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

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
# Returns formatted JSON
```

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

```
# Map over array
```

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
#    5432 Electronics
```

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

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:


```
1 validation error for User
```

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
    pincode: 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", "pincode": "380001"},
        {"street": "456 Park Ave", "city": "Gandhinagar", "pincode": "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 ScrapedProduct(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 = ScrapedProduct(**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
with open('validation_errors.json', 'w') as f:
```

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

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"/>
```

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

```xml
<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": [...]
```

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: Mathematical Foundation

The Problem

Simple percent agreement doesn't account for chance agreement

Example: Two annotators randomly labeling 50/50 classes

- Expected chance agreement: 50%
- Need to measure agreement **beyond chance**

Cohen's Kappa Formula

$$\kappa = \frac{P_o - P_e}{1 - P_e}$$

where:

Cohen's Kappa: Binary Classification Example

Scenario: Spam Detection (2 Annotators, 100 Emails)

Annotator 1: 60 spam, 40 not spam

Annotator 2: 55 spam, 45 not spam

Confusion Matrix

	A2: Spam	A2: Not Spam	Total
A1: Spam	50	10	60
A1: Not	5	35	40
Total	55	45	100

Cohen's Kappa: Step-by-Step Calculation

Step 1: Calculate Observed Agreement (P_o)

$$P_o = \frac{\text{agreements}}{\text{total}} = \frac{50+35}{100} = \frac{85}{100} = 0.85$$

Step 2: Calculate Expected Agreement (P_e)

$$P_e = P(\text{both say spam}) + P(\text{both say not spam})$$

$$P_e = \frac{60}{100} \times \frac{55}{100} + \frac{40}{100} \times \frac{45}{100}$$

$$P_e = 0.60 \times 0.55 + 0.40 \times 0.45 = 0.33 + 0.18 = 0.51$$

Cohen's Kappa: Final Calculation

Step 3: Compute Kappa

$$\kappa = \frac{P_o - P_e}{1 - P_e} = \frac{0.85 - 0.51}{1 - 0.51} = \frac{0.34}{0.49} = 0.694$$

Interpretation: κ = 0.694 indicates **substantial agreement**

Interpretation Scale (Landis & Koch, 1977)

Kappa Value	Interpretation
< 0	No agreement (worse than chance)
0.00-0.20	Slight agreement
0.21-0.40	Fair agreement

Cohen's Kappa: Binary Classification in Python

```
import numpy as np
from sklearn.metrics import cohen_kappa_score, confusion_matrix

# Annotator labels (0 = not spam, 1 = spam)
annotator1 = np.array([1]*50 + [0]*5 + [1]*10 + [0]*35)
annotator2 = np.array([1]*50 + [1]*5 + [0]*10 + [0]*35)

# Calculate kappa
kappa = cohen_kappa_score(annotator1, annotator2)
print(f"Cohen's Kappa: {kappa:.3f}")  # 0.694

# Confusion matrix
cm = confusion_matrix(annotator1, annotator2)
print("Confusion Matrix:")
print(cm)
#     [[35  5]
#      [10 50]]

# Manual calculation for verification
po = (cm[0,0] + cm[1,1]) / cm.sum()
pe = ((cm[0,:].sum() * cm[:,0].sum()) +
      (cm[1,:].sum() * cm[:,1].sum())) / (cm.sum() ** 2)
kappa_manual = (po - pe) / (1 - pe)
print(f"Manual Kappa: {kappa_manual:.3f}")  # 0.694
```

Cohen's Kappa: Multi-Class Example

Scenario: Sentiment Analysis (3 Classes)

Classes: Positive, Negative, Neutral

100 movie reviews, 2 annotators

Confusion Matrix

	A2: Pos	A2: Neg	A2: Neu	Total
A1: Pos	35	2	3	40
A1: Neg	1	28	1	30
A1: Neu	4	5	21	30
Total	40	35	25	100

Multi-Class Kappa: Calculation

Step 1: Observed Agreement

$$P_o = \frac{35 + 28 + 21}{100} = \frac{84}{100} = 0.84$$

Step 2: Expected Agreement

For each class i:

$$P_e(i) = P(A1 = i) \times P(A2 = i)$$

$$P_e = \sum_i P_e(i)$$

$$P_e = \frac{40 \times 40}{100^2} + \frac{30 \times 35}{100^2} + \frac{30 \times 25}{100^2}$$

$$P_e = 0.16 + 0.105 + 0.075 = 0.34$$

Multi-Class Kappa: Result

Step 3: Compute Kappa

$$\kappa = \frac{0.84 - 0.34}{1 - 0.34} = \frac{0.50}{0.66} = 0.758$$

Interpretation: κ = 0.758 indicates **substantial agreement**

```
# Multi-class example
annotator1 = ['pos']*35 + ['neg']*28 + ['neu']*21 + \
             ['pos']*2 + ['neg']*1 + ['neu']*5 + \
             ['pos']*3 + ['neg']*1 + ['neu']*4
annotator2 = ['pos']*35 + ['pos']*2 + ['pos']*3 + \
             ['neg']*28 + ['neg']*1 + ['neg']*5 + \
             ['neu']*21 + ['neu']*1 + ['neu']*4

kappa = cohen_kappa_score(annotator1, annotator2)
print(f"Multi-class Kappa: {kappa:.3f}")  # 0.758
```

Weighted Kappa: For Ordinal Data

When Classes Have Order

Example: Rating scale (1, 2, 3, 4, 5 stars)

- Disagreement 1→2 less severe than 1→5
- Use **weighted kappa**

Linear Weights

$$w_{ij} = 1 - \frac{|i - j|}{k - 1}$$

where k is number of categories

Quadratic Weights (more common)

Weighted Kappa: Example

```python
from sklearn.metrics import cohen_kappa_score

# Star ratings: 1-5
annotator1 = [5, 4, 3, 5, 2, 1, 4, 3, 2, 5]
annotator2 = [4, 4, 3, 5, 3, 2, 4, 2, 2, 4]

# Unweighted kappa
kappa_unweighted = cohen_kappa_score(annotator1, annotator2)
print(f"Unweighted: {kappa_unweighted:.3f}")  # 0.383

# Linear weights
kappa_linear = cohen_kappa_score(annotator1, annotator2,
                                  weights='linear')
print(f"Linear weighted: {kappa_linear:.3f}")  # 0.600

# Quadratic weights (penalizes larger disagreements more)
kappa_quadratic = cohen_kappa_score(annotator1, annotator2,
                                     weights='quadratic')
print(f"Quadratic weighted: {kappa_quadratic:.3f}")  # 0.733
```

Cohen's Kappa for Regression: Challenge

Problem

Kappa is for **categorical** data, not continuous values

Solutions for Regression Agreement

1. **Intraclass Correlation Coefficient (ICC)**

$$\text{ICC} = \frac{\sigma_b^2}{\sigma_b^2 + \sigma_w^2}$$

   - σ_b^2: between-subject variance
   - σ_w^2: within-subject variance (measurement error)

2. **Concordance Correlation Coefficient (CCC)**

ICC for Regression: Python Example

```
import numpy as np
from scipy import stats
import pingouin as pg  # pip install pingouin

# Two annotators rating 10 images on continuous scale [0, 100]
annotator1 = np.array([85, 72, 90, 65, 78, 88, 92, 70, 80, 75])
annotator2 = np.array([82, 75, 88, 68, 76, 85, 90, 73, 78, 77])

# Create dataframe for pingouin
import pandas as pd
data = pd.DataFrame({
    'image': list(range(10)) * 2,
    'rater': ['A1']*10 + ['A2']*10,
    'score': np.concatenate([annotator1, annotator2])
})

# Calculate ICC
icc = pg.intraclass_corr(data=data, targets='image',
                          raters='rater', ratings='score')
print(icc[['Type', 'ICC', 'CI95%']])
# ICC(2,1) ≈ 0.91 indicates excellent agreement

# Pearson correlation (not same as ICC!)
pearson_r = np.corrcoef(annotator1, annotator2)[0, 1]
print(f"Pearson r: {pearson_r:.3f}")  # 0.94
```

Fleiss' Kappa

Multiple Annotators

```python
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

Beyond Classification: Computer Vision Metrics

Different Annotation Tasks

Classification: Assign label to entire image

- Metric: Cohen's Kappa

Object Detection (OD): Locate objects with bounding boxes

- Metric: IoU (Intersection over Union)

Semantic Segmentation: Classify each pixel

- Metric: Pixel-wise IoU, Dice Coefficient

Instance Segmentation: Separate object instances

- Metric: Mask IoU, Average Precision

Intersection over Union (IoU)

Definition

Measures overlap between two regions (boxes or masks)

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}} = \frac{A \cap B}{A \cup B}$$

Properties:

- Range: [0, 1]
- IoU = 1: Perfect overlap
- IoU = 0: No overlap
- Also called **Jaccard Index**

IoU for Bounding Boxes: Visual Example

```
Annotator 1 Box:  ┌──────────┐
                  │          │
                  │    ┌─────┼────┐   Annotator 2 Box
                  │    │XXXX │    │
                  └────┼─────┘    │
                       │          │
                       └──────────┘

    Intersection (XXXX): 20 pixels
    Union (both boxes):  80 pixels
    IoU = 20/80 = 0.25
```

Interpretation:

- IoU > 0.5: Good detection
- IoU > 0.7: Very good detection
- IoU > 0.9: Excellent detection

IoU: Mathematical Calculation

Bounding Box Format

Box = (x_min, y_min, x_max, y_max)

Annotator 1: (10, 10, 50, 50) → 40×40 = 1600 pixels

Annotator 2: (30, 30, 70, 70) → 40×40 = 1600 pixels

Intersection Calculation

```
x_left = max(10, 30) = 30
y_top = max(10, 30) = 30
x_right = min(50, 70) = 50
y_bottom = min(50, 70) = 50

width = max(0, 50 - 30) = 20
height = max(0, 50 - 30) = 20
```

IoU: Union and Final Result

Union Calculation

$$\text{Union} = \text{Area}_1 + \text{Area}_2 - \text{Intersection}$$
$$\text{Union} = 1600 + 1600 - 400 = 2800 \text{ pixels}$$

IoU Result

$$\text{IoU} = \frac{400}{2800} = 0.143$$

Interpretation: IoU = 0.143 indicates **poor agreement** between annotators

- Below 0.5 threshold → need re-annotation or guideline clarification

IoU Implementation: Python

```
def calculate_iou(box1, box2):
    """
    Calculate IoU between two bounding boxes

    Args:
        box1, box2: (x_min, y_min, x_max, y_max)

    Returns:
        iou: float [0, 1]
    """
    # Intersection coordinates
    x_left = max(box1[0], box2[0])
    y_top = max(box1[1], box2[1])
    x_right = min(box1[2], box2[2])
    y_bottom = min(box1[3], box2[3])

    # Intersection area
    if x_right < x_left or y_bottom < y_top:
        return 0.0

    intersection = (x_right - x_left) * (y_bottom - y_top)

    # Box areas
    area1 = (box1[2] - box1[0]) * (box1[3] - box1[1])
    area2 = (box2[2] - box2[0]) * (box2[3] - box2[1])

    # Union
    union = area1 + area2 - intersection

    return intersection / union if union > 0 else 0.0
```

IoU Example: Good vs Bad Agreement

```python
# Good agreement (IoU = 0.84)
box1_good = (100, 100, 200, 200)  # 100x100
box2_good = (110, 110, 210, 210)  # 100x100
iou_good = calculate_iou(box1_good, box2_good)
print(f"Good agreement IoU: {iou_good:.3f}")  # 0.840

# Moderate agreement (IoU = 0.55)
box1_mod = (100, 100, 200, 200)
box2_mod = (130, 130, 230, 230)
iou_mod = calculate_iou(box1_mod, box2_mod)
print(f"Moderate agreement IoU: {iou_mod:.3f}")  # 0.550

# Poor agreement (IoU = 0.14)
box1_poor = (100, 100, 200, 200)
box2_poor = (170, 170, 270, 270)
iou_poor = calculate_iou(box1_poor, box2_poor)
print(f"Poor agreement IoU: {iou_poor:.3f}")  # 0.143

# No overlap (IoU = 0)
box1_none = (100, 100, 200, 200)
box2_none = (250, 250, 350, 350)
iou_none = calculate_iou(box1_none, box2_none)
print(f"No overlap IoU: {iou_none:.3f}")  # 0.000
```

Mean IoU for Multiple Objects

Scenario: Image with 3 Objects

Annotator 1 draws 3 boxes: [box1_a1, box2_a1, box3_a1]

Annotator 2 draws 3 boxes: [box1_a2, box2_a2, box3_a2]

Calculate Mean IoU

```
def mean_iou(boxes_annotator1, boxes_annotator2):
    """
    Calculate mean IoU across multiple objects
    Assumes boxes are in corresponding order
    """
    assert len(boxes_annotator1) == len(boxes_annotator2)

    ious = []
    for box1, box2 in zip(boxes_annotator1, boxes_annotator2):
        iou = calculate_iou(box1, box2)
        ious.append(iou)
```

Semantic Segmentation: Pixel-wise IoU

Task: Classify Every Pixel

Example: Road segmentation

- Each pixel labeled: {road, car, person, background}

Pixel-wise IoU Formula

For class c:

$$\mathrm{IoU}_c = \frac{TP_c}{TP_c + FP_c + FN_c}$$

where:

- TP_c: True positives (both annotators label as class c)

Mean IoU (mIoU) for Segmentation

Average Over All Classes

$$\mathrm{mIoU} = \frac{1}{K}\sum_{c=1}^{K}\mathrm{IoU}_c$$

where K is number of classes

Example: 3 Classes

Class	TP	FP	FN	IoU
Road	850	50	100	0.85
Car	180	20	30	0.78
Person	90	15	10	0.78

Segmentation IoU: Python Implementation

```python
import numpy as np

def segmentation_iou(mask1, mask2, num_classes):
    """
    Calculate IoU for semantic segmentation

    Args:
        mask1, mask2: (H, W) arrays with class indices
        num_classes: number of classes

    Returns:
        class_ious: IoU for each class
        mean_iou: average IoU
    """
    ious = []

    for cls in range(num_classes):
        # Binary masks for current class
        mask1_cls = (mask1 == cls)
        mask2_cls = (mask2 == cls)

        # Intersection and union
        intersection = np.logical_and(mask1_cls, mask2_cls).sum()
        union = np.logical_or(mask1_cls, mask2_cls).sum()

        if union == 0:
            iou = float('nan')  # Class not present in either mask
        else:
            iou = intersection / union

        ious.append(iou)

    # Mean IoU (ignore NaN values)
    mean_iou = np.nanmean(ious)

    return ious, mean_iou
```

Segmentation IoU: Example

```
# Create example segmentation masks (100x100 image, 3 classes)
H, W = 100, 100
num_classes = 3

# Annotator 1 mask
mask_a1 = np.zeros((H, W), dtype=int)
mask_a1[:50, :] = 0    # Top half: class 0 (road)
mask_a1[50:75, :] = 1  # Middle: class 1 (car)
mask_a1[75:, :] = 2    # Bottom: class 2 (person)

# Annotator 2 mask (slight differences)
mask_a2 = np.zeros((H, W), dtype=int)
mask_a2[:48, :] = 0    # Slightly less road
mask_a2[48:77, :] = 1  # Slightly more car
mask_a2[77:, :] = 2    # Slightly less person

# Calculate IoU
class_ious, mean_iou = segmentation_iou(mask_a1, mask_a2, num_classes)

print("Class-wise IoU:")
for cls, iou in enumerate(class_ious):
    print(f"  Class {cls}: {iou:.3f}")
print(f"Mean IoU: {mean_iou:.3f}")

# Output:
# Class 0: 0.960  (road)
# Class 1: 0.897  (car)
# Class 2: 0.920  (person)
# Mean IoU: 0.926
```

Dice Coefficient: Alternative to IoU

Formula

$$\text{Dice} = \frac{2 \times |A \cap B|}{|A| + |B|}$$

Relation to IoU:

$$\text{Dice} = \frac{2 \times \text{IoU}}{1 + \text{IoU}}$$
$$\text{IoU} = \frac{\text{Dice}}{2 - \text{Dice}}$$

Properties

- Range: [0, 1]

Dice vs IoU Comparison

```python
def dice_coefficient(mask1, mask2):
    """Calculate Dice coefficient between two binary masks"""
    intersection = np.logical_and(mask1, mask2).sum()
    return (2.0 * intersection) / (mask1.sum() + mask2.sum())

def iou_to_dice(iou):
    return (2 * iou) / (1 + iou)

def dice_to_iou(dice):
    return dice / (2 - dice)

# Example values
ious = [0.5, 0.7, 0.9, 0.95]
print("IoU  -> Dice")
for iou in ious:
    dice = iou_to_dice(iou)
    print(f"{iou:.2f} -> {dice:.3f}")

# Output:
# 0.50 -> 0.667
# 0.70 -> 0.824
# 0.90 -> 0.947
# 0.95 -> 0.974
```

Object Detection: Matching and mAP

Challenge: Which boxes correspond?

Annotator 1: 5 boxes

Annotator 2: 4 boxes

Solution: Hungarian matching algorithm

- Match boxes to maximize total IoU
- Unmatched boxes penalize agreement

Mean Average Precision (mAP)

Standard metric for object detection datasets:

1. Match boxes using IoU threshold (e.g., 0.5)

Instance Segmentation Agreement

Combines Detection + Segmentation

Each instance has:

- Bounding box
- Pixel-wise mask

Agreement Metrics

1. **Box IoU**: For object localization
2. **Mask IoU**: For pixel-wise segmentation
3. **Combined**: Both must exceed threshold

```
def instance_agreement(boxes1, masks1, boxes2, masks2,
                       box_threshold=0.5, mask_threshold=0.7):
```

Labeling Tools for Different Tasks

Classification

- Label Studio
- Prodigy
- Amazon SageMaker Ground Truth

Object Detection

- LabelImg (bounding boxes)
- CVAT (Computer Vision Annotation Tool)
- VoTT (Visual Object Tagging Tool)

Segmentation

Quality Control for Vision Tasks

Object Detection QC

Check:

1. **Mean IoU** between annotators > 0.7
2. **Object count agreement** (did both find same # objects?)
3. **Class confusion** (mismatched class labels)

```python
def detection_quality_metrics(boxes1, boxes2, labels1, labels2):
    # Object count
    count_diff = abs(len(boxes1) - len(boxes2))

    # Mean IoU (for matched pairs)
    ious = [calculate_iou(b1, b2)
            for b1, b2 in zip(boxes1, boxes2)]
    mean_iou = np.mean(ious) if ious else 0
```

Segmentation Quality Metrics

Boundary Precision

Measure agreement at object boundaries:

$$\text{Boundary F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- Precision: % of predicted boundary pixels near ground truth
- Recall: % of ground truth boundary pixels near prediction

```
from scipy.ndimage import distance_transform_edt

def boundary_f1(mask1, mask2, threshold=2):
    """
    Calculate F1 score for boundary agreement
    threshold: maximum distance (in pixels) for match
    """
    # Find boundaries (edge pixels)
```

Summary: Agreement Metrics by Task

Task	Metric	Range	Formula
Classification	Cohen's κ	[-1, 1]	$(P_o - P_e)/(1 - P_e)$
	Fleiss' κ	[-1, 1]	Multi-rater extension
Regression	ICC	[0, 1]	$\sigma_b^2/(\sigma_b^2 + \sigma_w^2)$
Bounding Box	IoU	[0, 1]	$\|A \cap B\|/\|A \cup B\|$
Segmentation	mIoU	[0, 1]	Mean IoU over classes
	Dice	[0, 1]	$2\|A \cap B\|/(\|A\| + \|B\|)$
Boundary	Boundary F1	[0, 1]	Precision-recall at edges

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

```python
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