Week 2: Data Validation

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

Continuing the Netflix Story

Last week: We collected movie data from OMDb API

The data we have:

- 50+ movies in JSON format
- Features: title, year, genre, rating, director, etc.
- Saved in `movies_raw.json` and `movies.csv`

This week: Before we can train a model, we need to validate and clean this data

The problem: Real-world data is messy!

The Reality of Data Quality

Question: Is our collected data ready for machine learning?

Answer: Almost certainly not!

Common issues we'll find:

- Missing values (some movies lack box office data)
- Wrong data types (ratings as strings instead of numbers)
- Inconsistent formats (runtime: "148 min" vs "148")
- Duplicates (same movie collected twice)
- Outliers (impossible values)

Today's goal: Detect, validate, and fix these issues

The Data Validation Pipeline

```
RAW DATA → INSPECT → VALIDATE → CLEAN → VERIFIED DATA
```

Week 2 (Today):

1. **Inspect**: Use command-line tools (jq, csvkit)
2. **Validate**: Define schemas with Pydantic
3. **Clean**: Fix issues with pandas
4. **Verify**: Confirm data quality

Output: Clean, validated dataset ready for ML

Today's Agenda

1. **Understanding Data Quality** (what can go wrong)
2. **Tool #1: jq** (inspect and filter JSON)
3. **Tool #2: csvkit** (analyze CSV files)
4. **Tool #3: Pydantic** (schema validation)
5. **Tool #4: pandas** (data cleaning)
6. **Best Practices** (validation pipelines)

Part 1: Understanding Data Quality

What makes data "good" for machine learning?

Characteristics of Good Data

1. **Completeness**:

- All required fields are present
- Missing values are minimal and documented

2. **Accuracy**:

- Values are correct and match reality
- No typos or data entry errors

3. **Consistency**:

- Same format across all records
- Units are standardized

Characteristics of Good Data (continued)

4. Validity:

- Values are within acceptable ranges
- Data types match expectations

5. Uniqueness:

- No duplicate records
- Clear primary keys

6. Timeliness:

- Data is current and up-to-date
- Timestamps are accurate

Data Quality Issues in Our Movie Dataset

Let me show examples from our actual data:

Issue 1: Missing Values

```
{
"Title": "Inception",
"BoxOffice": "N/A",
"Metascore": "N/A"
}
```

Issue 2: String Numbers

```
{
"imdbRating": "8.8",
"Year": "2010"
}
```

Data Quality Issues (continued)

Issue 3: Inconsistent Formats

```
{
"Runtime": "148 min"
}
vs
{
"Runtime": "90 minutes"
}
```

Issue 4: Comma-Separated Strings

```
{
"Genre": "Action, Sci-Fi, Thriller",
"Actors": "Leonardo DiCaprio, Joseph Gordon-Levitt"
}
```

Impact on Machine Learning

Why these issues matter:

Missing values:

- Can't compute features
- Models may crash or give wrong predictions

Wrong types:

- Mathematical operations fail
- "8.8" + "7.5" = "8.87.5" (string concatenation!)

Inconsistent formats:

- Feature extraction breaks
- "148 min" parsed differently than "90 minutes"

Part 2: jq - JSON Validation

Command-line JSON processor

What is jq?

jq: Lightweight command-line JSON processor

Why use it?

- Quickly inspect JSON structure
- Filter and transform data
- Validate JSON syntax
- Extract specific fields
- Count records, find unique values

Installation:

```
# Mac
brew install jq

# Linux
```

Basic jq Usage

Pretty-print JSON:

```
cat movies_raw.json | jq
```

Get number of movies:

```
cat movies_raw.json | jq 'length'
# Output: 52
```

Extract single field:

```
cat movies_raw.json | jq '.[0].Title'
# Output: "Inception"
```

Inspecting Our Movie Data

Get all titles:

```
cat movies_raw.json | jq '.[].Title'
```

Get titles and ratings:

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

Count movies by year:

```
cat movies_raw.json | jq '.[].Year' | sort | uniq -c
```

Finding Data Quality Issues with jq

Find movies with missing box office:

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

Find movies with no Metascore:

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

Check for missing fields:

```
cat movies_raw.json | jq '.[] | select(.Genre == null or .Genre == "")'
```

Filtering with jq

Movies rated above 8.0:

```
cat movies_raw.json | jq '.[] | select(.imdbRating | tonumber > 8.0) | .Title'
```

Movies from 2010s:

```
cat movies_raw.json | jq '.[] | select(.Year | tonumber >= 2010 and tonumber < 2020)'
```

Action movies only:

```
cat movies_raw.json | jq '.[] | select(.Genre | contains("Action"))'
```

Statistics with jq

Average rating:

```
cat movies_raw.json | jq '[.[].imdbRating | tonumber] | add/length'
```

Min and max year:

```
cat movies_raw.json | jq '[.[].Year | tonumber] | min, max'
```

Count by genre (first genre only):

```
cat movies_raw.json | jq '.[].Genre' | cut -d',' -f1 | sort | uniq -c
```

Part 3: csvkit - CSV Analysis

Swiss Army knife for CSV files

What is csvkit?

csvkit: Suite of command-line tools for working with CSV

Why use it?

- Inspect CSV structure and statistics
- Convert between formats
- Query CSVs like databases
- Clean and validate data

Installation:

```
pip install csvkit
```

csvkit Tools

Main tools:

- `csvstat` : Summary statistics
- `csvlook` : Pretty-print CSV
- `csvcut` : Select columns
- `csvgrep` : Filter rows
- `csvsort` : Sort by column
- `csvjoin` : Join CSV files
- `csvstack` : Concatenate CSVs

Inspecting CSV with csvlook

View first few rows:

```
csvlook movies.csv | head -20
```

Pretty table output:

```
| title      | year | rating | genre                |
| ---------- | ---- | ------ | -------------------- |
| Inception  | 2010 | 8.8    | Action, Sci-Fi       |
| The Matrix | 1999 | 8.7    | Action, Sci-Fi       |
```

Much more readable than raw CSV!

Summary Statistics with csvstat

Get statistics for all columns:

```
csvstat movies.csv
```

Output includes:

- Number of rows and columns
- Data types detected
- Unique values count
- Min, max, mean (for numbers)
- Most common values
- Missing values count

csvstat Example Output

```
             1. title
     Type of data:          Text
    Contains null values:  False
      Unique values:         52
 Longest value:         45 characters
Most common values:    Inception (1x)

            2. rating
    Type of data:          Number
     Contains null values:  True
      Unique values:         48
     Min:                   7.5
     Max:                   9.3
     Mean:                  8.12
```

Filtering with csvgrep

Movies rated above 8.5:

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

Movies from 2010:

```
csvgrep -c year -m 2010 movies.csv
```

Action movies:

```
csvgrep -c genre -m "Action" movies.csv
```

Selecting Columns with csvcut

Get only title and rating:

```
csvcut -c title,rating movies.csv
```

Get columns 1, 3, and 5:

```
csvcut -c 1,3,5 movies.csv
```

List all column names:

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

Sorting with csvsort

Sort by rating (descending):

```
csvsort -c rating -r movies.csv
```

Sort by year, then rating:

```
csvsort -c year,rating movies.csv
```

Save sorted output:

```
csvsort -c rating -r movies.csv > movies_sorted.csv
```

Finding Issues with csvkit

Check for missing values:

```
csvstat movies.csv | grep "Contains null"
```

Find duplicate titles:

```
csvcut -c title movies.csv | tail -n +2 | sort | uniq -d
```

Check data types:

```
csvstat -c rating movies.csv
# Should be Number, not Text!
```

Part 4: Pydantic - Schema Validation

Type-safe data validation in Python

What is Pydantic?

Pydantic: Python library for data validation using type hints

Why use it?

- Define expected data structure
- Automatically validate incoming data
- Convert types when possible
- Raise clear errors for invalid data
- Generate JSON schemas

Installation:

```
pip install pydantic
```

Defining a Movie Schema

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

class Movie(BaseModel):
    title: str
    year: int
    genre: str
    director: str
    rating: float = Field(ge=0, le=10)
    votes: Optional[int] = None
    runtime: Optional[str] = None
    box_office: Optional[str] = None
```

This schema defines:

- Required vs optional fields
- Data types for each field

Using the Schema

```
# Valid movie
movie_data = {
"title": "Inception",
"year": 2010,
"genre": "Action, Sci-Fi",
"director": "Christopher Nolan",
"rating": 8.8
}

movie = Movie(**movie_data)
print(movie.title)  # "Inception"
print(movie.rating)  # 8.8
```

Pydantic automatically validates!

Validation Errors

```
# Invalid: rating out of range
bad_movie = {
    "title": "Bad Movie",
    "year": 2020,
    "genre": "Drama",
    "director": "Someone",
    "rating": 15.0  # Invalid!
}

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

Output:

```
rating: ensure this value is less than or equal to 10
```

Type Conversion

```python
# Pydantic converts types when possible
movie_data = {
    "title": "Inception",
    "year": "2010",      # String -> int
    "genre": "Action",
    "director": "Nolan",
    "rating": "8.8"      # String -> float
}

movie = Movie(**movie_data)
print(type(movie.year))    # <class 'int'>
print(type(movie.rating))  # <class 'float'>
```

Automatic type coercion!

Custom Validators

```
 from pydantic import validator

    class Movie(BaseModel):
             title: str
             year: int
            rating: float

         @validator('year')
def year_must_be_reasonable(cls, v):
        if v < 1888 or v > 2030:
      raise ValueError('invalid year')
                return v

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

Validating Our Movie Dataset

```python
import json
from typing import List
from pydantic import ValidationError

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

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

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

Handling Validation Errors

```
# Report errors
print(f"Valid movies: {len(valid_movies)}")
print(f"Invalid movies: {len(errors)}")

# Show first few errors
for error in errors[:3]:
print(f"\nMovie: {error['title']}")
for err in error['errors']:
field = err['loc'][0]
message = err['msg']
print(f"  {field}: {message}")
```

Output:

```
Valid movies: 48
Invalid movies: 4

Movie: Some Movie
```

Part 5: pandas - Data Cleaning

Powerful data manipulation library

Why pandas for Validation?

pandas: Python library for data analysis

Use cases:

- Load and inspect data
- Handle missing values
- Convert data types
- Remove duplicates
- Detect outliers
- Compute statistics

Already installed (part of common ML stack)

Loading Our Movie Data

```python
import pandas as pd

# Load from CSV
df = pd.read_csv('movies.csv')

# Or from JSON
with open('movies_raw.json') as f:
    movies_data = json.load(f)
df = pd.DataFrame(movies_data)

# Basic info
print(df.shape)      # (52, 13)
print(df.columns)    # List of columns
print(df.head())     # First 5 rows
```

Inspecting Data Quality

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

# Check for missing values
print(df.isnull().sum())

# Get summary statistics
print(df.describe())

# Check for duplicates
print(df.duplicated().sum())
```

Example: Missing Values

```
# Count missing per column
missing = df.isnull().sum()
print(missing[missing > 0])
```

Output:

```
box_office    15
metascore      8
awards         3
```

Interpretation:

- 15 movies missing box office data
- 8 movies missing metascore
- 3 movies missing awards

Handling Missing Values

Strategy 1: Drop rows with missing critical fields:

```
# Drop if rating is missing
df_clean = df.dropna(subset=['rating'])
```

Strategy 2: Fill with defaults:

```
# Fill missing box_office with 0
df['box_office'] = df['box_office'].fillna(0)
```

Strategy 3: Drop columns with too many missing:

```
# Drop if > 50% missing
threshold = len(df) * 0.5
df_clean = df.dropna(thresh=threshold, axis=1)
```

Type Conversion

```
# Check current types
print(df['rating'].dtype)  # object (string!)

# Convert to numeric
df['rating'] = pd.to_numeric(df['rating'], errors='coerce')
df['year'] = pd.to_numeric(df['year'], errors='coerce')

# Convert to int
df['votes'] = df['votes'].str.replace(',', '').astype(int)

# Check again
print(df['rating'].dtype)  # float64
```

Cleaning String Data

```
      # Extract runtime as integer
       df['runtime_minutes'] = (
               df['runtime']
         .str.extract(r'(\d+)')[0]
                .astype(int)
                   )

          # Before: "148 min"
             # After:  148

# Clean box office (remove $ and commas)
       df['box_office_clean'] = (
              df['box_office']
           .str.replace('$', '')
           .str.replace(',', '')
               .astype(float)
                   )
```

Removing Duplicates

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

# Remove duplicates (keep first)
df_clean = df.drop_duplicates(subset=['title'], keep='first')

# Or keep the one with more data
df_clean = df.sort_values('rating', ascending=False)
df_clean = df_clean.drop_duplicates(subset=['title'], keep='first')
```

Detecting Outliers

```python
# Check rating distribution
print(df['rating'].describe())

# Find potential outliers (> 3 std deviations)
mean = df['rating'].mean()
std = df['rating'].std()

outliers = df[
    (df['rating'] < mean - 3*std) |
    (df['rating'] > mean + 3*std)
]

print(f"Found {len(outliers)} outliers")
print(outliers[['title', 'rating']])
```

Validating Value Ranges

```
# Check if ratings are in valid range
invalid_ratings = df[
(df['rating'] < 0) | (df['rating'] > 10)
]
print(f"Invalid ratings: {len(invalid_ratings)}")

# Check if years are reasonable
invalid_years = df[
(df['year'] < 1888) | (df['year'] > 2030)
]
print(f"Invalid years: {len(invalid_years)}")
```

Complete Cleaning Pipeline

```python
def clean_movie_data(df):
    # 1. Remove duplicates
    df = df.drop_duplicates(subset=['title'])

    # 2. Convert types
    df['rating'] = pd.to_numeric(df['rating'], errors='coerce')
    df['year'] = pd.to_numeric(df['year'], errors='coerce')

    # 3. Drop rows with missing critical fields
    df = df.dropna(subset=['title', 'year', 'rating'])

    # 4. Validate ranges
    df = df[
        (df['rating'] >= 0) & (df['rating'] <= 10) &
        (df['year'] >= 1888) & (df['year'] <= 2030)
    ]

    # 5. Clean string columns
    df['runtime_min'] = df['runtime'].str.extract(r'(\d+)')[0]

    return df
```

Running the Pipeline

```python
# Load raw data
df_raw = pd.read_csv('movies_raw.csv')
print(f"Raw data: {len(df_raw)} movies")

# Clean
df_clean = clean_movie_data(df_raw)
print(f"Clean data: {len(df_clean)} movies")

# Validate
print("\nData quality report:")
print(f"Missing values:\n{df_clean.isnull().sum()}")
print(f"\nDuplicates: {df_clean.duplicated().sum()}")
print(f"\nData types:\n{df_clean.dtypes}")

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

Part 6: Building a Validation Pipeline

Putting it all together

Validation Pipeline Architecture

```
  1. RAW DATA (movies_raw.json)
                  ↓
 2. SCHEMA VALIDATION (Pydantic)
      ↓ (filter invalid records)
   3. TYPE CONVERSION (pandas)
                  ↓
4. MISSING VALUE HANDLING (pandas)
                  ↓
  5. DUPLICATE REMOVAL (pandas)
                  ↓
  6. OUTLIER DETECTION (pandas)
                  ↓
 7. CLEAN DATA (movies_clean.csv)
                  ↓
 8. VALIDATION REPORT (report.txt)
```

Step 1: Schema Validation

```
from pydantic import BaseModel, ValidationError
import json

class Movie(BaseModel):
    title: str
    year: int
    rating: float
    genre: str
    director: str

# Load and validate
with open('movies_raw.json') as f:
    raw_data = json.load(f)

valid = []
invalid = []

for movie in raw_data:
    try:
        m = Movie(**movie)
        valid.append(m.dict())
    except ValidationError:
        invalid.append(movie)
```

Step 2: Data Cleaning

```
import pandas as pd

# Convert to DataFrame
df = pd.DataFrame(valid)

# Clean
df['rating'] = pd.to_numeric(df['rating'], errors='coerce')
df['year'] = pd.to_numeric(df['year'], errors='coerce')

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

# Handle missing
df = df.dropna(subset=['rating', 'year'])

# Validate ranges
df = df[(df['rating'] >= 0) & (df['rating'] <= 10)]
```

Step 3: Generate Report

```python
def generate_validation_report(raw, clean, invalid):
    report = []

    report.append("DATA VALIDATION REPORT")
    report.append("=" * 50)
    report.append(f"Input records: {len(raw)}")
    report.append(f"Schema failures: {len(invalid)}")
    report.append(f"Output records: {len(clean)}")
    report.append(f"Records dropped: {len(raw) - len(clean)}")
    report.append("")

    report.append("Data Quality Metrics:")
    report.append(f"  Completeness: {len(clean)/len(raw)*100:.1f}%")
    report.append(f"  Duplicates removed: {len(raw) - len(clean.drop_duplicates())}")

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

Complete Validation Script

```
# validate_movies.py
import json
import pandas as pd
from pydantic import BaseModel, ValidationError

class Movie(BaseModel):
title: str
year: int
rating: float
genre: str

def validate_and_clean(input_file, output_file):
# Load
with open(input_file) as f:
raw = json.load(f)

# Validate with Pydantic
valid = []
for movie in raw:
try:
valid.append(Movie(**movie).dict())
except ValidationError:
pass
```

Complete Script (continued)

```
# Clean with pandas
df = pd.DataFrame(valid)
df['rating'] = pd.to_numeric(df['rating'], errors='coerce')
df = df.dropna()
df = df.drop_duplicates()

# Save
df.to_csv(output_file, index=False)

# Report
print(f"Validated {len(df)}/{len(raw)} movies")
return df

# Run
if __name__ == "__main__":
df = validate_and_clean('movies_raw.json', 'movies_clean.csv')
```

Best Practices

1. Validate early:

- Check data as soon as you collect it
- Catch issues before they propagate

2. Be explicit:

- Define schemas clearly
- Document validation rules

3. Log everything:

- Record what was rejected and why
- Generate validation reports

Best Practices (continued)

4. Fail gracefully:

- Don't crash on bad data
- Handle errors and continue

5. Preserve raw data:

- Keep original files
- Cleaning is separate step

6. Automate:

- Make validation repeatable
- Run on every new data batch

Summary: Validation Workflow

For our Netflix movie dataset:

1. **Inspect**: Use jq and csvkit to explore
2. **Define schema**: Create Pydantic models
3. **Validate**: Check against schema
4. **Clean**: Fix issues with pandas
5. **Report**: Document data quality
6. **Save**: Export clean dataset

Output: Validated `movies_clean.csv` ready for feature engineering

Next Steps

Week 3: Data Labeling

- Annotation tasks for vision and text
- Using Label Studio
- Inter-annotator agreement
- Building high-quality training data

Homework:

- Validate your Week 1 movie dataset
- Fix all data quality issues
- Generate a validation report
- Prepare for Week 3 labeling exercises

Key Takeaways

1. **Data validation is critical**: Bad data = bad models
2. **Use multiple tools**: jq for quick checks, Pydantic for schemas, pandas for cleaning
3. **Automate validation**: Build repeatable pipelines
4. **Document everything**: Track what was changed and why
5. **Preserve raw data**: Never overwrite originals
6. **Validate continuously**: Check data at every pipeline stage

Resources

Command-line tools:

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

Python libraries:

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

Additional reading:

- Data validation patterns
- Schema design best practices

Questions?

Next class: Lab session - hands-on data validation with your movie dataset