How to Convert Categorical Data to Numeric Using Pandas

Authored by

stats writer

December 3, 2025

Understanding Categorical Data and the Need for Encoding

In the realm of data science and statistical modeling, data is often classified into various types. A crucial distinction exists between numerical data and categorical variables. A **categorical variable** represents classifications or groupings, such as gender, country, or product type. While humans easily process these labels, most computational models, particularly algorithms used in machine learning, require all input features to be represented numerically.

The process of converting categorical labels into a quantitative format is known as **encoding**. Without effective encoding, algorithms cannot perform mathematical operations or correctly interpret the relationships between features. For example, a categorical variable like 'Color' (Red, Blue, Green) cannot be directly fed into a linear regression model. Therefore, mastering the techniques available in libraries like Pandas for this transformation is fundamental to any data preparation workflow.

There are several methods for encoding categorical data, each suited to different scenarios. The choice often depends on whether the variable is **nominal** (categories without inherent order, like team names) or **ordinal** (categories with a meaningful rank, like education levels). Two primary strategies we will explore involve using `pd.factorize()` for label encoding and briefly discussing `pd.get_dummies()` for one-hot encoding, as both are highly relevant when working with a DataFrame.

The Primary Method: Leveraging `pandas.factorize()`

One of the most straightforward and effective ways to convert a categorical column into numerical representations in Pandas is by utilizing the built-in function `pd.factorize()`. This function is designed to find unique values in a sequence and map them to numerical labels, starting from zero. It is an excellent choice when performing **label encoding**--assigning a unique integer to each unique categorical value.

When `pd.factorize()` is executed, it returns two important components: an array of integer codes representing the categories, and a unique list of the original categories (the index). For our purposes, when we aim to replace the categorical column directly with its numerical representation, we only need the first element of the returned tuple, which contains the integer codes. This results in a clean, indexed numerical column suitable for many analytical tasks.

However, it is crucial to use `pd.factorize()` judiciously. While it is fast and efficient, assigning arbitrary numerical values (0, 1, 2, etc.) implies an ordinal relationship that might not exist. If the categories are nominal (e.g., 'Apple', 'Banana', 'Cherry'), the resulting numbers (0, 1, 2) could mislead a machine learning model into assuming that 'Cherry' (2) is somehow "greater" than 'Apple' (0). We will discuss this caveat further when comparing it to one-hot encoding.

The original syntax for applying this transformation to a specific column within a DataFrame is provided below:

df = pd.factorize(df)

Detailed Syntax for Single Column Conversion

The syntax shown above provides an atomic operation for transforming a single series. Let us break down exactly what happens during this assignment. We are overwriting the specified column in the DataFrame (`df`) with the results of the factorization operation performed on its current contents.

The key part is `pd.factorize(df)`. The `factorize()` function analyzes the input series and assigns integer codes sequentially based on the order of appearance of the unique values. The index is used specifically to select the array of integer codes. The index would return the unique array of categories themselves, which is useful if we need to store the mapping for future use or interpretation.

This method is highly beneficial when dealing with columns that have a large number of unique categories (high cardinality). Unlike methods that require explicitly mapping each category to an integer using dictionaries, `factorize()` handles the creation of this numerical mapping automatically and efficiently, making data preparation much faster, particularly with large datasets loaded into Pandas.

Practical Application: Converting One Categorical Variable

To illustrate the practical application of `pd.factorize()` for a single column, let us work with a sample dataset representing sports team performance statistics. We aim to convert the 'team' column, which holds categorical identifiers (A, B, C), into a numerical feature.

First, we define the starting DataFrame:

import pandas as pd

#create DataFrame
df = pd.DataFrame({'team': ,
'position': ,
'points': ,
'rebounds': })

#view DataFrame

df

team position points rebounds
0 A G 5 11
1 A G 7 8
2 A F 7 10
3 B G 9 6
4 B F 12 6
5 B C 9 5
6 C G 9 9
7 C F 4 12
8 C C 13 10

Now, we execute the transformation specifically targeting the 'team' column:

#convert 'team' column to numeric
df = pd.factorize(df)

#view updated DataFrame
df

team position points rebounds
0 0 G 5 11
1 0 G 7 8
2 0 F 7 10
3 1 G 9 6
4 1 F 12 6
5 1 C 9 5
6 2 G 9 9
7 2 F 4 12
8 2 C 13 10

The output clearly shows that the 'team' column is now represented by integers. The factorize() function assigned codes based on the order it encountered them in the data:

The first unique value encountered, '**A**', was mapped to **0**.
The second unique value, '**B**', was mapped to **1**.
The third unique value, '**C**', was mapped to **2**.

This resulting numerical column is ready for integration into most predictive models, providing a

simple yet powerful encoding mechanism for nominal categorical variables when computational speed is prioritized.

Converting Multiple Categorical Columns Efficiently

When preparing larger datasets, manual conversion of every single categorical column can be tedious and prone to error. Fortunately, Pandas allows us to identify all columns of a certain data type--typically `object` or `category`--and apply the `factorize()` method across them simultaneously using the `apply()` function in conjunction with `select_dtypes()`.

The general strategy involves three steps: first, identifying all columns containing categorical or string data; second, applying the `factorize()` transformation to these columns; and finally, updating the DataFrame with the newly encoded values.

The general syntax for this comprehensive conversion is as follows:

#identify all categorical variables (those with 'object' dtype)
cat_columns = df.select_dtypes().columns

#convert all categorical variables to numeric using a lambda function
df = df.apply(lambda x: pd.factorize(x))

Let us re-examine our sports dataset, which contains two categorical columns: 'team' and 'position'. We will use the automated syntax to convert both simultaneously.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'team': ,
'position': ,
'points': ,
'rebounds': })

#view DataFrame
df

team position points rebounds
0 A G 5 11
1 A G 7 8
2 A F 7 10
3 B G 9 6
4 B F 12 6

5 B C 9 5
6 C G 9 9
7 C F 4 12
8 C C 13 10

Executing the batch conversion logic:

#get all categorical columns
cat_columns = df.select_dtypes().columns

#convert all categorical columns to numeric
df = df.apply(lambda x: pd.factorize(x))

#view updated DataFrame
df

team position points rebounds
0 0 0 5 11
1 0 0 7 8
2 0 1 7 10
3 1 0 9 6
4 1 1 12 6
5 1 2 9 5
6 2 0 9 9
7 2 1 4 12
8 2 2 13 10

In this final result, both 'team' and 'position' columns are now numerical. Note that the categories in 'position' ('G', 'F', 'C') were also assigned sequential codes (0, 1, 2) based on their first appearance in that specific column.

Alternative Encoding Strategy: One-Hot Encoding with `get_dummies()`

While `pd.factorize()` implements a form of label encoding, another immensely popular and often safer method for dealing with nominal categorical data is **One-Hot Encoding**, achieved using the Pandas `get_dummies()` function. This approach transforms a single categorical column into multiple binary (0 or 1) columns, often referred to as **dummy variables**.

The core concept is to create a new column for every unique value found in the original categorical feature. For any given row, only one of these new dummy columns will contain a `1` (indicating

presence), while all others contain `0` (indicating absence). This eliminates the false sense of order or magnitude that label encoding can introduce, which is critical when training linear models in machine learning.

Although the primary examples focused on `factorize()`, understanding `get_dummies()` is essential for a complete data preparation toolkit. The basic syntax is simply: `pd.get_dummies(df)`. The resulting output is a new DataFrame containing the binary columns, which must then be concatenated back to the main dataset while often dropping the original categorical column and one of the dummy variables to prevent multicollinearity (the Dummy Variable Trap).

For instance, if we applied this to the 'position' column ('G', 'F', 'C'), we would create three new columns: 'position_G', 'position_F', and 'position_C'. This expansion of features is the trade-off for eliminating implicit order bias, and developers must manage the potential increase in the dimensionality of their dataset.

Choosing the Right Method: Label Encoding vs. One-Hot Encoding

Deciding between **Label Encoding** (using `factorize()`) and **One-Hot Encoding** (using `get_dummies()`) is a key decision in the feature engineering stage. The choice should be driven by the nature of the data and the requirements of the downstream model.

Use Label Encoding (`factorize()`) when:

The categorical feature is **ordinal** (has a meaningful inherent order, e.g., Small, Medium, Large). In this case, the assigned integers (0, 1, 2) accurately reflect the underlying structure.
You are using algorithms that handle integer-encoded categorical data well or are non-linear, such as Tree-based models (Decision Trees, Random Forests, Gradient Boosted Trees), where the ordinal relationship is less of a concern.
The column has very **high cardinality** (many unique values). One-hot encoding a high-cardinality feature can lead to a massive, sparse DataFrame, which is computationally inefficient.

Use One-Hot Encoding (`get_dummies()`) when:

The categorical feature is **nominal** (has no inherent order, e.g., 'Team A', 'Team B'). This prevents the model from incorrectly assuming a mathematical relationship between the labels.
You are using linear models (e.g., Linear Regression or Logistic Regression) which are highly sensitive to the scale and interpretation of numerical inputs.
The number of unique categories (cardinality) is relatively low, making the resulting increase in feature count manageable.

Both methods offer powerful ways to bridge the gap between human-readable text labels and the numerical input required by machine learning systems. Understanding the implications of each encoding type ensures that the data preparation process supports the success of the modeling phase. For comprehensive details on the specific functionality used in this article, please refer to the official Pandas documentation for `factorize()`.

Note: You can find the complete documentation for the pandas **factorize()** function here.