

DATA 3464: Fundamentals of Data Processing

Numeric Data Transformations

Charlotte Curtis

January 27, 2026

Topic overview

- Why transformations are necessary
- Common transformations
- Dimensionality reduction

Resources used:

- [Feature Engineering Chapter 6](#)
- Hands on Machine Learning with Scikit-Learn and Tensorflow/PyTorch, Chapter 4.
Available at [MRU Library](#)
- [Scikit-learn user guide: Chapter 7](#)
- Introduction to Machine Learning with Python. Available at [MRU Library](#)

Common 1:1 transformations

"Most models work best when each feature (and in regression also the target) is loosely Gaussian distributed" -- Introduction to Machine Learning with Python

- Scaling: normalization or standardization
- Nonlinear transforms: log, square root, polynomial
- Fancier methods: Box-Cox, Yeo-Johnson

A brief intro to gradient descent

- Many linear models minimize some cost function through **gradient descent**
- The **gradient** is a vector of partial derivatives

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

for some scalar-valued $f(\mathbf{x})$

Descending the gradient

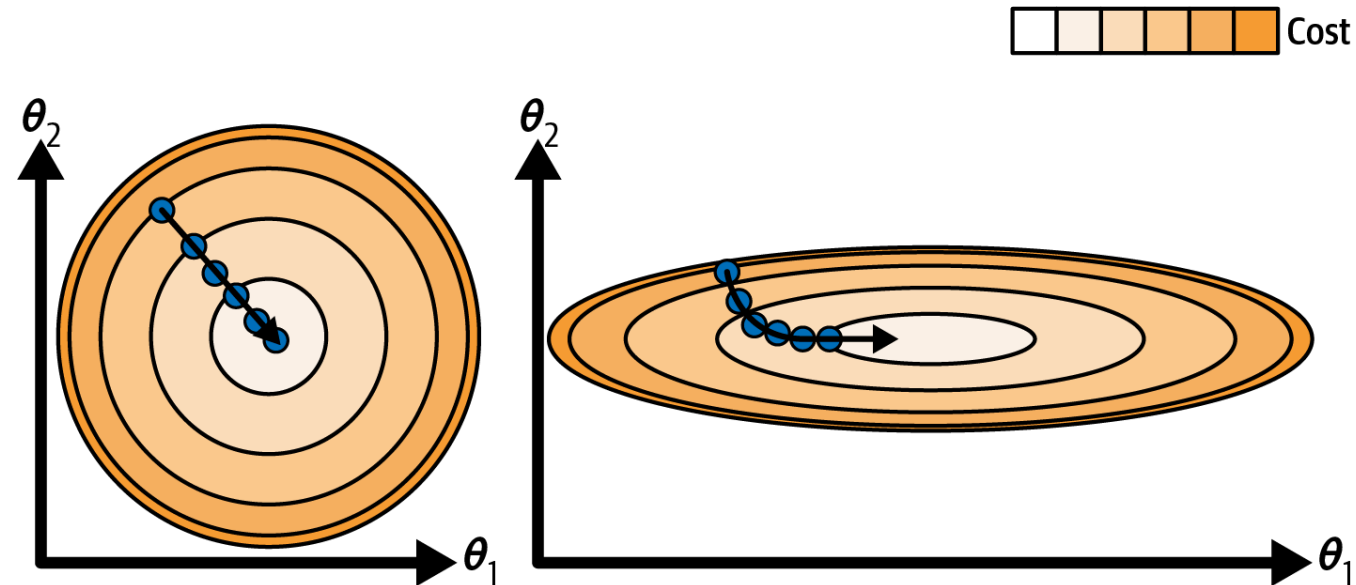
For a loss (or cost) function such as $MSE(\theta) = \frac{1}{m} (\mathbf{X}\theta - \mathbf{y})^T (\mathbf{X}\theta - \mathbf{y})$

1. Start with a random θ
2. Calculate the gradient ∇_{θ} for the current θ
3. Update θ as $\theta = \theta - \eta \nabla_{\theta}$
4. Repeat 2-3 until some stopping criterion is met

where η is the **learning rate**, or the size of step to take in the direction opposite the gradient.

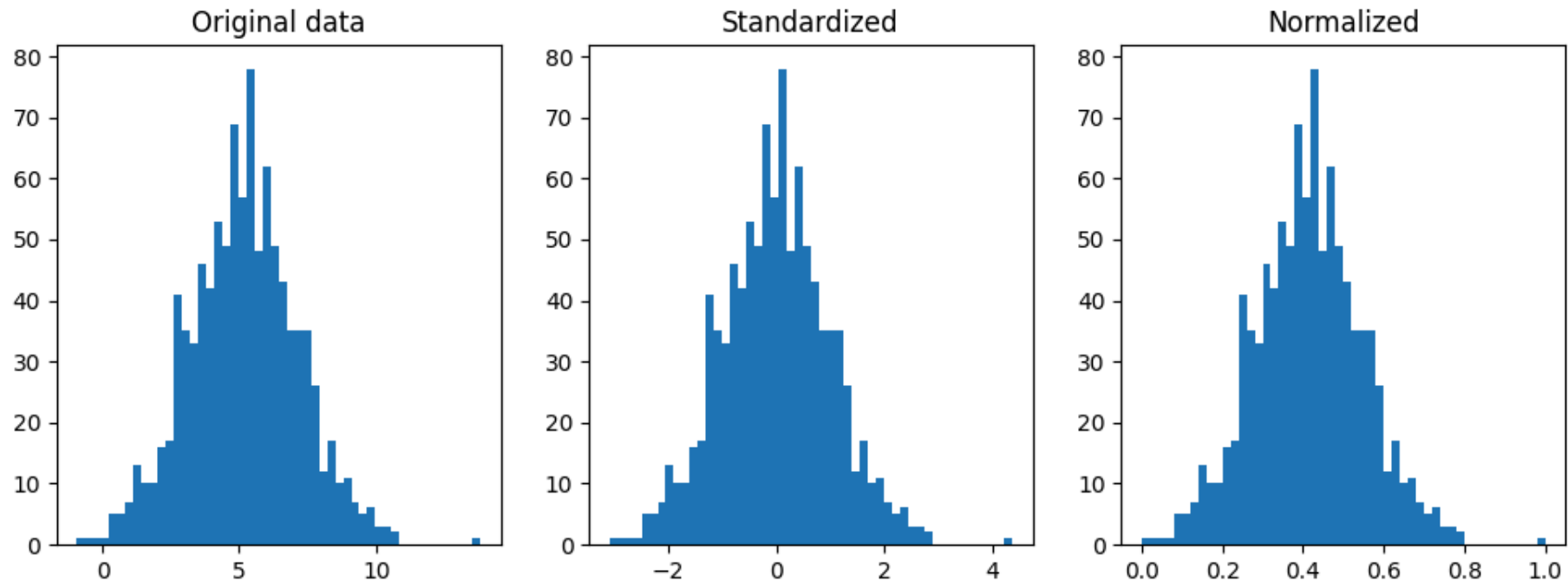
Visualizing in 2D

- The gradient has $m + 1$ dimensions, where m is the number of features
- step size η is a scalar parameter



Main takeaway: feature should be more or less on the same scale

Approaches to scaling



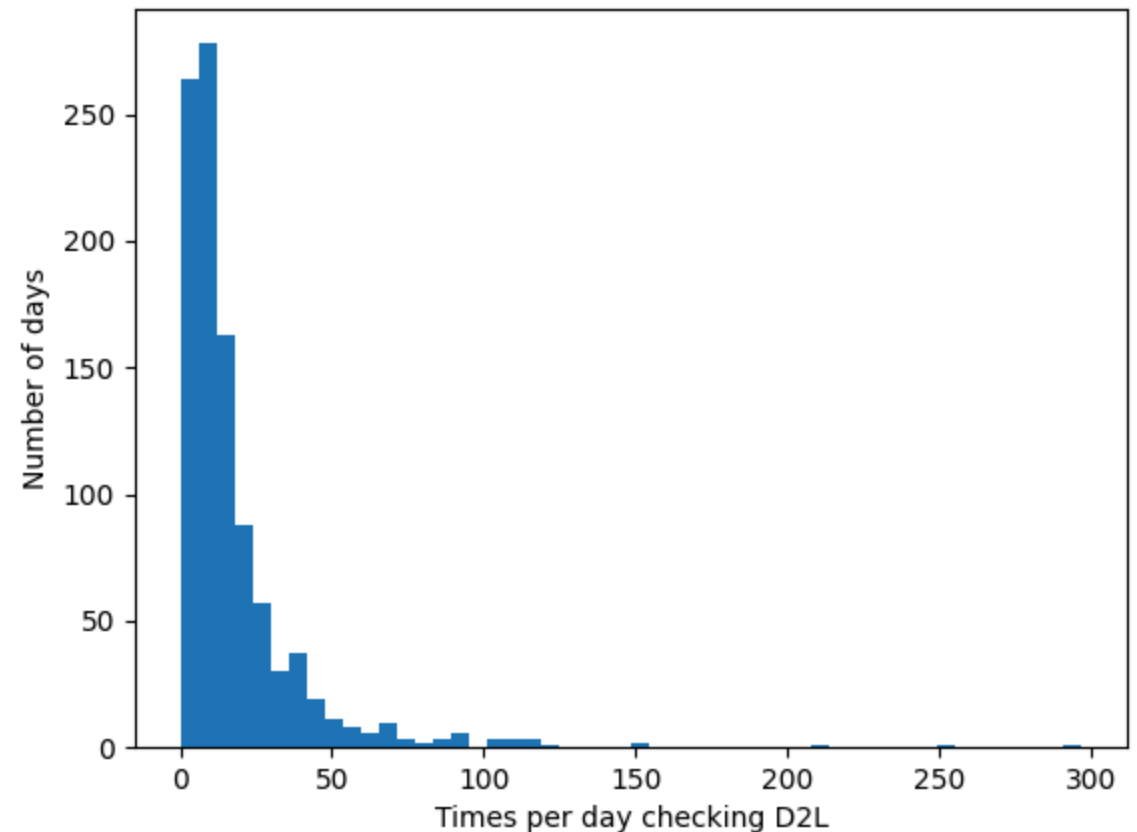
Standardize: $x_{scaled} = \frac{x - \mu_x}{\sigma_x}$

Normalize:

$$x_{scaled} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Nonlinear transforms

- Common case: count data
- Example: Ask 1000 students how often they checked D2L that day
- Not a Gaussian distribution!
- What about the central limit theorem?



Where we left off on January 27

Transformations in training vs inference

- Define functions, e.g.

```
def standardize(X, mu, sigma):  
    return (X - mu) / sigma
```

- Compute scaling parameters **on the training data**, then stash them somewhere:

```
mu = X_train.mean()  
sigma = X_train.std()  
# ... later on, during inference  
X = standardize(X, mu, sigma)
```

What would happen if `standardize` instead computed values on the fly?

What else am I missing here?

Manual approach in the wild

You may run across [magic numbers](#), e.g from the [PyTorch tutorials](#):

```
import torch
from torchvision import transforms, datasets

data_transform = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                          std=[0.229, 0.224, 0.225])
])
```

This really should have a comment! Derived from [ImageNet](#).

An alternative solution: Scikit-learn

Pipeline s

- Hard-coding scaling (and other) parameters is okay, provided you can **justify the choice** and **document where they came from**
- Scikit-learn has a handy [Pipeline](#) class that handles this for you
- Each step in the pipeline has a `fit` and `transform` method
 - `fit` computes parameters from the training data
 - `transform` applies the transformation
 - `fit_transform` does both -- **only use on training data!**
- You can call these functions on the whole pipeline to fit or apply all in one go

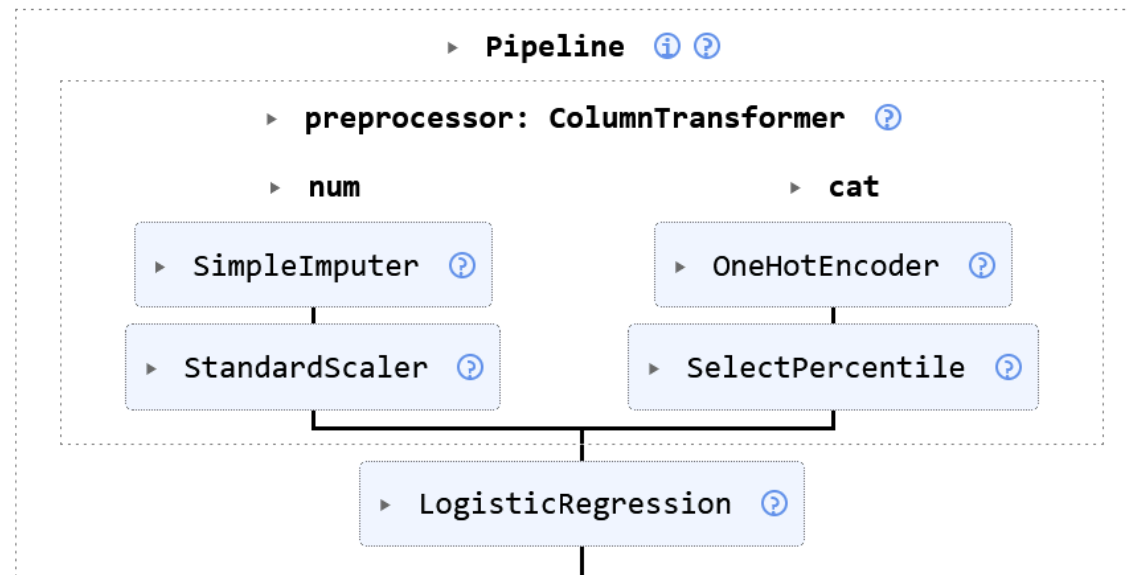
Pipeline

- The `Pipeline` class chains things together in a linear fashion
- Output from one step is input to the next
- `make_pipeline` is slightly simpler syntax (no names needed)

```
from sklearn.pipeline import make_pipeline, Pipeline
pipeline = make_pipeline(
    StandardScaler(),
    SGDClassifier()
)
# or, equivalently:
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('sgd', SGDClassifier())
])
```

ColumnTransformer

- The [ColumnTransformer](#) selects specific features and processes in parallel
- Most of the time mixed dataset need this



1:many transformations

- So far our numeric transformations have been 1:1
- 1:many creates multiple features from 1, like binning
- **Basis expansion** introduces nonlinearity to a continuous feature, e.g.:

$$x \rightarrow [\beta_1 x, \beta_2 x^2, \beta_3 x^3]$$

- More exciting is the **spline** version where you can define different polynomials for different ranges of x

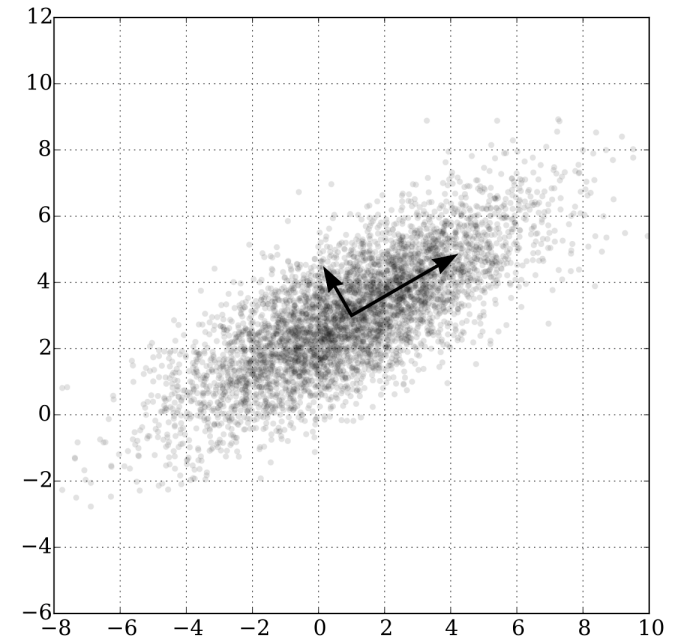
many:many transformations

- Linear projection methods create new features:

$$X^* = XA$$

where A is the *projection matrix* and X^* is the transformed data

- We can use this for **dimensionality reduction** by only keeping some of X^*
- Example: Principle Component Analysis (PCA) finds the set of orthogonal vectors that explain the most variance



Summary

- Numerical data often needs to be transformed to fit model assumptions
- Standardizing (and maybe normalizing) are very common
- Nonlinear transformations can also be beneficial
- Dimensionality reduction can help with model or visualization
- As usual, **let the data be your guide**

Coming up next

- Assignment 1 - due Friday, ish (Monday is fine, and let me know if you need more than that)
- Lab: practice with transformation pipelines
- Next topics: Missing data and interaction effects