Data Wrangling

Comprehensive Notes on Data Wrangling and Transformations

Data wrangling and transformations are essential steps in preparing raw data for analysis and modeling. These processes involve cleaning, structuring, and enriching data to make it usable and meaningful for machine learning and statistical analysis.

We'll cover **data wrangling**, **data cleaning**, **feature engineering**, and **data transformations** comprehensively.

---

1. What is Data Wrangling?

Data wrangling is the process of cleaning, restructuring, and enriching raw data into a desired format for better decision-making in data analysis. It includes several tasks such as handling missing values, removing duplicates, correcting data types, and applying transformations.

Key Steps in Data Wrangling

- **Data Cleaning**: Handling missing values, removing duplicates, correcting errors.
- **Data Transformation**: Normalization, scaling, and encoding data.
- **Feature Engineering**: Creating new features, transforming existing ones, or reducing dimensionality.
- **Data Integration**: Merging multiple data sources into a single cohesive dataset.
- **Data Validation**: Ensuring data accuracy and consistency.

---

2. Data Cleaning

2.1 Handling Missing Values

Missing data can occur due to various reasons (e.g., data entry errors, sensor failures). There are several strategies to handle missing values:

Strategies for Handling Missing Data

- **Removing Missing Values**:

```
# Remove rows with missing values
df.dropna(inplace=True)

# Remove columns with more than 30% missing values
df.dropna(axis=1, thresh=int(0.7 * len(df)), inplace=True)
```

- **Filling Missing Values**:
  - **Fill with Mean, Median, or Mode**:

```
df['column'].fillna(df['column'].mean(), inplace=True)
df['column'].fillna(df['column'].median(), inplace=True)
df['column'].fillna(df['column'].mode()[0], inplace=True)
```

  - **Forward/Backward Fill**:

```
df.fillna(method='ffill', inplace=True)  # Forward fill
df.fillna(method='bfill', inplace=True)  # Backward fill
```

2.2 Removing Duplicates

Removing duplicates helps to ensure that your dataset is clean and consistent.

```
# Drop duplicate rows
df.drop_duplicates(inplace=True)
```

2.3 Correcting Data Types

Data types may need to be corrected to align with your analysis needs:

```
# Convert columns to numeric data type
df['column'] = pd.to_numeric(df['column'], errors='coerce')

# Convert to datetime
df['date_column'] = pd.to_datetime(df['date_column'])
```

---

3. Data Transformations

Data transformations involve modifying the structure of your data to improve model performance and interpretability. This includes scaling, normalization, encoding, and more.

3.1 Scaling and Normalization

Why Normalize or Scale Data?

- Many machine learning algorithms (e.g., KNN, SVM) are sensitive to the scale of data.
- Normalization ensures that all features contribute equally to the distance metric.

Standardization (Z-score normalization)

Standardization rescales data to have a mean of 0 and a standard deviation of 1.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df_scaled = scaler.fit_transform(df[['feature1', 'feature2']])
```

Min-Max Normalization

Normalizes data to a range between 0 and 1.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
df_normalized = scaler.fit_transform(df[['feature1', 'feature2']])
```

Robust Scaling

Useful for datasets with outliers.

```
from sklearn.preprocessing import RobustScaler

scaler = RobustScaler()
df_robust = scaler.fit_transform(df[['feature1', 'feature2']])
```

---

4. Encoding Categorical Variables

Machine learning models require numerical input, so categorical features need to be encoded.

4.1 One-Hot Encoding

Converts categorical variables into a series of binary columns.

```
df_encoded = pd.get_dummies(df, columns=['categorical_column'],
drop_first=True)
```

4.2 Label Encoding

Assigns an integer to each category.

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['encoded_column'] = le.fit_transform(df['categorical_column'])
```

4.3 Target Encoding (Mean Encoding)

Replaces each category with the mean of the target variable.

```
# Calculate mean encoding
encoding = df.groupby('category_column')['target'].mean()
df['encoded_column'] = df['category_column'].map(encoding)
```

---

5. Feature Engineering

5.1 Binning

Convert numerical values into discrete bins or categories.

```
df['binned'] = pd.cut(df['age'], bins=[0, 18, 35, 60, 100], labels=['Child',
'Young', 'Adult', 'Senior'])
```

5.2 Polynomial Features

Create interaction terms or polynomial features to capture non-linear relationships.

```
from sklearn.preprocessing import PolynomialFeatures
```

```
poly = PolynomialFeatures(degree=2, include_bias=False)
df_poly = poly.fit_transform(df[['feature1', 'feature2']])
```

---

6. Log, Square Root, and Box-Cox Transformations

6.1 Log Transformation

Used to reduce skewness in data.

```
df['log_feature'] = np.log(df['feature'] + 1)
```

6.2 Square Root Transformation

Useful for reducing the effect of large values.

```
df['sqrt_feature'] = np.sqrt(df['feature'])
```

6.3 Box-Cox Transformation

Transforms non-normal dependent variables into a normal shape.

```
from scipy import stats

df['boxcox_feature'], _ = stats.boxcox(df['feature'] + 1)
```

---

7. Feature Selection Techniques

7.1 Correlation Matrix

Useful for identifying highly correlated features.

```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 8))
```

```
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title("Correlation Matrix")
plt.show()
```

7.2 Mutual Information

Measure the dependency between variables.

```
from sklearn.feature_selection import mutual_info_classif

mi = mutual_info_classif(X, y)
mi_series = pd.Series(mi, index=X.columns)
mi_series.sort_values(ascending=False)
```

---

8. Putting It All Together

Data Wrangling Pipeline

Below is an end-to-end data wrangling pipeline:

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, LabelEncoder

# Load dataset
df = pd.read_csv('ugransome.csv')

# 1. Handle missing values
df.fillna(method='ffill', inplace=True)

# 2. Remove duplicates
df.drop_duplicates(inplace=True)

# 3. Convert data types
df['date'] = pd.to_datetime(df['date'])

# 4. Encode categorical variables
le = LabelEncoder()
df['sentiment_encoded'] = le.fit_transform(df['sentiment'])

# 5. Feature scaling
scaler = StandardScaler()
```

```
df[['score', 'sentiment_encoded']] = scaler.fit_transform(df[['score',
'sentiment_encoded']])

# 6. Feature engineering (log transformation)
df['log_score'] = np.log(df['score'] + 1)

# 7. One-hot encoding for categorical columns
df = pd.get_dummies(df, columns=['nature'], drop_first=True)

# 8. Split into features and target
X = df.drop('threat', axis=1)
y = df['threat']

# View processed dataset
print(df.head())
```

Alright! Let's dive deeper into the advanced techniques you can use for data preprocessing, dimensionality reduction, feature selection, and handling imbalanced datasets.

---

1. Dimensionality Reduction Techniques

Dimensionality reduction is used to reduce the number of features in a dataset while retaining as much information as possible. This helps to improve model performance, reduce overfitting, and decrease computation time.

1.1 Principal Component Analysis (PCA)

PCA is a popular technique for reducing dimensionality by projecting the data onto a lower-dimensional subspace that retains the maximum variance.

Steps for PCA:

- Standardize the dataset.
- Apply PCA to transform the data.
- Choose the number of components that explain a significant amount of variance.

Code Example:

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Standardize the dataset
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA
pca = PCA(n_components=2)  # Reduce to 2 components
X_pca = pca.fit_transform(X_scaled)

# Plot explained variance ratio
plt.plot(range(1, len(pca.explained_variance_ratio_) + 1),
pca.explained_variance_ratio_)
plt.title("Explained Variance by Components")
plt.xlabel("Principal Components")
plt.ylabel("Explained Variance")
plt.show()
```

When to use PCA:

- When you have high-dimensional data.
- To reduce noise in the dataset.
- When features are correlated.

1.2 t-SNE (t-Distributed Stochastic Neighbor Embedding)

t-SNE is used for visualizing high-dimensional data by reducing it to 2 or 3 dimensions while preserving local structure.

Code Example:

```
from sklearn.manifold import TSNE
import seaborn as sns

# Apply t-SNE
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X_scaled)

# Plot t-SNE results
plt.figure(figsize=(10, 6))
```

```
sns.scatterplot(x=X_tsne[:, 0], y=X_tsne[:, 1], hue=y, palette='viridis')
plt.title("t-SNE Visualization")
plt.show()
```

When to use t-SNE:

- When you want to visualize high-dimensional data.
- For clustering and anomaly detection.
- When you have labeled data and want to observe class separability.

1.3 Linear Discriminant Analysis (LDA)

LDA is a supervised technique that reduces dimensions while maximizing class separability.

Code Example:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Apply LDA
lda = LinearDiscriminantAnalysis(n_components=2)
X_lda = lda.fit_transform(X, y)

# Plot LDA results
plt.figure(figsize=(10, 6))
sns.scatterplot(x=X_lda[:, 0], y=X_lda[:, 1], hue=y, palette='coolwarm')
plt.title("LDA Visualization")
plt.show()
```

---

2. Feature Selection Techniques

Feature selection helps to identify the most important features that contribute to the prediction, thus improving model performance.

2.1 Recursive Feature Elimination (RFE)

RFE is a technique that recursively removes the least important features based on the model's coefficients.

Code Example:

```
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier

# Initialize the model
model = RandomForestClassifier()

# Apply RFE
rfe = RFE(model, n_features_to_select=5)
X_rfe = rfe.fit_transform(X, y)

# Check selected features
selected_features = X.columns[rfe.support_]
print("Selected features:", selected_features)
```

2.2 Mutual Information

Mutual information measures the dependency between variables and helps to identify which features are most important.

Code Example:

```
from sklearn.feature_selection import mutual_info_classif

mi = mutual_info_classif(X, y)
mi_series = pd.Series(mi, index=X.columns)
mi_series.sort_values(ascending=False).plot(kind='bar')
plt.title("Feature Importance based on Mutual Information")
plt.show()
```

2.3 SelectKBest

SelectKBest selects the top k features based on statistical tests.

Code Example:

```
from sklearn.feature_selection import SelectKBest, f_classif

selector = SelectKBest(score_func=f_classif, k=5)
X_kbest = selector.fit_transform(X, y)

# Check selected features
selected_features = X.columns[selector.get_support()]
print("Selected features using SelectKBest:", selected_features)
```

3. Handling Imbalanced Datasets

Imbalanced datasets are common in real-world applications, where one class significantly outnumbers the other(s). This can lead to biased models.

3.1 SMOTE (Synthetic Minority Over-sampling Technique)

SMOTE generates synthetic samples for the minority class to balance the dataset.

Code Example:

```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

# Check class distribution after SMOTE
print("Class distribution after SMOTE:",
pd.Series(y_resampled).value_counts())
```

3.2 Random Over-Sampling and Under-Sampling

- **Over-Sampling**: Duplicate examples in the minority class.
- **Under-Sampling**: Remove examples from the majority class.

Code Example for Random Under-Sampling:

```
from imblearn.under_sampling import RandomUnderSampler

undersampler = RandomUnderSampler(random_state=42)
X_under, y_under = undersampler.fit_resample(X, y)

print("Class distribution after under-sampling:",
pd.Series(y_under).value_counts())
```

4. Advanced Data Augmentation for Time-Series and Image Data

4.1 Time-Series Augmentation

For time-series data, techniques like jittering, scaling, or time-warping can be applied to generate synthetic data.

Code Example: Jittering

```
def add_jitter(data, noise_level=0.01):
    noise = np.random.normal(loc=0.0, scale=noise_level, size=data.shape)
    return data + noise

augmented_data = add_jitter(time_series_data)
```

4.2 Image Data Augmentation

Data augmentation techniques like rotation, flipping, and zooming help to improve generalization in deep learning models.

Code Example: Image Augmentation using Keras

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

datagen.fit(x_train)
```

---

5. Putting It All Together: Comprehensive Data Preparation Pipeline

End-to-End Example Using UG Ransom Dataset

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.decomposition import PCA
from imblearn.over_sampling import SMOTE

# Load dataset
df = pd.read_csv('ugransome.csv')

# Step 1: Handle Missing Values
df.fillna(method='ffill', inplace=True)

# Step 2: Remove Duplicates
df.drop_duplicates(inplace=True)

# Step 3: Encode Categorical Variables
label_encoder = LabelEncoder()
df['sentiment_encoded'] = label_encoder.fit_transform(df['sentiment'])

# Step 4: Scale Features
scaler = StandardScaler()
X = df[['score', 'sentiment_encoded']]
X_scaled = scaler.fit_transform(X)
y = df['nature']

# Step 5: Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Step 6: Handle Imbalance using SMOTE
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_pca, y)

# Step 7: Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled,
test_size=0.2, random_state=42)

print("Data preparation pipeline completed.")
```

Visualization

```
import matplotlib.pyplot as plt
import seaborn as sns
```

```
sns.scatterplot(x=X_resampled[:, 0], y=X_resampled[:, 1], hue=y_resampled,
palette='viridis')
plt.title("PCA Reduced Data (After SMOTE)")
plt.show()
```