Python Feature Engineering Cookbook

Stephen CUI[1]

2022 年 12 月 19 日

[1]cuixuanStephen@gmail.com

目录

Chapter 1

Imputing Missing Data

Missing data, that is, the absence of values for certain observations, is an unavoidable problem in most data sources.

The act of replacing missing data with statistical estimates of missing values is called imputation. The goal of any imputation technique is to produce a complete dataset. There are multiple imputation methods that we can use, depending on whether the data is missing at random, the proportion of missing values, and the machine learning model we intend to use.

1.1 Technical requirements

We will use the Credit Approval dataset from the UCI Machine Learning Repository (https://archive.ics.uci.edu/). To prepare the dataset, follow these steps:

1. Visit https://archive.ics.uci.edu/ml/machine-learning-databases/credit-screening/.

2. Click on **crx.data** to download the data.

```python
import random
import numpy as np
import pandas as pd
data = pd.read_csv('data/crx.data', header=None)
varname = [f"A{s}" for s in range(1, 17)]
data.columns = varname
data = data.replace("?", np.nan)
data["A2"] = data["A2"].astype("float")
data["A14"] = data["A14"].astype("float")
data["A16"] = data["A16"].map({"+": 1, "-": 0})
data.rename(columns={"A16":"target"}, inplace=True)
```

```python
random.seed(37)
values = list(set([random.randint(0, len(data)) for p in range(0, 100)]))
data.loc[values, ["A3", "A8", "A9", "A10"]] = np.nan
data.to_csv('data/credit_approval_uci.csv', index=False)
```

Tip

Make sure you store the dataset in the same folder from which you will execute the code in the recipes.

处理后的数据集有一个target字段以及15个（A1-A15）特征字段组成。其中A11, A12, A13, A15, Target不包含缺失值（Figure 1.1）。

1.2 Removing observations with missing data

Complete Case Analysis (CCA), also called list-wise deletion of cases, consists of discarding observations with missing data. CCA can be applied to both categorical and numerical variables. With CCA, we preserve the distribution of the variables after the imputation, provided the data is missing at random and only in a small proportion of observations. However, if data is missing for many variables, CCA may lead to the removal of a large portion of the dataset.

1.2.1 How to do it...

```python
import matplotlib.pyplot as plt
import pandas as pd
data = pd.read_csv('data/credit_approval_uci.csv')
with plt.style.context('ggplot'):
    data.isnull().mean().sort_values(ascending=True).plot.bar(rot=45)
    plt.ylabel("Proportion of missing data")
    plt.title("Proportion of missing data per variable")
```

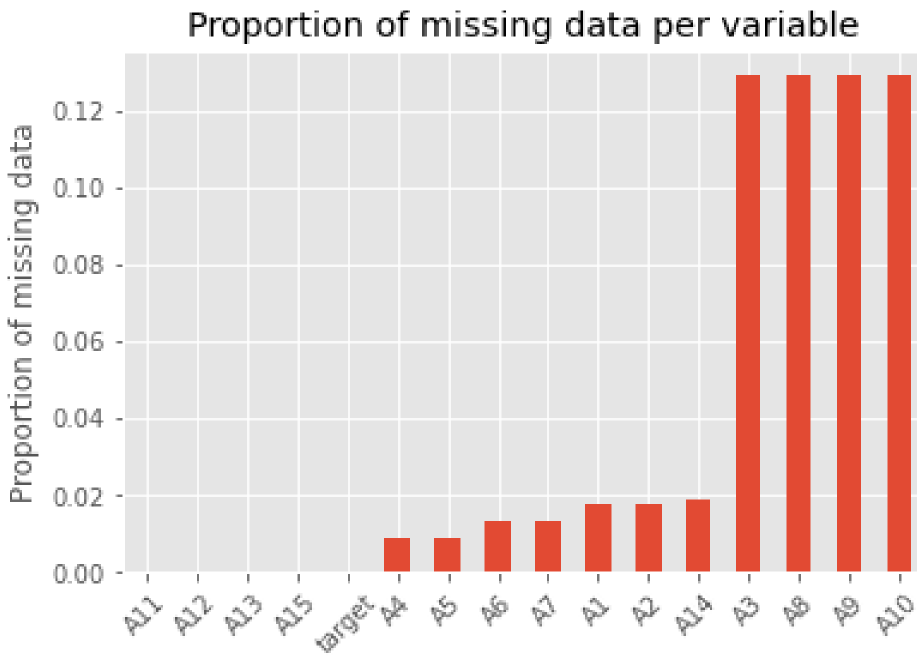

图 1.1: Proportion of missing data per variable

```python
data_cca = data.dropna()
print(f"Total number of observations: {len(data)}")
print(f"Number of observations without missing data: {len(data_cca)}")
from feature_engine.imputation import DropMissingData
cca = DropMissingData(variables=None, missing_only=True)
cca.fit(data)
print(cca.variables_)
data_cca = cca.transform(data)
```

Tip

The dropna() method drops observations with any missing value by default. We can remove observations with missing data in a subset of variables like this: `data.dropna(subset=["A3","A4"])`.

Tip

To remove observations with missing data in a subset of variables, use DropMissingData(variables=['A3', 'A4']). To remove observations with missing values in at least 5% of the variables, use DropMissingData(threshold=0.95).

1.3 Performing mean or median imputation

Mean or median imputation consists of replacing missing values with the mean or median variable. The mean or median is calculated using a train set, and these values are used to impute missing data in train and test sets, as well as in all future data we intend to use with the machine learning model. Scikit-learn and feature-engine transformers learn the mean or median from the train set and store these parameters for future use out of the box.

> **Tip**
>
> Use mean imputation if variables are normally distributed and median imputation otherwise. Mean and median imputation may distort the distribution of the original variables if there is a high percentage of missing data.

1.3.1 How to do it...

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from feature_engine.imputation import MeanMedianImputer

data = pd.read_csv('data/credit_approval_uci.csv')

X_train, X_test, y_train, y_test = train_test_split(data.drop('target',
                                                              axis='columns'),
                                                    data['target'],
                                                    test_size=.3,
                                                    random_state=39)
numeric_vars = X_train.select_dtypes(exclude='O').columns.to_list()
numeric_vars
median_values = X_train[numeric_vars].median().to_dict()
median_values
for df in [X_train, X_test]:
# unable to modify X_train and X_test
# because df pointer to different address
#     df = df.fillna(value=median_values)
    df.fillna(value=median_values, inplace=True)
# equivalent to following codes
# X_train = X_train.fillna(value=median_values)
# X_test = X_test.fillna(value=median_values)
```

```

X_train, X_test, y_train, y_test = train_test_split(data.drop('target',
                                                              axis='columns'),
                                                    data['target'],
                                                    test_size=.3,
                                                    random_state=39)
remaining_vars = [var for var in X_test.columns if var not in numeric_vars]
remaining_vars
imputer = SimpleImputer(strategy='median')
ct = ColumnTransformer([('imputer', imputer, numeric_vars)],
                       remainder='passthrough')
ct.fit(X_train)
ct.named_transformers_.imputer.statistics_
# warning: unable to perform median imputation
# for df in [X_train, X_test]:
#     df = ct.transform(df)
# return NumPy arrays
X_train = ct.transform(X_train)
X_test = ct.transform(X_test)
X_train = pd.DataFrame(X_train, columns=numeric_vars + remaining_vars)
X_train
X_test = pd.DataFrame(X_test, columns=numeric_vars + remaining_vars)
X_test

X_train, X_test, y_train, y_test = train_test_split(data.drop('target',
                                                              axis='columns'),
                                                    data['target'],
                                                    test_size=.3,
                                                    random_state=39)
imputer = MeanMedianImputer(imputation_method='median', variables=numeric_vars)
imputer.fit(X_train)
imputer.imputer_dict_
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)
print(X_train[numeric_vars].isnull().any().any())
print(X_test[numeric_vars].isnull().any().any())
```

1.3.2 How it works...

The mean or median values should be learned from the train set variables. Thus, we divided the dataset into train and test sets using scikit-learn's `train_test_split()` function. The function takes the predictor variables, the target, the fraction of observations to retain in the test set, and a `random_state` value for reproducibility as arguments. We obtained a train set with 70% of the original observations and a test set with 30% of the original observations. The 70:30 split was done at random.

To replace the missing values using scikit-learn, we used `SimpleImputer()` with strategy set to ”median”. To impute only numerical variables, we used `ColumnTransformer()`, which takes the imputer and the numerical variable names in a list as parameters. With the passthrough argument set to ”remainder”, we make `ColumnTransformer()` **return all the variables in the final output, the imputed ones followed by the remaining ones.**

Tip

SimpleImputer() operates on the entire DataFrame and returns NumPy arrays. To impute just a subset of variables, we need to use ColumnTransformer(). In contrast, MeanMedianImputer() can take an entire DataFrame and yet it will only impute the specified variables, returning a pandas DataFrame.

1.4 Imputing categorical variables

Categorical variables usually contain strings as values, instead of numbers. We replace missing data in categorical variables with the most frequent category, or with a different string. Frequent categories are estimated using the train set and then used to impute values in the train, test, and future datasets. Thus, we need to learn and store these values, which we can do using scikit-learn and feature- engine's out-of-the-box transformers.

1.4.1 How to do it...

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from feature_engine.imputation import CategoricalImputer

data = pd.read_csv('../data/credit_approval_uci.csv')

X_train, X_test, y_train, y_test = train_test_split(data.drop('target',
                                                              axis='columns'),
                                                    data['target'],
                                                    test_size=.3,
```

```
                                                    random_state=37)
# using pandas
categorical_vars = X_train.select_dtypes(include='O').columns.to_list()
# important: iloc[0]
frequent_values = X_train[categorical_vars].mode().iloc[0].to_dict()
print(frequent_values)

X_train = X_train.fillna(value=frequent_values)
X_test = X_test.fillna(value=frequent_values)
print(X_train[categorical_vars].isnull().any().any())
print(X_test[categorical_vars].isnull().any().any())

# Imputation with a string
X_train, X_test, y_train, y_test = train_test_split(data.drop('target',
                                                                axis='columns'),
                                                    data['target'],
                                                    test_size=.3,
                                                    random_state=37)
imputation_dict = {var: 'no_data' for var in categorical_vars}
X_train = X_train.fillna(value=imputation_dict)
X_test = X_test.fillna(value=imputation_dict)
print(X_train['A1'].value_counts())

# using scikit-learn
X_train, X_test, y_train, y_test = train_test_split(data.drop('target',
                                                                axis='columns'),
                                                    data['target'],
                                                    test_size=.3,
                                                    random_state=37)

remaining_vars = [
    var for var in X_train.columns if var not in categorical_vars
]

imputation_dict = {var: 'not_data' for var in categorical_vars}
imputer = SimpleImputer(strategy='most_frequent')

ct = ColumnTransformer([('Imputer', imputer, categorical_vars)],
```

```
                        remainder='passthrough')

ct.fit(X_train)
# this string "Imputer" here must be identical to
# the string 'Imputer' in ColumnTransformer
print(ct.named_transformers_.Imputer.statistics_)

X_train = ct.transform(X_train)
X_test = ct.transform(X_test)

X_train = pd.DataFrame(X_train, columns=categorical_vars + remaining_vars)
print(X_train[categorical_vars].isnull().any().any())

# using feature-engine
X_train, X_test, y_train, y_test = train_test_split(data.drop('target',
                                                              axis='columns'),
                                                    data['target'],
                                                    test_size=.3,
                                                    random_state=37)

imputer = CategoricalImputer(imputation_method='frequent', variables=categorical_vars)

imputer.fit(X_train)

print(imputer.imputer_dict_)

X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)
print(X_train[categorical_vars].isnull().any().any())
print(X_test[categorical_vars].isnull().any().any())
```

1.4.2 How it works...

`SimpleImputer()` returns a NumPy array by default. We converted this array into a pandas DataFrame. We had to pass the variable names in the correct order: the imputed variables are located first in the array, followed by the remaining variables. 这个需要特别的注意。

Tip

Note that, unlike SimpleImputer(), CategoricalImputer() will only impute categorical variables, unless specifically told not to do so by setting the ignore_format parameter to True.

1.4.3 Replacing missing values with an arbitrary number

When replacing missing values with arbitrary numbers, we need to be careful not to select a value close to the mean, the median, or any other common value of the distribution.

Tip

Arbitrary number imputation can be used when data is not missing at random, when we are building non-linear models, and when the percentage of missing data is high. This imputation technique distorts the original variable distribution.

1.4.4 How to do it...

```python
# Replacing missing values with an arbitrary number

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from feature_engine.imputation import ArbitraryNumberImputer

data = pd.read_csv('../data/credit_approval_uci.csv')

X_train, X_test, y_train, y_test = train_test_split(data.drop('target', axis='columns'),
                                                    data['target'],
                                                    test_size=.3,
                                                    random_state=37)

arbitrary_cols = ['A2', 'A3', 'A8', 'A11']
print(X_train[arbitrary_cols].max().max())

X_train[arbitrary_cols] = X_train[arbitrary_cols].fillna(99)
X_test[arbitrary_cols] = X_test[arbitrary_cols].fillna(99)
print(X_train[arbitrary_cols].isnull().any().any())
print(X_test[arbitrary_cols].isnull().any().any())
```

```
23 # using scikit-learn
24 X_train, X_test, y_train, y_test = train_test_split(data.drop('target', axis='columns'),
25                                                     data['target'],
26                                                     test_size=.3,
27                                                     random_state=37)
28 imputer = SimpleImputer(strategy='constant', fill_value=99)
29 imputer.fit(X_train[arbitrary_cols])
30 X_train[arbitrary_cols] = imputer.transform(X_train[arbitrary_cols])
31 X_test[arbitrary_cols] = imputer.transform(X_test[arbitrary_cols])
32 print(X_train[arbitrary_cols].isnull().any().any())
33 print(X_test[arbitrary_cols].isnull().any().any())
34
35
36 # using feature-engine
37 X_train, X_test, y_train, y_test = train_test_split(data.drop('target', axis='columns'),
38                                                     data['target'],
39                                                     test_size=.3,
40                                                     random_state=37)
41 imputer = ArbitraryNumberImputer(arbitrary_number=99, variables=arbitrary_cols)
42
43 X_train = imputer.fit_transform(X_train)
44 X_test = imputer.transform(X_test)
45 print(X_train[arbitrary_cols].isnull().any().any())
46 print(X_test[arbitrary_cols].isnull().any().any())
```

第16行，首先要找到需要处理缺失值字段的最大值，然后将填充的任意值设置为比这个最大值要大。

第18-19行，也可以使用`imputation_dict`为每列单独设置填充值，将字典的key和value分别设置为字段名和想要填充的任意值。

第28行，如果你的数据集存在有缺失值的分类变量，那么`SimpleImputer()`也会用value将其填充。

On line 41，`ArbitraryNumberImputer()` can automatically select all numerical variables in the train set if we set the `variables` parameter to `None`.

1.5 Finding extreme values for imputation

Replacing missing values with a value at the end of the variable distribution (extreme values) is equivalent to replacing them with an arbitrary value, but instead of identifying the arbitrary values manually, these values are automatically selected as those at the very end of the variable distribution. Missing data can be replaced with a value that is greater or smaller than the remaining values in the variable. To select a value that is greater, we can use the mean plus a factor of the standard deviation, or the 75th quantile + (IQR * 1.5), where IQR is the IQR given by the 75th quantile - the 25th quantile. To replace missing data with values that are smaller than the remaining values, we can use the mean minus a factor of the standard deviation, or the 25th quantile – (IQR * 1.5).

Tip

End-of-tail imputation may distort the distribution of the original variables, so it may not be suitable for linear models.

1.5.1 How to do it...

```
import pandas as pd
from sklearn.model_selection import train_test_split
from feature_engine.imputation import EndTailImputer

data = pd.read_csv('../data/credit_approval_uci.csv')

numeric_vars = [
    var for var in data.select_dtypes(exclude='O').columns.to_list() if var != 'target'
]
print(numeric_vars)

X_train, X_test, y_train, y_test = train_test_split(
    data[numeric_vars],
    data['target'],
    test_size=.3,
    random_state=0)

IQR = X_train.quantile(0.75) - X_train.quantile(0.25)
print(IQR)

imputation_dict = (X_train.quantile(.75) + 1.5 * IQR).to_dict()
print(imputation_dict)

X_train = X_train.fillna(value=imputation_dict)
X_test = X_test.fillna(value=imputation_dict)

print(X_train.isnull().any().any())
```

```
28 print(X_test.isnull().any().any())
29
30 # We can also replace missing data with values at the left tail of the distribution
31 X_train, X_test, y_train, y_test = train_test_split(
32     data[numeric_vars],
33     data['target'],
34     test_size=.3,
35     random_state=0)
36
37 imputer = EndTailImputer(
38     imputation_method='iqr',
39     tail='right',
40     fold=3,
41     variables=None)
42 imputer.fit(X_train)
43
44 print(imputer.imputer_dict_)
45
46 X_train = imputer.transform(X_train)
47 X_test = imputer.transform(X_test)
48
49 print(X_train.isnull().any().any())
50 print(X_test.isnull().any().any())
```

1.5.2 How it works...

On line 21, If we want to use the Gaussian approximation instead of the IQR proximity rule, we can calculate the value to replace missing data using `imputation_dict = (X_train.mean() + 3 * X_train.std()).to_dict()`.

On line 37-41, To use the mean and standard deviation to calculate the replacement values, we need to set `imputation_method="Gaussian"`. We can use 'left' or 'right' in the tail argument to specify the side of the distribution where we' ll place the missing values.

1.6 Marking imputed values

A missing indicator is a binary variable that takes the value 1 or True to indicate whether a value was missing, or 0 or False otherwise. It is common practice to replace missing observations with the mean, median, or most frequent category while simultaneously marking those missing observations with missing indicators.

在数据库中，使用null查询会降低查询的性能，因此常见的操作是将缺失值转化为0来进行处理。

1.6.1 How to do it...

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from feature_engine.imputation import (
    AddMissingIndicator,
    CategoricalImputer,
    MeanMedianImputer,
)

data = pd.read_csv('../data/credit_approval_uci.csv')

X_train, X_test, y_train, y_test = train_test_split(
    data.drop('target', axis='columns'), data['target'],
    test_size=.3, random_state=37
)

varnames = ['A1', 'A3', 'A4', 'A5', 'A6', 'A7', 'A8']

indicators = [f'{var}_na' for var in varnames]

X_train[indicators] = X_train[varnames].isnull().astype(int)
X_test[indicators] = X_test[varnames].isnull().astype(int)

X_train.sample(5)

# using feature-engine
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('target', axis='columns'), data['target'],
    test_size=.3, random_state=37
)

imputer = AddMissingIndicator(
    variables=None,
    missing_only=True,
```

```python
)
imputer.fit(X_train)
imputer.variables_

X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)

pipe = Pipeline([
    ('ind', AddMissingIndicator(missing_only=True)),
    ('cat', CategoricalImputer(imputation_method='frequent')),
    ('num', MeanMedianImputer(imputation_method='mean'))
])

X_train = pipe.fit_transform(X_train)
X_test = pipe.fit_transform(X_test)

# using scikit-learn
X_train, X_test, y_train, y_test = train_test_split(
    data.drop('target', axis='columns'), data['target'],
    test_size=.3, random_state=37
)

num_vars = X_train.select_dtypes(exclude='O').columns.to_list()
cat_vars = X_train.select_dtypes(include='O').columns.to_list()

pipe = ColumnTransformer(
    [
        ('num_imputer',
            SimpleImputer(strategy='mean',
                          add_indicator=True),
            num_vars),
        ('cat_imputer',
            SimpleImputer(strategy='most_frequent',
                          add_indicator=True),
            cat_vars)]
)
X_train = pipe.fit_transform(X_train)
X_test = pipe.fit_transform(X_test)
```

这段代码中，仔细看Pipeline的操作。

On line 45, 此处的Pipeline首先在DataFrame后面添加了存在missing value的字段，字段名设置为原字段名_na，然后将分类型变量设置用众数填充，数值型变量用均值填充。 Feature-engine imputers automatically identify all numerical or categorical variables, modifying only the appropriate variables. So there is no need to slice the data or pass the variable names as arguments to the transformers in this case

1.7 Performing multivariate imputation by chained equations (MICE)

Multivariate imputation methods, as opposed to univariate imputation, use multiple variables to estimate the missing values. In other words, the missing values of a variable are modeled based on the other variables in the dataset. **Multivariate Imputation by Chained Equations (MICE)** models each variable with missing values as a function of the remaining variables and uses that estimate for imputation.

The following steps are required to perform MICE:

1. A simple univariate imputation is performed for every variable with missing data, for example, median imputation.
2. One specific variable is selected, say, `var_1`, and the missing values are set back to missing.
3. A model is trained to predict `var_1` using the remaining variables as input features.
4. The missing values of `var_1` are replaced with the new estimates.
5. Steps 2 to 4 are repeated for each of the remaining variables.

这里有个MICE的例子，更详细的内容见Paper。

Once all the variables have been modeled based on the rest, a cycle of imputation is concluded. Multiple imputation cycles are carried out, typically 10. The idea is that by the end of the cycles, the distribution of the imputation parameters should have converged, which means that we should have found the best estimates for the missing data.

1.7.1 How to do it...

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import BayesianRidge
from sklearn.impute import IterativeImputer

variables = ['A2', 'A3', 'A8', 'A11', 'A14', 'A15', 'target']

data = pd.read_csv('../data/credit_approval_uci.csv',
                   usecols=variables)
```

```

X_train, X_test, y_train, y_test = train_test_split(
    data.drop('target', axis='columns'), data['target'],
    test_size=.3, random_state=37
)

imputer = IterativeImputer(
    estimator=BayesianRidge(),
    max_iter=10, random_state=37
)

imputer.fit(X_train)
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)
# Remember that scikit-learn returns NumPy arrays and not DataFrames.
pd.DataFrame(X_train).isnull().any().any()
pd.DataFrame(X_test).isnull().any().any()
```

1.7.2 See also

- A Multivariate Technique for Multiply Imputing Missing Values Using a Sequence of Regression Models
- mice: Multivariate Imputation by Chained Equations in R

1.8 Estimating missing data with nearest neighbors

In imputation with **K-Nearest Neighbors (KNN)**, missing values are replaced with the mean value from their k closest neighbors. The neighbors of each observation are found utilizing distances like the Euclidean distance, and the replacement value can be estimated as the mean or weighted mean of the neighbor’s value, where further neighbors have less influence on the replacement value.

1.8.1 How to do it...

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.impute import KNNImputer

variables = ['A2', 'A3', 'A8', 'A11', 'A14', 'A15', 'target']
data = pd.read_csv('../data/credit_approval_uci.csv',
```

```
                     usecols=variables)

X_train, X_test, y_train, y_test = train_test_split(
    data.drop('target', axis='columns'),
    data['target'], test_size=.3, random_state=37
)

imputer = KNNImputer(n_neighbors=5, weights='distance')
imputer.fit(X_train)
X_train = imputer.transform(X_train)
X_test = imputer.transform(X_test)
pd.DataFrame(X_train).isnull().any().any()
pd.DataFrame(X_test).isnull().any().any()
```

Chapter 2

Encoding Categorical Variables

Categorical variables are those whose values are selected from a group of categories or labels. In some categorical variables, the labels have an intrinsic order. These are called ordinal categorical variables. Variables in which the categories do not have an intrinsic order are called nominal categorical variables。

The values of categorical variables are often encoded as strings. To train mathematical or machine learning models, we need to transform those strings into numbers. The act of replacing strings with numbers is called categorical encoding.

补充： 数据的四个等级

1. nominal level（定类等级）
2. ordinal level（定序等级）
3. interval level（定距等级）
4. ratio level（定比等级）

The values of categorical variables are often encoded as strings. To train mathematical or machine learning models, we need to transform those strings into numbers. The act of replacing strings with numbers is called categorical encoding.

2.1 Technical requirements

We will also use the open-source Category Encoders Python library, which can be installed using pip:

```
pip install category_encoders
```

We will also use the Credit Approval dataset(section 1.1 Technical requirements).

```python
import random
import numpy as np
import pandas as pd

data = pd.read_csv('../data/credit_approval_uci.csv')

cat_cols = [
    c for c in data.columns if data[c].dtypes == 'O'
]
num_cols = [
    c for c in data.columns if data[c].dtypes != 'O'
]

data[num_cols] = data[num_cols].fillna(0)
data[cat_cols] = data[cat_cols].fillna('Missing')

if(not data.isnull().any().any()):
    print('not exist missing value')
```

2.2 Creating binary variables through one-hot encoding

In one-hot encoding, we represent a categorical variable as a group of binary variables, where each binary variable represents one category. The binary variable takes a value of 1 if the category is present in an observation, or 0 otherwise.

A categorical variable with k unique categories can be encoded using $k-1$ binary variables. For the Color variable, which has three categories ($k=3$; red, blue, and green), we need to create two ($k-1=2$) binary variables to capture all the information so that the following occurs:

- If the observation is red, it will be captured by the red variable (red = 1, blue = 0).
- If the observation is blue, it will be captured by the blue variable (red = 0, blue = 1).
- If the observation is green, it will be captured by the combination of red and blue (red = 0, blue = 0).

Encoding into $k-1$ binary variables is well-suited for linear models. **There are a few occasions in which we may prefer to encode the categorical variables with k binary variables:**

- When training decision trees since they do not evaluate the entire feature space at the same time.
- When selecting features recursively.
- When determining the importance of each category within a variable.

2.2.1 How to do it...

In this recipe, we will compare the one-hot encoding implementations of pandas, scikit-learn, Feature-engine, and Category Encoders.

```
import pandas as pd
from sklearn.model_selection import train_test_split

data = pd.read_csv('../data/credit_approval_uci.csv')
```

```
# Let' s do one-hot encoding using pandas
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(labels=['target'], axis='columns'), data['target'],
    test_size=.3,
    random_state=0
)

X_train['A4'].unique()

dummies = pd.get_dummies(X_train['A4'], drop_first=True)
dummies.head()

X_train_enc = pd.get_dummies(X_train, drop_first=True)
X_test_enc = pd.get_dummies(X_test, drop_first=True)
X_train_enc.head()

X_train_enc.columns

X_test_enc = pd.concat([X_test, X_test_enc], axis='columns')

X_test_enc.drop(
    labels=X_test_enc.select_dtypes(include='O').columns,
    axis='columns',
    inplace=True
)
```

```
# Let' s do one-hot encoding using scikit-learn
from sklearn.preprocessing import OneHotEncoder

```

```python
X_train, X_test, y_train, y_test = train_test_split(
    data.drop(labels=['target'], axis='columns'), data['target'],
    test_size=.3,
    random_state=0
)

# handle_unknown='ignore'
encoder = OneHotEncoder(drop='first', sparse=False)

vars_categorical = X_train.select_dtypes(include='O').columns.to_list()

encoder.fit(X_train[vars_categorical])

encoder.categories_

X_train_enc = encoder.transform(
    X_train[vars_categorical]
)

X_test_enc = encoder.transform(
    X_test[vars_categorical]
)

encoder.get_feature_names_out()

X_test_enc = pd.DataFrame(X_test_enc)
X_test_enc.columns = encoder.get_feature_names_out()

X_test_enc.index = X_test.index

X_test_enc = pd.concat([X_test, X_test_enc], axis='columns')

X_test_enc.drop(
    labels=X_test_enc.select_dtypes(include='O').columns,
    axis='columns',
    inplace=True
)
```

```
# Let's perform one-hot encoding with Feature-engine
from feature_engine.encoding import OneHotEncoder

X_train, X_test, y_train, y_test = train_test_split(
    data.drop(labels=['target'], axis='columns'), data['target'],
    test_size=.3,
    random_state=0
)

ohe_enc = OneHotEncoder(drop_last=True)
ohe_enc.fit(X_train)

ohe_enc.variables_

ohe_enc.encoder_dict_

X_train_enc = ohe_enc.transform(X_train)
X_test_enc = ohe_enc.transform(X_test)

ohe_enc.get_feature_names_out()
```

2.2.2 Performing one-hot encoding of frequent categories

One-hot encoding represents each variable’s category with a binary variable. Hence, one-hot encoding of highly cardinal variables or datasets with multiple categorical features can expand the feature space dramatically. This, in turn, may increase the computational cost of using machine learning models or deteriorate their performance. To reduce the number of binary variables, we can perform one-hot encoding of the most frequent categories. One-hot encoding the top categories is equivalent to treating the remaining, less frequent categories as a single, unique category.