Preprocessing Techniques

Introduction

Data preprocessing is an essential step in the data science and deep learning pipeline. Before any model can be trained or analysis performed, raw data must be cleaned, organized, and transformed into a format suitable for processing. This is because real-world data is often problematic—filled with missing values, inconsistencies, noise, and irrelevant information.

The goal of data preprocessing is to improve the quality of the data and enhance the performance and accuracy of models built from it. This step typically includes tasks such as handling missing data, removing duplicates, normalizing or scaling values, encoding categorical variables, and selecting relevant features.

A careful prepararion of the data ensures that deep models can learn effectively and deliver reliable insights or predictions. In essence, data preprocessing acts as the foundation upon which successful data-driven solutions are built.

Most preprocessing techniques are data dependent, i.e. taylored to the particularities of the specific data type. In this lab session, we present some of the most frequently-used preprocessing techniques, applicable to most data sets and in the most general terms.

Data Normalization

In deep learning, the performance of a model depends not only on the architecture and training procedure but also on how the data is prepared. One of the most critical preprocessing steps is data normalization, which helps improve training speed, stability, and convergence.
Normalization ensures that the input data to a model is scaled appropriately, so that the network can learn efficiently without suffering from issues like exploding/vanishing gradients, slow convergence, or poor generalization.
Deep learning models, especially those using gradient-based optimization (like stochastic gradient descent), are sensitive to the scale and distribution of input features. When features have different ranges:

- The cost function contours become skewed, making gradient descent inefficient.
- Features with larger values can dominate the learning process.
- Activation functions like sigmoid and tanh can saturate, leading to vanishing gradients.
- Weight initialization becomes less effective.
- Learning rate becomes harder to tune.

Data standardization is a very frequently-used preprocessing step in deep learning pipelines. It refers to transforming input features so that they have a mean of zero and a standard deviation of one. This ensures that the data supplied to a neural network is well-formed, which facilitates faster convergence and better performance during training.

Neural networks are sensitive to the scale and distribution of input data. Without standardization, features with larger numeric ranges can dominate the gradient updates, leading to poor training dynamics. Standardized data helps in:

- Accelerating convergence of optimization algorithms like SGD, Adam, etc.
- Improving numerical stability during backpropagation.
- Preventing some neurons from becoming "stuck" due to large input values (saturation).
- Avoiding bias towards features with larger ranges or magnitudes.

Given a feature vector *x*, standardization transforms it as follows:

$$x_{\text{standardized}} = \frac{x - \mu}{\sigma}$$

Where:

- μ is the mean of the feature x across the training set
- σ is the standard deviation of feature x across the training set

Each feature (dimension) is standardized independently.

Min-max scaling is a preprocessing technique used to rescale the values of features in a dataset so that they fall within a specified range, typically [0,1] or [−1,1]. This method is widely used in deep learning to improve model performance and training stability.

The min-max normalization of a feature *x* is given by:

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

- *x* is the original feature value
- x_{min} and x_{max} are the minimum and maximum values of the feature in the training set
- *x'* is the scaled feature in the [0, 1] range

In more general terms, in order to scale to any range [a, b], the formula becomes:

$$x' = a + \frac{(x - x_{\min})(b - a)}{x_{\max} - x_{\min}}$$

Both approaches, standardization and min-max scaling, are referred to using the term *normalization*, in different contexts. The term *normalization* is used loosely in practice to mean "rescaling the data so that it is more suitable to deep models". Thus, depending on the context, the term can refer to either standardization or min-max scaling. However, the two approaches have disctinct advantages and downsides:

Standardization:

Advantages:

- Centered around 0 - useful for models where weights are initialized around zero (e.g., in many deep learning models).
- Handles outliers better than min-max scaling, since it doesn't compress all data to a limited range.
- Works well when the feature distributions are Gaussian or near-normal.
- Widely used in batch normalization**,** which stabilizes training.

Downsides:

- Not bounded - the output can be any real number, which may not suit activations like sigmoid or tanh that expect input in a small range.
- Sensitive to non-normal distributions: If the data is heavily skewed, standardization may not help much.
- Somewhat affected by outliers, especially if the mean and standard deviation are significantly distorted by extreme values.

Min-max Scaling:

Advantages:

- Keeps values in a fixed range (usually [0, 1] or [-1, 1]), which is beneficial for activations like sigmoid or tanh.
- Fast convergence in gradient descent when used with compatible activation functions.
- Particularly useful when features have known bounds or follow a uniform distribution.

Downsides:

- Highly sensitive to outliers**:** One extreme value can skew the scale and compress the rest of the data.

- Not ideal for data with a non-uniform distribution or data that contains noise or outliers.
- Requires knowing or fixing the min and max values, which can be problematic in online learning or streaming data contexts.

Handling Missing Values

One common issue encountered in real-world datasets is the presence of missing values, which can arise due to data corruption, human error, sensor failures, or unavailability of information at collection time. Thus, handling missing values is an important step in any data preprocessing pipeline, and its importance extends to deep learning just as much as to traditional machine learning. In deep learning, the quality of data has a significant influence on the ability of models to generalize and converge efficiently.

While deep learning frameworks often require complete numerical input tensors, they do not inherently support missing values (e.g., `NaN`s or nulls), so it's essential to preprocess the data to handle these gaps. Among the several strategies available, *value imputation* stands out as one of the most practical and commonly applied methods. In this lab session, we address three on the most common value imputation approaches used in deep learning pipelines: mean imputation, median imputation, and K-Nearest Neighbors (KNN)-based imputation.

While deep learning models such as autoencoders can learn patterns from incomplete data in some semi-supervised setups, most supervised learning scenarios require explicit preprocessing to fill in missing values.

Mean Imputation

Mean imputation replaces missing values in a feature with the mean of the available values for that feature across all instances. For a given vector $X = [x_1, x_2, ..., x_n]$, any missing values are replaces using an estimated value determined as folows:

$$\hat{x}_i = \frac{1}{|X_{\text{obs}}|} \sum_{x_j \in X_{\text{obs}}} x_j$$

Instance	Feature A	Feature A, imputed
1	7.0	7.0
2	NaN	**5.5**
3	5.0	5.0
4	6.0	6.0
5	NaN	**5.5**
6	4.0	4.0

Mean imputation is simple to implement and fast. It maintains the number of instances, which is important for batch-based training in deep learning. Due to its computational efficiency, it is compatible with real-time preprocessing pipelines. However, this approach can distort variance and reduce data variability. It is also sensitive to outliers and it assumes the data is missing completely at random, which is often unrealistic.

Median Imputation

Median imputation replaces missing values with the median of the non-missing values in the feature. The median is the value is the central position of the sorted sequence of the existing feature values:

$$\hat{x}_i = \text{median}(X_{\text{obs}})$$

Median imputation is more robust to outliers than mean iputation. It preserves the central tendency better in skewed distributions. Like mean imputation, it is fast and easy to implement. However, like mean imputation, it reduces data variance, and it is less informative if a large portion of the data is missing.

Instance	Feature A	Feature A, imputed
1	12.0	12.0
2	NaN	**10.0**
3	8.0	8.0
4	15.0	15.0
5	NaN	**10.0**
6	10.0	10.0
7	7.0	7.0

KNN-based Imputation

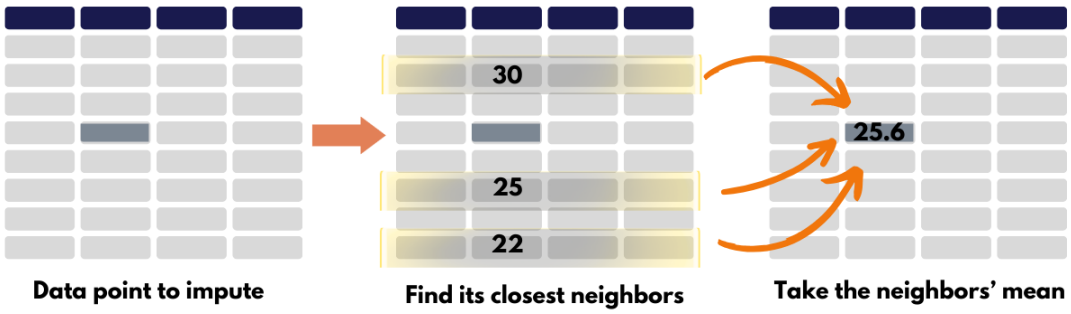
K-Nearest Neighbors (KNN) imputation fills in missing values using the values from the K most similar instances in the dataset, where similarity is measured based on non-missing features:

- For each sample with a missing value, find the K nearest samples based on a chosen distance metric (e.g., Euclidean, Cos similarity).
- Use the mean or median of the corresponding values from the K neighbors for the missing feature.
- Impute the missing value with this aggregated statistic.

For missing feature x_i in instance X, find K nearest neigbors $\{X_1, ..., X_k\}$, then:

$$\hat{x}_i = \frac{1}{K} \sum_{j=1}^{K} x_i^{(j)}$$

Or use the median of $\{x_i^{(j)}\}$



The KNN-based approach takes into account the relationships between features, preserving multivariate structure. It is adaptible to both continuous and categorical data and it is more accurate the simple imputation methods in many situation. However, it is computationally-expensive, especially for large data sets, and requires tuning the parameter K. Also, it is sensitive to the choice of distance metrics and to the scaling of features.

Encoding Categorical Variables

Encoding categorical variables is an important preprocessing step in machine learning and deep learning workflows. In traditional machine learning algorithms, improper encoding can lead to poor performance. In deep learning, where neural networks are capable of handling large-scale and complex data, encoding becomes even more essential, for accuracy, efficiency and convergence.

Categorical variables are variables that represent discrete groups or categories. Unlike numerical variables, they have no inherent ordering (in the case of nominal variables), or the order is non-numeric (in ordinal variables).

Examples:

- Nominal: `"color" = red, blue, green`
- Ordinal: `"education level" = high school, bachelor, master, PhD`

Deep learning models expect numerical input, so these variables must be converted into a suitable format before supplying them to the model.

One-Hot Encoding

One-hot encoding converts each category into a binary vector. Each position in the vector corresponds to one possible category. The category is marked with a 1, while all others are marked 0.

Considering a nominal feature “Color” with three possible values: {Red, Green, Blue}, one hot encoding involves the generation of three new binary features, in pace of the original nominal feature:

Color	Color_Red	Color_Blue	Color_Green
Red	1	0	0
Green	0	1	0
Blue	0	0	1

Nominal features that don't have an implicit ordering are not commonly encoded as integers (Such as Red = 0, Green = 1, Blue = 2), because the model might infer a false relationship between the values (i.e., that Green is "greater than" Red), or that there is a linear progression of the values, which is meaningless for categories such as colors. Furthermore, neural networks rely heavily on numeric calculations, such as computing dot products and Euclidean distances. If integer labels are used, these computations may falsely reflect proximity or similarity between categories, which is not valid for nominal data. Transforming a single nominal feature into multiple binary variables means that each new feature corresponds to one and only one category. These binary features are mutually exclusive, which avoids unwanted correlations and ensures the network can learn separate weights for each category.

Considering the following data set:

Customer_ID	Total_Transactions	Favorite_Product
201	15	Electronics
202	9	Groceries
203	23	Clothing
204	5	Electronics
205	12	Groceries

The Favorite_Product feature is categorical, specifically, nominal. We one-hot-encode the Favorite_Product category into three binary features: Electronics, Clothing, Groceries. In the resulting dataset, the nominal categories are transformed into mutually exclusive binary features, adequate for serving as input into a deep learning model:

Customer_ID	Total_Transactions	Electronics	Clothing	Groceries
201	15	1	0	0
202	9	0	0	1
203	23	0	1	0
204	5	1	0	0
205	12	0	0	1

Tasks

For completing the tasks, the code samples that are provided with this documentation can be used as starting points.

1. Apply standardization and min-max scaling to a dataset (commonly, the features are scaled to [0, 1]). Compare convergence in terms of number of required epochs and loss behavior, as well as resulting accuracy, for:
    - the original data
    - the standardized data
    - the scaled data

2. Apply mean, median and KNN-based value imputation to a data set with missing values. Compare convergence when using the three approaches. Study the effectiveness of the three approaches when values are missing in different amounts (5% of values are missing, 10% of the values are missing etc.).
   Compare the three methods to the approach where the instances with missing values are simply removed from the data set.

3. Apply one-hot encoding to a data set where some of the features are categorical. Train and evaluate a neural network-based classifier on the resulting data set. Note: a categorical feature with only two values can be encoded using labels {0, 1} – we create separate features when the categorical variable has 3 values or more.