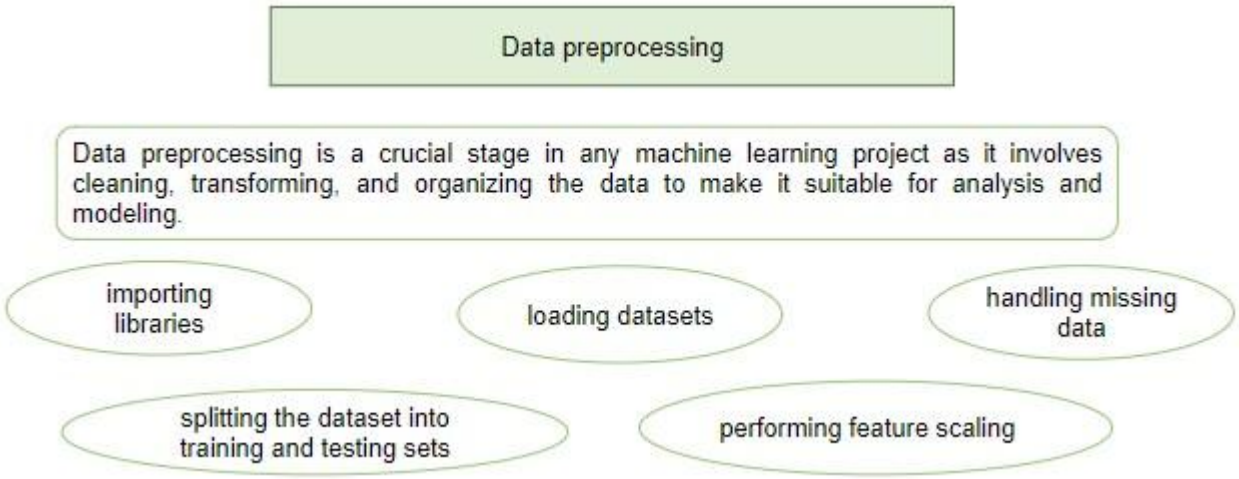
Data Preprocessing

The essential steps to prepare data for machine learning models

Data preprocessing is a crucial stage in any machine learning project as it involves cleaning, transforming, and organising the data to make it suitable for analysis and modeling. This chapter covers importing libraries, loading datasets, handling missing data, splitting the dataset into training and testing sets, and performing feature scaling.



A solid understanding of how to preprocess your data using Python effectively will equip you with the necessary skills to build a strong foundation for building machine learning models. All our coding exercises are conducted using Google Colab, a powerful collaborative coding and experimentation platform.

To begin with, ensure you have a Google account, as Google Colab requires it for accessing and saving your work. Once logged in, navigate to the Google Colab homepage or search for "Google Colab" in your browser.

Upon reaching the Google Colab interface, you'll be greeted with an option to create a new notebook. Click on this to open a new Python notebook where we'll conduct our coding exercises.

Once the notebook is opened, you'll notice a familiar environment resembling the Python programming environment. Here, you can write and execute Python code cells seamlessly.

We'll start by importing the necessary libraries for our data preprocessing tasks.

[GC] (refer to the format as mentioned in the .ipynb file)

```
#Importing libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

We've imported three essential libraries: NumPy, Matplotlib, and Pandas. NumPy helps with numerical operations and managing arrays, which is crucial for mathematical computations in machine learning. Matplotlib facilitates data visualisation, enabling us to create various plots to understand patterns and relationships in our datasets. Pandas simplifies data manipulation and analysis by providing robust data structures like DataFrame, making tasks such as cleaning and organising data easier during the model-building process.

[Add about run this line of code importing libraries]

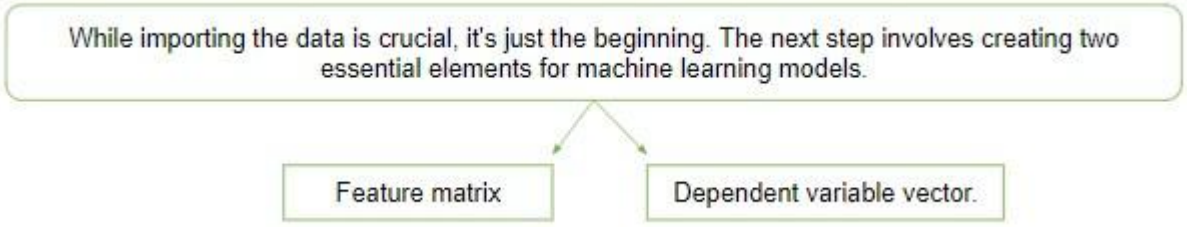
Next, proceed to import our data. You can download the dataset from the Github link provided.
Before we import the data, we'll need to upload the data to our Google Colab notebook before we import it. Here's how to do that:

1. **Click on the Files tab in the left sidebar of your Colab notebook.**
2. **Select Upload. This will open a file selection dialogue.**
3. **Navigate to the location where you downloaded the CSV file from the Github link.**
4. **Select the CSV file and click Open.**

The file will be uploaded to your Colab notebook and will be ready to use for further analysis.
[GC]

```
# Importing the dataset
df = pd.read_csv('Data.csv')
```

This line now reads a CSV (Comma Separated Values) file named 'Data.csv' and stores its contents in a pandas DataFrame, which is essentially a table-like data structure. The DataFrame is assigned to the variable df.
While importing the data is crucial, it's just the beginning. The next step involves creating two essential elements for machine learning models: a Feature matrix and a Dependent variable vector.



The Feature Matrix (X) is the matrix that holds the independent variables, also known as features or predictors. You'll use these characteristics to build a model that predicts the dependent variable. Imagine it as a collection of all the columns in your DataFrame except the one you're trying to predict.
So, according to our dataset, the matrix of features will contain the variables 'Country', 'Age' and 'Salary'. The code to declare the matrix of features will be as follows:
[GC]

```
X = df.iloc[:, :-1].values
```

Here, df.iloc[:, :-1] part of the code uses DataFrame indexing with iloc, which allows for integer-based indexing. The: before the comma indicates that we want to select all rows of the DataFrame, and :-1 after the comma suggests that we want to select all columns except the last one. If the dependent variable matrix is not in the previous column of the DataFrame, you need to specify the column's index containing the dependent variable.
.values: This part converts the selected DataFrame subset into a NumPy array. Machine learning algorithms in Python often work with NumPy arrays rather than DataFrames.

The Dependent Variable Vector (y) on the other hand stores the dependent variable, also called the **target variable or response variable.**

The Dependent Variable Vector (y), on the other hand, stores the dependent variable, also called the target variable or response variable. This is the value we're trying to predict using the features. It typically corresponds to a single column in our DataFrame.
So, the dependent variable vector (y) would contain only the "Purchased" column.
The code to declare the dependent variable vector will be as follows
[GC]
y = df.iloc[:, -1].values
This part of the code is again using DataFrame indexing with iloc. Here, : before the comma indicates that we want to select all rows of the DataFrame, and -1 after the comma indicates that we want to select only the last column.
.values: Similar to before, this part converts the selected DataFrame subset into a NumPy array.
Now, with the feature matrix (X) and dependent variable vector (y) prepared, let's visualise their contents using the print function.
[GC]
print(X)
print(y)
[Output]
Creating X and y is a vital step in preparing your data for machine learning. It allows the algorithms to learn from the features and make informed predictions about the dependent variable.
Now, let's proceed to the next step, where we handle missing values in our dataset.
The next step is handling the missing values in the dataset.
[Double-click on the data in Google Collab to show that our data contains missing values]
Missing values, where data points are absent for some features, are a common challenge in real-world datasets. In our dataset, a customer from Germany who is 40 years old and has purchased a product has a missing salary value. This type of missing information can be problematic.
So why are missing values a concern?
Missing values can potentially lead to errors during the training process of our machine-learning model. Due to these gaps, the algorithm might misinterpret the data patterns or make inaccurate predictions. Therefore, it's crucial to address missing values effectively.
Various techniques exist for handling them, each with advantages and disadvantages. The first common approach is Deletion. This technique removes entire data points (rows) from the dataset if they contain missing values. Deletion can be a reasonable option for limited missing values in large datasets, but it's not a one-size-fits-all solution.Other techniques for handling missing values.

Another common approach for handling missing values is imputation. This technique involves replacing the missing values with estimates. Here's an example of imputation:
Mean/Median imputation: This method replaces missing values with the column's average (mean) or median value containing the missing data.
Let's see how this technique could be applied to the missing salary value in our example using the scikit-learn library.
Scikit-learn provides the SimpleImputer class for various imputation strategies.
[GC]

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
imputer.fit(X[:, 1:3])
X[:, 1:3] = imputer.transform(X[:, 1:3])
print(X)
```

Here first import the SimpleImputer class from the scikit-learn library, which is used for handling missing values through various strategies.
Next, in the imputer = SimpleImputer(...), creating an instance of the SimpleImputer class and assigning it to the variable imputer. This object will impute (fill in) missing values in our data.
missing_values=np.nan argument specifies how the SimpleImputer should identify missing values. In this case, np.nan (which stands for "Not a Number") is being used as the indicator for missing values. This is a common representation of disappeared values in NumPy arrays, which are often used in machine learning.
strategy='mean' argument defines the strategy the SimpleImputer will use to fill in missing values. Here, we're choosing the 'mean' approach, which means the imputer will replace missing values with the average (mean) of the corresponding feature (column) in the data.
imputer.fit(X[:, 1:3]) line fits the imputer instance to the data. It calculates the mean for each feature (column) specified in X[:, 1:3], which represents columns 1 and 2 as Python indexing starts from 0.
Next X[:, 1:3] = imputer.transform(X[:, 1:3]) transforms the missing values in columns 1 and 2 of X by replacing them with the earlier mean value.
And finally print(X) prints the updated matrix of features X to see the changes made by imputation.
The print statement shows the imputation is made, and the data is as follows:
[Output]
It's also clear that the NaN values in the Age and Salary column have been replaced by the column's mean value.
Now that we've successfully addressed missing values in our dataset using mean imputation. This ensures our machine learning model has a complete picture for each data point. However, another important step before we proceed is encoding categorical data.
Now the question arises: Why is encoding categorical data necessary?
Many machine learning algorithms primarily work with numerical data. However, real-world datasets often contain categorical features, such as "country," "product type," or "hair colour." These features represent distinct categories but don't have an inherent numerical ordering.

For instance, a value of "Germany" in a "country" column doesn't inherently mean it's "greater than" or "less than" a value of "France." If we directly feed these categorical features into our model, it might misinterpret their relationships.
Encoding solves this issue by transforming categorical data into a numerical representation that the machine-learning model can understand. Various encoding techniques exist, each with its advantages and disadvantages. One popular approach is one-hot encoding. Consider our dataset's "Country" column as an example. If we simply assign numerical codes (0, 1, 2...) to each country, then that wouldn't be ideal as the machine learning model might interpret a higher number as signifying a "better" country, which is not the case. So, one-hot encoding addresses this by creating a new binary feature for each unique category.
In the "Country" example, we'd have separate features for "France," "Germany," "Spain," and any other countries present. Each new feature would be a binary column (0 or 1), indicating the presence or absence of that specific country. For instance, a data point from France would have a 1 in the "France" feature and 0s in all other country features (like Germany and Spain). This approach effectively captures the categorical nature of the data while providing a numerical representation suitable for machine learning algorithms.
So, how would one-hot encoding look in our dataset?

One-hot encoding.

Let's consider the "Country" column in our dataset as an example.
If we, simply assign numerical codes (0, 1, 2...) to each country, then that wouldn't be ideal as the machine learning model might interpret a higher number as signifying a "better" country, which is not the case.

So one-hot encoding addresses this by creating a new **binary** feature for each unique category.

In the "Country" example, we'd have separate features for "France," "Germany," "Spain," and any other countries present.

Imagine we have a dataset with just the "Country" column containing France, Germany, and Spain. After one-hot encoding, our dataset might look like this:

Encoding the dependent variable

The 'Purchased' column in this dataset.

Since the 'Purchased' column only contains two distinct categories, "Yes" and "No", we can employ a simpler encoding technique compared to one-hot encoding.

Here, we'll leverage LabelEncoder from scikit-learn's preprocessing library.

So, the original "Country" column is replaced by these three new columns after one-hot encoding. As you can see, each row now has three features (France, Germany, Spain) instead of just the original "Country" column. Each feature indicates the

presence, which is indicated by 1 or absence, indicated by 0 of that specific country for that data point. So, one crucial advantage of using one-hot encoding in this scenario is that it avoids introducing an artificial order between the countries.

By applying one-hot encoding, we transform the categorical "Country" information into a format that machine learning models can understand and use to make better predictions. Now, applying one-hot encoding to the "Country" column, an Independent variable, using the popular scikit-learn library.

[GC]

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [0])], remainder='passthrough')
X = np.array(ct.fit_transform(X))
print(X)
```

First import the necessary modules, ColumnTransformer from sklearn.compose and OneHotEncoder from sklearn.preprocessing.

Then, we initialise a ColumnTransformer object, ct, which serves as a versatile tool for applying different transformations to specific columns within our data. To create the ColumnTransformer object, we call the ColumnTransformer class and pass in the required parameter. Inside the parentheses, we provide two parameters: transformers and remainder.

Within the transformers parameter, we specify the type of transformation we want to apply, the name of the transformation step for reference, and the index of the column(s) we want to transform. The name given to the transformation step, in this case, 'encoder', is arbitrary and can be anything you choose. OneHotEncoder() specifies the OneHotEncoder class for the transformation. [0] indicates that one-hot encoding will only apply to the first column.

The second parameter, remainder='passthrough', tells the ColumnTransformer to leave any columns not explicitly mentioned in the transformers list unchanged. In this case, the second and third columns will remain untouched.

After setting up the ColumnTransformer object ct, which specifies that we want to apply one-hot encoding to the first column and leave the remaining columns unchanged, the next step is to apply these transformations to our dataset X.

Here's what happens with this line X = np.array(ct.fit_transform(X)):

fit_transform is a method belonging to the ColumnTransformer class. This specific method combines two functionalities:

First, it performs fitting, meticulously analysing the data in X to understand its structure and characteristics, particularly those relevant to the specified transformers. In our case, it examines the first column to determine which unique values exist for one-hot encoding.

Second, it performs transforming. So once equipped with knowledge about the data, it applies the designated transformations to their respective columns. It meticulously converts the first column into its one-hot encoded representation while retaining the second and third columns as-is due to the remainder='passthrough' setting.

ct.fit_transform(X) calls upon the ColumnTransformer object we've created, ct, and invokes its fit_transform method.

The output of ct.fit_transform(X) is a transformed version of X, containing the one-hot encoded first column and the unchanged second and third columns.

Then, np.array() converts the result of ct.fit_transform(X) into a NumPy array. This step is often necessary because many machine learning algorithms in Scikit-learn expect input data to be in NumPy array format. Ensuring compatibility with these algorithms by converting the result to a NumPy array.
After converting the result into a NumPy array, it's commonly assigned back to X to replace the original data. This is done because the transformed data is typically used for further analysis or modeling.
So, in summary, X = np.array(ct.fit_transform(X)) fits the ColumnTransformer ct to the dataset X, applies the specified transformations, converts the result into a NumPy array, and finally assigns it back to X. This ensures that X now contains the transformed data ready for further processing or analysis.
Now print this transformed dataset.
[GC]
print(X)
[Output]
The first column has been transformed into a one-hot encoded representation. The original "Country" column has been replaced with three new features. Each feature is a binary column (0 or 1) indicating the presence or absence of that specific country for that data point.
Now, moving on to encoding the dependent variable, which is the 'Purchased' column in this dataset. Since the 'Purchased' column only contains two distinct categories, "Yes" and "No", we can employ a simpler encoding technique compared to one-hot encoding. Here, we'll leverage LabelEncoder from scikit-learn's preprocessing library. Unlike one-hot encoding, which creates a separate binary feature for each unique category, LabelEncoder assigns a numerical value (label) to each category, in this case, "No" can be mapped to 0 and "Yes" can be mapped to 1. This is a simple and effective way to encode binary categorical variables, where the order of the labels doesn't matter, unlike in ordinal encoding.
[GC]

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
y = le.fit_transform(y)
print(y)
```

We start by importing the LabelEncoder class from sklearn.preprocessing module. This class will help us encode the 'Purchased' column.
Next, we create an object of the LabelEncoder class and store it in the variable ‘le’. This instance will be used to perform the label encoding.
We then again use the fit_transform() method of the LabelEncoder object (le) on the 'Purchased' column (y). This single method fits the encoder to the 'Purchased' column and transforms it. The fit_transform() method analyses the data to understand the categories (in this case, "Yes" and "No") and assigns numerical labels accordingly. The transformed data is stored back in the variable y.
[Also here, it's not necessary to convert the output into a NumPy array because the fit_transform method already returns a NumPy array.]
Finally, let’s print the transformed 'Purchased' column (y) to observe the encoded values.

[GC]
print(y)
[Output]
As we can see the categorical data in the 'Purchased' column is converted into numerical labels, where "No" is represented as 0 and "Yes" is represented as 1. This allows us to use this data effectively in machine learning algorithms that require numerical inputs. In most cases, LabelEncoder assigns numerical labels alphabetically or based on the dataset's appearance order.
Now that we've successfully encoded the independent and dependent variables, we're ready for the next crucial step: splitting our dataset into training and testing sets.
Splitting the dataset into train and test sets consists of making two separate sets: one training set where we're going to train our machine learning model on existing observations and one test set where we're going to evaluate the performance of our model on new observations. It is important to understand that these new observations are exactly like future data that we will get and on which we will deploy our machine learning model. The training set acts as a practice ground for your model, allowing it to learn patterns and relationships from the data. The test set, on the other hand, simulates real-world scenarios where the model encounters unseen data. Let's implement the train-test split. So how are we going to do this? Well, we will do it with a function provided by the scikit-learn library. This library provides a convenient tool specifically designed for this task.
Scikit-learn offers a function called train_test_split within its model_selection module. This function efficiently splits your data into training and testing sets, perfectly suited for our needs.
It generates two pairs:
A training set containing features (X_train) and target variables (y_train).
A testing set containing features (X_test) and target variables (y_test).
[GC]
from sklearn.model_selection import train_test_split
Alright, so now that we have this function, let's use it! But first, let's create the four variables returned by the train_test_split function: X_train, X_test, y_train, and y_test. These variables will hold the split data, ready for training and testing our model.
[GC]
X_train, X_test, y_train, y_test
So that's the four variables returned by this train_test_split function, and since it is the function that returns these variables, let's take that function right away and add an equal sign and 'train_test_split', then some parenthesis.[add on the previous line of code]
[GC]
X_train, X_test, y_train, y_test = train_test_split()
The parameters we'll use. [add on the previous line of code]
[GC]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 1)
The first two, X and y, are our feature and target variables, respectively.
The 'test_size' parameter tells the function what proportion of the data should be allocated to the test set. Here, we're setting it to 0.2, which means 20% of the data

will be used for testing and the rest 80% for training. You can adjust this value based on your needs. A common approach is to use 80% for training and 20% for testing. Next is the random_state parameter which is an optional argument that controls the randomness used to shuffle the data before splitting. This parameter allows us to set a seed for the random number generator, which is set to 1 here This means that this split is now identified by this number. This ensures that every time we run the code, we get the same random split. It's helpful for reproducibility, meaning we can get the same results every time we run our code, which is important for debugging and sharing our work with others.

And that's it.

Now that we understand how train_test_split works, let's put it into practice and see how the training and testing data actually look! This will give us a better sense of how the data is divided and what the model will be working with.

[GC]

print(X_train)

[Output]

[GC]

print(X_test)

[Output]

[GC]

print(y_train)

[Output]

[GC]

print(y_test)

[Output]

[Explain about the output]

`X_train` contains 80% of the input data `X`, randomly selected for training the model, while `X_test` holds the remaining 20% for testing. Correspondingly, `y_train` comprises labels aligned with `X_train` for training, and `y_test` contains labels for testing.

Even after splitting our data into training and testing sets, we might encounter issues if our features have significantly different ranges. This is where feature scaling comes in. Feature scaling is a data preprocessing technique that transforms our features to a common scale, that

prevents one feature from dominating over others, ensuring that each feature contributes proportionately to the learning process of the machine learning model. If features have significantly different ranges and are not scaled, the model might give more weight to features with larger scales, leading to biased results. By scaling the features to a common scale, feature scaling helps mitigate this issue, ensuring that all features are equally considered during model training and evaluation.

It's important to note that feature scaling is applied after the train-test split because the test set is supposed to be a brand new set on which we will evaluate our machine learning model. We're not supposed to work with it for training to avoid data leakage. This maintains the integrity of our evaluation process, allowing us to assess the performance of our model on unseen data accurately.

There are two common techniques for feature scaling: Normalization and Standardization.

Normalisation is a good choice when we don't know the underlying distribution of our data
and Standardization is good to use when our data follows a normal distribution.
Now, let's perform feature scaling using the scikit-learn library. Utilizing the StandardScaler class from the preprocessing module, which facilitates standardisation on the matrix of features of the training set and the matrix of features of the test set. Let's start by importing this class from scikit-learn:
[GC]
from sklearn.preprocessing import StandardScaler
With this, we can access the StandardScaler class from the preprocessing module.
Now, let's create an object of the StandardScaler class and we'll call it sc for standard scaler that we'll use to perform the standardization.
[GC]
sc = StandardScaler()
The parentheses in StandardScaler() are empty to use the default settings for calculating mean and standard deviation during data fitting.
Moving to the next step. But before that, let's address a crucial question: Do we apply feature scaling to dummy variables?
In our dataset, the first three columns represent the dummy variables resulting from the one-hot encoding of the "Country" column. Considering their binary nature, we might choose not to scale them. Instead, we'll focus on scaling the continuous features, such as age and salary, to ensure they're on a similar scale for better model performance.
Now that we've addressed the question of whether to apply feature scaling to dummy variables, let's proceed with standardizing our continuous features. As mentioned, we want to ensure that features like age and salary are on a similar scale for improved model performance.
We'll apply standardization to our continuous features using sc.fit_transform function and assign the transformed values back to X_train. Here's the line of code:
[GC]
X_train[:, 3:] = sc.fit_transform(X_train[:, 3:])
X_train[:, 3:] selects all features from the training data X_train starting from the fourth column or columns from index 3 up to the last column.
Next, we'll apply the fit_transform() method of the 'sc' object to our training data. This method performs two crucial steps in one go, which is fit and transform.
.fit() method analyses the features in X_train[:, 3:] to compute each feature's mean and standard deviation.
The transform() method transforms the data using those values. It essentially applies the standardization formula we discussed earlier to center and scale the features.
Now, let's apply the same transformation to our testing data, X_test, to ensure consistency in scaling between our training and testing datasets:
[GC]
X_test[:, 3:] = sc.transform(X_test[:, 3:])
Similar to what we did with X_train, X_test[:, 3:] selects all features from the testing data X_test starting from the fourth column, ensuring that only the continuous features are standardized. Next, we use the transform() method of the sc object. However, notice that we're using transform() instead of fit_transform(). This is

because we've already computed the mean and standard deviation during the fitting stage with our training data.
The transform() method then applies the same transformation to the testing data using the mean and standard deviation calculated from the training data. This ensures that the testing data is scaled in the same way as the training data, maintaining consistency and preventing data leakage.
After applying the transformations to both our training and testing datasets, let's examine the transformed data:
[GC]
print(X_train)
[Output]
As you can see here the last two columns age and salary have been standardized.
Now, check the transformed testing data:
[GC]
print(X_test)
[Output]
Similarly, the testing data X_test now also contains standardised continuous features.
We're done with data preprocessing and ready to start building the machine learning models that will perform predictions.