Week 2 Lab: Data Validation

CS 203: Software Tools and Techniques for AI

Duration: 3 hours

Prof. Nipun Batra & Teaching Assistants

Lab Overview

Goal: Validate and clean the movie dataset from Week 1

What you'll do:

- Inspect data with command-line tools (jq, csvkit)
- Define validation schemas with Pydantic
- Clean data with pandas
- Build automated validation pipeline

Skills practiced:

- Command-line data analysis
- Schema-based validation
- Data cleaning techniques
- Building reproducible pipelines

Setup Check

Verify your environment:

```
# Install jq (JSON processor)
brew install jq  # Mac
sudo apt-get install jq  # Linux

# Install Python packages
pip install csvkit pydantic pandas

# Verify installations
jq --version
csvstat --version
python -c "import pydantic, pandas; print('Ready!')"
```

Part 1: Command-Line Inspection

Using jq and csvkit to explore data quality

Exercise 1.1: Inspect JSON with jq

Task: Explore your `movies_raw.json` from Week 1

Step 1: Pretty-print the JSON

```
cat movies_raw.json | jq '.' | head -50
```

Step 2: Count total movies

```
cat movies_raw.json | jq 'length'
```

Step 3: Get first movie's title and rating

```
cat movies_raw.json | jq '.[0] | {title: .Title, rating: .imdbRating}'
```

Exercise 1.1: Solution Discussion

What did you find?

- How many movies in your dataset?
- What fields does each movie have?
- Are all fields present in every movie?

Common observations:

- Some movies have `"N/A"` values
- Ratings are strings, not numbers
- Runtime includes "min" suffix

Exercise 1.2: Find Missing Data with jq

Task: Identify data quality issues

Find movies with missing box office:

```
cat movies_raw.json | jq '.[] | select(.BoxOffice == "N/A") | .Title'
```

Count how many:

```
cat movies_raw.json | jq '[.[] | select(.BoxOffice == "N/A")] | length'
```

Find movies with missing Metascore:

```
cat movies_raw.json | jq '.[] | select(.Metascore == "N/A") | .Title'
```

Exercise 1.2: Your Turn

Try these on your own:

1. Find all movies from the 2010s
2. Find movies with rating above 8.5
3. Count movies by genre (first genre only)
4. Calculate average rating

Hint: Use `jq` filters like `select()`, `tonumber`, `contains()`, and aggregation functions

Exercise 1.3: CSV Analysis with csvkit

Task: Analyze your `movies.csv` file

View the data:

```
csvlook movies.csv | head -30
```

Get column names:

```
csvcut -n movies.csv
```

Get summary statistics:

```
csvstat movies.csv
```

Exercise 1.3: Understanding csvstat Output

The output shows for each column:

- **Type of data**: Text, Number, Date
- **Contains null values**: True/False
- **Unique values**: How many distinct values
- **Min, Max, Mean**: For numeric columns
- **Most common values**: Top values

Question: Which columns have the most missing values in your dataset?

Exercise 1.4: Filter and Sort CSV

Get only high-rated movies (rating > 8.5):

```
csvgrep -c rating -r "^[89]\." movies.csv
```

Sort by rating (highest first):

```
csvsort -c rating -r movies.csv | csvlook | head -20
```

Extract specific columns:

```
csvcut -c title,year,rating,genre movies.csv | csvlook
```

Exercise 1.4: Your Challenge

Create a filtered dataset of movies:

- From years 2010-2020
- Rating above 8.0
- Save to new CSV file

Steps:

1. Filter by year range
2. Filter by rating
3. Save output

Hint: Pipe multiple commands together!

Part 1 Checkpoint

What you've learned:

- Using jq to inspect JSON and find issues
- Using csvkit tools to analyze CSV files
- Identifying missing data and quality problems
- Basic filtering and statistics

Data quality issues found:

- Missing values ("N/A")
- String numbers that should be numeric
- Inconsistent formatting

Part 2: Python Validation with Pydantic

Building type-safe data models

Exercise 2.1: Define a Movie Schema

Task: Create a Pydantic model for movies

Create file: `models.py`

```python
from pydantic import BaseModel, Field
from typing import Optional

class Movie(BaseModel):
    Title: str
    Year: str
    imdbRating: str
    Genre: str
    Director: str
    Runtime: Optional[str] = None
    BoxOffice: Optional[str] = None

# Test it
movie_data = {
    "Title": "Inception",
    "Year": "2010",
    "imdbRating": "8.8",
    "Genre": "Action, Sci-Fi",
    "Director": "Christopher Nolan"
}
```

Exercise 2.1: Run It

```
python models.py
```

Expected output:

```
Inception
```

What happened?

- Pydantic validated the data structure
- All required fields present
- Types match expectations

Exercise 2.2: Add Type Conversion

Task: Convert string fields to proper types

Update `models.py`:

```
from pydantic import BaseModel, Field, validator
from typing import Optional

class Movie(BaseModel):
    Title: str
    Year: int  # Changed to int
    imdbRating: float  # Changed to float
    Genre: str
    Director: str
    Runtime: Optional[str] = None
    BoxOffice: Optional[str] = None

    @validator('Year', pre=True)
    def parse_year(cls, v):
        return int(v)

    @validator('imdbRating', pre=True)
```

Exercise 2.2: Test Conversion

```
# Same data, but types converted
movie = Movie(**movie_data)
print(type(movie.Year))       # <class 'int'>
print(type(movie.imdbRating)) # <class 'float'>
print(movie.Year + 5)         # 2015 (math works!)
```

Pydantic automatically converts compatible types!

Exercise 2.3: Add Validation Rules

Task: Add constraints to ensure data quality

```python
class Movie(BaseModel):
    Title: str
    Year: int = Field(ge=1888, le=2030)  # Valid year range
    imdbRating: float = Field(ge=0, le=10)  # Valid rating
    Genre: str
    Director: str
    Runtime: Optional[str] = None
    BoxOffice: Optional[str] = None

    @validator('Title')
    def title_not_empty(cls, v):
        if not v or v.strip() == '':
            raise ValueError('Title cannot be empty')
        return v

    @validator('Genre')
    def genre_not_empty(cls, v):
        if not v or v.strip() == '':
            raise ValueError('Genre cannot be empty')
        return v
```

Exercise 2.3: Test Validation

Try these invalid movies:

```python
# Invalid year
bad_movie = {
    "Title": "Future Movie",
    "Year": "2050",  # Too far in future!
    "imdbRating": "8.0",
    "Genre": "Sci-Fi",
    "Director": "Someone"
}

try:
    Movie(**bad_movie)
except ValidationError as e:
    print(e)
```

What error do you get?

Exercise 2.4: Validate Your Dataset

Task: Validate all movies from `movies_raw.json`

Create file: `validate_movies.py`

```
import json
from pydantic import ValidationError
from models import Movie

# Load data
with open('movies_raw.json') as f:
    movies_data = json.load(f)

# Validate each movie
valid_movies = []
invalid_movies = []

for i, movie_data in enumerate(movies_data):
    try:
        movie = Movie(**movie_data)
        valid_movies.append(movie.dict())
    except ValidationError as e:
        invalid_movies.append({
            'index': i,
            'title': movie_data.get('Title', 'Unknown')
```

Exercise 2.4: Report Results

```python
# Print summary
print(f"Valid movies: {len(valid_movies)}")
print(f"Invalid movies: {len(invalid_movies)}")

# Show first few invalid
if invalid_movies:
    print("\nFirst 3 invalid movies:")
    for inv in invalid_movies[:3]:
        print(f"\n{inv['title']}:")
        print(f"  {inv['errors'][:200]}...")

# Save valid movies
with open('movies_valid.json', 'w') as f:
    json.dump(valid_movies, f, indent=2)
```

Exercise 2.4: Discussion

Questions:

1. How many movies passed validation?
2. What were the most common validation errors?
3. Should we fix the data or adjust the schema?

Common issues:

- Missing required fields
- "N/A" values that can't convert to numbers
- Invalid year values

Part 2 Checkpoint

What you've learned:

- Defining data schemas with Pydantic
- Type conversion and validation
- Custom validators for business rules
- Handling validation errors gracefully

Next: Use pandas to clean the data before validation

Part 3: Data Cleaning with pandas

Fixing issues before validation

Exercise 3.1: Load and Inspect

Task: Load data into pandas DataFrame

Create file: `clean_movies.py`

```
import pandas as pd
import json

# Load JSON
with open('movies_raw.json') as f:
    movies_data = json.load(f)

df = pd.DataFrame(movies_data)

# Inspect
print(df.shape)
print(df.columns.tolist())
print(df.head())
print(df.info())
```

Exercise 3.1: Check Data Quality

```
# Check missing values
print("\nMissing values:")
print(df.isnull().sum())

# Check data types
print("\nData types:")
print(df.dtypes)

# Check for "N/A" strings
na_columns = ['BoxOffice', 'Metascore', 'imdbRating']
for col in na_columns:
    if col in df.columns:
        na_count = (df[col] == 'N/A').sum()
        print(f"{col}: {na_count} 'N/A' values")
```

Exercise 3.2: Handle Missing Values

Task: Replace "N/A" with actual null values

```
# Replace "N/A" strings with None
df = df.replace('N/A', None)

# Check again
print("\nAfter replacing N/A:")
print(df.isnull().sum())
```

Strategy:

- Replace "N/A" with None (pandas recognizes this)
- Decide per column: drop or fill?

Exercise 3.3: Convert Data Types

Task: Convert string columns to proper types

```
# Convert Year to int
df['Year'] = pd.to_numeric(df['Year'], errors='coerce')

# Convert imdbRating to float
df['imdbRating'] = pd.to_numeric(df['imdbRating'], errors='coerce')

# Convert imdbVotes (remove commas first)
if 'imdbVotes' in df.columns:
    df['imdbVotes'] = (
        df['imdbVotes']
        .str.replace(',', '')
        .astype(float)
    )

# Check types
print(df.dtypes)
```

Exercise 3.4: Clean Runtime

Task: Extract numeric runtime from "148 min"

```python
# Extract just the number
if 'Runtime' in df.columns:
    df['runtime_minutes'] = (
        df['Runtime']
        .str.extract(r'(\d+)')[0]
        .astype(float)
    )

# Show results
print(df[['Runtime', 'runtime_minutes']].head())
```

Before: "148 min"

After: 148.0

Exercise 3.5: Clean Box Office

Task: Convert "$292,587,330" to 292587330

```
if 'BoxOffice' in df.columns:
    df['box_office_clean'] = (
        df['BoxOffice']
        .str.replace('$', '', regex=False)
        .str.replace(',', '', regex=False)
        .astype(float)
    )

print(df[['BoxOffice', 'box_office_clean']].head())
```

Exercise 3.6: Remove Duplicates

Task: Check for and remove duplicate movies

```python
# Check for duplicates by title
duplicates = df[df.duplicated(subset=['Title'], keep=False)]
print(f"Found {len(duplicates)} duplicate records")

if len(duplicates) > 0:
    print(duplicates[['Title', 'Year', 'imdbRating']])

# Remove duplicates (keep first occurrence)
df_clean = df.drop_duplicates(subset=['Title'], keep='first')
print(f"After deduplication: {len(df_clean)} movies")
```

Exercise 3.7: Validate Value Ranges

Task: Check for impossible values

```
# Check rating range
invalid_ratings = df_clean[
    (df_clean['imdbRating'] < 0) |
    (df_clean['imdbRating'] > 10)
]
print(f"Invalid ratings: {len(invalid_ratings)}")

# Check year range
invalid_years = df_clean[
    (df_clean['Year'] < 1888) |
    (df_clean['Year'] > 2030)
]
print(f"Invalid years: {len(invalid_years)}")

# Remove invalid rows
df_clean = df_clean[
    (df_clean['imdbRating'] >= 0) &
    (df_clean['imdbRating'] <= 10) &
    (df_clean['Year'] >= 1888) &
    (df_clean['Year'] <= 2030)
]
```

Exercise 3.8: Handle Missing Critical Fields

Task: Decide what to do with missing values

```
# Drop rows missing critical fields
critical_fields = ['Title', 'Year', 'imdbRating', 'Genre']

print(f"Before: {len(df_clean)} movies")

df_clean = df_clean.dropna(subset=critical_fields)

print(f"After: {len(df_clean)} movies")
print(f"Dropped: {len(df) - len(df_clean)} movies")

# For optional fields, keep as None
# BoxOffice, Metascore can be missing
```

Exercise 3.9: Save Cleaned Data

Task: Export cleaned dataset

```python
# Save to CSV
df_clean.to_csv('movies_clean.csv', index=False)

# Save to JSON
df_clean.to_json('movies_clean.json', orient='records', indent=2)

print(f"Saved {len(df_clean)} clean movies")

# Print summary
print("\nData Quality Summary:")
print(f"  Total movies: {len(df_clean)}")
print(f"  Missing values:\n{df_clean.isnull().sum()}")
print(f"  Duplicates: {df_clean.duplicated().sum()}")
```

Part 3 Checkpoint

What you've learned:

- Loading data with pandas
- Identifying data quality issues
- Cleaning string data (removing prefixes, extracting numbers)
- Type conversion
- Handling missing values
- Removing duplicates
- Validating value ranges

Your cleaned dataset is now ready for ML!

Part 4: Complete Validation Pipeline

Putting it all together

Exercise 4.1: Build the Pipeline

Task: Create automated validation script

Create file: `pipeline.py`

```python
import json
import pandas as pd
from pydantic import BaseModel, Field, ValidationError
from typing import Optional

class Movie(BaseModel):
    Title: str
    Year: int = Field(ge=1888, le=2030)
    imdbRating: float = Field(ge=0, le=10)
    Genre: str
    Director: str
    Runtime: Optional[str] = None
    BoxOffice: Optional[str] = None

def clean_data(input_file):
    """Clean raw movie data"""
    # Load
    with open(input_file) as f:
```

Exercise 4.1: Pipeline Functions

```
    # Clean
    df = df.replace('N/A', None)
    df['Year'] = pd.to_numeric(df['Year'], errors='coerce')
    df['imdbRating'] = pd.to_numeric(df['imdbRating'],
                                      errors='coerce')

    # Drop invalid
    df = df.dropna(subset=['Title', 'Year', 'imdbRating'])
    df = df[(df['imdbRating'] >= 0) & (df['imdbRating'] <= 10)]
    df = df[(df['Year'] >= 1888) & (df['Year'] <= 2030)]

    # Remove duplicates
    df = df.drop_duplicates(subset=['Title'])

    return df

def validate_data(df):
    """Validate with Pydantic"""
    valid = []
    invalid = []
```

Exercise 4.1: Validation Loop

```
    for i, row in df.iterrows():
        try:
            movie = Movie(**row.to_dict())
            valid.append(movie.dict())
        except ValidationError as e:
            invalid.append({
                'title': row.get('Title', 'Unknown'),
                'errors': str(e)
            })

    return valid, invalid

def save_results(valid, invalid, output_file):
    """Save validated data and report"""
    # Save valid movies
    with open(output_file, 'w') as f:
        json.dump(valid, f, indent=2)

    # Generate report
    report = f"""
VALIDATION REPORT
=================
Total valid: {len(valid)}
Total invalid: {len(invalid)}
    """
    print(report)
```

Exercise 4.1: Run the Pipeline

```python
def main():
    # Run pipeline
    print("Step 1: Cleaning data...")
    df = clean_data('movies_raw.json')
    print(f"  Cleaned: {len(df)} movies")

    print("\nStep 2: Validating with Pydantic...")
    valid, invalid = validate_data(df)

    print("\nStep 3: Saving results...")
    save_results(valid, invalid, 'movies_validated.json')

    if invalid:
        print(f"\nWarning: {len(invalid)} movies failed validation")
        for inv in invalid[:3]:
            print(f"  - {inv['title']}")

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

Exercise 4.2: Run Your Pipeline

```
python pipeline.py
```

Expected output:

```
Step 1: Cleaning data...
  Cleaned: 48 movies

Step 2: Validating with Pydantic...

Step 3: Saving results...

VALIDATION REPORT
=================
Total valid: 48
Total invalid: 0
```

Exercise 4.3: Add Logging

Task: Track what's happening

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

def clean_data(input_file):
    logger.info(f"Loading data from {input_file}")
    with open(input_file) as f:
        data = json.load(f)

    logger.info(f"Loaded {len(data)} records")
    df = pd.DataFrame(data)

    # Clean steps with logging
    logger.info("Replacing N/A values")
    df = df.replace('N/A', None)

    logger.info("Converting data types")
    # ... rest of cleaning
```

Exercise 4.4: Generate Data Quality Report

Task: Create detailed quality report

```
def generate_report(df_raw, df_clean, valid, invalid):
    report = []

    report.append("=" * 60)
    report.append("DATA QUALITY REPORT")
    report.append("=" * 60)
    report.append("")

    # Input stats
    report.append("INPUT DATA:")
    report.append(f"  Total records: {len(df_raw)}")
    report.append(f"  Columns: {len(df_raw.columns)}")
    report.append("")

    # Cleaning stats
    report.append("CLEANING RESULTS:")
    report.append(f"  Records after cleaning: {len(df_clean)}")
    report.append(f"  Records dropped: {len(df_raw) - len(df_clean)}")
    report.append(f"  Drop rate: {(len(df_raw)-len(df_clean))/len(df_raw)*100:.1f}%")
```

Exercise 4.4: Report Details

```
    # Missing values
    report.append("")
    report.append("MISSING VALUES (clean data):")
    missing = df_clean.isnull().sum()
    for col in missing[missing > 0].index:
        pct = missing[col] / len(df_clean) * 100
        report.append(f"  {col}: {missing[col]} ({pct:.1f}%)")

    # Validation
    report.append("")
    report.append("VALIDATION RESULTS:")
    report.append(f"  Valid movies: {len(valid)}")
    report.append(f"  Invalid movies: {len(invalid)}")
    report.append(f"  Validation pass rate: {len(valid)/len(df_clean)*100:.1f}%")

    # Save report
    with open('validation_report.txt', 'w') as f:
        f.write('\n'.join(report))

    print('\n'.join(report))
```

Exercise 4.5: Test Your Complete Pipeline

Run the full pipeline on your data:

```
python pipeline.py
cat validation_report.txt
```

Verify:

1. All steps complete without errors
2. Valid movies saved to JSON
3. Report shows reasonable statistics
4. Clean data is ready for ML

Mini Project: Extend the Pipeline

Choose one enhancement:

Option A: Add more validation rules

- Validate Genre format (comma-separated)
- Check Director name length
- Validate Runtime format

Option B: Add data enrichment

- Calculate age of movie (current year - release year)
- Extract first genre as primary genre
- Clean and standardize director names

Option C: Add visualizations

Lab Wrap-Up

What you've accomplished:

- Inspected data quality with command-line tools
- Built type-safe validation with Pydantic
- Cleaned messy data with pandas
- Created automated validation pipeline
- Generated data quality reports

Next week: Data Labeling

- Annotation tasks for vision and text
- Using Label Studio
- Inter-annotator agreement metrics
- Building high-quality labeled datasets

Homework

Before next class:

1. **Complete your pipeline**: Validate your full movie dataset
2. **Generate report**: Document all data quality issues found
3. **Clean dataset**: Create final `movies_clean.csv`
4. **Optional**: Add one enhancement from the mini project

Deliverables:

- `movies_clean.csv` - Your validated dataset
- `validation_report.txt` - Quality report
- `pipeline.py` - Your validation script

Resources

Command-line tools:

- jq tutorial: https://stedolan.github.io/jq/tutorial/
- csvkit documentation: https://csvkit.readthedocs.io/

Python libraries:

- Pydantic docs: https://pydantic-docs.helpmanual.io/
- pandas user guide: https://pandas.pydata.org/docs/user_guide/

Data validation:

- Best practices for data quality
- Common validation patterns

Questions?

Remember:

- Start with inspection (jq, csvkit)
- Clean before validating (pandas)
- Use schemas for validation (Pydantic)
- Automate everything (pipeline.py)
- Document your findings (report)

Get help:

- Teaching assistants during lab
- Discussion forum for questions
- Office hours this week