

Machine Learning Engineering

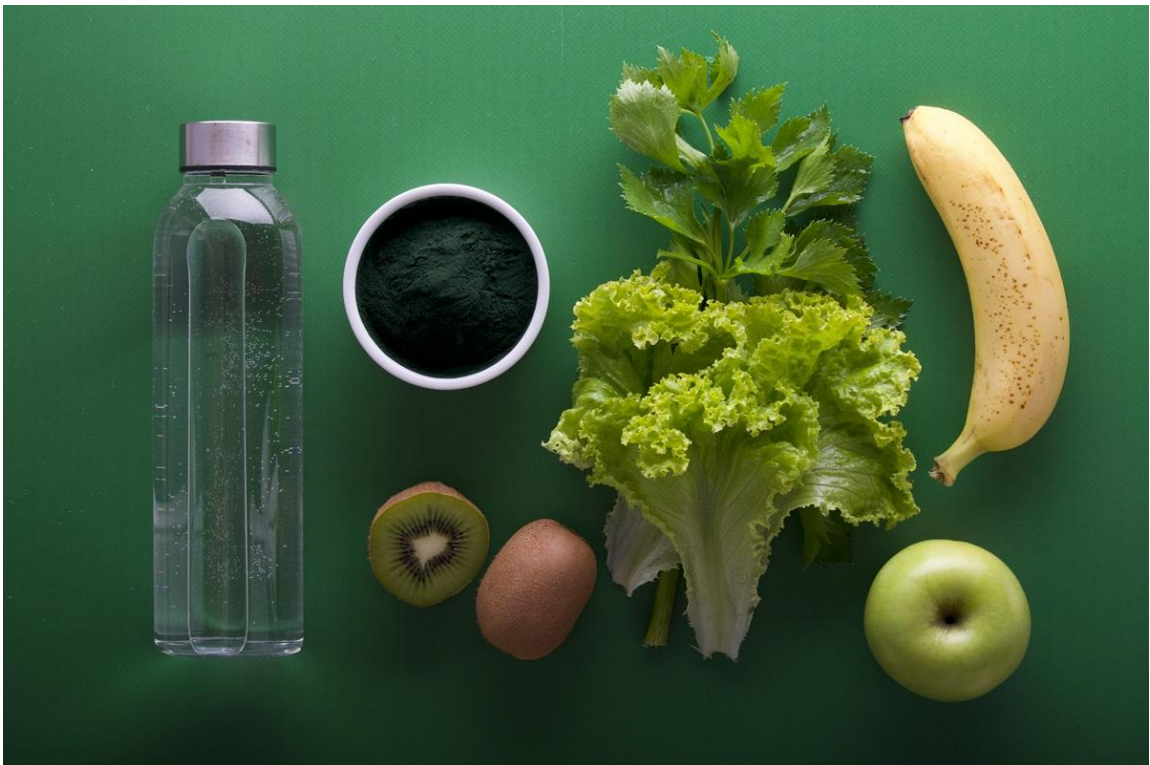
CSI 4106 - Fall 2025

Marcel Turcotte

Version: Oct 5, 2025 12:29

Preamble

Message of the Day



[uOttawa developing AI-powered tool to personalize diets for people living with inflammatory bowel diseases \(IBD\)](#), 2025-09-23.

The Weston Family Foundation has awarded a \$10 million grant to a collaborative research project between the University of Ottawa and the University of Alberta. This initiative aims to conduct research into inflammatory bowel disease (IBD).

In Canada, over 300,000 individuals suffer from chronic IBD, which includes conditions such as Crohn's disease and ulcerative colitis. The project will leverage data from RapidAIM, a platform developed at the University of Ottawa designed to quickly evaluate the effects of various compounds on individual microbiomes.

A key objective of this research is to lay the groundwork for training an artificial intelligence (AI) model. This model will be capable of generating personalized dietary recommendations, tailored to the unique microbiome profiles of individuals with IBD.

Learning Objectives

- **Explain** feature extraction, **compare** encoding methods, and **justify** choices based on data nature.
- **Apply** normalization/standardization for feature scaling and **handle** missing values using imputation.
- **Define** class imbalance, **explore** solutions like resampling and SMOTE, and ensure proper application.
- **Apply** concepts to real-world datasets, analyze results, and understand the machine learning pipeline.
- **Recognize** dataset size impact, **discuss** data augmentation, and **explore** data effectiveness in ML.

Machine Learning Engineering

Machine Learning Engineering

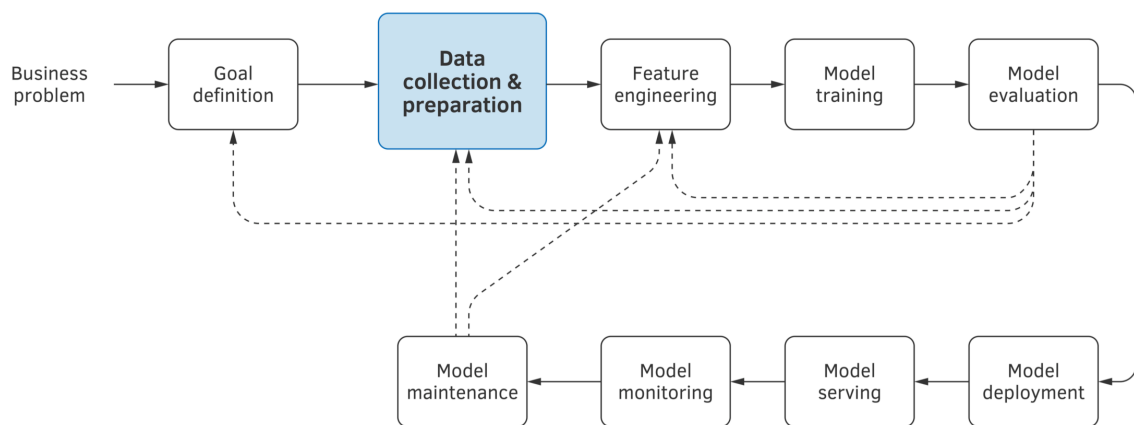


Figure 3.1: Machine learning project life cycle.

Attribution: A. Burkov (2020), Figure 3.1

The courses [SDS 3386: Data Science Lab](#) and [CSI 4142: Fundamentals of Data Science](#) provide comprehensive coverage of machine learning engineering. For those not enrolled in these courses, we provide an overview of essential concepts.

As illustrated, the life cycle of a machine learning project typically involves multiple iterations. Each cycle enhances understanding of the problem and reveals limitations.

Insufficient data quality often precipitates project failure.

Characteristics of High-Quality Data

- **Volume:** Is the dataset sufficiently large to support generalization?
- **Relevance:** Do the features accurately model the problem?
- **Representation:** Does the data reflect the conditions expected in production?
 - Are these features obtainable from real-world inputs?
 - Does the dataset adequately cover and represent feature examples?
- **Bias:** Is the data free from bias?
- **Label Consistency:** Are the labels consistent throughout the dataset?

Machine learning Project Cycle

Gathering data and extracting features are the first two steps of a machine learning project.

1. Gather **adequate** data.
2. Extract **features** from the raw data:
 - This process is **labor-intensive**.
 - It necessitates **creativity**.
 - **Domain knowledge** is highly beneficial.

Encoding

Why?

After completing Assignment 1, you generated a `.csv` file containing cleaned data.

Why would additional steps be necessary?

Dataset - Adult

```
In [2]: import numpy as np
np.random.seed(42)

from sklearn.datasets import fetch_openml

adult = fetch_openml(name='adult', version=2)

print(adult.DESCR)
```

The 'Adult' dataset contains several attributes characterized by categorical values. This dataset will serve as a basis for a brief discussion on encoding these categorical values.

Adult - Workclass

```
In [3]: adult.data['workclass'].unique()

['Private', 'Local-gov', NaN, 'Self-emp-not-inc', 'Federal-gov', 'State-go
v', 'Self-emp-inc', 'Without-pay', 'Never-worked']
Categories (8, object): ['Federal-gov', 'Local-gov', 'Never-worked', 'Privat
e', 'Self-emp-inc', 'Self-emp-not-inc', 'State-gov', 'Without-pay']
```

Adult - Education

```
In [4]: adult.data['education'].unique()

['11th', 'HS-grad', 'Assoc-acdm', 'Some-college', '10th', ..., 'Assoc-voc',
'9th', '12th', '1st-4th', 'Preschool']
Length: 16
Categories (16, object): ['10th', '11th', '12th', '1st-4th', ..., 'Masters',
'Preschool', 'Prof-school', 'Some-college']
```

Adult - Marital Status

```
In [5]: adult.data['marital-status'].unique()

['Never-married', 'Married-civ-spouse', 'Widowed', 'Divorced', 'Separated',
'Married-spouse-absent', 'Married-AF-spouse']
Categories (7, object): ['Divorced', 'Married-AF-spouse', 'Married-civ-spous
e', 'Married-spouse-absent', 'Never-married', 'Separated', 'Widowed']
```

Categorical Data

Key Points on Data Representation

- **Numerical Representation:** Some learning algorithms require data to be in numerical form.

Encoding Methods

Consider the `workclass` attribute, which has 8 distinct values like 'Federal-gov', 'Local-gov', and so on.

- **Which** encoding method is preferable and **why?**
 1. `w` = 1, 2, 3, 4, 5, 6, 7, or 8
 2. `w` = [0,0,0], [0,0,1], [0,1,0], ..., or [1,1,1]
 3. `w` = [1,0,0,0,0,0,0,0], [0,1,0,0,0,0,0,0], ..., or [0,0,0,0,0,0,0,1]

Encoding for Categorical Data

One-Hot Encoding: This method should be preferred for categorical data.

- **Increases Dimensionality:** One-hot encoding increases the dimensionality of feature vectors.
- **Avoids Bias:** Other encoding methods can introduce biases.
- **Example of Bias:** Using the first method, `w = 1, 2, 3`, etc., implies that 'Federal-gov' and 'Local-gov' are similar, while 'Federal-gov' and 'Without-pay' are not.
- **Misleading Similarity:** The second method, `w = [0,0,0], [0,0,1]`, etc., might mislead the algorithm by suggesting similarity based on numeric patterns.

Definition

One-Hot Encoding: A technique that converts categorical variables into a binary vector representation, where each category is represented by a vector with a single '1' and all other elements as '0'.

There are numerous other vector representations, among which “**embeddings**” stand out for their remarkable efficiency.

OneHotEncoder

```
In [6]: from numpy import array
from sklearn.preprocessing import OneHotEncoder

work = adult.data[['workclass']]

onehot_encoder = OneHotEncoder()

onehot_encoder.fit(work)
values_encoded = onehot_encoder.transform(work)

for i in range(5): print(values_encoded.toarray()[i])
```

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

Consistency is Key: Ensure you use the same encoding on: Validation Set, Test Set, and Production Data.

A student from my research group faced a challenging debugging issue. They mistakenly created a new encoder for the test set using

`onehot_encoder.fit(X_test['some_attribute'])`, which produced a vector representation different from the one used during training. Consequently, the results on the training set were poor, while the results on the training set appeared satisfactory.

While `Pandas` offers a method called `get_dummies()` for one-hot encoding, it is important to note the following distinctions:

- **Category Memory:** `OneHotEncoder` retains the categories it was trained on, whereas `get_dummies()` does not.
- **Consistency in Production:** It is crucial to use the same encoding scheme in production as was used during training to ensure accurate results.
- **Vector Length Discrepancies:** If `get_dummies()` encounters a different number of categories in new data, it will produce vectors of varying lengths, leading to potential errors.
- **Handling Missing Values:** When `get_dummies()` processes missing values, it generates an additional column to accommodate them.

Ensuring consistency in encoding across **training**, **validation**, and **production** datasets is essential to maintain the integrity and accuracy of your machine learning models.

Case Study

- **Dataset:** [Heart Disease](#)
 - **Examples:** 303, **features:** 13, **target:** Presence/absence of disease
- **Categorical Data:**
 - **sex:** 1 = male, 0 = female
 - **cp (chest pain type):**
 - 1: Typical angina
 - 2: Atypical angina
 - 3: Non-anginal pain
 - 4: Asymptomatic
 - **Other:** 'fbs', 'restecg', 'exang', 'slope', 'thal'

To simplify the analysis: Examples with missing values were dropped, no hyperparameter tuning was performed, numerical values were scaled for solver convergence.

Here are some suggestions for further investigation:

- Assess the impact of omitting missing values on the dataset.
- Implement hyperparameter tuning to determine whether L_1 or L_2 regularization enhances model performance.

Case Study

```
In [7]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```

from sklearn.datasets import fetch_openml
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

# Load the 'Heart-Disease' dataset from OpenML
data = fetch_openml(name='Heart-Disease', version=1, as_frame=True)
df = data.frame

# Replace '?' with NaN and convert columns to numeric
for col in df.columns:
    df[col] = pd.to_numeric(df[col], errors='coerce')

# Drop rows with missing values
df.dropna(inplace=True)

# Define features and target
X = df.drop(columns=['target'])
y = df['target']

# Columns to encode with OneHotEncoder
columns_to_encode = ['sex', 'cp', 'fbs', 'restecg', 'exang', 'slope', 'thal']

# Identify numerical columns
numerical_columns = X.columns.difference(columns_to_encode)

# Split the dataset into training and testing sets before transformations
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Apply OneHotEncoder and StandardScaler using ColumnTransformer
column_transformer = ColumnTransformer(
    transformers=[
        ('onehot', OneHotEncoder(), columns_to_encode),
        ('scaler', StandardScaler(), numerical_columns)
    ]
)

# Fit the transformer on the training data and transform both training and test sets
X_train_processed = column_transformer.fit_transform(X_train)
X_test_processed = column_transformer.transform(X_test)

# Initialize and train logistic regression model
model = LogisticRegression(max_iter=1000)
model = model.fit(X_train_processed, y_train)

```

In the context of using `ColumnTransformer`, the second element of the triplets, typically an estimator, can also be replaced with the options `drop` or `passthrough`. The `drop` option excludes the specified column from the transformation process, while `passthrough` retains the column in its original state without any modifications.

Case study - results

```
In [8]: # Predict and evaluate the model
y_pred = model.predict(X_test_processed)

print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0.0	0.87	0.93	0.90	29
1.0	0.93	0.88	0.90	32
accuracy			0.90	61
macro avg	0.90	0.90	0.90	61
weighted avg	0.90	0.90	0.90	61

Case study - chest pain (cp)

```
In [9]: # Retrieve feature names after transformation using get_feature_names_out()
feature_names = column_transformer.get_feature_names_out()

# Get coefficients and map them to feature names
coefficients = model.coef_[0]

# Create a DataFrame with feature names and coefficients
coef_df = pd.DataFrame({
    'Feature': feature_names,
    'Coefficient': coefficients
})

# Display coefficients associated with 'cp'
cp_features = coef_df[coef_df['Feature'].str.contains('_cp')]
print("\nCoefficients associated with 'cp':")
print(cp_features)
```

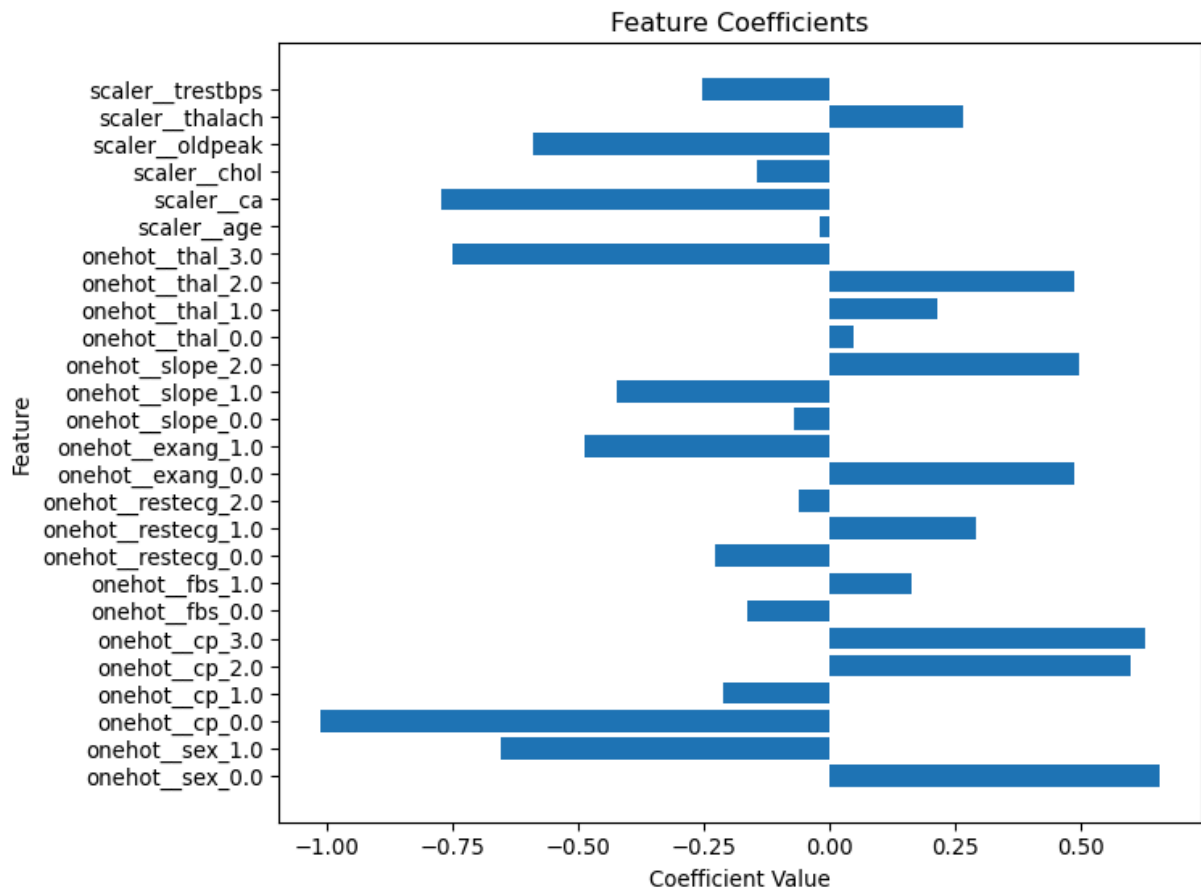
Coefficients associated with 'cp':

	Feature	Coefficient
2	onehot__cp_0.0	-1.013382
3	onehot__cp_1.0	-0.212284
4	onehot__cp_2.0	0.599934
5	onehot__cp_3.0	0.628824

Case study - coefficients

```
In [10]: # Visualize the coefficients

plt.figure(figsize=(8, 6))
plt.barh(coef_df['Feature'], coef_df['Coefficient'])
plt.title('Feature Coefficients')
plt.xlabel('Coefficient Value')
plt.ylabel('Feature')
plt.tight_layout()
plt.show()
```

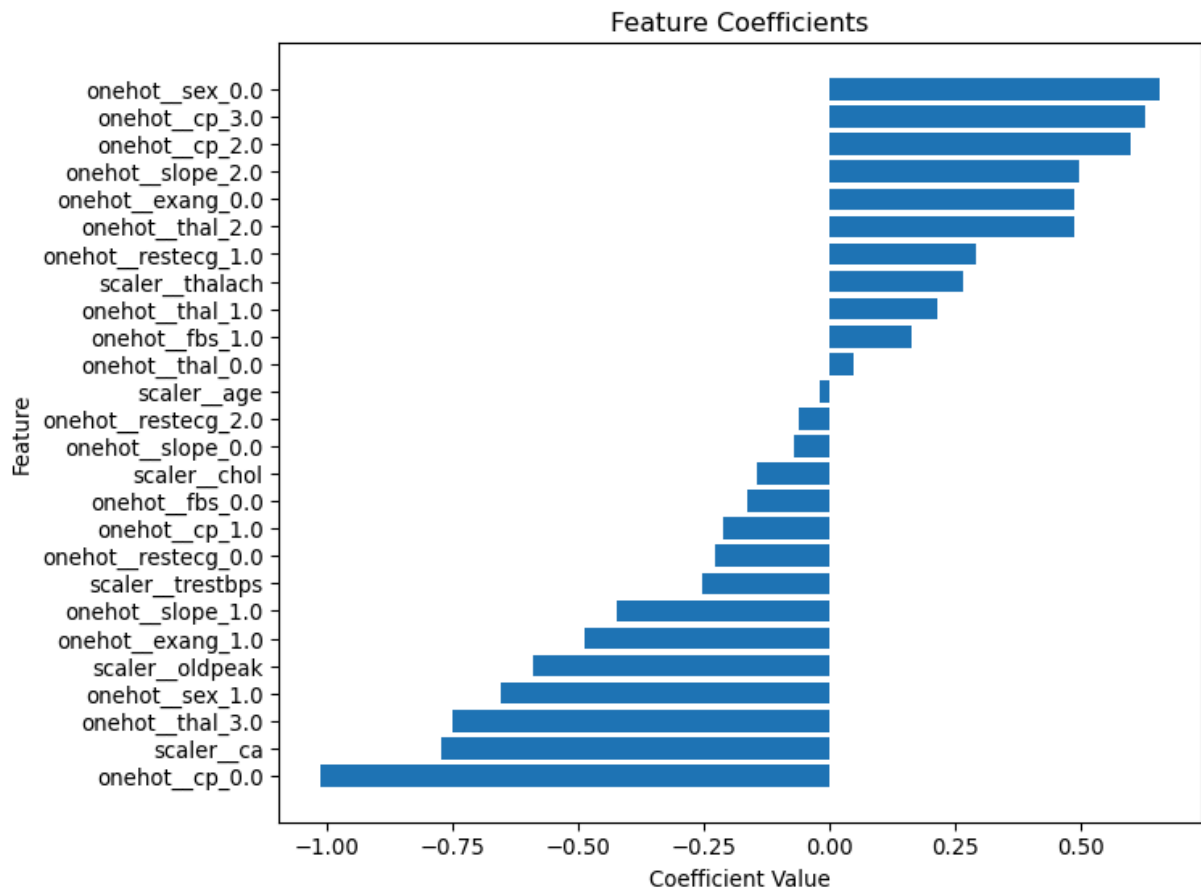



Positive coefficients in a logistic regression model signify that **higher values of the corresponding feature contribute positively** to the probability of an example belonging to 'target = 1.0'. Negative coefficients indicate the opposite effect.

Case study - coefficients (sorted)

```
In [11]: # Visualize the coefficients

plt.figure(figsize=(8, 6))
coef_df.sort_values(by='Coefficient', inplace=True)
plt.barh(coef_df['Feature'], coef_df['Coefficient'])
plt.title('Feature Coefficients')
plt.xlabel('Coefficient Value')
plt.ylabel('Feature')
plt.tight_layout()
plt.show()
```



Definition

Ordinal encoding is a technique that assigns numerical values to categorical attributes based on their inherent order or rank.

Feature Engineering - Ordinal

For attributes with values such as **'Poor'**, **'Average'**, and **'Good'**, an ordinal encoding would make sense.

...

However!

```
In [12]: from numpy import array
from sklearn.preprocessing import OrdinalEncoder

X = [['Poor'], ['Average'], ['Good'], ['Average'], ['Average']]

encoder = OrdinalEncoder()

encoder.fit(X)
encoder.transform(X)
```

```
array([[2.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

OrdinalEncoder (revised)

```
In [13]: from numpy import array
         from sklearn.preprocessing import OrdinalEncoder

         X = [['Poor'], ['Average'], ['Good'], ['Average'], ['Average']]

         encoder = OrdinalEncoder(categories=[['Poor', 'Average', 'Good']])

         encoder.fit(X)

         X_encoded = encoder.transform(X)

         X_encoded
```

```
array([[0.],
       [1.],
       [2.],
       [1.],
       [1.]])
```

The desired order of the categories must be explicitly provided to the encoder; otherwise, it defaults to alphabetical order.

An ordinal encoder is appropriate when categorical attributes have a clear, inherent order or ranking, such as 'Low', 'Medium', and 'High', or 'Poor', 'Average', and 'Good'. This encoding method preserves the ordinal relationships among categories.

When data is inherently ordinal, this encoding is more compact and can be advantageous for machine learning models. However, if there is any uncertainty about the ordinal nature of the data, it is safer to use a `OneHotEncoder`.

Definition

Discretization involves grouping ordinal values into discrete categories.

AKA **binning**, **bucketing**, or **quantization**.

Feature Engineering: Binning

Example: Categorizing ages into bins such as 'infant', 'child', 'teen', 'adult', and 'senior citizen'.

...

Advantages:

- Enables the algorithm to learn effectively with fewer training examples.

Disadvantages:

- Requires domain expertise to define meaningful categories.
- May lack generalizability; for example, the starting age for 'senior citizen' could be 60, 65, or 70[1].

Providing hints or predefined bins can help a decision tree algorithm generate more compact trees, as it reduces the need for the classifier to independently learn decision boundaries.

However, introducing such a strong bias may hinder the algorithm's ability to discover meaningful decision boundaries on its own.

Cross-validation is an effective method to determine the best encoding scheme, but it is essential to withhold the test set until the final evaluation phase of the project to prevent data leakage and ensure unbiased assessment.

FunctionTransformer

[1] Your instructor is concerned with your choice of cutoff

```
In [14]: import pandas as pd
import numpy as np
from sklearn.preprocessing import FunctionTransformer

bins = [0, 1, 13, 20, 60, np.inf]
labels = ['infant', 'kid', 'teen', 'adult', 'senior citizen']

transformer = FunctionTransformer(
    pd.cut, kw_args={'bins': bins, 'labels': labels, 'retbins': False}
)

X = np.array([0.5, 2, 15, 25, 97])
transformer.fit_transform(X)

['infant', 'kid', 'teen', 'adult', 'senior citizen']
Categories (5, object): ['infant' < 'kid' < 'teen' < 'adult' < 'senior citizen']
```

See also: [KBinsDiscretizer](#)

Scaling

What is it?

Scaling attributes ensures that their values fall within **comparable ranges**.

Why?

Why is scaling data a best practice?

Scaling attributes ensures that their values fall within comparable ranges. Numerous studies indicate that this transformation improves model performance.

Some algorithms, such as k-nearest neighbors, experience reduced efficiency when faced with significant disparities in attribute scales. Consider two attributes: one representing a proportion as a real number between 0 and 1, and the other measuring a volume in milliliters. In this case, the distance calculated between two examples will be primarily influenced by variations in volume rather than those in proportion. Tree-based algorithms are typically insensitive to the scale of the data, and therefore, scaling is not necessary.

Furthermore, optimization methods, such as gradient descent, reach convergence more quickly when the attributes have been scaled.

Normalization can be significantly influenced by the presence of outliers, which often compresses the data into a narrower range. To mitigate this effect, a common approach is to clip the data within a fixed interval, $[a, b]$. In this method, values below a are set to a , and values exceeding b are set to b . Typically, the clipping thresholds are chosen based on percentile values, such as the 5th and 95th percentiles, to ensure a balanced trimming of extreme values.

Scenario

We pretend to predict a house price using **k-Nearest Neighbors (KNN) regression** with two features:

- x_1 : number of rooms (small scale)
- x_2 : square footage (large scale)

Data (three houses)

```
In [15]: import numpy as np
import pandas as pd

# Three examples (rooms, sqft); prices only for b and c (training)
point_names = ["a", "b", "c"]
X = np.array([
```

```

    [4, 1500.0], # a (query)
    [8, 1520.0], # b (train)
    [4, 1300.0], # c (train)
], dtype=float)

prices = pd.Series([np.nan, 520_000, 390_000], index=point_names, name="price")

df = pd.DataFrame(X, columns=["rooms", "sqft"], index=point_names)
display(df)
display(prices.to_frame())

```

Note. We'll treat **b** and **c** as the training set, and **a** as the query whose price we want to predict.

Euclidean distances (unscaled)

When one feature has a much larger scale (e.g., square footage), it can dominate the sum.

```

In [16]: from sklearn.metrics import pairwise_distances

dist_unscaled = pd.DataFrame(
    pairwise_distances(df.values, metric="euclidean"),
    index=df.index, columns=df.index
)
dist_unscaled

```

```

In [17]: print("Nearest to 'a' (unscaled):", dist_unscaled.loc["a"].drop("a").idxmin())

Nearest to 'a' (unscaled): b

```

Proper scaling for modeling

For a fair ML workflow, compute scaling parameters on the training data (**b**, **c**) only, then transform both train and query:

```

In [18]: from sklearn.preprocessing import StandardScaler

train_idx = ["b", "c"]
query_idx = ["a"]

scaler = StandardScaler()

scaler.fit(df.loc[train_idx])    # fit only on training points

Z = pd.DataFrame(
    scaler.transform(df),
    columns=df.columns, index=df.index
)

```

Z

Euclidean distances (after scaling)

```
In [19]: dist_scaled = pd.DataFrame(  
    pairwise_distances(Z.values, metric="euclidean"),  
    index=Z.index, columns=Z.index  
)  
dist_scaled
```

```
In [20]: print("Nearest to 'a' (scaled):", dist_scaled.loc["a"].drop("a").idxmin())
```

Nearest to 'a' (scaled): c

KNN regressor

We'll run a 1-NN regressor (so the prediction is exactly the nearest neighbor's price) **with and without** scaling.

```
In [21]: from sklearn.neighbors import KNeighborsRegressor  
from sklearn.pipeline import Pipeline  
  
X_train = df.loc[train_idx].values           # b, c  
y_train = prices.loc[train_idx].values       # prices for b, c  
X_query = df.loc[query_idx].values          # a
```

KNN regressor (no scaling)

```
In [22]: # 1) No scaling  
  
knn_plain = KNeighborsRegressor(n_neighbors=1, metric="euclidean")  
  
knn_plain.fit(X_train, y_train)  
  
pred_plain = knn_plain.predict(X_query)[0]
```

KNN regressor (with scaling)

```
In [23]: # 2) With scaling (pipeline fits scaler only on training, then KNN on scaled  
  
knn_scaled = Pipeline([  
    ("scaler", StandardScaler()),  
    ("knn", KNeighborsRegressor(n_neighbors=1, metric="euclidean"))  
)  
  
knn_scaled.fit(X_train, y_train)
```

```
pred_scaled = knn_scaled.predict(X_query)[0]
```

I would like to highlight the use of a `Pipeline` in our approach. The resulting object functions equivalently to a classifier or regressor, possessing a `fit` method. Notably, the `predict` method within this pipeline retains a two-step process. When employed on `X_query`, it initially scales the data before applying the regressor. Importantly, the scaler is exclusively trained on the training dataset.

KNN regressor (results)

```
In [24]: pd.DataFrame(  
    {  
        "prediction (no scaling)": [pred_plain],  
        "prediction (with scaling)": [pred_scaled],  
        "nearest neighbor (no scaling)": [point_names[1] if pred_plain==price else point_names[0],  
        "nearest neighbor (with scaling)": [point_names[1] if pred_scaled==price else point_names[0],  
    },  
    index=["a"]  
)
```

Normalization

Learning algorithms perform optimally when feature values have **similar ranges**, such as $[-1,1]$ or $[0,1]$.

- This accelerates optimization (e.g., gradient descent).

Normalization:

$$\frac{x_i^{(j)} - \min^{(j)}}{\max^{(j)} - \min^{(j)}}$$

See: [sklearn.preprocessing.MinMaxScaler](#)

Standardization

Standardization (AKA z-score normalization) transforms each feature to have a **normal distribution** with a mean (μ) of 0 and a standard deviation (σ) of 1.

$$\frac{x_i^{(j)} - \mu^{(j)}}{\sigma^{(j)}}$$

Note: The range of values is not bounded!

See: [sklearn.preprocessing.StandardScaler](#)

Standardization or Normalization?

- Treat scaling as a hyperparameter and **evaluate both** normalization and standardization.
- **Standardization** is generally more robust to **outliers** than normalization.
- Guidelines from Andriy Burkov (2019), § 5:
 - Use **standardization** if features are approximately **normally distributed**.
 - Prefer **standardization** in the presence of **outliers**.
 - Otherwise, use **normalization**.

Do you see why standardization is generally more robust to **outliers** than normalization?

An effective strategy for mitigating the impact of outliers in data is the application of a logarithmic transformation to the values. This technique reduces the skewness of the data, thereby diminishing the disproportionate influence of extreme values.

Feature	Standardization	Normalization
Output Range	Not bounded	Typically [0, 1]
Center	Mean at 0	Not centered
Sensitivity to Outliers	Low	High
Primary Use Case	Default choice; for models assuming zero-centered data	When bounded input is required (e.g., image processing, some neural networks)

Case Study - Normal Distribution

```
In [25]: import numpy as np
np.random.seed(7)

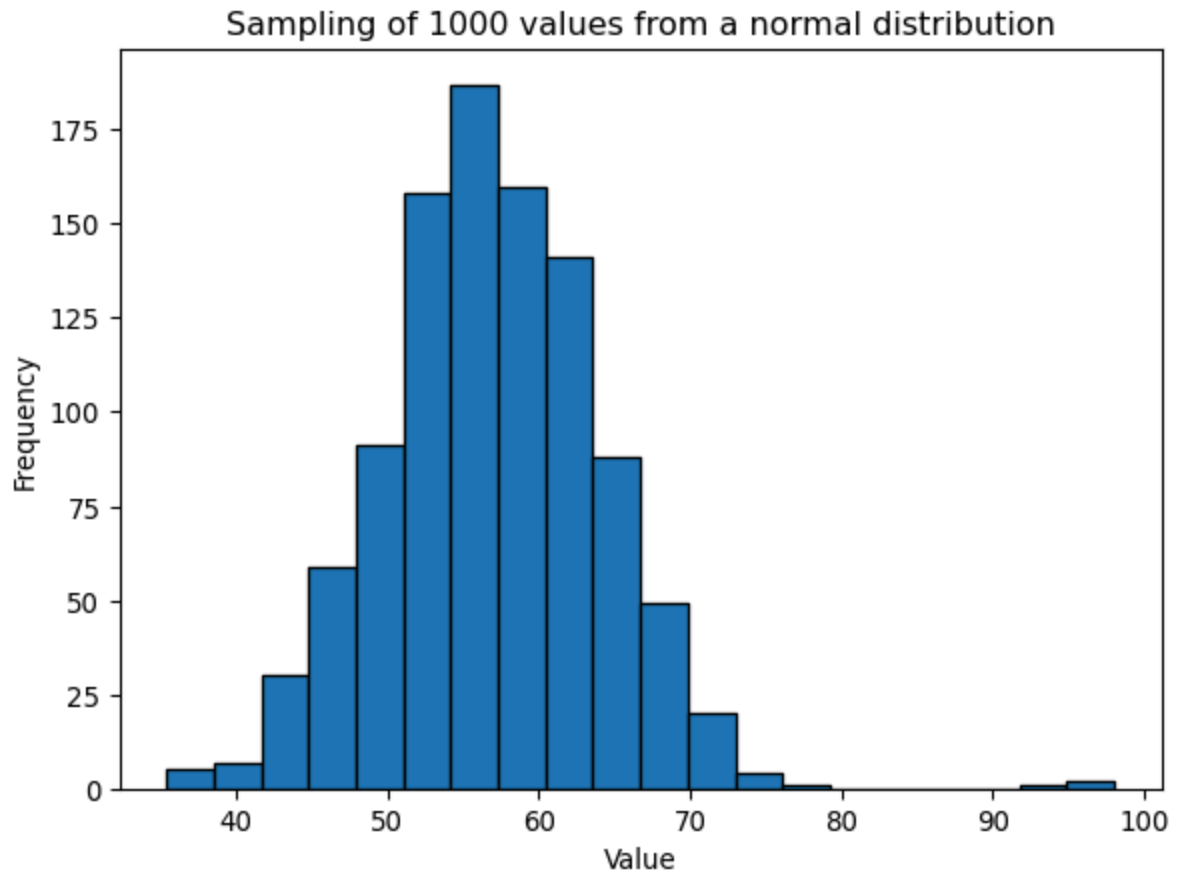
# Sample characteristics
sample_size = 1000
mu = 57
sigma = 7

# Generate values
norm_values = sigma * np.random.randn(sample_size) + mu

# Add three outliers
norm_values = np.append(norm_values, [92, 95, 98])
```

Case Study - Normal Distribution

In [26]:



Normalization

```
In [27]: from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

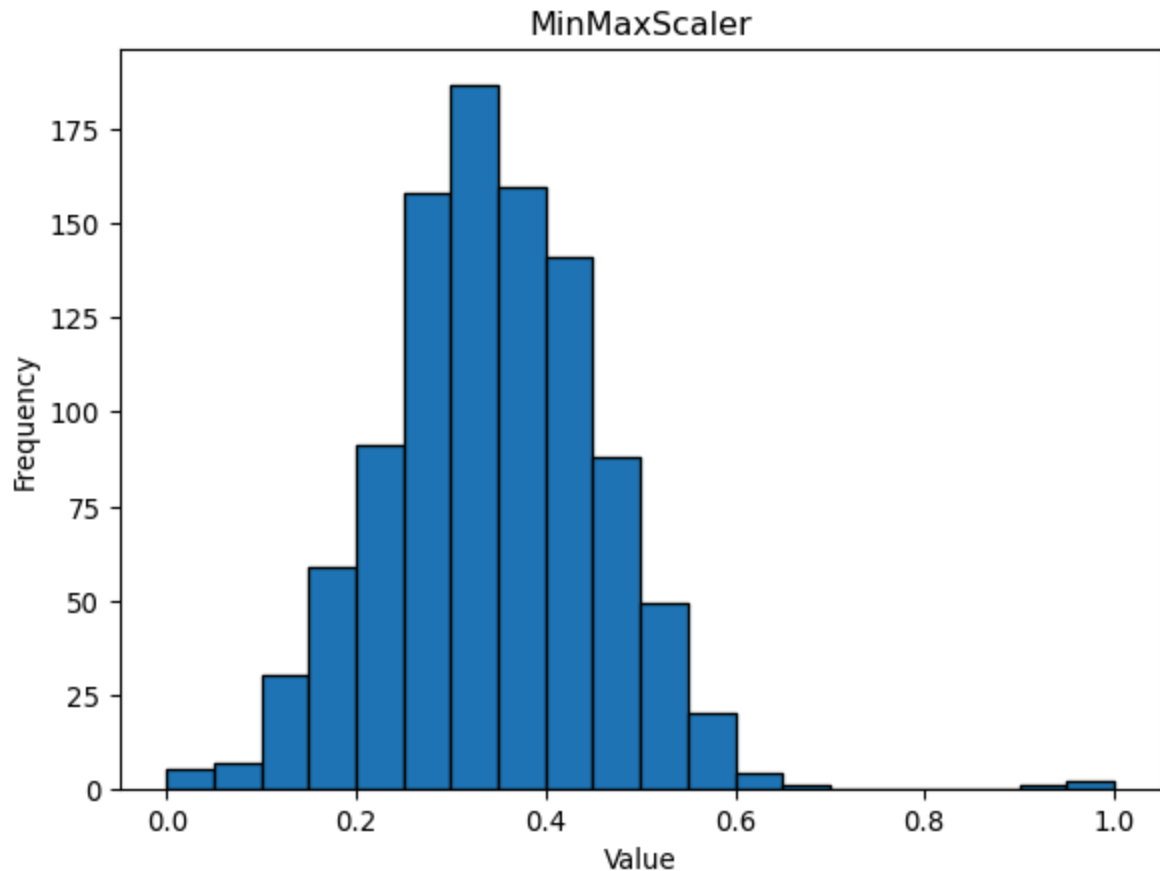
minmax_norm_values = scaler.fit_transform(norm_values.reshape(-1, 1))

# Plot the histogram
plt.hist(minmax_norm_values, bins=20, edgecolor='black')

plt.title(f'MinMaxScaler')

plt.xlabel('Value')
plt.ylabel('Frequency')

plt.show()
```



Normalization (`MinMaxScaler`) of values from a normal distribution containing outliers.

Observe that the majority of values fall within the range of 0 to 0.6, while the interval from 0.6 to 1.0 is notably sparse. Additionally, although the samples are derived from a normal distribution, the data is centered around 0.3 rather than the expected mean of 0.5.

Standardization

```
In [28]: from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

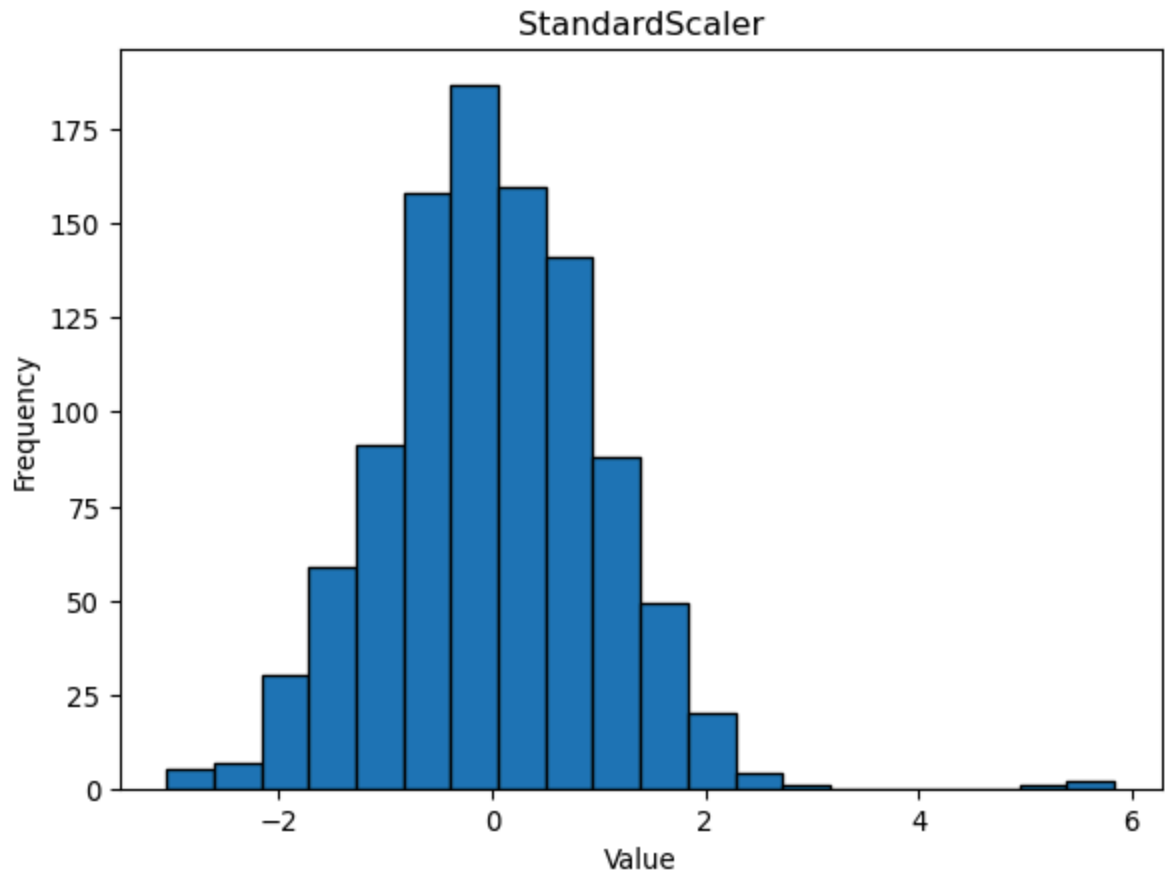
standard_norm_values = scaler.fit_transform(norm_values.reshape(-1, 1))

# Plot the histogram
plt.hist(standard_norm_values, bins=20, edgecolor='black')

plt.title(f'StandardScaler')

plt.xlabel('Value')
plt.ylabel('Frequency')

plt.show()
```

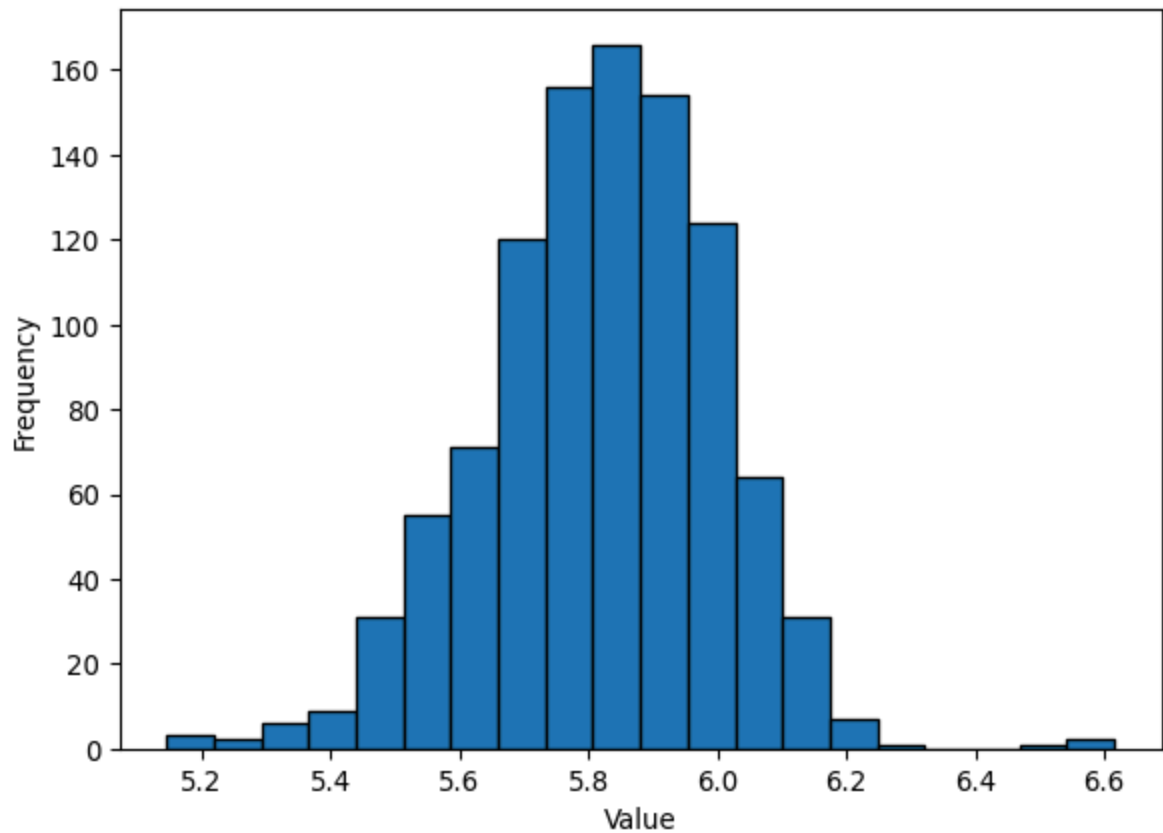


Standardization (`StandardScaler`) of values from a normal distribution containing outliers.

When examining the graph resulting from standardization, it bears a resemblance to that of normalization. Nonetheless, a closer inspection of the values reveals that the data distribution is centered around zero, with approximately 95% of the observations falling within the range of -2 to 2. It is important to note that, unlike normalization, the `StandardScaler` does not impose bounds on the range of output values.

Logarithm

In [29]:



Logarithm of values from a normal distribution containing outliers.

Logarithm & Standardization

```
In [30]: from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

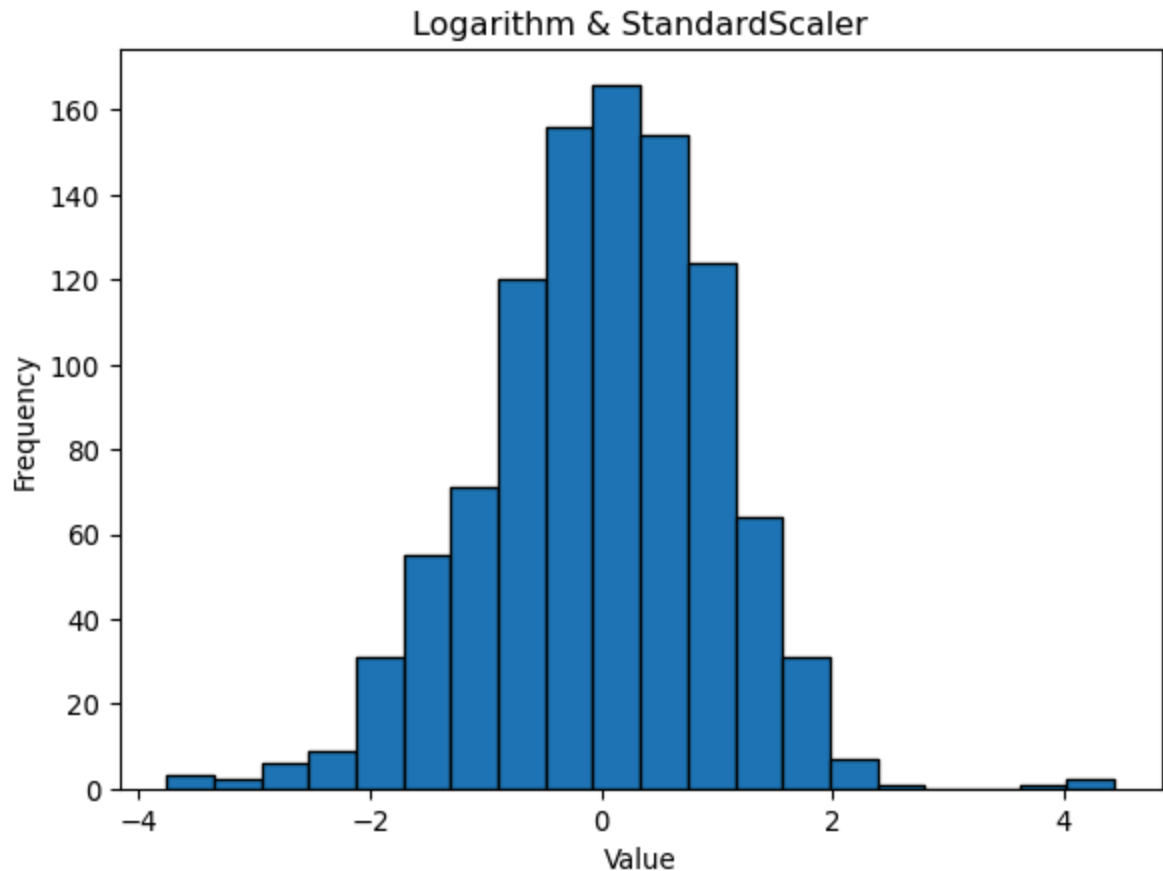
standard_log_norm_values = scaler.fit_transform(log_norm_values.reshape(-1,

# Plot the histogram
plt.hist(standard_log_norm_values, bins=20, edgecolor='black')

plt.title(f'Logarithm & StandardScaler')

plt.xlabel('Value')
plt.ylabel('Frequency')

plt.show()
```



Standardization (`StandardScaler`) of values from a normal distribution containing outliers.

Exponential Distribution

```
In [31]: # Sample size
sample_size = 1000

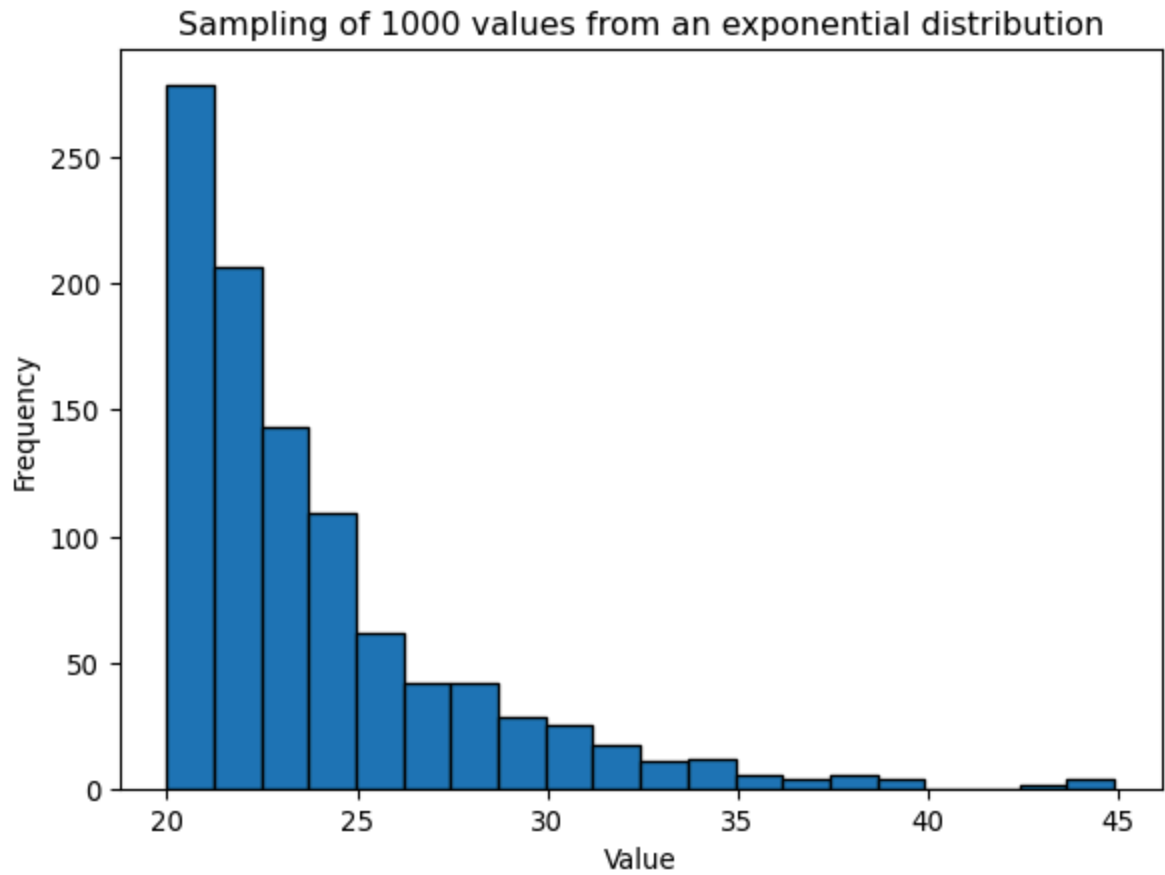
# Generate values
exp_values = np.random.exponential(scale=4, size=sample_size) + 20
```

In the NumPy expression `np.random.exponential(scale=4, size=sample_size) + 20`, the parameter `scale` refers to the inverse rate (or the mean) of the exponential distribution from which the random samples are generated. Specifically, the exponential distribution is defined by its rate parameter, and `scale` is the reciprocal of this rate, i.e., $scale = \frac{1}{\lambda}$.

Thus, `scale=4` means that the mean of the exponential distribution is 4. The argument `size=sample_size` specifies the number of random samples to generate. After generating these samples, 20 is added to each one, thus shifting the entire distribution by 20 units.

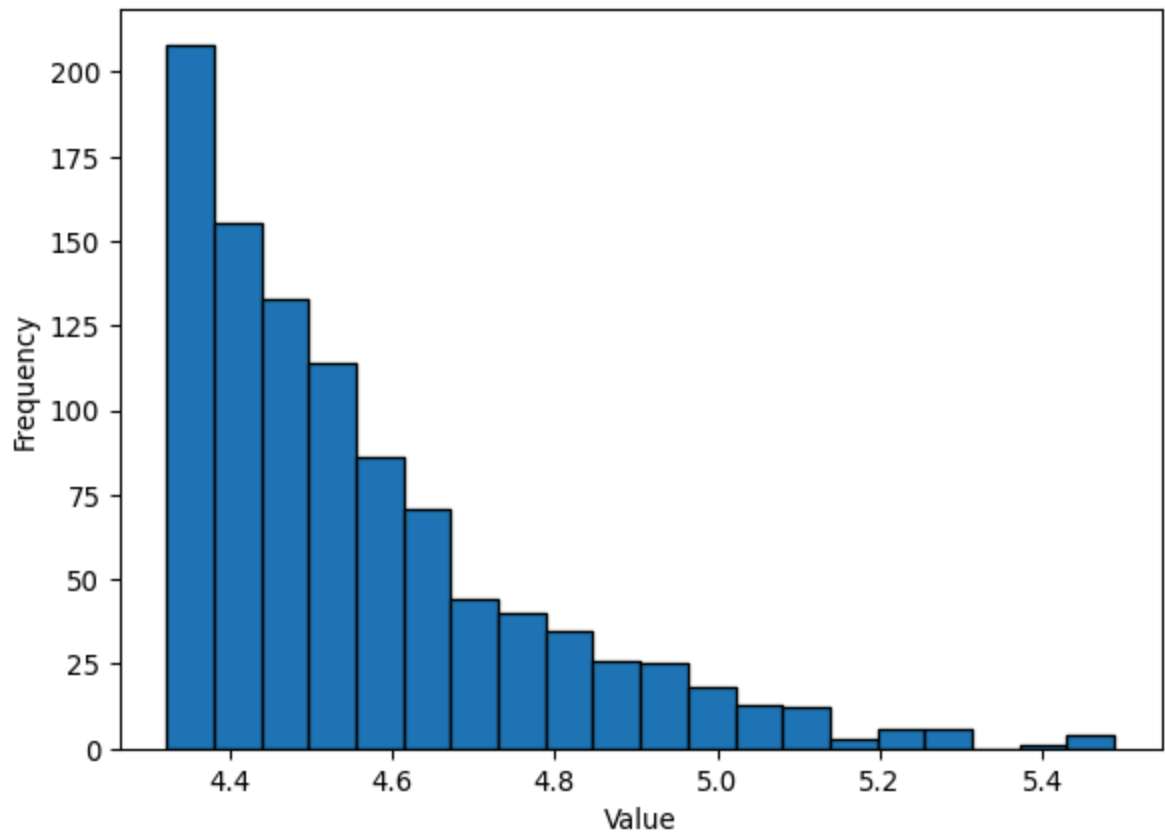
Exponential Distribution

In [32]:



Logarithm

In [33]:



Logarithm of values from an exponential distribution.

Normalization

```
In [34]: from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

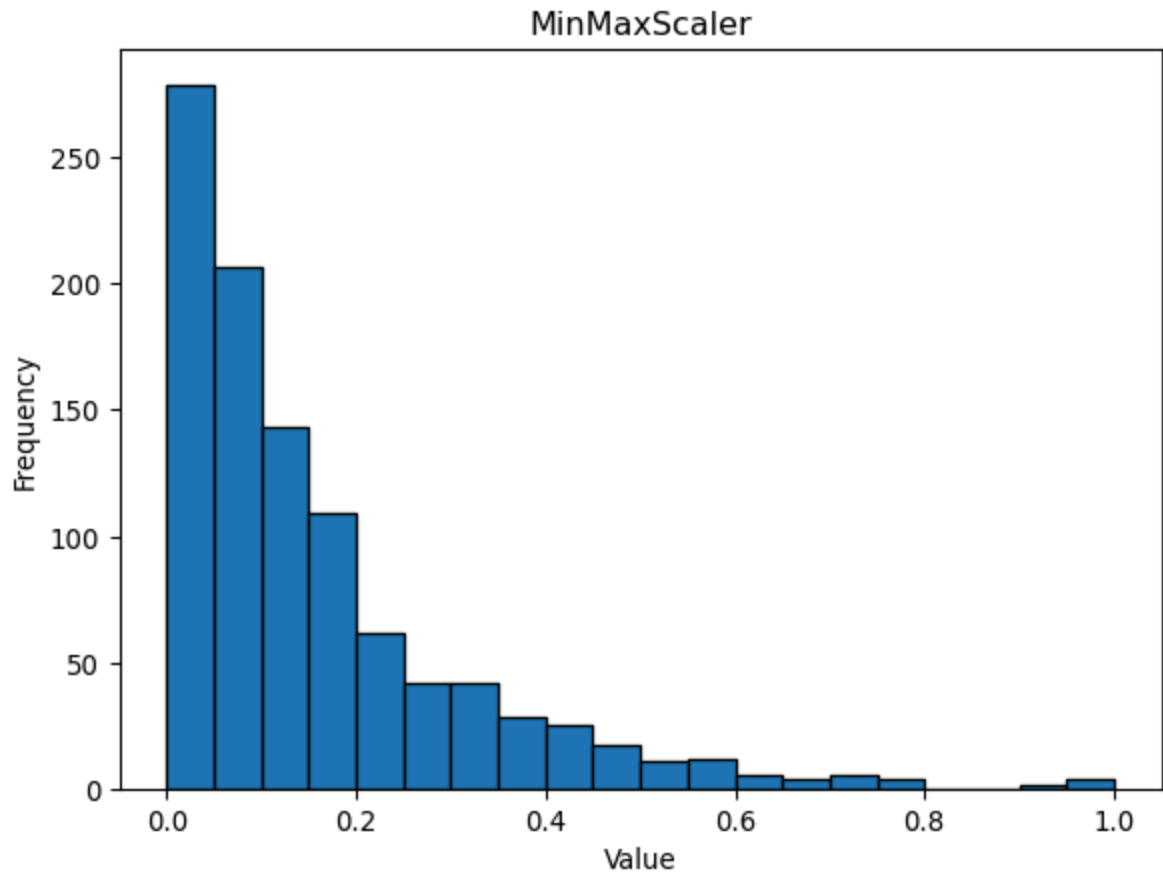
minmax_exp_values = scaler.fit_transform(exp_values.reshape(-1, 1))

# Plot the histogram
plt.hist(minmax_exp_values, bins=20, edgecolor='black')

plt.title(f'MinMaxScaler')

plt.xlabel('Value')
plt.ylabel('Frequency')

plt.show()
```

Normalization (`MinMaxScaler`) of values from an exponential distribution.

Standardization

```
In [35]: from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

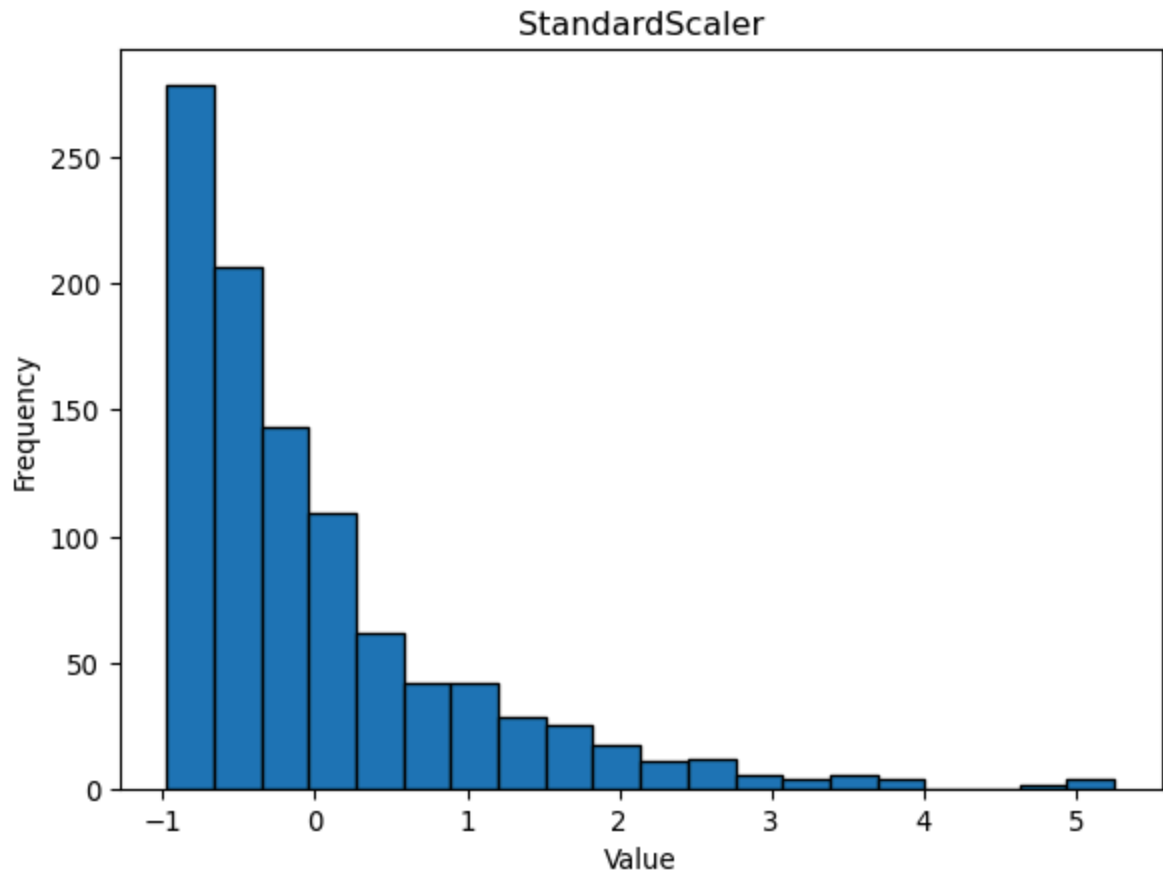
standard_exp_values = scaler.fit_transform(exp_values.reshape(-1, 1))

# Plot the histogram
plt.hist(standard_exp_values, bins=20, edgecolor='black')

plt.title(f'StandardScaler')

plt.xlabel('Value')
plt.ylabel('Frequency')

plt.show()
```



Standardization (`StandardScaler`) of values from an exponential distribution.

Logarithm & Standardization

```
In [36]: from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

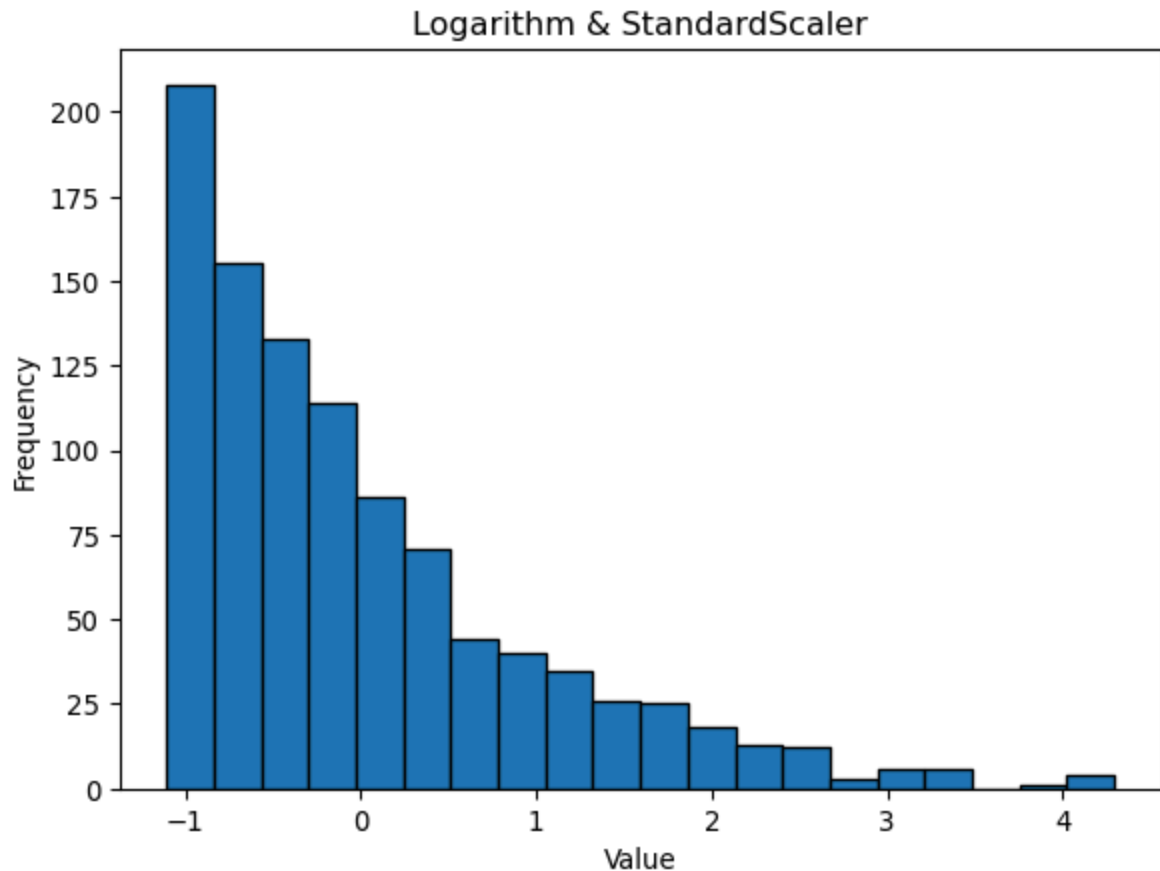
standard_log_exp_values = scaler.fit_transform(log_exp_values.reshape(-1, 1))

# Plot the histogram
plt.hist(standard_log_exp_values, bins=20, edgecolor='black')

plt.title(f'Logarithm & StandardScaler')

plt.xlabel('Value')
plt.ylabel('Frequency')

plt.show()
```



Logarithm and standardization (`StandardScaler`) of values from an exponential distribution.

Avoid Data Leakage!

The **correct workflow**:

- Split your data into a **training set** and a **testing set**.
- Fit your scaler (e.g., `StandardScaler`) on the ****training data **** (`scaler.fit(X_train)`).
 - This step learns the mean and standard deviation of the training data.
- Transform both the training data (`scaler.transform(X_train)`) and the testing data (`scaler.transform(X_test)`) using the scaler that was fitted on the training data.

You must learn the scaling parameters (like the mean, standard deviation, min, or max) from the training data only. Never use the full dataset before splitting.

Why? The test set is supposed to simulate new, unseen data. If you use information from the test set to scale your training data, your model is “cheating” by getting a sneak peek at the test data. This is called data leakage and will lead to an overly optimistic evaluation of your model’s performance.

Missing Values

Definition

Missing values refer to the absence of data points or entries in a dataset where a value is expected.

Age is a good example, as some patients may withhold their age due to privacy concerns.

Handling Missing Values

- **Drop Examples**
 - Feasible if the dataset is **large** and **outcome is unaffected**.
- **Drop Features**
 - Suitable if it **does not impact** the project's outcome.
- **Use Algorithms Handling Missing Data**
 - Example: `XGBoost`
 - **Note:** Some algorithms like `sklearn.linear_model.LinearRegression` **cannot handle** missing values.
- **Data Imputation**
 - Replace missing values with computed values.

Definition

Data imputation is the process of replacing missing values in a dataset with substituted values, typically using statistical or machine learning methods.

Data Imputation Strategy

Replace missing values with **mean** or **median** of the **attribute**.

```
In [37]: from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")

X = imputer.fit_transform(X)
```

...

- **Cons:** Ignores feature correlations and complex relationships.
- **Mode Imputation:** Replace missing values with the most frequent value; also ignores feature correlations.

Data imputation inherently relies on several assumptions, which may not always hold true.

Randomness Assumption: Many methods (e.g., mean/median imputation) assume that missingness is unrelated to any data.

Model Bias: Incorrect randomness assumptions can lead to biased estimates and flawed conclusions.

Information Loss: Imputation can obscure patterns, leading to loss of valuable information for advanced models.

Proceed with caution!

Data Imputation Strategy

Special Value Method: Replace missing values with a value outside the normal range (e.g., use -1 or 2 for data normalized between [0,1]).

- **Objective:** Enable the learning algorithm to recognize and appropriately handle missing values.

The `SimpleImputer` class includes a parameter named `add_indicator`. When this parameter is set to `True`, it generates an additional feature for each attribute with missing values. This feature indicates whether the value for a given example has been imputed. This mechanism enables classifiers to develop strategies specifically tailored to handle missing data. For example, if using the median is ineffective for a certain attribute, the classifier may learn to disregard that feature. Additionally, if missing values correlate more strongly with one class over another, the classifier can incorporate this information into its learning process.

Data Imputation Strategy

- **Middle-Range Imputation:** Replace missing values with a value in the middle of the normal range (e.g., use 0 for data distributed in the range [-1,1]).
 - **Categorical Data:** Use small non-zero numerical values.
 - Example: Use [0.25, 0.25, 0.25, 0.25] instead of [1, 0, 0, 0] for 'Poor', [0, 1, 0, 0] for 'Average', [0, 0, 1, 0] for 'Good', and [0, 0, 0, 1] for 'Excellent'.
 - **Objective:** Minimize the impact of imputed values on the results.

Selection of Method: The effectiveness of imputation methods can vary, and it is essential to compare multiple techniques to determine the best approach for your specific dataset.

Alternative Approach

- **Problem Definition:** Predict unknown (missing) labels for given examples.
- Have you encountered this kind of problem before?
- **Relevance:** This can be framed as a supervised learning problem.
 - Let \hat{x}_i be a new example: $[x_i^{(1)}, x_i^{(2)}, \dots, x_i^{(j-1)}, x_i^{(j+1)}, \dots, x_i^{(D)}]$.
 - Let $\hat{y}_i = x_i^j$.
 - **Training Set:** Use examples where x_i^j is not missing.
 - **Method:** Train a classifier on this set to predict (impute) the missing values.

Using ML for Imputation

1. Instance-Based Method:

- Use k nearest neighbors (k-NN) to find the k closest examples and impute using the non-missing values from the neighborhood.

2. Model-Based Methods:

- Employ advanced techniques such as random forests, tensor decomposition, or deep neural networks.

[Consult this guide](#) or [KNNImputer](#).

Why Use these Methods?

- **Advantages:**
 - Effectively handle complex relationships and correlations between features.
- **Disadvantages:**
 - Cost-intensive in terms of labor, CPU time, and memory resources.

Class Imbalance

Definition

The **class imbalance problem** is a scenario where the number of instances in one class significantly outnumbers the instances in other classes.

...

Models tend to be biased towards the **majority class**, leading to **poor performance on the minority class**.

Standard evaluation metrics like accuracy may be misleading in the presence of class imbalance.

Solutions

- **Resampling:** Techniques such as **oversampling** the minority class or **undersampling** the majority class.
- **Algorithmic Adjustments:** Using cost-sensitive learning or modifying decision thresholds.
 - See `class_weight` for `SGDClassifier`.
- **Synthetic Data:** Generating synthetic samples for the minority class using methods like SMOTE (Synthetic Minority Over-sampling Technique).

Apply solutions only to the **training set** to prevents **data leakage**!

Resampling and SMOTE are implemented in `imbalanced-learn`.

Chawla et al. (2002) presents the original work, whereas Pradipta et al. (2021) is a recent review.

Oversampling

- **Oversampling** can lead to overfitting, especially if the synthetic samples are very similar to the existing ones.
- **Impact:** The model may perform well on training data but generalize poorly to unseen data.

Undersampling

- **Loss of Information** reduces the number of instances in the majority class.
- **Impact:** Potentially discards valuable information and can lead to underfitting.
- **Reduced Model Performance:** Smaller training dataset may not capture the complexity of the problem.
- **Impact:** Can result in a less accurate and less robust model.

What is `class_weight`?

- `class_weight` is an option you pass to `SGDClassifier(...)` to tell the algorithm to **emphasize** certain classes more (or less) during training.
- It scales the per-example loss depending on the class label, so that "rare" or "important" classes contribute more to the objective.

How it affects training (intuitively)

1. Suppose you have classes (A) and (B). You set `class_weight = {A: w_A, B: w_B}` (or use `"balanced"` to compute weights automatically).
2. During SGD, for a training instance $((x, y))$, the loss term $(\ell(y, f(x)))$ is multiplied by the class weight (w_y) .
3. The gradient / parameter update for that training point is scaled by (w_y) .
 - So samples of class (A) with higher (w_A) "pull harder" on the model.
4. Effect: the model is biased to reduce errors more on classes with higher weights.

Automatic balancing: `class_weight = "balanced"`

- If you set `class_weight="balanced"`, `SGDClassifier` will internally compute class weights as

$$w_j = \frac{N}{K \cdot n_j}$$

where

- N = total number of samples
 - K = number of classes
 - n_j = count of samples in class (j)
- Thus rarer classes get higher weights.

Key facts & caveats

- `class_weight` is only considered if provided in the **constructor**, not (effectively) in the `fit(...)` call.
- The class weight is **multiplied** with any `sample_weight` (if you also supply that) in computing the effective weight for each example.
- If you set class weights too aggressively, training may become unstable, or the model may overfit minority classes.
- [compute_class_weight — scikit-learn 1.6.1 documentation](#)
- [SGDClassifier – class_weights & sample_weights #3928 - GitHub](#)
- [How does sample_weight work in SGDClassifier? - Stack Overflow](#)

New Features

Definition

Feature engineering is the process of **creating**, **transforming**, and **selecting** variables (attributes) from raw data to improve the performance of machine learning models.

Here, our focus is on **creating** new attributes from raw data.

Exploration

```
In [38]: import numpy as np

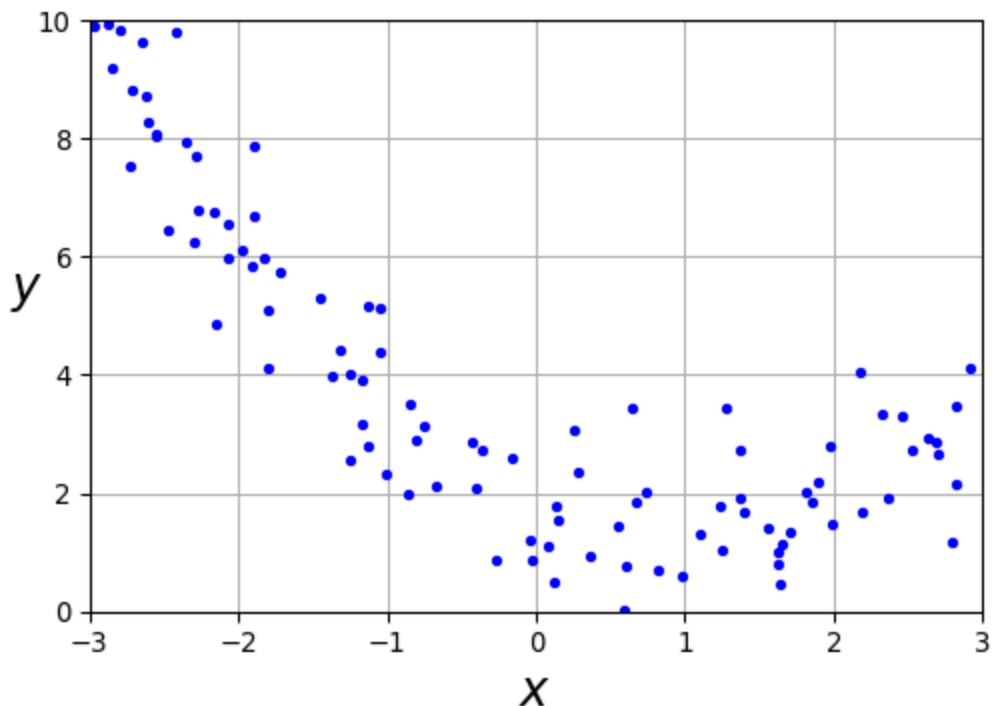
np.random.seed(42)

X = 6 * np.random.rand(100, 1) - 3
y = 0.5 * X ** 2 - X + 2 + np.random.randn(100, 1)

import matplotlib as mpl
import matplotlib.pyplot as plt

plt.figure(figsize=(6,4))

plt.plot(X, y, "b.")
plt.xlabel("$x$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
plt.grid(True)
plt.show()
```



Attribution: Géron (2022), Chapter 4

In machine learning experiments, specifying the seed of the random number generator is crucial for ensuring reproducibility. By setting a fixed seed, programmers can guarantee that the same sequence of random numbers will be generated each time the experiment is run. This consistency is vital for several reasons:

1. **Reproducibility:** It allows other programmers to replicate the experiment with the exact same conditions, facilitating verification and validation of results.
2. **Comparative Analysis:** It enables consistent comparison between different models or algorithms under the same initial conditions, ensuring that observed differences are due to the models themselves rather than variations in the random initialization.
3. **Debugging:** It aids in debugging by providing a stable environment where issues can be consistently reproduced and investigated.

Linear Regression

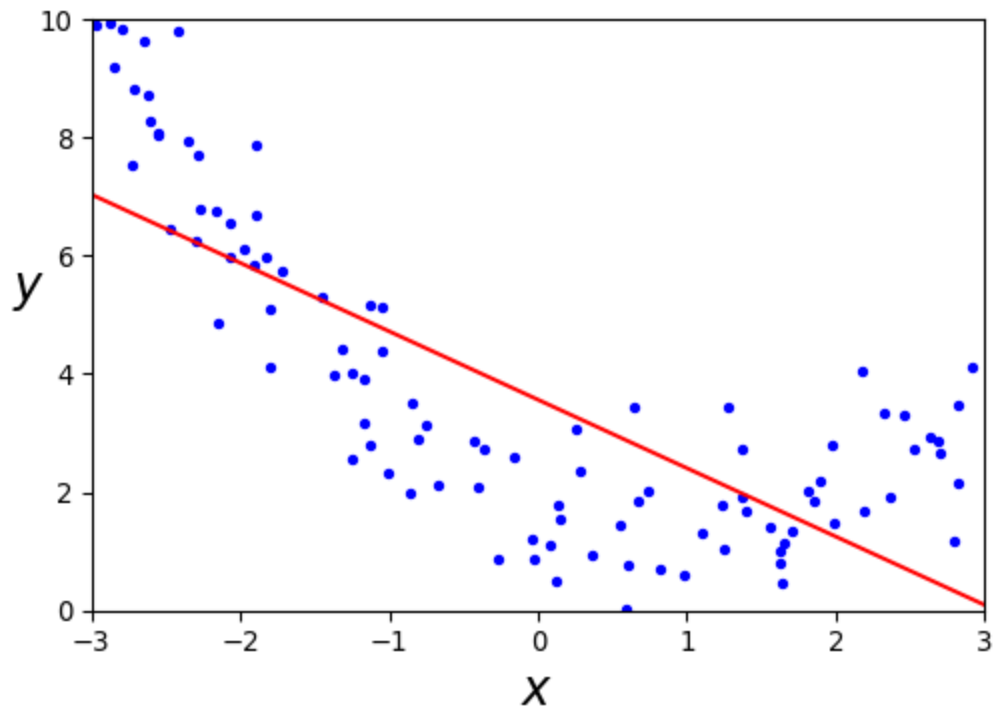
```
In [39]: from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y)

X_new = np.array([[-3], [3]])
y_pred = lin_reg.predict(X_new)

plt.figure(figsize=(6,4))

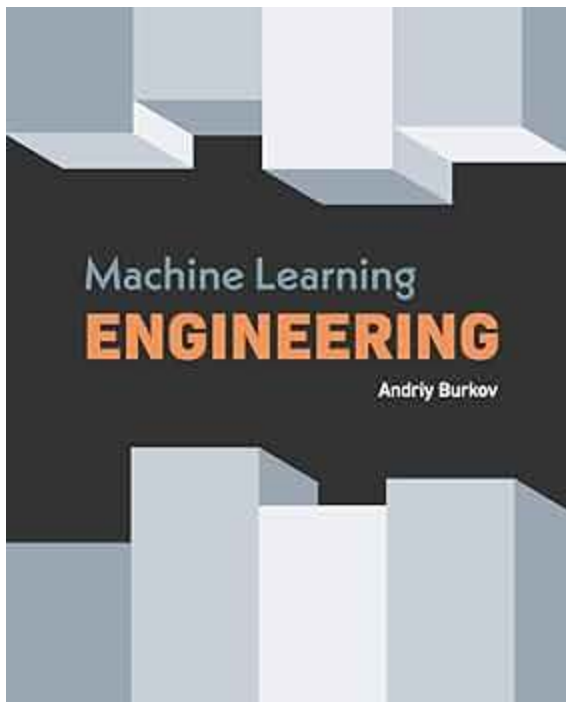
plt.plot(X, y, "b.")
plt.plot(X_new, y_pred, "r-")
plt.xlabel("$x$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
plt.show()
```



A **linear model** inadequately represents this dataset.

Adjusting the slope invariably leads to a scenario where residuals (errors) are minimized for certain data points while remaining significant for others.

Machine Learning Engineering



- **Machine Learning Engineering** by [Andriy Burkov](#) (A. Burkov 2020).

- Covers data collection, storage, preprocessing, feature engineering, model testing and debugging, deployment, retirement, and maintenance.
- From the author of [The Hundred-Page Machine Learning Book](#) (Andriy Burkov 2019).
- Available under a “read first, buy later” model.

PolynomialFeatures

```
In [40]: from sklearn.preprocessing import PolynomialFeatures

poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
```

...

```
In [41]: X[0]

array([-0.75275929])
```

...

```
In [42]: X_poly[0]

array([-0.75275929,  0.56664654])
```

...

**** sklearn.preprocessing.PolynomialFeatures ****

Generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, if an input sample is two dimensional and of the form $[a, b]$, the degree-2 polynomial features are $[1, a, b, a^2, ab, b^2]$.

PolynomialFeatures

Given two features a and b , `PolynomialFeatures` with `degree=3` would add a^2 , a^3 , b^2 , b^3 , as well as, ab , a^2b , ab^2 !

...

Warning

`PolynomialFeatures(degree=d)` adds $\frac{(D+d)!}{d!D!}$ features, where D is the original number of features.

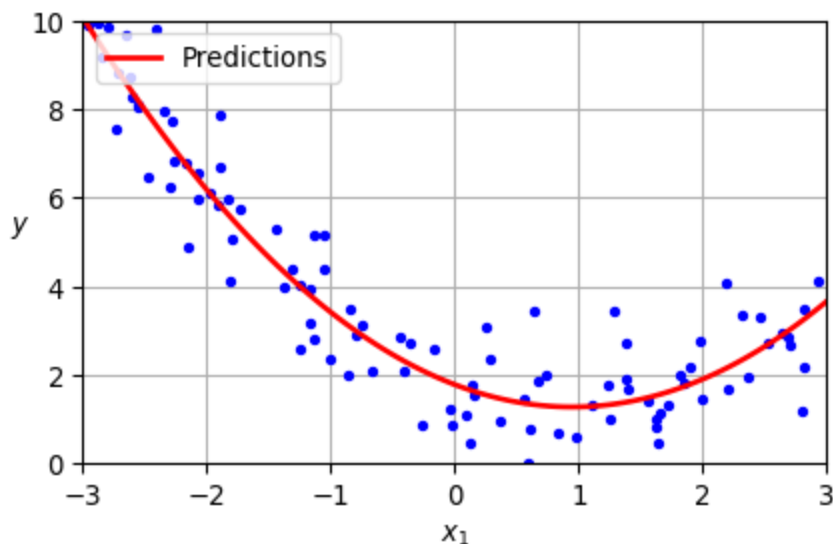
Additionally, you have the option to engineer new features of your own.

Polynomial Regression

```
In [43]: lin_reg = LinearRegression()
lin_reg = lin_reg.fit(X_poly, y)

X_new = np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)

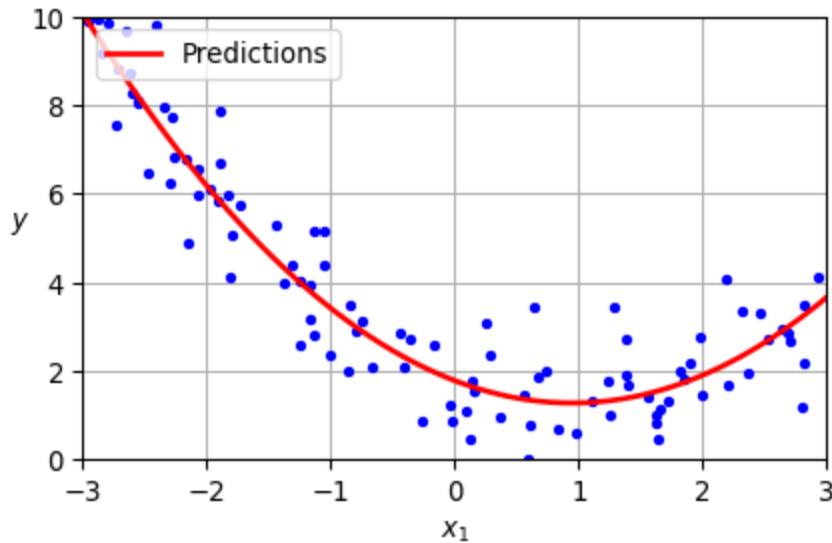
plt.figure(figsize=(5, 3))
plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("$x_1$")
plt.ylabel("$y$", rotation=0)
plt.legend(loc="upper left")
plt.axis([-3, 3, 0, 10])
plt.grid()
plt.show()
```



LinearRegression on PolynomialFeatures

```
In [44]: lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
```

Polynomial Regression



The data was generated according to the following equation, with the inclusion of Gaussian noise.

$$y = 0.5x^2 - 1.0x + 2.0$$

Presented below is the learned model.

$$\hat{y} = 0.56x^2 + (-1.06)x + 1.78$$

```
In [46]: lin_reg.coef_, lin_reg.intercept_
(array([-1.06633107,  0.56456263]), array([1.78134581]))
```

1. Linearity is about the coefficients, not the raw inputs

- Logistic regression is linear in its **parameters** (weights).
- The decision boundary comes from:

$$\hat{y}_i = \sigma(\theta_0 + \theta_1 z_i^{(1)} + \theta_2 z_i^{(2)} + \dots + \theta_{D'} z_i^{(D')})$$

where each $z_i^{(j)}$ is a feature.

- If you define $z_i^{(j)}$ to be a polynomial function of the original variables (e.g. $z_i^{(3)} = (x_i^{(1)})^2$), the model is still *linear in the $z_i^{(j)}$* .

2. PolynomialFeatures only changes the feature space

- `PolynomialFeatures` maps input $(x_i^{(1)}, x_i^{(2)})$ to a new vector: $(x_i^{(1)}, x_i^{(2)}, (x_i^{(1)})^2, x_i^{(1)}x_i^{(2)}, (x_i^{(2)})^2, \dots)$.
- Logistic regression then finds a linear decision boundary in this **expanded feature space**.

- When you map it back to the original $(x_i^{(1)}, x_i^{(2)})$ space, that “linear” decision surface can appear curved or wiggly, but mathematically it’s still a hyperplane in the transformed space.

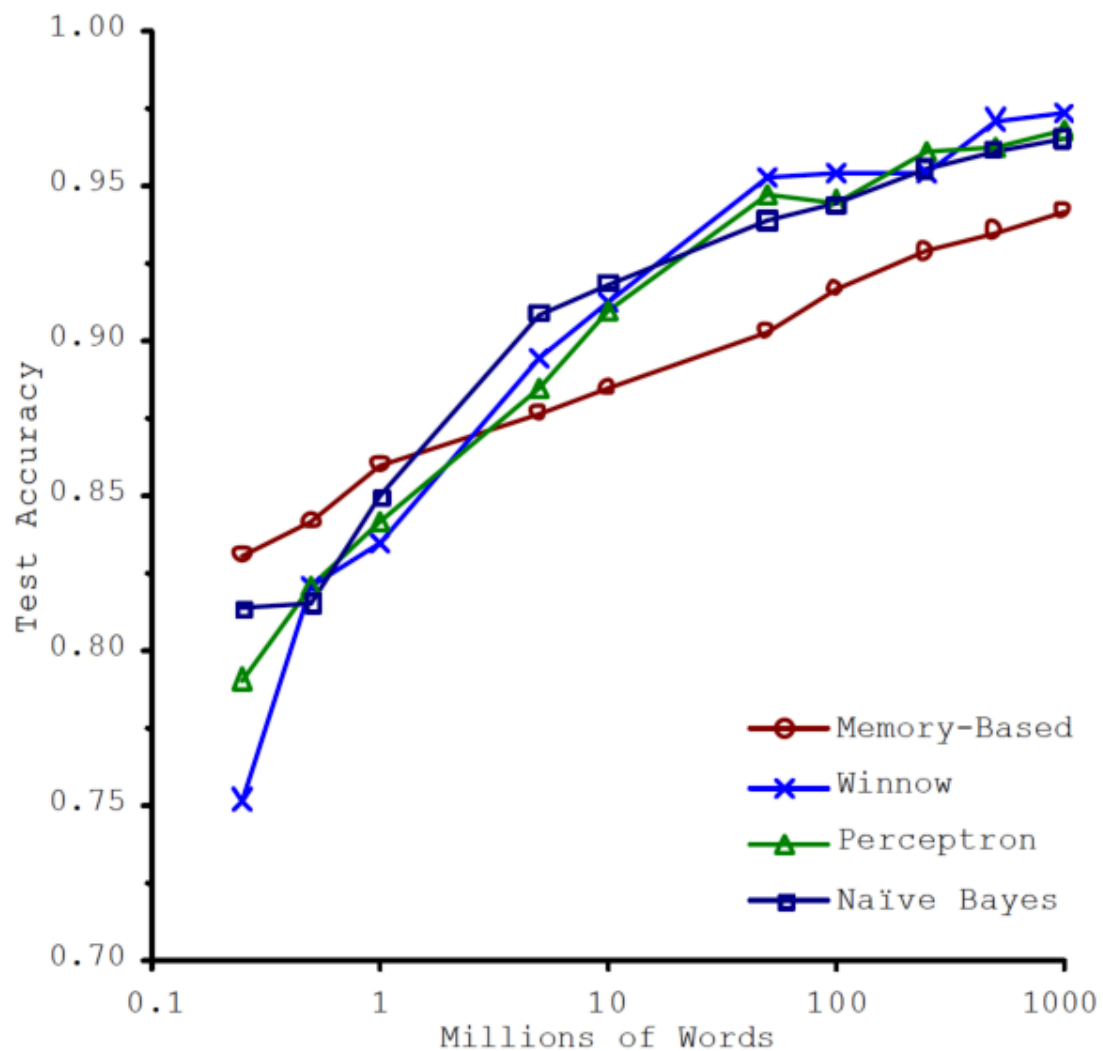
Key message:

“Logistic regression always finds a linear boundary in the features you give it. If you feed it polynomials, the linear boundary in that space looks nonlinear in the original space.”

Data

Size does matter

“However, these results suggest that we may want to reconsider the trade-off between spending time and money on algorithm development versus **spending it on corpus development** algorithms themselves.”



Attribution: Banko and Brill (2001)

Unreasonable Effectiveness of Data

<https://youtu.be/yvDCzhbjYWs>

Halevy, Norvig, and Pereira (2009) and Kaplan et al. (2020).

Peter Norvig's presentation, titled "The Unreasonable Effectiveness of Data," runs for just over one hour. It is noteworthy that the paper on which the presentation is based was published in 2009, predating the success of AlexNet.

The substantial improvements observed with AlexNet in 2012 highlighted the benefits of training deep neural networks on large image datasets.

Similarly, modern models like GPT, Gemini, Claude, and LLaMA have achieved significant advancements in language capabilities by training on vast amounts of text data, encompassing nearly all written material since the inception of human civilization.

Neural scaling laws describe how the performance of neural networks varies with changes in key factors such as dataset size, number of parameters, and computational cost Kaplan et al. (2020).

Definition

Data augmentation is a technique used to increase the diversity of a dataset by applying various transformations to the existing data.

Purpose: Enhance the robustness and generalization capability of machine learning models.

Data Augmentation

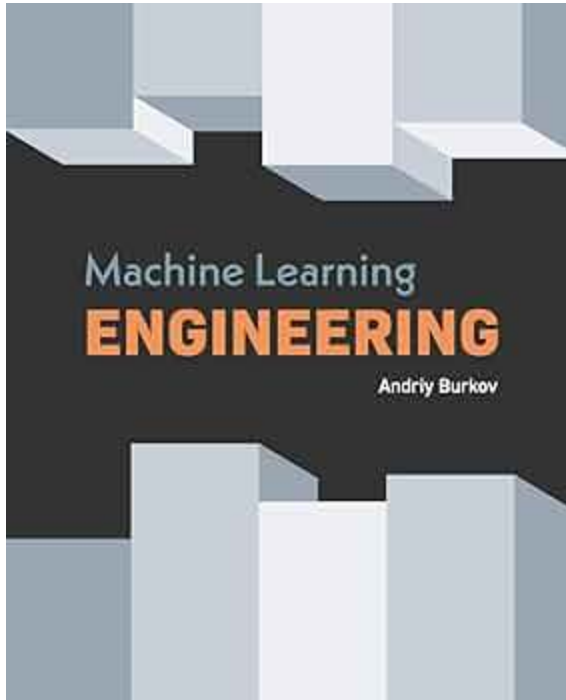
- **For Images:** Rotations, translations, scaling, flipping, adding noise, etc.
 - [How to find ancient geoglyphs using machine learning?](#), Sakai et al. (2024)
- **For Text:** Synonym replacement, random insertion, deletion, and swapping of words.

Generative Adversarial Networks (GANs) (a form of deep learning) can be used to generate new, synthetic data that mimics the distribution of the original dataset.

See also: Shumailov et al. (2024).

Prologue

Further readings



- **Machine Learning Engineering** by [Andriy Burkov](#) (A. Burkov 2020).
- Covers data collection, storage, preprocessing, feature engineering, model testing and debugging, deployment, retirement, and maintenance.
- From the author of **The Hundred-Page Machine Learning Book** (Andriy Burkov 2019).
- Available under a “read first, buy later” model.

Summary

- **Attribute Encoding:** Evaluation of techniques to avoid bias and possibly speed up the training.
- **Preprocessing:**
 - Data Scaling
 - Handling Missing Values
 - Managing Class Imbalance
- **Training Set Size:** Impact on model efficacy and generalization.

Next lecture

- We will introduce artificial neural networks.

References

Banko, Michele, and Eric Brill. 2001. "Scaling to Very Very Large Corpora for Natural Language Disambiguation." In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, 26–33. ACL '01. USA: Association for Computational Linguistics. <https://doi.org/10.3115/1073012.1073017>.

Burkov, A. 2020. *Machine Learning Engineering*. True Positive Incorporated. <https://books.google.ca/books?id=HeXizQEACAAJ>.

Burkov, Andriy. 2019. *The Hundred-Page Machine Learning Book*. Andriy Burkov.

Chawla, N V, K W Bowyer, L O Hall, and W P Kegelmeyer. 2002. "SMOTE: Synthetic minority over-sampling technique." *Journal of Artificial Intelligence Research* 16: 321–57.

Géron, Aurélien. 2022. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. 3rd ed. O'Reilly Media, Inc.

Halevy, Alon, Peter Norvig, and Fernando Pereira. 2009. "The Unreasonable Effectiveness of Data." *IEEE Intelligent Systems* 24 (2): 8–12.

Kaplan, Jared, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. "Scaling Laws for Neural Language Models." <https://arxiv.org/abs/2001.08361>.

Pradipta, Gede Angga, Retantyo Wardoyo, Aina Musdholifah, I Nyoman Hariyasa Sanjaya, and Muhammad Ismail. 2021. "SMOTE for Handling Imbalanced Data Problem : A Review." *2021 Sixth International Conference on Informatics and Computing (ICIC)* 00: 1–8. <https://doi.org/10.1109/icip54025.2021.9632912>.

Russell, Stuart, and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson. <http://aima.cs.berkeley.edu/>.

Sakai, Masato, Akihisa Sakurai, Siyuan Lu, Jorge Olano, Conrad M. Albrecht, Hendrik F. Hamann, and Marcus Freitag. 2024. "AI-accelerated Nazca survey nearly doubles the number of known figurative geoglyphs and sheds light on their purpose." *Proceedings of the National Academy of Sciences* 121 (40): e2407652121. <https://doi.org/10.1073/pnas.2407652121>.

Shumailov, Ilia, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarín Gal. 2024. "AI models collapse when trained on recursively generated data." *Nature* 631 (8022): 755–59. <https://doi.org/10.1038/s41586-024-07566-y>.

Appendix: Pipeline

Pipeline

```
In [47]: from sklearn.pipeline import Pipeline
        from sklearn.preprocessing import StandardScaler

        num_pipeline = Pipeline([
            ('imputer', SimpleImputer(strategy="median")),
            ('attrs_adder', CombinedAttributesAdder()),
            ('std_scaler', StandardScaler()),
        ])

        training_num_tr = num_pipeline.fit_transform(training_num)

        full_pipeline = ColumnTransformer([
            ("num", num_pipeline, num_attrs),
            ("cat", OneHotEncoder(), ["sequence"]),
        ])

        training_prepared = full_pipeline.fit_transform(training)
```

Marcel **Turcotte**

Marcel.Turcotte@uOttawa.ca

School of Electrical Engineering and **Computer Science** (EECS)

University of Ottawa



Attribution: Generated by DALL-E, via ChatGPT (GPT-4), OpenAI, September 30, 2024.

See next page for interactions.

GPT-4o interactions 2024-09-30

I presented GPT-4o with an image of the main character from [Marcel the Shell With Shoes On](#) and had the following interactions.

Marcel: How would you describe the subject of the image, its artistic quality, technique, and rendering style?

GPT-4o: The image depicts a whimsical character that resembles a small shell with one large, expressive eye. It stands on two legs wearing shoes, giving it a cute and almost anthropomorphic appearance. The character has a surreal and quirky look, which lends it a sense of personality and charm.

Artistic Quality:

The image is rendered with a high degree of polish and smoothness, possibly created digitally. The textures, shading, and lighting are soft and subtle, giving the character a smooth and rounded look. The rendering captures a delicate balance between realism (in the textures of the shell and shoes) and cartoon-like abstraction (in the exaggerated eye and small stature).

Technique:

The artist appears to have employed a blend of digital painting and 3D-like modeling techniques, with careful attention to detail in the reflections in the eye and the textures of the shell and shoes. The color palette is muted, using soft tones that evoke a gentle, friendly atmosphere.

Rendering Style:

The rendering style is a mix of minimalist and hyperrealist elements. The minimalism is seen in the simplified forms of the character—particularly the body and legs—while the hyperrealism comes through in the detailing of the eye and the subtle textures. This gives the character a tactile, almost tangible quality while maintaining its fantastical, imaginative essence.

The overall effect of the design is both charming and surreal, combining everyday objects (a shell, shoes) with anthropomorphic and fantastical elements to create a unique, playful character.

Marcel: Create an image based on the following description: The image depicts a whimsical character that resembles a small shell with one large, expressive eye. It stands on two legs wearing shoes, giving it a cute and almost anthropomorphic appearance. The character has a surreal and quirky look, which lends it a sense of personality and charm.

Artistic Quality:

The image is rendered with a high degree of polish and smoothness, possibly created digitally. The textures, shading, and lighting are soft and subtle, giving the character a smooth and rounded look. The rendering captures a delicate balance between realism (in the textures of the shell and shoes) and cartoon-like abstraction (in the exaggerated eye and small stature).

Technique:

The artist appears to have employed a blend of digital painting and 3D-like modeling techniques, with careful attention to detail in the reflections in the eye and the textures

of the shell and shoes. The color palette is muted, using soft tones that evoke a gentle, friendly atmosphere.

Rendering Style:

The rendering style is a mix of minimalist and hyperrealist elements. The minimalism is seen in the simplified forms of the character—particularly the body and legs—while the hyperrealism comes through in the detailing of the eye and the subtle textures. This gives the character a tactile, almost tangible quality while maintaining its fantastical, imaginative essence.

The overall effect of the design is both charming and surreal, combining everyday objects (a shell, shoes) with anthropomorphic and fantastical elements to create a unique, playful character.

GTP-4o: Here is the image based on the description you provided. Let me know if you'd like to make any adjustments!