Programming for Engineers

Lecture 9 - Scientific Python: Loading and Processing Data

Radoslav Škoviera

Table of contents

Overview

1. NumPy: Numerical Data Handling

NumPy (Numerical Python) is the foundational package for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy's array-oriented computing is essential for scientific computing tasks, enabling high-performance operations on large datasets.

2. SciPy: Scientific Computations

```
!pip install scipy
```

SciPy is a library used for scientific and technical computing. It builds on NumPy by adding a collection of algorithms and high-level commands for data manipulation and analysis. SciPy includes modules for optimization, integration, interpolation, eigenvalue problems, algebraic and differential equations, and others, making it a powerful tool for scientific applications.

3. Pandas: Tabular Data Processing

```
!pip install pandas
```

Pandas is a library for data manipulation and analysis. It offers data structures and operations for manipulating numerical tables and time series. Pandas introduces two new data structures to Python: Series and DataFrame, which are built on top of NumPy arrays. These structures allow for fast and efficient data manipulation.

To use Pandas with excel files, you need to install the `openpyxl` package.

```
!pip install openpyxl
```

4. scikit-learn: Machine Learning Preprocessing

```
!pip install scikit-learn
```

scikit-learn is a machine learning library for the Python programming language. It features various classification, regression, and clustering algorithms, including support-vector machines, random forests, gradient boosting, k-means, and DBSCAN. Designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy, scikit-learn is widely used for its simplicity and efficiency in implementing machine learning models.

5. scikit-image: Image Processing

```
!pip install scikit-image
```

scikit-image is a collection of algorithms for image processing. It is designed to interoperate with NumPy and SciPy, providing a versatile toolkit for image analysis. scikit-image includes algorithms for segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more. It is widely used in academic research and industry for processing and analyzing images.

Modules

NumPy: Numerical Data Handling

Loading Data

We have covered NPY and NPZ loading in previous lectures. It is also possible to load structured text (CSV) files using NumPy.

```
# Load data from a text file
data = np.loadtxt('data.csv', delimiter=',')

# Save array to a file
np.savetxt('output.csv', data, delimiter=',')
```

Pandas: Tabular Data Processing

Even though NumPy can load CSV data, it is best to use the library dedicated to loading and processing of tabular data: Pandas.

```
import pandas as pd
```

Loading Data

```
df_csv = pd.read_csv('sales.csv', parse_dates=['order_date'])
df_excel = pd.read_excel('sales.xlsx')
```

Creating tables

```
# From dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Score': [85, 92, 78]
}

df = pd.DataFrame(data)
print(df)
```

```
      Name  Age  Score
0    Alice   25     85
1      Bob   30     92
2  Charlie   35     78
```

Data Exploration

```
print("Display first few rows")
print(df_csv.head())
print("--" * 20)

print("Display last few rows")
print(df_csv.tail())
print("--" * 20)

print("Display summary statistics")
df_csv.info()
```

```
print("--" * 20)

print("Data types")
print(df_csv.dtypes)
print("--" * 20)

print("Summary statistics")
print(df_csv.describe())
print("--" * 20)

print("Unique values")
print(df_csv.nunique())
print("--" * 20)

print("Single column stats")
print("Mean sales:", df_csv['sales'].mean())
print("--" * 20)
```

```
Display first few rows
    order_id order_date     category region  sales  quantity  returned
0  ORD100083 2023-03-25        Books  South  19.06        14     False
1  ORD100366 2024-01-02     Clothing   East  30.88        11     False
2  ORD100564 2024-07-18        Books  South  10.31        11     False
3  ORD100490 2024-05-05        Books   East  20.66        11     False
4  ORD100507 2024-05-22  Electronics  South  19.22        13     False
----------------------------------------
Display last few rows
      order_id order_date  category region  sales  quantity  returned
995  ORD100387 2024-01-23     Books  North  11.48         5     False
996  ORD100324 2023-11-21     Books   East  28.33         3     False
997  ORD100861 2025-05-11     Books  South  37.67         6     False
998  ORD100708 2024-12-09  Clothing  North  13.89         6     False
999  ORD100484 2024-04-29  Clothing  North  16.61        18     False
----------------------------------------
Display summary statistics
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   order_id    1000 non-null   object
 1   order_date  1000 non-null   datetime64[ns]
```

```
 2   category    1000 non-null   object
 3   region      1000 non-null   object
 4   sales       1000 non-null   float64
 5   quantity    1000 non-null   int64
 6   returned    1000 non-null   bool
dtypes: bool(1), datetime64[ns](1), float64(1), int64(1), object(3)
memory usage: 48.0+ KB
----------------------------------------
Data types
order_id              object
order_date    datetime64[ns]
category              object
region                object
sales                float64
quantity               int64
returned                bool
dtype: object
----------------------------------------
Summary statistics
                order_date        sales     quantity
count                 1000  1000.000000  1000.000000
mean   2024-05-14 12:00:00    21.866070     9.956000
min    2023-01-01 00:00:00     4.340000     1.000000
25%    2023-09-07 18:00:00    13.725000     5.000000
50%    2024-05-14 12:00:00    19.770000    10.000000
75%    2025-01-19 06:00:00    26.900000    15.000000
max    2025-09-26 00:00:00    86.240000    19.000000
std                    NaN    11.861339     5.346649
----------------------------------------
Unique values
order_id      1000
order_date    1000
category         4
region           4
sales          870
quantity        19
returned         2
dtype: int64
----------------------------------------
Single column stats
Mean sales: 21.86607
----------------------------------------
```

Accessing Data

```
print("Column:", df['Age'])              # Column
print("Multiple columns:", df[['Name', 'Score']]) # Multiple columns
print("Row by index:", df.iloc[1])             # Row by index
print("Row by label:", df.loc[1])              # Row by label
print("Rows by index:", df.iloc[1:3])           # Rows by index
print("Rows by label:", df.loc[1:3])            # Rows by label
print("Rows by condition:", df[df['Age'] > 30]) # Rows by condition
print("Column by index:", df.iloc[:, 1])        # Column by index
print("Column by label:", df['Age'])            # Column by label
```

```
Column: 0    25
1    30
2    35
Name: Age, dtype: int64
Multiple columns:       Name  Score
0    Alice     85
1      Bob     92
2  Charlie     78
Row by index: Name     Bob
Age       30
Score     92
Name: 1, dtype: object
Row by label: Name     Bob
Age       30
Score     92
Name: 1, dtype: object
Rows by index:       Name  Age  Score
1      Bob   30     92
2  Charlie   35     78
Rows by label:       Name  Age  Score
1      Bob   30     92
2  Charlie   35     78
Rows by condition:       Name  Age  Score
2  Charlie   35     78
Column by index: 0    25
1    30
2    35
Name: Age, dtype: int64
Column by label: 0    25
```

```
1    30
2    35
Name: Age, dtype: int64
```

Filtering and selection

```python
# Filter by condition
print("Filter by condition:", df[df['Age'] >= 30])

print("Filter by multiple conditions:", df[(df['Age'] >= 30) & (df['Score'] < 90)])
```

```
Filter by condition:       Name  Age  Score
1      Bob   30     92
2  Charlie   35     78
Filter by multiple conditions:       Name  Age  Score
2  Charlie   35     78
```

Grouping and aggregation

```python
df_csv.groupby('region').agg({'sales': 'sum'})
df_csv.groupby('category').agg({'sales': ['mean', 'count']})
```

	sales	
	mean	count
category		
Books	21.376166	253
Clothing	22.035588	238
Electronics	22.467233	253
Home	21.598516	256

Saving Data

```python
# Save to CSV
df.to_csv('cleaned_data.csv', index=False)
```

```
# Save to Excel
df.to_excel('cleaned_data.xlsx', index=False)
```

SciPy: Scientific Computations

optimize: Function Minimization & Curve Fitting

The `scipy.optimize` package offers algorithms for function minimization (scalar or multi-dimensional), root-finding, and curve-fitting.

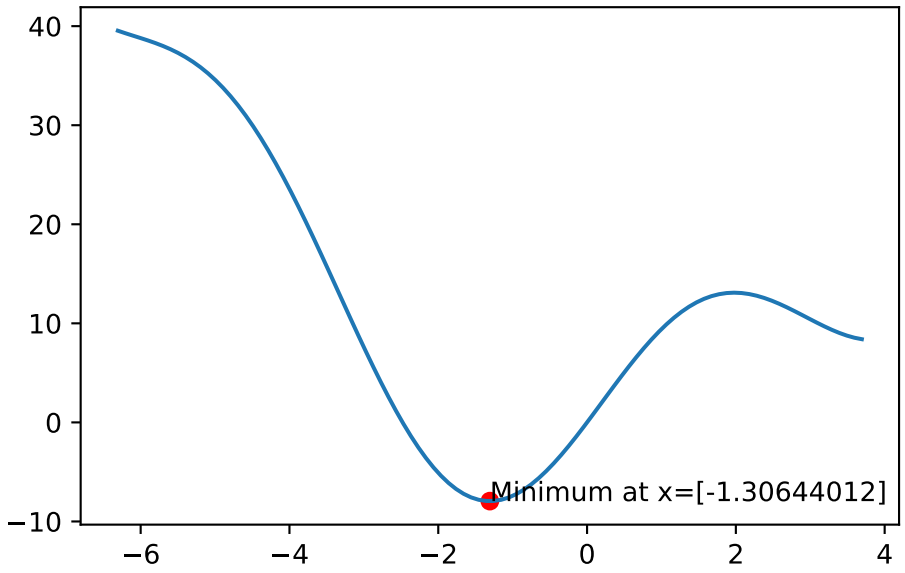
Key Functions & Classes

- **`scipy.optimize.minimize`**: General-purpose minimization of scalar functions of one or more variables.
- **`scipy.optimize.curve_fit`**: Non-linear least squares fitting of a function to data.
- **`scipy.optimize.root`**: Find roots of a function.
- **`scipy.optimize.least_squares`**: Solve nonlinear least-squares with bounds.

Example: Minimizing a Non-Convex Function

```
import numpy as np
from scipy.optimize import minimize
from matplotlib import pyplot as plt


def f(x):
    return x**2 + 10*np.sin(x)

res = minimize(f, x0=0.0, method='BFGS')
print("Minimum at x =", res.x, "with value f(x) =", res.fun)

x_vals = np.linspace(res.x - 5, res.x + 5, 100)
y_vals = f(x_vals)

plt.plot(x_vals, y_vals)
plt.scatter(res.x, f(res.x), color='red')
plt.annotate(f"Minimum at x={res.x}", (res.x, f(res.x)))
plt.show()
```

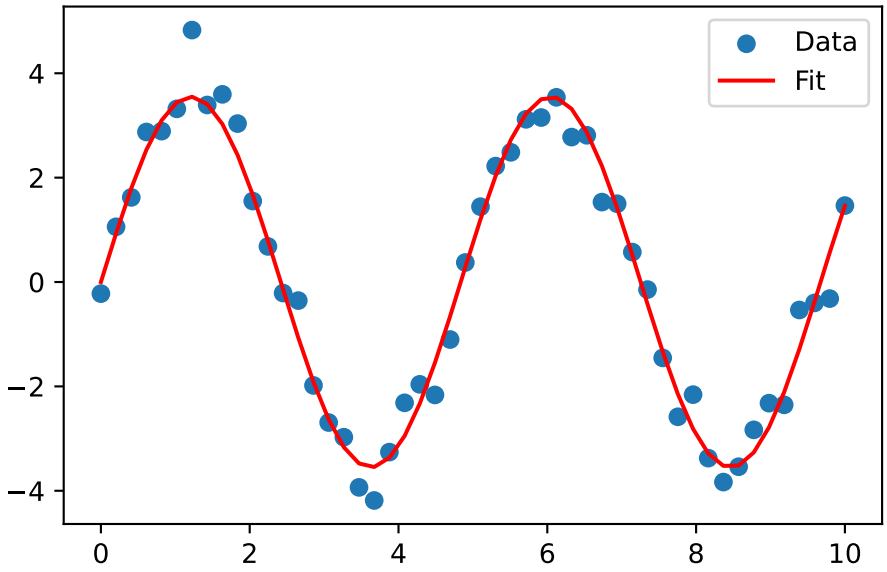
```
Minimum at x = [-1.30644012] with value f(x) = -7.945823375615215
```



Example: Curve Fitting

```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Generate synthetic data
x = np.linspace(0, 10, 50)
y = 3.5 * np.sin(1.3 * x) + np.random.normal(0, 0.5, x.size)

# Define model
def model(x, a, b):
    return a * np.sin(b * x)

# Fit parameters
params, cov = curve_fit(model, x, y)
print("Fitted params:", params)

# Plot
plt.scatter(x, y, label='Data')
plt.plot(x, model(x, *params), 'r-', label='Fit')
plt.legend()
plt.show()
```

```
Fitted params: [3.55125647 1.29908483]
```



---

integrate: Numerical Integration & ODE Solvers

`scipy.integrate` provides functions to compute definite integrals, solve ordinary differential equations (ODEs), and perform multi-dimensional integration.

Key Functions

- **`scipy.integrate.quad`**: Adaptive quadrature for single integrals.
- **`scipy.integrate.solve_ivp`**: Modern ODE solver interface.

Example: Definite Integral

```python
from scipy.integrate import quad

f = lambda t: np.exp(-t**2)
result, error = quad(f, 0, np.inf)
print("Integral of exp(-t^2) from 0 to ∞ =", result)
```

```
Integral of exp(-t^2) from 0 to ∞ = 0.8862269254527579
```

interpolate: Data Interpolation

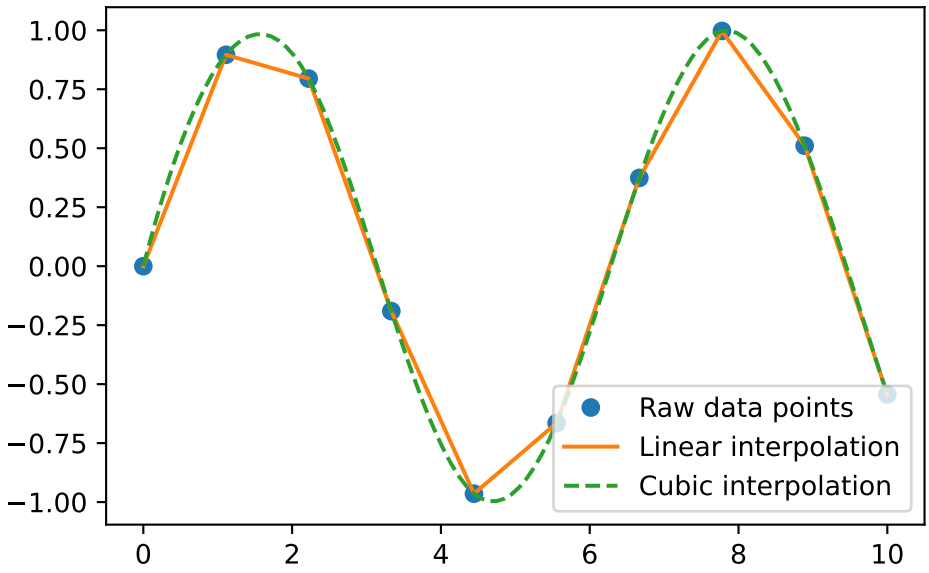
`scipy.interpolate` offers classes and functions for one- and multi-dimensional interpolation and smoothing splines.

Key Classes & Functions

- **`scipy.interpolate.interp1d`**: 1-D linear and spline interpolation.
- **`scipy.interpolate.griddata`**: Interpolation over irregular 2-D data.
- **`scipy.interpolate.BarycentricInterpolator`**, `UnivariateSpline`, `RectBivariateSpline`.

Example: 1-D Interpolation

```
from scipy.interpolate import interp1d

# Original coarse data
x_raw = np.linspace(0, 10, 10)
y_raw = np.sin(x_raw)

# Create interpolator
f_linear = interp1d(x_raw, y_raw, kind='linear')
f_cubic = interp1d(x_raw, y_raw, kind='cubic')

# Evaluate at finer grid
x_fine = np.linspace(0, 10, 100)
plt.plot(x_raw, y_raw, 'o', label='Raw data points')
plt.plot(x_fine, f_linear(x_fine), '-', label='Linear interpolation')
plt.plot(x_fine, f_cubic(x_fine), '--', label='Cubic interpolation')
plt.legend()
plt.show()
```



---

signal: Digital Signal Processing

`scipy.signal` provides signal processing tools: filtering, spectral analysis, window functions, and convolution.

Key Functions

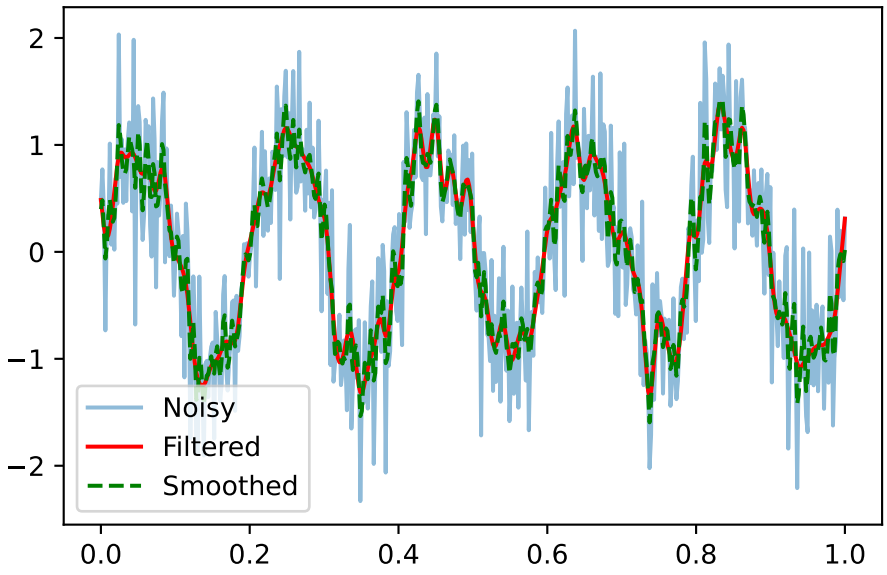
- **`scipy.signal.butter`**, **`scipy.signal.sosfilt`** / **`filtfilt`**: Digital filter design and application.
- **`scipy.signal.welch`**: Power spectral density estimation.
- **`scipy.signal.convolve`**, `correlate`, `decimate`, `resample`.

Example: Butterworth Low-Pass Filter

```
from scipy.signal import butter, sosfiltfilt, convolve

# Design filter
sos = butter(4, 0.2, btype='low', output='sos')
# Create noisy signal
t = np.linspace(0, 1, 500)
sig = np.sin(2*np.pi*5*t) + 0.5*np.random.randn(500)
```

```
# Apply zero-phase filter
filtered = sosfiltfilt(sos, sig)

# Convolution with Gaussian kernel
kx = np.arange(-4, 5)
kernel = np.exp(-kx**2 / 2)
kernel /= kernel.sum()
print(kernel)
smoothed = convolve(sig, kernel, mode='same')

plt.plot(t, sig, alpha=0.5, label='Noisy')
plt.plot(t, filtered, 'r-', label='Filtered')
plt.plot(t, smoothed, 'g--', label='Smoothed')
plt.legend()
plt.show()
```

```
[1.33830625e-04 4.43186162e-03 5.39911274e-02 2.41971446e-01
 3.98943469e-01 2.41971446e-01 5.39911274e-02 4.43186162e-03
 1.33830625e-04]
```



fft: Fourier Transforms

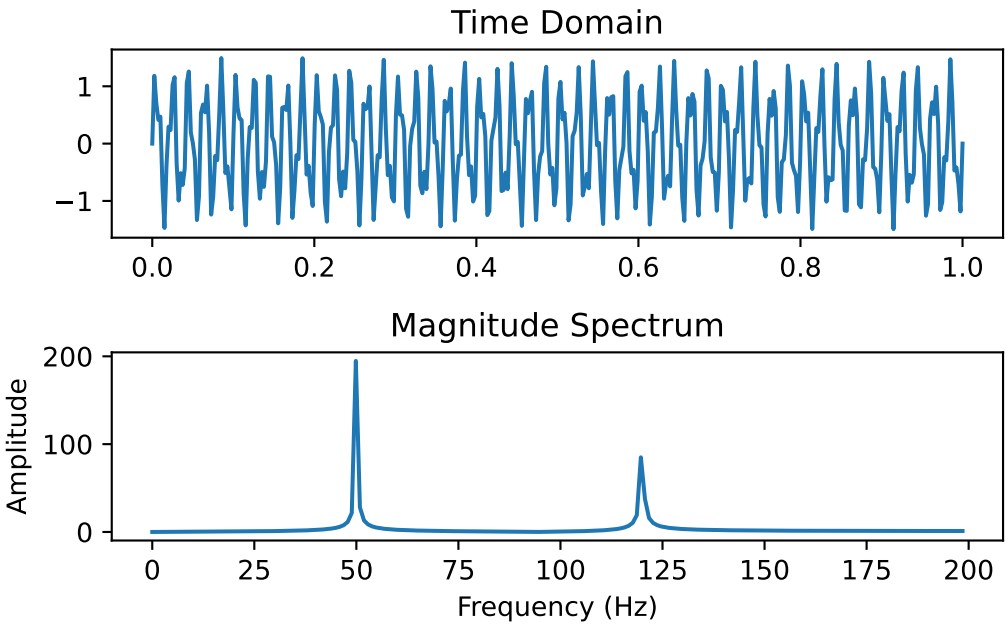
`scipy.fft` Fast Fourier Transform routines for one- and multi-dimensional arrays.

Key Functions

- **`scipy.fft.fft`, `ifft`**: Forward and inverse 1-D FFT.
- **`scipy.fft.fft2`, `ifft2`**: 2-D transforms.
- **`rfft`, `irfft`**: Real-input optimized transforms.

Example: 1-D FFT Spectral Analysis

```
from scipy.fft import fft, fftfreq

# Signal
t = np.linspace(0, 1, 400)
x = np.sin(2*np.pi*50*t) + 0.5*np.sin(2*np.pi*120*t)
# Compute FFT
X = fft(x)
freqs = fftfreq(t.size, d=t[1]-t[0])
fig, [ax1, ax2] = plt.subplots(2, 1)
ax1.plot(t, x)
ax1.set_title("Time Domain")
ax2.plot(freqs[:200], np.abs(X)[:200])
plt.title("Magnitude Spectrum")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude")
plt.tight_layout()
plt.show()
```



---

stats: Statistical Functions & Tests

`scipy.stats` provides a vast collection of statistical distributions, descriptive statistics, and hypothesis tests.

Key Functions & Classes

- **`scipy.stats.norm`, `gamma`, ...**: Continuous distributions.
- **`scipy.stats.ttest_ind`**, **`ttest_rel`**: T-tests for independent and related samples.
- **`scipy.stats.pearsonr`**, **`spearmanr`**: Correlation coefficients.
- **`scipy.stats.kstest`**, **`chisquare`**: Goodness-of-fit tests.

Example: Two-Sample T-Test

```
from scipy.stats import ttest_ind

# Generate two samples
a = np.random.randn(100) + 0.5
b = np.random.randn(100)
```

```
stat, p = ttest_ind(a, b)
print("t-statistic =", stat, "p-value =", p)
```

```
t-statistic = 3.2832265469906057 p-value = 0.0012132727069695383
```

---

sparse: Sparse Matrix Tools

`scipy.sparse` supports sparse matrix representations for memory-efficient storage and fast arithmetic on large, sparse arrays.

Key Classes

- `lil_matrix`, `csr_matrix`, `csc_matrix`, `coo_matrix`.
- Methods: `.dot()`, `.tocsc()`, `.toarray()`.

---

spatial: KD-Tree for Nearest Neighbors

`scipy.spatial.KDTree` provides efficient nearest-neighbor searches in k-dimensional space.

Key Methods

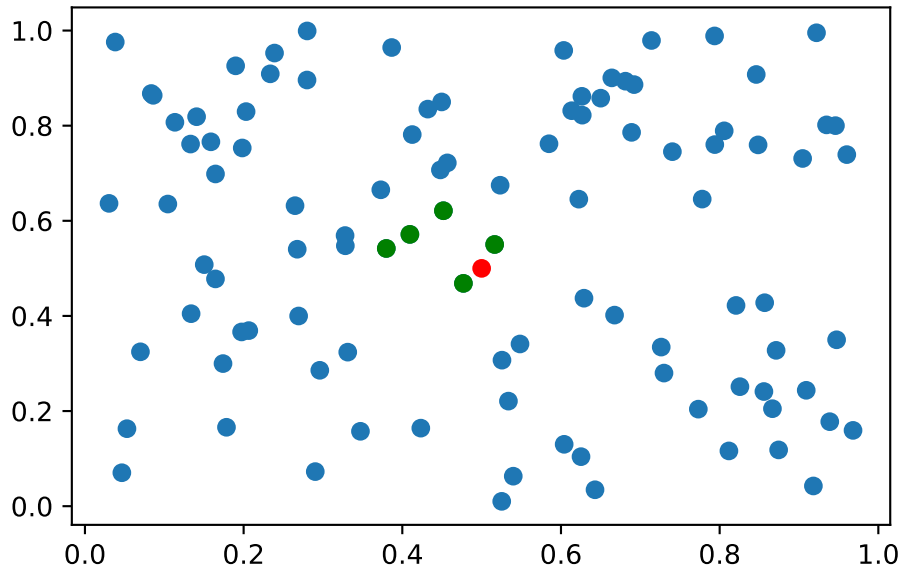
- `query`, `query_ball_point`, `query_pairs`.

Example: Nearest-Neighbor Query

```
from scipy.spatial import KDTree

points = np.random.rand(100, 2)
tree = KDTree(points)
dist, idx = tree.query([0.5, 0.5], k=5)
print("Nearest indices:", idx)

# Plot points and nearest neighbors
plt.scatter(points[:, 0], points[:, 1])
```

```
plt.scatter([0.5], [0.5], c='r')
plt.scatter(points[idx, 0], points[idx, 1], c='g')
plt.show()
```

```
Nearest indices: [65 98 46 48 92]
```



Further Reading

- Official SciPy Reference: https://docs.scipy.org/doc/scipy/reference/
- "SciPy Lecture Notes" for deeper dives: https://scipy-lectures.org/

scikit-learn: Machine Learning Preprocessing

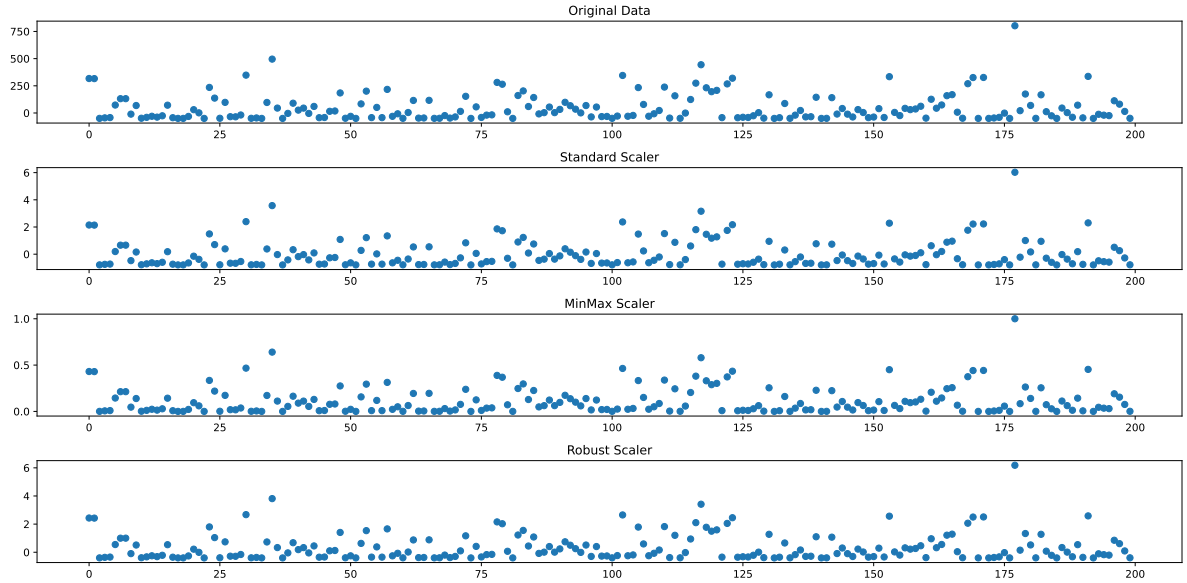
`scikit-learn` provides a unified interface to many machine-learning algorithms and data tools from preprocessing to model selection and evaluation.

Preprocessing `sklearn.preprocessing`

Scaling & Normalization

- **`StandardScaler`**: Centers features to zero mean and unit variance. Critical for algorithms assuming Gaussian-distributed features (e.g. SVM, linear models).
- **`MinMaxScaler`**: Scales features to a fixed range [0,1], preserving shape of original distribution but sensitive to outliers.
- **`RobustScaler`**: Uses median and IQR (inter-quartile range) for centering/scaling, robust to outliers.

```
import numpy as np
from matplotlib import pyplot as plt
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler

N = 200
input_data = np.random.randn(N, 1)**2 * 100 - 50
sc1 = StandardScaler().fit(input_data)
data_std = sc1.transform(input_data)
sc2 = MinMaxScaler().fit(input_data)
data_mm = sc2.transform(input_data)
sc3 = RobustScaler().fit(input_data)
data_rb = sc3.transform(input_data)

# Plot
x_vals = np.arange(N)
fig, ax = plt.subplots(4, 1, figsize=(16, 8))
ax[0].scatter(x_vals, input_data)
ax[1].scatter(x_vals, data_std)
ax[2].scatter(x_vals, data_mm)
ax[3].scatter(x_vals, data_rb)
ax[0].set_title('Original Data')
ax[1].set_title('Standard Scaler')
ax[2].set_title('MinMax Scaler')
ax[3].set_title('Robust Scaler')
plt.tight_layout()
plt.show()
```



Data Splitting and Model Selection `sklearn.model_selection`

- **`train_test_split`**: Quick split for train/test sets.
- **`KFold`**, **`StratifiedKFold`**: Cross-validation iterators preserving distribution.

```
from sklearn.model_selection import train_test_split, KFold
X = np.arange(100).reshape(10, 10)
y = np.arange(10)

X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2, random_state=0)
kf = KFold(n_splits=5, shuffle=True)

for train_index, test_index in kf.split(X, y):
    print("TRAIN:", train_index, "TEST:", test_index)
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

```
TRAIN: [0 1 2 4 5 7 8 9] TEST: [3 6]
TRAIN: [1 2 3 4 5 6 7 9] TEST: [0 8]
TRAIN: [0 1 3 4 5 6 7 8] TEST: [2 9]
TRAIN: [0 2 3 5 6 7 8 9] TEST: [1 4]
TRAIN: [0 1 2 3 4 6 8 9] TEST: [5 7]
```

Estimators

Linear Models

- **`LinearRegression`**: Ordinary Least Squares regression.
- **`LogisticRegression`**: Regularized logistic classifier; supports `l1`/`l2` penalties.

```
from sklearn.linear_model import LinearRegression, LogisticRegression

X_train = np.arange(100).reshape(10, 10)
y_tr_reg = np.arange(10)
y_tr_clf = np.array([0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

lr = LinearRegression().fit(X_train, y_tr_reg)
logr = LogisticRegression().fit(X_train, y_tr_clf)

print("Linear Regression intercept:", lr.intercept_)
print("Linear predictions:")
print([f"{v:.2f}" for v in lr.predict(X_train)])
print("Actual values:")
print(y_tr_reg)

print("Logistic Regression intercept:", logr.intercept_)
print("Logistic predictions:")
print(logr.predict(X_train))
print("Actual values:")
print(y_tr_clf)
```

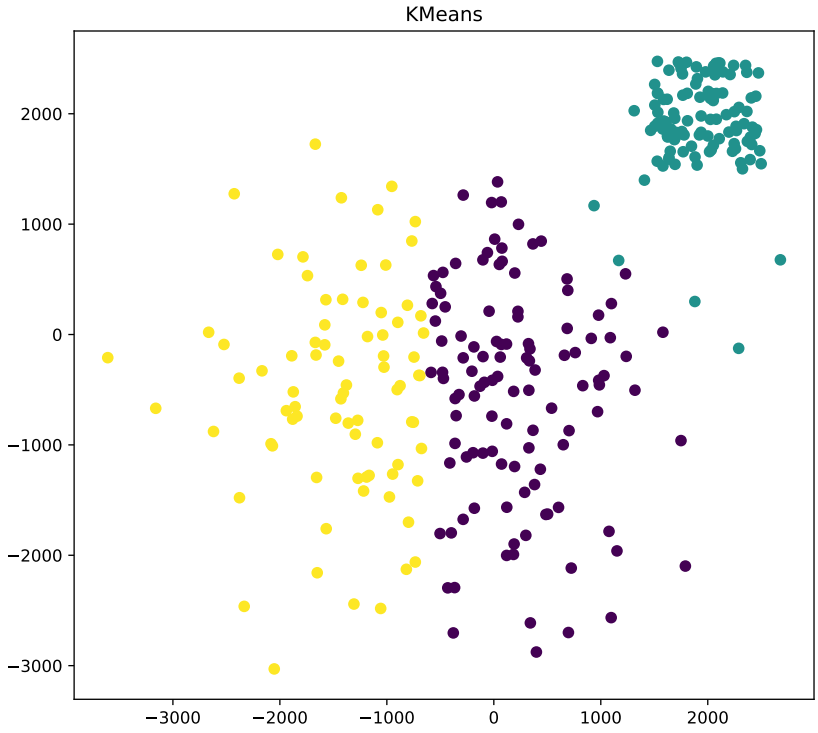
```
Linear Regression intercept: -0.4499999999999993
Linear predictions:
['0.00', '1.00', '2.00', '3.00', '4.00', '5.00', '6.00', '7.00', '8.00', '9.00']
Actual values:
[0 1 2 3 4 5 6 7 8 9]
Logistic Regression intercept: [-36.85911953]
Logistic predictions:
[0 0 0 0 1 1 1 1 1 1]
Actual values:
[0 0 0 0 1 1 1 1 1 1]
```

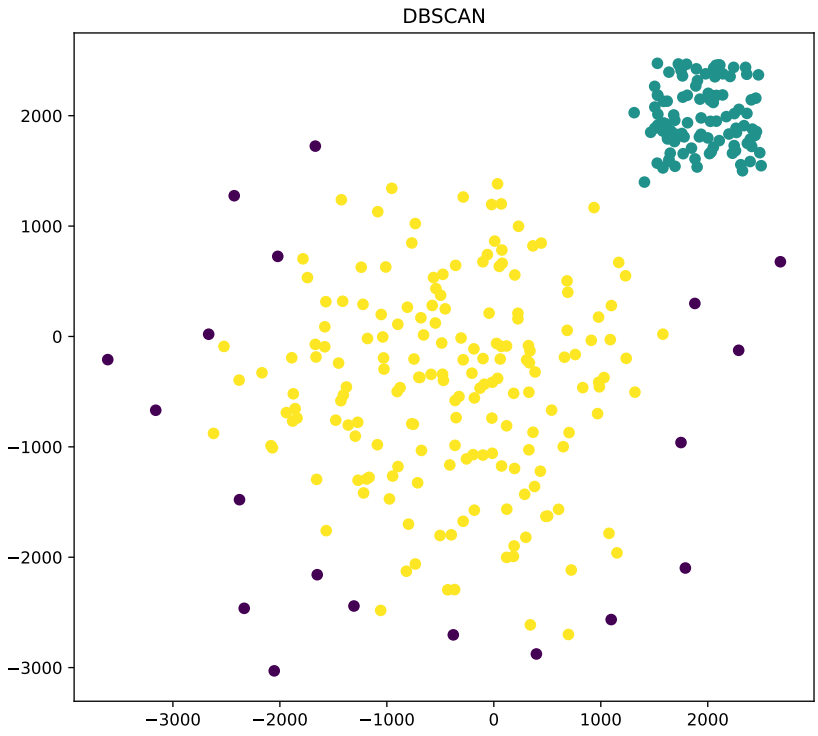
Clustering

- **`KMeans`**: Centroid-based clustering.

- **DBSCAN**: Density-based clustering; identifies arbitrarily shaped clusters.

```
from sklearn.cluster import KMeans, HDBSCAN
X = np.vstack([
    np.random.rand(100, 2) * 1000 + 1500,
    np.random.randn(200, 2) * 1000 - 400
])
km = KMeans(n_clusters=3).fit(X)
db = HDBSCAN().fit(X)

# Plot
fig, ax = plt.subplots(2, 1, figsize=(8, 16))
ax[0].scatter(X[:, 0], X[:, 1], c=km.labels_)
ax[1].scatter(X[:, 0], X[:, 1], c=db.labels_)
ax[0].set_title('KMeans')
ax[1].set_title('DBSCAN')
plt.show()
```


KMeans
2000
1000
0
−1000
−2000
−3000
−3000
−2000
−1000
0
1000
2000


DBSCAN
2000
1000
0
−1000
−2000
−3000
−3000
−2000
−1000
0
1000
2000

Evaluation Metrics `sklearn.metrics`

- **Classification**: `accuracy_score`, `confusion_matrix`, `roc_auc_score`.
- **Regression**: `mean_squared_error`, `r2_score`.
- **Clustering**: `silhouette_score`, `adjusted_rand_score`.

```
from sklearn.metrics import accuracy_score, mean_squared_error

X_train = np.arange(100).reshape(10, 10)
X_test = np.arange(101, 201).reshape(10, 10)
y_train = np.arange(10)
y_test = np.arange(10, 20).astype(float)

lr = LinearRegression().fit(X_train, y_train)

y_pred = lr.predict(X_test)

print("Accuracy:", accuracy_score(y_test.astype(int), y_pred.astype(int)))
print("MSE:", mean_squared_error(y_test, y_pred))
```

```
Accuracy: 1.0
MSE: 0.009999999999999468
```

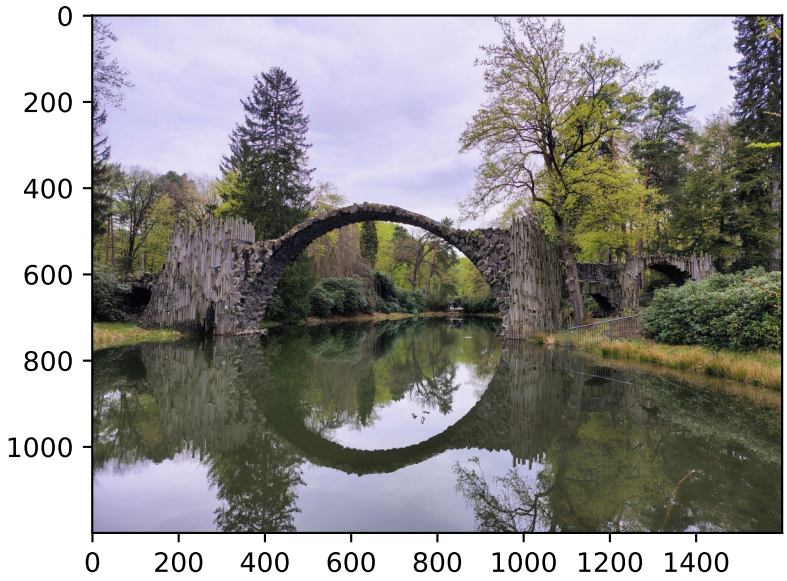
scikit-image: Image Processing

`scikit-image` is a library of image processing algorithms built on NumPy and SciPy.

I/O Plugins `skimage.io`

- **`imread/imsave`**: Read/write images in multiple formats via plugins (PIL, imageio, tifffile, GDAL).
- **`ImageCollection`**, **`MultiImage`**: Efficiently handle batches of images or multi-frame TIFFs.

```
from skimage import io
import os
# base_path = os.path.dirname(__file__)
base_path = os.getcwd()
path = os.path.join(base_path, 'bridge.jpg')
img = io.imread(path)
plt.imshow(img)
io.imsave(os.path.join(base_path, 'out.png'), img)
```

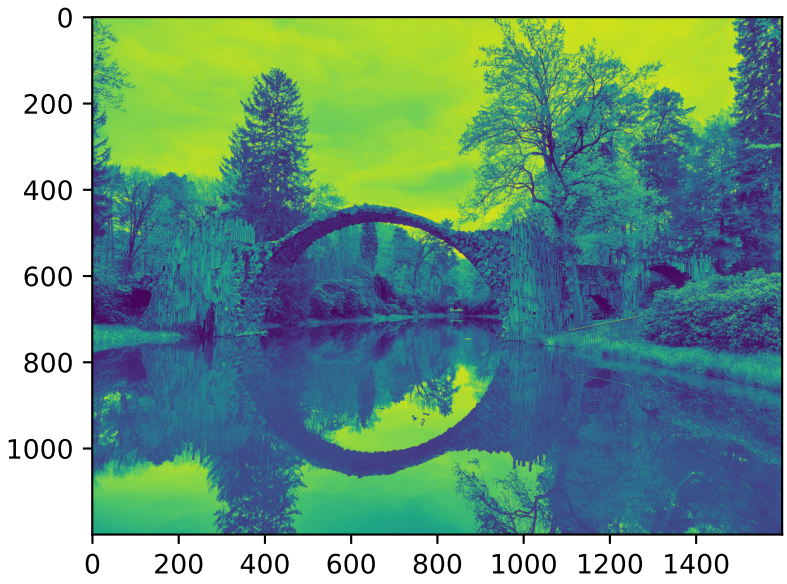



```
# Load a collection of images
col = io.imread_collection(
    os.path.join(base_path, multi_frames, 'frames_*.png')
)
stack = io.concatenate_images(col)
```

Color Space (`skimage.color`)

- **`rgb2gray`**, **`gray2rgb`**, **`rgb2hsv`**: Convert between color spaces.

```
from skimage.color import rgb2gray, gray2rgb
gray = rgb2gray(img)
rgb = gray2rgb(gray)
plt.imshow(gray)
```

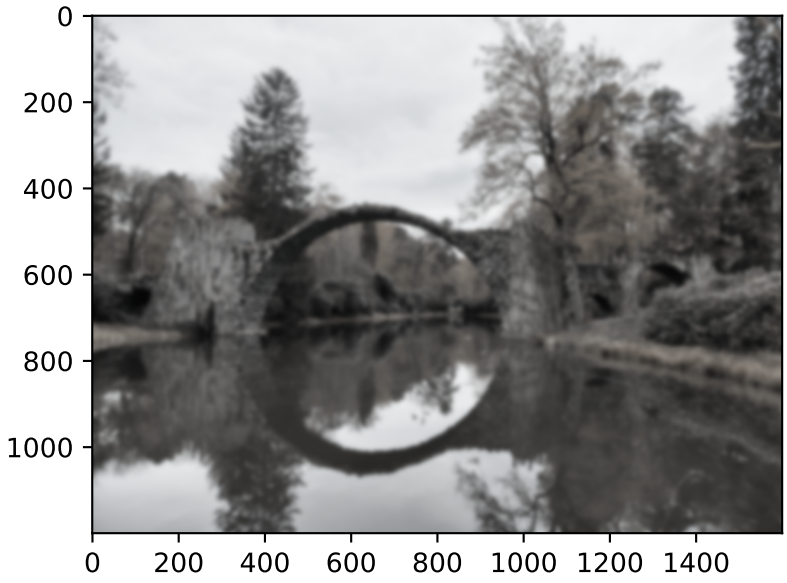


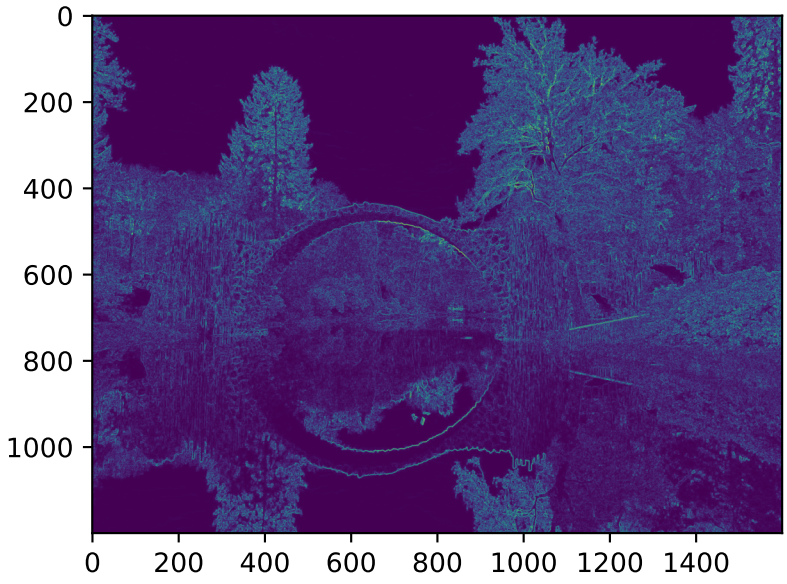
Filtering (`skimage.filters`)

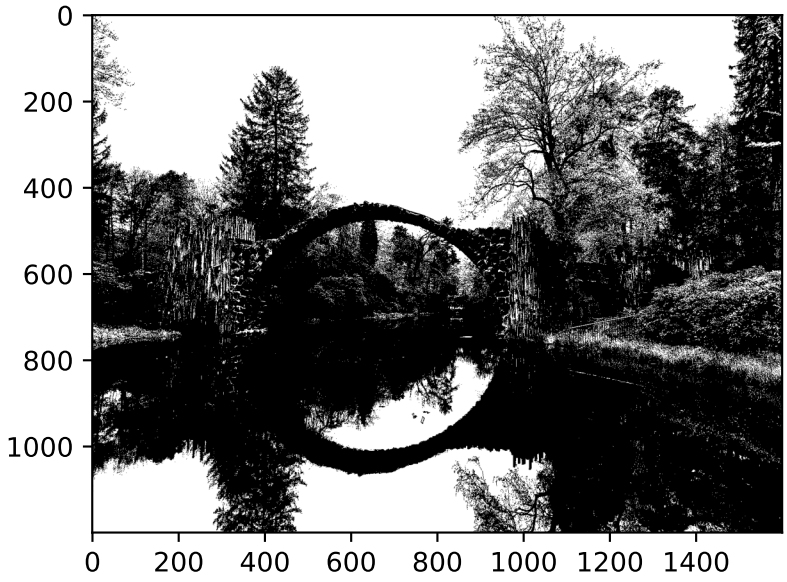
- **Edge Detectors**: `sobel`, `scharr`, `prewitt` compute image gradients.
- **Noise Reduction**: `gaussian`, `median`, **`threshold_otsu`** for binarization.

```
from skimage.filters import sobel, gaussian, threshold_otsu
from matplotlib import pyplot as plt

edges = sobel(gray)
blur = gaussian(img, sigma=5)
th = threshold_otsu(gray)
binary = gray > th

plt.imshow(blur)
plt.show()
plt.imshow(edges)
plt.show()
```

```
plt.imshow(binary, cmap='gray')
plt.show()
```



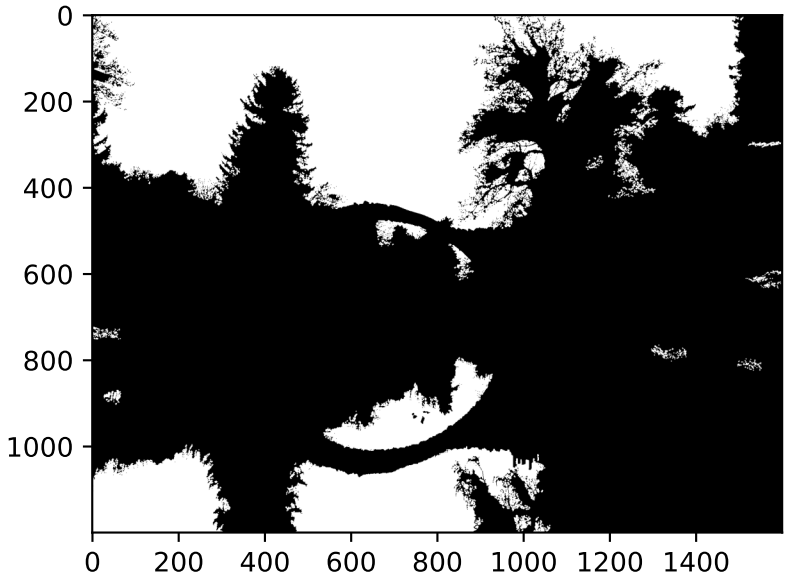




Morphology (`skimage.morphology`)

- **Basic Ops**: `erosion`, `dilation`, `opening`, `closing`.
- **Advanced**: `skeletonize`, `remove_small_objects`, **`area_closing`**.

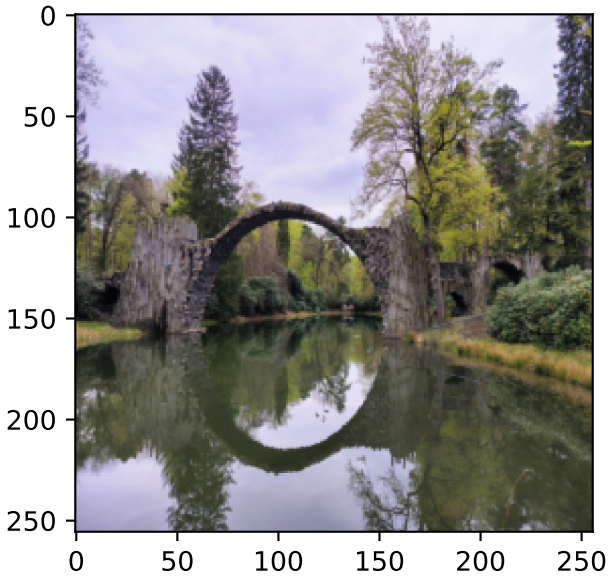
```
from skimage.morphology import opening, remove_small_objects
mask = gray > th
clean = remove_small_objects(mask, min_size=512)
opened = opening(clean)

plt.imshow(opened, cmap='gray')
plt.show()
```

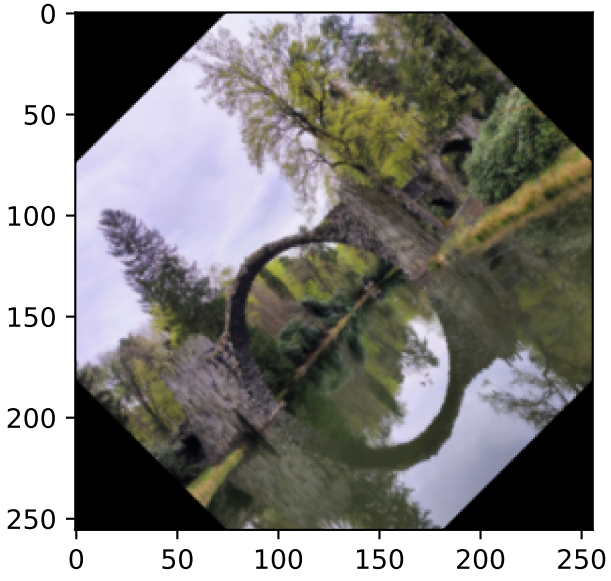


Geometric Transforms (`skimage.transform`)

- **`resize`**, **`rotate`**, **`warp`** for image warping.
- **Hough Transforms**: `hough_line`, `probabilistic_hough_line` for line detection.

```
from skimage.transform import resize, rotate
small = resize(img, (256,256))
rotated = rotate(small, 45)

plt.imshow(small)
plt.show()
plt.imshow(rotated)
plt.show()
```


0
50
100
150
200
250
0
50
100
150
200
250


0
50
100
150
200
250
0
50
100
150
200
250