Week 2 Lab: Data Validation & Labeling

CS 203: Software Tools and Techniques for AI

Duration: 3 hours

Prof. Nipun Batra & Teaching Assistants

Lab Overview

Today's Goals

By the end of this lab, you will:

- Master command-line tools for data inspection
- Validate data with Pydantic models
- Set up and use Label Studio
- Calculate inter-annotator agreement
- Build a complete validation pipeline

Structure

- **Part 1**: Command-line tools (45 min)
- **Part 2**: Python validation with Pydantic (60 min)
- **Part 3**: Label Studio annotation (60 min)
- **Part 4**: Complete pipeline (15 min)

Setup Check (10 minutes)

Install Required Tools

```
# macOS command-line tools
brew install jq

# Linux command-line tools
sudo apt-get install jq

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

# Verify installations
jq --version
csvstat --version
python -c "import pydantic; print(pydantic.__version__)"

# Start Label Studio (in separate terminal)
label-studio start
```

Get Sample Data

Part 1: Command-Line Data Inspection

Exercise 1.1: jq Basics (15 min)

Task: Explore and validate JSON data

Create sample file `users.json`:

```
[
  {"name": "Alice", "age": 25, "email": "alice@example.com", "city": "Ahmedabad"},
  {"name": "Bob", "age": "thirty", "email": "bob@invalid", "city": "Gandhinagar"},
  {"name": null, "age": 35, "email": "charlie@example.com"},
  {"name": "Diana", "age": -5, "email": "diana@example.com", "city": "Surat"}
]
```

Tasks:

1. Pretty print the JSON
2. Extract all names
3. Find users with missing names

Exercise 1.1: Solution

```
# 1. Pretty print
jq '.' users.json

# 2. Extract all names
jq '.[].name' users.json

# 3. Find users with missing names
jq '.[] | select(.name == null or .name == "")' users.json

# 4. Find users with invalid ages (not a number or < 0)
jq '.[] | select((.age | type) != "number" or .age < 0)' users.json

# Output:
# {"name":"Bob","age":"thirty",...}  # age is string
# {"name":"Diana","age":-5,...}      # age is negative

# 5. Count users per city
jq '[.[].city] | group_by(.) | map({city: .[0], count: length})' users.json

# Or simpler (if you just want counts):
jq -r '.[].city' users.json | sort | uniq -c
```

Exercise 1.2: jq Advanced Filtering (15 min)

Task: Validate scraped article data

Sample `articles.json`:

```
[
  {"title": "Article 1", "url": "https://example.com/1", "views": 100, "rating": 4.5},
  {"title": "", "url": "not-a-url", "views": -10, "rating": 4.5},
  {"title": "Article 3", "url": "https://example.com/3", "views": 50, "rating": 6.0}
]
```

Requirements:

1. Find articles with empty titles
2. Find articles with invalid URLs (no http/https)
3. Find articles with negative views
4. Find articles with rating > 5
5. Create a clean dataset with only valid articles

Exercise 1.2: Solution

```
# 1. Empty titles
jq '.[] | select(.title == "")' articles.json

# 2. Invalid URLs (not starting with http)
jq '.[] | select(.url | startswith("http") | not)' articles.json

# 3. Negative views
jq '.[] | select(.views < 0)' articles.json

# 4. Rating > 5
jq '.[] | select(.rating > 5)' articles.json

# 5. Clean dataset (all validations)
jq '
  map(select(
    .title ≠ "" and
    (.url | startswith("http")) and
    .views ≥ 0 and
    .rating ≥ 0 and .rating ≤ 5
  ))
' articles.json > clean_articles.json

# Bonus: Average views
jq '[.[].views] | add / length' clean_articles.json
```

Exercise 1.3: csvkit Practice (15 min)

Task: Analyze CSV data

Create `products.csv`:

```
name,price,category,stock,rating
Laptop,999.99,Electronics,10,4.5
Mouse,25.50,Electronics,50,4.2
Book,15.00,Books,100,4.8
Keyboard,invalid,Electronics,20,4.1
Chair,150.00,Furniture,5,3.9
"",99.99,Electronics,15,4.0
```

Tasks:

1. Get summary statistics for all columns
2. Clean the file and identify errors
3. Count products per category
4. Find average price by category

Exercise 1.3: Solution

```
# 1. Summary statistics
csvstat products.csv

# 2. Clean and find errors
csvclean products.csv
# Creates products_out.csv and products_err.csv

cat products_err.csv
# Shows: line 4 (Keyboard with invalid price)
#        line 6 (empty name)

# 3. Count per category
csvstat -c category products.csv
# Or using csvcut and sort:
csvcut -c category products.csv | tail -n +2 | sort | uniq -c

# 4. Average price by category
csvsql --query "
  SELECT category, AVG(CAST(price AS REAL)) as avg_price
  FROM products
  WHERE price != 'invalid'
  GROUP BY category
" products.csv

# 5. Extract Electronics only
csvgrep -c category -m Electronics products.csv > electronics.csv
```

Part 1 Checkpoint

What You've Learned

- jq for JSON validation and filtering
- csvkit for CSV analysis and cleaning
- Unix tools for quick data inspection
- Finding data quality issues

Common Issues Found

- Missing values (null, empty strings)
- Wrong types (string instead of number)
- Invalid ranges (negative values, ratings > 5)
- Malformed data (invalid URLs, emails)

Share your findings: What data issues did you discover?

Part 2: Python Validation with Pydantic

Exercise 2.1: Basic Pydantic Model (20 min)

Task: Create a validation model for user data

```
from pydantic import BaseModel, ValidationError
import json

# TODO: Define a User model with:
# - name: required string, min length 1
# - age: required int, between 0 and 120
# - email: required string (bonus: use EmailStr)
# - city: required string

# Test data
test_data = [
    {"name": "Alice", "age": 25, "email": "alice@example.com", "city": "Ahmedabad"},
    {"name": "", "age": 25, "email": "bob@example.com", "city": "Surat"},
    {"name": "Charlie", "age": -5, "email": "charlie@example.com", "city": "Mumbai"},
    {"name": "Diana", "age": "thirty", "email": "invalid", "city": "Pune"}
]
```

Exercise 2.1: Solution

```
from pydantic import BaseModel, Field, field_validator, EmailStr
from typing import List
import json

class User(BaseModel):
    name: str = Field( ... , min_length=1)
    age: int = Field( ... , ge=0, le=120)
    email: EmailStr  # Validates email format
    city: str

test_data = [
    {"name": "Alice", "age": 25, "email": "alice@example.com", "city": "Ahmedabad"},
    {"name": "", "age": 25, "email": "bob@example.com", "city": "Surat"},
    {"name": "Charlie", "age": -5, "email": "charlie@example.com", "city": "Mumbai"},
    {"name": "Diana", "age": "thirty", "email": "invalid", "city": "Pune"}
]

valid_users = []
errors = []

for i, data in enumerate(test_data):
    try:
        user = User(**data)
        valid_users.append(user.model_dump())
    except ValidationError as e:
        errors.append({
            'index': i,
            'data': data,
            'errors': e.errors()
        })

print(f"Valid: {len(valid_users)}")
print(f"Errors: {len(errors)}")

# Print error details
for err in errors:
    print(f"\nRecord {err['index']}: {err['data']}")
    print(f"Errors: {err['errors']}")
```

Exercise 2.2: Validating Scraped Data (25 min)

Task: Create a comprehensive model for your Week 1 scraped data

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

# TODO: Define model based on YOUR scraped data structure
# Example for articles:

class Article(BaseModel):
    title: str = Field(..., min_length=1, max_length=500)
    url: HttpUrl
    author: str
    published_date: Optional[datetime] = None
    views: int = Field(..., ge=0)
    rating: float = Field(..., ge=0, le=5)
    tags: List[str] = []

    @field_validator('tags')
    @classmethod
    def clean_tags(cls, v):
        # Clean and normalize tags
        return [tag.strip().lower() for tag in v if tag.strip()]

# TODO:
# 1. Load your scraped JSON data
# 2. Validate each record
# 3. Save valid records to clean_data.json
# 4. Save errors to validation_errors.json
# 5. Generate statistics report
```

Exercise 2.2: Solution Template

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

# Define your model (customize for your data)
class ScrapedItem(BaseModel):
    # Add your fields here
    pass

# Load data
with open('week1_scraped_data.json') as f:
    raw_data = json.load(f)

# Validate
valid_data = []
error_log = []

for i, item in enumerate(raw_data):
    try:
        validated = ScrapedItem(**item)
        valid_data.append(validated.model_dump())
    except ValidationError as e:
        error_log.append({
            'record_number': i,
            'data': item,
            'errors': [
                {
                    'field': err['loc'][0],
                    'error': err['msg'],
                    'value': err.get('input')
                }
                for err in e.errors()
            ]
        })

# Save results
with open('clean_data.json', 'w') as f:
    json.dump(valid_data, f, indent=2)

with open('validation_errors.json', 'w') as f:
    json.dump(error_log, f, indent=2)

# Generate report
total = len(raw_data)
valid = len(valid_data)
invalid = len(error_log)

print("=" * 50)
print("VALIDATION REPORT")
print("=" * 50)
print(f"Total records: {total}")
print(f"Valid records: {valid} ({valid/total*100:.1f}%)")
print(f"Invalid records: {invalid} ({invalid/total*100:.1f}%)")
print()

# Error analysis
if error_log:
    error_fields = {}
    for log in error_log:
        for err in log['errors']:
            field = err['field']
            error_fields[field] = error_fields.get(field, 0) + 1

    print("Errors by field:")
    for field, count in sorted(error_fields.items(), key=lambda x: x[1], reverse=True):
        print(f"  {field}: {count}")
```

Exercise 2.3: Custom Validators (15 min)

Task: Add complex validation logic

```
from pydantic import BaseModel, field_validator, model_validator
import re

class Product(BaseModel):
    name: str
    sku: str  # Format: XXX-NNNN (3 letters, dash, 4 numbers)
    price: float
    discount_price: Optional[float] = None

    @field_validator('sku')
    @classmethod
    def validate_sku(cls, v):
        # TODO: Validate SKU format
        pattern = r'^[A-Z]{3}-\d{4}$'
        if not re.match(pattern, v):
            raise ValueError(f'SKU must match format XXX-NNNN: {v}')
        return v

    @model_validator(mode='after')
    def validate_discount(self):
        # TODO: Ensure discount_price < price
        if self.discount_price is not None:
            if self.discount_price ≥ self.price:
                raise ValueError('Discount price must be less than regular price')
        return self

# Test
products = [
    {"name": "Laptop", "sku": "ELC-1001", "price": 999.99, "discount_price": 899.99},
    {"name": "Mouse", "sku": "ELC1001", "price": 25.99},  # Invalid SKU
    {"name": "Keyboard", "sku": "ELC-2001", "price": 50.00, "discount_price": 60.00},  # Invalid discount
]

# TODO: Validate and categorize
```

Part 2 Checkpoint

What You've Built

- Pydantic models for data validation
- Custom validators for complex rules
- Error handling and logging
- Validation reports

Key Takeaways

- Type hints provide automatic validation
- Field constraints enforce data quality
- Custom validators handle complex logic
- Structured error messages aid debugging

Part 3: Label Studio Annotation

Exercise 3.1: Text Classification (20 min)

Task: Set up sentiment analysis project

1. Start Label Studio: `label-studio start`
2. Create new project: "Sentiment Analysis"
3. Import data (create `reviews.json`):

```
[
  {"text": "This product exceeded my expectations! Absolutely love it."},
  {"text": "Terrible quality. Broke after one day. Do not buy."},
  {"text": "It's okay. Nothing special but does the job."},
  {"text": "Best purchase ever! Highly recommend to everyone."},
  {"text": "Waste of money. Very disappointed."}
]
```

4. Configure labeling interface (next slide)
5. Label all reviews

Text Classification Config

```
<View>
  <Header value="Sentiment Analysis"/>

  <Text name="review" value="$text"/>

  <Choices name="sentiment" toName="review" choice="single-radio">
    <Choice value="Positive"/>
    <Choice value="Negative"/>
    <Choice value="Neutral"/>
  </Choices>

  <Rating name="confidence" toName="review" maxRating="5"
          defaultValue="3"/>

  <TextArea name="notes" toName="review"
            placeholder="Optional notes..."
            rows="2"/>
</View>
```

Steps:

Exercise 3.2: Named Entity Recognition (20 min)

Task: Annotate entities in text

Create `articles.json`:

```
[
  {"text": "Apple CEO Tim Cook announced new products in Cupertino on Monday."},
  {"text": "IIT Gandhinagar welcomed 500 students from across India in August."},
  {"text": "The meeting between PM Modi and President Biden took place in Delhi."}
]
```

NER Config:

```
<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>
```

Exercise 3.3: Image Annotation (20 min)

Task: Object detection annotation

1. Collect 5-10 images (or use provided samples)
2. Create project: "Object Detection"
3. Use Rectangle Labels config:

```
<View>
  <Image name="image" value="$image"/>
  <RectangleLabels name="label" toName="image">
    <Label value="Person" background="#FF0000"/>
    <Label value="Vehicle" background="#0000FF"/>
    <Label value="Animal" background="#00FF00"/>
    <Label value="Building" background="#FFFF00"/>
  </RectangleLabels>
</View>
```

4. Draw bounding boxes around objects
5. Export in COCO format

Export and Analysis

Export Options

1. Click Export button in project
2. Choose format:
    - JSON: Full annotations
    - CSV: Tabular format
    - COCO: For object detection
    - YOLO: For YOLO models

Load Exported Data

```
import json

# Load Label Studio JSON export
with open('project-1-at-2024-01-15-10-30.json') as f:
    annotations = json.load(f)

# Extract labels
labels = []
```

Inter-Annotator Agreement

Exercise 3.4: Calculate Cohen's Kappa (20 min)

Scenario: Two annotators labeled the same 20 reviews

```
from sklearn.metrics import cohen_kappa_score, confusion_matrix
import pandas as pd

# Simulated annotations from two annotators
annotator1 = ['pos', 'neg', 'pos', 'neu', 'pos', 'neg', 'pos', 'neu',
              'neg', 'pos', 'pos', 'neu', 'neg', 'pos', 'pos', 'neg',
              'neu', 'pos', 'neg', 'neu']

annotator2 = ['pos', 'neg', 'neu', 'neu', 'pos', 'neg', 'pos', 'pos',
              'neg', 'pos', 'neu', 'neu', 'neg', 'pos', 'pos', 'neg',
              'neu', 'pos', 'neg', 'pos']

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

# Interpretation
if kappa < 0:
    interpretation = "No agreement"
elif kappa < 0.20:
    interpretation = "Slight agreement"
elif kappa < 0.40:
    interpretation = "Fair agreement"
elif kappa < 0.60:
    interpretation = "Moderate agreement"
elif kappa < 0.80:
    interpretation = "Substantial agreement"
else:
    interpretation = "Almost perfect agreement"

print(f"Interpretation: {interpretation}")

# Confusion matrix
cm = confusion_matrix(annotator1, annotator2,
                      labels=['pos', 'neg', 'neu'])
cm_df = pd.DataFrame(cm,
                     index=['pos', 'neg', 'neu'],
                     columns=['pos', 'neg', 'neu'])
```

Fleiss' Kappa for Multiple Annotators

```python
import numpy as np
from statsmodels.stats.inter_rater import fleiss_kappa

# Example: 3 annotators, 5 items, 3 categories
# Format: rows = items, columns = categories
# Values = number of annotators who chose that category

data = np.array([
    [0, 0, 3],  # Item 1: all 3 chose category 3
    [1, 2, 0],  # Item 2: 1 chose cat 1, 2 chose cat 2
    [0, 3, 0],  # Item 3: all 3 chose category 2
    [2, 1, 0],  # Item 4: 2 chose cat 1, 1 chose cat 2
    [0, 0, 3],  # Item 5: all 3 chose category 3
])

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

# Real example: Load from Label Studio export
# and convert to this format

def calculate_fleiss_from_annotations(annotations, num_annotators=3):
    """
    Convert Label Studio annotations to Fleiss' Kappa format
    """
    # Group by item
    items = {}
    for ann in annotations:
        item_id = ann['id']
        label = ann['annotations'][0]['result'][0]['value']['choices'][0]

        if item_id not in items:
            items[item_id] = []
        items[item_id].append(label)

    # Convert to category counts
    categories = ['pos', 'neg', 'neu']
    data = []

    for item_id, labels in items.items():
        counts = [labels.count(cat) for cat in categories]
        data.append(counts)

    return fleiss_kappa(np.array(data))
```

Identify Disagreements

```
import pandas as pd

# Find items where annotators disagree
annotator1 = ['pos', 'neg', 'pos', 'neu', 'pos']
annotator2 = ['pos', 'neg', 'neu', 'neu', 'neg']
texts = [
    "Great product!",
    "Terrible quality.",
    "It's okay, I guess.",
    "Not sure about this.",
    "Absolutely amazing!"
]

# Create DataFrame
df = pd.DataFrame({
    'text': texts,
    'annotator1': annotator1,
    'annotator2': annotator2
})

# Find disagreements
df['disagree'] = df['annotator1'] ≠ df['annotator2']
disagreements = df[df['disagree']]

print("Disagreements:")
print(disagreements)

# These items need review and discussion!
```

Next steps:

Part 3 Checkpoint

What You've Accomplished

- Set up Label Studio projects
- Annotated text, entities, and images
- Exported labeled data
- Calculated inter-annotator agreement
- Identified items needing review

Labeling Best Practices

- Clear, specific guidelines
- Training and calibration
- Regular quality checks
- Resolve disagreements through discussion
- Document edge cases

Part 4: Complete Validation Pipeline

Mini Project (15 minutes)

Task: Build end-to-end pipeline

```
"""
Complete Data Validation and Labeling Pipeline

1. Load scraped data (Week 1)
2. Quick check with jq/csvkit (command line)
3. Validate with Pydantic
4. Save clean data
5. Import to Label Studio
6. Label subset (5-10 items)
7. Export labels
8. Calculate agreement (if multiple annotators)
9. Generate final report
"""

# Template structure
import json
from pydantic import BaseModel, ValidationError
import subprocess

# Step 1: Load data
with open('scraped_data.json') as f:
    raw_data = json.load(f)

# Step 2: Quick jq check
result = subprocess.run(
    ['jq', 'length', 'scraped_data.json'],
    capture_output=True, text=True
)
print(f"Total records: {result.stdout.strip()}")

# Step 3: Validate with Pydantic
class DataModel(BaseModel):
    # Define your schema
    pass

valid_data = []
errors = []

for item in raw_data:
    try:
        validated = DataModel(**item)
        valid_data.append(validated.model_dump())
    except ValidationError as e:
        errors.append({'data': item, 'error': str(e)})

# Step 4: Save clean data
with open('clean_for_labeling.json', 'w') as f:
    json.dump(valid_data, f, indent=2)

print(f"\nValidation complete:")
print(f"  Valid: {len(valid_data)}")
print(f"  Errors: {len(errors)}")
```

Project Rubric

Criteria	Points
Data validation (Pydantic)	30%
Error handling & logging	20%
Label Studio setup	20%
Annotation quality	15%
Agreement calculation	10%
Documentation	5%

Deliverables

1. Validation script with Pydantic models
2. Clean dataset (JSON/CSV)
3. Validation error log
4. Label Studio annotations (exported)

Case Study: Research Paper Metadata

Scenario

Scraped 1000 research papers from arXiv. Need to validate and label.

Validation (Pydantic):

```
from pydantic import BaseModel, HttpUrl, field_validator
from datetime import date

class Paper(BaseModel):
    title: str = Field( ... , min_length=10)
    authors: List[str] = Field( ... , min_items=1)
    abstract: str = Field( ... , min_length=100)
    arxiv_id: str
    pdf_url: HttpUrl
    published: date
    categories: List[str]

    @field_validator('arxiv_id')
    @classmethod
    def validate_arxiv_id(cls, v):
        # Format: YYMM.NNNNN
        import re
```

Case Study: Research Paper Metadata (cont.)

Labeling (Label Studio):

```
<View>
  <Header value="Research Paper Classification"/>

  <Text name="title" value="$title"/>
  <Text name="abstract" value="$abstract"/>

  <Choices name="field" toName="title" choice="multiple">
    <Choice value="Machine Learning"/>
    <Choice value="Computer Vision"/>
    <Choice value="NLP"/>
    <Choice value="Robotics"/>
    <Choice value="Theory"/>
  </Choices>

  <Choices name="quality" toName="title" choice="single-radio">
    <Choice value="High Impact"/>
    <Choice value="Medium Impact"/>
    <Choice value="Low Impact"/>
  </Choices>

  <Number name="relevance" toName="title" min="1" max="5"/>
</View>
```

Case Study: E-commerce Products

Pipeline

```
# 1. Validate product data
class Product(BaseModel):
    name: str
    price: float = Field( ... , gt=0)
    category: str
    rating: float = Field( ... , ge=0, le=5)
    image_url: HttpUrl
    description: str

# 2. jq check for duplicates
# jq 'group_by(.name) | map(select(length > 1))' products.json

# 3. csvstat for price ranges per category
# csvstat -c category,price products.csv

# 4. Label Studio: Categorize products
# - Add missing categories
# - Rate description quality
# - Flag inappropriate content

# 5. Calculate agreement on category labels
# Multiple annotators → Fleiss' Kappa

# 6. Export clean, labeled data for product search ML model
```

Debugging Common Issues

Problem 1: Pydantic Validation Too Strict

```
# Issue: Many valid records failing validation

# Solution: Make fields optional or loosen constraints
from typing import Optional

class Product(BaseModel):
    name: str
    price: Optional[float] = None  # Allow missing prices
    category: Optional[str] = "Uncategorized"  # Default value

    @field_validator('price', mode='before')
    @classmethod
    def handle_missing_price(cls, v):
        if v == '' or v is None:
            return None
        return float(v)
```

Problem 2: Label Studio Import Fails

Best Practices Checklist

Before Validation

- [ ] Explore data with jq/csvstat
- [ ] Understand data distributions
- [ ] Identify common patterns
- [ ] Check for obvious errors

During Validation

- [ ] Start with loose constraints
- [ ] Gradually tighten rules
- [ ] Log all errors with context
- [ ] Keep valid and invalid data separate

After Validation

- [ ] Review error patterns

Tools Summary

Task	Tool	Command/Code
JSON validation	jq	`jq '.' file.json`
CSV stats	csvstat	`csvstat data.csv`
CSV cleaning	csvclean	`csvclean data.csv`
Python validation	Pydantic	`BaseModel` classes
Annotation	Label Studio	Web interface
Agreement	scikit-learn	`cohen_kappa_score()`

Advanced Topics (Optional)

Great Expectations

```
import great_expectations as gx

# Create expectation suite
context = gx.get_context()

# Load data
batch = context.sources.pandas_default.read_csv("data.csv")

# Add expectations
batch.expect_column_values_to_not_be_null("name")
batch.expect_column_values_to_be_between("age", 0, 120)
batch.expect_column_values_to_match_regex("email",
    r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$')

# Validate
results = batch.validate()

# Generate documentation
context.build_data_docs()
```

Homework for Next Week

Assignments

1. **Complete your validation pipeline**
    - Validate all Week 1 scraped data
    - Generate comprehensive error report
    - Document common issues
2. **Label 50+ items**
    - Use Label Studio
    - Export in multiple formats
    - Calculate agreement if working in pairs
3. **Read**
    - Great Expectations documentation
    - Pydantic advanced features

Excellent Work Today!

You've Learned:

- Command-line data inspection (jq, csvkit)
- Python validation with Pydantic
- Data labeling with Label Studio
- Inter-annotator agreement metrics

Next Week:

LLM APIs and Multimodal AI

Questions? Office hours tomorrow 3-5 PM

Quick Feedback

What worked well today?

What was challenging?

What would you like more practice with?

Thank you! See you next week!