Data Preparation & Ethical Data Handling

AI Masters Capstone Project - Presentation 2

Jonathan Agustin

November 2024

What We'll Cover Today

- Ethical data handling: fundamentals and importance
- Automated preprocessing: cleaning, transforming, validating
- Privacy & compliance: embedding regulations into data pipelines
- Effective validation sets (Thomas, 2017): beyond naive splits
- Detecting & mitigating bias before training

Goal: robust, transparent, ethical data pipelines for trustworthy ML.

Why Ethical Data Handling Matters

- Data = human lives and opportunities
- Trust & credibility: keys to long-term AI adoption
- Fairness, privacy, compliance: protect users and organizations

Ethical standards underpin the entire ML lifecycle.

Automated Data Preprocessing

- Standardize cleaning: handle missing values, outliers, inconsistencies
- Consistent transformations: encoding, scaling, normalization
- Transparency: reproducible steps reduce hidden biases

Automation yields a trustworthy data layer for ML.

Data Loading and Column Types

```
import pandas as pd
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# Load and identify column types
df = pd.read_csv("raw_data.csv")

# Separate numeric and categorical columns
numeric_cols = df.select_dtypes(include=['float','int']).columns
categorical_cols = df.select_dtypes(include=['object']).columns

# Initial data check
print(f"Numeric columns: {len(numeric_cols)}")
print(f"Categorical columns: {len(categorical_cols)}")
```

Clear separation of data types enables proper handling.

Processing Numeric Features

```
# Handle missing values in numeric columns
imputer = SimpleImputer(strategy='mean')
df[numeric_cols] = imputer.fit_transform(df[numeric_cols])

# Scale numeric features
scaler = StandardScaler()
df[numeric_cols] = scaler.fit_transform(df[numeric_cols])

# Verify no missing values remain
print("Missing values after imputation:",
      df[numeric_cols].isnull().sum().sum())
```

Consistent scaling ensures fair model training.

Processing Categorical Features

```
# Encode categorical variables
encoder = OneHotEncoder(
    sparse_output=False,
    handle_unknown='ignore'
)
# Transform and create new dataframe
encoded = encoder.fit_transform(df[categorical_cols])
encoded_df = pd.DataFrame(
    encoded,
    columns=encoder.get_feature_names_out(categorical_cols)
)
# Combine with numeric features
df = pd.concat([df.drop(columns=categorical_cols), encoded_df],
               axis=1)
```

Careful encoding preserves categorical information.

Setting Up Data Validation

```
import pandera as pa
from pandera.typing import DataFrame, Series

class InputSchema(pa.SchemaModel):
    age: Series[int] = pa.Field(ge=0, le=120)
    income: Series[float] = pa.Field(ge=0)

    class Config:
        strict = True
        coerce = True
```

Schema definitions protect against invalid data.

Implementing Validation Checks

```python
def validate_dataset(df: pd.DataFrame) -> pd.DataFrame:
    try:
        validated_df = InputSchema.validate(df)
        print("Validation passed!")
        return validated_df
    except pa.errors.SchemaError as e:
        print("Validation failed:", str(e))
        raise RuntimeError("Data validation failed")

# Run validation
validated_df = validate_dataset(df)
```

Early validation prevents downstream issues.

Time-Based Training Splits

```
# Define time-based splits
training_cutoff = '2017-07-31'
validation_start = '2017-08-01'
validation_end = '2017-08-15'

# Create splits
df_train = df[df['date'] <= training_cutoff]
df_val = df[
    (df['date'] > training_cutoff) &
    (df['date'] <= validation_end)
]

print(f"Training samples: {len(df_train)}")
print(f"Validation samples: {len(df_val)}")
```

Time-based splits reflect real-world conditions.

Entity-Based Training Splits

```python
# Get list of known entities
known_entities = ['id1', 'id2', 'id3']  # Example IDs

# Split based on entities
df_train = df[df['entity_id'].isin(known_entities)]
df_val = df[~df['entity_id'].isin(known_entities)]

# Verify no overlap
assert len(set(df_train['entity_id']) &
           set(df_val['entity_id'])) == 0

print(f"Unique entities in train: {df_train['entity_id'].nunique()}")
print(f"Unique entities in val: {df_val['entity_id'].nunique()}")
```

Entity-based validation tests true generalization.

Privacy Protection: Pseudonymization

```
import hashlib
from typing import List

def pseudonymize_column(series: pd.Series) -> pd.Series:
    return series.apply(lambda x:
        hashlib.sha256(str(x).encode()).hexdigest())

# Columns that need pseudonymization
sensitive_cols = ['user_id', 'email', 'phone']

# Apply pseudonymization
for col in sensitive_cols:
    if col in df.columns:
        df[f"{col}_hashed"] = pseudonymize_column(df[col])
        df = df.drop(columns=[col])
```

Setting Up Bias Detection

```
from aif360.datasets import BinaryLabelDataset
from aif360.metrics import BinaryLabelDatasetMetric

# Create dataset with protected attributes
dataset = BinaryLabelDataset(
    df=df,
    label_names=['decision'],
    protected_attribute_names=['gender', 'race']
)

# Define privileged and unprivileged groups
privileged_groups = [{'gender': 1}]
unprivileged_groups = [{'gender': 0}]
```

Identifying bias is the first step to addressing it.

Measuring and Mitigating Bias

```
from aif360.algorithms.preprocessing import Reweighing
# Measure initial bias
metrics = BinaryLabelDatasetMetric(
    dataset,
    unprivileged_groups=unprivileged_groups,
    privileged_groups=privileged_groups
)
# Print initial disparities
print("Disparate impact:", metrics.disparate_impact())
print("Statistical parity difference:", metrics.statistical_parity_difference())
# Apply reweighing
reweighing = Reweighing(
    unprivileged_groups=unprivileged_groups,
    privileged_groups=privileged_groups
)
transformed_dataset = reweighing.fit_transform(dataset)
```

A Holistic Data Strategy

- Combine cleaning, validation, privacy, and bias mitigation
- Document processes for transparency and audits
- Monitor and update as data evolves
- Build trust through consistent ethical practices

A comprehensive approach ensures lasting ethical compliance.

Implementing the Complete Pipeline

```python
def ethical_data_pipeline(raw_df: pd.DataFrame) -> pd.DataFrame:
    # 1. Initial validation
    validated_df = validate_dataset(raw_df)

    # 2. Privacy protection
    protected_df = protect_privacy(validated_df)

    # 3. Preprocessing
    processed_df = preprocess_data(protected_df)

    # 4. Bias detection and mitigation
    final_df = mitigate_bias(processed_df)

    return final_df
```

A structured pipeline ensures consistent processing.

Monitoring and Maintenance

```python
def monitor_pipeline_health(df: pd.DataFrame) -> None:
    # Track data distribution
    log_distribution_metrics(df)

    # Check for drift
    drift_detected = check_for_drift(df)

    # Validate fairness metrics
    fairness_metrics = compute_fairness_metrics(df)

    # Alert if thresholds exceeded
    if needs_attention(drift_detected, fairness_metrics):
        trigger_review_process()
```

Continuous monitoring maintains ethical standards.

Next Steps

- Next presentation: Automating Model Training & Ethical Evaluation
- Apply these data preparation principles to model development
- Maintain focus on fairness and accountability
- Prepare for deployment considerations

Strong data foundations enable ethical model development.

References

- Thomas, R. (2017). *How (and why) to create a good validation set*. https://rachel.fast.ai/posts/2017-11-13-validation-sets/
- *aif360* toolkit: https://github.com/Trusted-AI/AIF360
- *pandera* library: https://pandera.readthedocs.io/
- GDPR guidelines: https://gdpr.eu/