Scikit-Learn: Data Preprocessing

Prof. Pranay Kumar Saha

April 15, 2025

Overview

- Handling Missing Data
- Feature Scaling
- Outlier Detection
- Dimensionality Reduction

Generating Sample Dataset

```
import numpy as np
import pandas as pd

np.random.seed(42)
data = np.random.randn(500, 4) * [10, 5, 1, 0.5] + [50, 30, 10, 5]
df = pd.DataFrame(data,
        columns=['Feature1', 'Feature2', 'Feature3', 'Feature4'])
```

df.head()

	Feature1	Feature2	Feature3	Feature4
0	54.967142	29.308678	10.647689	5.761515
1	47.658466	28.829315	11.579213	5.383717
2	45.305256	32.712800	9.536582	4.767135
3	52.419623	20.433599	8.275082	4.718856
4	39.871689	31.571237	9.091976	4.293848

Handling Missing Data: Imputation

- **Mean Imputation**: For each feature (column), find the average of all non-missing values and replace missing entries with that average.

$$x_{\text{imputed}} = \frac{1}{N} \sum_{i=1}^{N} x_i \quad \text{(excluding missing values)}$$

- **Median Imputation**: Replace missing values with the median of the feature.
- **Mode Imputation (Most Frequent)**: Replace missing values with the most common (categorical or discrete) value in the feature.

Handling Missing Data

```
from sklearn.impute import SimpleImputer

df.iloc[10:15, 2] = np.nan  # Introduce missing values in 'Feature3'

# 2. Create an imputer instance (mean strategy)
imputer = SimpleImputer(strategy='mean')

# 3. Fit the imputer to Feature3 and transform
df[['Feature3']] = imputer.fit_transform(df[['Feature3']])
```

How StandardScaler Works

StandardScaler transforms each feature to have:

- **Mean** = 0
- **Standard Deviation** = 1

Formula: Each value x is transformed using:

$$z = \frac{x - \mu}{\sigma}$$

where:

- μ is the mean of the feature
- σ is the standard deviation of the feature

Effect: - Features are centered around zero.
- Useful for models that assume normally distributed data (e.g., logistic regression, SVM).
- Prevents dominance of large-scale features.

StandardScaler (Mean 0, Std 1)

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_data = scaler.fit_transform(df)
df_scaled = pd.DataFrame(scaled_data, columns=df.columns)
```

MinMaxScaler (Range 0 to 1)

```
from sklearn.preprocessing import MinMaxScaler

minmax_scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data_mm = minmax_scaler.fit_transform(df)
df_minmax = pd.DataFrame(scaled_data_mm, columns=df.columns)
```

```
df_minmax.head()
```

	Feature1	Feature2	Feature3	Feature4
0	0.507007	-0.201021	0.614230	1.446150
1	-0.284059	-0.295971	1.549968	0.704719
2	-0.538762	0.473250	-0.501903	-0.505330
3	0.231272	-1.958951	-1.769110	-0.600078
4	-1.126872	0.247135	-0.948521	-1.434160

Adding Outlier into the data

```
outliers = np.array([[200, 300, 20, 10],
                     [180, 280, 25, 7],
                     [160, -100, -5, 2]])
 outliers_df = pd.DataFrame(outliers, columns=df.columns)
 df = pd.concat([df, outliers_df], ignore_index=True)
```

Isolation Forest

- Anomaly detection, also known as outlier detection, is the process of identifying data points that deviate significantly from the norm or expected behavior within a dataset
- Isolation Forest is a powerful and efficient algorithm specifically designed for anomaly detection.
- The Isolation Forest is based on the principle of recursively partitioning data until each data point is isolated. This partitioning is done randomly.

Steps1:Random Partitioning

it builds multiple isolation trees (iTrees). For each iTree:

1. A random subsample of the data is selected (without replacement).
2. The tree is constructed by recursively selecting a random feature and a random split value within the range of that feature.
3. This process continues until each data point in the subsample is isolated in its own leaf node, or a predefined maximum tree depth is reached.

Step2: Path Length

1. The key idea is that anomalies will require fewer random partitions to be isolated compared to normal instances.
2. The path length of a data point in an iTree is the number of edges traversed from the root node to the terminal (leaf) node where the point is isolated.
3. Shorter path lengths indicate higher susceptibility to isolation, suggesting a higher likelihood of being an anomaly.

Step3: Averaging Over Multiple Trees

- **Forest of iTrees:** Because the partitioning is random, the algorithm constructs a forest (often 100+ iTrees).
- **Robust Isolation Measure:** The average path length of a data point across all trees provides a more stable indicator of how easily it can be isolated.
- **Why It Matters:** Averaging over many trees reduces variance from any single random partition, improving anomaly detection accuracy.

Anomaly Score

- The anomaly score $s(x, n)$ quantifies how anomalous a data point x is, given a sample size n.

Formula:

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}}$$

Components:

- $h(x)$: Path length of x in a single iTree.
- $E(h(x))$: Average path length of x across all iTrees.
- $c(n)$: Normalization factor approximating the average path length of an unsuccessful search in a BST:

Anomaly Score (contd...)

- $c(n)$: Approximates the average path length of an unsuccessful search in a BST.

$$c(n) = 2H(n-1) - \frac{2(n-1)}{n}$$

where the harmonic number $H(i)$ can be approximated as:

$$H(i) \approx \ln(i) + 0.5772156649$$

Interpretation:

- $s(x, n) \approx 1$: Strong anomaly.
- $s(x, n) \ll 0.5$: Likely normal instance.
- $s(x, n) \approx 0.5$: No clear anomaly within the dataset.

Hyperparameter

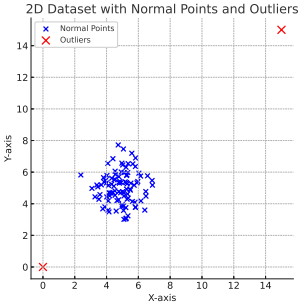
The main hyperparameters of Isolation Forest are:

- **Number of Trees (n_estimators):** The number of iTrees to build in the forest. More trees generally lead to more reliable results but increase computational cost.
- **Subsampling Size (max_samples):** The number of data points to use for building each iTree. Smaller subsamples can lead to faster training and can sometimes improve performance by reducing the ”swamping” and ”masking” effects (where normal instances obscure anomalies or vice-versa).

Example

Consider a simple 2D dataset with a cluster of normal points and a few isolated outliers.

1. **Data Generation:** Imagine a dataset most of the coordinates (5, 5) and some few outlier points far away, such as (15, 15) and (0, 0).

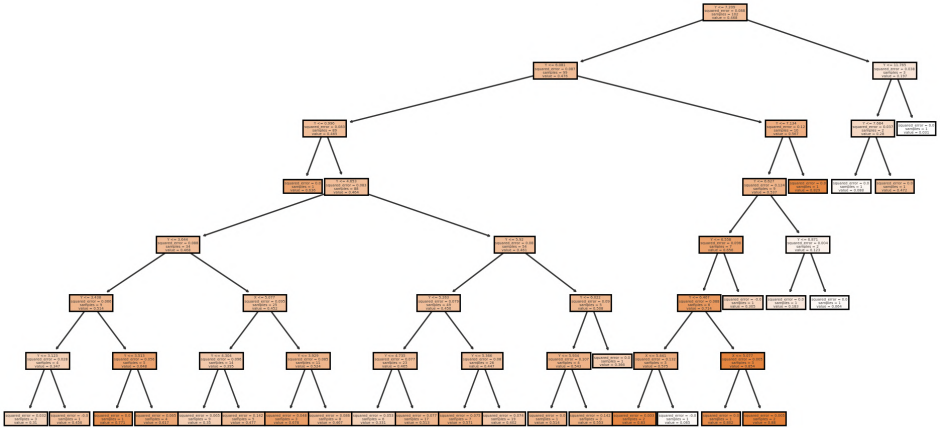


Example

2. **Tree Construction:** The Isolation Forest algorithm would randomly select features (either the x-coordinate or the y-coordinate) and split values.
3. **Isolation:** The outlier points, being far from the main cluster, would likely be isolated very quickly (with short path lengths). For example, a single split on the x-coordinate at x=10 might immediately isolate the point (15, 15). The points within the cluster would require many more splits to be isolated.
4. **Anomaly Score Calculation:** After building multiple trees and averaging the path lengths, the outlier points would have significantly shorter average path lengths, resulting in anomaly scores close to 1. The inlier points would have longer average path lengths and scores closer to 0.5 or lower.

Isolation Tree

Visualization of a Single Tree in Isolation Forest



Parameter in IsolationForest

Parameter	Description	Effect
`contamination`	Proportion of expected outliers	Controls the threshold for classifying anomalies.
`n_estimators`	Number of trees in the ensemble	More trees improve stability but increase computation time.
`max_samples`	Number of samples per tree	Affects tree depth and anomaly detection sensitivity.
`max_features`	Number of features considered per split	Limits feature selection for splits.
`random_state`	Controls randomness of splits	Ensures reproducibility.

Table: Parameters of IsolationForest

Using IsolationForest

```
from sklearn.ensemble import IsolationForest

iso_forest = IsolationForest(
    contamination=0.01,  # Expect 1% anomalies
    n_estimators=100,  # Use 100 trees for better stability
    max_samples=256,  # Use 256 random samples per tree
    max_features=2,  # Consider only 2 features per split
    random_state=42  # Ensure reproducibility
)

outlier_labels = iso_forest.fit_predict(df)  # 1 for inlier, -1 for outlier

df['Outlier'] = outlier_labels
df['Outlier'].value_counts()
```

Output

```
Outlier
 1    497
-1      6
Name: count, dtype: int64
```

Removing Outliers

```
df_no_outliers = df[df['Outlier'] == 1].drop(columns='Outlier')
df_no_outliers.shape
```

Two cluster example

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import IsolationForest

# Generate data
rng = np.random.RandomState(42)

# Generate train data - normal data
X_train = 0.3 * rng.randn(100, 2)
X_train = np.r_[X_train + 2, X_train - 2]  # Two clusters

# Generate some outliers
X_outliers = rng.uniform(low=-4, high=4, size=(20, 2))
```

Two cluster example

```
X = np.r_[X_train, X_outliers]

# Fit the model
clf = IsolationForest(n_estimators=100, max_samples='auto',
 ↪  contamination='auto', random_state=42)
clf.fit(X_train)  # Fit on the training data (normal data)

# Predict -  1 for inliers, -1 for outliers
y_pred = clf.predict(X)
```

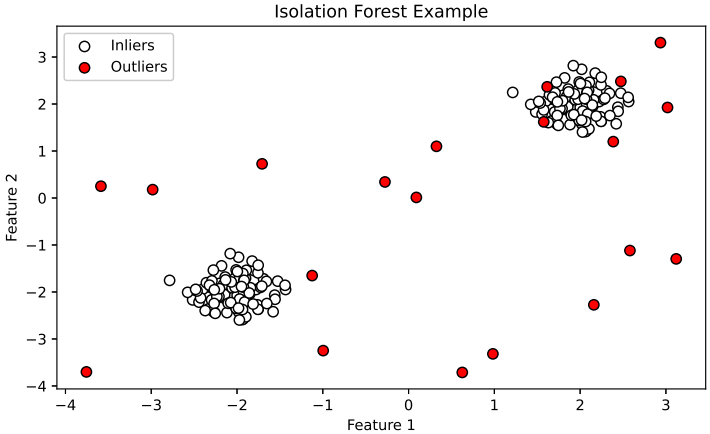
Code: Plot the above example

```
# Plot the data
plt.figure(figsize=(8, 6))

# Plot inliers (normal points)
plt.scatter(X[:200, 0], X[:200, 1], c='white', edgecolors='k', s=20,
 ↪  label='Inliers')

# Plot outliers
plt.scatter(X[200:, 0], X[200:, 1], c='red', edgecolors='k', s=20,
 ↪  label='Outliers')
```

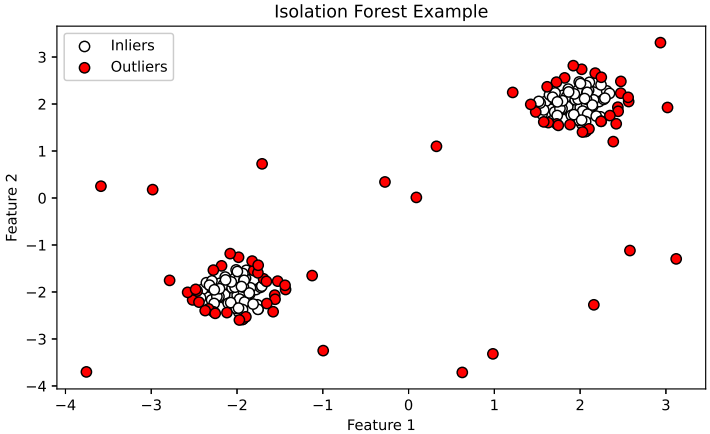
Plot


Isolation Forest Example
Inliers
Outliers
Feature 1
Feature 2
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3

code:Plot

```
for i, color in enumerate(['white', 'red']):
idx = (y_pred == (-1 if color == 'red' else 1))
plt.scatter(X[idx, 0], X[idx, 1], c=color, edgecolors='k', s=50,
 ↪  label=('Outliers' if color == 'red' else 'Inliers'))
```

Plot


Isolation Forest Example
Inliers
Outliers
Feature 1
Feature 2
−4
−3
−2
−1
0
1
2
3

Principal Component Analysis (PCA)

- Having too many features in data can cause problems like overfitting (good on training data but poor on new data), slower computation, and lower accuracy.
- Principal Component Analysis (PCA) is a widely used linear dimensionality reduction technique. Its primary goal is to find a new set of uncorrelated variables, called principal components, that capture the maximum variance in the data.
- It prioritizes the directions where the data varies the most (because more variation = more useful information.

Covariance Matrix

The covariance matrix describes the relationships between different variables in a dataset. For a dataset with n variables, the covariance matrix is an $n \times n$ matrix where each element (i, j) represents the covariance between variable i and variable j. The diagonal elements represent the variances of the individual variables.

Eigenvectors and Eigenvalues

Eigenvectors and eigenvalues are fundamental concepts in linear algebra. For a square matrix (like the covariance matrix), an eigenvector is a non-zero vector that, when multiplied by the matrix, results in a scaled version of itself. The scaling factor is the corresponding eigenvalue. Mathematically:

$$Av = \lambda v$$

where A is the matrix, v is the eigenvector, and λ is the eigenvalue.
In the context of PCA, the eigenvectors of the covariance matrix represent the directions of the principal components, and the eigenvalues represent the amount of variance explained by each principal component.

Projection onto Principal Components

Once the principal components (eigenvectors) are found, the original data can be projected onto these new axes. This projection transforms the data from the original coordinate system to the coordinate system defined by the principal components. The projected data points are the coordinates of the data in the new space.

Mathematically, if X is the standardized data matrix and V is the matrix whose columns are the selected eigenvectors (principal components), the projection P is calculated as:

$$P = XV$$

PCA Algorithm

1. **Data Standardization:** Standardize the data to have zero mean and unit variance for each feature. This is crucial because PCA is sensitive to the scale of the variables.
2. **Covariance Matrix Calculation:** Calculate the covariance matrix of the standardized data.
3. **Eigenvalue Decomposition:** Perform eigenvalue decomposition on the covariance matrix to obtain the eigenvectors (principal components) and eigenvalues.

PCA Algorithm (contd.)

4. **Selection of Principal Components:** Sort the eigenvectors by their corresponding eigenvalues in descending order. Select the top k eigenvectors, where k is the desired number of dimensions for the reduced data.
5. **Projection of Data:** Project the original (standardized) data onto the selected k principal components. This is done by taking the dot product of the standardized data matrix and the matrix formed by the selected eigenvectors.

Mathematical Example

Let's consider a simple 2D dataset with two data points: $x_1 = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$ and $x_2 = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$.

Step 1: Data Standardization

First, we calculate the mean of each feature:

$$\mu_1 = \frac{2+4}{2} = 3, \quad \mu_2 = \frac{4+2}{2} = 3$$

The mean vector is $\mu = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$. Next, we subtract the mean from each data point:

$$x_1' = x_1 - \mu = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \quad x_2' = x_2 - \mu = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Then, we calculate the standard deviation of each feature (after mean subtraction):

$$\sigma_1 = \sqrt{\frac{(-1)^2 + 1^2}{2-1}} = \sqrt{2}, \quad \sigma_2 = \sqrt{\frac{1^2 + (-1)^2}{2-1}} = \sqrt{2}$$

Finally, we divide each centered data point by the corresponding standard deviation:

$$x_1'' = \begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}, \quad x_2'' = \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{bmatrix}$$

Step 2: Covariance Matrix Calculation

The covariance matrix is calculated as:

$$\Sigma = \frac{1}{n-1} X^T X = \frac{1}{2-1} \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

Step 3: Eigenvalue Decomposition

We need to find the eigenvalues and eigenvectors of the covariance matrix Σ. We solve the characteristic equation:

$$\det(\Sigma - \lambda I) = 0$$

$$\det\left(\begin{bmatrix} 1-\lambda & -1 \\ -1 & 1-\lambda \end{bmatrix}\right) = (1-\lambda)^2 - (-1)^2 = \lambda^2 - 2\lambda = 0$$

The eigenvalues are $\lambda_1 = 2$ and $\lambda_2 = 0$.
For $\lambda_1 = 2$:

$$(\Sigma - 2I)v_1 = 0 \Rightarrow \begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix}\begin{bmatrix} v_{11} \\ v_{12} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

A corresponding eigenvector is $v_1 = \begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$ (normalized).

For $\lambda_2 = 0$:

$$(\Sigma - 0I)v_2 = 0 \Rightarrow \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} v_{21} \\ v_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, v_2 = \begin{bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$$

Step 4: Selection of Principal Components

Since $\lambda_1 = 2$ is the largest eigenvalue, the first principal component is $v_1 = \begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix}$. We choose to keep only this component for dimensionality reduction.

Step 5: Projection of Data

We project the standardized data onto the first principal component:

$$X_{\text{reduced}} = Xv_1 = \begin{bmatrix} -1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} -1/\sqrt{2} \\ 1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

The reduced data consists of the scalar values 1 and -1.

Implementation using Sklearn

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import numpy as np

# Sample data (replace with your data)
X = np.array([[1, 2], [2, 4], [3, 6], [4, 8]])

# 1. Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Implementation using Sklearn

```
# 2. Create a PCA object and specify the number of components
pca = PCA(n_components=1)  # Reduce to 1 dimension

# 3. Fit the PCA model and transform the data
X_reduced = pca.fit_transform(X_scaled)

print(X_reduced)
```

Encoding Categorical Variables

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder,
 ↪  OrdinalEncoder
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1000, n_features=4,
 ↪  random_state=42)
df = pd.DataFrame(X, columns=['num_feat1', 'num_feat2', 'num_feat3',
 ↪  'num_feat4'])

# Add categorical features
df['cat_feat1'] = np.random.choice(['A', 'B', 'C'], size=1000)
df['cat_feat2'] = np.random.choice(['Low', 'Medium', 'High'],
 ↪  size=1000)
```

Encoding Categorical Variables

```
# One-Hot Encoding (for nominal categories)
onehot_encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
cat_feat1_encoded = pd.DataFrame(
    onehot_encoder.fit_transform(df[['cat_feat1']]),
    columns=onehot_encoder.get_feature_names_out(['cat_feat1'])
)
```

Creating a Preprocessing Pipeline

```
from sklearn.pipeline import Pipeline
numeric_features = ['num_feat1', 'num_feat2', 'num_feat3', 'num_feat4']
categorical_features = ['cat_feat1', 'cat_feat2']

numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(drop='first', sparse=False))
])
```

Scikit-Learn: Data Preprocessing

Thank You for Listening!