Vectorization, Normalization, Outliers Detection

Data Analysis with Python

Burak Gündüz
github.com/burakgunduztr

EDA (Exploratory Data Analysis)

We're in the era of fast code and data flows, where understanding the quality of that stream is much more important than ever.

Python list

1,000,000 pointer (any object)

reference count (8 bytes)
pointer to its type (8 bytes)
the actual 8-byte double-precision float
plus padding…

24-32 bytes each, around 36 MB

NumPy array

Large library of array functions, fast compact (fixed type)

import numpy as np

1,000,000 floats (float64)

8 bytes each, around 8 MB

```
# Python list - can mix types
py_list = [1, 'hello', 3.14, None]

# NumPy array - single type
import numpy as np
np_array = np.array([1, 2, 3, 4])  # all int64 (or float64)
```

```
# Common patterns
np.array([1, 2, 3], dtype=np.int32)      # 32-bit integers
np.array([1, 2, 3], dtype=np.float64)    # 64-bit floats (default)
np.array([1, 2, 3], dtype='float32')     # String notation
np.array([1, 2, 3]).astype(np.float32)   # Convert existing array

# Check type
arr.dtype

# Convert type
arr.astype(np.float64)
```

Popular use-cases:

EDA, Science, Statistic

Linear algebra

FFT (Fast Fourier Transform)

Random generations

Vectorized Operations

Expressing operations to whole array in C-level, instead Python loops

```python
import numpy as np
import time

# Setup data
size = 1_000_000
py_list = list(range(size))
np_array = np.arange(size)
```

np.arrange(size) creates integer array including given number of elements

```python
# NON-VECTORIZED (Python list with loop)
start = time.time()
squared_list = []
for x in py_list:
    squared_list.append(x ** 2)
python_time = time.time() - start
print(f"Python loop: {python_time:.4f} seconds")

# NON-VECTORIZED (Python list comprehension - still a loop!)
start = time.time()
squared_list = [x ** 2 for x in py_list]
list_comp_time = time.time() - start
print(f"List comprehension: {list_comp_time:.4f} seconds")

# VECTORIZED (NumPy)
start = time.time()
squared_array = np_array ** 2
numpy_time = time.time() - start
print(f"NumPy vectorized: {numpy_time:.4f} seconds")

print(f"\nSpeedup: {python_time / numpy_time:.1f}x faster")
```

NumPy calculates internally in C tight row buffer (or SIMD) and then assigning result at the end

```
Python loop: 0.1234 seconds
List comprehension: 0.0856 seconds
NumPy vectorized: 0.0012 seconds
Speedup: 102.8x faster
```

```
Traditional loop (one at a time):

| 1 | → | 2 | → | 3 | → | 4 | ... (sequential)

SIMD (parallel processing):

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | → ALL processed in ONE CPU instruction
```

Quick statistic checkup

Statistic	Formula
Mean (μ)	$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$
Variance (σ²)	$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$
Standard Deviation (σ)	$\sigma = \sqrt{\sigma^2} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$

Mean: Average of all values.
Represents the center point of the data.

Variance: Average squared deviation from mean.
Measures how spread out the data is. Units are squared (e.g., cm²). Always ≥ 0

Standard Deviation: Square root of variance. Measures spread in original units (e.g., cm).
More interpretable than variance.

```python
data = np.array([2, 4, 4, 4, 5, 5, 7, 9])

# Mean
mean = np.mean(data)
print(f"Mean: {mean}")  # → 5.0

# Variance
variance = np.var(data)
print(f"Variance: {variance}")  # → 4.0

# Standard Deviation
std = np.std(data)
print(f"Std Dev: {std}")  # → 2.0
```

Why Normalization matters?

Same scale for different features

having “mean 0, std 1” over complete array
makes them comparable, none of them dominance

Easier for many ML algorithms

k-NN (k-Nearest Neighbors)
classification and regression that makes predictions
PCA (Principal Component Analysis)
transforms high-dimensional data into a lower-dimensional
helps optimization and convergence

Interpretation

Normalized 2.0 “2 std above mean”
0.5 “half a standard deviation below”
how unusual, how far from average

Pandas and its DataFrame

One object (container) holds data in a table

import pandas as pd

Operation	axis=0 (rows)	axis=1 (columns)
Direction	Down ↓	Across →
Sum	Sum each column	Sum each row
Mean	Mean of each column	Mean of each row
Delete	Delete a row	Delete a column
Concatenate	Stack vertically (more rows)	Stack horizontally (more columns)

```
import pandas as pd

# Sales data
sales = pd.DataFrame({
    'Jan': [100, 150, 200],
    'Feb': [110, 160, 210],
    'Mar': [120, 170, 220]
}, index=['Product A', 'Product B', 'Product C'])

print(sales)
#             Jan  Feb  Mar
# Product A   100  110  120
# Product B   150  160  170
# Product C   200  210  220

# axis=0: Total sales per month (sum DOWN each column)
monthly_totals = sales.sum(axis=0)
print(monthly_totals)
# Jan    450
# Feb    480
# Mar    510

# axis=1: Total sales per product (sum ACROSS each row)
product_totals = sales.sum(axis=1)
print(product_totals)
# Product A    330
# Product B    480
# Product C    630
```

.groupBy() and .agg()

Code:

```
df.groupby('Product').agg(
    total_sales=('Sales', 'sum'),
    avg_quantity=('Quantity', 'mean'),
    min_cost=('Cost', 'min'),
    max_cost=('Cost', 'max')
)
```

Output:

Product	total_sales	avg_quantity	min_cost	max_cost
A	420	2.67	60	120
B	490	3.33	90	110
C	200	1.50	50	65

Code:

```
df.groupby('Product')['Sales'].agg([
    'sum', 'mean', 'count', 'min', 'max', 'std', 'median'
])
```

Output:

Product	sum	mean	count	min	max	std	median
A	420	140.0	3	100	200	52.92	120.0
B	490	163.33	3	150	180	15.28	160.0
C	200	100.0	2	90	110	14.14	100.0

.groupBy() and .agg()

Code:

```
df.groupby(['Product', 'Region']).agg({
    'Sales': ['sum', 'mean'],
    'Quantity': 'sum'
})
```

Output:

Product	Region	Sales sum	Sales mean	Quantity sum
A	North	220	110.0	4
A	South	200	200.0	4
B	North	310	155.0	7
B	South	180	180.0	3
C	North	110	110.0	2
C	South	90	90.0	1

Code:

```
# Filter BEFORE groupby
df[df['Sales'] > 100].groupby('Product')['Sales'].sum()
```

Output:

Product	Sales
A	320
B	490
C	110

Goal	Code
Single aggregation	`df.groupby('col')['value'].sum()`
Multiple aggregations	`df.groupby('col')['value'].agg(['sum', 'mean'])`
Different per column	`df.groupby('col').agg({'col1': 'sum', 'col2': 'mean'})`
Named aggregations	`df.groupby('col').agg(total=('value', 'sum'))`
Multiple groups	`df.groupby(['col1', 'col2'])['value'].sum()`
All statistics	`df.groupby('col')['value'].describe()`
Pivot view	`df.pivot_table(values='val', index='row', columns='col')`

Why data cleaning matters?

Sources are messy

Bad data breaks analysis

aggregations, models, visualizations

Supporting system expectations

reporting, search, recommendation engines

Data cleaning pipeline

Handle missing values

Remove duplicates

Flag and process unexpected values

Add a new column If needed

```
import pandas as pd

# Original data
data = {
    'Name': ['Alice', 'Bob', '', 'David'],
    'Age': [25, '', 35, 28],
    'Salary': ['50000', '60000', '', '55000'],
    'Dept': ['Sales', 'IT', 'N/A', 'Sales']
}
df = pd.DataFrame(data)

# Step 1: Replace placeholders with NA
df = df.replace(['', 'N/A'], pd.NA)

# Step 2: Fill missing values
df = df.fillna({'Name': 'Unknown', 'Age': '30', 'Salary': '0', 'Dept': 'Other'})

# Step 3: Convert types
df['Age'] = df['Age'].astype(int)
df['Salary'] = df['Salary'].astype(int)
```

Data cleaning pipeline

Approach 1: fillna() + astype() (What I Used)

```python
# Works if you KNOW the data format is valid after filling
df = df.replace(['', 'N/A'], pd.NA)
df = df.fillna({'Age': '30', 'Salary': '0'})  # Fill with string numbers
df['Age'] = df['Age'].astype(int)  # Safe - all values are numeric strings
df['Salary'] = df['Salary'].astype(int)
```

When to use: Data is clean, just has missing values

astype() has no errors= param.
use pd.to_numeric()

errors='ignore' #leave as-is
'coerce' #NaN
'raise' #Error!

Approach 2: pd.to_numeric(errors='coerce') + fillna() + astype()

```python
# Better if data might have INVALID non-numeric values
df = pd.DataFrame({
    'Age': ['25', 'N/A', '35', 'invalid', '28']
})

# Step 1: Convert, coerce invalid to NaN
df['Age'] = pd.to_numeric(df['Age'], errors='coerce')
# Result: [25.0, NaN, 35.0, NaN, 28.0]

# Step 2: Fill NaN
df['Age'] = df['Age'].fillna(30)
# Result: [25.0, 30.0, 35.0, 30.0, 28.0]

# Step 3: Convert to int (now safe!)
df['Age'] = df['Age'].astype(int)
# Result: [25, 30, 35, 30, 28]
```

When to use: Data might contain invalid values like 'invalid', '???', 'unknown'

Data cleaning pipeline

Method	Purpose	Example
`replace()`	Replace specific values	`df.replace('N/A', np.nan)`
`dropna()`	Remove rows with NaN	`df.dropna()`
`dropna(subset=[...])`	Remove rows with NaN in specific columns	`df.dropna(subset=['Age'])`
`dropna(how='all')`	Remove only completely empty rows	`df.dropna(how='all')`
`fillna()`	Fill NaN with value	`df.fillna(0)`
`fillna({...})`	Fill different columns differently	`df.fillna({'Age': 30, 'Salary': 50000})`
`ffill()`	Forward fill	`df['col'].ffill()`
`bfill()`	Backward fill	`df['col'].bfill()`
`astype()`	Convert data type	`df['col'].astype(int)`

.apply() vs .cut()

```python
# Method 1: apply() (SLOW)
def salary_bracket(salary):
    if salary < 50000:
        return 'Entry'
    elif salary < 75000:
        return 'Mid'
    elif salary < 100000:
        return 'Senior'
    else:
        return 'Executive'

start = time.time()
df['Level_Apply'] = df['Salary'].apply(salary_bracket)
time_apply = time.time() - start

# Method 2: cut() (FAST)
bins = [0, 50000, 75000, 100000, 200000]
labels = ['Entry', 'Mid', 'Senior', 'Executive']

start = time.time()
df['Level_Cut'] = pd.cut(df['Salary'], bins=bins, labels=labels)
time_cut = time.time() - start
```

```
Results:
  Employee  Salary Level_Apply Level_Cut
0    E0000   78839      Senior    Senior
1    E0001  144670   Executive Executive
2    E0002   52889         Mid       Mid
3    E0003  132073   Executive Executive
4    E0004  116619   Executive Executive

apply(): 0.0267s
cut():   0.0008s
Speedup: 33.4x
```

Aspect	.apply()	.cut()
Vectorized	❌ No (row-by-row)	✅ Yes
Speed	Slow	Fast (10-50x faster)
Use case	Complex logic	Binning continuous values
Readability	Less clear	Very clear
Performance	Poor on large data	Excellent

Quick recap for terminology

Vectorization

is a way of implementing operations

Normalization

is a type of transformation

They're not the same process!

You can normalize in a vectorized way
or
Implement other (non-normalization) operations in a vectorized way

IQR (Interquartile Range)

It's a measure of spread, values are outside treated as Outliers

Finding quartiles > IQR (Q3-Q1) > Defining Fences (J.Tukey's 1.5x rule) > Filtering data

gives us width of the middle 50% , spread of the mid. of the distribution

```python
import numpy as np

data = np.array([2, 4, 4, 4, 5, 5, 7, 9, 15, 100])

Q1  = np.percentile(data, 25)     # 4.0
Q3  = np.percentile(data, 75)     # 8.5
IQR = Q3 - Q1                     # 4.5

lower = Q1 - 1.5 * IQR            # 4.0 - 6.75 = -2.75
upper = Q3 + 1.5 * IQR            # 8.5 + 6.75 = 15.25

outliers = data[(data < lower) | (data > upper)]
```

```
Q1      = 4.0
Q3      = 8.5
IQR     = 4.5
Lower   = -2.75
Upper   = 15.25
Outliers: [100]
```

Library	Code
NumPy	`np.percentile(data, 75) - np.percentile(data, 25)`
SciPy	`stats.iqr(data)`
Pandas	`s.quantile(0.75) - s.quantile(0.25)`

Recommended: `stats.iqr(data)` — cleanest, one-liner, purpose-built.

```
                ┌── Whisker (extends to furthest point within fence)
                │
  Outlier →  •     (beyond upper fence)
                │
                ╎  (max within fence)
                │
                ┌────────┐
                │        │
                │        │  ← Box: Contains middle 50% of data
                │        │     (from Q1 to Q3)
                ├────────┤  ← Q3 (75th percentile)
                │        │
                ├────────┤  ← Median (Q2, 50th percentile)
                │        │
                ├────────┤  ← Q1 (25th percentile)
                │        │
                └────────┘

                │
                ╎  (min within fence)
                │
  Outlier →  •     (beyond lower fence)
                │
                └── Whisker
```

Data Analysis with Python

Std

(Standart Deviation)

measure of spread

“typical distance” from the mean
one number, doesn’t change each x

think a song dataset with
mean=120 BPM, std=10 (It’s also BPM)

allow us to call: relative tight, much wider or above

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N}}$$

lower std: values cluster near mean
higher std: values spread out or multimodel

Normalization

Z-Score Standardization (StandardScaler)

rescale each value

“How many standard deviations”
above or below from the mean

$$z = \frac{x_i - \bar{x}}{\sigma}$$

(gives us std unit result)

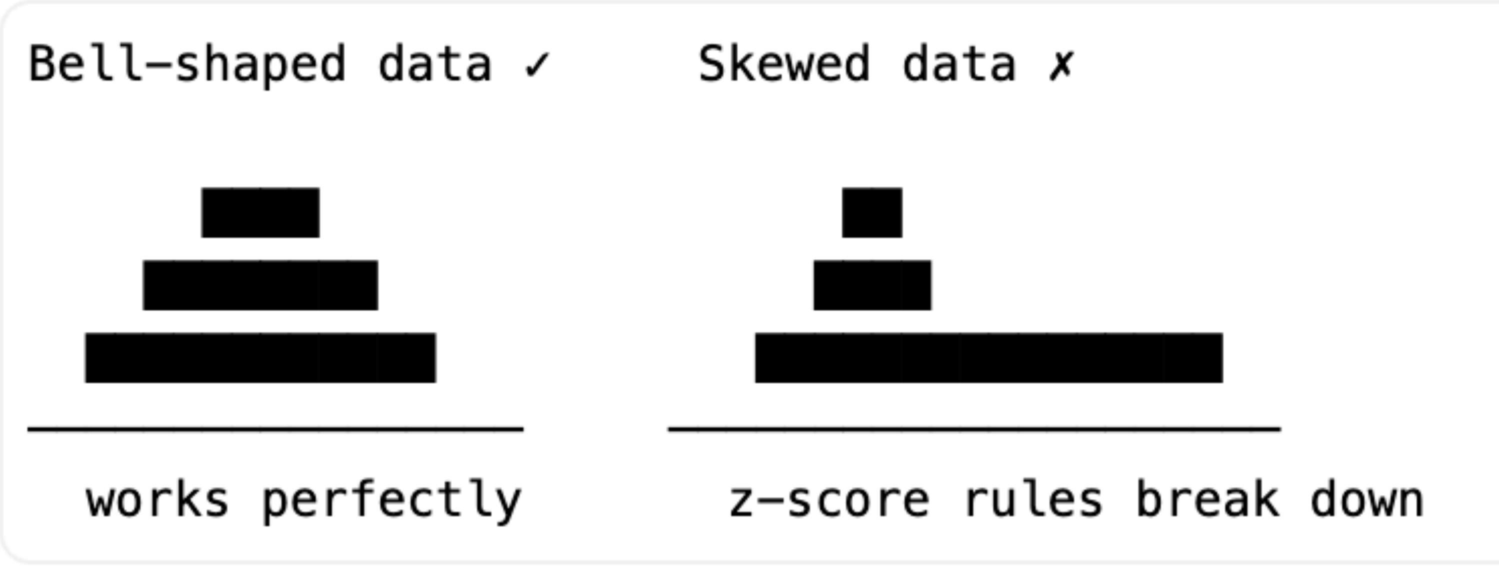
z = 0	Exactly at the mean
z > 0	Above the mean
z < 0	Below the mean
z = +1	1 std above the mean
z = −2	2 stds below the mean

think a tempo (BPM) dataset with
mean=120, std=20

think a duration (sec.) dataset with
mean=180, std=10

allows us “scale-free comparison”

abs(z)>2 is commonly used
as flagged outlier



Z-score is calculated per row after mean and std calculated in the column

Term	In Statistics	In a DataFrame
Column	A variable / feature	`df['Age']`
Row	One observation / data point	A single person, transaction, etc.
Mean & Std	Computed per **column**	One value summarizing the whole feature
Z-score	Computed per **row**	One score per individual observation

> Each column gets its **own** mean and std. You never mix columns when computing z-scores — Age's mean is never used to normalize Salary.

Normalization (Min-Max)

Min-Max Normalization (MinMaxScaler)

$$x_{norm} = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

```
Original data:    60 ──────────── 120 ──────────── 180
                   ↑                                 ↑
                  min                               max

After scaling:    0 ──────────── 0.5 ──────────── 1.0
```

- The **minimum** always maps to `0`
- The **maximum** always maps to `1`
- Everything else falls **proportionally** in between

Min-Max vs Z-Score

	Min-Max	Z-Score
Output range	Always 0 to 1	Typically −3 to +3
Handles outliers	Poorly — outlier pulls everything toward 0	Better — outlier just gets a large z value
Preserves zero	No — 0 in original ≠ 0 in result	No
Best for	Neural networks, image pixels, bounded features	Statistics, outlier detection, comparing features
Sensitive to new data	Yes — new min/max changes all values	Yes — new mean/std changes all values

> Use **Min-Max** when you need a strict 0–1 boundary (e.g. neural network inputs, percentage scores). Use **Z-score** when you care about how unusual a value is relative to the distribution.

Over the Math/Programming Formulas

Mathematical Formula (i=1 to n)

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

Programming Formula (i=0 to n-1)

$$s^2 = \frac{1}{n-1} \sum_{i=0}^{n-1} (x_i - \bar{x})^2$$

Where:

- **n** = number of data points
- **n-1** = denominator (Bessel's correction, ddof=1)
- **i starts at 0** and goes to **n-1** (n iterations total)
- **x_i** = data point at index i
- **x̄** = mean of the data

DDOF (Delta Degrees of Freedom)

with Bessel's correction bias

averaging with n
denominator is too large

unbiased estimate of population variance
(averaging over the right count (matches the count of free pieces))

What	ddof=0 (population)	ddof=1 (sample)
Variance	$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$	$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$
Std	$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$	$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$
Z-score	$z_i = \frac{x_i - \bar{x}}{\sigma}$ (uses σ above)	$z_i = \frac{x_i - \bar{x}}{s}$ (uses s above)
Min-max [0, 1]	$\frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$	*(no ddof)*

```
import numpy as np
from scipy import stats
from sklearn.preprocessing import StandardScaler, MinMaxScaler

data = [2, 4, 4, 4, 5, 5, 7, 9]

# Variance
np.var(data, ddof=1)              # 4.571

# Standard Deviation
np.std(data, ddof=1)              # 2.138

# Z-Score
stats.zscore(data, ddof=1)        # [-1.402, -0.468, ..., 1.870]

# Min-Max [0, 1]
MinMaxScaler().fit_transform(np.array(data).reshape(-1, 1))  # [[0.], [0.286], ..., [1.]]
```

Applying for Apple's Swift

import simd

```swift
// Process 4 floats simultaneously in a single CPU instruction
let a = SIMD4<Float>(1.0, 2.0, 3.0, 4.0)
let b = SIMD4<Float>(5.0, 6.0, 7.0, 8.0)

let sum     = a + b           // [6, 8, 10, 12]
let product = a * b           // [5, 12, 21, 32]
let dotProd = (a * b).sum()   // 70.0

// Normalize a vector to unit length
func normalize(_ v: SIMD4<Float>) -> SIMD4<Float> {
    return v / sqrt((v * v).sum())
}
```

Approach	Best For
Pure Swift	Small datasets, readability
SIMD	Fixed-size vectors, geometry, ML inference
vDSP (Accelerate)	Large arrays, audio, signal processing
BNNS / CreateML	Full ML pipelines on Apple platforms

import Accelerate

```swift
var data: [Float] = [10, 20, 30, 40, 50]

// --- Vectorized arithmetic ---
var result = [Float](repeating: 0, count: data.count)
var scalar: Float = 2.0
vDSP_vsmul(data, 1, &scalar, &result, 1, vDSP_Length(data.count))
// result = [20, 40, 60, 80, 100]

// --- Min-Max Normalization using vDSP ---
func vDSPNormalize(_ input: [Float]) -> [Float] {
    var minVal: Float = 0, maxVal: Float = 0
    vDSP_minv(input, 1, &minVal, vDSP_Length(input.count))
    vDSP_maxv(input, 1, &maxVal, vDSP_Length(input.count))

    var shifted = [Float](repeating: 0, count: input.count)
    var negMin = -minVal
    vDSP_vsadd(input, 1, &negMin, &shifted, 1, vDSP_Length(input.count))

    var range = maxVal - minVal
    var output = [Float](repeating: 0, count: input.count)
    vDSP_vsdiv(shifted, 1, &range, &output, 1, vDSP_Length(input.count))
    return output
}

let normalized = vDSPNormalize([10, 20, 30, 40, 50])
// [0.0, 0.25, 0.5, 0.75, 1.0]
```

Thank you.

Burak Gündüz

Data Analysis with Python

github.com/burakgunduztr