences if sentences > 0 else 0

        return {
            'sentences': sentences,
            'words': words,
            'avg_words_per_sentence': avg_words_per_sentence
        }

    readability_data = text_series.apply(calculate_readability)

    # Convert to DataFrame
    readability_df = pd.DataFrame(list(readability_data))

    # Summary statistics
    print("Text Readability Analysis:")
    print("=" * 30)
    print(f"Average words per text: {readability_df['words'].mean():.1f}")
    print(f"Average sentences per text: {readability_df['sentences'].mean():.1f}")
    print(f"Average words per sentence: {readability_df['avg_words_per_sentence'].mean():.1f}")

    return readability_df

# Example comprehensive text analysis
def comprehensive_text_analysis(df, text_column):
    """Perform comprehensive text analysis."""

    print(f"Comprehensive Text Analysis for '{text_column}' column")
    print("=" * 60)

    # 1. Basic preprocessing
    df = preprocess_text_column(df, text_column)

    # 2. Sentiment analysis
    print("\n2. Sentiment Analysis:")
    sentiment_results = analyze_sentiment_textblob(df[text_column])
    df = pd.concat([df, sentiment_results], axis=1)
    # 3. Word frequency
    print("\n3. Word Frequency Analysis:")
    word_freq = analyze_word_frequency(df[f'{text_column}_clean'], top_n=15)

    # 4. Readability analysis
    print("\n4. Readability Analysis:")
    readability = analyze_text_readability(df[text_column])
    df = pd.concat([df, readability], axis=1)

    # 5. Create summary report
    print("\n5. Summary Report:")
    print(f"Total texts analyzed: {len(df)}")
    print(f"Average text length: {df[f'{text_column}_length'].mean():.1f} characters")
    print(f"Average word count: {df[f'{text_column}_word_count'].mean():.1f} words")
    print(f"Sentiment distribution:")
    print(df['sentiment_category'].value_counts())

    return df

# Example usage
# comprehensive_results = comprehensive_text_analysis(extended_df, 'text')
```

Assignment 11: Build a Shiny Application

Learning Objectives

- Create interactive web applications with Streamlit (Python's equivalent to R Shiny)
- Design user interfaces for data exploration
- Deploy interactive dashboards
- Integrate multiple data analysis components

Introduction to Streamlit

```
python
```

```
# app.py - Basic Streamlit Application
import streamlit as st
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from datetime import datetime, date

# App configuration
st.set_page_config(
    page_title="Social Science Data Dashboard",
    page_icon="📊 ",
    layout="wide",
    initial_sidebar_state="expanded"
)

def load_sample_data():
    """Create sample social science dataset."""
    np.random.seed(42)
    n = 1000

    data = {
        'id': range(1, n+1),
        'age': np.random.normal(35, 12, n).astype(int),
        'income': np.random.lognormal(10.5, 0.5, n).astype(int),
        'education': np.random.choice(['High School', 'Bachelor', 'Master', 'PhD'], n,
                                   p=[0.3, 0.4, 0.2, 0.1]),
        'satisfaction': np.random.normal(7, 1.5, n),
        'region': np.random.choice(['North', 'South', 'East', 'West'], n),
        'employment_status': np.random.choice(['Employed', 'Unemployed', 'Retired'], n,
                                           p=[0.7, 0.1, 0.2]),
        'survey_date': pd.date_range(start='2023-01-01', end='2023-12-31', periods=n)
    }

    df = pd.DataFrame(data)

    # Add some correlations
    df.loc[df['education'] == 'PhD', 'income'] *= 1.5
    df.loc[df['education'] == 'Master', 'income'] *= 1.3
    df.loc[df['age'] > 65, 'employment_status'] = 'Retired'

    # Clean up data
```

```python
    df['age'] = df['age'].clip(18, 80)
    df['satisfaction'] = df['satisfaction'].clip(1, 10)
    df['income'] = df['income'].clip(20000, 200000)

    return df

def main():
    """Main application function."""

    # Title and description
    st.title("📊 Social Science Data Dashboard")
    st.markdown("---")
    st.markdown("Interactive dashboard for exploring social science survey data")

    # Sidebar for controls
    st.sidebar.header("Dashboard Controls")

    # Load data
    df = load_sample_data()

    # Data filtering controls
    st.sidebar.subheader("Data Filters")

    # Age filter
    age_range = st.sidebar.slider(
        "Age Range",
        min_value=int(df['age'].min()),
        max_value=int(df['age'].max()),
        value=(int(df['age'].min()), int(df['age'].max()))
    )

    # Education filter
    education_options = st.sidebar.multiselect(
        "Education Level",
        options=df['education'].unique(),
        default=df['education'].unique()
    )

    # Region filter
    region_options = st.sidebar.multiselect(
        "Region",
        options=df['region'].unique(),
        default=df['region'].unique()
    )
    # Apply filters
    filtered_df = df[
        (df['age'] >= age_range[0]) &
        (df['age'] <= age_range[1]) &
        (df['education'].isin(education_options)) &
        (df['region'].isin(region_options))
    ]

    # Main dashboard tabs
    tab1, tab2, tab3, tab4 = st.tabs(["📈 Overview", "🔍 Detailed Analysis", "🗺️ Geographic View", "📊 Custom Ana

    with tab1:
        display_overview(filtered_df)

    with tab2:
        display_detailed_analysis(filtered_df)

    with tab3:
        display_geographic_view(filtered_df)

    with tab4:
        display_custom_analysis(filtered_df)

def display_overview(df):
    """Display overview dashboard."""

    st.header("Data Overview")

    # Key metrics
    col1, col2, col3, col4 = st.columns(4)

    with col1:
        st.metric("Total Responses", len(df))

    with col2:
        avg_age = df['age'].mean()
        st.metric("Average Age", f"{avg_age:.1f} years")

    with col3:
        avg_income = df['income'].mean()
        st.metric("Average Income", f"${avg_income:,.0f}")

    with col4:
        avg_satisfaction = df['satisfaction'].mean()
        st.metric("Average Satisfaction", f"{avg_satisfaction:.1f}/10")

    st.markdown("---")

    # Charts
    col1, col2 = st.columns(2)

    with col1:
        st.subheader("Age Distribution")
        fig, ax = plt.subplots(figsize=(8, 6))
        ax.hist(df['age'], bins=20, edgecolor='black', alpha=0.7)
        ax.set_xlabel('Age')
        ax.set_ylabel('Frequency')
        st.pyplot(fig)

    with col2:
        st.subheader("Education Distribution")
        education_counts = df['education'].value_counts()
        fig, ax = plt.subplots(figsize=(8, 6))
        ax.pie(education_counts.values, labels=education_counts.index, autopct='%1.1f%%')
        st.pyplot(fig)

    # Income vs Satisfaction scatter plot
    st.subheader("Income vs Satisfaction")
    fig = px.scatter(df, x='income', y='satisfaction', color='education',
                    title='Income vs Satisfaction by Education Level')
    st.plotly_chart(fig, use_container_width=True)

def display_detailed_analysis(df):
    """Display detailed analysis."""

    st.header("Detailed Analysis")

    # Analysis type selection
    analysis_type = st.selectbox(
        "Select Analysis Type",
        ["Correlation Analysis", "Group Comparison", "Time Series Analysis"]
    )

    if analysis_type == "Correlation Analysis":
        st.subheader("Correlation Analysis")

        # Calculate correlations
        numeric_cols = ['age', 'income', 'satisfaction']
        corr_matrix = df[numeric_cols].corr()

        # Display correlation heatmap
        fig, ax = plt.subplots(figsize=(8, 6))
        sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', center=0, ax=ax)
        st.pyplot(fig)

        # Display correlation table
        st.subheader("Correlation Coefficients")
        st.dataframe(corr_matrix)

    elif analysis_type == "Group Comparison":
        st.subheader("Group Comparison")

        # Group by selection
        group_by = st.selectbox("Group by:", ['education', 'region', 'employment_status'])
        metric = st.selectbox("Metric:", ['income', 'satisfaction', 'age'])

        # Create box plot
        fig = px.box(df, x=group_by, y=metric, title=f'{metric.title()} by {group_by.title()}')
        st.plotly_chart(fig, use_container_width=True)

        # Summary statistics by group
        group_stats = df.groupby(group_by)[metric].agg(['mean', 'median', 'std', 'count']).round(2)
        st.subheader(f"{metric.title()} Statistics by {group_by.title()}")
        st.dataframe(group_stats)

    elif analysis_type == "Time Series Analysis":
        st.subheader("Time Series Analysis")

        # Group data by month
        df['month'] = df['survey_date'].dt.to_period('M')
        monthly_data = df.groupby('month').agg({
            'satisfaction': 'mean',
            'income': 'mean',
            'age': 'mean'
        }).reset_index()
        monthly_data['month'] = monthly_data['month'].astype(str)

        # Time series plot
        metric_ts = st.selectbox("Select metric for time series:", ['satisfaction', 'income', 'age'])

        fig = px.line(monthly_data, x='month', y=metric_ts,
                    title=f'Monthly Average {metric_ts.title()}')
        fig.update_xaxes(tickangle=45)
        st.plotly_chart(fig, use_container_width=True)

def display_geographic_view(df):
    """Display geographic analysis."""

    st.header("Geographic Analysis")

    # Regional summary
    regional_summary = df.groupby('region').agg({
        'income': ['mean', 'median'],
        'satisfaction': 'mean',
        'age': 'mean',
        'id': 'count'
    }).round(2)

    regional_summary.columns = ['Avg Income', 'Median Income', 'Avg Satisfaction', 'Avg Age', 'Count']

    st.subheader("Regional Summary Statistics")
    st.dataframe(regional_summary)

    # Regional comparison charts
    col1, col2 = st.columns(2)

    with col1:
        fig = px.bar(df.groupby('region')['income'].mean().reset_index(),
                    x='region', y='income', title='Average Income by Region')
        st.plotly_chart(fig, use_container_width=True)

    with col2:
        fig = px.bar(df.groupby('region')['satisfaction'].mean().reset_index(),
                    x='region', y='satisfaction', title='Average Satisfaction by Region')
        st.plotly_chart(fig, use_container_width=True)

def display_custom_analysis(df):
    """Display custom analysis tools."""

    st.header("Custom Analysis")

    st.markdown("Create your own visualizations and analyses.")

    # Variable selection
    col1, col2 = st.columns(2)
```

```
with col1:
    x_var = st.selectbox("X-axis variable:",
                         ['age', 'income', 'satisfaction', 'education', 'region', 'employment_status'])

with col2:
    y_var = st.selectbox("Y-axis variable:",
                         ['satisfaction', 'income', 'age', 'education', 'region', 'employment_status'])

# Chart type selection
chart_type = st.selectbox("Chart type:", ['Scatter Plot', 'Box Plot', 'Bar Chart', 'Histogram'])

# Color coding option
color_var = st.selectbox("Color by (optional):",
                         [None, 'education', 'region', 'employment_status'])

# Generate visualization
if st.button("Generate Visualization"):
    if chart_type == "Scatter Plot":
        fig = px.scatter(df, x=x_var, y=y_var, color=color_var,
                         title=f'{y_var.title()} vs {x_var.title()}')
    elif chart_type == "Box Plot":
        fig = px.box(df, x=x_var, y=y_var, color=color_var,
                     title=f'{y_var.title()} by {x_var.title()}')
    elif chart_type == "Bar Chart":
        if color_var:
            grouped_data = df.groupby([x_var, color_var])[y_var].mean().reset_index()
            fig = px.bar(grouped_data, x=x_var, y=y_var, color=color_var,
                         title=f'Average {y_var.title()} by {x_var.title()}')
        else:
            grouped_data = df.groupby(x_var)[y_var].mean().reset_index()
            fig = px.bar(grouped_data, x=x_var, y=y_var,
                         title=f'Average {y_var.title()} by {x_var.title()}')
    else:  # Histogram
        fig = px.histogram(df, x=x_var, color=color_var,
                           title=f'Distribution of {x_var.title()}')

    st.plotly_chart(fig, use_container_width=True)

# Data download
st.subheader("Download Filtered Data")

if st.button("Prepare Data for Download"):
    csv = df.to_csv(index=False)
```

```python
        st.download_button(
            label="Download CSV",
            data=csv,
            file_name=f'social_science_data_{datetime.now().strftime("%Y%m%d")}.csv',
            mime='text/csv'
        )

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

Running the Application

```bash
# To run the Streamlit app, save the code above as 'app.py' and run:
# streamlit run app.py
```

Advanced Streamlit Features

```python
```

```
# advanced_features.py - Additional Streamlit components

import streamlit as st
import pandas as pd
import numpy as np
from datetime import datetime

def advanced_input_components():
    """Demonstrate advanced input components."""

    st.header("Advanced Input Components")

    # File uploader
    uploaded_file = st.file_uploader("Upload your dataset", type=['csv', 'xlsx'])

    if uploaded_file is not None:
        if uploaded_file.name.endswith('.csv'):
            df = pd.read_csv(uploaded_file)
        else:
            df = pd.read_excel(uploaded_file)

        st.success(f"Loaded {len(df)} rows and {len(df.columns)} columns")
        st.dataframe(df.head())

    # Date inputs
    start_date = st.date_input("Start date", datetime(2023, 1, 1))
    end_date = st.date_input("End date", datetime(2023, 12, 31))

    # Number inputs
    sample_size = st.number_input("Sample size", min_value=10, max_value=10000, value=100)

    # Text input
    analysis_notes = st.text_area("Analysis notes", "Enter your observations here...")

    # Checkbox and radio buttons
    include_outliers = st.checkbox("Include outliers in analysis")
    analysis_method = st.radio("Analysis method", ["Descriptive", "Inferential", "Predictive"])

    return {
        'uploaded_file': uploaded_file,
        'date_range': (start_date, end_date),
        'sample_size': sample_size,
        'notes': analysis_notes,
```

```python
        'include_outliers': include_outliers,
        'method': analysis_method
    }

def create_dashboard_with_state():
    """Create dashboard with session state management."""

    # Initialize session state
    if 'analysis_results' not in st.session_state:
        st.session_state.analysis_results = None

    if 'current_dataset' not in st.session_state:
        st.session_state.current_dataset = None

    st.header("Stateful Dashboard")

    # Load or generate data
    if st.button("Generate New Dataset"):
        # Generate sample data
        np.random.seed()  # Use random seed
        n = st.sidebar.number_input("Dataset size", 100, 5000, 1000)

        df = pd.DataFrame({
            'x': np.random.randn(n),
            'y': np.random.randn(n),
            'category': np.random.choice(['A', 'B', 'C'], n),
            'value': np.random.exponential(2, n)
        })

        st.session_state.current_dataset = df
        st.success("New dataset generated!")

    # Display current dataset
    if st.session_state.current_dataset is not None:
        st.subheader("Current Dataset")
        st.dataframe(st.session_state.current_dataset.head())

        # Run analysis
        if st.button("Run Analysis"):
            df = st.session_state.current_dataset

            results = {
                'mean_x': df['x'].mean(),
                'mean_y': df['y'].mean(),
```

~~~
                'correlation': df['x'].corr(df['y']),
                'category_counts': df['category'].value_counts().to_dict()
            }

            st.session_state.analysis_results = results
            st.success("Analysis complete!")

    # Display results
    if st.session_state.analysis_results is not None:
        st.subheader("Analysis Results")
        results = st.session_state.analysis_results

        col1, col2, col3 = st.columns(3)
        with col1:
            st.metric("Mean X", f"{results['mean_x']:.3f}")
        with col2:
            st.metric("Mean Y", f"{results['mean_y']:.3f}")
        with col3:
            st.metric("Correlation", f"{results['correlation']:.3f}")

        st.write("Category Distribution:", results['category_counts'])

def deployment_guide():
    """Guide for deploying Streamlit applications."""

    st.header("Deployment Guide")

    st.markdown("""
    ### Deploying Your Streamlit App

    #### 1. **Streamlit Cloud (Recommended for beginners)**
    - Push your code to GitHub
    - Connect your GitHub repo to Streamlit Cloud
    - Deploy with one click
    - Free for public repositories

    #### 2. **Heroku**
```bash
    # Create requirements.txt
    pip freeze > requirements.txt

    # Create setup.sh
    mkdir -p ~/.streamlit/
    echo "[server]\\n\\
~~~

```
    port = $PORT\\n\\
    enableCORS = false\\n\\
    headless = true\\n\\
    \\n\\
    " > ~/.streamlit/config.toml

    # Create Procfile
    echo "web: sh setup.sh && streamlit run app.py" > Procfile
```

```
#### 3. **Local Development Best Practices**
- Use virtual environments
- Include requirements.txt
- Document your code
- Test with different data files

#### 4. **File Structure**
```

```
    my_streamlit_app/
    ├── app.py              # Main application
    ├── requirements.txt    # Dependencies
    ├── data/               # Sample data files
    │   └── sample_data.csv
    ├── utils/              # Helper functions
    │   ├── __init__.py
    │   ├── data_processing.py
    │   └── visualizations.py
    └── README.md           # Documentation
```

```
""")

# Sample requirements.txt
with st.expander("Sample requirements.txt"):
    st.code("""
```

streamlit>=1.25.0

pandas>=1.5.0

numpy>=1.21.0

matplotlib>=3.5.0

seaborn>=0.11.0

plotly>=5.0.0

scikit-learn>=1.1.0
""")

Additional utility functions for the app

def create_report_generator():
"""Create automated report generation feature."""

```
st.header("Report Generator")

report_type = st.selectbox("Report Type", ["Executive Summary", "Detailed Analysis", "Technical Report"])

if st.button("Generate Report"):
    if report_type == "Executive Summary":
        report = generate_executive_summary()
    elif report_type == "Detailed Analysis":
        report = generate_detailed_report()
    else:
        report = generate_technical_report()

    st.markdown(report)

    # Download button
    st.download_button(
        label="Download Report",
        data=report,
        file_name=f"{report_type.lower().replace(' ', '_')}_{datetime.now().strftime('%Y%m%d')}.md",
        mime="text/markdown"
    )
```

def generate_executive_summary():
"""Generate executive summary report."""

```
return """
```

Executive Summary

Key Findings

- Sample size: 1,000 respondents
- Average satisfaction score: 7.2/10

- Primary demographic: Adults aged 25-45
- Regional variations observed

Recommendations

1. Focus improvement efforts on lowest-scoring regions
2. Investigate factors driving high satisfaction in top-performing areas
3. Consider targeted interventions for specific demographic groups

Next Steps

- Conduct follow-up interviews with select respondents
- Implement recommended changes
- Schedule quarterly review meetings """

def generate_detailed_report():
"""Generate detailed analysis report."""

```
return """
```

Detailed Analysis Report

Methodology

This analysis was conducted using survey data collected from...

Data Overview

- **Sample Size**: 1,000 respondents
- **Collection Period**: January - December 2023
- **Response Rate**: 78%

Key Variables

1. **Demographics**: Age, education, region
2. **Outcomes**: Income, satisfaction scores
3. **Behavioral**: Employment status, survey responses

Statistical Results

Correlation Analysis

- Strong positive correlation between education and income (r=0.65)
- Moderate correlation between age and satisfaction (r=0.34)

Regional Differences

- Significant variation across regions (F=12.3, p<0.001)
- Post-hoc tests reveal North vs South difference

Limitations

- Self-reported data subject to bias
- Cross-sectional design limits causal inference
- Regional sampling may not be representative """

def generate_technical_report():
"""Generate technical report."""

```
return """
```

Technical Analysis Report

Data Processing Pipeline

1. **Data Collection**: Survey responses via online platform
2. **Data Cleaning**: Missing value imputation, outlier detection
3. **Feature Engineering**: Derived variables, categorization
4. **Analysis**: Descriptive statistics, hypothesis testing, modeling

Technical Specifications

- **Platform**: Python 3.9+ with pandas, scikit-learn
- **Statistical Methods**: ANOVA, correlation analysis, regression
- **Visualization**: matplotlib, seaborn, plotly
- **Deployment**: Streamlit dashboard

Code Repository

Available at: github.com/username/social-science-analysis

Reproducibility

- Random seed: 42
- Package versions documented in requirements.txt
- Analysis pipeline automated via scripts

Future Enhancements

- Real-time data integration
- Advanced ML models
- Interactive parameter tuning """

Final Project Integration

Bringing It All Together

```python
```

```
# final_project_template.py - Template for final project

import streamlit as st
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
from datetime import datetime
import requests

# Import custom modules (create these based on previous assignments)
# from utils.data_wrangling import clean_data, prepare_features
# from utils.machine_learning import run_ml_analysis
# from utils.text_analysis import analyze_text_data
# from utils.geo_analysis import create_maps

class SocialScienceAnalysisPlatform:
    """Complete platform for social science data analysis."""

    def __init__(self):
        self.data = None
        self.analysis_results = {}

    def load_data(self, source_type, **kwargs):
        """Load data from various sources."""
        if source_type == 'file':
            return self.load_from_file(kwargs['file'])
        elif source_type == 'api':
            return self.load_from_api(kwargs['url'])
        elif source_type == 'web':
            return self.scrape_web_data(kwargs['url'])
        else:
            return self.generate_sample_data()

    def run_complete_analysis(self):
        """Run comprehensive analysis pipeline."""
        if self.data is None:
            st.error("No data loaded!")
            return

        # Step 1: Data exploration and visualization
        self.analysis_results['exploration'] = self.explore_data()
        # Step 2: Data wrangling and cleaning
        self.data_clean = self.wrangle_data()

        # Step 3: Statistical analysis
        self.analysis_results['statistics'] = self.statistical_analysis()

        # Step 4: Machine learning (if applicable)
        if self.has_ml_targets():
            self.analysis_results['ml'] = self.machine_learning_analysis()

        # Step 5: Text analysis (if text columns exist)
        if self.has_text_data():
            self.analysis_results['text'] = self.text_analysis()

        # Step 6: Geospatial analysis (if location data exists)
        if self.has_geo_data():
            self.analysis_results['geo'] = self.geo_analysis()

        return self.analysis_results

    def explore_data(self):
        """Perform exploratory data analysis."""
        st.subheader("📊 Data Exploration")

        # Basic info
        col1, col2, col3 = st.columns(3)
        with col1:
            st.metric("Rows", len(self.data))
        with col2:
            st.metric("Columns", len(self.data.columns))
        with col3:
            st.metric("Missing Values", self.data.isnull().sum().sum())

        # Data preview
        st.write("Data Preview:")
        st.dataframe(self.data.head())

        # Summary statistics
        st.write("Summary Statistics:")
        st.dataframe(self.data.describe())

        return {"status": "completed", "timestamp": datetime.now()}
```

```python
    def wrangle_data(self):
        """Clean and prepare data."""
        st.subheader("🔧 Data Wrangling")

        data_clean = self.data.copy()

        # Handle missing values
        missing_strategy = st.selectbox("Missing value strategy:",
                                        ["Drop rows", "Fill with mean", "Fill with median", "Forward fill"])

        if missing_strategy == "Drop rows":
            data_clean = data_clean.dropna()
        elif missing_strategy == "Fill with mean":
            numeric_cols = data_clean.select_dtypes(include=[np.number]).columns
            data_clean[numeric_cols] = data_clean[numeric_cols].fillna(data_clean[numeric_cols].mean())
        # Add other strategies...

        st.success(f"Data cleaned: {len(data_clean)} rows remaining")

        return data_clean

    def has_ml_targets(self):
        """Check if dataset has suitable ML targets."""
        # Simple heuristic: look for numeric columns that could be targets
        numeric_cols = self.data.select_dtypes(include=[np.number]).columns
        return len(numeric_cols) > 1

    def has_text_data(self):
        """Check if dataset has text data."""
        text_cols = self.data.select_dtypes(include=['object']).columns
        return len(text_cols) > 0

    def has_geo_data(self):
        """Check if dataset has geographic data."""
        geo_indicators = ['lat', 'lon', 'latitude', 'longitude', 'address', 'zip', 'country']
        return any(indicator in str(col).lower() for col in self.data.columns for indicator in geo_indicators)

def main_application():
    """Main Streamlit application."""

    st.set_page_config(page_title="Social Science Analysis Platform", layout="wide")

    st.title("🎓 Social Science Analysis Platform")
    st.markdown("Complete toolkit for social science research and data analysis")
    # Initialize platform
    if 'platform' not in st.session_state:
        st.session_state.platform = SocialScienceAnalysisPlatform()

    platform = st.session_state.platform

    # Sidebar navigation
    st.sidebar.title("Navigation")
    page = st.sidebar.selectbox("Choose analysis module:", [
        "📁 Data Loading",
        "🔍 Data Exploration",
        "🧹 Data Wrangling",
        "🔬 Statistical Analysis",
        "🤖 Machine Learning",
        "📝 Text Analysis",
        "🗺️ Geospatial Analysis",
        "📊 Dashboard",
        "📋 Report Generation"
    ])

    # Page routing
    if page == "📁 Data Loading":
        data_loading_page(platform)
    elif page == "🔍 Data Exploration":
        data_exploration_page(platform)
    elif page == "🧹 Data Wrangling":
        data_wrangling_page(platform)
    elif page == "🔬 Statistical Analysis":
        statistical_analysis_page(platform)
    elif page == "🤖 Machine Learning":
        machine_learning_page(platform)
    elif page == "📝 Text Analysis":
        text_analysis_page(platform)
    elif page == "🗺️ Geospatial Analysis":
        geospatial_analysis_page(platform)
    elif page == "📊 Dashboard":
        dashboard_page(platform)
    else:
        report_generation_page(platform)

def data_loading_page(platform):
    """Data loading interface."""
    st.header("📁 Data Loading")

    data_source = st.selectbox("Select data source:", [
        "Upload File",
        "Load Sample Data",
        "Connect to API",
        "Web Scraping"
    ])

    if data_source == "Upload File":
        uploaded_file = st.file_uploader("Choose a file", type=['csv', 'xlsx', 'json'])

        if uploaded_file is not None:
            try:
                if uploaded_file.name.endswith('.csv'):
                    platform.data = pd.read_csv(uploaded_file)
                elif uploaded_file.name.endswith('.xlsx'):
                    platform.data = pd.read_excel(uploaded_file)
                elif uploaded_file.name.endswith('.json'):
                    platform.data = pd.read_json(uploaded_file)

                st.success(f"Loaded {len(platform.data)} rows and {len(platform.data.columns)} columns")
                st.dataframe(platform.data.head())

            except Exception as e:
                st.error(f"Error loading file: {str(e)}")

    elif data_source == "Load Sample Data":
        dataset_type = st.selectbox("Sample dataset:", [
            "Survey Data",
            "Economic Indicators",
            "Social Media Posts",
            "Geographic Data"
        ])

        if st.button("Load Sample Data"):
            platform.data = generate_sample_dataset(dataset_type)
            st.success("Sample data loaded!")
            st.dataframe(platform.data.head())

def generate_sample_dataset(dataset_type):
    """Generate sample datasets for different types."""

    np.random.seed(42)
```

```
n = 1000

if dataset_type == "Survey Data":
    return pd.DataFrame({
        'respondent_id': range(1, n+1),
        'age': np.random.normal(35, 12, n).clip(18, 80).astype(int),
        'income': np.random.lognormal(10.5, 0.5, n).clip(20000, 200000).astype(int),
        'education': np.random.choice(['High School', 'Bachelor', 'Master', 'PhD'], n),
        'satisfaction': np.random.normal(7, 1.5, n).clip(1, 10),
        'region': np.random.choice(['North', 'South', 'East', 'West'], n),
        'survey_date': pd.date_range('2023-01-01', periods=n, freq='1H')
    })

elif dataset_type == "Economic Indicators":
    dates = pd.date_range('2020-01-01', '2023-12-31', freq='M')
    return pd.DataFrame({
        'date': dates,
        'gdp_growth': np.random.normal(2.5, 1.5, len(dates)),
        'unemployment_rate': np.random.normal(5.2, 1.8, len(dates)).clip(0, 15),
        'inflation_rate': np.random.normal(2.1, 1.2, len(dates)),
        'consumer_confidence': np.random.normal(65, 10, len(dates)).clip(0, 100),
        'country': np.random.choice(['USA', 'Canada', 'UK', 'Germany'], len(dates))
    })

elif dataset_type == "Social Media Posts":
    return pd.DataFrame({
        'post_id': range(1, n+1),
        'text': [f"This is sample social media post number {i}" for i in range(1, n+1)],
        'likes': np.random.poisson(50, n),
        'shares': np.random.poisson(10, n),
        'timestamp': pd.date_range('2023-01-01', periods=n, freq='1H'),
        'platform': np.random.choice(['Twitter', 'Facebook', 'Instagram'], n),
        'sentiment': np.random.choice(['Positive', 'Negative', 'Neutral'], n)
    })

else:  # Geographic Data
    return pd.DataFrame({
        'location_id': range(1, n+1),
        'latitude': np.random.uniform(25, 49, n),  # US bounds
        'longitude': np.random.uniform(-125, -66, n),
        'population': np.random.lognormal(10, 1, n).astype(int),
        'median_income': np.random.normal(55000, 15000, n).clip(20000, 150000).astype(int),
        'city': [f"City_{i}" for i in range(1, n+1)],
        'state': np.random.choice(['CA', 'TX', 'NY', 'FL', 'IL'], n)
```

```python
        })

def dashboard_page(platform):
    """Interactive dashboard page."""

    st.header("📊 Interactive Dashboard")

    if platform.data is None:
        st.warning("Please load data first!")
        return

    # Dashboard configuration
    st.sidebar.subheader("Dashboard Settings")

    # Variable selection
    numeric_cols = platform.data.select_dtypes(include=[np.number]).columns.tolist()
    categorical_cols = platform.data.select_dtypes(include=['object']).columns.tolist()

    if numeric_cols:
        primary_metric = st.sidebar.selectbox("Primary metric:", numeric_cols)
        secondary_metric = st.sidebar.selectbox("Secondary metric:", numeric_cols)

    if categorical_cols:
        group_by = st.sidebar.selectbox("Group by:", [None] + categorical_cols)

    # Main dashboard
    if numeric_cols:
        # Key metrics
        col1, col2, col3, col4 = st.columns(4)

        with col1:
            st.metric("Total Records", len(platform.data))

        with col2:
            st.metric(f"Avg {primary_metric}", f"{platform.data[primary_metric].mean():.2f}")

        with col3:
            st.metric(f"Max {primary_metric}", f"{platform.data[primary_metric].max():.2f}")

        with col4:
            missing_pct = (platform.data[primary_metric].isnull().sum() / len(platform.data)) * 100
            st.metric("Missing %", f"{missing_pct:.1f}%")

        # Visualizations
        col1, col2 = st.columns(2)

        with col1:
            # Distribution plot
            fig, ax = plt.subplots(figsize=(8, 6))
            ax.hist(platform.data[primary_metric].dropna(), bins=30, alpha=0.7)
            ax.set_title(f'Distribution of {primary_metric}')
            ax.set_xlabel(primary_metric)
            ax.set_ylabel('Frequency')
            st.pyplot(fig)

        with col2:
            # Scatter plot if two numeric variables selected
            if len(numeric_cols) > 1:
                fig = px.scatter(platform.data, x=primary_metric, y=secondary_metric,
                               color=group_by if group_by else None,
                               title=f'{primary_metric} vs {secondary_metric}')
                st.plotly_chart(fig, use_container_width=True)

        # Group analysis if categorical variable selected
        if group_by:
            st.subheader(f"Analysis by {group_by}")

            grouped_stats = platform.data.groupby(group_by)[primary_metric].agg(['mean', 'median', 'count']).round(

            st.dataframe(grouped_stats)

            # Group comparison plot
            fig = px.box(platform.data, x=group_by, y=primary_metric,
                        title=f'{primary_metric} by {group_by}')
            st.plotly_chart(fig, use_container_width=True)

def report_generation_page(platform):
    """Report generation interface."""

    st.header("📋 Report Generation")

    if platform.data is None:
        st.warning("Please load and analyze data first!")
        return

    # Report configuration
    report_sections = st.multiselect("Include sections:", [
        "Executive Summary",
        "Data Overview",
        "Statistical Analysis",
        "Key Findings",
        "Visualizations",
        "Methodology",
        "Recommendations"
    ], default=["Executive Summary", "Data Overview", "Key Findings"])

    report_format = st.selectbox("Report format:", ["Markdown", "HTML", "PDF"])

    if st.button("Generate Report"):
        report_content = generate_comprehensive_report(platform.data, report_sections)

        # Display report
        st.markdown(report_content)

        # Download options
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        filename = f"social_science_report_{timestamp}"

        if report_format == "Markdown":
            st.download_button(
                "Download Report (Markdown)",
                report_content,
                f"{filename}.md",
                "text/markdown"
            )
        elif report_format == "HTML":
            html_content = f"<html><body>{report_content}</body></html>"
            st.download_button(
                "Download Report (HTML)",
                html_content,
                f"{filename}.html",
                "text/html"
            )

def generate_comprehensive_report(data, sections):
    """Generate comprehensive analysis report."""

    report = "# Social Science Data Analysis Report\n\n"
    report += f"**Generated on:** {datetime.now().strftime('%B %d, %Y at %I:%M %p')}\n\n"
    report += "---\n\n"

    if "Executive Summary" in sections:
        report += "## Executive Summary\n\n"
        report += f"This report analyzes a dataset containing {len(data)} observations "
        report += f"across {len(data.columns)} variables. "

        numeric_cols = data.select_dtypes(include=[np.number]).columns
        if len(numeric_cols) > 0:
            primary_var = numeric_cols[0]
            mean_val = data[primary_var].mean()
            report += f"The primary metric ({primary_var}) has a mean value of {mean_val:.2f}.\n\n"

        report += "### Key Highlights\n"
        report += f"- Dataset size: {len(data):,} records\n"
        report += f"- Variables analyzed: {len(data.columns)}\n"
        report += f"- Missing data: {data.isnull().sum().sum()} values\n"
        report += f"- Analysis completed: {datetime.now().strftime('%B %Y')}\n\n"

    if "Data Overview" in sections:
        report += "## Data Overview\n\n"
        report += "### Dataset Structure\n"
        report += f"- **Rows:** {len(data):,}\n"
        report += f"- **Columns:** {len(data.columns)}\n"
        report += f"- **Memory Usage:** {data.memory_usage(deep=True).sum() / 1024**2:.1f} MB\n\n"

        report += "### Variable Types\n"
        dtype_counts = data.dtypes.value_counts()
        for dtype, count in dtype_counts.items():
            report += f"- **{dtype}:** {count} variables\n"
        report += "\n"

        report += "### Missing Values\n"
        missing_summary = data.isnull().sum()
        missing_vars = missing_summary[missing_summary > 0]

        if len(missing_vars) > 0:
            for var, missing_count in missing_vars.items():
                pct_missing = (missing_count / len(data)) * 100
                report += f"- **{var}:** {missing_count} ({pct_missing:.1f}%)\n"
        else:
            report += "No missing values detected.\n"
        report += "\n"

    if "Statistical Analysis" in sections:
        report += "## Statistical Analysis\n\n"

        numeric_cols = data.select_dtypes(include=[np.number]).columns
        if len(numeric_cols) > 0:
            report += "### Descriptive Statistics\n\n"

            desc_stats = data[numeric_cols].describe()
            report += desc_stats.round(2).to_markdown()
            report += "\n\n"

            # Correlations
            if len(numeric_cols) > 1:
                report += "### Correlation Analysis\n\n"
                corr_matrix = data[numeric_cols].corr()

                # Find strongest correlations
                corr_pairs = []
                for i in range(len(corr_matrix.columns)):
                    for j in range(i+1, len(corr_matrix.columns)):
                        var1 = corr_matrix.columns[i]
                        var2 = corr_matrix.columns[j]
                        corr_val = corr_matrix.iloc[i, j]
                        corr_pairs.append((abs(corr_val), var1, var2, corr_val))

                corr_pairs.sort(reverse=True)

                report += "**Strongest correlations:**\n"
                for _, var1, var2, corr_val in corr_pairs[:5]:
                    report += f"- {var1} ↔ {var2}: {corr_val:.3f}\n"
                report += "\n"

    if "Key Findings" in sections:
        report += "## Key Findings\n\n"

        findings = []

        # Data quality findings
        missing_pct = (data.isnull().sum().sum() / (len(data) * len(data.columns))) * 100
        if missing_pct > 10:
            findings.append(f"⚠️ High proportion of missing values ({missing_pct:.1f}%) may impact analysis reliability
        elif missing_pct < 1:
            findings.append("✅ Excellent data quality with minimal missing values")

        # Distribution findings
        numeric_cols = data.select_dtypes(include=[np.number]).columns
        for col in numeric_cols[:3]:  # Top 3 numeric columns
            skew = data[col].skew()
            if abs(skew) > 2:
                findings.append(f"📊 {col} shows significant skewness (skew = {skew:.2f})")

        # Categorical findings
        categorical_cols = data.select_dtypes(include=['object']).columns
        for col in categorical_cols[:2]:  # Top 2 categorical columns
            unique_count = data[col].nunique()
            total_count = len(data)
            if unique_count / total_count > 0.5:
                findings.append(f"🏷️ {col} has high cardinality ({unique_count} unique values)")

        for i, finding in enumerate(findings, 1):
            report += f"{i}. {finding}\n"

        if not findings:
            report += "No significant data quality issues identified.\n"
        report += "\n"

    if "Recommendations" in sections:
        report += "## Recommendations\n\n"

        recommendations = [
            "**Data Quality**: Implement data validation checks to prevent missing values in critical variables",
            "**Analysis Depth**: Consider advanced statistical techniques for deeper insights",
            "**Visualization**: Create interactive dashboards for stakeholder engagement",
            "**Documentation**: Maintain detailed metadata for reproducible research",
            "**Validation**: Cross-validate findings with external datasets when possible"
        ]

        for i, rec in enumerate(recommendations, 1):
            report += f"{i}. {rec}\n"
        report += "\n"

    report += "---\n\n"
    report += "*Report generated using Python Social Science Analysis Platform*\n"

    return report

# Final integration note
def integration_notes():
    """Notes on integrating all course components."""

    st.header("🎯 Final Project Integration Guide")
st.markdown("""
### Combining All Course Elements

Your final project should demonstrate mastery of all course topics:

#### 1. **Data Pipeline** (HW02-HW03)
- Load data from multiple sources
- Clean and wrangle complex datasets
- Handle missing values appropriately

#### 2. **Programming Skills** (HW04-HW05)
- Write modular, well-documented functions
- Implement error handling and debugging
- Use version control and reproducible workflows

#### 3. **Advanced Analysis** (HW06-HW10)
- Apply appropriate statistical methods
- Implement machine learning when suitable
- Analyze text and geospatial data as needed

#### 4. **Communication** (HW11)
- Create interactive applications
- Generate automated reports
- Design user-friendly interfaces

### Project Success Criteria

✅ **Technical Competency**
- Clean, well-documented code
- Appropriate statistical methods
- Proper data handling

✅ **Social Science Relevance**
- Address meaningful research questions
- Consider ethical implications
- Interpret results in context

✅ **Communication Excellence**
- Clear visualizations
- Accessible explanations
- Professional presentation

### Getting Started Checklist
```

```
    - [ ] Choose a research question
    - [ ] Identify data sources
    - [ ] Plan analysis approach
    - [ ] Set up development environment
    - [ ] Create project structure
    - [ ] Begin with exploratory analysis
    - [ ] Iterate and refine
    - [ ] Document throughout
    - [ ] Test with real users
    - [ ] Prepare final presentation
    """)

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

Course Summary

Key Learning Outcomes

By completing these assignments, students will have gained:

Technical Skills:

- Python programming fundamentals
- Data manipulation with pandas
- Statistical analysis and visualization
- Machine learning basics
- Text and geospatial analysis
- Web development with Streamlit

Research Skills:

- Reproducible research practices
- Data collection and cleaning
- Statistical inference
- Result interpretation and communication

Professional Skills:

- Project management

- Documentation and version control
- Collaboration and presentation
- Ethical data use considerations

Final Notes for Students

1. **Start Simple**: Each assignment builds on previous knowledge. Master the basics before moving to advanced topics.
2. **Practice Regularly**: The best way to learn programming is by doing. Work through examples and modify them.
3. **Ask Questions**: Don't hesitate to seek help when stuck. Programming is collaborative.
4. **Document Everything**: Good documentation makes your work reproducible and shareable.
5. **Think Like a Social Scientist**: Always consider the broader implications and context of your analysis.
6. **Keep Learning**: These notes provide a foundation. Continue exploring new tools and techniques.

Good luck with your social science computing journey!