DEPARTMENT OF BIOMEDICAL ENGINEERING

24AI671- FUNDAMENTALS OF DATA SCIENCE

UNIT II

DATA COLLECTION, CLEANING AND PREPROCESSING

Syllabus topic
Data Acquisition Methods
Data Storage: CSV, SQL, NoSQL
Missing Data Handling
Outlier Detection
Noise Reduction
Feature Engineering
Data Transformation and Scaling
Data Wrangling Tools – Python, Pandas, NumPy
Unit summary, revision table and question bank

6. FEATURE ENGINEERING

A **feature** is an individual measurable property of the data used as input to a model – for example age, salary or temperature. **Feature engineering** is the process of **using domain knowledge to create, transform, select and extract features** from raw data so that machine-learning algorithms work better. A simple model with well-designed features often beats a complex model fed with raw data; this is why feature engineering is said to be the **most time-consuming and most valuable part of a data-science project**.

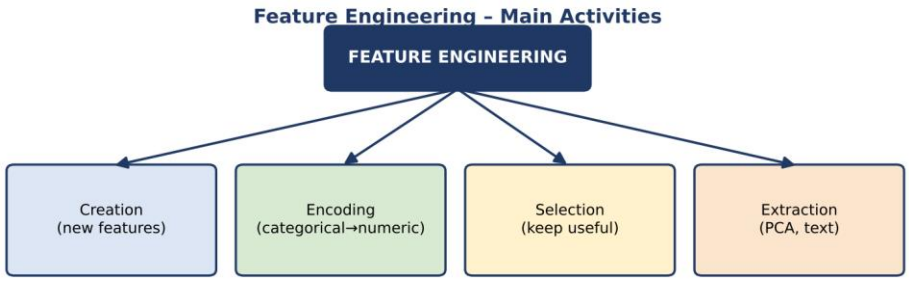


Fig 6.1 Main activities of feature engineering

Why raw data is often not enough. Suppose we want to predict the electricity demand of a building from a column called *timestamp* such as '2026-03-08 19:30'. A model cannot learn anything from this text directly. After feature engineering we extract **hour = 19, day of week = Sunday, is_weekend = 1, month = 3** – and now the model can learn that demand is high in the evening and different on weekends. The raw data and the algorithm are the same; only the **representation** has improved, and so has the result. This is why feature engineering is described as an **art that combines domain knowledge with creativity**.

6.1 Importance of Feature Engineering

- Improves the **accuracy** of models by giving them more informative inputs.
- Converts data into the **numeric form** that algorithms require (text and categories cannot be used directly).
- **Reduces overfitting and training time** by removing irrelevant and repeated features.
- Makes models **simpler and more interpretable**.

6.2 Feature Creation (Construction)

New features are made by combining or transforming existing columns using domain knowledge.

- **Domain-derived features** – BMI = weight / height², age from date of birth, speed = distance / time, power = voltage × current.
- **Date-time features** – extract year, month, day, weekday, hour, is-weekend, is-holiday, season from a timestamp.
- **Interaction features** – product or ratio of two features, e.g. price × quantity, debt ÷ income.
- **Polynomial features** – squares and cubes (x^2, x^3) to capture curved relationships.
- **Aggregate features** – group statistics such as average spend per customer, number of orders last month.
- **Binning / discretisation** – convert a continuous value into groups, e.g. age → child, youth, adult, senior.
- **Text features** – word counts, Bag-of-Words, TF-IDF, text length.

Code

```
import pandas as pd

p = pd.DataFrame({"h": [170, 160], "w": [68, 48]})              # cm, kg
p["BMI"] = (p["w"] / (p["h"] / 100) ** 2).round(1)  # derived feature
print(p)

dt = pd.DataFrame({"date": pd.to_datetime(["2026-01-15", "2026-03-08"])})
dt["year"]    = dt["date"].dt.year
dt["month"]   = dt["date"].dt.month
dt["weekday"] = dt["date"].dt.day_name()
dt["weekend"] = dt["date"].dt.dayofweek >= 5  # Sat/Sun -> True
print(dt)

ages = pd.Series([4, 17, 35, 62])
print(pd.cut(ages, bins=[0, 12, 19, 59, 120],
             labels=["child", "teen", "adult", "senior"]).tolist())
```

Output

```
   h   w   BMI
0  170  68  23.5
1  160  48  18.7

       date  year  month      weekday  weekend
0 2026-01-15  2026      1     Thursday    False
1 2026-03-08  2026      3       Sunday     True

['child', 'teen', 'adult', 'senior']
```

6.3 Encoding Categorical Variables

Machine-learning algorithms need numbers, so categorical text must be **encoded**.

Encoding	Description	Example	Use when
Label encoding	Each category gets an integer 0, 1, 2…	Blue=0, Green=1, Red=2	Target labels; tree-based models
Ordinal encoding	Integers that follow the natural order	Low=0, Medium=1, High=2	Ordinal data
One-hot encoding	One binary column per category	Colour_Red, Colour_Green, Colour_Blue	Nominal data with few categories
Dummy encoding	One-hot with one column dropped (k−1 columns)	Drop Colour_Blue	Linear models – avoids dummy-variable trap
Frequency / count encoding	Replace category by how often it occurs	City → number of customers	High-cardinality columns

Encoding	Description	Example	Use when
Target (mean) encoding	Replace by the average of the target for that category	City → average sales	High cardinality; risk of leakage
Binary / hash encoding	Convert the code to binary digits across a few columns	Category 5 → 101	Very many categories

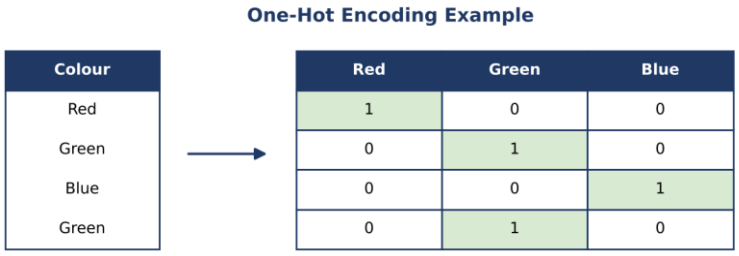


Fig 6.2 One-hot encoding of a Colour column

Code

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder, OrdinalEncoder, OneHotEncoder

t = pd.DataFrame({"Colour": ["Red", "Green", "Blue", "Green"]})
print(pd.get_dummies(t, columns=["Colour"], dtype=int))  # one-hot
# dummy
print(pd.get_dummies(t, columns=["Colour"], drop_first=True, dtype=int))

print(LabelEncoder().fit_transform(["Red", "Green", "Blue", "Green"]))
oe = OrdinalEncoder(categories=[["Low", "Medium", "High"]])
print(oe.fit_transform([["Low"], ["High"], ["Medium"]]).ravel())
```

Output

```
   Colour_Blue  Colour_Green  Colour_Red
0            0             0           1
1            0             1           0
2            1             0           0
3            0             1           0

   Colour_Green  Colour_Red
0             0           1
1             1           0
2             0           0
3             1           0

[2 1 0 1]
[0. 2. 1.]
```

More about frequency and target encoding. A column such as *City* may have hundreds of values, and one-hot encoding would create hundreds of columns. **Frequency encoding** replaces each city by how many times it appears (Chennai → 250). **Target encoding** replaces it by the average of the target variable for that category – for example if the sales in Chennai are 10 and 12, Chennai is encoded as 11. Target encoding is powerful but it uses the answer column, so it must be computed **only on training data (preferably with cross-validation)**; otherwise the model sees the target through the feature and the results are falsely good (**target leakage**).

6.4 Feature Selection

Feature selection chooses the **most useful subset of the original features** and discards irrelevant or redundant ones. It reduces overfitting, speeds up training and improves interpretability. It is also a defence against the **curse of dimensionality** – as the number of features grows, the data becomes sparse and distance-based algorithms lose meaning.

Approach	How it works	Examples	Pros / Cons
Filter methods	Score each feature by a statistical test, independent of the model	Correlation, Chi-square, ANOVA F-test, mutual information, variance threshold	Fast, simple / ignores feature interactions
Wrapper methods	Train a model repeatedly on different subsets and pick the best subset	Forward selection, backward elimination, Recursive Feature Elimination (RFE)	Accurate / slow, risk of overfitting
Embedded methods	Selection happens during model training	Lasso (L1) regression, tree feature importance, Elastic Net	Good balance / model specific

Using correlation for selection. The correlation coefficient r lies between −1 and +1 and measures the **linear relationship** between two numeric columns: +1 means they rise together perfectly, −1 means one falls as the other rises, and 0 means no linear relationship. A common rough guide is $|r| < 0.3$ weak, 0.3–0.7 moderate and > 0.7 strong. Features that are **strongly correlated with the target** are useful, whereas two features that are **strongly correlated with each other** (e.g. height in cm and height in inches) carry the same information, so one of them can be dropped. In Pandas, `df.corr(numeric_only=True)` gives the whole correlation matrix. Remember that correlation shows association, **not causation**.

- **Remove constant and near-constant features** (variance close to 0).
- **Remove highly correlated features** ($|r| > 0.9$) – keep only one of the pair.
- **Keep features that are strongly related to the target.**

Code

```
from sklearn.datasets import load_iris
from sklearn.feature_selection import SelectKBest, f_classif, RFE
from sklearn.linear_model import LogisticRegression

X, y = load_iris(return_X_y=True)                # 4 features

sel = SelectKBest(score_func=f_classif, k=2).fit(X, y)   # filter method
print("SelectKBest keeps:", sel.get_support().tolist())

# wrapper
rfe = RFE(LogisticRegression(max_iter=500), n_features_to_select=2).fit(X, y)
print("RFE keeps:", rfe.support_.tolist())
```

Output

```
SelectKBest keeps: [False, False, True, True]
RFE keeps: [False, False, True, True]
```

Both methods keep the last two features – **petal length and petal width** – of the Iris dataset.

6.5 Feature Extraction and PCA

Feature extraction creates **new, fewer features by combining the original ones**; the new features are different from the original columns. The most popular technique is **Principal Component Analysis (PCA)**, an unsupervised method that projects the data onto new axes called **principal components**, arranged so that the first component captures the maximum variance, the second captures the next largest variance at 90° to the first, and so on.

Intuition behind PCA. Imagine plotting the height and weight of students. The points form a slanted, elongated cloud because tall students are usually heavier. Instead of using the two original axes, PCA draws a **new axis along the direction in which the cloud is longest** (the first principal component – roughly 'body size') and a second axis at right angles to it (which captures only small differences). If almost all the variation is along the first axis, one number per student is enough, and we have reduced two features to one with very little loss of information. In the Iris example below, four measurements are compressed into two components that keep 97.8% of the information, which also makes it possible to **plot the data in 2-D**. PCA components are combinations of the original features, so they are harder to explain; and since PCA depends on variance, the data must be **standardised first**.

Steps of PCA: (1) standardise the data, (2) compute the covariance matrix, (3) find its eigenvalues and eigenvectors, (4) sort the eigenvectors by decreasing eigenvalue, (5) keep the top k eigenvectors as components, and (6) project the data on them.

Code

```
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris

X, _ = load_iris(return_X_y=True)          # 150 samples x 4 features
pca = PCA(n_components=2)
```

```
Z = pca.fit_transform(X)                    # reduced to 2 features
print(Z.shape, pca.explained_variance_ratio_.round(3))
```

Output

```
(150, 2) [0.925 0.053]
```

The first two components together retain **97.8%** (92.5% + 5.3%) of the information of the four original features.

Point	Feature Selection	Feature Extraction
Idea	Choose a subset of original features	Build new features from the original ones
Result	Original columns kept (interpretable)	New columns (components) – harder to interpret
Examples	Correlation filter, RFE, Lasso	PCA, LDA, autoencoders
Use	Remove useless or repeated columns	Compress many correlated columns

6.6 Practical Guidelines

1. Understand the domain – good features usually come from domain knowledge.
2. Handle missing values and outliers **before** creating features.
3. Create features, then **remove** the useless ones – more features are not always better.
4. Fit all encoders, scalers and selectors **only on training data** to avoid leakage.
5. Evaluate every new feature using **cross-validation**.

7. DATA TRANSFORMATION AND SCALING

Data transformation converts data from one form into another that is more suitable for analysis, and **scaling** is the special case in which numeric features are brought to a **common range or scale**. Real data has features in very different ranges – for example **age (18–60)** and **salary (₹ 15,000 – ₹ 1,50,000)**. Algorithms that use distances or gradients would be dominated by the larger-valued feature unless the features are scaled.

7.1 Why Scaling is Needed

- **Distance-based algorithms** (KNN, K-Means, SVM, hierarchical clustering) treat a large-range feature as more important. For two people with ages 25 and 45 and salaries ₹ 30,000 and ₹ 31,000, the salary difference (1,000) completely hides the age difference (20) in the Euclidean distance.
- **Gradient-descent algorithms** (linear / logistic regression, neural networks) converge much faster when features have similar scales.

- **PCA and regularised models** (Ridge, Lasso) are affected by feature variance and need scaled data.
- **Tree-based models** (Decision Tree, Random Forest, XGBoost) are **not affected** by scaling because they use split thresholds.

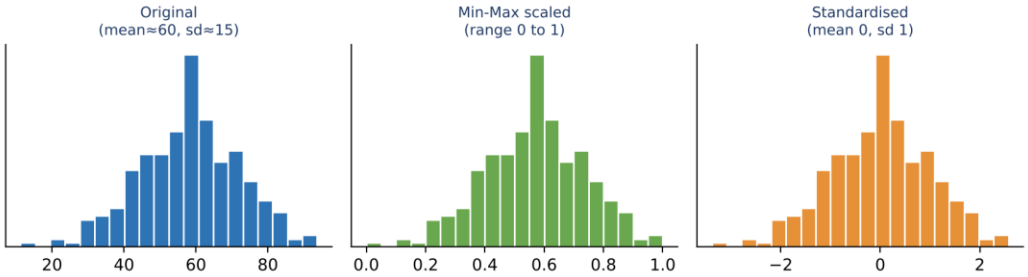


Fig 7.1 Effect of Min-Max scaling and standardisation on the same data (shape unchanged, only the scale changes)

A numerical example of why scaling matters

Three people have (age, salary): **A = (25, ₹ 30,000)**, **B = (45, ₹ 31,000)** and **C = (26, ₹ 40,000)**. Without scaling, the Euclidean distance A–B = $\sqrt{(20^2 + 1000^2)}$ = **1000.2** and A–C = $\sqrt{(1^2 + 10000^2)}$ = **10000.0**. A KNN model would say that B is ten times closer to A than C is, even though B is 20 years older and C is only 1 year older – the salary completely dominates. Now apply Min-Max scaling using age range 18–60 and salary range ₹ 15,000 – ₹ 1,50,000: A = (0.167, 0.111), B = (0.643, 0.119), C = (0.190, 0.185). The distances become A–B = **0.476** and A–C = **0.078**, so C is correctly found to be the person most similar to A. Scaling lets **every feature contribute fairly**.

7.2 Scaling Techniques

7.2.1 Min-Max scaling (normalisation)

Rescales the values to a **fixed range, usually 0 to 1**, by using the minimum and maximum of the column.

$$x' = (x - x_min) / (x_max - x_min)$$

Example: values 10, 20, 30, 40, 50 (min = 10, max = 50). For 35 → (35 − 10) / (50 − 10) = **0.625**. The scaled column is 0, 0.25, 0.5, 0.75, 1. To scale to another range [a, b] use $x' = a + (x - x_min)(b - a)/(x_max - x_min)$.

Advantage: keeps the original shape and gives a bounded range, good for neural networks and image pixels. **Disadvantage:** very sensitive to outliers – one extreme value squeezes all other values into a tiny range.

7.2.2 Standardisation (Z-score scaling)

Transforms the data so that it has **mean = 0 and standard deviation = 1**.

$$z = (x - \mu) / \sigma$$

Example: for 10, 20, 30, 40, 50, mean μ = 30 and σ = √200 = 14.14. Then z(10) = (10 − 30)/14.14 = **−1.414**, z(20) = **−0.707**, z(30) = **0**, z(40) = **0.707**, z(50) = **1.414**.

Advantage: not bounded, less affected by outliers than Min-Max, preferred for most algorithms (SVM, logistic regression, PCA, K-Means). **Disadvantage:** the result is not in a fixed range.

7.2.3 Robust scaling

Uses the **median and IQR**, which are not affected by outliers.

$$x' = (x - \text{median}) / \text{IQR}$$

For 10, 20, 30, 40, 500 the median is 30 and IQR = 40 − 20 = 20, so the scaled values are −1, −0.5, 0, 0.5 and 23.5. The ordinary values stay in a sensible range, while the outlier stays clearly visible instead of squeezing the others.

7.2.4 Other scaling methods

Method	Formula / idea	Remark
Max-Abs scaling	$x' = x / \lvert x_max \rvert$ (range −1 to 1)	Keeps sparse data sparse; no centring
Decimal scaling	$x' = x / 10^j$ where j is the smallest integer so that $\max(\lvert x' \rvert) < 1$	Moves the decimal point; e.g. 986 → 0.986
Unit-vector (L2) normalisation	Each row is divided by its length: $x' = x / \lVert x \rVert$	Used for text vectors, cosine similarity

7.3 Data Transformation Techniques

Many features are **skewed** (a long tail on one side), such as income, house price and population. Several algorithms (linear models, LDA) work best when features are close to a normal distribution. Transformations **change the shape** of the distribution.

Skewness. A distribution is **right (positively) skewed** when it has a long tail towards large values – income, house price and waiting time are typical examples – and in such data the **mean is greater than the median**. A **left (negatively) skewed** distribution has a long tail towards small values, and the mean is smaller than the median. Many statistical methods assume a symmetric, bell-shaped distribution, so skewed features are transformed: the **log** or **square-root** transformation pulls in the long right tail, and **Box-Cox / Yeo-Johnson** choose the best power automatically. After a transformation the predictions can be converted back with the inverse function (for log, use the exponential `np.expm1`).

Transformation	Formula	Use
Log transform	$x' = \log(x)$ or $\log(1 + x)$	Strong right skew; compresses large values; x must be positive
Square root	$x' = \sqrt{x}$	Moderate right skew; count data
Reciprocal	$x' = 1 / x$	Very strong skew
Square / cube	$x' = x^2, x^3$	Left-skewed data
Box-Cox	$x' = (x^k - 1) / k$, $k \neq 0$; $\log(x)$ when $k = 0$	Finds the best power automatically; positive data only

Transformation	Formula	Use
Yeo-Johnson	Generalisation of Box-Cox	Works for zero and negative values also
Quantile transform	Maps data to a uniform or normal distribution by rank	Heavy-tailed data; strongly reduces outliers

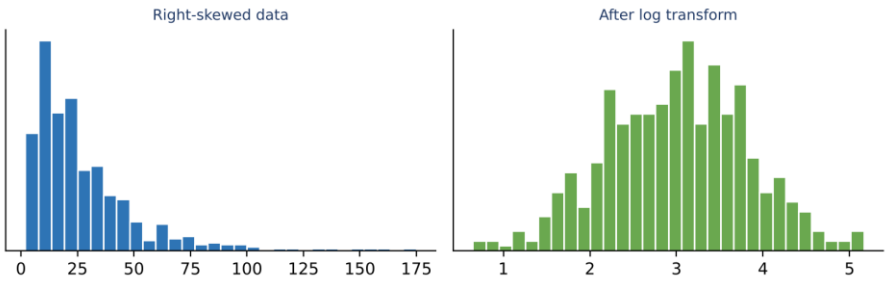


Fig 7.2 A log transformation reduces right skew

Other data-transformation operations

- **Discretisation (binning)** – converting continuous values to intervals (age groups, marks grades).
- **Aggregation** – summarising data, e.g. daily readings to monthly averages.
- **Attribute construction** – new columns from existing ones (see Feature Engineering).
- **Encoding** – category to number conversion (see Section 6.3).
- **Reshaping** – converting between wide and long formats using `melt()` and `pivot()`.

7.4 Comparison of Scalers

Scaler	Output range	Affected by outliers?	Typical use
Min-Max (MinMaxScaler)	0 to 1 (fixed)	Very much	Neural networks, image data, KNN when no outliers
Standard (StandardScaler)	Unbounded; mean 0, sd 1	Moderately	SVM, logistic regression, PCA, K-Means
Robust (RobustScaler)	Unbounded; median 0	Hardly	Data with many outliers
Max-Abs (MaxAbsScaler)	−1 to 1	Very much	Sparse data
Normalizer	Unit length per row	Not applicable	Text and cosine similarity

Terminology note

The words **normalisation** and **standardisation** are used loosely. In most books, **normalisation = Min-Max scaling to [0, 1]**, and **standardisation = Z-score scaling**. In

statistics, ‘normalising’ may also mean making data follow a normal distribution (the transformations of Section 7.3).

7.5 Python Implementation

Code

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

a = np.array([[10], [20], [30], [40], [50]])
print(MinMaxScaler().fit_transform(a).ravel())
print(StandardScaler().fit_transform(a).ravel().round(3))

b = np.array([[10], [20], [30], [40], [500]])       # contains an outlier
print(RobustScaler().fit_transform(b).ravel())

print(np.log1p([1, 10, 100, 1000]).round(3))        # log(1 + x)
```

Output

```
[0.   0.25 0.5  0.75 1.  ]
[-1.414 -0.707  0.     0.707  1.414]
[-1.  -0.5  0.   0.5 23.5]
[0.693 2.398 4.615 6.909]
```

Avoiding data leakage – fit on train, transform on test

The scaling parameters (min, max, mean, standard deviation) must be **learned only from the training set** and the same values applied to the test set. Doing fit on the whole data lets information from the test set leak into training and gives over-optimistic results.

Code

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

X_train, X_test, y_train, y_test = train_test_split(
      X, y, test_size=0.2, random_state=42)      # 80% train, 20% test

sc = StandardScaler()
X_train_s = sc.fit_transform(X_train)   # learn mean and sd from TRAIN only
X_test_s  = sc.transform(X_test)        # apply the same values to TEST
```

A complete pre-processing pipeline

Scikit-learn’s **Pipeline** and **ColumnTransformer** apply imputation, scaling and encoding in a single, repeatable object and automatically avoid leakage.

Code

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

```
num_cols = ["Age", "Marks"]
cat_cols = ["City"]

num_pipe = Pipeline([("fill", SimpleImputer(strategy="median")),
                     ("scale", StandardScaler())])
cat_pipe = Pipeline([("fill", SimpleImputer(strategy="most_frequent")),
                     ("onehot", OneHotEncoder(handle_unknown="ignore"))])

prep = ColumnTransformer([("num", num_pipe, num_cols),
                          ("cat", cat_pipe, cat_cols)])
X_ready = prep.fit_transform(df)        # cleaned, scaled, encoded array
```

Which scaler should be chosen?

- Data is **bounded and has no major outliers**, or the algorithm needs inputs in 0–1 (neural networks, image pixels) → **Min-Max**.
- Data is roughly **bell-shaped**, or the algorithm assumes centred data (SVM, logistic regression, PCA, K-Means) → **Standardisation**.
- Data has **many outliers** that should not control the scale → **Robust scaling**.
- Data is **sparse** (many zeros, such as text counts) → **Max-Abs** scaling, which keeps the zeros as zeros.

Converting back. After a model predicts in scaled units, the values can be restored to the original units using `scaler.inverse_transform()`. For example, if StandardScaler was fitted on 10, 20, 30, applying `inverse_transform` to the scaled values returns exactly 10, 20, 30. Do not scale the **target** column unless the algorithm needs it, and remember to invert the predictions in that case.

Which algorithms need scaling?

Need scaling	Do not need scaling
KNN, K-Means, hierarchical clustering	Decision Tree
SVM, PCA	Random Forest
Linear / Logistic regression (with regularisation or gradient descent)	Gradient Boosting, XGBoost
Neural networks	Naive Bayes