Week 2: Data Validation

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

The Data Quality Crisis

Reality Check:

- You collected 1,000 movies from OMDb.
- You try to train a model.
- **Crash!** `ValueError: could not convert string to float: 'N/A'`

Common Issues:

1. **Missing Data**: `NULL`, `NaN`, `""`, `"N/A"`, `"–"`.
2. **Type Mismatches**: Rating `"8.5"` (string) instead of `8.5` (float).
3. **Outliers**: A movie from year `20250` (typo).
4. **Drift**: Last month's data format $\neq$ today's data format.

Data Quality Dimensions

Six dimensions of data quality:

1. **Accuracy**: Values correctly represent the real-world entity
2. **Completeness**: All required data is present
3. **Consistency**: No contradictions (e.g., age=5, birth_year=1950)
4. **Timeliness**: Data is up-to-date and relevant
5. **Validity**: Data conforms to defined formats and constraints
6. **Uniqueness**: No unwanted duplicates

ML Impact: Poor quality in any dimension → degraded model performance.

Types of Data Quality Issues

Structural issues:

- Missing values (nulls, empty strings)
- Wrong data types (strings instead of numbers)
- Invalid formats (dates, emails, phone numbers)

Semantic issues:

- Outliers and anomalies
- Inconsistent units (km vs miles)
- Invalid categories
- Logical contradictions

Temporal issues:

The Cost of Poor Data Quality

Training time:

- Model fails to converge
- Excessive debugging time
- Wasted compute resources

Production time:

- Incorrect predictions
- Model failures and crashes
- Data pipeline failures
- User trust erosion

Rule: 1 hour of validation saves 10 hours of debugging.

The Validation Pipeline

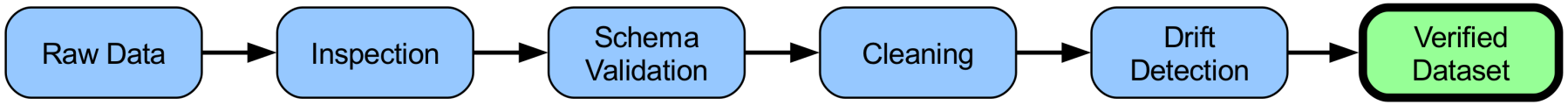

diagram-generators/data_validation_pipeline.py

Tools:

- **Inspect**: `jq` (JSON), `csvkit` (CSV).
- **Schema**: `Pydantic` (Row-level), `Pandera` (Batch-level).
- **Clean**: `pandas`.

Part 1: Inspection (CLI Tools)

Look at your data *before* loading it into Python.

Data Profiling: First Look

Before writing any code, ask:

1. How many records?
2. How many features?
3. What are the data types?
4. How many nulls per column?
5. What's the value distribution?

Tools: `head` , `wc` , `jq` , `csvstat`

Why CLI first?

- Fast for large files (GB-scale)
- No Python dependencies

jq: JSON Power Tool

Find Broken Records:

Movies where `BoxOffice` is missing ("N/A").

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

Check Distribution:

Count movies by Year.

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

Quick Sanity Check:

Are there any ratings > 10?

```
cat movies.json | jq '.[] | select(.imdbRating | tonumber > 10)'
```

jq Advanced Patterns

Compute statistics:

```
# Average rating
cat movies.json | jq '[.[].imdbRating | tonumber] | add/length'

# Find max budget
cat movies.json | jq '[.[].Budget | tonumber] | max'
```

Filter and transform:

```
# High-rated recent movies
cat movies.json | jq '
  .[] |
  select(.Year | tonumber > 2020) |
  select(.imdbRating | tonumber > 8.0) |
  {title: .Title, rating: .imdbRating}
'
```

csvkit: SQL for CSVs

Statistics (`csvstat`):

Gives mean, median, null count, unique values.

```
csvstat movies.csv
```

Output:

```
  1. "Title"
        Type of data:          Text
        Contains null values:  False
        Unique values:         1000

  2. "Year"
        Type of data:          Number
        Contains null values:  False
        Min:                   1920
        Max:                   2024
        Mean:                  1995.3
```

csvkit: More Commands

Query (`csvsql`):

Run SQL directly on CSV!

```
csvsql --query "SELECT Title FROM movies WHERE Rating > 9" movies.csv
```

Cut columns (`csvcut`):

```
csvcut -c Title,Year,Rating movies.csv
```

Look at specific rows (`csvlook`):

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

Join CSVs (`csvjoin`):

```
csvjoin -c movie_id movies.csv reviews.csv
```

Data Profiling with pandas-profiling

Automatic report generation:

```
from ydata_profiling import ProfileReport
import pandas as pd

df = pd.read_csv("movies.csv")
profile = ProfileReport(df, title="Movie Dataset Report")
profile.to_file("report.html")
```

Generates:

- Overview (rows, columns, missing %, duplicates)
- Variable distributions (histograms)
- Correlations (heatmap)
- Missing value patterns

Part 2: Schema Validation (Pydantic)

The Contract: Define what valid data *looks* like.

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

class Movie(BaseModel):
    title: str
    year: int = Field(gt=1888, lt=2030) # Constraints
    rating: float = Field(ge=0, le=10)
    genre: str

    # Custom Validator
    @validator('genre')
    def genre_must_be_valid(cls, v):
        allowed = {'Action', 'Comedy', 'Drama'}
        if v not in allowed:
            raise ValueError(f"Unknown genre: {v}")
        return v
```

Why Pydantic?

1. **Type Coercion**: Converts `"2010"` (str) -> `2010` (int) automatically.
2. **Early Failure**: Errors catch invalid data immediately.
3. **Documentation**: The code *is* the documentation of your data format.

```
try:
    # This will fail (Year out of bounds)
    m = Movie(title="Future", year=3000, rating=5.0, genre="Action")
except ValueError as e:
    print(e)
    # Output: 1 validation error for Movie
    # year: ensure this value is less than 2030
```

Pydantic Field Constraints

Numeric constraints:

- `gt` , `ge` : Greater than, greater or equal
- `lt` , `le` : Less than, less or equal
- `multiple_of` : Must be multiple of N

String constraints:

- `min_length` , `max_length` : Length bounds
- `regex` : Pattern matching
- `strip_whitespace` : Remove leading/trailing spaces

Example:

```
class User(BaseModel):
```

Pydantic Validators

Field validators (validate single field):

```
@validator('email')
def email_must_be_lowercase(cls, v):
    return v.lower()
```

Root validators (validate entire object):

```
@root_validator
def check_consistency(cls, values):
    start = values.get('start_date')
    end = values.get('end_date')
    if start and end and end < start:
        raise ValueError('end_date must be after start_date')
    return values
```

Pre vs Post validation:

Pydantic: Handling Missing Values

Optional fields:

```
from typing import Optional

class Movie(BaseModel):
    title: str  # Required
    budget: Optional[float] = None  # Optional, defaults to None
    release_date: Optional[str]  # Optional, no default (must be provided, can be None)
```

Default values:

```
class Movie(BaseModel):
    title: str
    rating: float = 5.0  # Default if not provided
    is_released: bool = True
```

Field with default factory:

Pydantic: Validation Modes

Strict mode (no coercion):

```
class StrictMovie(BaseModel):
    year: int  # "2010" will fail, must be int

    class Config:
        strict = True
```

Fail fast vs collect all errors:

```
# Default: fail on first error
m = Movie(title="", year="invalid")  # Fails on year

# Collect all errors
class Config:
    validate_all = True
# Now shows errors for both title and year
```

Pydantic: Custom Types

Create reusable validated types:

```
from pydantic import constr, conint, condecimal

# Constrained types
Username = constr(min_length=3, max_length=20, regex="^[a-zA-Z0-9_]+$")
PositiveInt = conint(gt=0)
Rating = condecimal(ge=0, le=10, decimal_places=1)

class User(BaseModel):
    username: Username
    age: PositiveInt
    rating: Rating
```

Benefits: Reuse validation logic across models.

Validation Strategy: Fail-Fast vs Fail-Safe

Fail-Fast (Pydantic default):

- Invalid data raises exception immediately
- Stops pipeline at first error
- **Use when**: Training pipeline, data must be perfect

Fail-Safe:

- Log errors, skip invalid records
- Continue processing
- **Use when**: Production inference, can't reject all data

```
# Fail-safe pattern
valid_movies = []
for raw_data in data:
```

Part 3: Cleaning with Pandas

Handling Missing Data:

1. **Drop**: If label is missing, drop row. `df.dropna(subset=['rating'])`
2. **Impute**: If feature is missing, fill with mean/median. `df.fillna(df.mean())`
3. **Flag**: Create a boolean column `is_missing`.

Type Conversion:

```
# Force numeric, turn errors ('N/A') into NaN
df['rating'] = pd.to_numeric(df['rating'], errors='coerce')
```

Missing Data Strategies

Types of missingness:

1. **MCAR** (Missing Completely At Random): Random, unrelated to data
    - **Strategy**: Impute with mean/median or drop
2. **MAR** (Missing At Random): Related to other observed variables
    - **Strategy**: Model-based imputation (KNN, regression)
3. **MNAR** (Missing Not At Random): Related to unobserved data
    - **Strategy**: Domain-specific imputation or drop

Impact on ML:

- Some algorithms (XGBoost, CatBoost) handle missing values natively
- Others (linear regression, SVM) require imputation

Imputation Techniques

Simple imputation:

```
# Mean for numeric features
df['age'].fillna(df['age'].mean(), inplace=True)

# Median (robust to outliers)
df['income'].fillna(df['income'].median(), inplace=True)

# Mode for categorical
df['country'].fillna(df['country'].mode()[0], inplace=True)

# Forward/backward fill (time series)
df['temperature'].fillna(method='ffill', inplace=True)
```

Advanced imputation:

```
from sklearn.impute import KNNImputer, SimpleImputer
```

Outlier Detection

Statistical methods:

Z-score:

```
from scipy import stats
import numpy as np

z_scores = np.abs(stats.zscore(df['price']))
outliers = df[z_scores > 3]  # More than 3 std dev
```

IQR (Interquartile Range):

```
Q1 = df['price'].quantile(0.25)
Q3 = df['price'].quantile(0.75)
IQR = Q3 - Q1

outliers = df[
    (df['price'] < Q1 - 1.5 * IQR) |
```

Outlier Handling Strategies

1. **Remove outliers**:

```
# Keep only values within 3 std dev
df = df[np.abs(stats.zscore(df['price'])) < 3]
```

2. **Cap outliers** (winsorization):

```
# Cap at 5th and 95th percentile
lower = df['price'].quantile(0.05)
upper = df['price'].quantile(0.95)
df['price'] = df['price'].clip(lower, upper)
```

3. **Transform** (log, sqrt):

```
# Log transform reduces impact of outliers
df['price_log'] = np.log1p(df['price'])
```

Data Type Validation and Conversion

Check data types:

```
df.dtypes  # Show current types
df.info()  # Detailed type information
```

Convert types:

```
# Numeric conversion
df['year'] = pd.to_numeric(df['year'], errors='coerce')

# Datetime conversion
df['release_date'] = pd.to_datetime(df['release_date'], errors='coerce')

# Category (memory efficient)
df['genre'] = df['genre'].astype('category')

# Boolean
df['is_released'] = df['is_released'].astype(bool)
```

Duplicate Detection and Handling

Find duplicates:

```
# Exact duplicates
duplicates = df[df.duplicated()]

# Duplicates based on specific columns
duplicates = df[df.duplicated(subset=['title', 'year'])]

# Keep only first occurrence
df_clean = df.drop_duplicates()

# Keep last occurrence
df_clean = df.drop_duplicates(keep='last')
```

Fuzzy duplicates (similar but not identical):

```
from fuzzywuzzy import fuzz
```

Data Consistency Checks

Logical constraints:

```
# Age should be positive
assert (df['age'] >= 0).all(), "Found negative ages"

# Dates should be in order
assert (df['end_date'] >= df['start_date']).all(), "End before start"

# Percentages should sum to 100
assert df[['cat1', 'cat2', 'cat3']].sum(axis=1).between(99, 101).all()
```

Cross-field validation:

```
# Birth year should match age
df['calculated_age'] = 2024 - df['birth_year']
inconsistent = df[abs(df['age'] - df['calculated_age']) > 1]

# Price should be positive
```

Part 4: Batch Validation (Pandera)

Pydantic checks one object at a time.

Pandera checks the entire DataFrame (statistical checks).

```
import pandera as pa

schema = pa.DataFrameSchema({
    "rating": pa.Column(float, checks=[
        pa.Check.ge(0),
        pa.Check.le(10),
        # Mean rating should be reasonable
        pa.Check.mean_in_range(5, 9)
    ]),
    "year": pa.Column(int, checks=pa.Check.gt(1900)),
})

schema.validate(df)
```

Pandera: Statistical Checks

Built-in checks:

```
schema = pa.DataFrameSchema({
    "price": pa.Column(float, checks=[
        pa.Check.greater_than(0),
        pa.Check.less_than(1000000),
        pa.Check.isin([10.0, 20.0, 30.0]),  # Only these values
        pa.Check.str_startswith("$"),  # For string columns
    ]),
    "category": pa.Column(str, checks=[
        pa.Check.isin(["A", "B", "C"]),
        pa.Check.str_length(1, 10),
    ])
})
```

Custom checks:

```
# Check that 95% of values are within 2 std dev
```

Pandera: DataFrame-Level Checks

Multi-column checks:

```
@pa.check("price", "discount", name="discount_validation")
def check_discount(price, discount):
    return discount < price

schema = pa.DataFrameSchema(
    columns={
        "price": pa.Column(float),
        "discount": pa.Column(float),
    },
    checks=check_discount
)
```

Row-wise checks:

```
def check_row_sum(df):
    return (df[['col1', 'col2', 'col3']].sum(axis=1) == 100).all()
```

Part 5: Data Drift

The Silent Killer.

Data valid today might be invalid tomorrow *statistically*.

Three types of drift:

1. **Schema Drift**: Structure changes
    - Field name changes (`imdbRating` → `rating`)
    - New fields added/removed
    - Type changes (int → float)
2. **Data Drift**: Input distribution changes
    - Feature statistics change (mean, variance)
    - New categories appear

Detecting Schema Drift

Compare schemas:

```
import pandas as pd

# Load reference schema
ref_df = pd.read_csv('training_data.csv')
new_df = pd.read_csv('production_data.csv')

# Check column names
missing_cols = set(ref_df.columns) - set(new_df.columns)
new_cols = set(new_df.columns) - set(ref_df.columns)

if missing_cols:
    raise ValueError(f"Missing columns: {missing_cols}")

# Check data types
for col in ref_df.columns:
    if col in new_df.columns:
        if ref_df[col].dtype != new_df[col].dtype:
            print(f"Type mismatch in {col}: {ref_df[col].dtype} vs {new_df[col].dtype}")
```

Detecting Data Drift

Statistical tests:

Kolmogorov-Smirnov test (continuous variables):

```
from scipy.stats import ks_2samp

# Compare distributions
statistic, p_value = ks_2samp(
    ref_df['rating'],
    new_df['rating']
)

if p_value < 0.05:
    print("Distribution has changed significantly")
```

Chi-square test (categorical variables):

```
from scipy.stats import chi2_contingency
```

Monitoring Data Drift

Track key metrics over time:

```
import json
from datetime import datetime

def log_data_statistics(df, dataset_name):
    stats = {
        "timestamp": datetime.now().isoformat(),
        "dataset": dataset_name,
        "num_rows": len(df),
        "num_cols": len(df.columns),
        "numeric_stats": {
            col: {
                "mean": float(df[col].mean()),
                "std": float(df[col].std()),
                "min": float(df[col].min()),
                "max": float(df[col].max()),
            }
            for col in df.select_dtypes(include=['number']).columns
        },
        "missing_pct": (df.isnull().sum() / len(df) * 100).to_dict()
    }
```

Data Drift Response Strategies

When drift detected:

1. **Retrain model**: Use recent data
2. **Adjust preprocessing**: Update scalers, encoders
3. **Alert humans**: Investigate root cause
4. **Reject data**: If too different, don't use
5. **Adaptive models**: Use online learning

Tools:

- **Evidently AI**: Drift detection and reports
- **Alibi Detect**: Statistical drift tests
- **Great Expectations**: Data validation framework

Advanced Data Validation Topics

Additional theory for production-grade data quality.

Data Quality Metrics and KPIs

Quantify data quality:

Completeness metrics:

```
completeness = (1 - df.isnull().sum() / len(df)) * 100
print(f"Completeness by column:\n{completeness}")

# Overall completeness
overall_completeness = (df.notnull().sum().sum() / df.size) * 100
```

Accuracy metrics (requires ground truth):

```
# For categorical data
accuracy = (df['predicted'] == df['actual']).mean()

# For numeric data (within tolerance)
tolerance = 0.01
accuracy = ((df['predicted'] - df['actual']).abs() < tolerance).mean()
```

Data Quality Scorecard

Aggregate quality score:

```python
def data_quality_score(df, reference_df=None):
    """Calculate overall data quality score (0-100)."""
    scores = {}

    # Completeness (30%)
    completeness = (df.notnull().sum().sum() / df.size) * 100
    scores['completeness'] = completeness * 0.3

    # Uniqueness (20%)
    duplicate_rate = df.duplicated().sum() / len(df)
    uniqueness = (1 - duplicate_rate) * 100
    scores['uniqueness'] = uniqueness * 0.2

    # Validity (30%)
    # Assuming you have validation rules
    valid_rate = 0.95  # Example: 95% of records pass validation
    scores['validity'] = valid_rate * 100 * 0.3

    # Consistency (20%)
    # Cross-field validation pass rate
    consistency_rate = 0.98  # Example
    scores['consistency'] = consistency_rate * 100 * 0.2

    total_score = sum(scores.values())
    return total_score, scores
```

Benford's Law for Fraud Detection

Benford's Law: In many real-world datasets, the first digit follows a logarithmic distribution.

Expected distribution:

- Digit 1: ~30.1%
- Digit 2: ~17.6%
- Digit 3: ~12.5%
- ...
- Digit 9: ~4.6%

Check for manipulation:

```
import numpy as np
import matplotlib.pyplot as plt
```

Data Lineage and Provenance

Data lineage: Track the origin and transformations of data.

Why it matters:

- Debug data quality issues
- Compliance (GDPR, audit trails)
- Reproducibility

Simple lineage tracking:

```
class DataLineage:
    def __init__(self, source):
        self.history = [{"operation": "load", "source": source}]

    def add_operation(self, operation, params=None):
        self.history.append({
            "operation": operation,
            "params": params,
            "timestamp": datetime.now().isoformat()
```

Sampling Strategies for Validation

Problem: Validating 1 billion rows is expensive.

Solution: Statistical sampling

Simple random sampling:

```
# Validate 1% of data
sample = df.sample(frac=0.01, random_state=42)
schema.validate(sample)
```

Stratified sampling (preserve distribution):

```
# Ensure sample has same genre distribution
sample = df.groupby('genre', group_keys=False).apply(
    lambda x: x.sample(frac=0.01, random_state=42)
)
schema.validate(sample)
```

Data Contracts

Data contract: Formal agreement between data producer and consumer.

Contract definition (YAML):

```yaml
dataset: movie_ratings
version: 1.0
owner: data-team@company.com
update_frequency: daily
sla:
  freshness: 24h
  completeness: 95%

schema:
  title:
    type: string
    nullable: false
  year:
    type: integer
    min: 1900
    max: 2030
  rating:
    type: float
    min: 0.0
    max: 10.0
  genre:
    type: string
    allowed_values: [Action, Comedy, Drama, Horror, Sci-Fi]
```

Data Versioning with DVC

Problem: Data changes over time, hard to reproduce experiments.

Solution: Version control for data (like Git for code).

DVC (Data Version Control):

```
# Initialize DVC
dvc init

# Track data file
dvc add data/movies.csv

# Commit DVC metadata (not the actual data)
git add data/movies.csv.dvc .gitignore
git commit -m "Add movies dataset v1"

# Tag version
git tag -a "data-v1.0" -m "Initial dataset"
```

Advanced Regex Patterns for Validation

Email validation (comprehensive):

```
import re

EMAIL_PATTERN = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

def validate_email(email):
    return re.match(EMAIL_PATTERN, email) is not None

# Pydantic integration
from pydantic import validator

class User(BaseModel):
    email: str

    @validator('email')
    def validate_email(cls, v):
        if not re.match(EMAIL_PATTERN, v):
            raise ValueError('Invalid email')
```

Date and Time Validation

Parsing dates safely:

```
from dateutil import parser
from datetime import datetime

def parse_date_robust(date_str):
    """Parse date with multiple format attempts."""
    formats = [
        '%Y-%m-%d',
        '%d/%m/%Y',
        '%m/%d/%Y',
        '%Y-%m-%d %H:%M:%S',
        '%d-%b-%Y',
    ]

    for fmt in formats:
        try:
            return datetime.strptime(date_str, fmt)
        except ValueError:
            continue

    # Fallback to dateutil parser
    try:
        return parser.parse(date_str)
    except:
        raise ValueError(f"Cannot parse date: {date_str}")

# Pydantic validator
from pydantic import BaseModel, validator

class Event(BaseModel):
    event_date: str

    @validator('event_date')
    def validate_date(cls, v):
        try:
            date = parse_date_robust(v)
            # Check date is not in future
            if date > datetime.now():
```

Timezone Handling

Problem: Date without timezone is ambiguous.

Best practices:

```
from datetime import datetime, timezone
import pytz

# Always use timezone-aware datetimes
dt_aware = datetime.now(timezone.utc)

# Convert timezones
eastern = pytz.timezone('US/Eastern')
dt_eastern = dt_aware.astimezone(eastern)

# Validate timezone-aware data
class Event(BaseModel):
    event_time: datetime

    @validator('event_time')
    def ensure_timezone(cls, v):
```

String Normalization

Unicode normalization:

```
import unicodedata

def normalize_string(s):
    """Normalize string for consistent comparison."""
    # NFD: Decompose (é → e + ´)
    # NFC: Compose (e + ´ → é)
    s = unicodedata.normalize('NFKC', s)

    # Case folding (better than lower() for Unicode)
    s = s.casefold()

    # Strip accents
    s = ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

    # Remove extra whitespace
    s = ' '.join(s.split())

    return s

# Example
```

Currency and Unit Validation

Parse currency amounts:

```
import re

def parse_currency(amount_str):
    """Parse currency string to float."""
    # Remove currency symbols and separators
    cleaned = re.sub(r'[^\d.,]', '', amount_str)

    # Handle different decimal separators
    if ',' in cleaned and '.' in cleaned:
        # Determine which is decimal separator
        if cleaned.rindex(',') > cleaned.rindex('.'):
            # European format: 1.234,56
            cleaned = cleaned.replace('.', '').replace(',', '.')
        else:
            # US format: 1,234.56
            cleaned = cleaned.replace(',', '')
    elif ',' in cleaned:
        # Could be European decimal or US thousands
        if len(cleaned.split(',')[-1]) == 2:
            # European decimal: 1234,56
            cleaned = cleaned.replace(',', '.')
        else:
            # US thousands: 1,234
            cleaned = cleaned.replace(',', '')

    return float(cleaned)

# Examples
```

Geospatial Data Validation

Coordinate validation:

```
from pydantic import BaseModel, validator

class Location(BaseModel):
    latitude: float
    longitude: float

    @validator('latitude')
    def validate_latitude(cls, v):
        if not -90 <= v <= 90:
            raise ValueError('Latitude must be between -90 and 90')
        return v

    @validator('longitude')
    def validate_longitude(cls, v):
        if not -180 <= v <= 180:
            raise ValueError('Longitude must be between -180 and 180')
        return v
```

Constraint Satisfaction and Referential Integrity

Foreign key validation:

```
def validate_referential_integrity(child_df, parent_df, foreign_key, primary_key):
    """Ensure all foreign keys exist in parent table."""
    # Find orphaned records
    orphaned = child_df[~child_df[foreign_key].isin(parent_df[primary_key])]

    if len(orphaned) > 0:
        raise ValueError(
            f"Found {len(orphaned)} orphaned records. "
            f"Foreign keys: {orphaned[foreign_key].tolist()}"
        )

# Example: Movies must have valid studio_id
validate_referential_integrity(
    child_df=movies_df,
    parent_df=studios_df,
    foreign_key='studio_id',
    primary_key='id'
)
```

Unit Testing for Data Validation

pytest fixtures for data:

```
import pytest
import pandas as pd

@pytest.fixture
def sample_movies_df():
    """Sample data for testing."""
    return pd.DataFrame({
        'title': ['Inception', 'Interstellar'],
        'year': [2010, 2014],
        'rating': [8.8, 8.6],
        'genre': ['Sci-Fi', 'Sci-Fi']
    })

def test_year_range(sample_movies_df):
    """Test all years are in valid range."""
    assert (sample_movies_df['year'] >= 1900).all()
    assert (sample_movies_df['year'] <= 2030).all()

def test_rating_range(sample_movies_df):
    """Test all ratings are between 0 and 10."""
    assert (sample_movies_df['rating'] >= 0).all()
    assert (sample_movies_df['rating'] <= 10).all()

def test_no_nulls_in_title(sample_movies_df):
```

Incremental Validation

Problem: Validating 1TB of data is slow.

Solution: Incremental validation (only validate new/changed data).

Implementation:

```
import hashlib

def compute_data_hash(df):
    """Compute hash of DataFrame."""
    return hashlib.md5(
        pd.util.hash_pandas_object(df, index=True).values
    ).hexdigest()

def incremental_validate(df, schema, checkpoint_file='validation_checkpoint.json'):
    """Only validate rows that changed since last validation."""
    # Load previous hashes
    try:
        with open(checkpoint_file) as f:
            checkpoint = json.load(f)
    except FileNotFoundError:
        checkpoint = {}

    # Compute current hash
    current_hash = compute_data_hash(df)

    if checkpoint.get('hash') == current_hash:
        print("Data unchanged, skipping validation")
        return True

    # Validate
    try:
```

Validation Performance Optimization

Parallel validation:

```
from concurrent.futures import ProcessPoolExecutor
import numpy as np

def validate_chunk(chunk, schema):
    """Validate a chunk of DataFrame."""
    try:
        schema.validate(chunk)
        return True, None
    except Exception as e:
        return False, str(e)

def parallel_validate(df, schema, n_workers=4):
    """Validate DataFrame in parallel chunks."""
    chunks = np.array_split(df, n_workers)

    with ProcessPoolExecutor(max_workers=n_workers) as executor:
        results = list(executor.map(
            lambda chunk: validate_chunk(chunk, schema),
            chunks
        ))

    # Check if all chunks passed
    all_passed = all(result[0] for result in results)

    if not all_passed:
        errors = [result[1] for result in results if not result[0]]
        raise ValueError(f"Validation failed:\n" + "\n".join(errors))

    return True
```

Data Quality Tools Ecosystem

Open-source tools:

Great Expectations:

```
import great_expectations as ge

df_ge = ge.from_pandas(df)

# Define expectations
df_ge.expect_column_values_to_be_between('year', min_value=1900, max_value=2030)
df_ge.expect_column_values_to_not_be_null('title')
df_ge.expect_column_values_to_be_in_set('genre', ['Action', 'Comedy', 'Drama'])

# Validate
validation_result = df_ge.validate()
print(validation_result)
```

Deepchecks:

Data Validation Best Practices

1. **Fail early**: Validate at ingestion, not at training time.
2. **Be explicit**: Write validation rules as code (not documentation).
3. **Test your tests**: Unit test your validation logic.
4. **Monitor in production**: Continuous validation, not one-time.
5. **Version your schemas**: Track schema changes over time.
6. **Sample intelligently**: Don't validate 100% if 1% is statistically sufficient.
7. **Profile first**: Understand your data before writing rules.
8. **Separate concerns**:
    - Row-level validation → Pydantic
    - Batch validation → Pandera

Production Validation Pipeline

Complete end-to-end example:

```
from typing import List
import logging

class DataValidationPipeline:
    def __init__(self, schema, drift_detector=None):
        self.schema = schema
        self.drift_detector = drift_detector
        self.logger = logging.getLogger(__name__)

    def validate(self, df, reference_df=None):
        """Run full validation pipeline."""
        results = {
            'passed': False,
            'errors': [],
            'warnings': [],
            'stats': {}
        }

        # 1. Schema validation
        try:
            self.schema.validate(df)
            self.logger.info("Schema validation passed")
        except Exception as e:
            results['errors'].append(f"Schema validation failed: {e}")
            return results

        # 2. Quality checks
        completeness = (1 - df.isnull().sum() / len(df))
        if (completeness < 0.95).any():
            results['warnings'].append(
                f"Low completeness: {completeness[completeness < 0.95].to_dict()}"
            )

        # 3. Drift detection
        if reference_df is not None and self.drift_detector:
            drift_detected = self.drift_detector.detect(df, reference_df)
            if drift_detected:
                results['warnings'].append("Data drift detected")

        # 4. Summary stats
        results['stats'] = {
            'num_rows': len(df),
            'num_cols': len(df.columns),
            'completeness': completeness.to_dict(),
            'duplicates': df.duplicated().sum()
        }

        results['passed'] = len(results['errors']) == 0
        return results

# Usage
```

Summary: The Checklist

Before training a model, ask:

1. [ ] **Schema**: Do all fields exist with correct types? (Pydantic)
2. [ ] **Nulls**: How are missing values handled?
3. [ ] **Ranges**: Are numbers within physical bounds? (Age > 0)
4. [ ] **Duplicates**: Are primary keys unique?
5. [ ] **Stats**: Does the distribution look normal? (Pandera)
6. [ ] **Consistency**: Do cross-field constraints hold?
7. [ ] **Lineage**: Can you trace data back to its source?
8. [ ] **Drift**: Has the distribution changed significantly?
9. [ ] **Quality score**: Is overall quality above threshold (>90%)?
10. [ ] **Testing**: Are validation rules tested?