

Week 2: Data Validation & Labeling

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra
IIT Gandhinagar

Today's Agenda (90 minutes)

1. Introduction (5 min)

- Why data quality matters
- Common data issues

2. Command-Line Data Inspection (25 min)

- jq for JSON validation
- csvkit tools
- Unix text processing

3. Python Validation with Pydantic (30 min)

- Schema validation
- Type checking

Why Data Quality Matters

The Reality

"Garbage in, garbage out" - Every data scientist ever

Statistics:

- 80% of AI project time: Data preparation
- 47% of newly created data records have at least one critical error
- Poor data quality costs organizations \$15M/year on average

Impact on ML Models

- **Training:** Bad data leads to poor model performance
- **Inference:** Unexpected inputs cause failures
- **Monitoring:** Drift detection requires clean baselines

Common Data Quality Issues

From Last Week's Scraping

- **Missing fields:** Null values, empty strings
- **Wrong types:** String instead of number
- **Malformed data:** Invalid JSON, broken HTML
- **Duplicates:** Same record multiple times
- **Inconsistent formats:** Dates, phone numbers, addresses
- **Outliers:** Extreme or impossible values

Today's Goal

Learn to detect, validate, and fix these issues!

Part 1: Command-Line Tools

Why Command Line?

- **Fast:** Process millions of rows instantly
- **Portable:** Works on any Unix system
- **Composable:** Chain tools with pipes
- **Memory-efficient:** Stream processing
- **Scriptable:** Automate validation pipelines

Tools We'll Cover

1. **jq:** JSON processor and validator
2. **csvkit:** CSV Swiss Army knife
3. **Unix tools:** head, tail, wc, sort, uniq

jq: JSON Query Language

What is jq?

Command-line JSON processor for:

- Validation and formatting
- Filtering and transformation
- Extraction and aggregation

Install:

```
# macOS  
brew install jq  
  
# Ubuntu/Debian  
apt-get install jq
```

jq Basics

Pretty Printing

```
# Ugly JSON
echo '{"name":"Alice","age":25,"city":"Ahmedabad"}' | jq

# Output:
{
  "name": "Alice",
  "age": 25,
  "city": "Ahmedabad"
}
```

Validate JSON

```
# Valid JSON
echo '{"valid": true}' | jq
```

jq Field Extraction

Basic Queries

```
# Extract single field
echo '{"name":"Alice","age":25}' | jq '.name'
# "Alice"

# Extract nested field
echo '{"user":{"name":"Alice"}}' | jq '.user.name'
# "Alice"

# Extract from array
echo ' [{"name":"Alice"}, {"name":"Bob"} ]' | jq '.[0].name'
# "Alice"

# All items in array
echo ' [{"name":"Alice"}, {"name":"Bob"} ]' | jq '.[].name'
# "Alice"
# "Bob"
```

jq Filtering and Transformation

Filtering

```
# Filter objects
echo ' [{"age":20},{ "age":30}]' | jq '.[] | select(.age > 25)'
# { "age": 30}

# Count items
echo '[1,2,3,4,5]' | jq 'length'
# 5

# Check if field exists
echo '{"name":"Alice"}' | jq 'has("age")'
# false
```

Transformation

jq Real-World Example

Validate Scraped Data

```
# Sample scraped data
cat articles.json
[
  {"title": "Article 1", "views": 100, "date": "2024-01-01"},
  {"title": "Article 2", "views": "invalid", "date": "2024-01-02"},
  {"title": null, "views": 50}
]

# Find articles with missing titles
jq '.[[] | select(.title = null or .title = "")' articles.json

# Find articles with invalid views
jq '.[[] | select(.views | type ≠ "number")' articles.json

# Get summary statistics
jq '[ [ [] views | select(type = "number")] | add / length' articles.json
```

csvkit: CSV Powertools

What is csvkit?

Suite of command-line tools for working with CSV:

- **csvstat**: Summary statistics
- **csvclean**: Find and fix errors
- **csvsql**: SQL queries on CSV
- **csvjson**: Convert CSV to JSON
- **csvcut**: Extract columns
- **csvgrep**: Filter rows

Install:

```
pip install csvkit
```

csvstat: Data Profiling

Quick Statistics

```
# Generate statistics for all columns
csvstat data.csv
```

```
# Example output:
```

```
1. "name"
```

```
  Type of data:  Text
  Unique values: 150
  Most common:   Alice (3)
```

```
2. "age"
```

```
  Type of data:  Number
  Min:           18
  Max:           65
  Mean:          35.2
  Median:        33
```

csvclean: Error Detection

Find Problems

```
# Check for errors
csvclean data.csv

# Creates two files:
# - data_out.csv (clean records)
# - data_err.csv (problematic records)

# Example error output:
line_number,msg,name,age
5,Expected 3 columns but found 2,Bob,
12,Expected 3 columns but found 4,Charlie,25,extra,data
```

Common Issues Found

- Inconsistent column counts

csvsql: Query CSV with SQL

Basic Queries

```
# Query CSV with SQL
csvsql --query "SELECT name, age FROM data WHERE age > 30" data.csv

# Join multiple CSVs
csvsql --query "
    SELECT u.name, o.total
    FROM users u
    JOIN orders o ON u.id = o.user_id
" users.csv orders.csv

# Group and aggregate
csvsql --query "
    SELECT city, AVG(age) as avg_age, COUNT(*) as count
    FROM data
    GROUP BY city
" data.csv
```

csvkit Pipeline Example

Complete Workflow

```
# 1. Convert scraped JSON to CSV
csvjson scraped_data.json > data.csv

# 2. Clean and validate
csvclean data.csv

# 3. Get statistics
csvstat data_out.csv

# 4. Extract relevant columns
csvcut -c name,price,rating data_out.csv > clean_data.csv

# 5. Filter high-rated items
csvgrep -c rating -r "^[4-5]" clean_data.csv > filtered.csv

# 6. Count results
```

Unix Text Processing

Essential Commands

```
# Count lines (rows)
```

```
wc -l data.csv
```

```
# First 10 rows
```

```
head -10 data.csv
```

```
# Last 10 rows
```

```
tail -10 data.csv
```

```
# Find duplicates
```

```
sort data.csv | uniq -d
```

```
# Count unique values
```

```
cut -d',' -f2 data.csv | sort | uniq -c
```

```
# Count specific pattern
```

Practical Unix Example

Analyze Scraped Data

```
# Count total records
wc -l products.csv
# 10000 products.csv

# Check for empty fields (assuming CSV with commas)
grep ',,' products.csv | wc -l
# 45 (45 records with empty fields)

# Find unique categories
cut -d',' -f3 products.csv | sort | uniq
# Electronics
# Books
# Clothing

# Count items per category
cut -d',' -f3 products.csv | sort | uniq -c | sort -rn
```

Part 2: Python Validation

Why Python After Command-Line?

Command-line: Quick exploration and filtering

Python: Complex validation logic and automation

Pydantic

Modern data validation using Python type annotations:

- **Type checking:** Automatic type conversion
- **Validation:** Custom rules and constraints
- **Serialization:** Convert to/from JSON
- **IDE support:** Auto-completion and type hints

Install:

Pydantic Basics

Define a Model

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: int
    email: str
    active: bool = True # Default value

# Valid data
user = User(name="Alice", age=25, email="alice@example.com")
print(user)
# name='Alice' age=25 email='alice@example.com' active=True

# Type conversion
user2 = User(name="Bob", age="30", email="bob@example.com")
print(user2.age, type(user2.age))
```

Pydantic Validation Errors

Handling Invalid Data

```
from pydantic import BaseModel, ValidationError

class User(BaseModel):
    name: str
    age: int
    email: str

# Invalid data
try:
    user = User(name="Charlie", age="invalid", email="charlie@example.com")
except ValidationError as e:
    print(e)
```

Output:

Pydantic Field Constraints

Built-in Validators

```
from pydantic import BaseModel, Field, field_validator

class Product(BaseModel):
    name: str = Field(..., min_length=1, max_length=100)
    price: float = Field(..., gt=0) # Greater than 0
    quantity: int = Field(..., ge=0) # Greater than or equal to 0
    category: str

    @field_validator('category')
    @classmethod
    def validate_category(cls, v):
        allowed = ['Electronics', 'Books', 'Clothing']
        if v not in allowed:
            raise ValueError(f'Category must be one of {allowed}')
        return v

# Test
product = Product(
    name="Laptop",
    price=999.99,
```

Pydantic for Scraped Data

Real Example

```
from pydantic import BaseModel, HttpUrl, field_validator
from typing import Optional
from datetime import datetime

class Article(BaseModel):
    title: str = Field(..., min_length=1)
    url: HttpUrl # Validates URL format
    author: str
    published_date: datetime
    views: int = Field(..., ge=0)
    tags: list[str] = []
    rating: Optional[float] = Field(None, ge=0, le=5)

    @field_validator('tags')
    @classmethod
    def validate_tags(cls, v):
        return [tag.strip().lower() for tag in v]

# Load scraped data
import json

with open('scraped_articles.json') as f:
    raw_data = json.load(f)

# Validate each article
valid_articles = []
errors = []

for item in raw_data:
    try:
        article = Article(**item)
```

Pydantic Nested Models

Complex Structures

```
from pydantic import BaseModel
from typing import List

class Address(BaseModel):
    street: str
    city: str
    zipcode: str

class User(BaseModel):
    name: str
    age: int
    addresses: List[Address]

# Usage
user_data = {
    "name": "Alice",
    "age": 25,
    "addresses": [
        {"street": "123 Main St", "city": "Ahmedabad", "zipcode": "380001"},
        {"street": "456 Park Ave", "city": "Gandhinagar", "zipcode": "382001"}
    ]
}
```

Pydantic Export and Serialization

Convert to JSON/Dict

```
from pydantic import BaseModel

class Product(BaseModel):
    name: str
    price: float
    in_stock: bool

product = Product(name="Laptop", price=999.99, in_stock=True)

# To dictionary
print(product.model_dump())
# {'name': 'Laptop', 'price': 999.99, 'in_stock': True}

# To JSON string
print(product.model_dump_json())
# {"name":"Laptop","price":999.99,"in_stock":true}
```

Complete Validation Pipeline

Putting It All Together

```
from pydantic import BaseModel, ValidationError, Field
import json
import csv

class ScrappedProduct(BaseModel):
    name: str = Field(..., min_length=1)
    price: float = Field(..., gt=0)
    url: str
    rating: float = Field(..., ge=0, le=5)

# Load scraped data
with open('scraped.json') as f:
    raw_data = json.load(f)

valid_data = []
error_log = []

# Validate
for i, item in enumerate(raw_data):
    try:
        product = ScrappedProduct(**item)
        valid_data.append(product.model_dump())
    except ValidationError as e:
        error_log.append({
            'line': i,
            'data': item,
            'errors': str(e)
        })

# Save valid data
with open('clean_products.json', 'w') as f:
    json.dump(valid_data, f, indent=2)

# Save error log
```

Part 3: Data Labeling

Why Label Data?

Supervised learning needs labels:

- Text classification: Sentiment, topics, intent
- Named Entity Recognition: Persons, places, organizations
- Image annotation: Bounding boxes, segmentation
- Quality assessment: Good/bad, relevant/irrelevant

Challenges

- Time-consuming and expensive
- Requires domain expertise
- Consistency across annotators

Label Studio

What is Label Studio?

Open-source data labeling tool supporting:

- Text: Classification, NER, Q&A
- Images: Detection, segmentation, keypoints
- Audio: Transcription, classification
- Video: Object tracking

Features:

- Web-based interface
- Multiple annotators
- Export formats: JSON, CSV, COCO, YOLO

Label Studio Setup

Installation

```
# Install
pip install label-studio

# Start server
label-studio start

# Opens browser at http://localhost:8080
```

Create Project

1. Sign up / Login
2. Create Project
3. Choose task type

Text Classification Example

Setup

Data format (tasks.json):

```
[
  {"text": "This product is amazing! Highly recommend."},
  {"text": "Terrible quality, broke after one week."},
  {"text": "Average product, nothing special."}
]
```

Labeling config:

```
<View>
  <Text name="text" value="$text" />
  <Choices name="sentiment" toName="text">
    <Choice value="Positive" />
    <Choice value="Negative" />
  </Choices>
</View>
```

Named Entity Recognition

NER Configuration

```
<View>
  <Text name="text" value="$text" />
  <Labels name="label" toName="text">
    <Label value="Person" background="red" />
    <Label value="Organization" background="blue" />
    <Label value="Location" background="green" />
    <Label value="Date" background="yellow" />
  </Labels>
</View>
```

Example text:

"Alice visited IIT Gandhinagar on January 15th, 2024."

Annotations:

Image Annotation

Bounding Box Configuration

```
<View>
  <Image name="image" value="$image" />
  <RectangleLabels name="label" toName="image">
    <Label value="Person" background="red" />
    <Label value="Car" background="blue" />
    <Label value="Bicycle" background="green" />
  </RectangleLabels>
</View>
```

Use cases:

- Object detection
- Face recognition
- Document layout analysis

Export Formats

Common Formats

```
# JSON (default)
{
  "id": 1,
  "data": {"text": "Sample text"},
  "annotations": [{
    "result": [
      {
        "value": {"choices": ["Positive"]},
        "from_name": "sentiment",
        "to_name": "text"
      }
    ]
  }]
}
```

```
# CSV
id,text,sentiment
1,"Sample text","Positive"
```

```
# COCO (for images)
{
  "images": [
    {
      "id": 1,
      "file_name": "Sample image.jpg",
      "width": 100,
      "height": 100,
      "coco_url": "http://example.com/sample_image.jpg"
    }
  ],
  "annotations": [
    {
      "id": 1,
      "image_id": 1,
      "caption": "A sample image with a positive sentiment.",
      "sentiment": "Positive"
    }
  ]
}
```

Inter-Annotator Agreement

Why Measure Agreement?

- **Quality control:** Ensure consistent labeling
- **Ambiguity detection:** Find unclear examples
- **Annotator training:** Identify who needs help
- **Dataset validation:** Assess label reliability

Common Metrics

1. **Percent Agreement:** Simple percentage
2. **Cohen's Kappa:** Agreement beyond chance (2 annotators)
3. **Fleiss' Kappa:** Agreement for 3+ annotators
4. **Krippendorff's Alpha:** Handles missing data

Cohen's Kappa

Formula

$$\kappa = (P_o - P_e) / (1 - P_e)$$

P_o = Observed agreement

P_e = Expected agreement by chance

Interpretation:

- $\kappa < 0$: No agreement (worse than chance)
- 0.0-0.20: Slight agreement
- 0.21-0.40: Fair agreement
- 0.41-0.60: Moderate agreement
- 0.61-0.80: Substantial agreement

Calculate Cohen's Kappa

Python Implementation

```
from sklearn.metrics import cohen_kappa_score

# Annotations from two annotators
annotator1 = ['pos', 'neg', 'pos', 'neu', 'pos', 'neg']
annotator2 = ['pos', 'neg', 'neu', 'neu', 'pos', 'neg']

kappa = cohen_kappa_score(annotator1, annotator2)
print(f"Cohen's Kappa: {kappa:.3f}")
# Cohen's Kappa: 0.632 (Substantial agreement)

# With label mapping
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(annotator1, annotator2)
print(cm)
#           pos  neg  neu
```

Fleiss' Kappa

Multiple Annotators

```
from statsmodels.stats.inter_rater import fleiss_kappa

# Format: rows = items, cols = categories
# Values = number of annotators who chose that category
data = [
    [0, 0, 3], # Item 1: 3 annotators chose category 3
    [1, 2, 0], # Item 2: 1 chose cat 1, 2 chose cat 2
    [0, 3, 0], # Item 3: 3 annotators chose category 2
    [2, 1, 0], # Item 4: 2 chose cat 1, 1 chose cat 2
]

kappa = fleiss_kappa(data)
print(f"Fleiss' Kappa: {kappa:.3f}")
```

Use when: 3+ annotators, not all annotate all items

Improving Annotation Quality

Best Practices

Before Labeling:

- Clear guidelines and examples
- Training sessions for annotators
- Pilot study on small sample

During Labeling:

- Regular check-ins and feedback
- Periodic agreement measurement
- Resolve disagreements through discussion

After Labeling:

Data Validation Workflow

Complete Pipeline

1. Scrape Data
|
2. Command-line quick check (jq, csvstat)
|
3. Python validation (Pydantic)
|
4. Generate error report
|
5. Fix or remove bad records
|
6. Label clean data (Label Studio)
|
7. Calculate inter-annotator agreement
|
8. Review disagreements
|

Great Expectations (Brief Intro)

What is Great Expectations?

Data quality framework for:

- Data profiling
- Validation rules
- Automated testing
- Documentation generation

```
import great_expectations as gx

# Create expectation
context = gx.get_context()
batch = context.sources.pandas_default.read_csv("data.csv")

# Define expectations
```

Best Practices Summary

Data Validation

- Always validate at data ingestion
- Use schema validation (Pydantic)
- Log errors for debugging
- Separate valid and invalid data
- Monitor data quality over time

Labeling

- Create clear annotation guidelines
- Use multiple annotators for critical data
- Measure inter-annotator agreement

Tools Comparison

Tool	Use Case	Pros	Cons
jq	JSON exploration	Fast, powerful	Learning curve
csvkit	CSV analysis	Easy, comprehensive	Slow on huge files
Pydantic	Python validation	Type - safe, modern	Python - only
Label Studio	Annotation	Full - featured, free	Setup required
Great Expectations	Production pipelines	Automated, documented	Complex setup

Lab Preview

What You'll Do Today

Part 1: Command-line validation (45 min)

- Use jq on scraped JSON from Week 1
- csvkit analysis and cleaning
- Unix text processing

Part 2: Pydantic validation (60 min)

- Define models for your scraped data
- Validate and clean datasets
- Generate error reports

Part 3: Label Studio (60 min)

Questions?

Get Ready for Lab!

What to install:

```
# Command-line tools
brew install jq # or apt-get install jq

# Python packages
pip install pydantic label-studio csvkit pandas scikit-learn statsmodels

# Start Label Studio
label-studio start
```

Bring:

- Your scraped data from Week 1
- Ideas for what you want to label

See You in Lab!

Remember: Clean data is the foundation of good AI

Next week: LLM APIs and multimodal AI